Database Management Systems

Ulysses G. Alviar
Elfredy V. Cadapan
Jaderick P. Pabico

University of the Philippines
OPEN UNIVERSITY

Database Management Systems
by Ulysses G. Alviar, Elfredy V. Cadapan and Jaderick P. Pabico

Published in the Philippines by the UP Open University
Office of Academic Support and and Instructional Services
2/F, National Computer Center
CP Garcia Avenue, Diliman, Quezon City 1101
Telefax (632) 426-1515
Email oasis@upou.net

First Published 1998
Reprinted 2001, 2006

Layout by Helen Mercado-Creer

Printed in the Philippines

Table of Contents

Unit I
Introduction to Database Management

Module 1
Fundamental Database Concepts

Ever since computers have existed, they have been used to store and process information. Everything we do with computers today, every application, relies on the computer's ability both to store huge amounts of data, and to execute arithmetic, financial, and other types of operations on that data in order to produce useful information.

In this module, we'll go over some concepts commonly used when discussing databases and database design.

Objectives

At the end of this module, you should be able to:

1. Define common database terms such as *database, database management system, record* and *field*;
2. Identify the most common problem areas dealt with in the database design; and
3. Define the three conceptual levels used in database definition and design.

Databases

A **database**, in computer terms, is a collection of data, typically data about one specific enterprise. Strictly speaking, a database does not have to be stored in a computer; employee records stored in a filing cabinet would qualify as a database, for example.

In normal conversation, people often say "database" when they actually mean "database management system". A **database management system** or DBMS is a collection of interrelated data, plus the software and hardware required to access that data in a useful manner. Generally, a DBMS refers to a database or set of databases stored in disk or other media,

a computer or set of computers upon which programs to access the database can run. A DBMS can be as simple as a list of phone numbers of your friends stored in a personal computer, or it can be as complex as a system run by a company that handles inventory, billing, ordering, and payroll for its worldwide operations.

A DBMS should provide a convenient and efficient interface for both storing and retrieving data, as well as for extracting useful information from the database. Due to the importance of databases to most organizations in general, the study and development of techniques for efficient DBMS is a major field of information technology.

SAQ 1-1

List five tasks around your workplace, home, or school which could be done faster or more efficiently with the help of a DBMS.

Fundamental Database Systems Concepts

Let's say a business (like a grocery store) wants to make use of computers to keep track of its inventory and supply functions. A list of suppliers (and their addresses and phone numbers is typed into a file, as well as a list of items which those suppliers sell, including the price of each item, which particular supplier carries it, and how much it weighs (for shipping and handling costs). The company would also want to keep track of how many of each item they have in stock, and which items have run out and have to be reordered.

Each item on lists such as these is commonly referred to as a record. One record may hold information pertaining to one supplier, or one item. Thus, a database of suppliers stored on a computer is stored in a set or list of records, one after the other.

A record, on closer examination, is composed of several sub-components, referred to as **fields**. Each field holds a single value or datum of the record. For a record holding information about a supplier, for example, we can have one field for the supplier's company name, another for his telephone number, and yet another for his address. You might think of a record as a set of boxes for holding information, each box holding a single field. For example:

Supplier	
Name	
Tel. No.	
Address	

Our database would be composed of lists of records with all records in one list having the same fields. For example:

Supplier	
Name	
Tel. No.	
Address	
Supplier	
Name	
Tel. No.	
Address	

And so on.

We now have a way of storing data on the computer. Once data are stored however, we need some way of extracting it in some useful form.

Returning to our example, the grocery store would then have a collection of small application programs, perhaps written by the company programmer, that:

- adds new items to the inventory, whenever a supplier makes a delivery;
- deletes items from the inventory, whenever items are sold;
- adds and deletes types of items, when the store decides to carry new brands or when some items are no longer available;
- changes prices of items, whenever the suppliers' prices change;

- changes supplier information for an item, when suppliers stop carrying a certain brand, for example, or when they start selling new ones;
- prints total purchases of every type of item, when the store manager wants to see how many of some type of item have been bought or sold; or which items have been selling more; and
- prints out graphs of sales per day, month, or year, for when the manager has to go out and do a presentation for the store owners or prospective suppliers.

