

---

THE BASIS OF AN EU CO-FINANCED PROJECT "QUALIFICATION IMPROVEMENT OF THE LITHUANIAN GEOGRAPHICAL INFORMATION MANAGERS" (NO BPD2004-ESF-2.2.0.-02-05/0143)

Civil servants learning programme  
DISTANCE LEARNING OF GEOGRAPHIC INFORMATION INFRASTRUCTURE

Training Material  
**GEOGRAPHIC DBMS**  
**GII-05**

Vilnius, 2008

Training material „Geographic DBMS“ (GII-05)

Author

Dave Cake

Reviewed and edited by

doc. dr. Žilvinas Stankevičius (Vilnius Gediminas Technical University)

Reviewed by

doc. dr. Gintautas Mozgeris (Lithuanian University of Agriculture)

From English translated and edited by

Astraneta UAB

Study material for civil servants distance learning programme “GII Distance Learning”, which has been approved by a law “RE Approval of Study Programmes for Civil Servants” (No 27V-72; Apr 17, 2007) by the State Service Department under the Ministry of the Interior has been prepared on the basis of an EU co-financed project “Qualification Improvement of the Lithuanian Geographical Information Managers” (No BPD2004-ESF-2.2.0.-02-05/0143)

Beneficiary of the project National Land Service under the Ministry of Agriculture.

Contractor of the project HNIT-BALTIC UAB, Vilnius Gediminas Technical University

© Author exclusive copy rights belongs to National Land Service under the Ministry of Agriculture

© Author moral rights belongs to Malaspina University-College

## **GEOGRAPHIC DBMS**

### ***Table of Contents***

1	Database Foundations.....	4
1.1	Background and Terminology.....	6
1.2	Data Modeling .....	18
1.3	The Relational Model.....	41
	References.....	72
2	Implementation .....	73
2.1	Implementing Class Diagrams.....	74
2.2	ESRI Data Structures .....	83
2.3	Geodatabase Elements .....	95
2.4	Creating a Geodatabase .....	109
	References.....	115
3	Multi-user Geodatabases.....	116
3.1	Multi-user Databases.....	117
3.2	The ESRI Multi-User Geodatabase .....	125
3.3	Geodatabase Administration .....	139
	References.....	145
4	Additional Topics .....	146
4.1	Structured Query Language (SQL).....	147
4.2	Spatial Indexing .....	164
4.3	Temporal Data Storage .....	172
4.4	Distributed DBMS.....	181
	References.....	191

# 1 Database Foundations

This module serves as the theoretical foundation for the course. Topic 1 defines the basic terminology of databases and an examination of the changes in database structure in recent years. Topic 2 focuses on the process of defining a data model, or database structure, to accommodate data for a specific purpose. We will examine a notation to describe data structures and gain practical experience creating models. We conclude this module with a discussion of the most common database structure, the relational model. We will learn the benefits of good database design to minimize integrity problems with our data, and practice the process of determining if our designs are properly structured.

Module Outline:

Topic 1:	Database Background and Terminology
Topic 2:	Database Design
Topic 3:	The Relational Model

## 1.0.1 Course Overview

This course is about managing geographic, or spatial data. To a large degree, that process involves relying on established database structures and techniques, but there are topics which are specific to the Geographic Information System (GIS) or spatial database realm. To address this topic, then, we will begin with universally-applicable database theory. We will learn the common terminology, structures and methods that are just as applicable to banking as to landuse planning. With a foundation in databases we will move on to concerns specific to geographic applications.

The structure of the course also moves from abstract to concrete. We begin with material relating to underlying structures in databases and methods used for creating high-level database designs. During the design stage we make an effort not to address parts of the database which depend on the hardware or software that will be used. We are more concerned with which data elements to include and how they relate to one another. We then begin discussing how to implement the database design, or how to create a real database using specific database application software. Finally, we will look at how to fill these data structures and how to use and maintain them. In these latter topics of the course, we will be addressing software-specific themes and discussing how the software is used to maintain the data.

By virtue of the volume of spatial data currently being produced, spatial databases and effective management of spatial data have become increasingly important for performance reasons and because of the pervasiveness of the technology; people are more likely to work with spatial data as part of their daily work. A typical GIS manager, particularly in a small organisation without a dedicated GIS *database* manager, will often be faced with database design decisions or work with consultants performing this work.

Module 1 of this course serves as the theoretical foundation for the course. Here we will look at different database models and terminology, and begin to design database models using the Unified Modeling Language (UML) class diagram notation. We will learn how to structure databases to minimize data integrity problems.

Module 2 begins to move from abstract design principles to more concrete implementation issues. Here we will explore how the UML designs translate into tables, rows and columns in a finished database. We will also begin to explore the process of creating geodatabases – defining structures, loading data into those structures, and maintaining them over time. We will discuss tools within the geodatabase itself which will help us preserve the integrity of our data, including the use of domains, subtypes and topology.

Module 3 will focus on the multiuser geodatabase. We will explore the general principles involved in having several people working on a database at the same time. For practical discussion, we will focus on the ArcSDE (Spatial Database Engine) product and its use. We will include a discussion of SDE from both a user's and an administrator's perspective.

Module 4 will explore additional topics relating to spatial database management. We will work with standard Structured Query Language (SQL) to manipulate databases, and discuss extensions to SQL to work perform spatial operations. We will also discuss spatial indexing, approaches for managing temporal data, and options for parallel or distributed database implementations.

## 1.1 Background and Terminology

We will begin with some very basic background material: what a database is, why database design is important and some supporting terminology. We will also consider some different types of database systems and data models.

### 1.1.1 Data, Information and Knowledge

**Data** are facts. A datum (one unit of data) is a symbol used to represent something such as a word, a number or a date. Raw data are facts that describe persons, places, things or events. Some references will consider Information and Data synonymous, but for the purposes of this course we will make a distinction. **Information** is data that has been placed in a meaningful context, or processed or presented into a form that is meaningful to the recipient.

We can see that what appears in Table 1-1 below is clearly a list of **names** and **numbers**, but we can't know the purpose or meaning of them without being told the context or interpretation.

**Table 1-1 Data without Context**

DeNiro, R.	49333223	C
Bunyan, P.	33456327	A
Gretzky, W.	22004837	B

This is essentially a list of facts, or data. The data is useless in its current form without an interpretation. Table 1-1Table 1-2, below, shows this same data placed in context.

**Table 1-2 Data with Context: Information**

Name	Student Number	Grade Average
DeNiro, R.	49333223	C
Bunyan, P.	33456327	A
Gretzky, W.	22004837	B

With context, we can easily see that this is a class list and resulting grades for a course. This is useful to someone who must make decisions about who passes a course and who fails. Here, the data become information.

The term Information here is one of many instances where a word with a wide meaning is appropriated for use with a narrow meaning. For those in management information systems (MIS) information must have value. We'll see this idea of taking words with broad meanings and applying them to a specific field with a very narrow meaning over and over in GIS. Every discipline has its own language, and part of learning GIS is understanding the terminology clearly.

**Knowledge** is derived from information. Information or data may be interpreted differently, depending on the existing knowledge of the person doing the interpreting. Information can be

acquired by being told, but Knowledge can be acquired by thinking. Thus new knowledge can be acquired without new information being added.

For example, suppose you were told that the temperature in Vilnius on June 11<sup>th</sup>, 2006 was 10° C. The context is that we know the measurement scale, and the time and location of the measurement. As a result this would be information, since the data ("10") is given meaning. Knowledge is the additional meaning or insight derived from any existing understanding the reader has regarding the information. For example, if you had been told the temperature in Vilnius every day for 10 years, you would know that 10° C is unusually cold for June.

In this course we will focus on information (as in databases) rather than knowledge (as in decision-support systems, or expert systems).

Good information has the following 6 characteristics:

<b>Accurate:</b>	Error free
<b>Complete:</b>	Contains all relevant facts
<b>Timely:</b>	Available when needed
<b>Verifiable:</b>	Can be checked for correctness
<b>Economical:</b>	Balance the cost-benefit; cost of collection and maintenance vs benefit to the organization
<b>Relevant:</b>	Important to the decision-maker; appropriate form (numeric/graphical); has appropriate scope, no information overload

These two final characteristics are particularly important when dealing with digital information stored in a database. There is always more information available in a given organisation that *could* be stored in a database. One of the jobs of a database designer is to discern what information is important and what is not, as it relates to the task at hand.

For example, for a college registrar's database, we would need to keep a number of pieces of information regarding students, such as where they live, what courses they've taken and what grades they've received. Other student information, like their driver's license number, their current employment status, marital status, number of children, etc., will exist, but it is important to consider whether keeping this extra information provides any value to the Registrar's office. Storing all these other things merely fills up our database and slows it down without adding value or functionality to our database. It's also very expensive to keep information up-to-date and accurate, so we only want to store the information we really need.

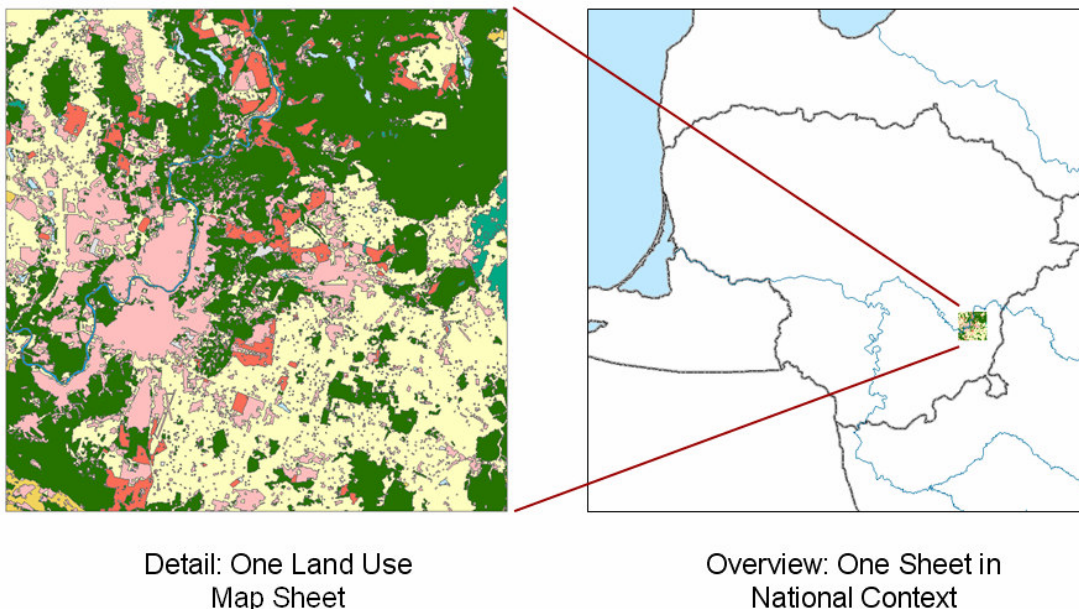
A **Database** is an organized collection of related data that is necessary for some purpose. Data are arranged to facilitate efficient access and alteration. A **Database Management System** (DBMS) is the software that manages the information. A DBMS allows the processes of **defining**, **constructing** and **manipulating** the database.

*Defining* a database is the process of building the empty structure which will hold our data. We will specify things like the data type (e.g., number, string, or date), constraints on possible values (e.g., Age must be greater than zero) and any relationships between things in our database (e.g., which course is taken by a student). *Constructing* the database is the process of filling its structures with data. The DBMS software will read and write values to media (e.g., a computer disk) which it

controls. *Manipulating* the database involves the ability of the DBMS to retrieve information from (**Query**) the database or to update values found there.

A DBMS must provide **efficient**, **convenient** and **safe multiuser** storage of and access to **massive** amounts of **persistent** data.

*Massive* refers to the large size of many databases. Many databases are several gigabytes in size for applications like banking. Databases become even larger if the database stores a history of all transactions. GIS databases in particular are typically very large. For example, consider the Lithuania LTDBK50000 Land Use mapping dataset. It contains polygons which identify areas of similar land use, such as Agricultural Areas, Forest or Built-up Areas. This fairly general 1:50,000 dataset may have several thousand polygons, constituting several megabytes of data per mapsheet. There are 135 map sheets for the country, so the land use for the entire country would be several hundred megabytes in size. More detailed mapping, such as 1:10,000, would create much larger files.



**Figure 1-1 Land use Database Example**

*Persistent* means that the data outlive the software that operates on them. It is often the case that one software application is popular for a few years, after which another takes its place as the market leader. GIS is a good example of this, where some user groups of GIS have changed applications several times over the last 15 years. Each time the GIS software is changed, all the data they hold must be translated, or reformatted, so that it can be used in the new GIS application. This is because the underlying data structure of the different GIS applications is different. Moving data from one software application to another can often be a long and complex process. This is less the case now, since translators for moving data between the major GIS applications are more common, but every time you change data formats there will be a significant amount of work.

*Multiuser* refers to the ability of a database to support many people or programs accessing the same database or even same data simultaneously. Multiuser use of a database requires careful

control. A common example of the need for careful control is the case of simultaneous withdrawals from a bank account.

A database must be *safe* from system failures and malicious users. Most large DBMS applications have a large proportion of their functionality devoted to data backup and recovery. Not only do DBMS manage efficient and regularly-scheduled backups, they also have the ability to manage transactions - we'll talk about this in much more detail later in the course, but essentially everything done to the data has what are called safe points, where the data integrity is preserved and a half-completed transaction can be rolled back to the last safe point if errors are encountered.

For example, if you were withdrawing \$100 from a bank machine, the transaction involves two tasks: \$100 is deducted from your account balance, and you are given \$100 in cash. If something catastrophic happened to the DBMS part way through this transaction (power went out, storage media failed, etc.), it would be quite bad if the money were deducted from your account but you had not yet been given the \$100, or that you were given the money but the \$100 was not yet deducted from your account. It depends on your point of view whether these cases are bad or not, but essentially we do not want a situation where one of those two tasks happens but not the other. In reality, the deduction from your account balance would happen first, and if the transaction failed before you got your money, the DBMS would *roll back* the transaction, recognizing that you had not received your money yet, and putting the \$100 back in your account.

DBMS also must have the ability to manage access to the data in order to protect them from malicious users. Some users will be allowed to change the data, while other users may be able merely to look at it. This will involve managing separate user accounts (usernames and passwords) and privileges (the ability to read or write specific information in the database).

For example, consider a sensitive dataset, such as where fishermen are fishing and the amount harvested in each area. We might establish three levels of access, and we could control which level a person has by using a username and password to validate the user. The three levels might be:

Restricted Access: this access level would be given to perhaps the entire organisation working with this data. People with this level of access might be able to see where fishing was taking place, but not see the rates of harvest or which vessel was fishing in each location.

Read-Only Access: People with this level of access could see all of the data, including vessel and harvest information, but would not be allowed to modify the data. People with this level might include managers or scientists who need to make decisions based on this data (such as quota assignments), but who do not actually maintain the data.

Full Privileges: People with this level of access would be the people actually editing and adding to this dataset. It would likely be a small number of people with very technical backgrounds.

This kind of a structure ensures that only those who need to see the sensitive details can do so, and that only a select few people can actually change the data. A very generalized view, which would not include anything sensitive, would be available to a very large user community. Users

that cannot authenticate themselves with a username and password have no access to the data at all.

If it is difficult for users to access or change the data, it ceases to be efficient. The DBMS should have a clear interface with simple commands to manipulate data. In this way, the data may be *conveniently* accessed. Most database management applications have a graphical user interface (GUI), as you are accustomed to seeing with Windows applications, that uses the mouse and keyboard to interact with the database. They will also have a standard way of interacting with the database using typed commands. This is called the **Structured Query Language**, or SQL, which we will learn about later in the course.

The data within a DBMS must be structured for *efficient* access. DBMS are "tuned" for performance, meaning everything from the way the data tables are structured to the way the files are physically stored on a disk must be optimized to ensure efficient access. DBMS have efficient ways of searching for data as well; they don't, for example, exhaustively search all their files in order to get the balance of one bank account.

### 1.1.2 Database Models

In the past, the first computer applications began by storing all data in files and file folders, much as you might organize your word-processing files today. The shortcomings of this approach quickly became obvious: retrieving information from a series of files might require an intimate knowledge of how and where a given piece of information is stored, and the speed of the retrieval is very slow. Significant programming is necessary to accommodate the specific way you store your files, and security functions are difficult to manage. The need for a structured way of storing and managing information became more of a priority particularly as volumes of data and the complexity of the data increased.

The need for better management of information led to a series of theoretical constructs being developed. Each was developed from its predecessors, and each sought to overcome previous limitations. We will briefly address each conceptual model, and review its strengths and weaknesses.

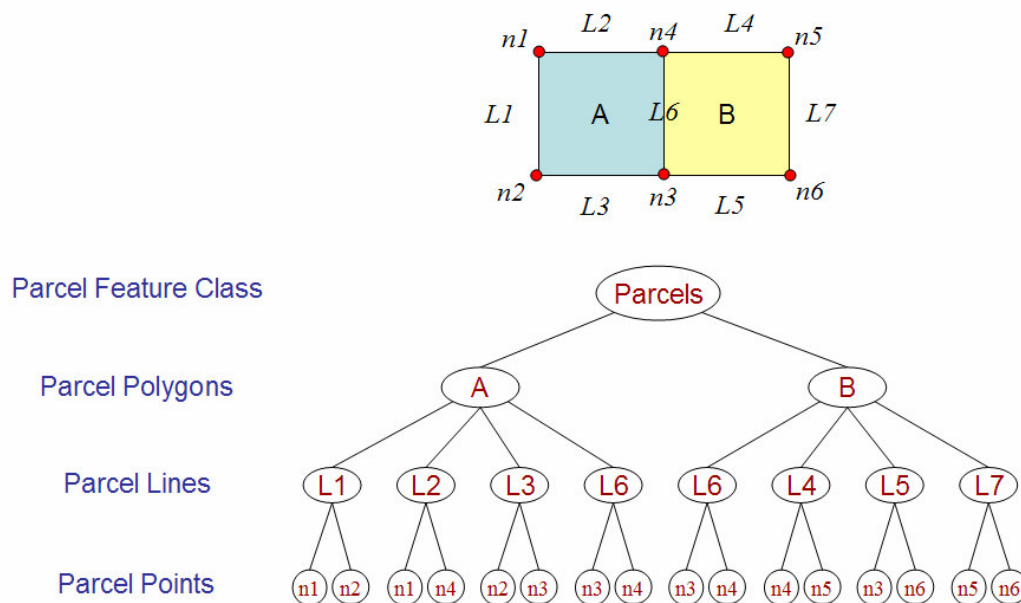
#### Hierarchical Model

The hierarchical database model stores database elements as a type of tree structure. The term *tree* refers to how a hierarchical diagram has the appearance of an upside-down tree; at the top is the root of the tree, along the bottom are the leaves. These terms *root* and *leaf* are used frequently when describing portions of a hierarchical structure. In this type of a database, a set of links connect database elements in what are called parent-child relationships. Each parent may have one or more child records, and this type of relationship is called a one-to-many (abbreviated to 1:M or 1-M) relationship. Examples of hierarchical relationships include:

- Biological Classification: e.g., each genus has many species
- Polygon – Line – Point: e.g., each line has many points

Figure 1-2 below shows an example of a hierarchical structure used to portray a set of land parcels. Note the inverted tree appearance of the diagram. The graphic at the top shows the

features that the database is trying to represent: two land parcel polygons (labeled A and B), seven lines (labeled L1 – L7) and six node, or point features (labeled n1 – n6). The polygons are formed by a series of enclosing lines, and each line is formed by a start and end point. These are the relationships noted by the connecting lines in the tree structure below. Parcel A, for example, is formed by lines L1, L2, L3 and L6. Line 1 connects the points n1 and n2.



**Figure 1-2 Hierarchical Database Model**

The primary advantage of the hierarchical model was that it was the first attempt to use a more structured approach than simply files and file folders to manage large volumes of data. The result is that the DBMS application introduces the ability to manage the security and integrity of the data in an integrated way.

The hierarchical structure also has the advantage of high speed of access to large datasets. For example, to find the point elements which form the perimeter of Parcel A in our example above, a search of this database simply traverses the links from the root to Parcel A, to its child records (L1, L2, L3 and L6), and their connected point features. We need not read the entire right-hand side of this diagram. In an unstructured list of points such a search might involve reading every point element in the database. The hierarchical model can vastly reduce the number of reads necessary to find a given database element.

However there are several disadvantages of the hierarchical model. Of primary concern is that linkages are only possible vertically, not horizontally or diagonally, which means that there is no relationship between different entities at the same level unless they share the same parent. As a result, many elements must be repeated in the database.

In our example above, where polygons share the same line (e.g., L6 above), the line entity must be repeated in the data model, since we cannot have a child record relating to more than one parent (i.e., L6 cannot be related to both polygon A and polygon B). Particularly with the point entities, we must repeat each point every time it is used in different line feature. Although there are only six points in the diagram above, we must store 16 point elements in the hierarchical model.

Also of note with the hierarchical model is its lack of **Structural Independence**. Structural independence exists when changes to the way data is stored have no bearing on the ability of the DBMS software to manage them. With a hierarchical model, changes to the way the data is stored require changes to all the application programs which access the database. As a result, managing the database and applications can be extremely complex and time-consuming.

Lastly, there are many real-world relationships which may not be modeled well with a strict hierarchy using only 1:M relationships. Consider the relationship found in a Library, where there are a set of Books and a set of Borrowers. Each Book may be borrowed by many different people, and a given person may borrow any number of books. This situation does not lend itself well to the hierarchical Parent-Child relationship, and this example is often called a Many-to-Many (abbreviated as M:M or M:N) relationship.

