Data Modeling

Windows Enterprise Support Database Services provides the following documentation about relational database design, the relational database model, and relational database software.

Introduction to Data Modeling

A brief overview of developing a conceptual data model as the first step in creating a relational database.

Overview of the Relational Model

Discusses data structures, relational operators, and normalization.

Introduction to Data Modeling

This document is an informal introduction to data modeling using the Entity-Relationship (ER) approach. It is intended for someone who is familiar with relational databases but who has no experience in data modeling. The basic techniques described are applicable to the development of microcomputer based relational database applications as well as those who use relational database servers such as MS SQL Server or Oracle.

The document is a practical guide, not an academic paper on either relational database design or data modeling. Readers interested in a rigorous treatment of these topics should consult the **bibliography**.

Topics

What is A Data Model

A data model is a conceptual representation of the data structures that are required by a database. The data structures include the data objects, the associations between data objects, and the rules which govern operations on the objects. As the name implies, the data model focuses on what data is required and how it should be organized rather than what operations will be performed on the data. To use a common analogy, the data model is equivalent to an architect's building plans.

A data model is independent of hardware or software constraints. Rather than try to represent the data as a database would see it, the data model focuses on representing the data as the user sees it in the "real world". It serves as a bridge between the concepts that make up real-world events and processes and the physical representation of those concepts in a database.

Methodology

There are two major methodologies used to create a data model: the Entity-Relationship (ER) approach and the Object Model. This document uses the Entity-Relationship approach.

Data Modeling In the Context of Database Design

Database design is defined as: "design the logical and physical structure of one or more databases to accommodate the information needs of the users in an organization for a defined set of applications". The design process roughly follows five steps:

1. planning and analysis
2. conceptual design
3. logical design
4. physical design
5. implementation

The data model is one part of the conceptual design process. The other, typically is the functional model. The data model focuses on what data should be stored in the database while the functional model deals with how the data is processed. To put this in the context of the relational database, the data model is used to design the relational tables. The functional model is used to design the queries which will access and perform operations on those tables.

Components of A Data Model

The data model gets its inputs from the planning and analysis stage. Here the modeler, along with analysts, collects information about the requirements of the database by reviewing existing documentation and interviewing end-users.

The data model has two outputs. The first is an entity-relationship diagram which represents the data strucures in a pictorial form. Because the diagram is easily learned, it is valuable tool to communicate the model to the end-user. The second component is a data document. This a document that describes in detail the data objects, relationships, and rules required by the database. The dictionary provides the detail required by the database developer to construct the physical database.

Why is Data Modeling Important?

Data modeling is probably the most labor intensive and time consuming part of the development process. Why bother especially if you are pressed for time? A common response by practitioners who write on the subject is that you should no more build a database without a model than you should build a house without blueprints.

The goal of the data model is to make sure that the all data objects required by the database are completely and accurately represented. Because the data model uses easily understood notations and natural language , it can be reviewed and verified as correct by the end-users.

The data model is also detailed enough to be used by the database developers to use as a "blueprint" for building the physical database. The information contained in the data model will be used to define the relational tables, primary and foreign keys, stored procedures, and triggers. A poorly designed database will require more time in the long-term. Without careful planning you may create a database that omits data required to create critical reports, produces results that are incorrect or inconsistent, and is unable to accommodate changes in the user's requirements.

Summary

A data model is a plan for building a database. To be effective, it must be simple enough to communicate to the end user the data structure required by the database yet detailed enough for the database design to use to create the physical structure.

The Entity-Relation Model (ER) is the most common method used to build data models for relational databases. The next section provides a brief introduction to the concepts used by the ER Model.

The Entity-Relationship Model

The Entity-Relationship (ER) model was originally proposed by Peter in 1976 [Chen76] as a way to unify the network and relational database views. Simply stated the ER model is a conceptual data model that views the real world as entities and relationships. A basic component of the model is the Entity-Relationship diagram which is used to visually represents data objects. Since Chen wrote his paper the model has been extended and today it is commonly used for database design For the database designer, the utility of the ER model is:

- it maps well to the relational model. The constructs used in the ER model can easily be transformed into relational tables.
- it is simple and easy to understand with a minimum of training. Therefore, the model can be used by the database designer to communicate the design to the end user.

- In addition, the model can be used as a design plan by the database developer to implement a data model in a specific database management software.

Basic Constructs of E-R Modeling

The ER model views the real world as a construct of entities and association between entities.

Entities

Entities are the principal data object about which information is to be collected. Entities are usually recognizable concepts, either concrete or abstract, such as person, places, things, or events which have relevance to the database. Some specific examples of entities are EMPLOYEES, PROJECTS, INVOICES. An entity is analogous to a table in the relational model.

Entities are classified as independent or dependent (in some methodologies, the terms used are strong and weak, respectively). An independent entity is one that does not rely on another for identification. A dependent entity is one that relies on another for identification.

An entity occurrence (also called an instance) is an individual occurrence of an entity. An occurrence is analogous to a row in the relational table.

Special Entity Types

Associative entities (also known as intersection entities) are entities used to associate two or more entities in order to reconcile a many-to-many relationship.

Subtypes entities are used in generalization hierarchies to represent a subset of instances of their parent entity, called the supertype, but which have attributes or relationships that apply only to the subset.

Associative entities and generalization hierarchies are discussed in more detail below.

Relationships

A Relationship represents an association between two or more entities. An example of a relationship would be:

employees are assigned to projects
projects have subtasks
departments manage one or more projects

Relationships are classified in terms of degree, connectivity, cardinality, and existence. These concepts will be discussed below.

Attributes

Attributes describe the entity of which they are associated. A particular instance of an attribute is a value. For example, "Jane R. Hathaway" is one value of the attribute Name. The domainof an attribute is the collection of all possible values an attribute can have. The domain of Name is a character string.

Attributes can be classified as identifiers or descriptors. Identifiers, more commonly called keys, uniquely identify an instance of an entity. A descriptor describes a non-unique characteristic of an entity instance.

Classifying Relationships

Relationships are classified by their degree, connectivity, cardinality, direction, type, and existence. Not all modeling methodologies use all these classifications.

Degree of a Relationship

The degree of a relationship is the number of entities associated with the relationship. The n-ary relationship is the general form for degree n. Special cases are the binary, and ternary ,where the degree is 2, and 3, respectively.

Binary relationships, the association between two entities is the most common type in the real world. A recursive binary relationship occurs when an entity is related to itself. An example might be "some employees are married to other employees".

A ternary relationship involves three entities and is used when a binary relationship is inadequate. Many modeling approaches recognize only binary relationships. Ternary or n-ary relationships are decomposed into two or more binary relationships.

Connectivity and Cardinality

The connectivity of a relationship describes the mapping of associated entity instances in the relationship. The values of connectivity are "one" or "many". The cardinality of a relationship is the actual number of related occurences for each of the two entities. The basic types of connectivity for relations are: one-to-one, one-to-many, and many-to-many.

A one-to-one (1:1) relationship is when at most one instance of a entity A is associated with one instance of entity B. For example, "employees in the company are each assigned their own office. For each employee there exists a unique office and for each office there exists a unique employee.

A one-to-many (1:N) relationships is when for one instance of entity A, there are zero, one, or many instances of entity B, but for one instance of entity B, there is only one instance of entity A. An example of a 1:N relationships is

> a department has many employees
> each employee is assigned to one department

A many-to-many (M:N) relationship, sometimes called non-specific, is when for one instance of entity A, there are zero, one, or many instances of entity B and for one instance of entity B there are zero, one, or many instances of entity A. An example is:

> employees can be assigned to no more than two projects at the same time;
> projects must have assigned at least three employees

A single employee can be assigned to many projects; conversely, a single project can have assigned to it many employee. Here the cardinality for the relationship between employees and projects is two and the cardinality between project and employee is three. Many-to-many relationships cannot be directly translated to relational tables but instead must be transformed into two or more one-to-many relationships using associative entities.

Direction

The direction of a relationship indicates the originating entity of a binary relationship. The entity from which a relationship originates is the parent entity; the entity where the relationship terminates is the child entity.

The direction of a relationship is determined by its connectivity. In a one-to-one relationship the direction is from the independent entity to a dependent entity. If both entities are independent, the direction is arbitrary. With one-to-many relationships, the entity occurring once is the parent. The direction of many-to-many relationships is arbitrary.

Type

An identifying relationship is one in which one of the child entities is also a dependent entity. A non-identifying relationship is one in which both entities are independent.

Existence

Existence denotes whether the existence of an entity instance is dependent upon the existence of another, related, entity instance. The existence of an entity in a relationship is defined as either mandatory or optional. If an instance of an entity must always occur for an entity to be included in a relationship, then it is mandatory. An example of mandatory existence is the statement "every project must be managed by a single department". If the instance of the entity is not required, it is optional. An example of optional existence is the statement, "employees may be assigned to work on projects".

Generalization Hierarchies

A generalization hierarchy is a form of abstraction that specifies that two or more entities that share common attributes can be generalized into a higher level entity type called a supertype or generic entity. The lower-level of entities become the subtype, or categories, to the supertype. Subtypes are dependent entities.

Generalization occurs when two or more entities represent categories of the same real-world object. For example, Wages_Employees and Classified_Employees represent categories of the same entity, Employees. In this example, Employees would be the supertype; Wages_Employees and Classified_Employees would be the subtypes.

Subtypes can be either mutually exclusive (disjoint) or overlapping (inclusive). A mutually exclusive category is when an entity instance can be in only one category. The above example is a mutually exclusive category. An employee can either be wages or classified but not both. An overlapping category is when an entity instance may be in two or more subtypes. An example would be a person who works for a university could also be a student at that same university. The completeness constraint requires that all instances of the subtype be represented in the supertype.

Generalization hierarchies can be nested. That is, a subtype of one hierarchy can be a supertype of another. The level of nesting is limited only by the constraint of simplicity. Subtype entities may be the parent entity in a relationship but not the child.

ER Notation

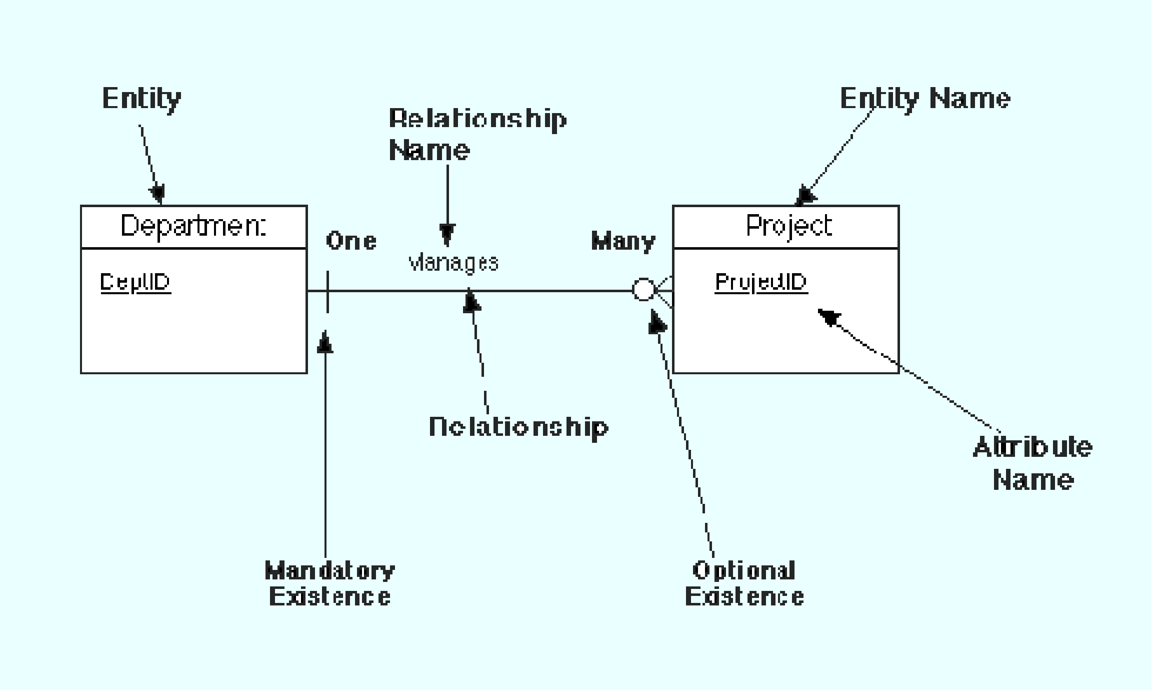
There is no standard for representing data objects in ER diagrams. Each modeling methodology uses its own notation. The original notation used by Chen is widely used in academics texts and journals but rarely seen in either CASE tools or publications by non-academics. Today, there are a number of notations used, among the more common are Bachman, crow's foot, and IDEFIX.

All notational styles represent entities as rectangular boxes and relationships as lines connecting boxes. Each style uses a special set of symbols to represent the cardinality of a connection. The notation used in this document is from Martin. The symbols used for the basic ER constructs are:

* entities are represented by labeled rectangles. The label is the name of the entity. Entity names should be singular nouns.
* relationships are represented by a solid line connecting two entities. The name of the relationship is written above the line. Relationship names should be verbs.
* attributes, when included, are listed inside the entity rectangle. Attributes which are identifiers are underlined. Attribute names should be singular nouns.
* cardinality of many is represented by a line ending in a crow's foot. If the crow's foot is omitted, the cardinality is one.
* existence is represented by placing a circle or a perpendicular bar on the line. Mandatory existence is shown by the bar (looks like a 1) next to the entity for an instance is required. Optional existence is shown by placing a circle next to the entity that is optional.

Examples of these symbols are shown in Figure 1 below:

Figure 1: ER Notation



Summary

The Entity-Relationship Model is a conceptual data model that views the real world as consisting of entities and relationships. The model visually represents these concepts by the Entity-Relationship diagram. The basic constructs of the ER model are entities, relationships, and attributes. Entities are concepts, real or abstract, about which information is collected. Relationships are associations between the entities. Attributes are properties which describe the entities. Next, we will look at the role of data modeling in the overall database design process and a method for building the data model. To proceed, see Data Modeling As Part of Database Design.

Data Modeling As Part of Database Design

The data model is one part of the conceptual design process. The other is the function model. The data model focuses on what data should be stored in the database while the function model deals with how the data is processed. To put this in the context of the relational database, the data model is used to design the relational tables. The functional model is used to design the queries that will access and perform operations on those tables.

Data modeling is preceeded by planning and analysis. The effort devoted to this stage is proportional to the scope of the database. The planning and analysis of a database intended to serve the needs of an enterprise will require more effort than one intended to serve a small workgroup.

The information needed to build a data model is gathered during the requirments analysis. Although not formally considered part of the data modeling stage by some methodologies, in reality the requirements analysis and the ER diagramming part of the data model are done at the same time.

Requirements Analysis

The goals of the requirements analysis are:

- to determine the data requirements of the database in terms of primitive objects
- to classify and describe the information about these objects
- to identify and classify the relationships among the objects
- to determine the types of transactions that will be executed on the database and the interactions between the data and the transactions
- to identify rules governing the integrity of the data

The modeler, or modelers, works with the end users of an organization to determine the data requirements of the database. Information needed for the requirements analysis can be gathered in several ways:

- review of existing documents - such documents include existing forms and reports, written guidelines, job descriptions, personal narratives, and memoranda. Paper documentation is a good way to become familiar with the organization or activity you need to model.
- interviews with end users - these can be a combination of individual or group meetings. Try to keep group sessions to under five or six people. If possible, try to have everyone with the same function in one meeting. Use a blackboard, flip charts, or overhead transparencies to record information gathered from the interviews.
- review of existing automated systems - if the organization already has an automated system, review the system design specifications and documentation

The requirements analysis is usually done at the same time as the data modeling. As information is collected, data objects are identified and classified as either entities, attributes, or relationship; assigned names; and, defined using terms familiar to the end-users. The objects are then modeled and analysed using an ER diagram. The diagram can be reviewed by the modeler and the end-users to determine its completeness and accuracy. If the model is not correct, it is modified, which sometimes requires additional information to be collected. The review and edit cycle continues until the model is certified as correct.