And so on. This rather unwieldy collection is a standard **file-processing system**, in common use when easy-to-use DBMSes were not widely available. Usually, such a DBMS has a design and structure that was not well-thought out beforehand; typically, more lists and programs are added onto the system as needs arise. In fact, some organizations still use their modern DBMS packages in this manner. Such a system has several major drawbacks:

- **Unplanned data redundancy and inconsistency.** In such a system, the same piece of data may be duplicated in several files. For example, the name of the supplier might be stored in the list of suppliers, and in the list of items (to identify which suppliers carry which items). This may result in wasted disk space and longer access time. But a worse problem would be keeping both copies of the data consistent across all files. For example, a supplier marked as "Acme Inc." in the list of suppliers must also be marked as "Acme Inc." in the list of items; a single error where "Acme Ltd." was placed instead might cause that supplier to suddenly have no items for sale, since the entry in the items list still reads "Acme Inc.". Also, any update to the supplier list must necessitate a similar update to the items list.

- **Isolating data.** Writing application programs to retrieve and correlate data gathered from many files would be difficult, especially if one wanted to watch out for the inconsistencies described above. Also, if more than one programmer were working on the project, they would have to agree on one format or (as is usually the case) the files may have to be converted to new formats before they would work with a newer version of the system.

- **Difficulty in accessing data.** Requests not anticipated by the programmers would require extra programming time, or would require the user to manually process the data on his/her own. For example, if the store owner needed to see a list of suppliers who supplied vegetables, they would either need to reclassify items to find out which items were "vegetables", then write an application program to print the list,

or they would print a list of suppliers and their items for sale, and pick out the suppliers they wanted manually. This is neither efficient nor convenient for both programmer and user.

- **Security.** Some functions of the system should not be available to all users. For example, the store manager might be the only one allowed to add new suppliers and items (to protect against embezzlement by employees), the cashier might be allowed to remove items from inventory, the stockroom staff may be allowed to add to inventory, and all other employees might only be able to display the current inventory levels, but not to edit anything. Implementing this sort of security scheme may be difficult, if not impossible, for some systems.

- **Data ranges.** The data stored may have pre-specified constraints on them-for example, the price of an item must never be negative. The application programs must detect such impossible values. Checking for these valid values must be done by the applications of the system, but updating these ranges at a later date (for example, if new postal regulations allow six-number zip codes, instead of only 4-number ones) may involve more work than the company expected.

- **Multiple access.** Larger database systems may be accessed at multiple terminals, and may allow simultaneous updating of the data. From our example, there may be a terminal at the manager's office, one at the cash register and another in the stockroom. If two users attempt to update the same data at the same time, there must be some sort of arbitration routine that checks that the data being inputted is still valid. Generally, this is a routine that allows only one user to edit a file at any given time, but allows more than one user to read a file at the same time. If the application programs were not written with multi-user use in mind, it may require rewriting or replacement of large parts of the old code.

Developments in database management systems have generally been in one or more of these problem areas.

SAQ 1-2

Select one application from the list you wrote down for SAQ 1-1. What problems would you expect to run into if you were tasked with creating a DBMS for it?