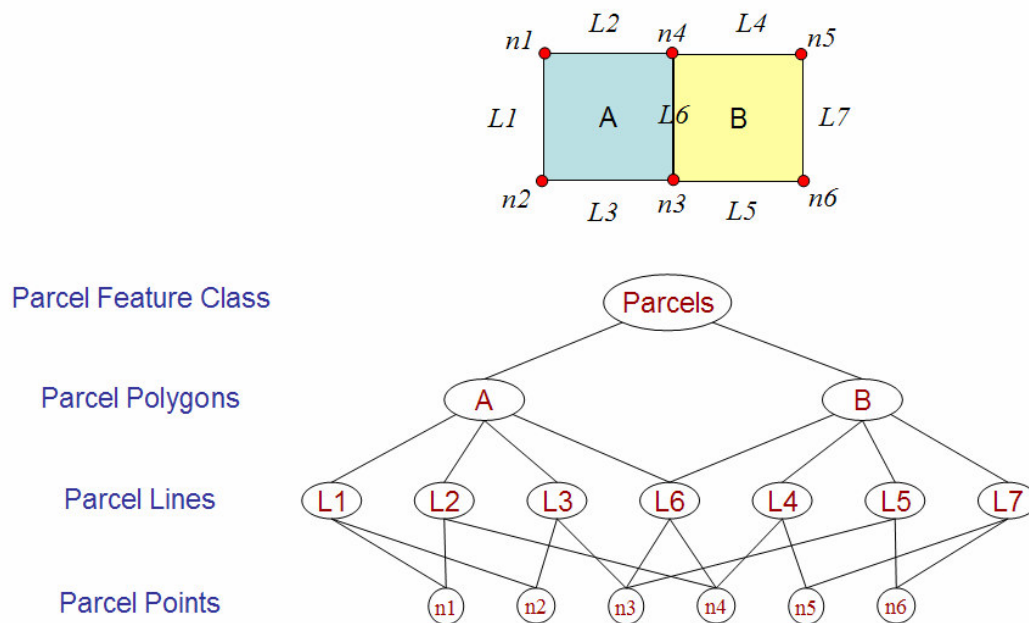
Hierarchical DBMSs were popular from the late 1960s, with the introduction of IBM's Information Management System (IMS) DBMS, through the 1970s and early 1980s.

### Network Model

Some data are more naturally modeled with more than one parent per child, or with relationships directly between child records which do not share a parent. The network model was developed to accommodate this. Unlike hierarchical data structures, network models are not restricted to paths vertically between parent and child - linkages are possible vertically, horizontally, and diagonally within the model.

Figure 1-3 below shows an example of a network implementation of the same set of parcels we modeled using the hierarchical structure.

The most striking difference from the hierarchical structure is that now there are fewer database elements, but far more links between elements. Note how the shared line L6 has two parents; both parcel A and parcel B link to the L6 element. Similarly, the point n3 is referenced by the three line features (L3, L5 and L6) which meet at this point.



**Figure 1-3 Network Database Model**

The advantage of the Network model is that it is conceptually simpler and far more flexible, in that it can effectively model more real-world relationships than the strict hierarchy. It also begins to adopt the idea of conforming to standards. The language used to interact with the database was standardized for all implementations of the network model, improving database administration and portability.

The primary disadvantage of the network model is the complexity of the system, since there are large numbers of links between database elements to manage. To access data elements, users of the database must have a firm understanding of the data structure because access is performed by traversing links within the structure. Also of concern is its lack of structural independence. Like the hierarchical model, changes to the structure require changes to the application programs.

The Conference on Data Systems Languages (CODASYL) formally defined the network model in 1971, but it was largely replaced by the Relational model in the 1980s.

### Relational Model

The basis of the relational model is the **Relation** (table), and it manages links between the many tables in a database. This model is based on the field of mathematics called Relational Algebra. We will study the relational model in detail later in the course. Essentially the relational model represents everything as a series of tables, with rows becoming the data elements. The Relational Database Management System (RDBMS) software must then manage the relationships between tables. This tabular structure is conceptually very easy for people to understand and work with.

Figure 1-4 below shows how our example land parcels might be represented in a relational database.

Table: Polygons

Polygon	Lines
A	L1, L2, L3, L6
B	L6, L4, L7, L5

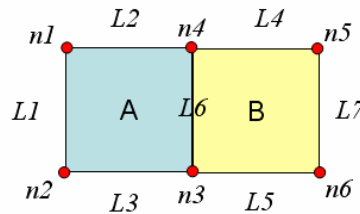


Table: Lines

Line	FromPt	ToPt
L1	n1	n2
L2	n1	n4
L3	n2	n3
L4	n4	n5
L5	n3	n6
L6	n3	n4
L7	n5	n6

Table: Points

Point	XCoord	YCoord
n1	x <sub>1</sub>	y <sub>1</sub>
n2	x <sub>2</sub>	y <sub>2</sub>
n3	x <sub>3</sub>	y <sub>3</sub>
n4	x <sub>4</sub>	y <sub>4</sub>
n5	x <sub>5</sub>	y <sub>5</sub>
n6	x <sub>6</sub>	y <sub>6</sub>

Figure 1-4 Relational Model

The primary advantage of the relational model is its achievement of structural independence. Here, changes to the internal data structure have no effect on the DBMS's data access. With the relational model the user no longer needs to know the physical details of the database structure. The relational model also has the advantages of a conceptually simple structure (with corresponding ease of design and implementation), and of the ability to perform ad hoc queries with the database using an industry-standard **Structured Query Language (SQL)**.

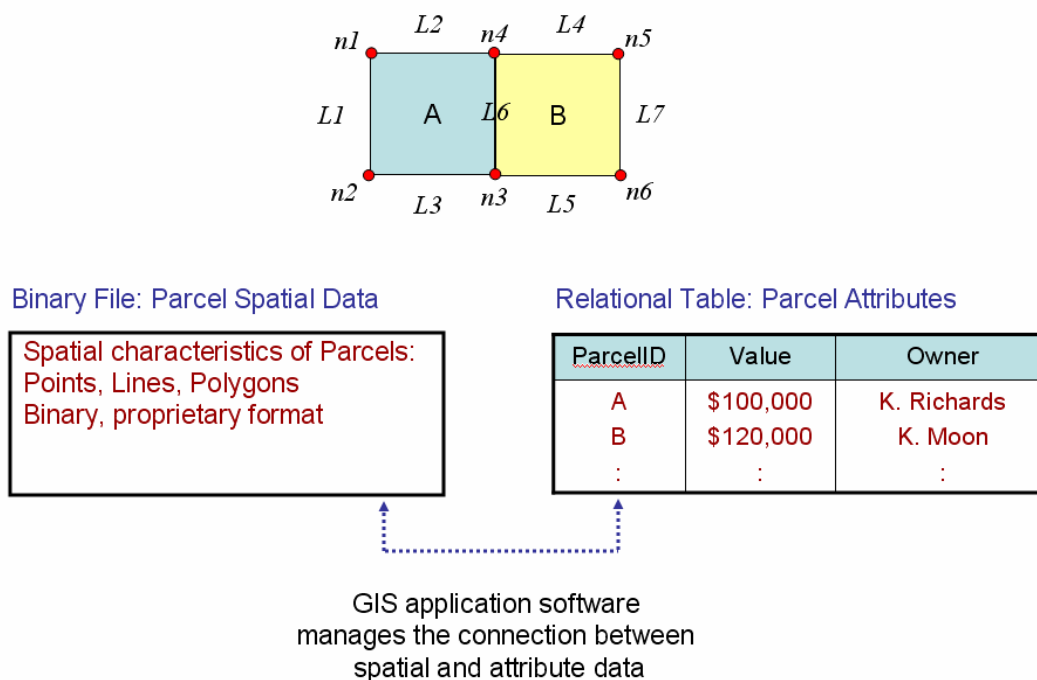
The conceptual simplicity of the relational model comes at a price. One of the major disadvantages of the relational model is the hardware and software overhead involved. Much of the complexity of managing relationships between elements is hidden from the user, and as a result, the Relational Database Management System (RDBMS) must perform significant work to access and manipulate the data. In fact, when the relational model was proposed by E.F. Codd in 1970, computers lacked the processing power to actually implement the model (Rob and Coronel, 2000). Today, with the increases in computer processing power, there are many effective relational implementations, but relational databases are still considered slower than other database systems.

The relational structure is a popular model for GIS. For example, the following relational database applications are widely used:

- INFO in ARC/INFO;
- MS Access for ArcMap;
- dBASE III for several PC-based GIS;
- ORACLE, SQL Server for several GIS uses.

## Georelational Structure

While several current GIS applications use an RDBMS to manage all their data (e.g., ESRI geodatabase structure), it was very common in the past to store the spatial and attribute data for a given feature class in separate structures. Spatial data (points, lines and polygons) were stored in a proprietary binary format, and feature attributes (such as a parcel owner, or a pipe diameter) were stored in a standard RDBMS format such as dBASE or INFO. The GIS software application would then manage the connection between these two file structures to ensure the correct attributes are applied to the correct spatial features. This method of separating data into two formats is called a **Georelational Structure**, or a **Dual Architecture**. Figure 1-5 below shows how our parcel data might be structured in the georelational structure.



**Figure 1-5 Georelational Structure**

The georelational structure is not a database model as we have discussed with respect to the hierarchical, network and relational models, but rather a structure which adapts the relational model to a specific purpose.

While current GIS development has seen the emergence of integrated geodatabases containing all necessary data, many dual architecture implementations are still in common use today. Examples of GIS using a dual architecture are: Arc/INFO, MapInfo, ArcView and MGE.

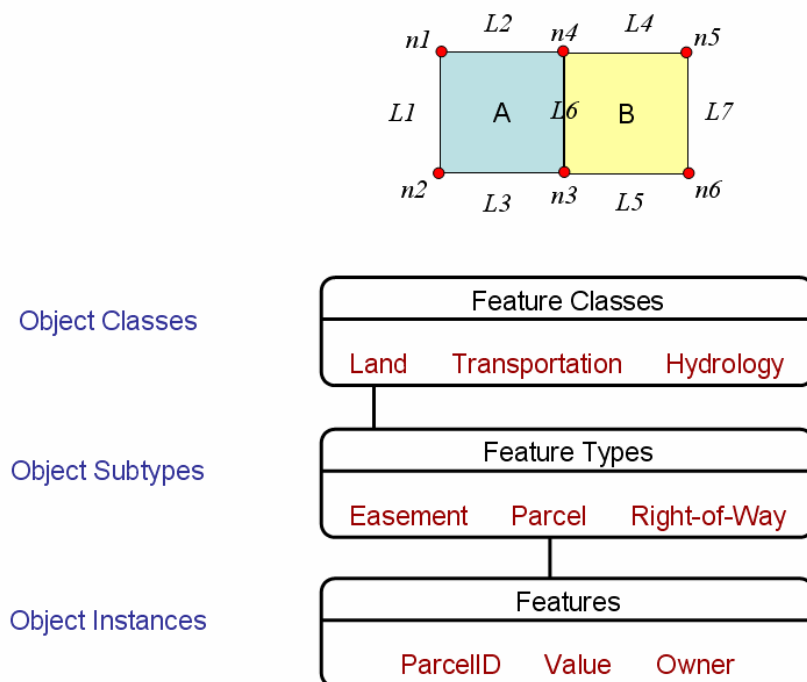
## Object-Oriented Model

The desire to more effectively represent the things and relationships found in the real world was driving development of a new, more flexible and complex database model. The Semantic Data Model (SDM) was developed in 1981, and was the first to include the notion of an **Object**. The

word *semantic* indicates meaning; the semantic model sought to include a broader understanding of objects, including their behaviors and relationships together in one structure.

Current Object-Oriented Database Models (OODBM), which have developed from the SDM foundation, include the basic database entities and their attributes (as is also present in the relational model), but also include relationships between attributes *within* the object, and also include all operations that can be performed by or with each object, such as changing its values, finding a specific value, or printing the value. This encapsulation of relationships, data and operations into a single object makes the object self-contained. It makes the object a building block which can be shared, re-used or built upon to create more complex objects.

Figure 1-6 shows how our parcel data might look when stored in an object-oriented model.



**Figure 1-6 Object-Oriented Model**

The basic idea is that users should not have to deal with rows and columns, but with objects and operations on those objects that more closely resemble real-world things. For example, you use the *Hire* operation of an employee object rather than insert records into an Employee table, establish links to the Department table, etc. Using Object-Oriented (OO) structures we embed “intelligence” into complex objects – very relevant to geographic objects and relationships which can be quite complex.

Like the relational model, the object-oriented model preserves structural integrity, so that data structure changes may be made without alteration to software or retrieval procedures. Its major advantage, though, is its ability to add semantic content and give the data greater meaning.

The greatest disadvantage of the OO model is the lack of standards, the most significant being a standard data access method, such as SQL for the relational model. The data access methods for OO structures can be complex, and have been compared to those methods used in hierarchical or network models. The complexity of OO models also adds significant processing overhead to

database interactions, making transactions in some implementations far slower than a relational structure would be.

The network and hierarchical models are still in use, but certainly not mainstream at present. These models may be considered technically obsolete (Date, 2000). The vast majority of database systems in use now are relational, and to a large degree this massive investment in relational technology has made a move to object-oriented systems more difficult. Many current relational implementations have added OO concepts, and many of these concepts were added to standard SQL in 1999.

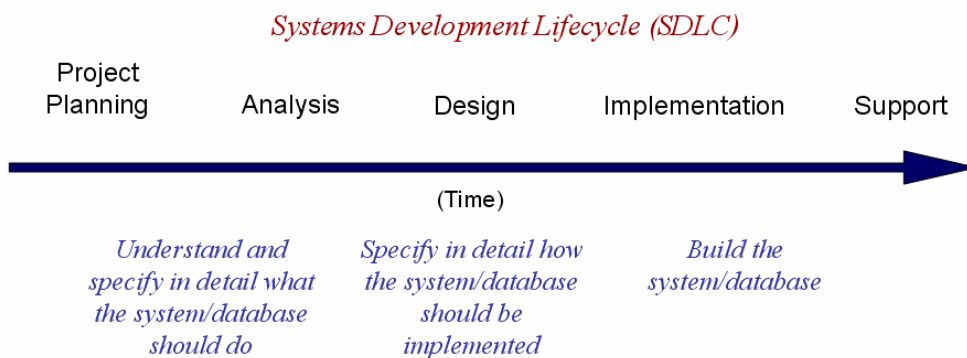
These relational implementations with additions of object-oriented capabilities (such as defining custom object types) are sometimes referred to as an **Object/Relational Model**. However, in reality these are most often a relational structure at the data level, and have object-oriented capabilities which are added at the application level of the RDBMS software. Most larger RDBMS programs (e.g., Oracle, DB2, Informix, SQL Server) make claims to support OO technology to varying degrees. It is expected that this merging of relational and object-oriented technologies will continue in the near future (Rob & Coronel, 2000).

## 1.2 Data Modeling

### 1.2.1 Introduction

In this topic, we will explore the design aspect of databases. Design is the process of modeling, or abstracting, the real world into a specific structure which best suits our database purpose. The purpose of a database is important to how data will be structured, and a detailed understanding of how the database will be used is required to make a useful and successful database model.

When we develop either software or database applications, we typically follow a process flow called the **Systems Development Lifecycle (SDLC)**. Early stages revolve around defining what the system or database needs to do. The design phase is where we document how the system will accomplish these requirements. Here we'll document everything from hardware and software requirements, software or database specifications, training requirements and data migration plans. Implementation is where we create the database, install the hardware and software, train staff and move existing data into the new structures. Support refers to ongoing activities like system enhancements, fixing bugs, and user support or training. Figure 1-7 shows this lifecycle graphically.

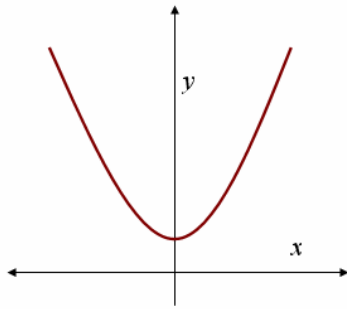


**Figure 1-7 Systems Development Lifecycle**

When we implement projects with a specific GIS focus, we need to add one more step, or rather we need to consider one more thing during the design phase: how database entities will be represented in a spatial sense. We need to look at what things in our database will reside as point, line or polygon feature classes, what scale will be appropriate for the capture of our data, and what sorts of spatial relationships might exist.

### 1.2.2 Data Models

A **model** is an abstraction describing a system or subset of a system. For example, a graph such as shown in Figure 1-8 might describe the travel of a projectile, or the behaviour of an economic activity. Obviously, in reality there will be subtleties or imperfections which the model cannot describe, but it can be viewed as a simple representation of what happens in the world of physics or economics.



**Figure 1-8 Example Model**

A **notation** is a graphical or textual set of rules for representing an abstraction. For example, the graph above can be mathematically symbolized by the equation:

$$y = x^2 + 2$$

The set of notation rules for mathematics tells us that  $x^2$  means that the value  $x$  should be raised to the power 2. Additionally, the symbol  $+$  indicates that 2 should be added to the value  $x^2$ . The rules of the notation allow us to interpret the mathematical symbols used in the model.

Systems are complex and hard to understand. Models can make certain aspects more clearly visible than in the real system. For example, it might be difficult to understanding everything that happens in a very complex database system. However, documenting the data structure as a diagram enables the reader to isolate certain aspects and study them in detail. A database model might enable a designer to:

- Express your ideas and communicate with other people
- Reason about the system:
  - detect errors
  - predict qualities
- Computer Aided Software Engineering (CASE) software may even generate parts of the real system:
  - programming code to manage behaviour
  - data structures (tables and fields) which will form our completed database

The **Logical Model** is a high-level abstraction of what you will be doing. In database terms, that means it's a generalized depiction of what you will be storing in the database, and how the things that are stored relate to other things and behave in the finished system or database. For example, the diagram in Figure 1-9 shows in very general terms how two things in a database might relate to one another. This database might store information about students and the dormitories they live in. We will discuss the notation used in this diagram in detail later in this module topic.



**Figure 1-9 Logical Model Example**

The **Physical Model** is the “nuts-and-bolts” representation in its detailed form. The physical model indicates how the data will be physically stored on media (file types, physical disk drives), and how the data are to be structured in terms of actual rows and columns in tables. For example, data definition language (DDL) can be used to describe in very specific terms how a new table will be created in a DBMS. We will discuss elements of this language, such as data types and primary keys in later lectures.

```
CREATE TABLE EMPLOYEE (  
    EMP_ID NUMBER(8),  
    LNAME VARCHAR2(30),  
    FNAME VARCHAR2(15),  
    HIRE_DT DATE,  
    SALARY NUMBER(8,2)  
);  
  
ALTER TABLE EMPLOYEE (  
    CONSTRAINT EMP_PK  
    PRIMARY KEY (EMP_ID)  
);
```

This lecture topic will concentrate on the higher-level logical modeling, and we will discuss the process of moving from a logical model to a physical model in later modules of this course.

### 1.2.3 Class Diagrams

The **Unified Modeling Language** (UML) is a notation used for specifying software and databases. As a result, it includes diagrams which describe a variety of different things, like how a user interacts with the software. Many of these things don’t specifically affect how data is structured.

Prior to 1997, a common notation existed called the Entity-Relationship Diagram (ERD). You will still commonly see the terminology used in that notation, so we will include the UML term and the ERD term wherever possible. You may use either set of terms when you discuss this material in assignments or exams.

UML includes 12 diagram types categorized into the following 3 areas:

Structural	Behavioural	Model Management
1. <b>Class Diagram</b>	5. Use Case Diagram	10. Packages
2. Object Diagram	6. Sequence Diagram	11. Subsystems
3. Component Diagram	7. Activity Diagram	12. Model
4. Deployment Diagram	8. Collaboration Diagram	
	9. Statechart Diagram	

We will concentrate on **Class Diagrams** to describe how to structure data. Class Diagrams delineate the database classes and the relationships which will be present in the completed system. A generalized class diagram is usually the starting point of an analysis phase.

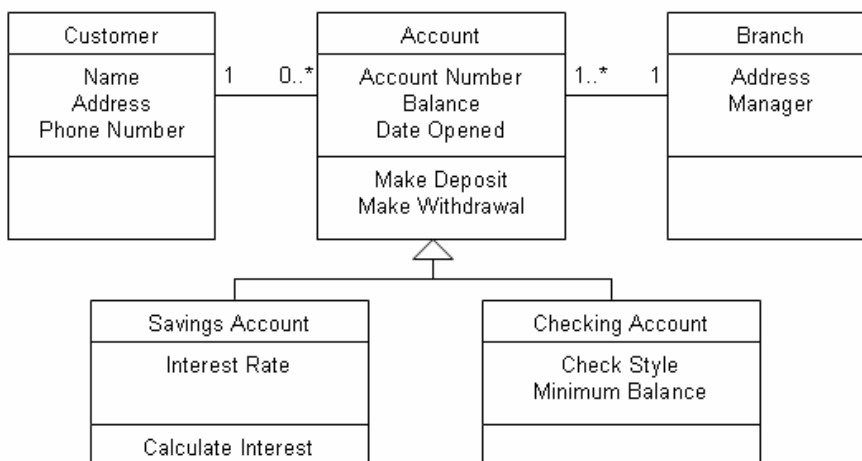
An **Object** is a concept, abstraction or thing that has meaning for an application. For example:

- John Smith, an employee
- A tree that needs to be planted
- A block of land to be reforested

**Classes** are groups of objects that share common characteristics, or attributes. An object is an **instance** of an object class. For example:

- The employee John Smith is an object *instance*, All Employees together are the object *class*

Class diagrams will provide us with the notation conventions to specify Objects and their attributes, as well as relationships between object classes. Figure 1-10 is an example of a UML Class Diagram. We'll look at this diagram in detail, discussing what the various parts of the notation do, and what the overall diagram tells us.

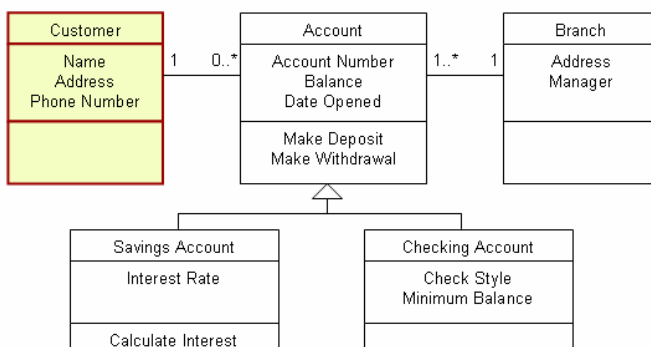


**Figure 1-10 Example Class Diagram**

In very general terms, this is a simple Class Diagram for a banking database. There are *Customers*, *Bank Accounts*, and *Bank Branches*. This also says that there are two different types of accounts: checking and savings. It says that there is a relationship between customers and their bank accounts, and there is a relationship between bank branches and the accounts held there.

## Classes

The central things in a Class diagram notation are, not surprisingly, the Object Classes. These are represented as Boxes in the diagram, and the Customer class is highlighted below in Figure 1-11.



**Figure 1-11 Classes in a Class Diagram**

Class boxes are divided into three sections:

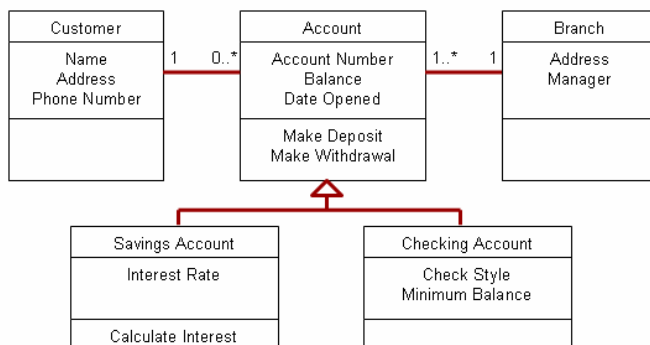
- the top section holds the name of the class;
- the middle section holds the **Properties**, or attributes of that object; and
- the bottom section holds the **Methods**, or actions either performed by or on the Class.