Three points to keep in mind during the requirements analysis are:

1. Talk to the end users about their data in "real-world" terms. Users do not think in terms of entities, attributes, and relationships but about the actual people, things, and activities they deal with daily.

2. Take the time to learn the basics about the organization and its activities that you want to model. Having an understanding about the processes will make it easier to build the model.
3. End-users typically think about and view data in different ways according to their function within an organization. Therefore, it is important to interview the largest number of people that time permits.

Steps In Building the Data Model

While ER model lists and defines the constructs required to build a data model, there is no standard process for doing so. Some methodologies, such as IDEFIX, specify a bottom-up development process were the model is built in stages. Typically, the entities and relationships are modeled first, followed by key attributes, and then the model is finished by adding non-key attributes. Other experts argue that in practice, using a phased approach is impractical because it requires too many meetings with the end-users. The sequence used for this document are:

1. Identification of data objects and relationships
2. Drafting the initial ER diagram with entities and relationships
3. Refining the ER diagram
4. Add key attributes to the diagram
5. Adding non-key attributes
6. Diagramming Generalization Hierarchies
7. Validating the model through normalization
8. Adding business and integrity rules to the Model

In practice, model building is not a strict linear process. As noted above, the requirements analysis and the draft of the initial ER diagram often occur simultaneously. Refining and validating the diagram may uncover problems or missing information which require more information gathering and analysis

Summary

Data modeling must be preceded by planning and analysis. Planning defines the goals of the database , explains why the goals are important, and sets out the path by which the goals will be reached. Analysis involves determining the requirements of the database. This is typically done by examining existing documentation and interviewing users.

An effective data model completely and accurately represents the data requirements of the end users. It is simple enough to be understood by the end user yet detailed enough to be used by a database designer to build the database. The model eliminates redundant data, it is independent of any hardware and software constraints, and can be adapted to changing requirements with a minimum of effort.

Data modeling is a bottom up process. A basic model, representing entities and relationships, is developed first. Then detail is added to the model by including information about attributes and business rules.

The next section discusses Identifying Data Objects and Relationships.

Identifying Data Objects and Relationships

In order to begin constructing the basic model, the modeler must analyze the information gathered during the requirements analysis for the purpose of:

- classifying data objects as either entities or attributes
- identifying and defining relationships between entities
- naming and defining identified entities, attributes, and relationships
- documenting this information in the data document

To accomplish these goals the modeler must analyze narratives from users, notes from meeting, policy and procedure documents, and, if lucky, design documents from the current information system.

Although it is easy to define the basic constructs of the ER model, it is not an easy task to distinguish their roles in building the data model. What makes an object an entity or attribute? For example, given the statement "employees work on projects". Should employees be classified as an entity or attribute? Very often, the correct answer depends upon the requirements of the database. In some cases, employee would be an entity, in some it would be an attribute.

While the definitions of the constructs in the ER Model are simple, the model does not address the fundamental issue of how to identify them. Some commonly given guidelines are:

- entities contain descriptive information
- attributes either identify or describe entities
- relationships are associations between entities

These guidelines are discussed in more detail below.

- Entities
- Attributes
  - Validating Attributes
  - Derived Attributes and Code Values
- Relationships
- Naming Data Objects
- Object Definition
- Recording Information in Design Document

Entities

There are various definitions of an entity:

> "Any distinguishable person, place, thing, event, or concept, about which information is kept" [BRUC92]
>
> "A thing which can be distinctly identified" [CHEN76]
>
> "Any distinguishable object that is to be represented in a database" [DATE86]
>
> "...anything about which we store information (e.g. supplier, machine tool, employee, utility pole, airline seat, etc.). For each entity type, certain attributes are stored". [MART89]

These definitions contain common themes about entities:

- an entity is a "thing", "concept" or, object". However, entities can sometimes represent the relationships between two or more objects. This type of entity is known as an associative entity.
- entities are objects which contain descriptive information. If an data object you have identified is described by other objects, then it is an entity. If there is no descriptive information associated with the item, it is not an entity. Whether or not a data object is an entity may depend upon the organization or activity being modeled.
- an entity represents many things which share properties. They are not single things. For example, King Lear and Hamlet are both plays which share common attributes such as name, author, and cast of characters. The entity describing these things would be PLAY, with King Lear and Hamlet being instances of the entity.
- entities which share common properties are candidates for being converted to generalization hierarchies (See below)
- entities should not be used to distinguish between time periods. For example, the entities 1st Quarter Profits, 2nd Quarter Profits, etc. should be collapsed into a single entity called Profits. An attribute specifying the time period would be used to categorize by time
- not every thing the users want to collect information about will be an entity. A complex concept may require more than one entity to represent it. Others "things" users think important may not be entities.

Attributes

Attributes are data objects that either identify or describe entities. Attributes that identify entities are called *key attributes*. Attributes that describe an entity are called non-key attributes. Key attributes will be discussed in detail in a latter section.

The process for identifying attributes is similar except now you want to look for and extract those names that appear to be descriptive noun phrases.

Validating Attributes

Attribute values should be *atomic*, that is, present a single fact. Having disaggregated data allows simpler programming, greater reusability of data, and easier implementation of changes. Normalization also depends upon the "single fact" rule being followed. Common types of violations include:

- simple aggregation - a common example is Person Name which concatenates first name, middle initial, and last name. Another is Address which concatenates, street address, city, and zip code. When dealing with such attributes, you need to find out if there are good reasons for decomposing them. For example, do the end-users want to use the person's first name in a form letter? Do they want to sort by zip code?
- complex codes - these are attributes whose values are codes composed of concatenated pieces of information. An example is the code attached to automobiles and trucks. The code represents over 10 different pieces of information about the vehicle. Unless part of an industry standard, these codes have no meaning to the end user. They are very difficult to process and update.
- text blocks - these are free-form text fields. While they have a legitimate use, an over reliance on them may indicate that some data requirements are not met by the model.
- mixed domains - this is where a value of an attribute can have different meaning under different conditions

Derived Attributes and Code Values

Two areas where data modeling experts disagree is whether derived attributes and attributes whose values are codes should be permitted in the data model.

Derived attributes are those created by a formula or by a summary operation on other attributes. Arguments against including derived data are based on the premise that derived data should not be stored in a database and therefore should not be included in the data model. The arguments in favor are:

- derived data is often important to both managers and users and therefore should be included in the data model
- it is just as important, perhaps more so, to document derived attributes just as you would other attributes
- including derived attributes in the data model does not imply how they will be implemented

A coded value uses one or more letters or numbers to represent a fact. For example, the value Gender might use the letters "M" and "F" as values rather than "Male" and "Female". Those who are against this practice cite that codes have no intuitive meaning to the end-users and add complexity to processing data. Those in favor argue that many organizations have a long history of using coded attributes, that codes save space, and improve flexibility in that values can be easily added or modified by means of look-up tables.

Relationships

Relationships are associations between entities. Typically, a relationship is indicated by a verb connecting two or more entities. For example:

> employees **are assigned** to projects

As relationships are identified they should be classified in terms of cardinality, optionality, direction, and dependence. As a result of defining the relationships, some relationships may be dropped and new relationships added. Cardinality quantifies the relationships between entities by measuring how many instances of one entity are related to a single instance of another. To determine the cardinality, assume the existence of an instance of one of the entities. Then determine how many specific instances of the second entity could be related to the first. Repeat this analysis reversing the entities. For example:

> employees **may be assigned** to no more than three projects at a time; every project has at least two employees assigned to it.

Here the cardinality of the relationship from employees to projects is three; from projects to employees, the cardinality is two. Therefore, this relationship can be classified as a many-to-many relationship.

If a relationship can have a cardinality of zero, it is an optional relationship. If it must have a cardinality of at least one, the relationship is mandatory. Optional relationships are typically indicated by the conditional tense. For example:

> an employee **may** be assigned to a project

Mandatory relationships, on the other hand, are indicated by words such as must have. For example:

> a student **must** register for at least three course each semester

In the case of the specific relationship form (1:1 and 1:M), there is always a parent entity and a child entity. In one-to-many relationships, the parent is always the entity with the cardinality of one. In one-to-many relationships, the choice of the parent entity must be made in the context of the business being modeled. If a decision cannot be made, the choice is arbitrary.

Naming Data Objects

The names should have the following properties:

* unique
* have meaning to the end-user
* contain the minimum number of words needed to uniquely and accurately describe the object

For entities and attributes, names are singular nouns while relationship names are typically verbs.

Some authors advise against using abbreviations or acronyms because they might lead to confusion about what they mean. Other believe using abbreviations or acronyms are acceptable provided that they are universally used and understood within the organization.

You should also take care to identify and resolve synonyms for entities and attributes. This can happen in large projects where different departments use different terms for the same thing.

Object Definition

Complete and accurate definitions are important to make sure that all parties involved in the modeling of the data know exactly what concepts the objects are representing.

Definitions should use terms familiar to the user and should precisely explain what the object represents and the role it plays in the enterprise. Some authors recommend having the end-users provide the definitions. If acronyms, or terms not universally understood are used in the definition, then these should be defined .

While defining objects, the modeler should be careful to resolve any instances where a single entity is actually representing two different concepts (homonyms) or where two different entities are actually representing the same "thing" (synonyms). This situation typically arises because individuals or organizations may think about an event or process in terms of their own function.

An example of a homonym would be a case where the Marketing Department defines the entity MARKET in terms of geographical regions while the Sales Departments thinks of this entity in terms of demographics. Unless resolved, the result would be an entity with two different meanings and properties.

Conversely, an example of a synonym would be the Service Department may have identified an entity called CUSTOMER while the Help Desk has identified the entity CONTACT. In reality, they may mean the same thing, a person who contacts or calls the organization for assistance with a problem. The resolution of synonyms is important in order to avoid redundancy and to avoid possible consistency or integrity problems.

Some examples of definitions are:

Employee	A person who works for and is paid by the organization.

Est_Time The number of hours a project manager estimates that project will require to complete. Estimated time is critical for scheduling a project and for tracking project time variances.

Assigned Employees in the organization may be assigned to work on no more than three projects at a time. Every project will have at least two employees assigned to it at any given time.

Recording Information in Design Document

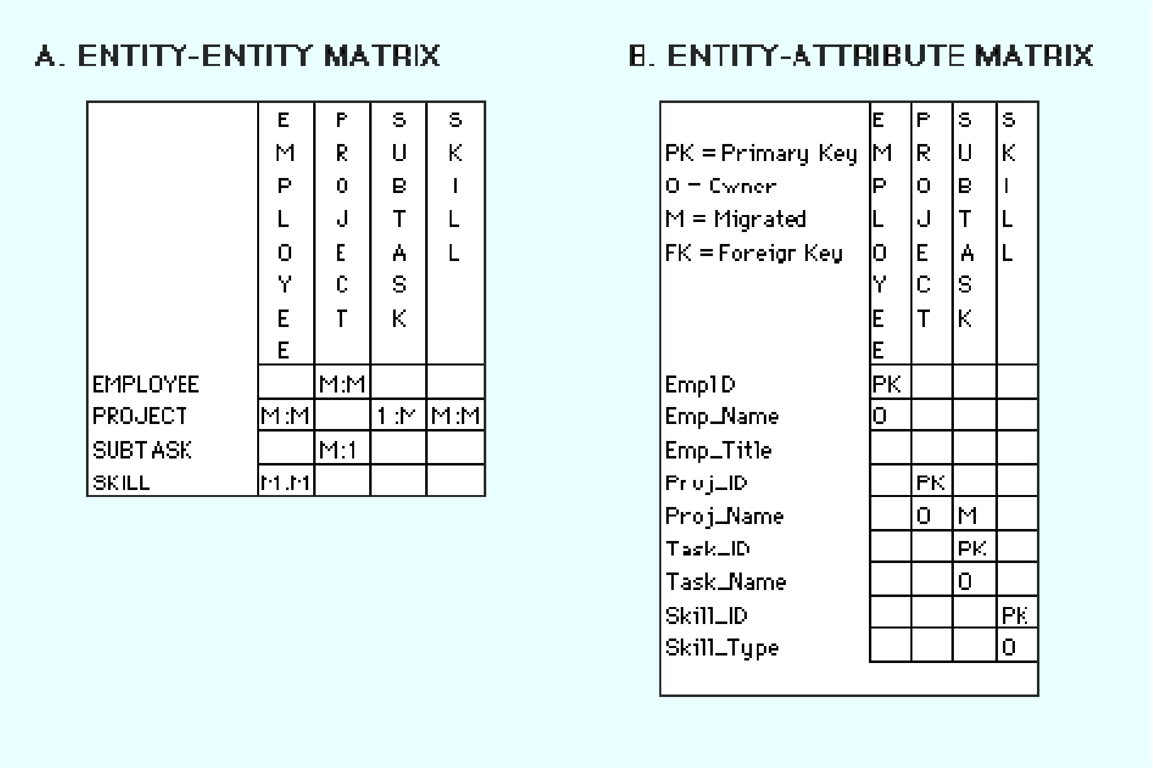
The design document records detailed information about each object used in the model. As you name, define, and describe objects, this information should be placed in this document. If you are not using an automated design tool, the document can be done on paper or with a word processor. There is no standard for the organization of this document but the document should include information about names, definitions, and, for attributes, domains.

Two documents used in the IDEF1X method of modeling are useful for keeping track of objects. These are the ENTITY-ENTITY matrix and the ENTITY-ATTRIBUTE matrix.

The ENTITY-ENTITY matrix is a two-dimensional array for indicating relationships between entities. The names of all identified entities are listed along both axes. As relationships are first identified, an "X" is placed in the intersecting points where any of the two axes meet to indicate a possible relationship between the entities involved. As the relationship is further classified, the "X" is replaced with the notation indicating cardinality.

The ENTITY-ATTRIBUTE matrix is used to indicate the assignment of attributes to entities. It is similar in form to the ENTITY-ENTITY matrix except attribute names are listed on the rows.

Figure 1 shows examples of an ENTITY-ENTITY matrix and an ENTITY-ATTRIBUTE matrix.
Figure 1:



A. ENTITY-ENTITY MATRIX

	EMPLOYEE	PROJECT	SUBTASK	SKILL
EMPLOYEE		M:M		
PROJECT	M:M		1:M	M:M
SUBTASK		M:1		
SKILL	M.M			

B. ENTITY-ATTRIBUTE MATRIX

PK = Primary Key
O – Cwner
M = Migrated
FK = Foreigr Key
	EMPLOYEE	PROJECT	SUBTASK	SKILL

Emp1D	PK			
Emp_Name	O			
Emp_Title				
Pr vj_ID		PK		
Proj_Name		O	M	
Task_ID			PK	
Task_Name			O	
Skill_ID				PK
Skill_Type				O

Summary

The first step in creating the data model is to analyze the information gathered during the requirements analysis with the goal of identifying and classifying data objects and relationships. The next step is **Developing the Basic Schema**.