Data Abstraction

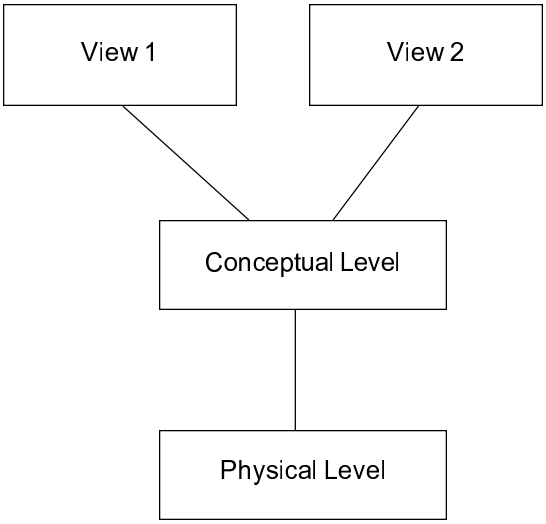
One major purpose of a database management system (or any software system, for that matter) is to make it easier for users to access and modify the data stored in the system. To this end, DBMSes provide users with an **abstracted** view of the system, concealing certain "low-level" details of data storage and maintenance. A store manager should not be expected to want to know how the data is stored on the computer's disk; he would want to know how to get the data out in a usable form. Conversely, a programmer seeking to make the data storage system more efficient might want to know exactly how the data goes onto the disk.

When describing DBMSs, we have several **levels of abstraction** that describe how each class of user or programmer looks at the system.

- **Physical level**. This is the "nuts-and-bolts" view of the system, where the data structures and file formats used by the system are described. This level is generally not visible to end-users, and is usually only used by the system programmers. Newer DBMSs even hide this level from the application programmers, giving them only the conceptual view of the data.

- **Conceptual level**. Here the user is given what data are actually stored in the system, and what relationships they have with each other. This is the level used by the database administrator, who is responsible for what information is allowed into the database.

- **View level**. Here only part of the database is given to the user. Security considerations may come into play here; a user view might allow someone to view a certain part of the database, but not to modify it. For example, an employee may look up his/her own salary rate, but he/she may not be allowed to look up the salary rates of his/her boss, or co-workers. The view given to the Head of Personnel, on the other hand, might not only allow the Head to view all salaries, but even to modify them as he/she sees fit.



Each level is more general in its description than the one before it. Let's say we have our inventory database, one part of which is a list of items in inventory. Each item has the following data stored:

The name of the item
The retail price of the item
How many of this item are currently in inventory

In a description of the database at the physical level (one which a systems programmer might use), we may find how each piece of data is stored in memory and on disk. If the DBMS were written in Pascal, for example, we may specify the item list to consist of a series of records, where each record is of the form:

```
type item = record
        name : string[80];
        price : real;
        quantity : integer;
end;
```

Here we see how many bytes each item stored would take up on disk or memory, what data types (for this particular programming language) the programmer must use, and the range of valid values we can store in the item list. We see that *name* is specified as a string of 80 characters in length; thus we cannot have items which have names that exceed 80 characters. A user who finds that he has to type in a longer name has to use abbreviations. Similarly, we see that *quantity* is stored as an integer; therefore we can never have negative values for quantity.

While this information may be quite useful for the programmer, most of it is not needed by the day-to-day user of the system. The office manager or cashier would not be expected to know or even care about how the data he or she is typing in is stored on disk; all they care about is that the data can be placed onto computer, and they can get that data out again. Here we need a higher, more general picture of the system, and here the conceptual and view levels of abstraction come in.

The database administrator (who in most cases is not the programmer, and might not have any programming skills) must decide upon what data must go into the system, and how each piece of data is interrelated to each other. Taking our previous example further, the grocery store manager, who will ultimately be the database administrator, knows that he has to store a database of items and a database of suppliers, and somehow store sufficient data in the system for it to determine which suppliers sell which items, and at which price. A description of the data at the conceptual level may only contain descriptions of the data definitions, i.e., we can describe the two lists above as :

> **items**, with fields name, price and quantity
> **suppliers**, with fields name, address, and telephone number

Furthermore, we may have descriptions of how such lists interrelate, such as "each supplier may carry several items, and an item may be offered by many suppliers". These descriptions are dependent to a large part on the data model (see Module 2) used by the DBMS designer.

The view level is much like the conceptual level, except that some parts of the database are not visible. A specific view is created for every class of user of the system. These classes are usually assigned on the basis of security, or how much of the database a given user is allowed to access.