If you've worked with Object-Oriented programming before, you'll be familiar with the terminology for Object, Property and Method. In ERD notation, the parallel concept to an object class is called an **Entity**. Just like Object Classes are the central things in a Class Diagram, the Entity is the central thing in an Entity-Relationship diagram.

In GIS, the term **Attribute** should be a familiar one. In a physical database, these correspond to columns in a feature attribute table. We will for the most part avoid discussing methods. In general, methods describe "behaviour" of databases, which really translates into things that programmers make the database do. As we start using the ESRI geodatabase, we'll find we can make use of behaviour – methods – that programmers have already built into the database.

## Associations

Associations are the relationships, or links, between object classes in the model. In our banking example, we show an association between a bank account and the branch it resides in, and a relationship between a bank account and the customer holding that account. Associations are indicated in a Class Diagram with a line between the classes. Figure 1-12 shows the banking diagram with the associations highlighted.

**Figure 1-12 Associations in a Class Diagram**

This diagram shows the association between Customers and their Bank Accounts, and Accounts and the Bank Branch which holds the account. There is also a special relationship between Account and the two types of Bank Accounts (Savings and Checking). We will discuss this special association shortly.

Associations are often named. For example, one might place the name "Owns" along the association between Customers and Accounts to show that a Customer *Owns* an Account. These named associations help us read and understand the diagram.

In ERD notation, the connection between things is called a **Relationship**, in UML, it's called an **Association**.

Associations can be of several types. The simplest and most common is noted by drawing a line between two, as shown in the association between a Customer and his or her Account. The three more complex association types are called:

- Aggregation
- Composition
- Inheritance

**Aggregation** can be described as being a special association which represent the *part-whole* relationship. In this situation, the whole object is composed of a set of parts, parts can join and leave the whole, and can be part of more than one whole. The idea here is that one Class is the sum of several parts in the other class. For example, a department might be considered to be little more than the employees who work there. An aggregation association is indicated in a class diagram using a hollow diamond shape placed on the association line. The diamond is placed on the side nearest the “whole” class. Figure 1-13 shows an example of such an association, with the department being the “whole”, and the employee being the “part”.



**Figure 1-13 Example of an Aggregation Association**

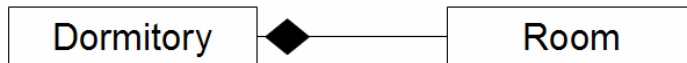
This is an unrestricted relationship, in a sense, because while the department is the whole and the employees are the parts, a given employee can move from one department to another, can conceptually exist on its own, or may be associated with more than one department.

Car-Wheel is a reasonable example of an aggregation relationship. Cars are really no more than the sum of all the parts. A wheel is a part of the whole object (the car), but it is common to pull wheels off one car and put them on another. In addition, it might not be unusual to have a wheel existing without being attached to a car (snow tires in your garage, perhaps).

**Composition** is a stronger relationship than aggregation, but it is a very similar concept. The distinction here is that while we're still talking about the part/whole association, the parts are more tightly bound to the whole. It is sometimes called a “has-a” relationship. The language there is a little ambiguous, but there are a few characteristics of a composition relationship which make it more rigorous than an aggregation association.

The “part” side of the part/whole idea, in composition, cannot exist on its own. In addition, we can think of the “part” objects as being created with the “whole” object, never being able to move to a different “whole” object, and destroyed with the “whole” object. Setting aside for a moment the idea of a brain transplant, we can think of a brain as a part of a body, which is created with the body, and stays with that body until death. A body is really nothing more than the sum of its parts, but the parts are bound very tightly to a given body.

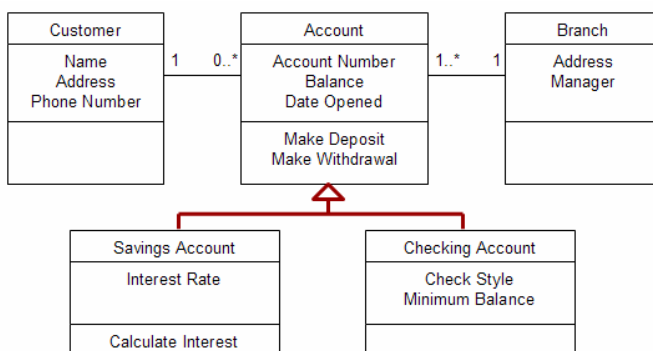
A composition association is indicated in a class diagram using a black diamond placed on the association line, and as with an aggregation association, the diamond is placed closest to the “whole” object. Figure 1-14 shows an example of a composition association.



**Figure 1-14 Example of a Composition Association**

This relationship might indicate a series of rooms and the dormitory buildings in which the rooms exist. The key here is that there really can’t be a notion of a “room” in the absence of the building it physically sits in. In addition, you can’t pull a room out of one building and put it in another. Not in practical terms, anyway.

The **Inheritance** association is a means of classifying things. There might exist several *subclasses*, which refine the definition of a *superclass*. Subclass objects *inherit* all properties and methods of the superclass, plus they may have additional properties of their own. They are indicated on in a class diagram by placing a hollow triangle along the association line, nearest to and pointing at, the superclass. For example, consider our banking diagram. Figure 1-15 shows the banking diagram with the inheritance association highlighted.



**Figure 1-15 Inheritance in a Class Diagram**

This diagram indicates that Accounts can be two types: Savings and Checking. The superclass in this case is the Account, the subclasses are the Savings and Checking accounts. The subclasses, such as a checking account, share all the characteristics of the “account” superclass, meaning they have account numbers, a balance and a date opened. They also, however, can have other characteristics which are unique to checking accounts, like the minimum monthly balance.

Bird-Swan is another example of an inheritance relationship. The biological classification scheme (kingdom, phylum, etc.) illustrates how each level inherits the characteristics of the level above, but might have some new characteristics. In this example, a Swan has the characteristics of all birds (wings, beak, etc.), but also has the characteristics that it is very large, white and has webbed feet.

You can think of the subclass-superclass, or inheritance relationship, to be the equivalent of the “is-a” relationship. A checking account “is-a” bank account.

Any time you want to have a more detailed classification, you can use the inheritance relationship and two or more subclasses. In a GIS, for example, we might have a superclass of Water Pipe,

with three subclasses of Concrete Pipe, Steel Pipe and Ceramic Pipe, each with perhaps their own attributes.

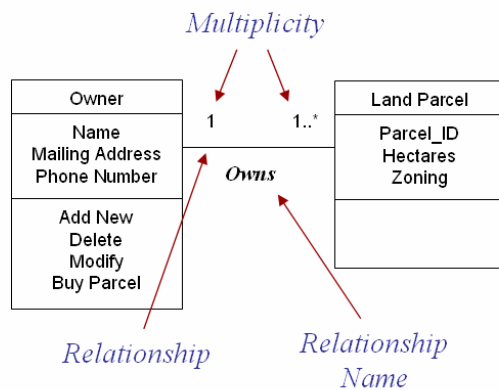
Deciding if an association is really one of the three complex associations can be tricky. In general, these are fairly rare occurrences, and the most common association is the one where there is a simple line drawn between classes. This means there is a relationship there, but it is not one of these complex relationship types.

The terminology is-a, part-of and has-a may help you determine which relationship type is applicable.

- Aggregation = “part-of”
- Inheritance = “is-a”
- Composition = “has-a”

## Multiplicity

Whenever we show an association between classes, we need to indicate how many objects in one class are related to an object in another class. We do this with both a maximum and a minimum number. Figure 1-16 shows an example of an association between land owners and the parcels of land they own.



**Figure 1-16 Relationship Symbology in the Class Diagram**

The connection between *Owners* and *Land Parcels* is explicitly stated in the diagram using a line which connects the two classes. A meaningful name “Owns” unambiguously tells us the nature of that relationship. We know that this line represents the fact that a *Land Parcel* is “owned” by an *Owner*.

The complexity lies in the Multiplicity notation. The possible values for a multiplicity notation are presented below in Table 1-3. Multiplicities should be shown for all associations **except Inheritance** associations (note in Figure 1-15 that there are no multiplicities in the inheritance association).

**Table 1-3 Class Diagram Multiplicity Notations**

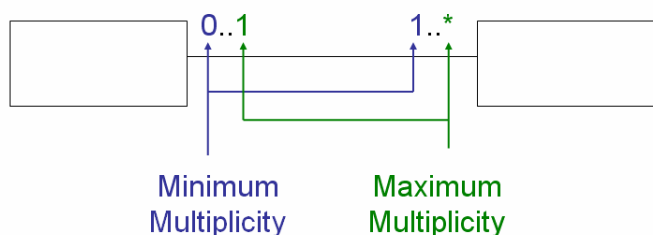
Notation	Alternative Short Form	Description
1..1	1	One and only one
0..1		Zero or one
0..*	*	Zero to any positive integer; Zero or more
1..*		One to any positive integer; One or more
n..m		Any positive integer from n to m; More than one

The multiplicity of an association is read in two directions, which is why there are two notations on the association line between *Owner* and *Land Parcel* in Figure 1-16. The association has the notation “1” on the left side, and “1..\*” on the right side. If we read this relationship from left-to-right, we include the right-hand multiplicity (the “destination” multiplicity). Translating the 1..\* notation to the English description in Table 1-3, the association reads “*Owners own one or more Land Parcels*”. This explicitly states two things: owners have to own at least one land parcel, and owners may own more than one land parcel. The former restriction might be present simply

because there is no point storing an Owner in our database if they do not own anything. The latter restriction accommodates owners of several land parcels. Reading right-to-left, we include the left-hand multiplicity notation (“1”), and it would read: “*Land Parcels are owned by one and only one owner*”. Note that the abbreviated notation “1” is the same as “1..1”. Again, this tells us two things: A land parcel *must* have an owner, and a land parcel cannot be owned by more than one person. This is where the analysis of the **Business Rules**, or the way the organization works or processes data come into play. If the agency had a need to store owners even after they sell their last parcel of land (i.e., they no longer own any land), then the parcel (right) side multiplicity cannot be 1..\*, since 1..\* indicates that owners must own at least one parcel. We must change this notation to read 0..\* to accommodate the possibility that owners may own zero land parcels.

Similarly, if it were possible for parcels to be owned by several people, we need to make another change to the diagram. The owner (left) side notation can no longer read “1”. The one tells that each parcel is owned by one and only one owner, which is no longer true. We must change the multiplicity to 1..\*, meaning that parcels still must have at least one owner, but more than one owner is permitted. One can see that very small changes to the multiplicities make a big difference to the types of situations which will be permitted in the final database.

You may have noticed that in both cases (reading left-to-right, then right-to-left), these notations told us two things: the minimum number of related instances, and the maximum number of related instances. These are called the **Minimum Multiplicity** and the **Maximum Multiplicity**, respectively. The minimum multiplicity is the number of the left of the dots, and the maximum multiplicity is on the right of the dots. Figure 1-17 shows the minimum and maximum multiplicities.



**Figure 1-17 Minimum and Maximum Multiplicity in the Class Diagram**

The minimum multiplicity can be thought of as a mandatory/optional flag. A minimum multiplicity of zero means that an object *doesn't have* to have a relationship with the other class. A minimum multiplicity of 1 means an object *must* have a relationship with at least one instance in the other class.

The Maximum Multiplicity can be read to determine the broad categorization of the relationship. The three types of relationships encountered in data modeling are:

- **1:1** One to One
- **1:M** One to Many – can also have M:1, but it's functionally the same
- **N:M** Many to Many – may also be denoted as M:M or M:N

In the case presented in Figure 1-17, the two maximum cardinalities are 1 and \*, so we read this to be a 1:M relationship.

At this very generalized level of abstraction, we don't use any other values than 0, 1 and \* for our multiplicities. Some students have asked how we represent the situation where we can relate one instance in one class to from 1 to 5 instances in the other class. This type of restriction is getting quite detailed – we'd handle this when we implement the database, and use what is called a *constraint* to force the maximum multiplicity (number of related records) to be five. At this level of abstraction, however, we are forced to state the multiplicity as 1..\*, rather than 1..5. Remember, at this point we are creating a Logical Data Model, and trying only to describe the categories of information stored and the relationships between categories.

When we begin a detailed design of the data model, we will produce a Physical Model, which defines how the logical model translates into tables, rows and columns in the completed database. We will discuss physical models in the following module of this course.

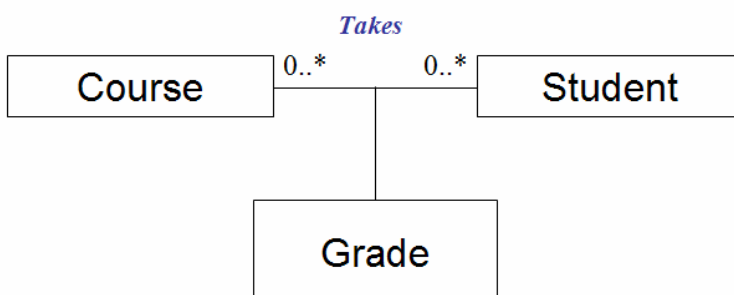
### Association Classes

There are some cases in a data model where we wish to attach properties, or attributes, to the relationship itself. This situation is where the attribute should come into play only where an association between two instances occurs. We need to place the attribute on the relationship because putting the attribute in either of the classes would cause problems.

For example, consider the relationship between a student, a course they study, and the grade they receive at the end of the course. If *Grade* is an attribute of a *Course* object class, then every instance, like GII-01 or GII-05, has a single grade assigned to it. In practical terms, this means that everyone who takes GII-05 gets the same grade.

If we place *Grade* in a *Student* class, it means that each student instance gets a single grade assigned to him/her. In practical terms this means that each student gets the same grade for every course they take.

Instead, we want the situation where each time a student takes (or is associated with) a course, a grade is assigned. To do this, we create an association class, and indicate it on the class diagram by dropping a vertical line from the association line, and adding a box with the relevant properties there. In this way, we can attach *Grade* directly to the association, and each time a student takes a course, a new grade is assigned. Figure 1-18 shows how the relationship between Course and Student would appear.



**Figure 1-18 Association Class Example**

The best way to clearly understand class diagrams is to review diagrams and to make your own diagrams. We will review here several examples and you will be given the opportunity to make your own during the module assignment.

### Example Diagrams:



This first example illustrates the relationship between an employee and an automobile supplied by the employer for his/her use. Perhaps the employees are involved in sales, and they require a vehicle to visit their customers.

Reading from left-to-right, this diagram says:

*“An employee is assigned 1 (and only 1) automobile.”*

Remember that “1” is a short form of “1..1”, so it means both a minimum and maximum of one automobile. The minimum multiplicity of 1 tells us that this is a mandatory relationship, and employees must be assigned an automobile. This model cannot accommodate storing employees not involved in the sales process such as accountants or administrators who do not have vehicles assigned to them. The maximum multiplicity tells us that employees cannot be assigned more than one vehicle. Everyone who works here gets a car – no exceptions!

Reading right-to-left, this diagram says:

*“An automobile is assigned to 0 or 1 employees.”*

The minimum multiplicity of zero tells us that this is an optional relationship, and vehicles do not have to be assigned to an employee (we might have cars owned by the company that are not assigned to anyone). The maximum multiplicity of 1 tells us that if assigned, an automobile is only assigned to one employee (i.e., automobiles cannot be shared). The implication is that we have at least as many cars as employees in our company, perhaps more.

Reading just the maximum multiplicities, we would say that this association is a one-to-one (1:1) relationship.



Our second example depicts the relationship between a series of dormitory buildings and the students who reside in them.

Reading left-to-right, this diagram says:

*“A dormitory houses zero or more students.”*

Again, the minimum multiplicity of zero tells us that dormitories do not have to house students (i.e., a dormitory can be empty). The maximum multiplicity tells us that any number of students can be housed in a given dormitory. As we have discussed, our logical model does not address the true maximum of the building, if perhaps at most 100 students can be accommodated at one time. The practical limits on the number of students would be implemented using a database constraint when the database is built. For our purposes, we will create generalized diagrams only, and any maximum multiplicity  $> 1$  will be indicated by the  $*$  notation.

Reading right-to-left, this diagram says:

*“A student is housed in 0 or 1 dormitory.”*

Again, the minimum multiplicity of zero tells us that students do not have to live in a dormitory. Many students choose to live in apartments or homes off-campus. The maximum multiplicity of 1 tells us that if a student lives in a dormitory, they can only live in one (i.e., they cannot be assigned two rooms at once).

Reading the maximum multiplicities, we would say that dormitory-student is a one-to-many (1:M) association.



Our third example indicates the relationship between students and clubs. Clubs might be university sporting or hobby organizations such as a “Swim Club”, or a “Photography Club” where students with similar interests can congregate.

Reading from left-to-right, this says:

*“A student may be a member of 0 or more clubs.”*

The minimum of zero tells us this is an optional association, and students are not forced to join a club. The maximum of  $*$  means that students may join as many clubs as they wish.

Reading right-to-left, we read:

*“A club may have one or more student members.”*

The minimum of 1 tells us that this is a mandatory association, and a club has to have at least one student to exist. The maximum of  $*$  indicates that there is no limit to how many students may join a given club.

Reading the maximum multiplicities, we would describe this as a many-to-many (M:M) association.

You can see that a very simple diagram can tell us quite a bit about the relationship shown and the types of situations we can accommodate in our data model. Very subtle changes to the model can make fairly significant changes to the situations our data model can handle. For example, Figure 1-19 shows the Dormitory-Student association from our second example above, except the left-most multiplicity has been changed from 0..1 to simply 1. Now our model can no longer handle students living off-campus. All students must now live in a dormitory.



**Figure 1-19 Modified Dormitory-Student Association**

If you create data models as part of your job, you will frequently be asked to create a data model based on conversations or a written description of what a database system must be able to store and the functions it must perform. You will have to determine the associations and multiplicities based on the Business Rules, or the constraints that apply to an organization and dictate how they do business. At some point you will have to ask, for example, whether students are allowed to live off-campus, or whether they must all live in a dormitory.

There will be many cases during an assignment or exam for this course where you will not be given an in-depth description of the business rules for a data model. In such a case, you must rely on your understanding of how the relationship works to create your model. The key to this process is to state your assumptions, so that the person marking your assignment or exam understands how you see the relationship. Remember: there may be no single, correct answer! If you were asked to create the Dormitory-Student association we have been discussing, but are not told explicitly whether students can live off-campus, or whether we are only storing students living in dormitories, you must choose one implementation and document why you did so, or the assumptions you made in doing so. You might have a thought process similar to the following:

*"I'm assuming that the purpose of this database is to manage the dormitories, perhaps to handle booking of new students into vacant rooms, billing student residents, etc. There is no point in storing a student who lives in an apartment off-campus, since they do not affect what we are trying to do."*

In such a case, you would create a diagram like that in Figure 1-19, and write down your assumptions below it. Your assumptions might look as simple as this:

**Assumptions:**

- *Database purpose is to manage dormitories, so all students in this model must live in a dormitory.*
- *Dormitories can be empty, to accommodate summer term when there are very few students.*

As another example, let's create a class diagram for the association between two classes: Advisor and Student. This situation describes an academic advising situation, where students work with an advisor to decide things such as which courses to take or what degree they would like to achieve. In the absence of any further information, we don't really know how this association works. For

example: Does a student have to have an advisor? Does a student simply “drop in” and speak with any available advisor, or do they make appointments to meet with one specific advisor who is familiar with their situation? We must make assumptions about how we understand this situation, or how we think it is most likely to function. In this way, the person marking your work will understand why you might have a slightly different answer than another student.

Your answer to the Advisor-Student association might look like the following:



*Assumptions:*

- *An advisor can advise many students, but must advise at least one. If they don't advise anyone, they aren't really an advisor!*
- *Students don't have to use the services of an advisor, but if they do, they can seek any available advisor.*

### Practice Exercises

Try the following class diagrams on your own. In all cases, state any assumptions which influenced your solution. Answers are found in the section immediately following these exercises.

1. Make a model to support the classes Painting and Gallery. We're trying to show the situation where an art gallery exhibits (displays) paintings.
2. Make a model to support the classes Building and Apartment, with the association indicating which apartments are physically in which buildings.
3. Model the classes Book, Publisher and Author. There may not be associations between all classes (as you see fit).
4. Model the classes Vehicle, Car & Motorcycle. Hint: this one is unusual.
5. Make a Class Diagram for the college Registrar's office. The office maintains data about each course, including the instructor, and the time and place of the course meetings and the students in each class, including the student name, number and dormitory. Each time a student enrolls in a class, a grade is recorded.
6. Make a Class Diagram for the Library. We need to keep track of items lent, including videos and books. We also need to maintain information on students and which items they borrow, including when an item is checked out and returned. Add a small number of relevant attributes. State any assumptions that need to be made.

Solutions:

There will be many variations on the solutions, depending upon the student's understanding of the situation being modeled. The important thing is that students match the notation (e.g., multiplicity) with the assumptions they have.

Students often find it frustrating that there is not one "right" answer, especially here where we do not have all the details. Normally you wouldn't use just your knowledge of the subject area – you would discuss with someone who knows the business before making design choices. For the purposes of our course, in an exam, for example, we don't have this in-depth knowledge of the subject area. In that case, STATE YOUR ASSUMPTIONS.

Below are presented the most likely solutions, but other reasonable answers are possible.

- 1. Make a model to support the classes Painting and Gallery. We're trying to show the situation where an art gallery exhibits (displays) paintings.**

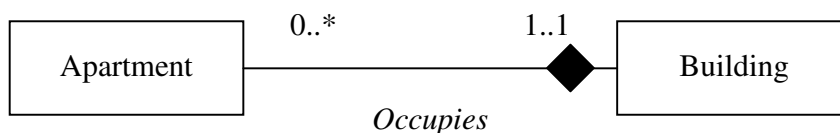
**Assumptions:**

- This models a snapshot in time – a painting cannot be exhibited in two galleries at the same time.
- Paintings do not have to be exhibited.
- A gallery could display sculptures or other visual arts projects, so they do not have to exhibit paintings.

**Notes:**

- Multiplicities might be 1..\* on the left and 0..\*, 1..1 or 1..\* on the right, depending upon the assumptions stated by students.