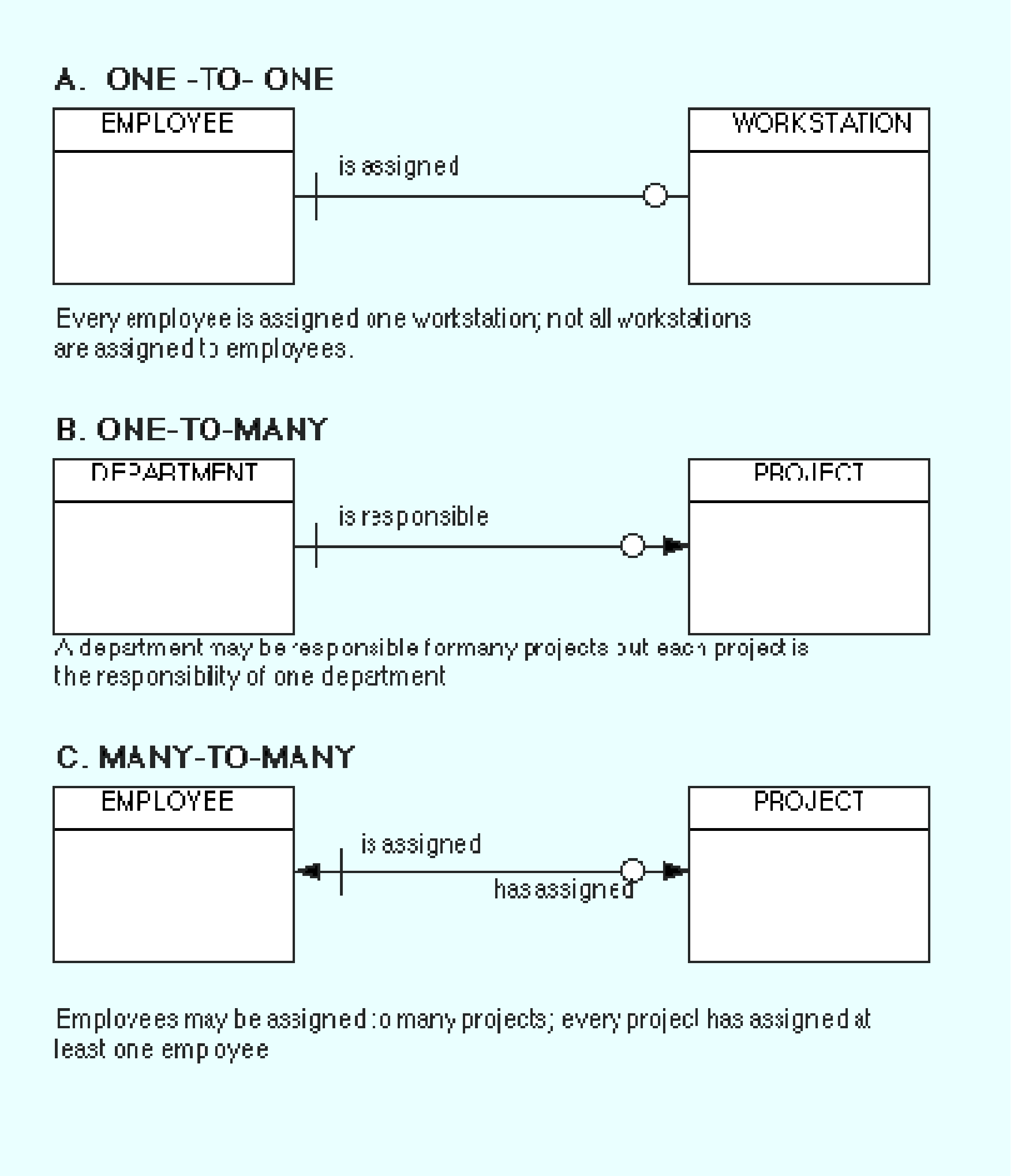
Developing the Basic Schema

Once entities and relationships have been identified and defined, the first draft of the entity relationship diagram can be created. This section introduces the ER diagram by demonstrating how to diagram binary relationships. Recursive relationships are also shown.

Binary Relationships

Figure 1 shows examples of how to diagram one-to-one, one-to-many, and many-to-many relationships.

Figure 1: Example of Binary Relationships


A. ONE -TO- ONE

Every employee is assigned one workstation; not all workstations are assigned to employees.

B. ONE-TO-MANY

A department may be responsible for many projects but each project is the responsibility of one department

C. MANY-TO-MANY

Employees may be assigned to many projects; every project has assigned at least one employee

One-To-One

Figure 1A shows an example of a one-to-one diagram. Reading the diagram from represents the relationship every employee is assigned a workstation. Because every employee must have a workstation, the symbol for mandatory existence—in this case the crossbar—is placed next to the EMPLOYEE entity. Reading from right to left, the diagram shows that not all workstation are assigned to employees. This condition may reflect that some workstations are kept for spares or for loans. Therefore, we use the symbol for optional existence, the circle, next to workstation. The cardinality and existence of a relationship must be derived from the "business rules" of the organization. For example, if all workstations owned by an organization were assigned to employees, then the circle would be replaced by a crossbar to indicate mandatory existence. One-to-one relationships are rarely seen in "real-world" data models. Some practioners advise that most one-to-one relationships should be collapsed into a single entity or converted to a generalization hierarchy.

One-To-Many

Figure 1B shows an example of a one-to-many relationship between DEPARTMENT and PROJECT. In this diagram, DEPARTMENT is considered the parent entity while PROJECT is the child. Reading from left to right, the diagram represents departments may be responsible for many projects. The optionality of the relationship reflects the "business rule" that not all departments in the organization will be responsible for managing projects. Reading from right to left, the diagram tells us that every project must be the responsibility of exactly one department.

Many-To-Many

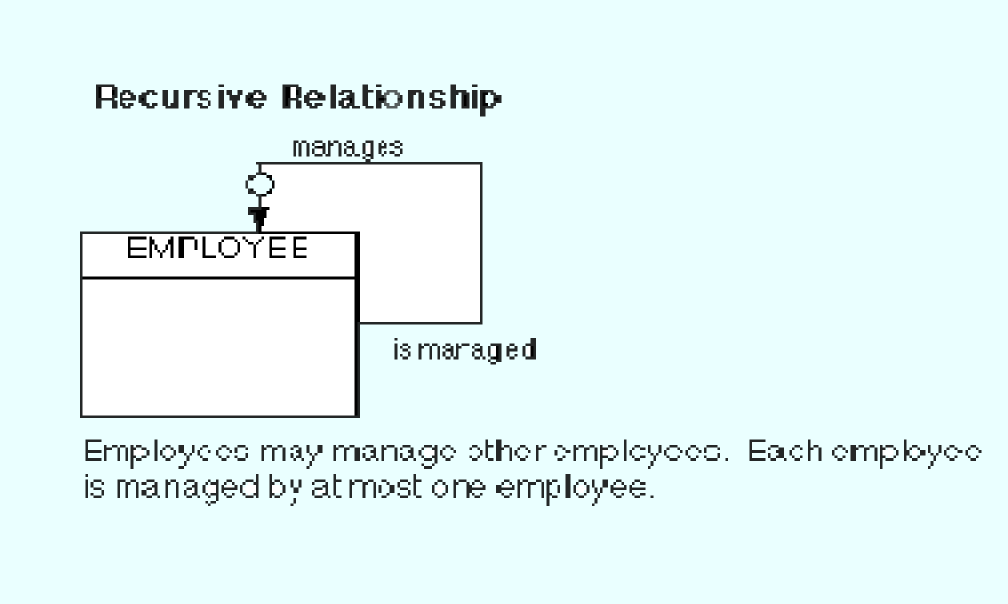
Figure 1C shows a many-to-many relationship between EMPLOYEE and PROJECT. An employee **may** be assigned to many projects; each project **must** have many employee Note that the association between EMPLOYEE and PROJECT is optional because, at a given time, an employee may not be assigned to a project. However, the relationship between PROJECT and EMPLOYEE is mandatory because a project must have at least two employees assigned. Many-To-Many relationships can be used in the initial drafting of the model but eventually must be transformed into two one-to-many relationships. The transformation is required because many-to-many relationships cannot be represented by the relational model. The process for resolving many-to-many relationships is discussed in the next section.

Recursive relationships

A recursive relationship is an entity is associated with itself. Figure 2 shows an example of the recursive relationship.

An employee may manage many employees and each employee is managed by one employee.

Figure 2: Example of Recursive Relationship



Employees may manage other employees. Each employee is managed by at most one employee.

Summary

The Entity-Relationship diagram provides a pictorial representation of the major data objects, the entities, and the relationships between them. Once the basic diagram is completed, the next step is **Refining The Entity-Relationship Diagram**

Refining The Entity-Relationship Diagram

This section discusses four basic rules for modeling relationships

Entities Must Participate In Relationships

Entities cannot be modeled unrelated to any other entity. Otherwise, when the model was transformed to the relational model, there would be no way to navigate to that table. The exception to this rule is a database with a single table.

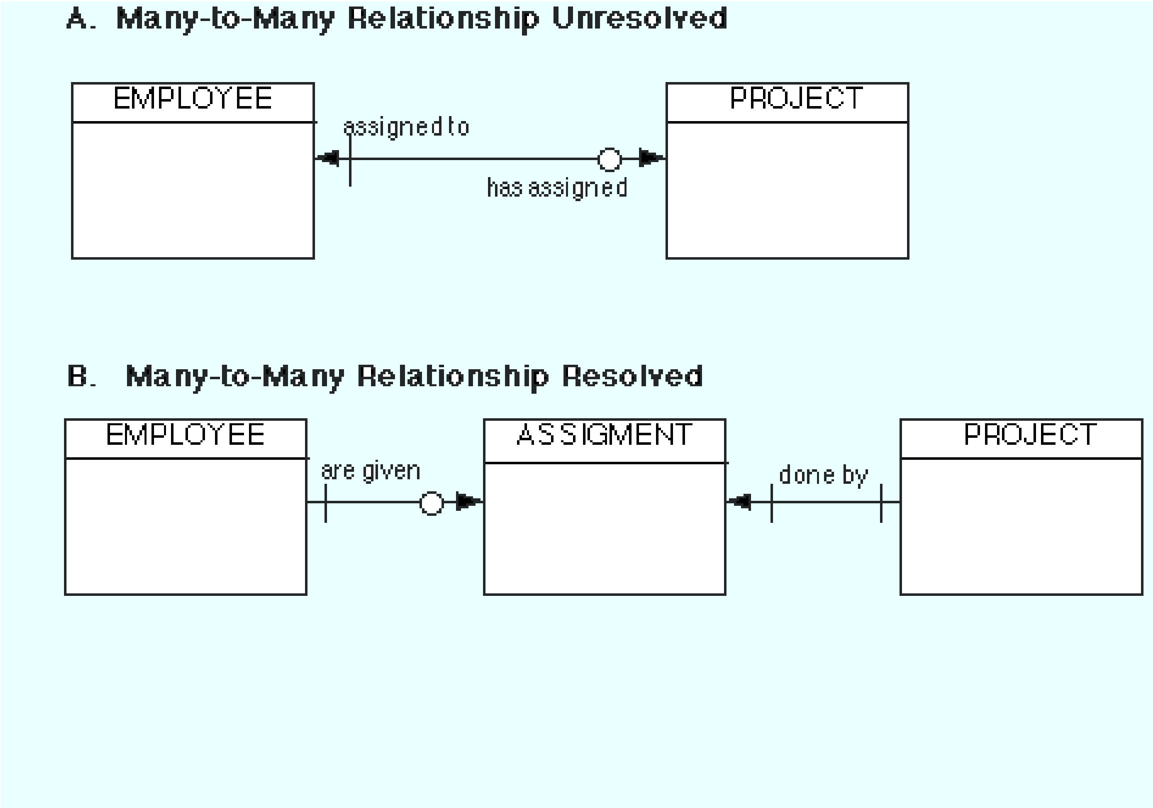
Resolve Many-To-Many Relationships

Many-to-many relationships cannot be used in the data model because they cannot be represented by the relational model. Therefore, many-to-many relationships must be resolved early in the modeling process. The strategy for resolving many-to-many relationship is to replace the relationship with an *association entity* and then relate the two original entities to the association entity. This strategy is demonstrated below Figure 6.1 (a) shows the many-to-many relationship:

> Employees may be **assigned to** many projects.
> Each project must have **assigned to** it more than one employee.

In addition to the implementation problem, this relationship presents other problems. Suppose we wanted to record information about employee assignments such as who assigned them, the start date of the assignment, and the finish date for the assignment. Given the present relationship, these attributes could not be represented in either EMPLOYEE or PROJECT without repeating information. The first step is to convert the relationship **assigned to** to a new entity we will call ASSIGNMENT. Then the original entities, EMPLOYEE and PROJECT, are related to this new entity preserving the cardinality and optionality of the original relationships. The solution is shown in Figure 1B.

Figure 1: Resolution of a Many-To-Many Relationship



Notice that the schema changes the semantics of the original relation to

> employees **may be given** assignments to projects
> and projects must **be done by** more than one employee assignment.

A many to many recursive relationship is resolved in similar fashion.

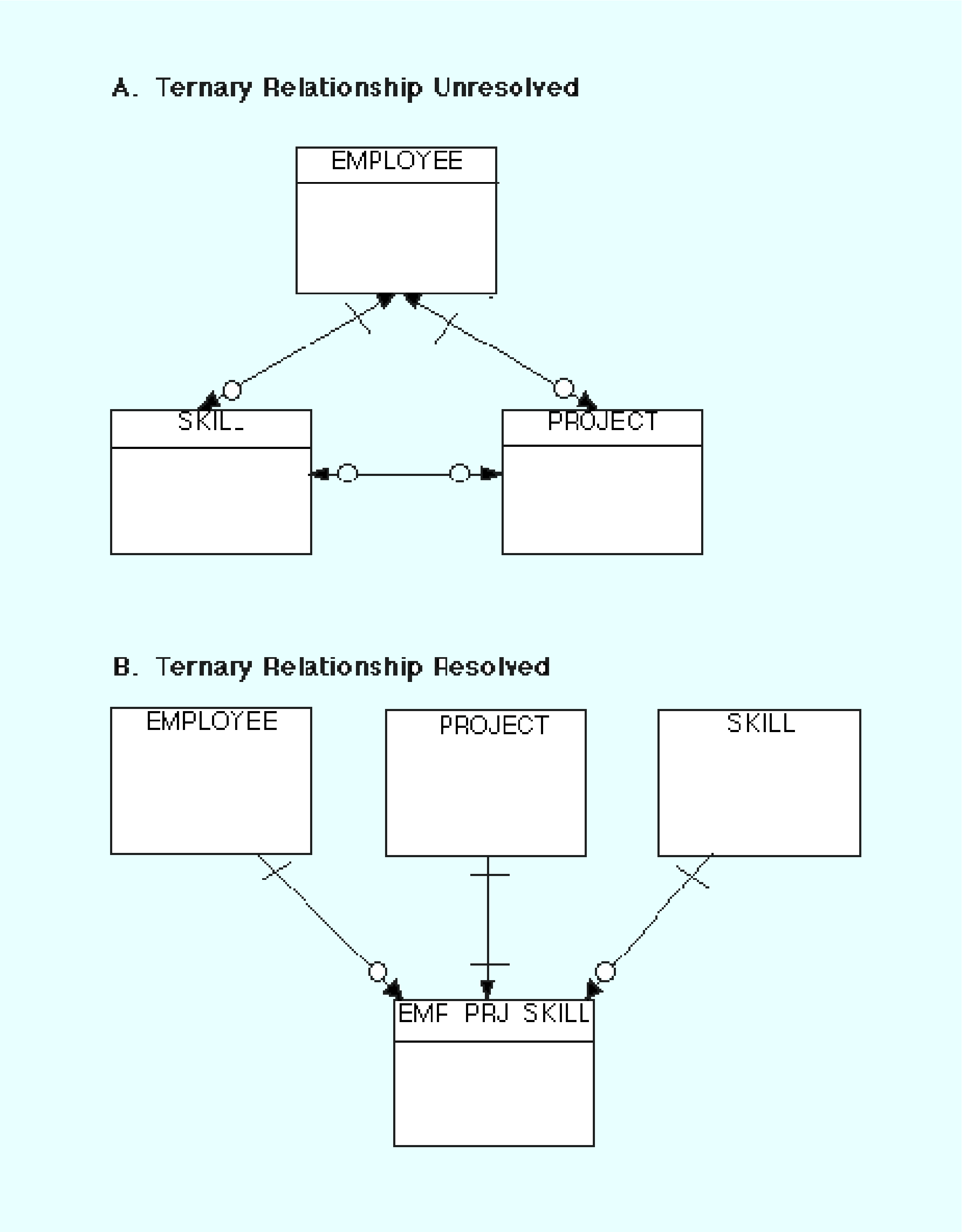
Transform Complex Relationships into Binary Relationships

Complex relationships are classified as ternary, an association among three entities, or n-ary, an association among more than three, where n is the number of entities involved. For example, Figure 2A shows the relationship

> Employees can use different skills on any one or more projects.
> Each project uses many employees with various skills.