From our example, the view given to the grocery store manager would include all the databases we've enumerated. The view granted to the cashier might only include the items database and the inventory levels database; he/she does not need to see the list of suppliers. The view granted to the stockroom chief, however, requires the list of suppliers, if he/she is in charge of ordering new items for the store.

SAQ 1-3

Write sample records and fields for the application you described in SAQ 1-1 and 1-2. How would these records appear in a document describing the database at the physical level? the conceptual level?

Summary

A database is a collection of data pertaining to one entity. A database management system (DBMS) is a collection of related databases, and the software required to access those databases in a useful manner. Designing proper DBMSs requires attention to several problem areas, such as data redundancy and inconsistency, data isolation, difficulty in data access, security, valid data ranges, and multiple access.

There are three conceptual levels we can use when describing a DBMS. The physical level pertains to the low-level structures and routines required by the database, such as those required by a system programmer. The conceptual level deals with the different databases that make up the system, and the relationships they have with each other. This is typically the level at which the database administrator works. The view level describes different "views" or portions of the DBMS which are accessible to a specific class of user. Different users may have different views of the data, depending on their needs and privileges.

References

Date, C J. 1990. *An introduction to database systems.* Reading, Massachusetts: Addison-Wesley Publishing Co., Inc. 5th ed.

Korth, HF and Silberschatz, A. 1986. *Database system concepts.* Singapore: McGraw-Hill Book Co., Intl. Ed.

Module 2
Data Models

Before a programmer can build a database management system it must first be designed. The design and planning of a database management system ranges from the general high-level system design, to the individual low-level data formats to be used. In the previous module, we saw how data abstraction can be used to specify the framework upon which the DBMS will be built. In this module we discuss several data models that may be used in planning and designing a database. We also examine several historical models that you may run into when maintaining or upgrading old database systems.

Objectives

At the end of this module, you should be able to:

1. Discuss and compare the various data models; and
2. Create a database design using these data models.

Entities, Attributes, and Keys

Data models are ways of organizing a database structure in order to better understand how to implement it on a given system. A data model looks at databases in terms of **entities** and **relationships**. An entity is a piece of data to be stored, pertaining to one real-world object, person, place, or what-have-you. In a banking database, a customer would be considered an entity. A bank account would also be considered another entity.

Entities have **attributes**, which are the smaller pieces of information on a particular entity. Examples of attributes would be names, dates, measurements; in our bank account example, attributes of the entity "customer" might be "name", "address" and so on. Attributes of the entity "account" might be "account number", "date opened", "current balance". Note that the assignment of attributes to specific entities is up to the database designer. It's up to him or her to make sure the attributes make sense.

To easily distinguish specific entities from each other, we have to know which attributes are different between them. For example, we can tell two bank customers apart by their last names. Therefore, we say that "last name" is a key to our database. This is well and good, assuming no two customers have the same last names. Unfortunately, such is not the case in real life; for example, relatives may open separate accounts with a bank.

We can further distinguish two people with the same last names by also taking into account their first names. Taken together, the attributes "first name" and "last name" constitute a superkey of our database. Using the superkey, we can identify one customer uniquely. Of course, it's perfectly possible that there exist two people with the same first and last names, but in many real world applications, this is considered rare enough to be ignored. (This is why many database applications will mistake you for someone else if that someone else has the same first and last name that you have.)

Actually, a superkey is a set of attributes that uniquely identify an entity. So, "first name", "last name", "address" and "phone number" taken together is also a superkey.

Taking too many attributes as a key will result in unwieldy and error-prone programs, however. We usually take the minimal superkey, or the primary key, as the key to use when searching for a specific entity in the database.

Relationships

Relationships are just what they sound like; they define how entities are related to each other. A customer can own a particular bank account, so here we say "owning" is a relationship between a customer and an account. This particular relationship is one-way; i.e., a customer may own a bank account, but the bank account can't own the customer (at least not in the literal sense!). We say a relationship is one-to-one when only one entity can have that relationship with another entity. In a monogamous society, for example, one husband can only have one wife (legally.) There-

fore, we can say the "married" relationship is one-to-one, between the entities "husband" and "wife".