- 2. Make a model to support the classes Building and Apartment, with the association indicating which apartments are physically in which buildings.**

**Assumptions:**

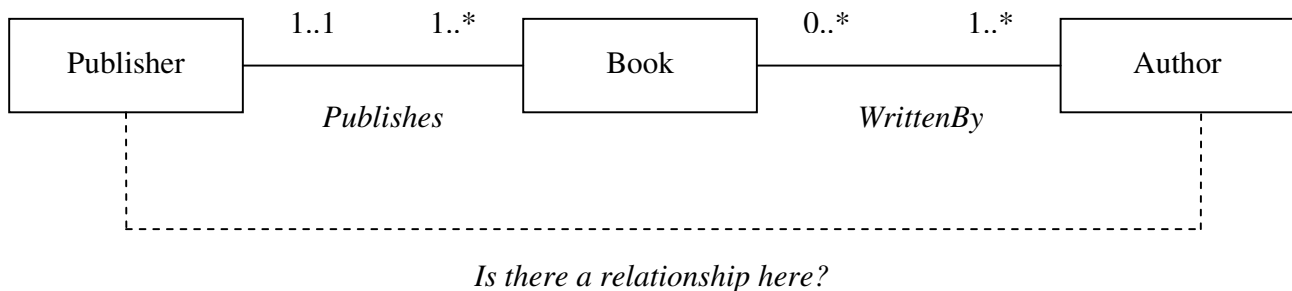
- A building might not be an apartment building – so can have zero apartments.

**Notes:**

- There's really not much variability in this one. Perhaps students might go with 1..\* on the left, and state that we're only managing apartment buildings here, so all buildings have to have at least one apartment.

- This is a fairly clear example of a composition association, much like our dormitory-room example in the lectures.

**3. Model the classes Book, Publisher and Author. There may not be associations between all classes (as you see fit).**



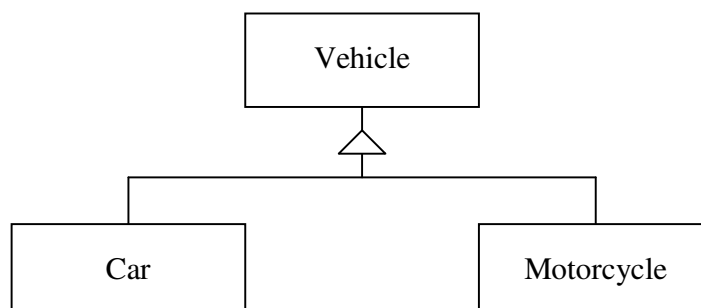
**Assumptions:**

- A book can only be published by one publisher, but must be published to be considered a book.
- A publishing company must publish at least one book
- A book may be written by many authors (e.g., a textbook)
- An author does not have to write a book (e.g., poets, etc.)

**Notes:**

- There will be quite a bit of variability here, depending upon whether students think a book may be published by several publishers, etc. Depends on their stated assumptions, examples.
- The most interesting thing about this one is whether there is a relationship between Author and Publisher, other than the fact that a book written by an author is published by some publisher. If this is the only relationship, it is accommodated by the relationship “through” the Book class. Some students have suggested that an Author may have a relationship with a publisher before they even write the book – perhaps they are paid an advance fee to write the book.

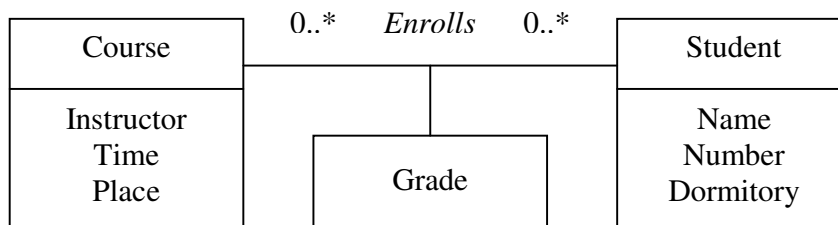
**4. Model the classes Vehicle, Car & Motorcycle. Hint: this one is unusual.**



**Notes:**

- This is a fairly clear example of an inheritance association. Cars and motorcycles are merely kinds of vehicles – a car “is-a” vehicle.
- Note that we do not use multiplicities with inheritance associations.

5. **Make a Class Diagram for the college Registrar's office. The office maintains data about each course, including the instructor, and the time and place of the course meetings and the students in each class, including the student name, number and dormitory. Each time a student enrolls in a class, a grade is recorded.**



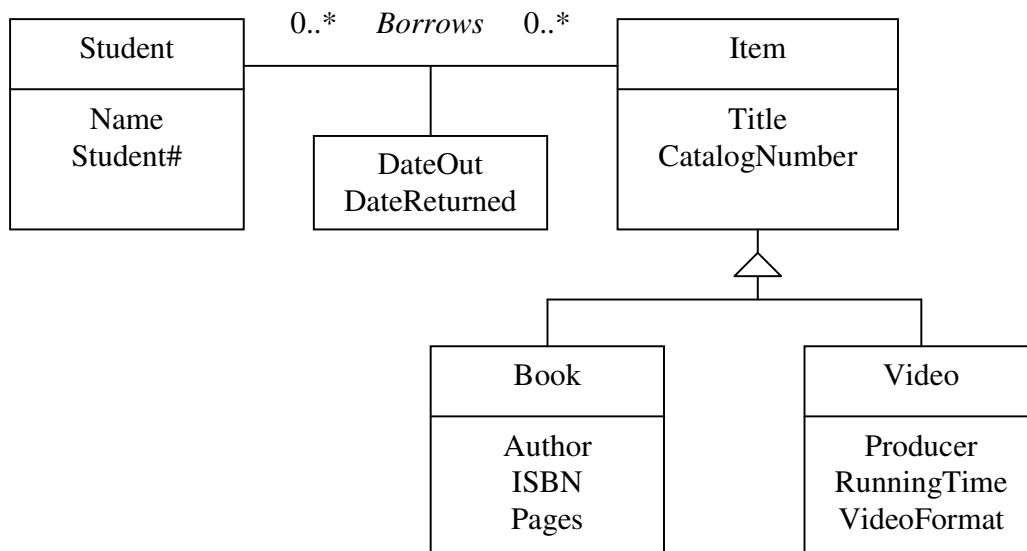
**Assumptions:**

- Courses do not have to have students enrolled (they may be not offered at all times)
- Students do not have to enrol in a course (they could audit courses, in which case they would not be assigned a grade, but they would need to pay fees, etc.)

**Notes:**

- Here we're not given the classes explicitly, so we have to pick them out from the text.
- We also have to include properties, not just the class name.
- Grade must be an association class, since as we discussed in the lecture, grade must apply to the association of a given course with a given student to be effective.
- Here we encounter nouns like Dormitory, and Instructor. One of the early design decisions is whether these are Classes, or merely attributes of another class. For example, Instructor could be a class by itself, but it could also be an attribute of the course. The key discussion point here is 1) whether we know anything else about the instructor (office hours, research interests, etc.) or not. If we have no attributes other than simply the name, maybe we should make Instructor an attribute of the course. 2) whether we have a need, as the registrar's office, to maintain instructor information. Again, the focus of the user of this data may dictate what is a class by itself and what is merely an attribute.
- We'll hit the same situation with Dormitory – if our business has little to do with managing dorm rooms, and we know nothing more than the building number, perhaps it is an attribute of Student rather than a class of its own.
- I've modeled Dormitory and Instructor as properties, but students may present a valid argument for breaking these out into two more classes in this diagram.
- Some students have asked whether the registrar's office itself is a class. If we assume that this is for a single university, then there is only one registrar's office and thus storing that information would be redundant. If we are modeling the service of several universities at once, then yes, we need to include the registrar's office to indicate which course is offered at which university.

6. **Make a Class Diagram for the Library.** We need to keep track of items lent, including videos and books. We also need to maintain information on students and which items they borrow, including when an item is checked out and returned. Add a small number of relevant attributes. State any assumptions that need to be made.



**Assumptions:**

- Items do not have to be lent, but they can be borrowed by a number of students
- Students do not have to borrow anything, but they can borrow a number of items.

**Notes:**

- There shouldn't be too much variation in responses here. Perhaps students represent the Book/Video concept with an attribute of Item called ItemType, or something. That would be acceptable.
- The Dates the item was signed out and returned must be an association class if we are to store a history of borrowing, so that each time a student borrows the item, we can store the sign-out and sign-in dates.
- Some students have asked whether the Library is a class. Like the Registrar's office, since there is only one library and it applies to all transactions, we need not store this information.

### 1.2.4 Modeling Hints

Remember: models are for communication. The main goal of a class diagram is to communicate and document a data model, so clarity should be one of the main considerations. Models can be made to be extremely complex. However, a simple model which meets the needs of the database will be a more effective and efficient one.

Remember: include only important things. Creating a data model is largely a fence-building exercise, where the designer must determine what information is important and what is not. Extraneous data should be omitted from the model, speeding up processing and reducing storage requirements in the finished system. Often deciding *what to model* is as important as how to model it.

Distinguishing Classes and Associations. Sometimes it is difficult to determine from a text description what should be a class and what should be the Association.

- Classes usually correspond to nouns
- Associations usually correspond to verbs

Distinguishing Classes and Properties. Properties are sometimes hard to distinguish from classes, because they are both nouns in a verbal description. For example, in exercise 5 above (the college registrar's database), students often ask whether *Instructor* is a property of a course, or whether it is a class with an association to the course.


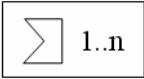
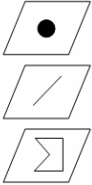
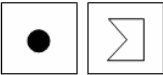

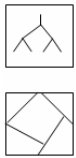
- If there is only one instructor for the course, and we know nothing more about the instructor, it is probably best modeled as a property of the course.
- If we know more about the instructor (perhaps an office location or research specialisation) or if more than one instructor teaches a given course, we should probably create a new class in which to store this information.
- This problem relates to relationships between attributes in a class, as well as relationships between different classes. The former is called Functional Dependency, and we will explore this in detail in the following lecture topic.

### 1.2.5 Modeling Spatial Data

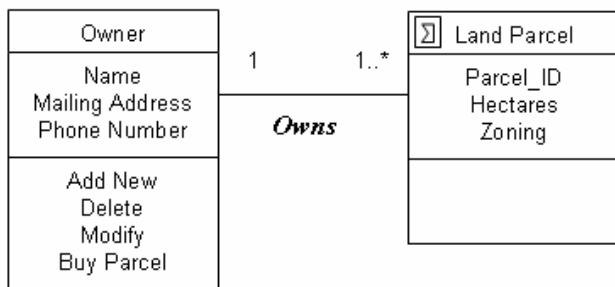
The standard class diagram is effective for most database design requirements. It has limitations, however, when representing spatial features and spatial relationships. Recently many (e.g., Shekhar, 2003) have suggested that standard Entity Relationship or UML Class Diagram notations be extended to include these spatial elements. **Pictograms** are often added to the notation to indicate the spatial semantics of the model.

A summary of the pictograms which may be present in a class diagram are summarised below in Table 1-4.

Table 1-4 Pictogram Element Summary

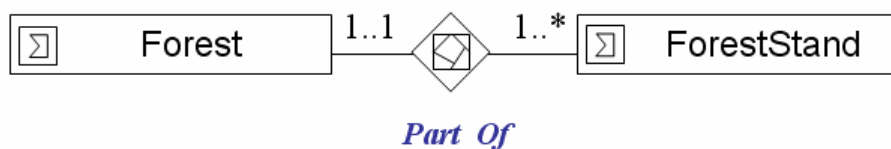
Pictogram	Meaning
<b>Basic Shapes</b> 	<p>In a vector model, features are represented as a point, line or polygon in the database. The following basic shape types may be represented:</p> <p>Point Feature</p> <p>Line Feature</p> <p>Polygon Feature</p>
<b>Multipart Features</b> 	<p>Some features are represented by groupings of simple shapes. An island nation such as Indonesia, for example, would require that several polygons together be considered a single feature. In these cases, we may note the multiplicity of the collection of basic shapes as part of the notation. Multiplicities may be represented as in a class diagram:</p> <ul style="list-style-type: none"> <li>▪ 0..1</li> <li>▪ 1..1</li> <li>▪ 0..n</li> <li>▪ 1..n</li> <li>▪ n..m</li> </ul>
<b>Derived Shape</b> 	<p>Some features in a GIS are made up of groupings of other features. For example, the spatial representation of Europe might be considered nothing more than the aggregate of the polygons representing member countries. Here we represent the shape by an italicized symbol.</p>
<b>Alternate Representations</b> 	<p>Some features might be represented by two different shape types, depending upon the scale of presentation. For example, a city might be represented as a point at very small scales, but as the map is zoomed in it might be presented as a polygon at larger scales. In such cases, several shape types may be indicated.</p>
<b>User-Defined Shape</b> 	<p>There are some unusual situations where a single shape type cannot properly represent the spatial meaning. For example, a city water network might be comprised of many shape types: points for valves, lines for pipes, and polygons for reservoirs. The * notation would be used in such a case. For more unusual situations, the ! notation might be used with an accompanying explanation.</p>
<b>Association Notations</b> 	<p>Notations may also be added to associations to denote spatial meaning:</p> <ul style="list-style-type: none"> <li>▪ Part_of – Network: features form part of a linear network</li> <li>▪ Part_of – Partition: features form part of a polygonal decomposition of space, such as land use polygons.</li> </ul>

For example, in the simple diagram we defined above depicting Owner-LandParcel, we might add a pictogram to the LandParcel class to indicate that parcels are polygon features. No pictogram is added to the Owner class, since Owner is a non-spatial object. The result might resemble the diagram in Figure 1-20.



**Figure 1-20 Owner-LandParcel with Pictogram**

Figure 1-21 shows an example of pictogram use which includes an association notation.



**Figure 1-21 Forest-ForestStand Association**

## 1.3 The Relational Model

In this topic we'll elaborate on the most common database structure – the relation model. Relational databases form the vast majority of database implementations worldwide, and the concepts are key to understanding GIS data structures and use.

This topic contains the most conceptually challenging material of the course. It is quite abstract, while much of the rest of the course will be highly technical and applied in nature. There are many examples used during the lecture material, and self-paced exercises are supplied with their solutions so students can practice this material.

### 1.3.1 Terminology

In a discussion of databases, you will encounter many terms which mean the same thing. In the following figures, we will discuss three terms, as might be used by different communities of GIS people:

- **User**, someone who uses the software and has a basic understanding of the underlying structures.
- **Relational Algebra** is an area of mathematics / computer science which deals with things like set theory.
- **Programmer** view is more the nuts-and-bolts view of the data structure.

Feel free to use any of the three terms shown in these figures for your assignments and exams.

	FCD	SOURCE	FEATURE	DEFINITION	REMARKS
▶	AA00350000	CCSM	Agricultural Land Reserve		
	AA00400000	CCSM	Agricultural Region	Land in agricultural use (excludin	
	AA10550000	CCSM	Farm	A tract of land devoted to agricult	
	AA10550110	CCSM	Farm - Dairy		
	AA10550120	TRIM	Farm - Experimental		
	AA10550130	CCSM	Farm - Fruit		
	AA10550140	CCSM	Farm - Fur		
	AA10550150	CCSM	Farm - Mixed		
	AA10550160	CCSM	Farm - Pig		
	AA10550170	CCSM	Farm - Potato		
	AA10550180	CCSM	Farm - Poultry	An establishment specializing in t	
	AA10550190	CCSM	Farm - Seed Crop		
	AA10550200	CCSM	Farm - Sheep		
	AA10550210	CCSM	Farm - Sod		
	AA10550220	CCSM	Farm - Tree		
	AA10550300	BCE	Farm - Homestead	A tract of land devoted to agricult	Added on behalf of Ministry of Agriculture
	AA10650000	CCSM	Feedlot	A tract of land on which livestock	
	AA23150000	CCSM	Ranch	An estate, with its buildings, corr	

User: Table

Relational Algebra: Relation

Programmer: File

Figure 1-22 Entire Table

Fcode	Source	Feature	Definition	Remarks
AA00350000	CCSM	Agricultural Land Reserve		
AA00400000	CCSM	Agricultural Region	Land in agricultural use (excludin	
AA10550000	CCSM	Farm	A tract of land devoted to agricult	
AA10550110	CCSM	Farm - Dairy		
AA10550120	TRIM	Farm - Experimental		
AA10550130	CCSM	Farm - Fruit		
AA10550140	CCSM	Farm - Fur		
AA10550150	CCSM	Farm - Mixed		
AA10550160	CCSM	Farm - Pig		
AA10550170	CCSM	Farm - Potato		
AA10550180	CCSM	Farm - Poultry	An establishment specializing in t	
AA10550190	CCSM	Farm - Seed Crop		
AA10550200	CCSM	Farm - Sheep		
AA10550210	CCSM	Farm - Sod		
AA10550220	CCSM	Farm - Tree		
AA10550300	BCE	Farm - Homestead	A tract of land devoted to agricult	Added on behalf of Ministry of Agriculture
AA10650000	CCSM	Feedlot	A tract of land on which livestock	
AA23150000	CCSM	Ranch	An estate, with its buildings, corr	

User: Row

Relational Algebra: Tuple

Programmer: Record

Figure 1-23 Single Row of a Table

Fcode	Source	Feature	Definition	Remarks
AA00350000	CCSM	Agricultural Land Reserve		
AA00400000	CCSM	Agricultural Region	Land in agricultural use (excludin	
AA10550000	CCSM	Farm	A tract of land devoted to agricult	
AA10550110	CCSM	Farm - Dairy		
AA10550120	TRIM	Farm - Experimental		
AA10550130	CCSM	Farm - Fruit		
AA10550140	CCSM	Farm - Fur		
AA10550150	CCSM	Farm - Mixed		
AA10550160	CCSM	Farm - Pig		
AA10550170	CCSM	Farm - Potato		
AA10550180	CCSM	Farm - Poultry	An establishment specializing in t	
AA10550190	CCSM	Farm - Seed Crop		
AA10550200	CCSM	Farm - Sheep		
AA10550210	CCSM	Farm - Sod		
AA10550220	CCSM	Farm - Tree		
AA10550300	BCE	Farm - Homestead	A tract of land devoted to agricult	Added on behalf of Ministry of Agriculture
AA10650000	CCSM	Feedlot	A tract of land on which livestock	
AA23150000	CCSM	Ranch	An estate, with its buildings, corr	

User: Column

Relational Algebra: Attribute

Programmer: Field

Figure 1-24 Single Column of a Table

### 1.3.2 Relations

We have discussed how the basis of the relational model is a relation, and that a relation is very much like a table. For a table to be considered a relation, in the formal sense, it must meet the following five criteria:

1. All entries in a given column must be of the same type. The “same type” means all the entries in a given column are a number, or all the entries are a date. We don’t want to mix textual data with numbers and dates.
2. Each Column has a unique name. Each column is given a name which describes what data are stored in that column, and these names must be unique within the table (i.e., the same name can be used in *different* relations, but not twice in the *same* table).
3. No rows in the table may be identical. If two rows have exactly the same information, in all columns, then the table is not a relation.

4. The order of the columns is insignificant. We don't want to have some implied meaning relating to the order of the columns. We want a situation where you could shuffle the columns around and the meaning of the data would not be affected.
5. The order of the rows is insignificant. Same as #4 above, but this time we don't want the order of the rows to have some implicit meaning. If you had a table of the *Top Five Reasons to Learn GIS*, this might fail our test, since the fact that a row is first might imply it is the most important reason. We can accomplish the same thing without failing test #5 by adding a column called *Rank*, for example, and storing a number from 1 to 5 to denote how important the reason is. A relation needs to be able to change the order of the rows without affecting the meaning of the data.

To illustrate the concept, consider the following tables and ask yourself if these meet the five criteria identified above. Identify any criterion which is not satisfied by the table.

**Table 1-5 Table Example #1**

Stock#	Description	Description	Cost
10035	Seat Cover	Bucket	\$50
26658	Air Freshener	Pine	\$3
6685	Tire	14R42	\$80
35891	Jack	2 Tonne	\$95

Table 1-5 has two column names called Description. Despite the fact that they in fact store slightly different information, this repeated column name causes this table to fail test #2 and is therefore not a relation.

**Table 1-6 Table Example #2**

Stock#	Name	Description	Note
10035	Seat Cover	Bucket	\$50
26658	Air Freshener	Pine	Yes
6685	Tire	14R42	\$80
35891	Jack	2 Tonne	Jan 1, 2006

Table 1-6 now has unique column names, but the final column has been modified as well. This column now contains a variety of different types of data: currency, binary (yes, no) and date. This causes the table to fail test #1, and is therefore not a relation.

**Table 1-7 Table Example #3**

Stock#	Description	Cost
10035	Seat Cover	\$50
26658	Air Freshener	\$3
6685	Tire	\$80
10035	Seat Cover	\$50

Table 1-7 contains duplicate rows (the first and last rows are identical, for all columns). This table violates test #3, and is therefore not a relation.

### 1.3.3 Functional Dependency

It's often the case that two columns in the same table have some relationship with each other. For example, in a table containing your driver's license number and your name (in different columns, but in the same row), there is a clear relationship between these two pieces of information.

**Functional Dependency** is a relationship between attributes of the same relation. Given the value of one attribute, you can determine, calculate, or look up the value of another attribute.

For example, in a large table of credit card balances, if we know the value of CustomerAccountNo, we can look up CustomerAccountBalance, and be confident that we have read the correct balance. If this is true, CustomerAccountBalance is **functionally dependent** on CustomerAccountNo. Or, we can also say that CustomerAccountNo **determines** CustomerAccountBalance.

A more formal definition would be:

**Given a relation R, attribute Y of R is functionally dependent on attribute X of R if and only if each X value in R has associated with it precisely one Y value in R.**

Using our example above, CustomerAccountNo determines CustomerAccountBalance only if for each value of CustomerAccountNo in the relation, there is one, and only one, possible CustomerAccountBalance.

In notation, a functional dependency is expressed as: Student# → StudentName. Student# determines StudentName, since for any given Student#, there can be one and only one StudentName. The attribute on the left of the arrow (Student#) is called a **determinant**.

For example, consider the following simple table, shown in Table 1-8:

**Table 1-8 Simple Functional Dependency Example**

Column1	Column2
A	1
A	1
B	1

First, let us consider whether  $\text{Column1} \rightarrow \text{Column2}$  (whether Column1 determines Column2). For each value in Column1, is it always the same value in Column2? Here, we find that every time an “A” appears in Column1, there is always a “1” in Column2 in the same row, and for each occurrence of “B” in Column1, there is always a “1” in Column2 in that row. This indicates that yes, for this limited set of rows, Column1 determines Column2.

Secondly, does  $\text{Column2} \rightarrow \text{Column1}$ ? Here, we find that for every occurrence of “1” in Column2, there can be either an “A” or a “B” in Column1. As a result, Column2 does not determine Column1.