Complex relationships cannot be directly implemented in the relational model so they should be resolved early in the modeling process. The strategy for resolving complex relationships is similar to resolving many-to-many relationships. The complex relationship replaced by an association entity and the original entities are related to this new entity. entity related through binary relationships to each of the original entities. The solution is shown in Figure 2 below.

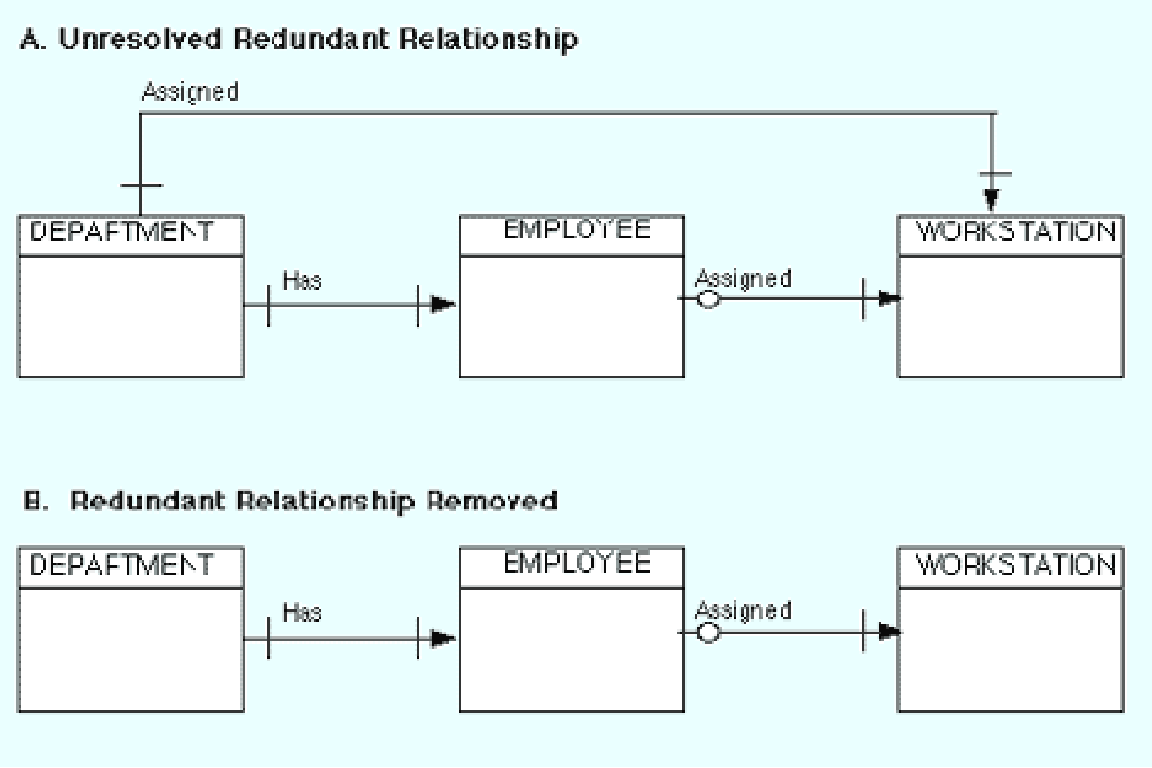
Figure 2: Transforming a Complex Relationship


A. Ternary Relationship Unresolved
EMPLOYEE
SKIL_
PROJECT
B. Ternary Relationship Resolved
EMPLOYEE
PROJECT
SKILL
EMF PRJ SKILL

Eliminate redundant relationships

A redundant relationship is a relationship between two entities that is equivalent in meaning to another relationship between those same two entities that may pass through an intermediate entity. For example, Figure 3A shows a redundant relationship between DEPARTMENT and WORKSTATION. This relationship provides the same information as the relationships DEPARTMENT has EMPLOYEES and EMPLOYEEs assigned WORKSTATION. Figure 3B shows the solution which is to remove the redundant relationship DEPARTMENT assigned WORKSTATIONS.

Figure 3: Removing A Redundant Relationship



Summary

The last two sections have provided a brief overview of the basic constructs in the ER diagram. The next section discusses **Primary and Foreign Keys**.

Primary and Foreign Keys

Primary and foreign keys are the most basic components on which relational theory is based. Primary keys enforce entity integrity by uniquely identifying entity instances. Foreign keys enforce referential integrity by completing an association between two entities. The next step in building the basic data model to

1. identify and define the primary key attributes for each entity
2. validate primary keys and relationships
3. migrate the primary keys to establish foreign keys

Define Primary Key Attributes

Attributes are data items that describe an entity. An *attribute instance* is a single value of an attribute for an instance of an entity. For example, Name and hire date are attributes of the entity EMPLOYEE. "Jane Hathaway" and "3 March 1989" are instances of the attributes name and hire date.

The *primary key* is an attribute or a set of attributes that uniquely identify a specific instance of an entity. Every entity in the data model must have a primary key whose values uniquely identify instances of the entity.

To qualify as a primary key for an entity, an attribute must have the following properties:

- it must have a non-null value for each instance of the entity
- the value must be unique for each instance of an entity
- the values must not change or become null during the life of each entity instance

In some instances, an entity will have more than one attribute that can serve as a primary key. Any key or minimum set of keys that could be a primary key is called a *candidate key*. Once candidate keys are identified, choose one, and only one, primary key for each entity. Choose the identifier most commonly used by the user as long as it conforms to the properties listed above. Candidate keys which are not chosen as the primary key are known as alternate keys.

An example of an entity that could have several possible primary keys is Employee. Let's assume that for each employee in an organization there are three candidate keys: Employee ID, Social Security Number, and Name.

Name is the least desirable candidate. While it might work for a small department where it would be unlikely that two people would have exactly the same name, it would not work for a large organization that had hundreds or thousands of employees. Moreover, there is the possibility that an employee's name could change because of marriage. Employee ID would be a good candidate as long as each employee were assigned a unique identifier at the time of hire. Social Security would work best since every employee is required to have one before being hired.

Composite Keys

Sometimes it requires more than one attribute to uniquely identify an entity. A primary key that made up of more than one attribute is known as a *composite key.* Figure 1 shows an example of a composite key. Each instance of the entity Work can be uniquely identified only by a composite key composed of Employee ID and Project ID.

Figure 1: Example of Composite Key

WORK

Employee ID	Project ID	Hours_Worked
01	01	200
01	02	120
02	01	50
02	03	120
03	03	100
03	04	200

Artificial Keys

An *artificial key*is one that has no meaning to the business or organization. Artificial keys are permitted when 1) no attribute has all the primary key properties, or 2) the primary key is large and complex.

Primary Key Migration

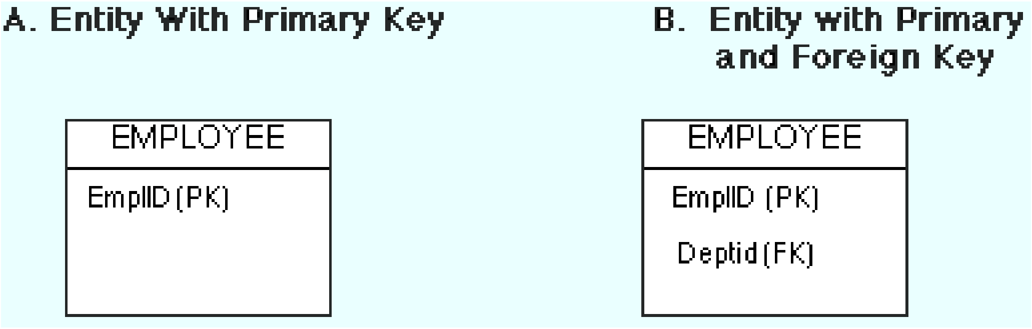
Dependent entities, entities that depend on the existence of another entity for their identification, inherit the entire primary key from the parent entity. Every entity within a generalization hierarchy inherits the primary key of the root generic entity.

Define Key Attributes

Once the keys have been identified for the model, it is time to name and define the attributes that have been used as keys.

There is no standard method for representing primary keys in ER diagrams. For this document, the name of the primary key followed by the notation (PK) is written inside the entity box. An example is shown in Figure 2A.

Figure 2: Entities with Key Attributes


A. Entity With Primary Key
EMPLOYEE
EmplID (PK)
B. Entity with Primary and Foreign Key
EMPLOYEE
EmplID (PK)
Deptid (FK)

Validate Keys and Relationships

Basic rules governing the identification and migration of primary keys are:

- Every entity in the data model shall have a primary key whose values uniquely identify entity instances.
- The primary key attribute cannot be optional (i.e., have null values).
- The primary key cannot have repeating values. That is, the attribute may not have more than one value at a time for a given entity instance is prohibited. This is known as the No Repeat Rule.
- Entities with compound primary keys cannot be split into multiple entities with simpler primary keys. This is called the Smallest Key Rule.
- Two entities may not have identical primary keys with the exception of entities within generalization hierarchies.
- The entire primary key must migrate from parent entities to child entities and from supertype, generic entities, to subtypes, category entities.

Foreign Keys

A *foreign key* is an attribute that completes a relationship by identifying the parent entity. Foreign keys provide a method for maintaining integrity in the data (called referential integrity) and for navigating between different instances of an entity. Every relationship in the model must be supported by a foreign key.

Identifying Foreign Keys

Every dependent and category (subtype) entity in the model must have a foreign key for each relationship in which it participates. Foreign keys are formed in dependent and subtype entities by migrating the entire primary key from the parent or generic entity. If the primary key is composite, it may not be split.

Foreign Key Ownership

Foreign key attributes are not considered to be owned by the entities to which they migrate, because they are reflections of attributes in the parent entities. Thus, each attribute in an entity is either owned by that entity or belongs to a foreign key in that entity.

If the primary key of a child entity contains all the attributes in a foreign key, the child entity is said to be "identifier dependent" on the parent entity, and the relationship is called an "identifying relationship." If any attributes in a foreign key do not belong to the child's primary key, the child is not identifier dependent on the parent, and the relationship is called "non identifying."

Diagramming Foreign Keys

Foreign keys attributes are indicated by the notation (FK) beside them. An example is shown in Figure 2 (b) above.

Summary

Primary and foreign keys are the most basic components on which relational theory is based. Each entity must have a attribute or attributes, the primary key, whose values uniquely identify each instance of the entity. Every child entity must have an attribute, the foreign key, that completes the association with the parent entity.

The next step in building the model is to **Add Attributes to the Model**.

Adding Attributes to the Model

Non-key attributes describe the entities to which they belong. In this section, we discuss the rules for assigning non-key attributes to entities and how to handle multivalued attributes.

Relate attributes to entities

Non-key attributes can be in only one entity. Unlike key attributes, non-key attributes never migrate, and exist in only one entity. from parent to child entities.

The process of relating attributes to the entities begins by the modeler, with the assistance of the end-users, placing attributes with the entities that they appear to describe. You should record your decisions in the entity attribute matrix discussed in the previous section. Once this is completed, the assignments are validated by the formal method of normalization.

Before beginning formal normalization, the rule is to place non-key attributes in entities where the value of the primary key determines the values of the attributes. In general, entities with the same primary key should be combined into one entity. Some other guidelines for relating attributes to entities are given below.

Parent-Child Relationships

- With parent-child relationships, place attributes in the parent entity where it makes sense to do so (as long as the attribute is dependent upon the primary key)
- If a parent entity has no non-key attributes, combine the parent and child entities.

Multivalued Attributes

If an attribute is dependent upon the primary key but is multivalued, has more than one value for a particular value of the key), reclassify the attribute as a new child entity. If the multivalued attribute is unique within the new entity, it becomes the primary key. If not, migrate the primary key from the original, now parent, entity.

For example, assume an entity called PROJECT with the attributes Proj_ID (the key), Proj_Name, Task_ID, Task_Name

PROJECT

Proj_ID	Proj_Name	Task_ID	Task_Name
01	A	01	Analysis
01	A	02	Design
01	A	03	Programming
01	A	04	Tuning
02	B	01	Analysis

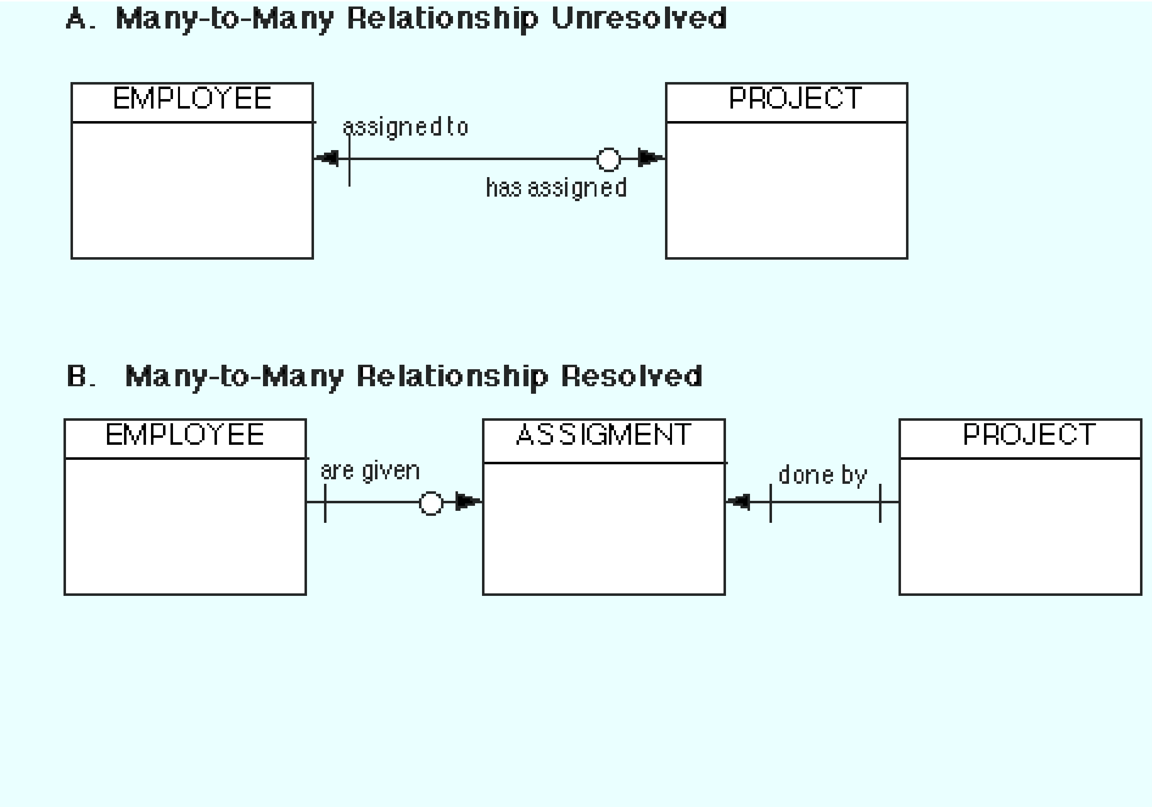
Task_ID and Task_Name have multiple values for the key attribute. The solution is to create a new entity, let's call it TASK and make it a child of PROJECT. Move Task_ID and Task_Name from PROJECT to TASK. Since neither attribute uniquely identifies a task, the final step would be to migrate Proj_ID to TASK.