A **one-to-many** relationship, on the other hand, means that one entity can have more than one of that type of relationship. Going back to our bank example, if a customer can own many accounts, but each account can only be owned by one customer (no joint accounts) then that relationship is one-to-many. Of course, from the perspective of the bank account, the relationship is many-to-one.

Lastly, the many-to-many relationship means we can have any number of relationships with other entities. If we now allow joint accounts in our banking example, and customers can own more than one account, then we have some customers who own more than one account, and some accounts owned by more than one customer.

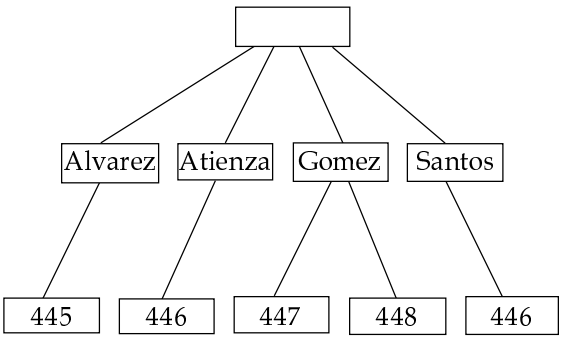
Data Models

The hierarchical model

One of the first popular data models to be developed was the hierarchical model, used in the late 1960's by IBM and MRI, among others. IBM's Information Management System (IMS) was based on the hierarchical model, and was developed for use in the Apollo space program.

Its name derives from the tree structures it uses to organize data - here, entities and relationships are in the form of a tree. By this time, algorithms for searching, storing, and sorting elements of trees were in common use, and were naturally put to work in implementing the various functions of the hierarchical databases.

Using our banking example, let us say we have two entities - customers, and bank accounts. A hierarchical model of the database system might look something like this:

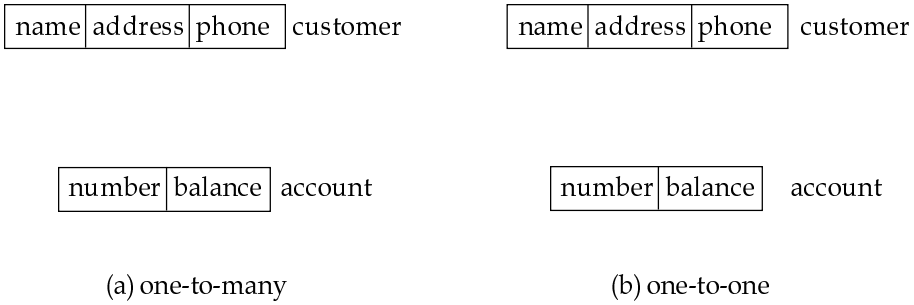


Here, we have a **dummy node** which marks where the tree starts, called the **root.** We also have links, denoted by the lines, between the records stored in the tree. Each level of the tree denotes one entity, thus all records on the second level are customers, and all records on the bottom level are accounts. If a customer owns an account, then there exists a link beween that customer record and the account record. Gomez therefore owns two accounts, 447 and 448, while the others own one each.

The links are actually pointers from one record to another. In practice, a link is simply a number that indicates the file position of the next record. The customer record containing "Alvarez", for instance, will contain the file position of the account record "445". There is no way to go from one customer to another; to search for another customer, you'll have to start over from the root node.

One drawback of the hierarchical model is that while one may have one-to-many relationships, many-to-one and many-to-many relationships (many customers own one account) result in redundant records. In the diagram, customers Atienza and Santos both own account number 446. Since in a tree we cannot have any cycles (we cannot have a child node owned by two parents) we have to duplicate the record for account 446, thus adding the need to keep both records updated.

When specifying the structure of a hierarchical database, we use a **tree-structure diagram**. This diagram consists of boxes for records, and arrows denoting relationships for the database. The direction of the arrow denotes what type of relationship is present.