Sometimes functional dependencies involve groups of attributes. For example, consider a table with Student numbers, course names and grades. If you were given just the student number, you could not be guaranteed to find a single Grade in the table, since students often take several courses. Likewise, if you were given only the course name, again, there would be several Grades associated with a given course because several students take the same course. In this case you need both the Student# and CourseName to find a unique Grade. This is shown in notation as:

$(\text{Student\#}, \text{CourseName}) \rightarrow \text{Grade}$

It can also be the case that a single attribute can determine several other attributes. For example, the following says that your Driver’s License Number determines both your name and address. In a table with this information, given your Driver’s License Number, there would be one and only one Name and Address.

$\text{Driver\#} \rightarrow \text{Name, Address}$       (Same as  $\text{Driver\#} \rightarrow \text{Name}$ ,  $\text{Driver\#} \rightarrow \text{Address}$ )

There are two notations for showing a dependency. To compare the two notations, consider the example presented in Table 1-9. In this example, I’ve given the three columns “aliases” of A, B and C so that the notation  $A \rightarrow B$  is easier to read. We’ll use this A, B, C notation quite a bit while discussing this topic.

**Table 1-9 Example Comparing Notations**

Student (A)	Course (B)	Instructor (C)
Tom	GII-05	Dave
Bill	GII-05	Michael
Tom	GII-01	Brad
Harry	GII-05	Dave

The dependencies present in this table are:  $C \rightarrow B$ , and  $AB \rightarrow C$ . As a self-test exercise, examine the table to verify why this is the case.

The two notations are:

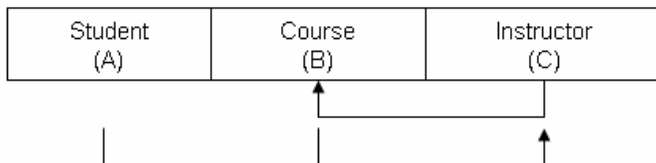
- 1) Using the “arrow” notation, as we have seen in previous examples. The dependencies for Table 1-9 appear like this:

$C \rightarrow B$

$AB \rightarrow C$

*Notation*

- 2) Using Dependency Diagrams. The same dependencies can be depicted by drawing arrows between columns in the table. This method can be easier to see at a glance what is happening in the table, but takes longer to write out.



*Diagram*

You may use either method for your assignments and exams, but we will stay with the notation method for lecture examples for consistency.

### 1.3.4 Keys

A **key** is a group of one or more attributes which uniquely identify a row in a relation. A key not only determines another column, but determines the entire row (all the columns). Consider a relation which stores a set of students, which activities they participate in, and the cost of taking part in the activities.

Table 1-10 (from Kroenke, 1998) shows the situation where a student can participate in only one activity. In this case, Student# is unique in the table, so it can serve as a key. Given a Student#, we can uniquely identify a single row in Table 1-10. The key for this relation is simply (Student#).

**Table 1-10 Student-Activity Relation, Student# as Key**

Student#	Activity	Fee
100	Skiing	\$200
150	Swimming	\$50
175	Squash	\$50
200	Squash	\$50

Table 1-11 shows the same table, but this time students are allowed to participate in several activities. In this situation, Student# is no longer unique. Given Student# 100, we can no longer identify a single row in the relation. We must now include both Student# and the Activity to uniquely identify a row. The key for this relation is (Student#, Activity). This is known as a **Composite Key**, because it involves more than one attribute.

**Table 1-11 Student-Activity Relation, Student# and Activity as Key**

Student#	Activity	Fee
100	Skiing	\$200
150	Swimming	\$50
175	Squash	\$50
100	Squash	\$50

A key is a set of one or more attributes (columns) of a relation that:

- uniquely identifies all other attributes in the relation
- is a minimal set of attributes – a combination that involves the least number of attributes that can still determine the entire relation. So if A is a key, AB would not be a key because it is not minimal, even if it uniquely identifies a row.

Sometimes there can be more than one possible key, all of which can satisfactorily serve as a key of a relation. These are called **Candidate Keys**.

### **Example 1:**

Consider the Student-Activity table shown in Table 1-10, and try to answer the following questions. Answers are discussed in the section immediately following.

- Is this a relation?
- Is Student# a key?
- Is Student# a determinant?
- Is Activity a key?

- e) Is it a determinant?
- f) Is Fee a key?
- g) Is it a determinant?

Solutions:

- a) Is this a relation?

To determine this, refer to the five tests identified in Section 1.3.2 above. In this case, our table violates none of the tests, so yes, this is a relation.

- b) Is Student# a key?

Does Student# uniquely identify a row? In this case, Student# is unique in the table, so yes, it can uniquely identify a row and would thus be considered a key.

- c) Is Student# a determinant?

Since we know that Student# is a key, it must also be a determinant since it can uniquely identify a row. To identify a row, it must determine all the other columns, so it must be true that Student#  $\rightarrow$  Activity and that Student#  $\rightarrow$  Fee. Yes, Student# is a determinant. Note that **keys are always determinants**.

- d) Is Activity a key?

Is it true that for a given Activity, such as Squash, you can uniquely identify a row? In this case, Squash occurs twice, so we cannot uniquely identify a row. Activity is not a key.

- e) Is it a determinant?

Is it true that for a given Activity there is always the same Student# or Fee? In this case, each Activity is always the same price, so Activity  $\rightarrow$  Fee. Activity doesn't determine Student#, though.

This is an example of an attribute (Activity) which is a determinant but not a key. Note that while a key is always a determinant, **a determinant is not always a key**.

- f) Is Fee a key?

Is it true that for a given Fee, such as \$50, you can uniquely identify a row? Not in this case, since \$50 occurs three times in the table. Fee is not a key.

- g) Is it a determinant?

Is it true that for a given Fee, such as \$50, we find only one Student# or one Activity? In this case, \$50 occurs with three different students and two activities. Therefore Fee determines neither Activity nor Student#. Fee is not a determinant.

### Determining Keys from Functional Dependencies

We can determine the key of a relation by examining the functional dependencies. Dependencies describe the relationship between attributes in the relation, so with this information we can determine how to uniquely identify a row.

The following rules can be used to help simplify or manipulate functional dependencies:

- |  |                             |
|--|-----------------------------|
| i. $A \rightarrow A$   | (self-determination)        |
| ii. If $A \rightarrow BC$ then $A \rightarrow B$ and $A \rightarrow C$     | (decomposition)             |
| iii. If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$     | (transitivity)              |
| iv. If $A \rightarrow BC$ and $C \rightarrow D$ , then $A \rightarrow BCD$ | (accumulation)              |
| v. If $A \rightarrow B$ , then $AC \rightarrow BC$                         | (augmentation)              |
| vi. If $A \rightarrow B$ and $A \rightarrow C$ , then $A \rightarrow BC$   | (union)                     |
| vii. If $A$ is a key, then $AB$ is not a key                               | (minimal set of attributes) |

We will use all of these rules to try to determine the minimum subset of attributes which can determine the entire relation. Probably the most effective way to demonstrate how we apply these rules is to work through several examples. In the examples which follow, we will make extensive use of notation such as  $R(A, B, C)$ . This simply describes a relation with three columns, and the columns are referred to as  $A$ ,  $B$  and  $C$  for simplicity.

The general approach we will use is to:

1. Try each single-attribute key (e.g.,  $A$ ,  $B$ , etc.) to determine if it is a key.
2. If a single attribute key is found, we do not need to check composite keys, since they would not be minimal.
3. If no single-attribute key is found, we should try all possible composite keys which include two attributes (e.g.,  $AB$ ,  $AC$ ,  $BC$ , etc.).
4. If a two-attribute composite key is found, there is no need to check for a three-attribute composite key.
5. If no two-attribute composite key is found, check for three-attribute composite keys (e.g.,  $ABC$ ,  $ABD$ ,  $ACD$ , etc.).
6. ...And so on, until a candidate key is found.

In practice, you will not be asked to check for composite keys involving more than two attributes, due to the complexity of testing all possible combinations. We will try to keep all of the exercises manageable in this respect.

**Example 2:**

**Given a relation  $R(A, B, C, D)$  with dependencies  $A \rightarrow BC$  and  $B \rightarrow D$ , determine the candidate key(s).**

Solution:

We first need to check all possible single-attribute keys. In  $R(A, B, C, D)$ , this involves checking each of A, B, C and D to see if they are candidate keys.

Try A:

$A \rightarrow A$	(self-determination)
$A \rightarrow BC$ so $A \rightarrow B, A \rightarrow C$	(decomposition)
$A \rightarrow B, B \rightarrow D$ so $A \rightarrow D$	(transitivity)
$A \rightarrow A, A \rightarrow B, A \rightarrow C, A \rightarrow D$ so $A \rightarrow ABCD$	(union)

Since  $A \rightarrow ABCD$ , A is a candidate key.

Try B:

$B \rightarrow B$	(self-determination)
$B \rightarrow D$	(given)
$B \rightarrow B, B \rightarrow D$ so $B \rightarrow BD$	(union)

No other dependencies have B or D as a left side, so we cannot determine any other columns. Since B only determines BD, it does not determine the entire relation, and is therefore not a candidate key.

Try C:

C determines nothing except itself (self-determination), so we cannot determine A, B or D given just C. Therefore C is not a candidate key.

Try D:

D determines nothing except itself (self-determination), so we cannot determine A, B or C given just D. Therefore D is not a candidate key.

We do not need to check for composite keys (keys involving more than one column), since a single-attribute key is minimal. Therefore, the only candidate key for  $R(A, B, C, D)$  is A.

**Example 3:**

**Given a relation  $R(A, B, C)$  with dependencies  $B \rightarrow C$  and  $C \rightarrow B$ , find the candidate keys.**

Solution:

We first need to check all possible single-attribute keys. In  $R(A, B, C)$ , this involves checking each of A, B and C to see if they are candidate keys.

Try A:

A determines nothing except itself (self-determination), so we cannot determine B or C given just A. Therefore A is not a candidate key.

Try B:

$B \rightarrow B$	(self-determination)
$B \rightarrow C$	(given)
$B \rightarrow B, B \rightarrow C$ so $B \rightarrow BC$	(union)

Given B, we can determine BC. With B and C, we cannot determine A. There is no dependency with A on the right side, so we cannot possibly determine A from B and/or C. Since B only determines BC, it does not determine the entire relation, and is therefore not a candidate key.

Try C:

$C \rightarrow C$	(self-determination)
$C \rightarrow B$	(given)
$C \rightarrow B, C \rightarrow C$ so $C \rightarrow BC$	(union)

Given C, we can determine BC. With B and C, we cannot determine A. There is no dependency with A on the right side, so we cannot possibly determine A from B and/or C. Since C only determines BC, it does not determine the entire relation, and is therefore not a candidate key.

We have now checked all the possible single-attribute keys and did not find a candidate key. We now have to check all the possible two-attribute keys. In  $R(A, B, C)$ , this means we need to check AB, AC and BC.

Try AB:

$AB \rightarrow AB$	(self-determination)
$AB \rightarrow AB$ so $AB \rightarrow A$ and $AB \rightarrow B$	(decomposition)
$AB \rightarrow B, B \rightarrow C$ so $AB \rightarrow C$	(transitivity)
$AB \rightarrow A, AB \rightarrow B, AB \rightarrow C$ so $AB \rightarrow ABC$	(union)

$AB \rightarrow ABC$ , so AB is a candidate key.

Try AC:

$AC \rightarrow AC$	(self-determination)
$AC \rightarrow AC$ so $AC \rightarrow A$ and $AC \rightarrow C$	(decomposition)
$AC \rightarrow C, C \rightarrow B$ so $AC \rightarrow B$	(transitivity)
$AC \rightarrow A, AC \rightarrow B, AC \rightarrow C$ so $AC \rightarrow ABC$	(union)

$AC \rightarrow ABC$ , so AC is a candidate key.

Try BC:

Given B and C, there is no way to determine A (A is never on the right side of a dependency), so BC cannot be a candidate key.

Here, we have found two candidate two-attribute composite keys (AB and AC). There is no need to search for three-attribute composite keys, since our two-attribute keys are minimal. Therefore, the two candidate keys for  $R(A, B, C)$  are AB and AC.

In reality, as you get more comfortable with reading functional dependencies and determining keys, you will start to see a pattern and understand that in many cases you can discard potential keys without even checking them. In the above examples, we tested every possible key to ensure students see all possible keys being evaluated. There are methods, though, by which we can reduce the number of possible keys we have to check.

We will use Example 2 above to illustrate this concept. In Example 2, we had a four attribute relation, so if we were to exhaustively check all combinations of keys as large as a two-attribute key, there are a total of 10 possible keys for us to check:

Single-attribute possibilities:	A, B, C and D.
Two-attribute possibilities:	AB, AC, AD, BC, BD and CD.

There are two concepts which should help you speed up the key checking process:

1. If an attribute is not on the left side of any dependency (i.e., it is not a determinant), it cannot possibly be a key by itself, or in combination with other non-determinant attributes, since it does not determine anything. In example 2 above, in  $R(A, B, C, D)$  we had the dependencies:

$A \rightarrow BC$   
 $B \rightarrow D$

Here, C and D are not determinants, so we need not check any key involving only these attributes. As a result, we need not check C, D or CD.

2. If an attribute is not on the right side of any dependency (i.e., it is not determined by anything), it must be a part of the candidate key(s) (either as a key by itself, or as part of a composite key), since the only way to determine it is to include it in the key. Again, using example 2 above, we had the dependencies:

$A \rightarrow BC$   
 $B \rightarrow D$

A cannot be determined by anything else (since it does not appear on the right side of any dependency), so we should expect that A must be either candidate key itself, or must be part of a composite key. By taking advantage of this observation, we can discard all the combinations without A. We really only need to check A, AB, AC and AD.

Keep in mind that if you are unsure whether to check a key combination or not, you can always test it against the dependencies using the rules (such as Transitivity, Union, etc.) defined above to find out if it is a candidate key.

In a professional setting (i.e., if you had to do this in the “real world”), it is important to consider that identifying functional dependencies should not be done exclusively by examining sample data sets.

Dependencies are determined by the semantics, mental models or business rules of the organisation developing the database. Examining our first example (Student# as Key in Table 1-10) might lead one to assume that students can only participate in one activity, but the Business Rules of the organisation will ultimately determine this (i.e.: ask someone who knows!). There might be nothing preventing students participating in more than one activity, but there might not be an example of it in the sample data you're looking at.

### Practice Exercises:

Try on your own to determine the keys of the following relations. Answers are found in the section immediately following these exercises.

1. Given a relation  $R(A, B, C)$  with functional dependency  $A \rightarrow BC$ , determine the candidate key(s).
2. Given a relation  $R(A, B, C, D)$  with dependencies  $A \rightarrow B$  and  $C \rightarrow D$ , find the candidate key(s).

Solutions:

- 1. Given a relation R (A, B, C) with functional dependency  $A \rightarrow BC$ , determine the candidate key(s).**

Solution:

We first need to check all possible single-attribute keys. In R(A, B, C), this involves checking each of A, B and C to see if they are candidate keys.

Try A:

$A \rightarrow BC$	(given)
$A \rightarrow A$	(self-determination)
$A \rightarrow A, A \rightarrow BC$ so $A \rightarrow ABC$	(union)

Since  $A \rightarrow ABC$ , A is a candidate key because it determines the entire relation.

Try B:

B determines nothing except itself (self-determination), so we cannot determine A or C given just B. Therefore B is not a candidate key.

Try C:

C determines nothing except itself (self-determination), so we cannot determine A or B given just C. Therefore C is not a candidate key.

We do not need to check for composite keys (keys involving more than one column), since a single-attribute key is minimal. Therefore, the only candidate key for R(A, B, C) is A.

- 2. Given a relation R(A, B, C, D) with dependencies  $A \rightarrow B$  and  $C \rightarrow D$ , find the candidate key(s).**

Solution:

We first need to check all possible single-attribute keys. In R(A, B, C, D), this involves checking each of A, B, C and D to see if they are candidate keys.

Try A:

$A \rightarrow A$	(self-determination)
$A \rightarrow B$	(given)
$A \rightarrow A, A \rightarrow B$ so $A \rightarrow AB$	(union)

A can determine AB, but cannot determine C or D. There are no dependencies which determine C or D given A or B. Therefore, A is not a key.

Try B:

B determines nothing except itself (self-determination), so it cannot be a candidate key.

Try C:

$C \rightarrow C$	(self-determination)
$C \rightarrow D$	(given)
$C \rightarrow C, C \rightarrow D$ so $C \rightarrow CD$	(union)

C can determine CD, but cannot determine A or B. There are no dependencies which determine A or B given C or D. Therefore, C is not a key.

Try D:

D determines nothing except itself (self-determination), so it cannot be a candidate key.

We have now checked all the possible single-attribute keys and did not find a candidate key. We now have to check all the possible two-attribute composite keys. In  $R(A, B, C, D)$ , this means we need to check AB, AC, AD, BC, BD and CD.

Try AB:

$A \rightarrow B$ , but that does not help – we're still at AB and cannot determine C or D. There is no dependency which allows us to determine C or D from A or B, so AB is not a key.

Try AC:

$AC \rightarrow AC$	(self-determination)
$AC \rightarrow AC$ so $AC \rightarrow A$ and $AC \rightarrow C$	(decomposition)
$AC \rightarrow A, A \rightarrow B$ so $AC \rightarrow B$	(transitivity)
$AC \rightarrow C, C \rightarrow D$ so $AC \rightarrow D$	(transitivity)
$AC \rightarrow A, AC \rightarrow B, AC \rightarrow C, AC \rightarrow D$ so $AC \rightarrow ABCD$	(union)

$AC \rightarrow ABCD$ , so AC is a candidate key.

Try AD:

$AD \rightarrow AD$	(self-determination)
$AD \rightarrow AD$ so $AD \rightarrow A$ and $AD \rightarrow D$	(decomposition)
$AD \rightarrow A, A \rightarrow B$ so $AD \rightarrow B$	(transitivity)
$AD \rightarrow A, AD \rightarrow B, AD \rightarrow D$ so $AD \rightarrow ABD$	(union)

AD can determine 3 of the 4 attributes, but cannot determine C. No dependency determines C, so AD cannot be a key.

Try BC:

$BC \rightarrow BC$	(self-determination)
$BC \rightarrow BC$ so $BC \rightarrow B$ and $BC \rightarrow C$	(decomposition)
$BC \rightarrow C, C \rightarrow D$ so $BC \rightarrow D$	(transitivity)
$BC \rightarrow B, BC \rightarrow C, BC \rightarrow D$ so $BC \rightarrow BCD$	(union)

BC can determine 3 of the 4 attributes, but cannot determine A. No dependency determines A, so BC cannot be a key.

Try BD:

Neither B nor D determine anything else, so we cannot determine A or C. BD cannot be a candidate key.

Try CD:

$C \rightarrow D$ , but that does not help – we still can only determine C and D. No dependency gives us A or B from C or D, so CD cannot be a key.

Here, we have found one candidate two-attribute composite key (AC). There is no need to search for three-attribute composite keys, since our two-attribute key is minimal. Therefore, the candidate key for R(A, B, C, D) is AC.

### 1.3.5 Normalization

We have discussed so far the importance of designing a data model which will fulfill the needs of an organization, and ensuring that relationships between object classes are properly defined (UML Class Diagrams). This process defines the relations of our database and the associations between relations. More recently, we discussed relationships within a relation (functional dependency). Understanding functional dependency is critical to ensuring we have a well-structured database.

A well-structured relation should do the following:

1. Minimize Redundancy. We want to ensure that data elements are not repeated unnecessarily. Where redundancy exists, we introduce storage space and retrieval inefficiency.
2. Allow efficient access to information. We want to make sure that we can find and read or update database elements quickly.
3. Allow inserts, modification and deletion without creating negative side effects or errors or inconsistencies. These negative side effects are called **Anomalies**, and there are three types of anomalies in a database: Deletion Anomalies, Addition Anomalies and Modification Anomalies.

To illustrate these anomalies, we will revisit the Student-Activity relation we have previously discussed. This relation is repeated here in Table 1-12.

**Table 1-12 Student-Activity Relation**

Student#	Activity	Fee
100	Skiing	\$200
150	Swimming	\$50
175	Squash	\$50
200	Squash	\$50

#### Deletion Anomaly

If student 100 in Table 1-12 decides they no longer wish to go skiing, we might delete this first row in the relation. In doing so, however, we not only lose the fact that student 100 is a skier, but the fact that skiing costs \$200. With one deletion, we lose two pieces of information.

#### Addition Anomaly

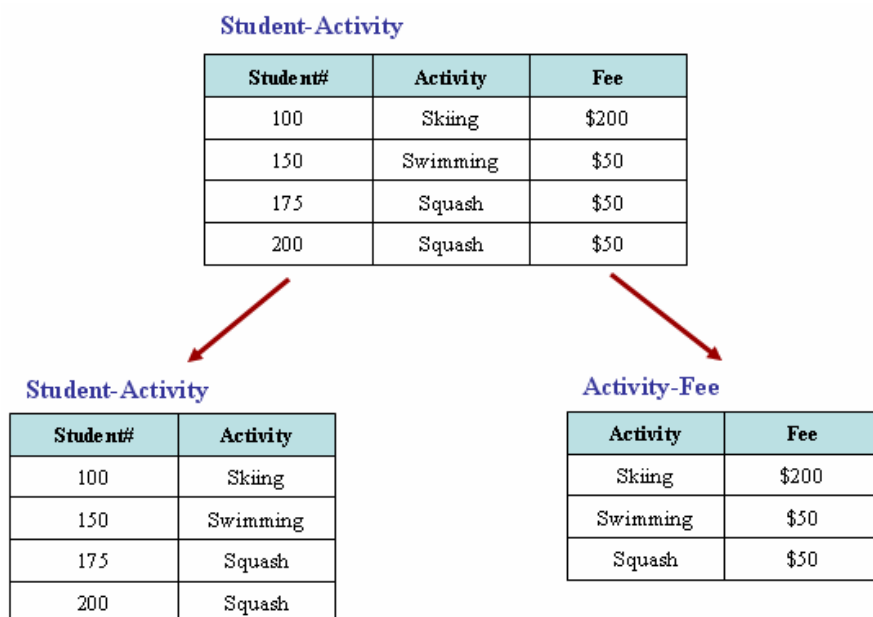
If we want to add the fact that Football costs \$20 to Table 1-12, we are forced to wait until a student wishes to play Football. We can't add information about one entity until we supply information about another.

#### Modification Anomaly

Note that Squash is present twice in Table 1-12, since two students have registered for this activity. If we changed the fact that Squash costs \$50, perhaps to raise the rate to \$60, we may forget to change the value in both places. This would lead to conflicting data, where the price for Squash is different in two different places. This is a modification anomaly.

All three of these problems arise because there are two “themes” or concepts in our Student-Activity relation: Students and what activities they are involved with, and Activities and their costs. Trying to put these two concepts together in one relation means we introduce negative side-effects to manipulating our data.