Attributes That Describe Relations

In some cases, it appears that an attribute describes a relationship rather than an entity (in the Chen notation of ER diagrams this is permissible). For example,

a MEMBER **borrows** BOOKS.

Possible attributes are the date the books were checked out and when they are due. Typically, such a situation will occur with a many-to-many relationship and the solution is the same. Reclassify the relationship as a new entity which is a child to both original entities. In some methodologies, the newly created is called an *associative entity*. See Figure 1 for an example of an converting a relationship into an associative entity.



Derived Attributes and Code Values

Two areas where data modeling experts disagree is whether derived attributes and attributes whose values are codes should be permitted in the data model.

Derived attributes are those created by a formula or by a summary operation on other attributes. Arguments against including derived data are based on the premise that derived data should not be stored in a database and therefore should not be included in the data model. The arguments in favor are:

- derived data is often important to both managers and users and therefore should be included in the data model.
- it is just as important, perhaps more so, to document derived attributes just as you would other attributes
- including derived attributes in the data model does not imply how they will be implemented.

A coded value uses one or more letters or numbers to represent a fact. For example, the value Gender might use the letters "M" and "F" as values rather than "Male" and "Female". Those who are against this practice cite that codes have no intuitive meaning to the end-users and add complexity to processing data. Those in favor argue that many organizations have a long history of using coded attributes, that codes save space, and improve flexibility in that values can be easily added or modified by means of look-up tables.

Including Attributes to the ER Diagram

There is disagreement about whether attributes should be part of the entity-relationship diagram. The IDEF1X standard specifies that attributes should be added. Many experienced practitioners, however, note that adding attributes, especially if there are a large number, clutters the diagram and detracts from its ability to present the end-user with an overview of how the data is structured.

Summary

By the end of this stage you should have:

1. identified, named, and defined data objects and relationships
2. recorded information about data objects and relationships in the data document
3. created and refined the ER diagram
4. assigned attributes to entities
5. added attributes to ER diagram (optional)

The next section discusses **Generalization Hierarchies**.

Generalization Hierarchies

Up to this point, we have discussed describing an object, the entity, by its shared characteristics, the attributes. For example, we can characterize an employee by their employee id, name, job title, and skill set.

Another method of characterizing entities is by both similarities and differences. For example, suppose an organization categorizes the work it does into internal and external projects. Internal projects are done on behalf of some unit within the organization. External projects are done for entities outside of the organization. We can recognize that both types of projects are similar in that each involves work done by employees of the organization within a given schedule. Yet we also recognize that there are differences between them. External projects have unique attributes, such as a customer identifier and the fee charged to the customer. This process of categorizing entities by their similarities and differences is known as *generalization*.

Description

A generalization hierarchy is a structured grouping of entities that share common attributes. It is a powerful and widely used method for representing common characteristics among entities while preserving their differences. It is the relationship between an entity and one or more refined versions. The entity being refined is called the *supertype* and each refined version is called the *subtype.* The general form for a generalization hierarchy is shown in Figure 8.1.

Generalization hierarchies should be used when (1) a large number of entities appear to be of the same type, (2) attributes are repeated for multiple entities, or (3) the model is continually evolving. Generalization hierarchies improve the stability of the model by allowing changes to be made only to those entities germane to the change and simplify the model by reducing the number of entities in the model.

Creating a Generalization Hierarchy

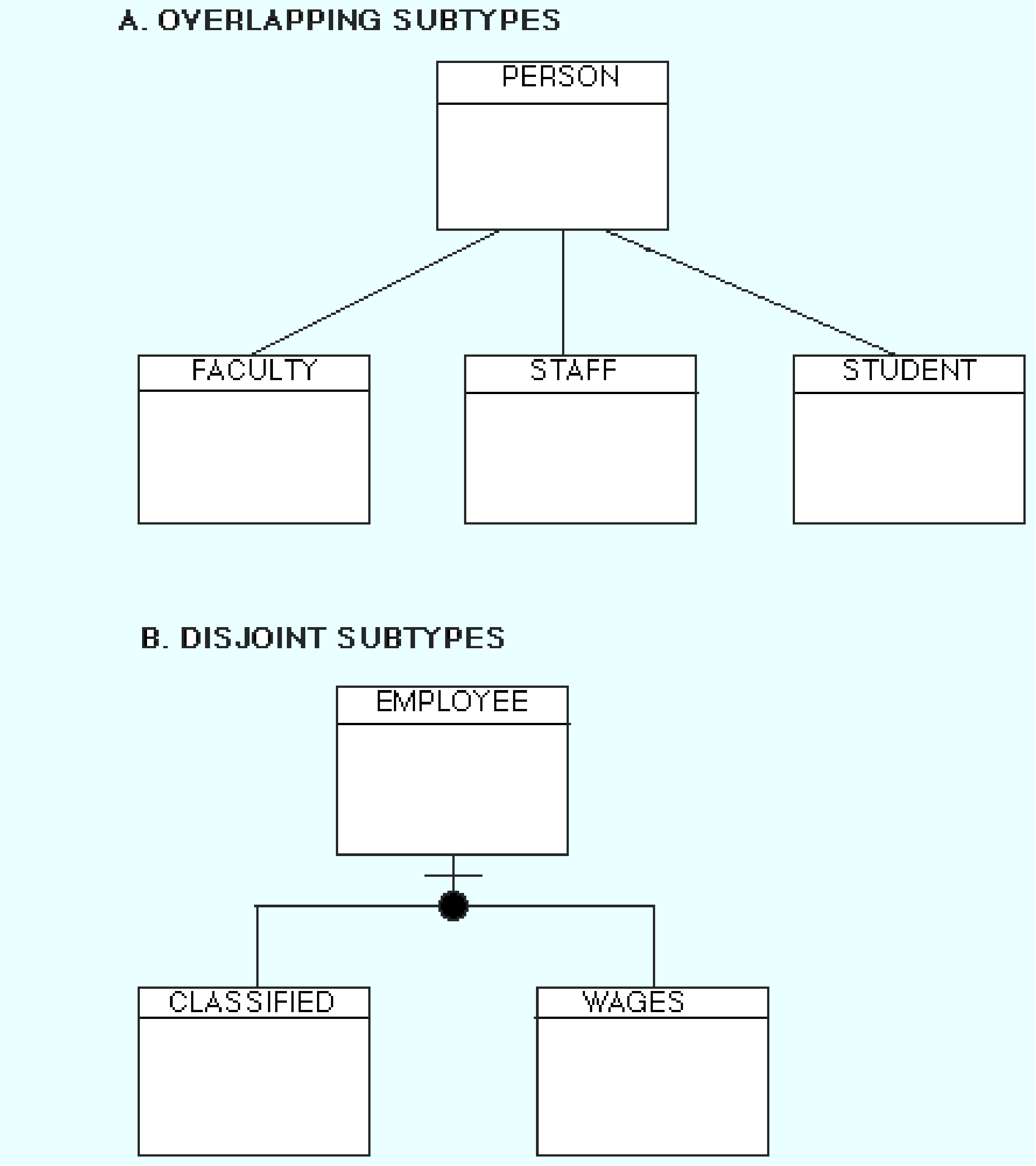
To construct a generalization hierarchy, all common attributes are assigned to the supertype. The supertype is also assigned an attribute, called a discriminator, whose values identify the categories of the subtypes. Attributes unique to a category, are assigned to the appropriate subtype. Each subtype also inherits the primary key of the supertype. Subtypes that have only a primary key should be eliminated. Subtypes are related to the supertypes through a one-to-one relationship.

Types of Hierarchies

A generalization hierarchy can either be overlapping or disjoint. In an overlapping hierarchy an entity instance can be part of multiple subtypes. For example, to represent people at a university you have identified the supertype entity PERSON which has three subtypes, FACULTY, STAFF, and STUDENT. It is quite possible for an individual to be in more than one subtype, a staff member who is also registered as a student, for example.

In a disjoint hierarchy, an entity instance can be in only one subtype. For example, the entity EMPLOYEE, may have two subtypes, CLASSIFIED and WAGES. An employee may be one type or the other but not both. Figure 1 shows A) overlapping and B) disjoint generalization hierarchy.

Figure 1: Examples of Generalization Hierarchies



Rules

The primary rule of generalization hierarchies is that each instance of the supertype entity must appear in at least one subtype; likewise, an instance of the subtype must appear in the supertype.

Subtypes can be a part of only one generalization hierarchy. That is, a subtype can not be related to more than one supertype. However, generalization hierarchies may be nested by having the subtype of one hierarchy be the supertype for another.

Subtypes may be the parent entity in a relationship but not the child. If this were allowed, the subtype would inherit two primary keys.

Summary

Generalization hierarchies are a structure that enables the modeler to represent entities that share common characteristics but also have differences.

The next and final step in the modeling process is to **Add Data Integrity Rules**.

Add Data Integrity Rules

Data integrity is one of the cornerstones of the relational model. Simply stated data integrity means that the data values in the database are correct and consistent.

Data integrity is enforced in the relational model by **entity** and **referential** integrity rules. Although not part of the relational model, most database software enforce attribute integrity through the use of domain information.

Entity Integrity

The entity integrity rule states that for every instance of an entity, the value of the primary key must exist, be unique, and cannot be null. Without entity integrity, the primary key could not fulfill its role of uniquely identifying each instance of an entity.

Referential Integrity

The referential integrity rule states that every foreign key value must match a primary key value in an associated table. Referential integrity ensures that we can correctly navigate between related entities.

Insert and Delete Rules

A foreign key creates a hierarchical relationship between two associated entities. The entity containing the foreign key is the *child*, or dependent, and the table containing the primary key from which the foreign key values are obtained is the *parent*.

In order to maintain referential integrity between the parent and child as data is inserted or deleted from the database certain insert and delete rules must be considered.

Insert Rules

Insert rules commonly implemented are:

* **Dependent**. The *dependent* insert rule permits insertion of child entity instance only if matching parent entity already exists.

* **Automatic**. The *automatic* insert rule always permits insertion of child entity instance. If matching parent entity instance does not exist, it is created.

* **Nullify**. The *nullify* insert rule always permits the insertion of child entity instance. If a matching parent entity instance does not exist, the foreign key in child is set to null.

* **Default**. The *default* insert rule always permits insertion of child entity instance. If a matching parent entity instance does not exist, the foreign key in the child is set to previously defined value.

- **Customized.** The *customized* insert rule permits the insertion of child entity instance only if certain customized validity constraints are met.

- **No Effect.** This rule states that the insertion of child entity instance is always permitted. No matching parent entity instance need exist, and thus no validity checking is done.

Delete Rules

- **Restrict**. The *restrict* delete rule permits deletion of parent entity instance only if there are no matching child entity instances.

- **Cascade.** The *cascade* delete rule always permits deletion of a parent entity instance and deletes all matching instances in the child entity.

- **Nullify**. The *nullify* delete rules always permits deletion of a parent entity instance. If any matching child entity instances exist, the values of the foreign keys in those instances are set to null.

- **Default.** The *default* rule always permits deletion of a parent entity instance. If any matching child entity instances exist, the value of the foreign keys are set to a predefined default value.

- **Customized.** The *customized* delete rule permits deletion of a parent entity instance only if certain validity constraints are met.

- **No Effect.** The *no effect* delete rule always permits deletion of a parent entity instance. No validity checking is done.

Delete and Instert Guidelines

The choice of which rule to use is determined by Some basic guidelines for insert and delete rules are given below.

- Avoid use of nullify insert or delete rules. Generally, the parent entity in a parent-child relationship has mandatory existence. Use of the null insert or delete rule would violate this rule.
- Use either automatic or dependent insert rule for generalization hierarchies. Only these rules will keep the rule that all instances in the subtypes must also be in the supertype.
- Use the cascade delete rule for generalization hierarchies. This rule will enforce the rule that only instances in the supertype can appear in the subtypes.

Domains

A domain is a valid set of values for an attribute which enforce that values from an insert or update make sense. Each attribute in the model should be assigned domain information which includes:

* **Data Type—**Basic data types are integer, decimal, or character. Most data bases support variants of these plus special data types for date and time.
* **Length—**This is the number of digits or characters in the value. For example, a value of 5 digits or 40 characters.
* **Date Format—**The format for date values such as dd/mm/yy or yy/mm/dd
* **Range—**The range specifies the lower and upper boundaries of the values the attribute may legally have
* **Constraints—**Are special restrictions on allowable values. For example, the Beginning_Pay_Date for a new employee must always be the first work day of the month of hire.
* **Null support—**Indicates whether the attribute can have null values
* **Default value (if any)—**The value an attribute instance will have if a value is not entered.

Primary Key Domains

The values of primary keys must be unique and nulls are not allowed.

Foreign Key Domains

The data type, length, and format of primary keys must be the same as the corresponding primary key. The uniqueness property must be consistent with relationship type. A one-to-one relationship implies a unique foreign key; a one-to-many relationship implies a non-unique foreign key.

Overview of the Relational Model

The relational model was formally introduced by Dr. E. F. Codd in 1970 and has evolved since then, through a series of writings. The model provides a simple, yet rigorously defined, concept of how users perceive data. The relational model represents data in the form of two-dimension tables. Each table represents some real-world person, place, thing, or event about which information is collected. A relational database is a collection of two-dimensional tables. The organization of data into relational tables is known as the **logical view** of the database. That is, the form in which a relational database presents data to the user and the programmer. The way the database software physically stores the data on a computer disk system is called the **internal view**. The internal view differs from product to product and does not concern us here.

A basic understanding of the relational model is necessary to effectively use relational database software such as Oracle, Microsoft SQL Server, or even personal database systems such as Access or Fox, which are based on the relational model.

This document is an informal introduction to relational concepts, especially as they relate to relational database design issues. It is not a complete description of relational theory.