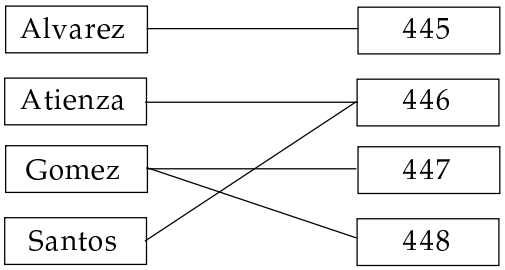


Here, a.) denotes a one-to-many relationship - the arrowhead shows which entity is the "one" - while b.) denotes a one-to-one relationship.

The network model

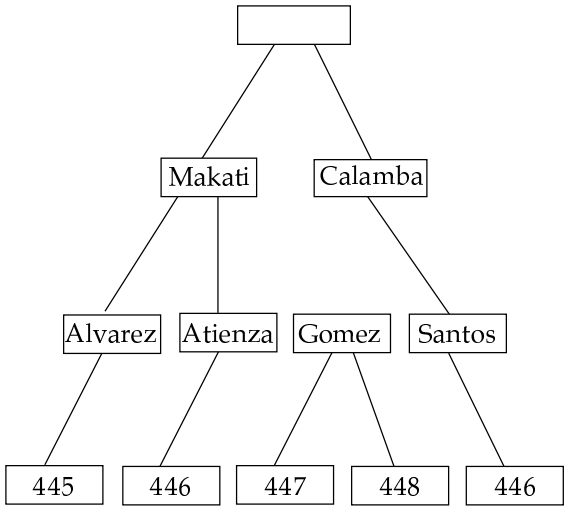
The network model emerged in the late '60s, almost at the same time as the hierarchical model, developed by GE. Many commercial DBMSes were based on this model, including IDS II (Honeywell), Total, and ADABAS (Software AG).

In the network model, data is again stored in records, but records may have any number of links to other records representing other entities, depending on the type of relationship. Thus, our example bank database using the network model would look like:



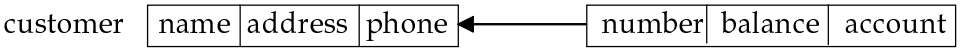
Note that the model handles both one-to-many and many-to-one relationships easily, and can handle many-to-many relationships with similar ease. Each customer has links to those accounts he or she owns, and each account is linked to its owners.

SAQ 2-1

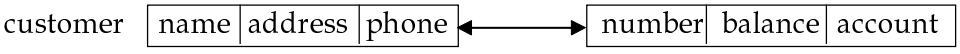


Given this three-level hierarchical database, we add a new entity, branch, with attributes "city" and "manager name". Assume a customer can only belong to one branch, and an account can only belong to one customer. How would the tree-structure diagram look?

A database built on the network model may be represented by a data-structure diagram, as shown below:



(a) one-to-many



(b) one-to-one

Again, as in the hierarchical tree-structure diagram, relationship types are denoted by arrows, with arrowheads denoting where only one instance of that entity may join in a relationship.

The links in a network model database denote file positions of the next record. Unfortunately, unlike the hierarchical model, there isn't just one starting point when searching through the database. Usually we keep pointers to one record of each entity type, and move them to different parts of the file in response to our search. If we are looking for an account owned by Gomez, for example, we first move the customer pointer to the file position of the record containing "Gomez", read the link to the account record, and move the account pointer to that position.

The relational model

The most common data model in use in the '80s and '90s is the relational model. Most DBMS packages for microcomputers use this model, hence its widespread use, whereas the other two models are usually only found on older mainframe and minicomputer systems.

Here we have the idea of relations between records organized into tables. Each record has relations with other records, and that relationship is stored in a table containing the primary keys of each entity.