**Normalization** is the process of structuring relations to minimize these side effects. The essence of normalization is that relations should only contain one “theme”. In the Student Activity example, we should break the Student-Activity table into two tables, as shown in Figure 1-25.



**Figure 1-25 Normalizing the Student-Activity Relation**

Splitting the original table into two tables resolves the insertion, modification and deletion anomalies and also reduces storage space – the fact that Squash is \$50 is now only stored once.

- Insertion: We can insert a new activity with accompanying fee without any students registered for it.
- Modification: There is no opportunity to modify only one instance of a repeated element, since there are no redundant records.
- Deletion: We can freely delete students and their activities without affecting the activities and their fees.

Normalization is classified in varying degrees, depending on how rigorously a relation is structured. In increasing degree of structure, the Normal Forms are:

1. First Normal Form (1NF)
2. Second Normal Form (2NF)

3. Third Normal Form (3NF)
4. Boyce-Codd Normal Form (BCNF)
5. Fourth Normal Form (4NF)
6. Fifth Normal Form (5NF)
7. Domain/Key Normal Form (DK/NF)

These forms are “nested”, meaning that a relation of one form is also of any form above it. For example, a relation in 3rd normal form is automatically also in 2NF and 1NF. Each form has one or more tests which determine whether a relation qualifies. The tests are applied in order, beginning with first normal form and moving up through the normal forms in increasing degrees of structural rigor. When a test is failed (e.g., 3NF test), the relation reverts to the last test it passed (e.g., 2NF).

### First Normal Form

A relation is in first normal form (1NF) if it meets the definition of a relation. The criteria for a table being a relation are::

1. All entries in a given column must be of the same type
2. Each Column has a unique name
3. The order of the columns is insignificant
4. No rows in the table may be identical
5. The order of the rows is insignificant

We have previously reviewed examples of whether a table is a relation or not, so this concept should be familiar to you.

### Second Normal Form

A relation is in second normal form (2NF) if it passes the first normal form test, and:

- there is a *single-attribute key*, **or**
- there is a *composite key* and *all non-key attributes are dependent on the entire key*

There are several terms used in this definition which may need clarifying:

Single-attribute Key: You likely remember from our discussion of keys that a single-attribute key is a key involving one column (e.g., “Student#”, or “A” as key). If we have a single-attribute key, the relation is automatically in 2NF, and we need not consider the composite key part of the rule.

Composite Key: Again, in our discussion of keys, we defined a composite key as one involving more than one attribute (e.g., “(Student#, Activity)”, or AB as key). If we have a composite key, we must look at the second part of the rule (and determine if all non-key attributes depend on the entire key).

**Non-Key Attributes:** A non-key attribute is one which is not part of any key. For example, in a relation  $R(A, B, C, D)$ , with candidate keys  $AB$  and  $AC$ , the only non-key attribute is  $D$ , since  $A$ ,  $B$  and  $C$  are all parts of at least one candidate key.

**Dependent on the Entire Key:** Here, we're concerned about composite keys (since a single-attribute key is automatically 2NF). A non-key attribute dependent on the entire key means that we cannot have the non-key attribute dependent on only part of the key (e.g.,  $C$  is part of  $AC$ ). For example, if we had a key of  $AC$  and a non-key attribute of  $D$ , if there was a dependency  $A \rightarrow D$ , this would fail the 2NF test, since  $D$  depends on  $A$ , which is only part of the key ( $AC$ ).  $D$  would have to depend on  $AC$  (the entire key) for this relation to pass the 2NF test.

Note that if there are no non-key attributes but composite keys, the relation passes the 2NF test.

For example, consider the Student-Activity relation we defined earlier, with  $(\text{Student\#}, \text{Activity})$  as the key. This relation is repeated here as Table 1-13 with its dependencies and keys defined following.

**Table 1-13 Second Normal Form Example**

Student#	Activity	Fee
100	Skiing	\$200
150	Swimming	\$50
175	Squash	\$50
100	Squash	\$50

**Dependencies:**  $\text{Activity} \rightarrow \text{Fee}$   
**Key:**  $(\text{Student\#}, \text{Activity})$

1NF: This table is a relation, so it passes the 1NF test.

2NF: To apply the 2NF test, we first consider whether this is a single-attribute key or not. In this case, we have a composite key, because the key involves two attributes ( $\text{Student\#}$  and  $\text{Activity}$ ). Since we have a composite key, the first part of the rule does not apply and we need to examine the second part.

With the key  $(\text{Student\#}, \text{Activity})$ , the only non-key attribute is  $\text{Fee}$ .  $\text{Fee}$  depends on  $\text{Activity}$ , which is only a part of the composite key, so this relation fails the 2NF test. It reverts to the last test it passed, which was the 1NF test, so we would say that this relation is in first normal form.

### Third Normal Form

A relation is in third normal form (3NF) if it passes the second normal form test, and:

- *there are no transitive dependencies*

Transitive Dependencies: We discussed in the previous section of this document (1.3.4 Keys) the transitive property of dependencies (e.g., if  $A \rightarrow B$  and  $B \rightarrow C$ , then  $A \rightarrow C$ ). If we have dependencies where a transitive situation exists, the relation fails the third normal form test.

- A good rule-of-thumb for finding transitive dependencies is **that there should be no dependencies where a non-key attribute is determined by another non-key attribute** (i.e. both sides of a functional dependency are non-key). If this is true, a transitive dependency exists and the relation fails the 3NF test.

For example, consider the Student-Activity relation in the case where a student can register for only one activity. The relation might be defined as:

<b>Relation:</b>	ACTIVITIES (Student#, Activity, Fee)
<b>Dependencies:</b>	Student# $\rightarrow$ Activity Student# $\rightarrow$ Fee Activity $\rightarrow$ Fee
<b>Keys:</b>	Student#

1NF: In this case, we are told that this is a relation, so we know it passes the 1NF test.

2NF: It has a single-attribute key (Student#), so we can quickly determine that it passes the 2NF test as well.

3NF: Since Student# is the key, it determines both Activity and Fee. Since Activity  $\rightarrow$  Fee, we have a transitive dependency, in that Student#  $\rightarrow$  Activity and Activity  $\rightarrow$  Fee. As verification, by our rule-of-thumb, Activity  $\rightarrow$  Fee is a dependency where both sides are non-key, so it fails that way too. This relation is therefore in second normal form.

**Boyce-Codd Normal Form**

A relation is in Boyce-Codd Normal Form (BCNF) if it passes the third normal form test, and:

- *every determinant is a candidate key*

Determinant: a determinant is the left-side of a functional dependency.

Candidate Key: a candidate key is one of several possible keys.

For every determinant to be a candidate key, all left-sides of functional dependencies must be a candidate key. If there is a determinant which is not a candidate key, the relation fails the BCNF test.

For example, consider the following relation:

<b>Relation:</b>	R (A, B, C)
<b>Dependencies:</b>	B → C C → B
<b>Keys:</b>	AB, AC

1NF: Again, we are told that R(A, B, C) is a relation, so we know this is at least 1NF.

2NF: The candidate keys are composite, so the first part of the rule does not apply. We need to determine if all non-key attributes depend upon the entire key. The candidate keys are AB and AC, so there are no non-key attributes. With no non-key attributes we pass the 2NF test.

3NF: There are no transitive dependencies here, but as a verification there are no dependencies where non-key attributes are found on both side of the dependency (both B and C are present in the candidate keys AB and AC). This relation therefore passes 3NF.

BCNF: B and C are both determinants, but neither are candidate keys, so this relation fails the BCNF test. This relation is therefore in third normal form.

Higher Normal Forms

There are 3 higher normal forms beyond Boyce-Codd normal form: fourth normal form, fifth normal form and domain-key normal form. These are increasingly abstract, and virtually all anomalies are removed by BCNF. Due to this, and the fact that this course is designed to provide a foundation in data structures rather than a comprehensive understanding of normalisation, we will omit these higher forms.

**Practice Exercises**

Try the following normal form examples on your own. Answers are found in the section immediately following these exercises.

1. For the following table, determine:
  - a) any functional dependencies (up to and including two-attribute dependencies)
  - b) the candidate keys (up to and including two-attribute keys)
  - c) its normal form

A	B	C
A	1	x
B	2	y
C	1	z
A	1	z
A	1	y
B	2	z

2. Relation:  $R(A, B, C)$   
Dependencies:  $C \rightarrow B$   
 $AB \rightarrow C$

Determine:

- a) the candidate keys (up to and including two-attribute keys)
- b) its normal form

3. Relation:  $R(A, B, C, D, E, F)$   
Dependencies:  $A \rightarrow B$   
 $C \rightarrow D$   
 $AC \rightarrow E$   
 $AC \rightarrow F$

Determine:

- a) the candidate keys (up to and including two-attribute keys)
- b) its normal form

4. For the following table, determine:
- any functional dependencies (up to and including two-attribute dependencies)
  - the candidate keys (up to and including two-attribute keys)
  - its normal form

A	B	C
1	a	1001
2	b	1001
3	a	2001
4	a	2001
5	a	3001

5. Relation:  $R(A, B, C, D)$   
Dependencies:  $C \rightarrow D$   
 $C \rightarrow A$   
 $B \rightarrow C$

Determine:

- the candidate keys (up to and including two-attribute keys)
- its normal form

Solutions:

1. For the following table, determine:

- any functional dependencies (up to and including two-attribute dependencies)
- the candidate keys (up to and including two-attribute dependencies)
- its normal form

A	B	C
A	1	x
B	2	y
C	1	z
A	1	z
A	1	y
B	2	z

FD's:  $A \rightarrow B$

Keys: **AC** (We won't show why all the other possible keys aren't keys, just why this one is a key)

$AC \rightarrow AC$

(self-determination)

$AC \rightarrow AC$  so  $AC \rightarrow A$  and  $AC \rightarrow C$

(decomposition)

$AC \rightarrow A$ ,  $A \rightarrow B$ , so  $AC \rightarrow B$

(transitivity)

$AC \rightarrow A$ ,  $AC \rightarrow B$ ,  $AC \rightarrow C$ , so  $AC \rightarrow ABC$

(union)

1NF: Meets all the requirements of a relation, so 1NF.

2NF: Not a single-attribute key.

Composite key, so non-key attributes must be dependent on entire key.

Non-key attribute is B, since AC is the key.

B depends upon only A, which is just part of the key (AC). Fails 2NF

3NF: n/a

BCNF: n/a

Result: 1NF

**2. Relation:**  $R(A, B, C)$

**Dependencies:**  $C \rightarrow B$

$AB \rightarrow C$

**Determine:**

**a) the candidate keys (up to and including two-attribute dependencies)**

**b) its normal form**

FD's:  $C \rightarrow B$

$AB \rightarrow C$  (given)

Keys: **AB, AC** (We won't show why all the other possible keys aren't keys, just why these ones are a key)

$AC \rightarrow AC$

(self-determination)

$AC \rightarrow AC$  so  $AC \rightarrow A$  and  $AC \rightarrow C$

(decomposition)

$AC \rightarrow C, C \rightarrow B$ , so  $AC \rightarrow B$

(transitivity)

$AC \rightarrow A, AC \rightarrow B, AC \rightarrow C$ , so  $AC \rightarrow ABC$

(union)

$AB \rightarrow AB$

(self-determination)

$AB \rightarrow C$

(given)

$AB \rightarrow AB, AB \rightarrow C$ , so  $AB \rightarrow ABC$  (union)

1NF: Given as a relation, so passes 1NF.

2NF: Not a single-attribute key.

Composite key, so non-key attributes must be dependent on entire key.

No non-key attributes, since AB, AC are candidate keys, so passes 2NF.

3NF: No transitive dependencies?  $AB \rightarrow C$  and  $C \rightarrow B$  appears to be a transitive dependency, but in reality it's not since  $C \rightarrow B$  just determines B again, which is part of  $AB \rightarrow C$  (i.e., C doesn't determine anything that isn't already part of AB). The other way to verify this is to search for dependencies where both sides of the FD are non-key. Since there are no non-key attributes, we can't have such a situation, so there are no transitive dependencies.

BCNF: All determinants are candidate keys? No, because we have a determinant (C), which is not a candidate key. This relation fails BCNF.

Result: 3NF

**3. Relation:**  $R(A, B, C, D, E, F)$

**Dependencies:**  $A \rightarrow B$   
 $C \rightarrow D$   
 $AC \rightarrow E$   
 $AC \rightarrow F$

**Determine:**

- a) the candidate keys (up to and including two-attribute dependencies)
- b) its normal form

FD's:  $A \rightarrow B$   
 $C \rightarrow D$   
 $AC \rightarrow E$   
 $AC \rightarrow F$

Keys: **AC**  
 $AC \rightarrow AC$  (self-determination)  
 $AC \rightarrow AC$  so  $AC \rightarrow A$  and  $AC \rightarrow C$  (decomposition)  
 $AC \rightarrow A, A \rightarrow B$ , so  $AC \rightarrow B$  (transitivity)  
 $AC \rightarrow C, C \rightarrow D$ , so  $AC \rightarrow D$  (transitivity)  
 $AC \rightarrow E$  (given)  
 $AC \rightarrow F$  (given)  
 $AC \rightarrow A, AC \rightarrow B, AC \rightarrow C, AC \rightarrow D,$   
 $AC \rightarrow E, AC \rightarrow F$  so  $AC \rightarrow ABCDEF$  (union)

1NF: Given as a relation; passes 1NF.

2NF: Not a single-attribute key.

Non-key attributes B, D, E and F (key is AC)

Both B and D are dependent on only parts of the key, so this relation fails 2NF.

3NF: n/a

BCBF: n/a

Result: 1NF

**4. For the following table, determine:**

- a) any functional dependencies (up to and including two-attribute dependencies)
- b) the candidate keys (up to and including two-attribute dependencies)
- c) its normal form

A	B	C
1	a	1001
2	b	1001
3	a	2001
4	a	2001
5	a	3001

FD's:  $A \rightarrow B$   
 $A \rightarrow C$

Keys:

<b>A</b>	
$A \rightarrow A$	(self-determination)
$A \rightarrow B$	(given)
$A \rightarrow C$	(given)
$A \rightarrow A, A \rightarrow B, A \rightarrow C$ so $A \rightarrow ABC$	(union)

1NF: Meets all the requirements of a relation, so 1NF.

2NF: Single-attribute key; passes 2NF.

3NF: No transitive dependencies. As a double-check, there are no functional dependencies where both sides are non-key. Passes 3NF.

BCNF: All determinants are candidate keys? Here the only determinant is A, which is also the key, so this passes BCNF.

Result: BCNF

**5. Relation:**  $R(A, B, C, D)$

**Dependencies:**  $C \rightarrow D$

$C \rightarrow A$

$B \rightarrow C$

**Determine:**

a) the candidate keys (up to and including two-attribute dependencies)

b) its normal form

FD's:  $C \rightarrow D$

$C \rightarrow A$

$B \rightarrow C$

Keys: **B**

$B \rightarrow B$

(self-determination)

$B \rightarrow C$

(given)

$B \rightarrow C, C \rightarrow A$ , so  $B \rightarrow A$

(transitivity)

$B \rightarrow C, C \rightarrow D$ , so  $B \rightarrow D$

(transitivity)

$B \rightarrow A, B \rightarrow B, B \rightarrow C, B \rightarrow D$ , so  $B \rightarrow ABCD$

(union)

1NF: Given; passes 1NF.

2NF: Single-attribute key; passes 2NF.

3NF: Transitive dependencies exist.

Two of them, actually:  $B \rightarrow C, C \rightarrow A$ ;  $B \rightarrow C, C \rightarrow D$

Fails 3NF.

BCNF: n/a

Result: 2NF

### 1.3.6 Denormalization

Normalization seeks to minimize redundancy and avoid anomalies. As such, it should form part of the design process of every database. In a basic sense, the normalization process accomplishes these goals by breaking tables into two or more tables, each with its own “theme”, or concept. We have a single theme in a relation if there are no dependencies other than those involving keys (which is the definition of Boyce-Codd normal form: all determinants are candidate keys).

Each time a relation is split into more than one table in a database, however, additional disk input/output and processing is required to join the information together for analysis or presentation. This additional overhead is acceptable where data integrity is an issue, but there are some cases where the additional cost of normalization (in terms of performance) is not warranted by the gains in data integrity. In these cases, relations are intentionally **Denormalized**, or left in a single-relation form even if it results in a lower normal form. Denormalization is a design trade-off between performance and integrity, and its usefulness will depend upon how the database will be used. In general, denormalization should be used sparingly due to the potential for problems (anomalies) in a denormalized database.

There are some fairly clear cases, though, where denormalization makes sense. For example, consider a very simple table with the following fields:

CUSTOMERS

CustomerID	CustomerName	City	Country	PostalCode
------------	--------------	------	---------	------------

There will naturally be a relationship between the customer and the location of the customer. This makes sense. However, the internal dependency  $\text{PostalCode} \rightarrow (\text{City}, \text{Country})$  means we do not have a single concept in this table. We have the possibility of our three anomalies:

- Insertion: we cannot insert a PostalCode and the City/Country in which it lies without a customer in that location
- Deletion: if we delete a customer, we lose what City/Country a PostalCode lies in.
- Modification: if we change the City or Country for a PostalCode in one place but not another, there is the possibility of conflicting values.

If we were seeking purity of design at all cost, we would break our original relation into two relations:

CUSTOMERS

CustomerID	CustomerName	PostalCode
------------	--------------	------------

POSTALCODES

PostalCode	City	Country
------------	------	---------

In this way, we can manage all Postal Codes and their locations (City/Country) in one relation, and the Customers and their locations in another. This is the theoretically correct implementation.

In most cases, though, a massive list of all Postal Codes and the City/Country in which they lie has no real value to an organisation. All they really need is the table of Customers and where the Customers are. The increased overhead of maintaining this list of Postal Codes, plus the increased processing and I/O required for the database to manage the split tables gives us no real

value in return. The insertion and deletion anomalies are irrelevant. For example, if we lost a Postal Code and the City/Country it was in, it has no effect on our business. We are probably willing to live with the possibility of modification anomalies as a side effect of gaining efficiency by putting all these attributes in a single table.

In this case, it makes sense to leave this structured as a single relation. The integrity gains are not worth the overhead of breaking it into two relations. In this case we will probably make a design decision to denormalize this relation.

## References

- Blaha, M., and Premerlani, W. *Object-Oriented Modeling and Design for Database Applications*. Prentice-Hall, 1998.
- Date, C.J. *An Introduction to Database Systems*. Addison-Wesley, 2000.
- Date, C.J. *The Database Relational Model: A Retrospective Review and Analysis*. Addison-Wesley, 2001.
- Elmasri, R., and Navathe, S.B. *Fundamentals of Database Systems*. Addison-Wesley, 2000.
- Kroenke, D.M. *Database Processing Fundamentals, Design and Implementation*. Prentice-Hall, 1998.
- Naiburg, E.J., and Maksimchuk, R.A. *UML for Database Design*. Addison-Wesley, 2001.
- Rob, P., and Coronel, C. *Database Systems: Design, Implementation and Management*. Thomson Learning, 2000.
- Satzinger, J.W., Jackson, R.B., and Burd, S.D. *Systems Analysis and Design in a Changing World*. Thompson Learning, 2000.
- Shekhar, S., and Chawla, S. *Spatial Databases: A Tour*. Prentice-Hall, 2003.
- Zeiler, M. *Modeling Our World*. Redlands, CA: ESRI Press, 1999.

## 2 Implementation

This module addresses the transition from theory to practice. In previous material, we learned that Class Diagrams are a mechanism for creating a conceptual model of what we need to store, and understanding Keys and Normalization let us know if we've got a good relational data model. In this module, we'll look at how to implement such data models.

Topic 1 looks at how we create database tables from a class diagram, including the notion of using keys to manage the relationships between tables. Topic 2 looks closely at the data structures used by the ArcMap application and its predecessors. Topic 3 looks in detail at the geodatabase model, including the various elements which help us model spatial and non-spatial relationships. The final topic briefly discusses the process of creating a new geodatabase, from its logical design, to a complete, useable database.

### Module Outline

- Topic 1:      Implementing Class Diagrams
- Topic 2:      ESRI Data Structures
- Topic 3:      Geodatabase Elements
- Topic 4:      Implementing a Geodatabase

There are a number of implementations of databases which store spatial data, including Oracle Spatial, IBM DB2 Spatial Extender and Informix Spatial DataBlade. To illustrate the concepts of a geodatabase implementation we will focus on the geodatabase implementation developed by Environmental Systems Research Institute (ESRI) for use with the ArcMap GIS application. This implementation was chosen because the practical component of this course will utilise the ArcMap application, and because the ESRI geodatabase is legitimately one of the leading implementations of a spatial database both in terms of the number of users making use of the technology and the rich set of capabilities available to users.

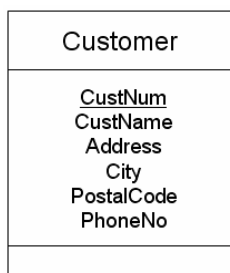
## 2.1 Implementing Class Diagrams

We've been talking so far in this course about the theoretical part of database design. Now that we know how to specify a database model, and how to tell whether it is a good one, we can now move on to actually implementing the model as real relations (tables) in a relational database. This topic will explore how the Class Diagram elements such as object classes and associations become relations. In addition, we will explore the use of keys in these new relations to manage the associations we wish to implement.

### 2.1.1 Implementing Classes

Translating the classes in our UML Class Diagrams into database table definitions is straightforward. Every class in our diagram becomes a table in our database, and each property of the class becomes a field in the table. Generally, the name of the table would be the same as the name of the class.

For example, consider the CUSTOMER class defined below. Note that we have amended the notation slightly, so that the key of the class is underlined. This will allow us to implement the associations later.