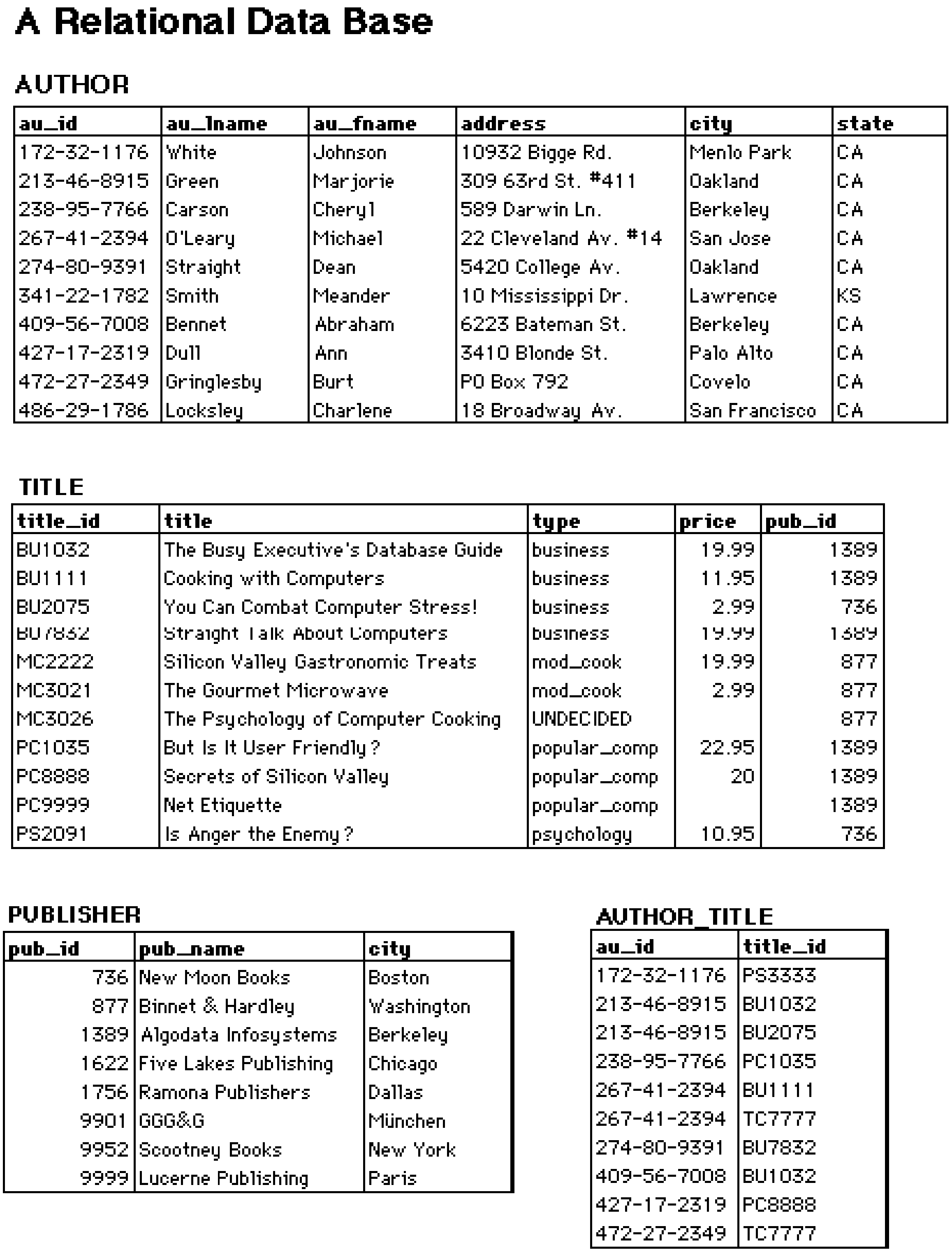
This section discusses the basic concepts—data structures, relationships, and data integrity—that are the basis of the relational model.

- Data Structure and Terminology
- Notation
- Properties of Relational Tables
- Relationships and Keys
- Data Integrity
- Relational Data Manipulation
- Normalization
- Advanced Normalization

Data Structure and Terminology

In the relational model, a database is a collection of relational tables. A relational table is a flat file composed of a set of named columns and an arbitrary number of unnamed rows. The columns of the tables contain information about the table. The rows of the table represent occurrences of the "thing" represented by the table. A data value is stored in the intersection of a row and column. Each named column has a domain, which is the set of values that may appear in that column. **Figure 1** shows the relational tables for a simple bibliographic database that stores information about book title, authors, and publishers.

Figure 1



A Relational Data Base

AUTHOR

au_id	au_lname	au_fname	address	city	state
172-32-1176	White	Johnson	10932 Bigge Rd.	Menlo Park	CA
213-46-8915	Green	Marjorie	309 63rd St. #411	Oakland	CA
238-95-7766	Carson	Cheryl	589 Darwin Ln.	Berkeley	CA
267-41-2394	O'Leary	Michael	22 Cleveland Av. #14	San Jose	CA
274-80-9391	Straight	Dean	5420 College Av.	Oakland	CA
341-22-1782	Smith	Meander	10 Mississippi Dr.	Lawrence	KS
409-56-7008	Bennet	Abraham	6223 Bateman St.	Berkeley	CA
427-17-2319	Dull	Ann	3410 Blonde St.	Palo Alto	CA
472-27-2349	Gringlesby	Burt	PO Box 792	Covelo	CA
486-29-1786	Locksley	Charlene	18 Broadway Av.	San Francisco	CA

TITLE

title_id	title	type	price	pub_id
BU1032	The Busy Executive's Database Guide	business	19.99	1389
BU1111	Cooking with Computers	business	11.95	1389
BU2075	You Can Combat Computer Stress!	business	2.99	736
BU7832	Straight Talk About Computers	business	19.99	1389
MC2222	Silicon Valley Gastronomic Treats	mod_cook	19.99	877
MC3021	The Gourmet Microwave	mod_cook	2.99	877
MC3026	The Psychology of Computer Cooking	UNDECIDED		877
PC1035	But Is It User Friendly?	popular_comp	22.95	1389
PC8888	Secrets of Silicon Valley	popular_comp	20	1389
PC9999	Net Etiquette	popular_comp		1389
PS2091	Is Anger the Enemy?	psychology	10.95	736

PUBLISHER

pub_id	pub_name	city
736	New Moon Books	Boston
877	Binnet & Hardley	Washington
1389	Algodata Infosystems	Berkeley
1622	Five Lakes Publishing	Chicago
1756	Ramona Publishers	Dallas
9901	GGG&G	München
9952	Scootney Books	New York
9999	Lucerne Publishing	Paris

AUTHOR_TITLE

au_id	title_id
172-32-1176	PS3333
213-46-8915	BU1032
213-46-8915	BU2075
238-95-7766	PC1035
267-41-2394	BU1111
267-41-2394	TC7777
274-80-9391	BU7832
409-56-7008	BU1032
427-17-2319	PC8888
472-27-2349	TC7777

There are alternate names used to describe relational tables. Some manuals use the terms tables, fields, and records to describe relational tables, columns, and rows, respectively. The formal literature tends to use the mathematical terms, relations, attributes, and tuples. Figure 2 summarizes these naming conventions.

Figure 2: Terminology

In This Document	***Formal Terms***	***Many Database Manuals***
Relational Table	Relation	Table
Column	Attribute	Field
Row	Tuple	Record

Notation

Relational tables can be expressed concisely by eliminating the sample data and showing just the table name and the column names. For example,

AUTHOR	(au_id, au_lname, au_fname, address, city, state, zip)
TITLE	(title_id, title, type, price, pub_id)
PUBLISHER	(pub_id, pub_name, city)
AUTHOR_TITLE	(au_id, pub_id)

Properties of Relational Tables

Relational tables have six properties:

1. **Values are atomic.**
2. **Column values are of the same kind.**
3. **Each row is unique.**
4. **The sequence of columns is insignificant.**
5. **The sequence of rows is insignificant.**
6. **Each column must have a unique name.**

Values Are Atomic

This property implies that columns in a relational table are not repeating group or arrays. Such tables are referred to as being in the "first normal form" (1NF). The atomic value property of relational tables is important because it is one of the cornerstones of the relational model.

The key benefit of the one value property is that it simplifies data manipulation logic.

Column Values Are of the Same Kind

In relational terms this means that all values in a column come from the same domain. A domain is a set of values which a column may have. For example, a Monthly_Salary column contains only specific monthly salaries. It never contains other information such as comments, status flags, or even weekly salary.

This property simplifies data access because developers and users can be certain of the type of data contained in a given column. It also simplifies data validation. Because all values are from the same domain, the domain can be defined and enforced with the Data Definition Language (DDL) of the database software.

Each Row is Unique

This property ensures that no two rows in a relational table are identical; there is at least one column, or set of columns, the values of which uniquely identify each row in the table. Such columns are called primary keys and are discussed in more detail in **Relationships and Keys**.

This property guarantees that every row in a relational table is meaningful and that a specific row can be identified by specifying the primary key value.

The Sequence of Columns is Insignificant

This property states that the ordering of the columns in the relational table has no meaning. Columns can be retrieved in any order and in various sequences. The benefit of this property is that it enables many users to share the same table without concern of how the table is organized. It also permits the physical structure of the database to change without affecting the relational tables.

The Sequence of Rows is Insignificant

This property is analogous the one above but applies to rows instead of columns. The main benefit is that the rows of a relational table can be retrieved in different order and sequences. Adding information to a relational table is simplified and does not affect existing queries.

Each Column Has a Unique Name

Because the sequence of columns is insignificant, columns must be referenced by name and not by position. In general, a column name need not be unique within an entire database but only within the table to which it belongs.

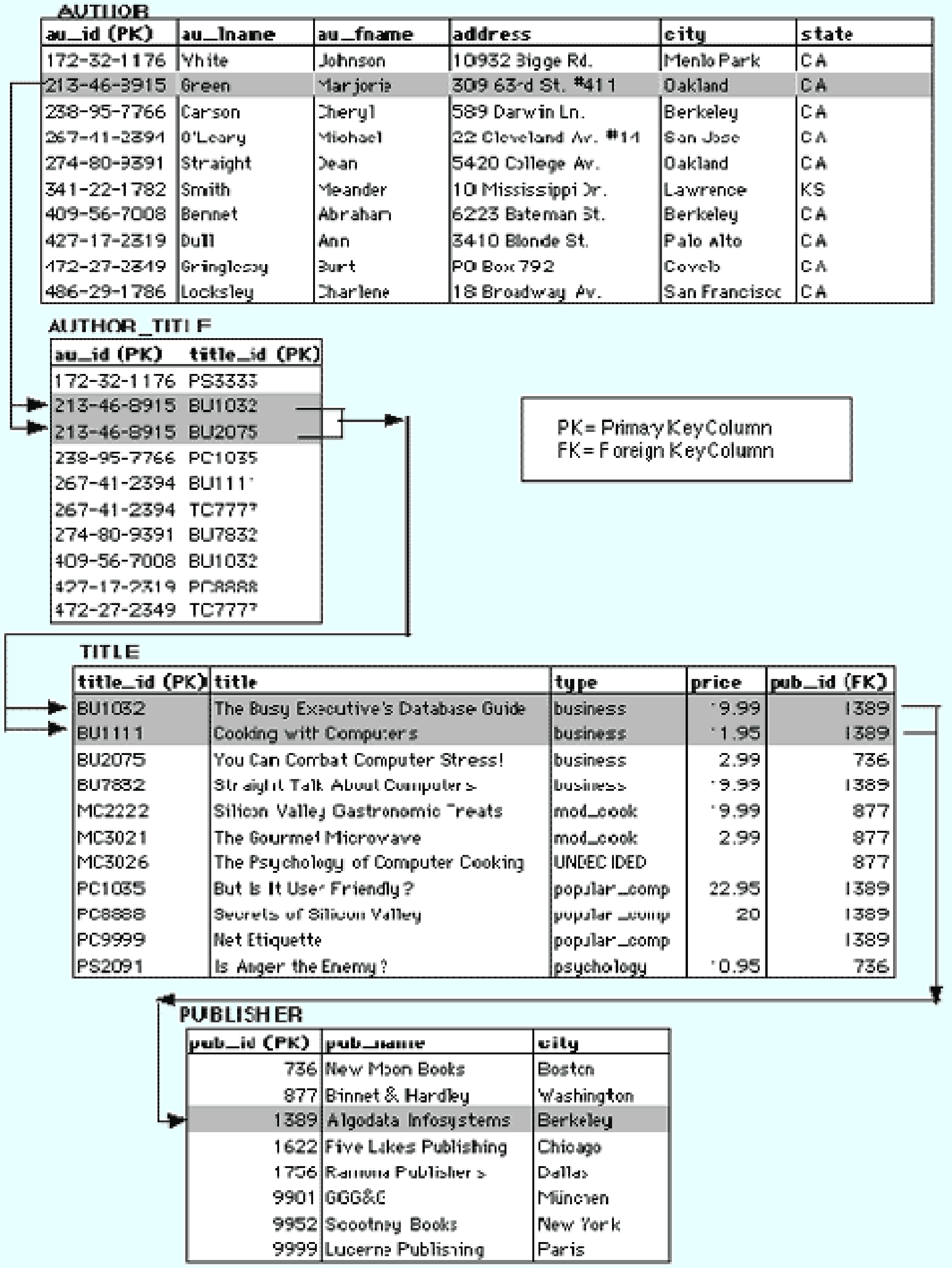
Relationships and Keys

A *relationship* is an association between two or more tables. Relationships are expressed in the data values of the primary and foreign keys.

A *primary key* is a column or columns in a table whose values **uniquely identify** each row in a table. A *foreign key* is a column or columns whose values are the same as the primary key of another table. You can think of a foreign key as a copy of primary key from another relational table. The relationship is made between two relational tables by matching the values of the foreign key in one table with the values of the primary key in another.

Keys are fundamental to the concept of relational databases because they enable tables in the database to be related with each other. Navigation around a relational database depends on the ability of the primary key to unambiguously identify specific rows of a table. Navigating between tables requires that the foreign key is able to correctly and consistently reference the values of the primary keys of a related table. For example, the figure below shows how the keys in the relational tables are used to navigate from AUTHOR to TITLE to PUBLISHER. AUTHOR_TITLE is an all key table used to link AUTHOR and TITLE. This relational table is required because AUTHOR and TITLE have a many-to-many relationship.

Navigating Between Tables Using Keys



AUTHOR

au_id (PK)	au_lname	au_fname	address	city	state
172-32-1176	White	Johnson	10932 Bigge Rd.	Menlo Park	CA
213-46-8915	Green	Marjorie	309 63rd St. #411	Oakland	CA
238-95-7766	Carson	Cheryl	589 Darwin Ln.	Berkeley	CA
267-41-2394	O'Leary	Michael	22 Cleveland Av. #14	San Jose	CA
274-80-9391	Straight	Dean	5420 College Av.	Oakland	CA
341-22-1782	Smith	Meander	10 Mississippi Dr.	Lawrence	KS
409-56-7008	Bennet	Abraham	6223 Bateman St.	Berkeley	CA
427-17-2319	Dull	Ann	3410 Blonde St.	Palo Alto	CA
472-27-2349	Gringlesby	Burt	PO Box 792	Covelo	CA
486-29-1786	Locksley	Charlene	18 Broadway Av.	San Francisco	CA

AUTHOR_TITLE

au_id (PK)	title_id (PK)
172-32-1176	PS3333
213-46-8915	BU1032
213-46-8915	BU2075
238-95-7766	PC1035
267-41-2394	BU1111
267-41-2394	TC7777
274-80-9391	BU7832
409-56-7008	BU1032
427-17-2319	PC8888
472-27-2349	TC7777

PK = Primary Key Column
FK = Foreign Key Column

TITLE

title_id (PK)	title	type	price	pub_id (FK)
BU1032	The Busy Executive's Database Guide	business	19.99	1389
BU1111	Cooking with Computers	business	11.95	1389
BU2075	You Can Combat Computer Stress!	business	2.99	736
BU7832	Straight Talk About Computers	business	19.99	1389
MC2222	Silicon Valley Gastronomic Treats	mod_cook	19.99	877
MC3021	The Gourmet Microwave	mod_cook	2.99	877
MC3026	The Psychology of Computer Cooking	UNDECIDED		877
PC1035	But Is It User Friendly?	popular_comp	22.95	1389
PC8888	Secrets of Silicon Valley	popular_comp	20	1389
PC9999	Net Etiquette	popular_comp		1389
PS2091	Is Anger the Enemy?	psychology	10.95	736

PUBLISHER

pub_id (PK)	pub_name	city
736	New Moon Books	Boston
877	Binnet & Hardley	Washington
1389	Algodata Infosystems	Berkeley
1622	Five Lakes Publishing	Chicago
1756	Ramona Publishers	Dallas
9901	GGG&G	München
9952	Scootney Books	New York
9999	Lucerne Publishing	Paris

Data Integrity

Data integrity means, in part, that you can correctly and consistently navigate and manipulate the tables in the database. There are two basic rules to ensure data integrity; entity integrity and referential integrity.