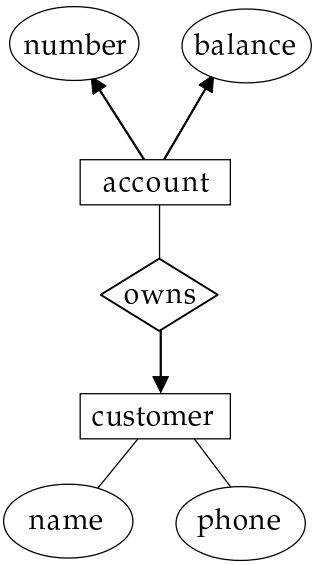
Alvarez	445
Atienza	446
Gomez	447
Gomez	448
Santos	446

Here, it is the customer entity which has relations with the account entity; each record in the "owns" table holds space for storing the primary key of its counterpart in the each entity. Thus, we can see which customers own which accounts.

Many-to-many relations can be stored using multiple primary keys for one of the keys of the customer entity, as in the case of Gomez, above. Also, note that while the primary key 446 is duplicated for both Atienza and Santos (denoting joint ownership of that account), only the key is duplicated, and not the entire record, as in the case of the hierarchical model.

Relational databases are represented conceptually with the use of **entity-relationship diagrams**. Briefly, each entity is represented by boxes, and relations by diamonds in the diagram. Lines show which entities have which relations, and what kind (again, with the arrowheads.)

For example:



The arrowhead from owns to customer denotes that one customer can own many accounts, but one account can only be owned by one customer. The ovals contain the attributes of each entity.

We have an entire unit devoted to the study of relational databases, so we'll stop here. In the meantime…

SAQ 2-2

How would you add the "branch" entity to our relational example above? Draw an example relational table for the "is a customer of" (as in "Gomez is a customer of our Calamba branch") relation. Assume the primary key for branch is "city" (only one branch per city).

Summary

In this module we discussed further database concepts, including the idea of entities, attributes, and relationships. Entities contain information on one real-world object, person, place, etc. and attributes describe that entity. Entities have relationships with each other, and we wish to store not only the attributes of each entity, but also the relationships each one may have with others.

Some attributes may be used to identify each entity uniquely. The minimal set of attributes that does so is called the primary key.

When describing a database's structure, we use one of several data models. We examined the hierarchical model, which organizes objects in the form of a tree, and is specified using the tree-structure diagram. The network model organizes objects in the form of a linked set of records, and is specified using a data-structure diagram. Lastly, we touched briefly on the relational model, which uses tables to denote relationships between entities. To create a relational database specification, we use an entity-relationship diagram.

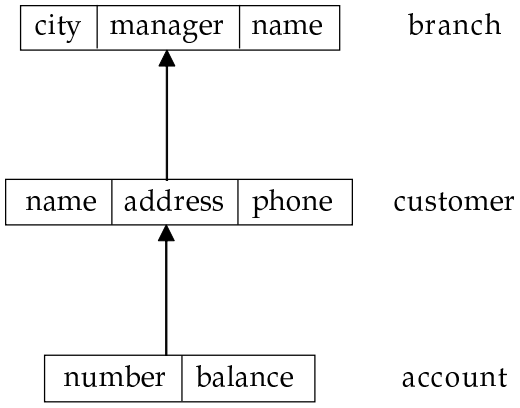
References

Date. J. 1990. *An introduction to database systems*. Reading, Massachusetts: Addison-Wesley Publishing Co., Inc. 5th ed.

Korth, H.F. and Silberschatz, A. 1986. *Database System Concepts.* Singapore. McGraw-Hill Book Co. Intl. Ed.

Answers to Self-Assessment Questions

ASAQ 2-1



Make sure your relationships are right. Once you hand the tree-structure diagram off to the programmer, a many-to-one relationship where it should have been one-to-many might take weeks of work to fix.

ASAQ 2-2

Here we are:

Calamba	Alvarez
Los Baños	Atienza
Calamba	Gomez
Makati	Santos

You can see that we have a one-to-many relationship here, where one branch can have many customers. Doing a many-to-many relation would be just as easy.