To implement this class, we simply create a table with fields as defined by the class properties. The notation we will use for defining the final database relations/tables appears as follows:

CUSTOMER( CustNumber, CustName, Address, City, Country, PostalCode, PhoneNumber)

This notation is similar to the Data Definition Language (DDL) definitions of tables one might use in Oracle, SQL Server and others, but it omits the data types and constraints necessary to complete the creation of the table. The following would be the DDL necessary to create the CUSTOMER class above in Oracle RDBMS.

```
/* Table Definition */
CREATE TABLE CUSTOMER (
    CUSTNUM NUMBER(8),
    CUSTNAME VARCHAR2(30),
    ADDRESS VARCHAR2(50),
    CITY VARCHAR2(20),
    POSTALCODE VARCHAR2(6),
    PHONENO NUMBER(10)
);
/* Primary Key Constraint */
ALTER TABLE CUSTOMER (
    CONSTRAINT CUST_PK
    PRIMARY KEY (CUSTNUM)
);
```

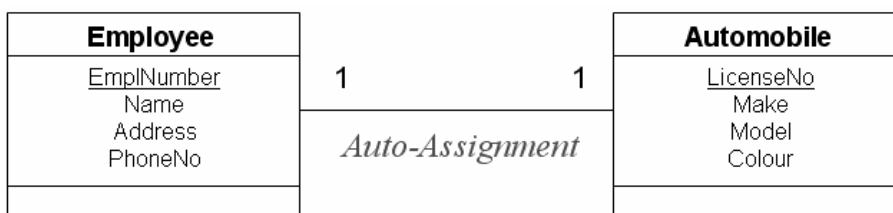
We will use the simplified notation rather than the DDL for two reasons:

1. The goal at this point in the course is to understand how tables are created and related to each other using keys, rather than involving ourselves with the detail of data types and constraints.
2. The physical details of a database implementation including data types and constraints are dependent on the RDBMS application you will be using. For example, a field containing string values (like CustomerName) would be a VARCHAR2 field in Oracle, an NVARCHAR field in SQL Server, and a TEXT field in MS Access. At this point we want to keep the database abstracted enough that we are not dependent on a specific database.

While implementing Classes is a simple process, translating the associations from a class diagram to the table definitions is slightly more complex. Each type of association (1:1, 1:M and M:M) must be implemented in a slightly different manner. We will examine each of these in succession in the sections which follow.

### 2.1.2 Implementing One-to-One (1:1) Associations

As an example of a 1:1 association, we will revisit the Employee-Automobile association we discussed in Module 1 of this course. Here we are depicting the relationship where each employee in an organisation is provided a vehicle for his/her use. This is a 1:1 relationship, where automobiles are not shared among employees.



**Figure 2-1 Auto-Assignment Association**

To create the tables for this association, we first create two relations, EMPLOYEE and AUTOMOBILE. We then place the key from one of the tables into the other table as an additional field to allow us to tell which car goes with which employee. This is called a **Foreign Key**. In the examples which follow, foreign keys are *italicized*.

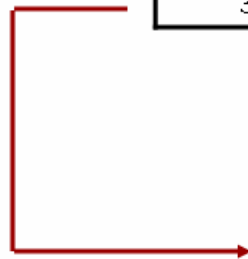
For 1:1 associations, we can have two possible implementations. We could take the key from the Employee table (EmplNumber) and place it in the Automobile table, or we can take the key from the Automobile table and place it in the Employee table, as shown below:

EMPLOYEE(EmplNumber, Name, Address, Phone)  
AUTOMOBILE(LicenseNo, *EmplNumber*, Make, Model, Colour)

**OR**

EMPLOYEE(EmplNumber, *LicenseNumber*, Name, Phone)  
AUTOMOBILE(LicenseNo, Make, Model, Colour)

Both of these two implementations are correct for a 1:1 relationship. 1:1 relationships are the only type of relationship where there are two equally valid representations. The latter of the two implementations might look like the tables in Figure 2-2, below, when data are loaded in physical tables.



<u>EmpIDNumber</u>	<u>LicenseNo</u>	Name	Address	PhoneNo
1	4658 BW	Jones, D.	123 Main St	555-1234
2	9231 PS	Bunyan, P.	456 Elm St	555-4321
3	1763 PS	Gretzky, W.	678 1 <sup>st</sup> Ave	555-0099

<u>LicenseNo</u>	Colour	Make	Model
4658 BW	Red	Chevy	Blazer
9231 PS	Black	Cadillac	DeVille
1763 PS	Blue	Toyota	Forerunner

**Figure 2-2 One Possible Auto-Assignment Implementation**

In this case, the EMPLOYEE table has been given the extra field, the foreign key *LicenseNo*. When the RDBMS application retrieves data from these two tables, it uses the foreign key to “look up” values in the related table. For example, if we wished to determine what sort of vehicle is assigned to Wayne Gretzky, we find the record for W. Gretzky in the Employee table. From this table, we extract the foreign key *LicenseNo*, with a value of “1763 PS”. We can then find this value, 1763 PS, in the Automobile table. From this we determine that W. Gretzky drives a Blue Toyota. In this manner the foreign key allows us to make the connection between the two tables.

As discussed, both representations are functionally equivalent, but if you expect to do a certain type of search more frequently, it may dictate which of the two possible representations would be best for performance reasons.

Table 2-1 shows a comparison of the two representations and the resulting retrieval operations based on the type of retrieval necessary.

**Table 2-1 Comparison of 1:1 Representations**

	First Representation (AUTOMOBILE has both keys)	Second Representation (EMPLOYEE has both keys)
<b>Have the car, want the employee</b>	<ul style="list-style-type: none"> <li>Take license number, look up that row in AUTO and read the EmployeeNumber</li> <li>Look in the EMPLOYEE table to find that employee</li> </ul>	<ul style="list-style-type: none"> <li>Take the license number, look directly in the EMPLOYEE table, read the employee information</li> </ul>
<b>Have the employee, want the car</b>	<ul style="list-style-type: none"> <li>Look in the AUTO table for the employee Number, and read the car information</li> </ul>	<ul style="list-style-type: none"> <li>Take the Employee number, look up that row in the Employee table and read the License Number</li> <li>Look in the AUTO table for that License Number, read the car information</li> </ul>

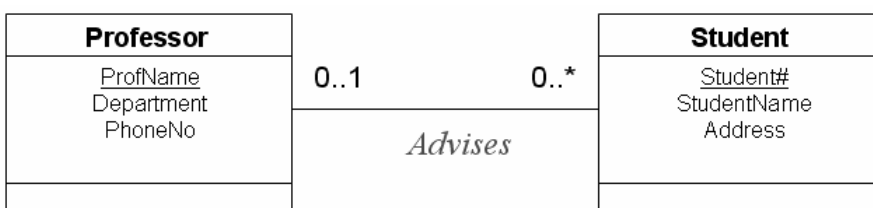
For example, if you expect to try to find a car given a specific employee most often, it makes sense to use the first representation (AUTOMOBILE has both keys) because it takes half as many “reads” from the tables.

### 2.1.3 Implementing One-to-Many (1:M) Associations

Converting a 1:M association to tabular form is more straightforward than 1:1 associations simply because there is only one correct implementation. In this case we need not consider the most common type of transaction in order to decide how to represent the tables.

When implementing a 1:M, we simply take the key from the “1” side of the association, and place it in the “Many” table as a foreign key. The “1” side is sometimes referred to as the “Parent” table, and the “M” side may be referred to as the “Child” table. You may recall this terminology from the description of the Hierarchical Database Model. The “Child” always gets the additional field (by adding the foreign key) when we create the table structure.

For example, consider the Professor-Student association shown in Figure 2-3. This relationship might depict that of a student to his/her advisor for a major project. In this case, a student may have at most one faculty advisor, while a professor may advise several students.

**Figure 2-3 Professor-Student Association**

As with 1:M associations, we begin by creating two tables to represent the two classes, and by adding the necessary fields to represent the class properties above. We then take the key of the Parent class (Professor; the “1” side of the association) *ProfName* and placing it in the Child class (Student; the “M” side of the association) as a foreign key field. The resulting relations would appear as shown below:

PROFESSOR(ProfName, Department, PhoneNo)  
 STUDENT(Student#, *ProfName*, StudentName, Address)

In physical tables, the result might look like as shown in Table 2-2, with STUDENT having the new foreign key field. One can see that the 1:M is accomplished because the *ProfName* (e.g., Michael) can be repeated within the STUDENT table, indicating that Michael advises several students.

**Table 2-2 Professor-Student Implementation**

PROFESSOR			STUDENT			
<u>ProfName</u>	Department	PhoneNo	<u>Student#</u>	<i>ProfName</i>	StudentName	Address
Michael	GEOG	9876	199701442	Michael	Ann	123 Main St
Thomas	CSC	1234	199983451	Michael	William	456 Elm St
James	GEOG	5542	199505432	Thomas	Albert	678 1st Ave

### 2.1.4 Implementing Many-to-Many (M:M) Associations

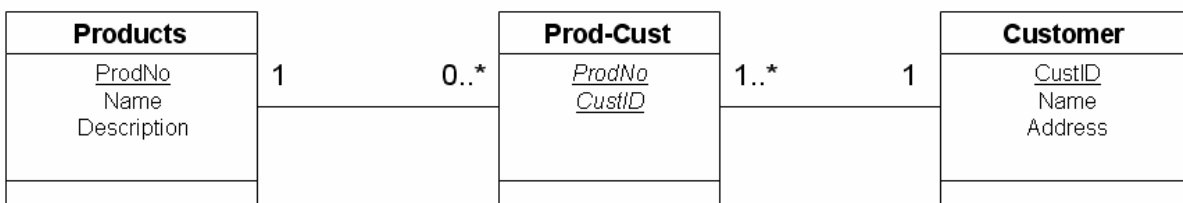
Many-to-many relationships cannot be directly represented with relations the same way 1:1 and 1:M relationships are. Some people like to consider that M:M relationships don't exist in physical tables at all (only conceptually in our Class Diagram), but that it is actually two 1:M relationships. In order to represent a M:M association, we will need to create a new relation (table) to represent the relationship itself.

For example, consider the Product-Customer association depicted in Figure 2-4. This association seeks to depict which products were purchased by which customer. Clearly, one product may be purchased by many customers, while a given customer may purchase several products.



**Figure 2-4 Product-Customer Association as Modeled**

To implement this association as relations, we will need to rewrite this as two 1:M associations, with a new class representing the association itself. The result appears as shown in Figure 2-5.



**Figure 2-5 Product-Customer Rewritten as Two 1:M Associations**

Both keys from the “1” tables (ProdNo and CustID) get populated to the “M” side of the relationships, and are placed in the new Prod-Cust table as foreign keys. The key of the Prod-Cust table is (ProdNo, CustID), since it takes both to identify a row. In the Prod-Cust table, both fields are part of the composite key, and both are foreign keys. This new table is sometimes called an Intersection Table, a Cross-Reference Table, or an Associative Table, depending on who you talk to. We’ll use the term **Associative Table**, since it matches our UML term Association

The resulting relations would appear as follows:

Products(ProdNo, Name, Description)

Prod-Cust(ProdNo, CustID)

Customer(CustID, Name, Address)

As physical tables, the result might appear as shown in Figure 2-6.

<u>ProdNo</u>	Name	Description
10035	Seat Cover	Bucket
26658	Air Freshener	Pine
6685	Tire	14R42
35891	Jack	2 Tonne

<u>ProdNo</u>	<u>CustID</u>
10035	1
10035	2
26658	2
26658	3
6685	1
6685	2

<u>CustID</u>	Name	Address
1	Jones, D.	123 Main St
2	Bunyan, P.	456 Elm St
3	Gretzky, W.	678 1st Ave

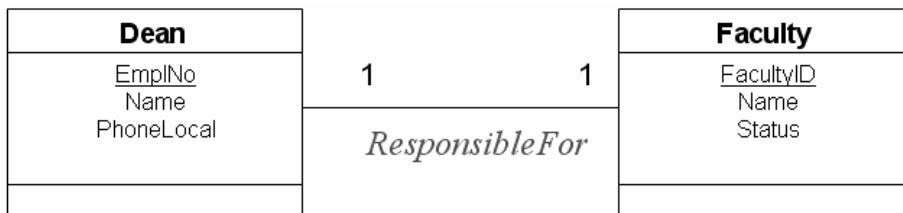
**Figure 2-6 Product-Customer Implementation**

### 2.1.5 Practice Exercises

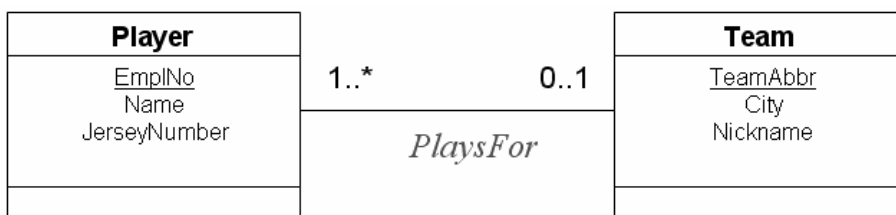
Try the following implementations on your own

Ensure that Keys are underlined and Foreign keys are in *italics*. Answers should include relations of the form: Relation(field1, field2, ...)

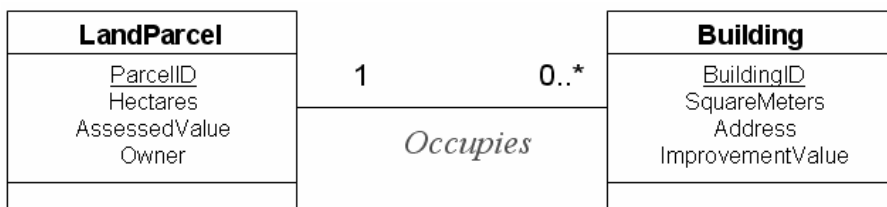
#### 1. Dean-Faculty



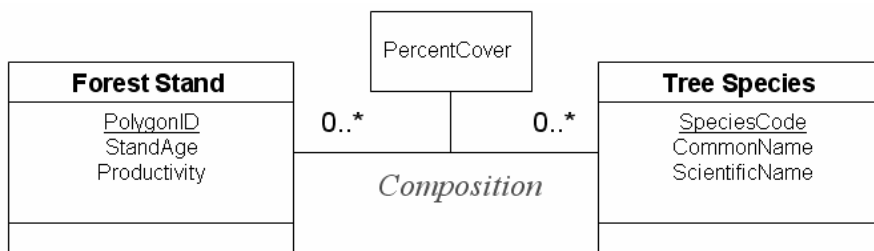
#### 2. Player-Team



#### 3. LandParcel-Building



## 4. ForestStand-TreeSpecies



## 2.2 ESRI Data Structures

### 2.2.1 Introduction

As we move from theoretical to applied and concrete, we should discuss the possible structures in the GIS we will be using. Before we can implement a geodatabase, we need the specific details and terminology of how a geodatabase works. We can then learn how to transfer objects and associations in a class diagram in to a functioning geodatabase.

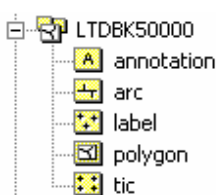
All GIS have specific data formats. During this course, you will be using ArcMap extensively, so it is important to understand the terminology and formats of the spatial data used by that application. Although the geodatabase is the primary focus of this course other formats such as coverages are still in use, so we will discuss them in a lesser level of detail. The material in this topic will review different types of spatial data, and the formats used to store them.

### 2.2.2 The Coverage

The **Coverage** is a vector data format developed around 1980 as the primary structure for Arc/INFO. This format stores spatial, attribute and topological information in a dual-architecture, or georelational structure. The spatial data are stored in binary files and attributes are stored in a table in the INFO database application.

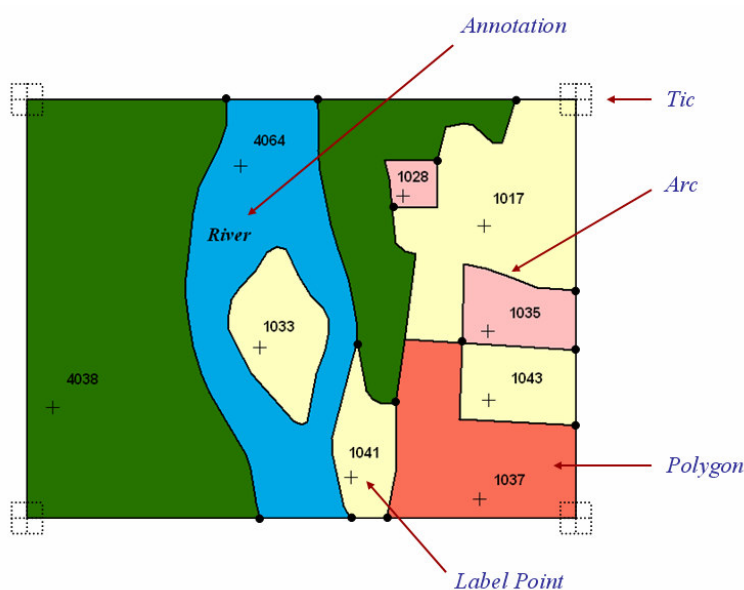
A coverage represents a single set of geographic objects, such as roads, parcels, soil units or forest stands. It is also multi-feature, or polymorphic, in that it stores several different **Feature Classes**, all relating to the same coverage. For example, a land use polygon coverage might include the polygons, the lines which form the perimeter of each polygon, the points which define the lines or line intersections, and other such information.

You can look at the different feature classes within a coverage using ArcCatalog, and you can add the classes for a coverage to a data view in ArcMap individually. Figure 2-7 shows the Lithuania 1:50,000 land use polygons (LTDBK50000) as a polygonal coverage. Each of the feature classes will be discussed in the material which follows.



**Figure 2-7 Polygonal Coverage as Viewed in ArcCatalog**

These feature classes correspond to different elements of the polygonal coverage. Figure 2-8 shows how the feature classes in the land use coverage appear when viewed graphically.

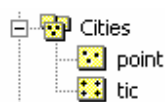


**Figure 2-8 Polygonal Coverage Viewed in ArcMap**

Different coverages will have different feature classes, depending upon what the coverage is meant to represent. For example, if the coverage is intended to store linear features such as streams or roads, there would be no Polygon feature class for that coverage. Figure 2-9 and Figure 2-10 show the feature classes present in coverages which store line and point data, respectively.



**Figure 2-9 Linear Coverage (Municipal Boundaries)**



**Figure 2-10 Point Coverage (Cities)**

There are other feature classes which may be stored in a coverage as well. The possible feature classes in a coverage and a brief explanation of each are summarized in Table 2-3, below.

**Table 2-3 Coverage Feature Classes**

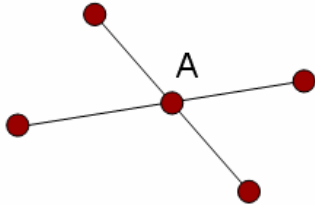
Feature Class	Purpose
<b>Point</b>	A point defined by an x,y coordinate pair used to represent a point feature.
<b>Arc</b>	A set of ordered x,y coordinates used to represent linear features and polygon boundaries.
<b>Node</b>	The endpoints of an arc or the point at which two or more arcs connect.
<b>Polygon</b>	A bounded area corresponding to a polygonal feature.
<b>Annotation</b>	A text string placed on a map, such as a geographic name.
<b>Tic</b>	A point used to georeference the coverage. Tics enforce the spatial referencing of a coverage.
<b>Route</b>	A linear feature composed of one or more arcs or parts of arcs. Routes allow the creation of complex linear features, for example all the tributaries in one river system being treated together as a single route feature.
<b>Link</b>	A series of point transformations used during rubber sheeting and adjustment.
<b>Region</b>	A polygonal feature composed of a series of one or more polygons. Regions allow the creation of complex polygonal features such as multi-part polygons (e.g., Indonesia) and overlapping polygons.

Coverages exist as a series of files on a storage device such as a hard disk. Each coverage has its own folder with the same name as the coverage, and an additional folder called *info* which stores the INFO database files for all the coverages in the same folder. A folder which holds one or more coverage is called a **Workspace**, and all coverages in the same workspace share the *info* folder. Since the files necessary for a given coverage are present in more than one folder in the workspace, users should not use operating system tools such as Windows Explorer to copy or move coverages as this may corrupt the data. Use ArcCatalog to move, copy or make any changes to coverages.

As you may have studied in previous courses, **Topology** defines spatial relationships between connecting or adjacent features in geographic data. In a coverage, the topological relationships are stored explicitly within the coverage files. For example, the coverage stores topological information such as how lines are connected to each other, which lines define certain polygons and which polygons are adjacent. This is why the various feature classes (e.g., points, lines, polygons) which form spatial features are all managed as discrete elements in a coverage. Creating and storing topological relationships facilitates many common analytical functions, such as modeling flow through the connecting lines in a network, combining adjacent polygons with similar characteristics or overlaying geographic features. Determining the topological relationships in the data and explicitly storing them allows such analyses to be performed much more quickly.

This highly-structured approach to data storage presents problems when representing some geographic phenomenon, however. In order to calculate and store the connectivity of line features, the coverage stores every line segment, between intersections with other lines, as a separate arc. For example, consider two linear features created in a coverage such as shown in Figure 2-11.

Even if the operator created these lines by entering two straight lines which cross (like drawing the character “X”), the coverage must create and store a new vertex at point A where the two lines cross. The result is four simple line features, all sharing the point A.



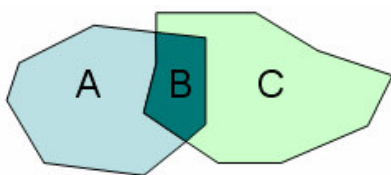
**Figure 2-11 Intersecting Arcs**

This may be useful for some applications, such as mapping underground infrastructure such as sewer pipes. In other situations, such as street networks, it may cause difficulties. There may be cases where an organisation wishes to represent all line segments which together form a given street as a single feature. In such a situation, we need the “street” feature to be made up of a series of segments which travel from intersection to intersection. We still need the lines to be broken when they intersect other roads (in order to allow network connectivity analyses), but we want the ability to manage several arcs as a single feature. This type of feature might be called a **Multi-Part Feature**.

To accommodate the use of multi-part linear features, coverages have a feature class called a Route, which allows the user to define new features composed of one or more simple arcs. These arcs may be connected in a single linear path, a series of connected branching paths, or may be entirely disjointed. Users may find that there is significant additional effort required to create and maintain these logical groupings of linear features.

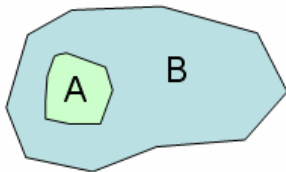
Polygon storage may also present problems in the coverage structure. Again, because the topology is calculated and stored explicitly in the data structure, any area which is enclosed within lines is considered a separate polygon. This may be acceptable for most applications, but there are some situations where this structure is inadequate.

Consider two polygons drawn such that they overlap in space, as shown in Figure 2-12. When two overlapping polygons are digitized, the coverage must store the result as three polygons. The overlapping area (labelled B below) must become a new polygon, and there becomes no true overlap between polygons A and C. These polygons might represent the home ranges of two bears, and a small portion of their home range is used by both bears. This cannot be accommodated with simple polygons in a coverage.