The *entity integrity* rule states that the value of the primary key, can never be a null value (a null value is one that has no value and is not the same as a blank). Because a primary key is used to identify a unique row in a relational table, its value must always be specified and should never be unknown. The integrity rule requires that insert, update, and delete operations maintain the uniqueness and existence of all primary keys.

The *referential integrity rule* states that if a relational table has a foreign key, then every value of the foreign key must either be null or match the values in the relational table in which that foreign key is a primary key.

Relational Data Manipulation

Relational tables are sets. The rows of the tables can be considered as elements of the set. Operations that can be performed on sets can be done on relational tables. The eight relational operations are:

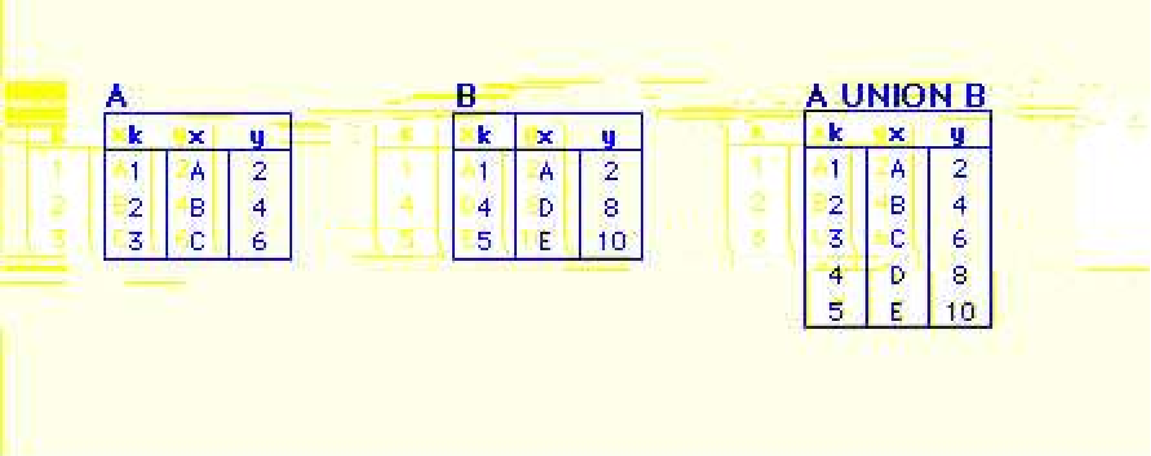
Union

The *union* operation of two relational tables is formed by appending rows from one table with those of a second table to produce a third. Duplicate rows are eliminated. The notation for the union of Tables A and B is A UNION B.

The relational tables used in the union operation must be union compatible. Tables that are union compatible must have the same number of columns and corresponding columns must come from the same domain. Figure1 shows the union of A and B.

Note that the duplicate row [1, A, 2] has been removed.

Figure1: A UNION B



A

k	x	y
1	A	2
2	B	4
3	C	6

B

k	x	y
1	A	2
4	D	8
5	E	10

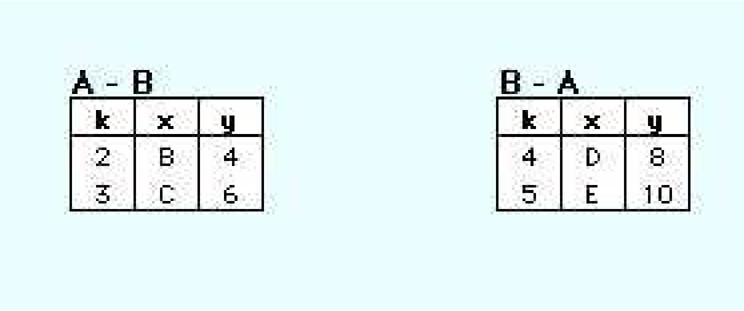
A UNION B

k	x	y
1	A	2
2	B	4
3	C	6
4	D	8
5	E	10

Difference

The *difference* of two relational tables is a third that contains those rows that occur in the first table but not in the second. The Difference operation requires that the tables be union compatible. As with arithmetic, the order of subtraction matters. That is, A - B is not the same as B - A. Figure2 shows the different results.

Figure 2: The Difference Operator



A - B

k	x	y
2	B	4
3	C	6

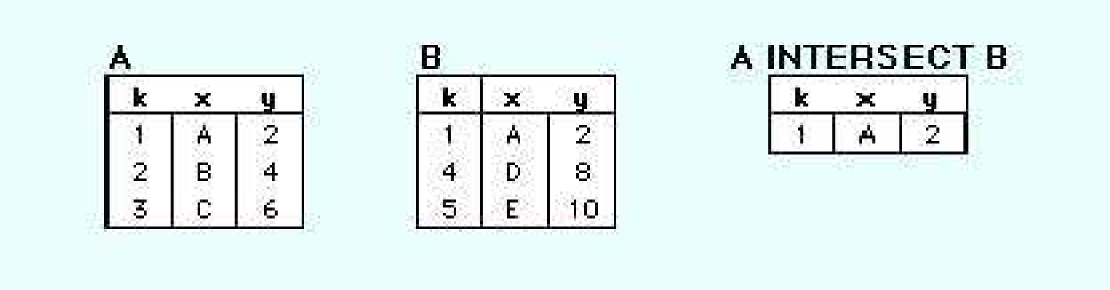
B - A

k	x	y
4	D	8
5	E	10

Intersection

The intersection of two relational tables is a third table that contains common rows. Both tables must be union compatible. The notation for the intersection of A and B is A [intersection] B = C or A INTERSECT B. Figure3 shows the single row [1, A, 2] appears in both A and B.

Figure3: Intersection



A

k	x	y
1	A	2
2	B	4
3	C	6

B

k	x	y
1	A	2
4	D	8
5	E	10

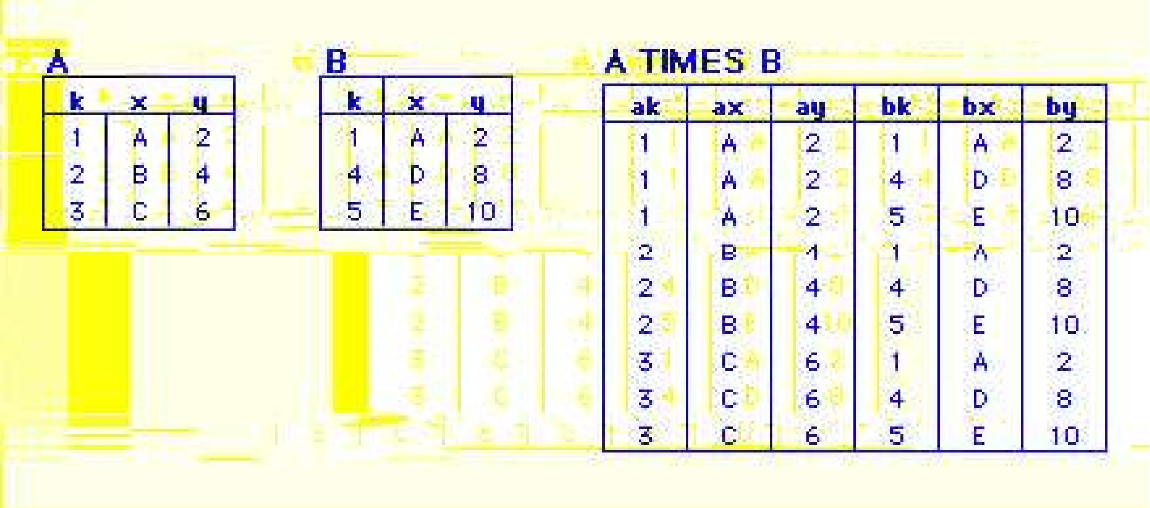
A INTERSECT B

k	x	y
1	A	2

Product

The product of two relational tables, also called the Cartesian Product, is the concatenation of every row in one table with every row in the second. The product of table A (having m rows) and table B (having n rows) is the table C (having m x n rows). The product is denoted as A X B or A TIMES B.

Figure 4: Product



A

k	x	y
1	A	2
2	B	4
3	C	6

B

k	x	y
1	A	2
4	D	8
5	E	10

A TIMES B

ak	ax	ay	bk	bx	by
1	A	2	1	A	2
1	A	2	4	D	8
1	A	2	5	E	10
2	B	4	1	A	2
2	B	4	4	D	8
2	B	4	5	E	10
3	C	6	1	A	2
3	C	6	4	D	8
3	C	6	5	E	10

The product operation is by itself, not very useful. However, it is often used as an intermediate process in a Join.

Projection

The *project* operator retrieves a *subset of columns* from a table, removing duplicate rows from the result.

Selection

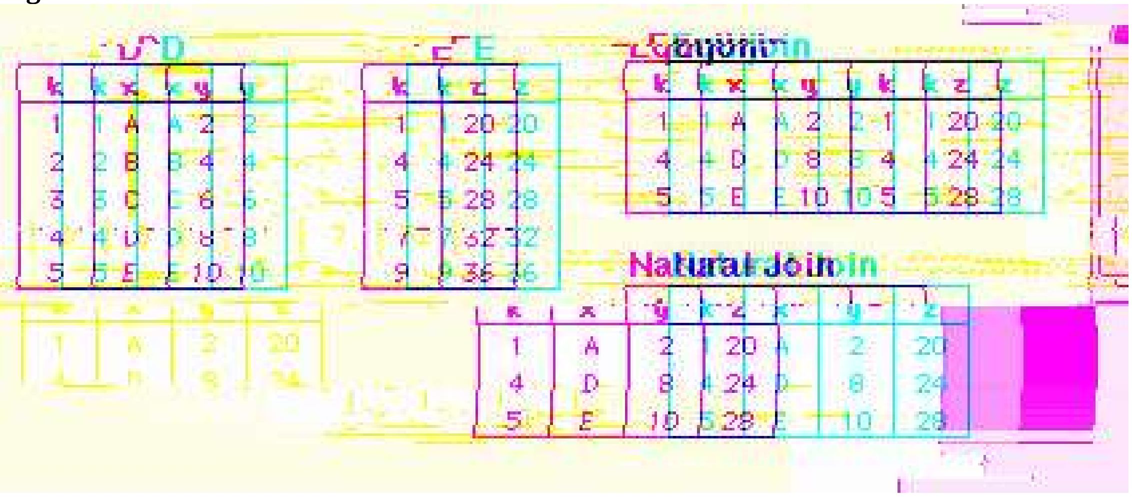
The select operator, sometimes called restrict to prevent confusion with the SQL SELECT command, retrieves subsets of rows from a relational table based on a value(s) in a column or columns.

Join

A *join* operation combines the product, selection, and, possibly, projection. The join operator horizontally combines (concatenates) data from one row of a table with rows from another or the same table when certain criteria are met. The criteria involve a relationship among the columns in the join relational table. If the join criterion is based on equality of column value, the result is called an *equijoin*. A *natural join*, is an equijoin with redundant columns removed.

Figure 5 illustrates a join operation. Tables D and E are joined based on the equality of k in both tables. The first result is an equijoin. Note that there are two columns named k; the second result is a natural join with the redundant column removed.

Figure 5: Join



D

k	x	y
1	A	2
2	B	4
3	C	6
4	D	8
5	E	10

E

k	z
1	20
4	24
5	28
7	32
9	36

Equijoin

k	x	y	k	z
1	A	2	1	20
4	D	8	4	24
5	E	10	5	28

Natural Join

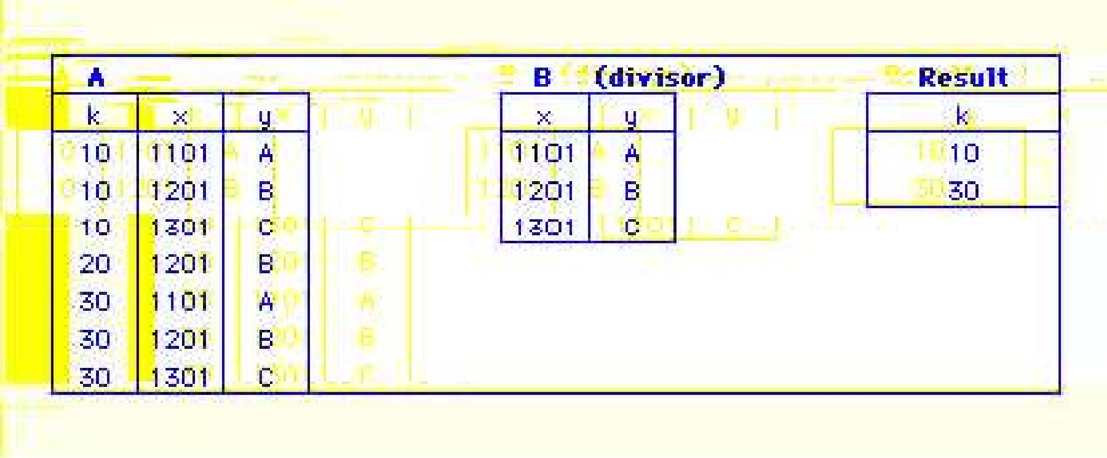
k	x	y	z
1	A	2	20
4	D	8	24
5	E	10	28

Joins can also be done on criteria other than equality.

Division

The division operator results in columns values in one table for which there are other matching column values corresponding to every row in another table.

Figure 6: Division



A

k	x	y
10	1101	A
10	1201	B
10	1301	C
20	1201	B
30	1101	A
30	1201	B
30	1301	C

B (divisor)

x	y
1101	A
1201	B
1301	C

Result

k
10
30

Normalization

Normalization is a design technique that is widely used as a guide in designing relational databases. Normalization is essentially a two step process that puts data into tabular form by removing repeating groups and then removes duplicated from the relational tables.

Normalization theory is based on the concepts of **normal forms**. A relational table is said to be a particular normal form if it satisfied a certain set of constraints. There are currently five normal forms that have been defined. In this section, we will cover the first three normal forms that were defined by E. F. Codd.

Basic Concepts

The goal of normalization is to create a set of relational tables that are free of redundant data and that can be consistently and correctly modified. This means that all tables in a relational database should be in the third normal form (3NF). A relational table is in 3NF if and only if all non-key columns are (a) mutually independent and (b) fully dependent upon the primary key. Mutual independence means that no non-key column is dependent upon any combination of the other columns. The first two normal forms are intermediate steps to achieve the goal of having all tables in 3NF. In order to better understand the 2NF and higher forms, it is necessary to understand the concepts of functional dependencies and lossless decomposition.

Functional Dependencies

The concept of functional dependencies is the basis for the first three normal forms. A column, Y, of the relational table R is said to be **functionally dependent** upon column X of R if and only if each value of X in R is associated with precisely one value of Y at any given time. X and Y may be composite. Saying that column Y is functionally dependent upon X is the same as saying the values of column X identify the values of column Y. If column X is a primary key, then all columns in the relational table R must be functionally dependent upon X.

A short-hand notation for describing a functional dependency is:

R.x —>; R.y

which can be read as in the relational table named R, column x functionally determines (identifies) column y.

Full functional dependence applies to tables with composite keys. Column Y in relational table R is fully functional on X of R if it is functionally dependent on X and not functionally dependent upon any subset of X. Full functional dependence means that when a primary key is composite, made of two or more columns, then the other columns must be identified by the entire key and not just some of the columns that make up the key.

Overview

Simply stated, normalization is the process of removing redundant data from relational tables by decomposing (splitting) a relational table into smaller tables by projection. The goal is to have only primary keys on the left hand side of a functional dependency. In order to be correct, decomposition must be lossless. That is, the new tables can be recombined by a natural join to recreate the original table without creating any spurious or redundant data.

Sample Data

Data taken from Date [Date90] is used to illustrate the process of normalization. A company obtains parts from a number of suppliers. Each supplier is located in one city. A city can have more than one supplier located there and each city has a status code associated with it. Each supplier may provide many parts. The company creates a simple relational table to store this information that can be expressed in relational notation as:

FIRST (s#, status, city, p#, qty)

where

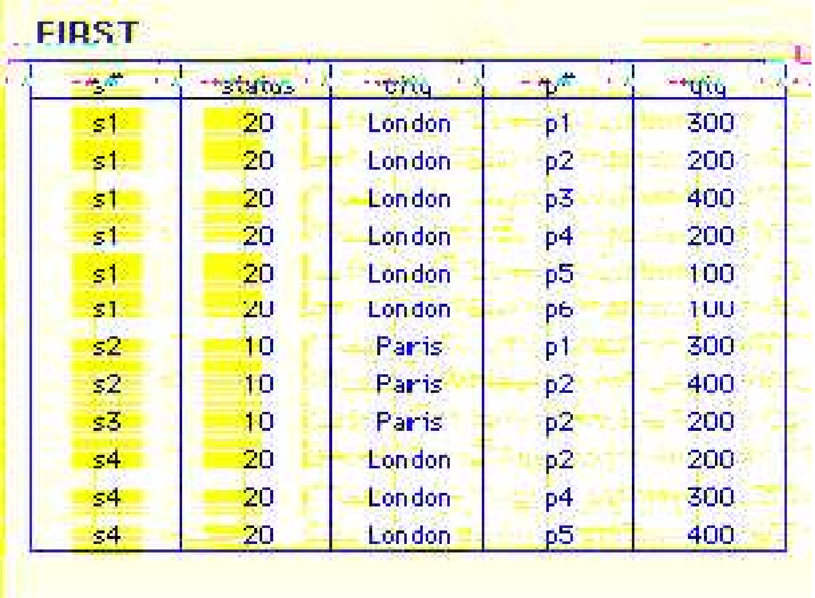
s#	supplier identifcation number (this is the primary key)
status	status code assigned to city
city	name of city where supplier is located
p#	part number of part supplied
qty>	quantity of parts supplied to date

In order to uniquely associate quantity supplied (qty) with part (p#) and supplier (s#), a composite primary key composed of s# and p# is used.

First Normal Form

A relational table, by definition, is in first normal form. All values of the columns are atomic. That is, they contain no repeating values. Figure1 shows the table FIRST in 1NF.

Figure 1: Table in 1NF



FIRST

s#	status	city	p#	qty
s1	20	London	p1	300
s1	20	London	p2	200
s1	20	London	p3	400
s1	20	London	p4	200
s1	20	London	p5	100
s1	20	London	p6	100
s2	10	Paris	p1	300
s2	10	Paris	p2	400
s3	10	Paris	p2	200
s4	20	London	p2	200
s4	20	London	p4	300
s4	20	London	p5	400

Although the table FIRST is in 1NF it contains redundant data. For example, information about the supplier's location and the location's status have to be repeated for every part supplied. Redundancy causes what are called *update anomalies*. Update anomalies are problems that arise when information is inserted, deleted, or updated. For example, the following anomalies could occur in FIRST:

- INSERT. The fact that a certain supplier (s5) is located in a particular city (Athens) cannot be added until they supplied a part.
- DELETE. If a row is deleted, then not only is the information about quantity and part lost but also information about the supplier.
- UPDATE. If supplier s1 moved from London to New York, then six rows would have to be updated with this new information.

Second Normal Form

The definition of second normal form states that only tables with composite primary keys can be in 1NF but not in 2NF.

> A relational table is in second normal form 2NF if it is in 1NF and every non-key column is fully dependent upon the primary key.

That is, every non-key column must be dependent upon the entire primary key. FIRST is in 1NF but not in 2NF because status and city are functionally dependent upon only on the column s# of the composite key (s#, p#). This can be illustrated by listing the functional dependencies in the table:

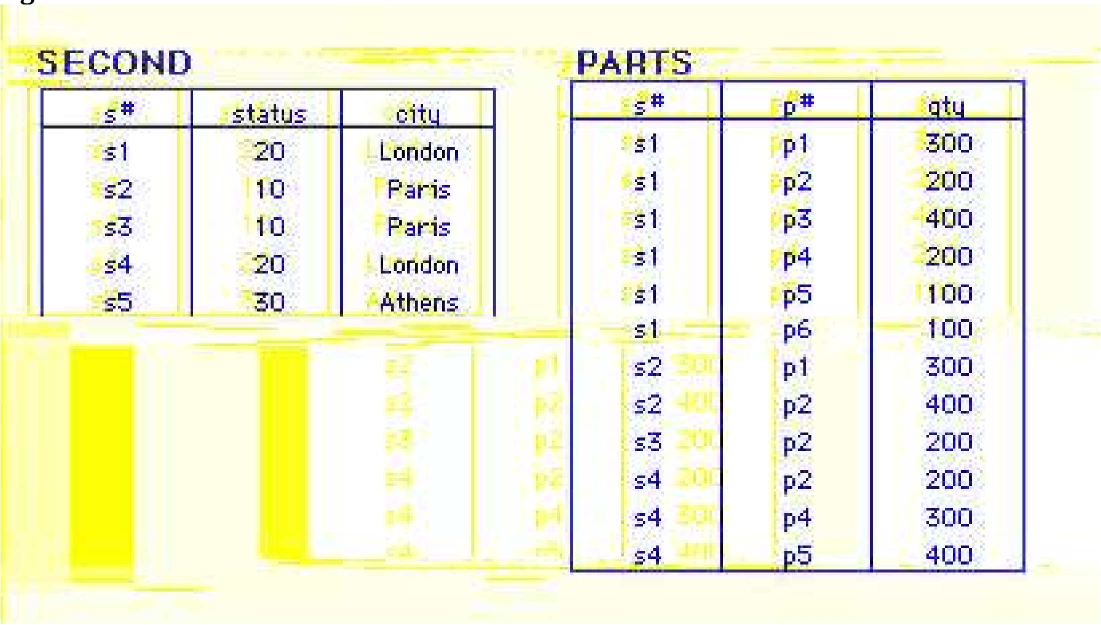
s# —> city, status
city —> status
(s#,p#) —> qty

The process for transforming a 1NF table to 2NF is:

1. Identify any determinants other than the composite key, and the columns they determine.
2. Create and name a new table for each determinant and the unique columns it determines.
3. Move the determined columns from the original table to the new table. The determinate becomes the primary key of the new table.
4. Delete the columns you just moved from the original table except for the determinate which will serve as a foreign key.
5. The original table may be renamed to maintain semantic meaning.

To transform FIRST into 2NF we move the columns s#, status, and city to a new table called SECOND. The column s# becomes the primary key of this new table. The results are shown below in Figure 2.

Figure 2: Tables in 2NF



SECOND

s#	status	city
s1	20	London
s2	10	Paris
s3	10	Paris
s4	20	London
s5	30	Athens

PARTS

s#	p#	qty
s1	p1	300
s1	p2	200
s1	p3	400
s1	p4	200
s1	p5	100
s1	p6	100
s2	p1	300
s2	p2	400
s3	p2	200
s4	p2	200
s4	p4	300
s4	p5	400

Tables in 2NF but not in 3NF still contain modification anomalies. In the example of SECOND, they are:

INSERT. The fact that a particular city has a certain status (Rome has a status of 50) cannot be inserted until there is a supplier in the city.

DELETE. Deleting any row in SUPPLIER destroys the status information about the city as well as the association between supplier and city.

Third Normal Form

The third normal form requires that all columns in a relational table are dependent only upon the primary key. A more formal definition is:

> A relational table is in third normal form (3NF) if it is already in 2NF and every non-key column is non transitively dependent upon its primary key. In other words, all nonkey attributes are functionally dependent only upon the primary key.

Table PARTS is already in 3NF. The non-key column, qty, is fully dependent upon the primary key (s#, p#). SUPPLIER is in 2NF but not in 3NF because it contains a *transitive dependency*. A transitive dependency is occurs when a non-key column that is a determinant of the primary key is the determinate of other columns. The concept of a transitive dependency can be illustrated by showing the functional dependencies in SUPPLIER:

SUPPLIER.s# —> SUPPLIER.status
SUPPLIER.s# —> SUPPLIER.city
SUPPLIER.city —> SUPPLIER.status

Note that SUPPLIER.status is determined both by the primary key s# and the non-key column city. The process of transforming a table into 3NF is:

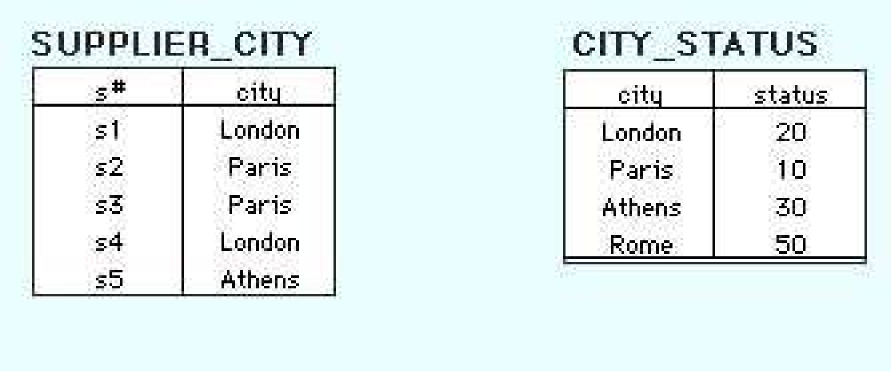
1.
Identify any determinants, other the primary key, and the columns they determine.

2. Create and name a new table for each determinant and the unique columns it determines.
3. Move the determined columns from the original table to the new table. The determinate becomes the primary key of the new table.
4.
Delete the columns you just moved from the original table except for the determinate which will serve as a foreign key.

5. The original table may be renamed to maintain semantic meaning.

To transform SUPPLIER into 3NF, we create a new table called CITY_STATUS and move the columns city and status into it. Status is deleted from the original table, city is left behind to serve as a foreign key to CITY_STATUS, and the original table is renamed to SUPPLIER_CITY to reflect its semantic meaning. The results are shown in Figure 3 below.

Figure 3: Tables in 3NF



SUPPLIER_CITY

s#	city
s1	London
s2	Paris
s3	Paris
s4	London
s5	Athens

CITY_STATUS

city	status
London	20
Paris	10
Athens	30
Rome	50

The results of putting the original table into 3NF has created three tables. These can be represented in "psuedo-SQL" as:

PARTS (#s, p#, qty)
Primary Key (s#,#p)
Foreign Key (s#) references SUPPLIER_CITY.s#

SUPPLIER_CITY(s#, city)
Primary Key (s#)
Foreign Key (city) references CITY_STATUS.city

CITY_STATUS (city, status)
Primary Key (city)

Advantages of Third Normal Form

The advantages to having relational tables in 3NF is that it eliminates redundant data which in turn saves space and reduces manipulation anomalies. For example, the improvements to our sample database are:

> INSERT. Facts about the status of a city, Rome has a status of 50, can be added even though there is not supplier in that city. Likewise, facts about new suppliers can be added even though they have not yet supplied parts.
>
> DELETE. Information about parts supplied can be deleted without destroying information about a supplier or a city. UPDATE. Changing the location of a supplier or the status of a city requires modifying only one row.

Advanced Normalization

After 3NF, all normalization problems involve only tables which have three or more columns and all the columns are keys. Many practitioners argue that placing entities in 3NF is generally sufficient because it is rare that entities that are in 3NF are not also in 4NF and 5NF. They further argue that the benefits gained from transforming entities into 4NF and 5NF are so slight that it is not worth the effort. However, advanced normal forms are presented because there are cases where they are required.

Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) is a more rigorous version of the 3NF deal with relational tables that had (a) multiple candidate keys, (b) composite candidate keys, and (c) candidate keys that overlapped .

BCNF is based on the concept of determinants. A determinant column is one on which some of the columns are fully functionally dependent.

A relational table is in BCNF if and only if every determinant is a candidate key.

Fourth Normal Form

A relational table is in the *fourth normal form* (4NF) if it is in BCNF and all multivalued dependencies are also functional dependencies.

Fourth normal form (4NF) is based on the concept of *multivalued dependencies* (MVD). A Multivalued dependency occurs when in a relational table containing at least three columns, one column has multiple rows whose values match a value of a single row of one of the other columns. A more formal definition given by Date is:

> given a relational table R with columns A, B, and C then
>
> R.A —>> R.B (column A multidetermines column B)
>
> is true if and only if the set of B-values matching a given pair of A-values and C-values in R depends only on the A-value and is independent of the C-value.
>
> MVD always occur in pairs. That is R.A —>> R.B holds if and only if R.A —>> R.C also holds.

Suppose that employees can be assigned to multiple projects. Also suppose that employees can have multiple job skills. If we record this information in a single table, all three attributes must be used as the key since no single attribute can uniquely identify an instance.

The relationship between emp# and prj# is a multivalued dependency because for each pair of emp#/skill values in the table, the associated set of prj# values is determined only by emp# and is independent of skill. The relationship between emp# and skill is also a multivalued dependency, since the set of Skill values for an emp#/prj# pair is always dependent upon emp# only.

To transform a table with multivalued dependencies into the 4NF move each MVD pair to a new table. The result is shown in Figure1.

Figure 1: Tables in 4NF

Fifth Normal Form

A table is in the *fifth normal form* (5NF) if it cannot have a lossless decomposition into any number of smaller tables.

While the first four normal forms are based on the concept of functional dependence, the fifth normal form is based on the concept of join dependence. Join dependency means that an table, after it has been decomposed into three or more smaller tables, must be capable of being joined again on common keys to form the original table. Stated another way, 5NF indicates when an entity cannot be further decomposed. 5NF is complex and not intuitive. Most experts agree that tables that are in the 4NF are also in 5NF except for "pathological" cases. Teorey suggests that true many-to-many-to-many ternary relations are one such case.

Adding an instance to an table that is not in 5NF creates spurious results when the tables are decomposed and then rejoined. For example, let's suppose that we have an employee who uses design skills on one project and programming skills on another. This information is shown below.

emp#	prj#	skill
1211	11	Design
1211	28	Program

Next we add an employee (1544) who uses programming skills on Project 11.

emp#	prj#	skill
1211	11	Design
1211	28	Program
1544	11	Program

Next, we project this information into three tables as we did above. However, when we rejoin the tables, the recombined table contains spurious results.

emp#	prj#	skill
1211	11	Design

1211	11	Program	<<—spurious data
1211	28	Program	
1544	11	Design	<<—spurious data
1544	11	Program	

By adding one new instance to a table not in 5NF, two false assertions were stated:

Assertion 1

- Employee 1211 has been assigned to Project 11.
- Project 11 requires programming skills.
- Therefore, Employee 1211 must use programming skills while assigned to Project 11.

Assertion 2

- Employee 1544 has been assigned to project 11.
- Project 11 needs Design skills.
- Therefore, Employee 1544 must use Design skills in Project 11.

Works Cited

Batini, C., S. Ceri, S. Kant, and B. Navathe. *Conceptual Database Design: An Entity Relational Approach*. The Benjamin/Cummings Publishing Company, 1991.

Date, C. J. An Introduction to Database Systems, 5th ed. Addison-Wesley, 1990.

Fleming, Candace C. and Barbara von Halle. *Handbook of Relational Database Design*. Addison-Wesley, 1989.

Kroenke, David. *Database Processing, 2nd ed*. Science Research Associates, 1983.

Martin, James. *Information Engineering*. Prentice-Hall, 1989.

Reingruber, Michael C. and William W. Gregory. *The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models*. John Wiley & Sons, Inc., 1994.

Simsion, Graeme. *Data Modeling Essentials: Analysis, Design, and Innovation*. International Thompson Computer Press, 1994.

Teory, Toby J. *Database Modeling & Design: The Basic Principles*, 2nd ed.. Morgan Kaufmann Publishers, Inc., 1994.