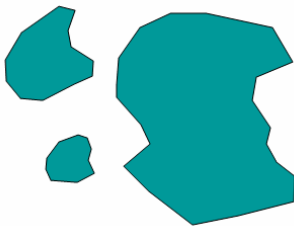
**Figure 2-12 Overlapping Polygons**

Another problem with polygons in a coverage is that they cannot represent void polygons, or areas within a larger polygon which have no value. For example, Figure 2-13 shows a situation where we are attempting to map water features. Polygon B might represent a lake that has an island in it (Polygon A), so we need to represent that the area of B is a water feature, but the area A is not a water feature. In a coverage, A becomes a separate polygon, and we must use attributes to denote that A is “not water”. Sometimes inexperienced users of coverages make mistakes such as simply calculating the area of all water features in a coverage, mistakenly adding the area of void polygons and overestimating the total area of water.



**Figure 2-13 Void Polygon**

As with linear features, there are times when it is helpful to accommodate multi-part polygons. Figure 2-14 shows an example of three discrete polygons. If we wish to present all three polygons as islands which together form a country, like the United Kingdom or Indonesia, we need to be able to use multi-part polygons.



**Figure 2-14 Multi-Part Polygon**

Coverages handle these complex polygon problems (overlaps, voids and multi-part polygons) using a Region feature class. Like a route, regions allow polygons to be combined into logical groupings for purposes such as previously discussed. As with routes, there is an additional level of effort required to create and manage region features.

Due to limitations imposed by the software of the time (e.g., the operating system or the INFO DBMS), several additional limitations are in effect regarding file and folder names. Users should bear the following restrictions in mind when using coverages.

- Due to limitations in the INFO DBMS, coverage names must be less than 13 characters in length, and can include letters, numbers, hyphens and underscores. No other characters may be used in the name.
- There cannot be spaces anywhere in the workspace path or coverage name. This means that you cannot, for example have a space in the path leading to a coverage. The path C:\Lab 6 Data\Lakes (with spaces) may result in ArcMap failing to read your coverage correctly.

Although ESRI has moved to the Geodatabase as its data structure, coverages are still widely used and distributed. As a result, students should be familiar enough with the structure to be able to manipulate and translate them when they are encountered. We will discuss the Geodatabase format later in this module. Since the newer ESRI product line (ArcGIS) is focused on the geodatabase, it has moved away from supporting the coverage format. Users can import, read and export coverages using ArcGIS applications, but coverages cannot be edited directly. To edit a coverage users must either export to a geodatabase or shapefile, or use the ArcEdit application.

### 2.2.3 Shapefile

While the coverage enforced rigorous controls over topological structuring of spatial data and allowed many analyses to be performed quickly, the complexity of the model meant that simple operations such as displaying the data were relatively slow. The Shapefile format was introduced in the mid-1990's with the ArcView 2.0 application, and was an attempt to create a simple format which could be read and displayed quickly.

As with the coverage, the shapefile is a vector format, and is georelational. Spatial data is stored in a proprietary binary format and attributes are stored in the dBase database format. A shapefile contains one feature type (point, line or polygon) per file. Shapefiles are probably the most common spatial database format today because the specification for the format is available and there are several free or inexpensive applications which can read shapefiles. In addition, most commercial GIS applications are able to read and import shapefiles.

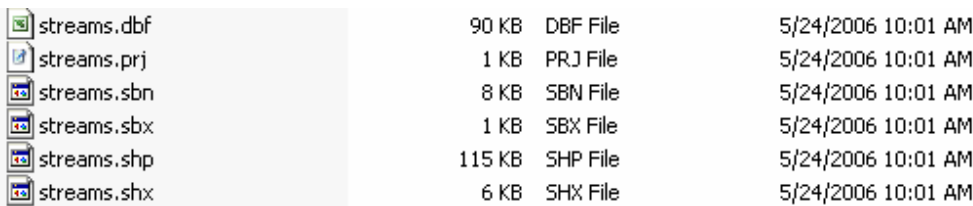
To simplify the storage of spatial data, shapefiles do not store or enforce topological relationships. As a result, there are some inefficiencies in storage such as shared polygon boundaries being repeated, but the coordinates defining a given spatial feature are self-contained. In this manner, the display of a given point, line or polygon does not involve reading information from several tables or records. Since shapefiles do not store topological relationships, they cannot ensure spatial integrity, such as the absence of gaps or overlaps in a cadastral parcel database. Topological relationships, such as shared lines or points, can be enforced during edit sessions at the application level, using **Map Topology** during an ArcMap edit session with specific topological editing tools, but the resulting spatial relationships are not stored and maintained in the shapefile itself.

Shapefiles exist as a group of files on a storage medium. A given shapefile, such as a set of roads, is not a single physical file as the name implies, but several separate files. The files all share the same filename, but have different file extensions. The essential three files are:

- SHP:** a binary file containing the x, y coordinates which define the spatial data.
- DBF:** a dBase file, and stores the attributes of the features. DBF is a common file format, and you can open this file using dbase, excel, etc. Attributes are related 1:1 to spatial objects.
- SHX:** an index file which makes finding related rows between the SHP and DBF faster.

Thus a shapefile which stores a set of roads would include at least three files, named Roads.SHP, Roads.DBF and Roads.SHX. There are 3 other files which may be present, depending upon how the shapefile has been used. If a spatial index is present for a shapefile, two additional files will be created: an SBN and SBX file. If a projection is defined for this shapefile, a PRJ file will also be

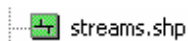
present. A shapefile which included all of these elements would include six files, all with the same filename. For example, Figure 2-15 shows how a shapefile containing stream lines might appear when viewed in Windows Explorer.



streams.dbf	90 KB	DBF File	5/24/2006 10:01 AM
streams.prj	1 KB	PRJ File	5/24/2006 10:01 AM
streams.sbn	8 KB	SBN File	5/24/2006 10:01 AM
streams.sbx	1 KB	SBX File	5/24/2006 10:01 AM
streams.shp	115 KB	SHP File	5/24/2006 10:01 AM
streams.shx	6 KB	SHX File	5/24/2006 10:01 AM

**Figure 2-15 Streams Shapefile in Windows Explorer**

Since shapefiles are discrete files, rather than a set of files stored in several folders, they can be moved or copied using an operating system tool such as Windows explorer. However, users must be sure to copy all files with the same filename or the shapefile may not function correctly. If an ESRI product such as ArcCatalog is used, all files relating to the single shapefile are managed for the user. For example, the same shapefile viewed in ArcCatalog appears simply as a single object, as shown in Figure 2-16.



**Figure 2-16 Streams Shapefile in ArcCatalog**

The shapefile format was created as GIS applications were moving to the desktop computer environment. As a result of the operating systems at that time, and of the dBase file format, there are many limitations imposed on the structure and naming of a shapefile. These limitations include:

- Filenames are limited to 8 characters in length. They must be an alphanumeric character (A-Z, 0-9) followed by 0-7 characters. Underscores ( \_ ) and hyphens ( - ) are the only non-alphanumeric characters allowed in the names.
- Field names are limited to 10 characters, and as with filenames, underscores and hyphens are the only non-alphanumeric characters permitted.
- Text fields may be a maximum of 255 characters in length. Very long text descriptions are not permitted.

Shapefiles are a simple format, designed for fast display and editing. However, the lack of structure to the spatial data may make the format more susceptible to problems. There is no empirical evidence to support this, but many years of personal experience using coverages and shapefiles have indicated that shapefiles are perhaps more susceptible to corruption and performance degradation than coverages, particularly when working with very large, complex datasets. Using current ArcGIS software, it may be more appropriate to move to the geodatabase format in such cases.

#### 2.2.4 Geodatabase

The geodatabase is the most recent structure used by ESRI software, and is actively being used and refined at present. A geodatabase is a collection of feature datasets (points, lines and polygons), tables, rasters and TINs. All spatial, attribute, tabular and topological data are stored in

a single relational database. One of the things that makes the geodatabase attractive is that unlike previous formats the geodatabase is not tied to a specific database application such as INFO or dBase. The geodatabase can be implemented using standard RDBMS applications such as Access, DB2, Informix, Oracle and SQL Server.

We earlier discussed the Object-Oriented data model, where data and behaviour were encapsulated in objects defined in the database. The geodatabase incorporates many aspects of the object-oriented model, including the notion of object behaviour. **Behaviour** refers to the ability of a Geodatabase to impose restrictions on the way a feature is created or maintained. These restrictions in turn ensure a level of data integrity which would previously be ensured manually. Below are examples of the types of behaviour which may be controlled in a geodatabase:

Domain Restrictions	Controls the allowable values for a feature attribute. For example, when you enter the land use of a new parcel of land, it must have a reasonable value, such as <i>Residential</i> or <i>Industrial</i> .
Topological Restrictions	Controls the spatial relationships between features. For example, you might wish to ensure that all building structures lie entirely within a land parcel (i.e., they cannot straddle a parcel boundary)
Geometric Restrictions	Controls the way features are spatially structured. For example, you might wish to ensure that building corners are always right (i.e., 90°) angles.
Relationship Restrictions	Controls the way features are related to one another. For example, when you delete or move a water pipe, all the valves associated with the pipe are deleted or moved as well.

The geodatabase can be considered an object-relational implementation. It relies heavily on a relational database for disk-based storage, query processing, security and transaction processing, but the GIS application provides functions relating to more object-oriented elements such as the behaviour attached to geographic objects and the management of associations and subtyping between object classes. In essence, the storage and retrieval of information is managed by the RDBMS and the semantics of the data are handled by the application.

There are three types of geodatabases for the 9.2 release of ArcGIS:

- Personal Geodatabase
- File Geodatabase
- ArcSDE Geodatabase (SDE stands for Spatial Database Engine)

The Personal Geodatabase is stored in an MS Access database. In this structure, all geodatabase elements are stored in a single Access database file (MDB file). The size of a personal geodatabase is limited to the 2 Gb limit imposed by Access, but ESRI documentation indicates that performance degrades significantly when the database size reaches between 250 and 500 Mb. The personal geodatabase supports several concurrent users performing read-only tasks, and a maximum of one user writing to the database. They support the full information model of geodatabase (i.e., behaviour, relationship classes, raster catalogs, etc.) and do not require any investment in database application software.

The File Geodatabase is a new structure which builds on the personal geodatabase. It also supplies a widely available, free and simple solution for many geodatabase applications. In addition, it removes the size limitations imposed by Access (file geodatabases have a 1 Terrabyte limit) and improves the performance of the database. As with the personal geodatabase, file geodatabases do not require that the user invest in significant RDBMS software. Data are stored in several files stored within a single folder on a storage medium.

File geodatabases again allow several people to read from the same database, and slightly improve the multi-user support for several people concurrently writing to the database. File geodatabases allow one person *per feature dataset* to edit, rather than the one person *per database* allowed in a personal geodatabase. For example, the file geodatabase would allow one person to edit road features, and another to edit parcel boundaries, even if both feature classes were in the same file geodatabase.





An ArcSDE geodatabases store their data within a standard RDBMS such as DB2, Oracle, Informix or SQL Server. They are primarily used in larger organizations with a large number of users, and they take advantage of the underlying RDBMS architecture to manage the data security, backup and integrity. ArcSDE geodatabases can support extremely large databases and a large number of concurrent users.

### 2.2.5 The Grid

A Grid is a data format analogous to a coverage, except used to store raster data. It was also developed in the 1980's as part of the Arc/INFO application. It has the ability to store discrete attribute values, which are stored as integers, or continuous values, stored as floating point grids. Integer grids may use a Value Attribute Table (VAT) to manage non-numeric values such as land uses or soil types.

A grid is stored in a manner similar to a coverage. It exists as a file folder named the same as the grid, and stores a number of files within the folder. Grids use the same workspace concept as coverages, so there will be an additional folder called *info* in the same folder as the grid folder. There may also be an AUX file for each grid, containing information such as a colormap, coordinate system, projection and statistics relating to the grid contents. This file will be created automatically the first time statistics are required to perform a task. There may also be a RDD (reduced resolution dataset) or Pyramid file for each grid. Pyramid files store reduced resolution versions of the grid which allow the speed of display at varying scales to be vastly improved.

For example, a grid called Elevation might appear to Windows Explorer as shown in Figure 2-17.

 Elevation	File Folder	6/20/2007 1:51 PM
 info	File Folder	6/20/2007 1:51 PM
 Elevation.aux	7 KB AUX File	6/20/2007 1:51 PM
 Elevation.rdd	366 KB RRD File	6/20/2007 1:50 PM

**Figure 2-17 Elevation Grid Viewed in Windows Explorer**

As with coverages, users should not use operating system tools such as Windows Explorer to move or copy grids, since information for a given grid is stored in several folders. ArcCatalog will manage all of these files, and should be used to copy or move a grid. In addition, grids are subject to the same INFO limitations as coverages: grid names are limited to 13 characters in length, and

cannot use non-alpha-numeric characters (underscore and hyphen are the exceptions) in either the name or workspace path.

### 2.2.6 Rasters in a Geodatabase

With the introduction of the geodatabase structure, raster data were integrated into the larger geodatabase framework, rather than treated as an entirely separate data structure from related vector data. Allowing the geodatabase to manage raster datasets will, in many cases improve performance. It also brings the strengths of the RDBMS environment such as security, multi-user support and error-recovery into play with raster data.

There are three ways in which rasters can form part of a geodatabase:

- A **Raster Dataset** is a single, mosaicked raster stored within the geodatabase.
- A **Raster Catalog** is a means of managing a set of discrete rasters, which may overlap or be discontinuous.
- A **Raster Attribute** is a means of storing a raster or image as an attribute of a feature.

A raster dataset combines several adjacent rasters into a single, seamless raster. The component rasters need to be homogenous in resolution, format and data type. Component rasters may be modified when added to a raster dataset; overlapping pixels will be resolved into a single set of pixels, and the pixel values may be resampled if resolutions or grid orientations do not match the destination raster dataset. Since the rasters being added to the dataset may need to be altered, updating a raster dataset can be very slow. Once created, however, a raster dataset is very fast to load and display. Very large rasters can be effectively used when stored in a raster dataset.

A raster catalog is essentially a means of managing available rasters. Rasters that form part of a raster catalog can be heterogenous in resolution, format and data type. Spatially, component rasters may overlap or be separated. Raster catalogs can make managing numerous rasters far easier. One example of the application of a raster catalog would be the storage of time-series rasters. One could more easily sort through and work with several images, for example, which all overlap in space, but which were taken over a period of time. In this way, users can quickly locate and view the time period of interest.

Raster catalogs are comparatively faster to create, since they do not require the significant processing of mosaicking the images and reconciling overlapping areas. They may be slower to display than a raster dataset, when many rasters are present, but users may view the available rasters as a “wire-frame” polygon which simply shows the spatial extent of available imagery or thematic rasters. Metadata can be managed for the entire catalog, and for individual component rasters.

Attribute rasters allow rasters to be stored directly in the attribute tables for features. A common use of this functionality would be to store a photograph as an attribute of a spatial feature. A building polygon might have a photograph of the home stored in the table. Photographs of a street reconstruction project might be attached to the street segment in question.

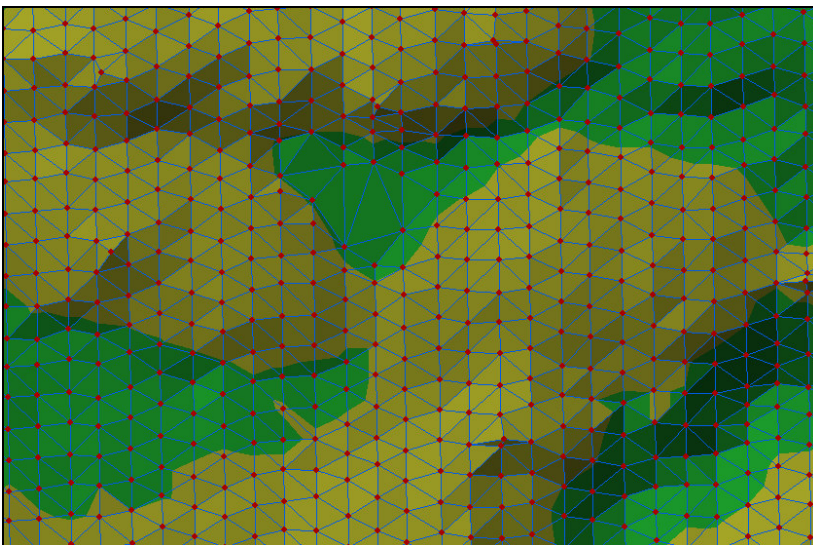
With all raster implementations, the raster data may be stored within the geodatabase (called a Managed Raster), or may stored outside the RDBMS in their native form and a reference to the

external raster maintained within the database (called an Unmanaged Raster). The physical storage of managed rasters varies depending upon the type of geodatabase in use. Managed rasters in a personal geodatabase are stored in a folder which resides next to the MDB file, and standard file formats are used (e.g., ERDAS IMAGINE, JPEG). File geodatabases store managed rasters in a proprietary format within the file geodatabase folder. ArcSDE databases always use managed rasters, and the data are stored directly within the relational database structure.

As with vector data, the size limits for rasters depend upon the type of geodatabase in use. Personal geodatabases are limited to 2 GB, each raster in a File geodatabase is limited to 1 TB, and ArcSDE geodatabases place no limits on raster size.

### 2.2.7 Triangulated Irregular Networks (TINs)

A Triangulated Irregular Network (TIN) is a dataset specifically used to describe surfaces, like elevation. Figure 2-18 shows an example of a TIN. Like a coverage, a TIN is stored as a folder of files, but unlike a coverage it has no associated INFO files. A TIN directory contains 7 files which describe the surface, encoded in binary format.



**Figure 2-18 Example TIN**

TINs are composed of:

- Nodes:** these are the source of the Z-coordinates used in creating the TIN. Nodes correspond to point features or points on linear features used to create the TIN. They form the intersections of triangles.
- Edges:** Edges join nodes to their nearest neighbours, and form the sides of the triangles in a TIN.
- Triangles:** these polygons form the facets of the TIN. Since the coordinates of each of the three corners of the triangle are known, characteristics such as slope, aspect and area may be determined for each triangle.

Also like a coverage, the TIN contains topology information explicitly stored in the file structure. It calculates and stores, for each triangle, the adjacent triangles, the edges which form the triangle, and the nodes (including Z-coordinates) of the three corners.

### **2.2.8 Terrain in a Geodatabase**

The increasing use of very high-resolution terrain data, such as Light Detection and Ranging (LiDAR), has meant that static-resolution representations such as TINs quickly become unwieldy at small scales when examining large areas (requiring large data volumes). For use with the geodatabase storage structure, the Terrain representation was developed. Terrains are a TIN-based structure with variable resolution.

Terrains are stored with a series of feature classes, such as mass elevation points and breaklines, and a set of rules which define how these features should be used to create a complete surface. Rules are also defined which dictate how the participating feature classes should be integrated at a variety of scales. For example, at very small scales it may not be necessary to perform processing required to integrate minor breaklines which will not visibly affect the terrain at that scale. In this case, the breakline features would participate in the terrain representation only at specified, larger scales. Depending upon the necessary level of detail for the given scale, terrains also make use of more or less of the mass points (and thus have more or less resulting triangles) depending upon the display requirements. In this way, terrains made up of billions of mass points can be quickly displayed at a variety of scales.

The terrain does not actually store the completed surface representation. Rather it calculates a TIN representation by referencing the appropriate source features for the current display scale. Thus a TIN created from a terrain structure is calculated on-the-fly and never actually stored permanently.

Storage limits for terrains are similar to those for other data formats (e.g., vector or raster data), and depend again on the type of geodatabase involved. ESRI literature indicates that a personal geodatabase terrain is limited to about 20 million points or less, file geodatabases are limited to hundreds of millions of points, and ArcSDE geodatabases are may be of unlimited size, subject to media limitations only.

## 2.3 Geodatabase Elements

### 2.3.1 Introduction

In the previous topic, we discussed in general terms what a geodatabase is, and how the data in a geodatabase is stored and accessed. In this topic, we will address the elements within the geodatabase model, and how these elements may be used to structure and manage behaviour of geographic data. The major elements of a geodatabase are summarized below, and these will be discussed in detail in the sections which follow:

1. Objects
2. Object classes
3. Features
4. Feature classes
5. Feature datasets
6. Relationships
7. Relationship classes
8. Domains
9. Subtypes
10. Topology

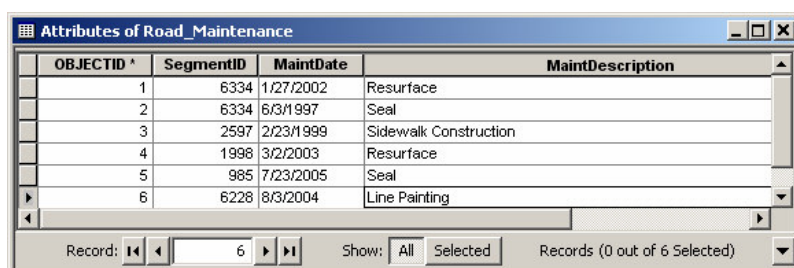
Some additional highly specialized geodatabase elements, such as Raster Datasets, Geometric Networks, Terrain Datasets, and Cartographic Representation are covered in other related courses, such as GII-04, GII-06.

### 2.3.2 Data Structure Elements

#### Objects and Object Classes

Object classes are tables in a geodatabase storing non-spatial data (e.g., Parcel owners). Objects are instances of an object class that have the same properties and behavior. Properties are stored in the table as attributes, while behaviour is implemented as a set of validation rules using elements such as domains or topology rules. We will discuss these in the sections which follow. As with our Class Diagram terminology, an Object is an individual (e.g., John Smith) within the Object Class (e.g., all parcel owners). Objects can be related to other objects via relationships.

When viewed in ArcMap, an object class will have no SHAPE field, or no attribute of type Geometry in which to store a spatial representation. Figure 2-19 shows an example of an object class table as seen in the ArcMap environment.



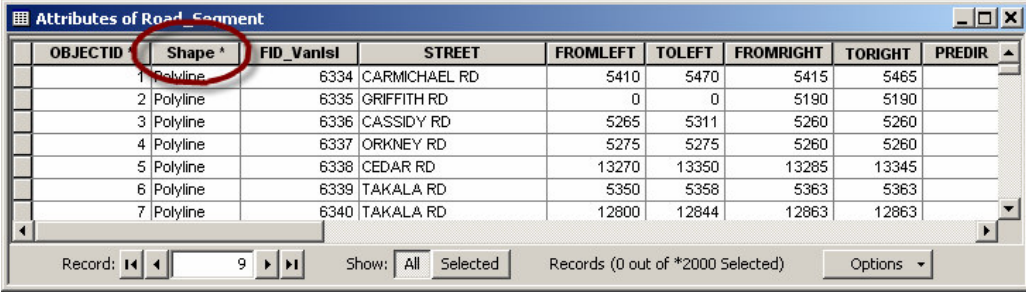
OBJECTID *	SegmentID	MaintDate	MaintDescription
1	6334	1/27/2002	Resurface
2	6334	6/3/1997	Seal
3	2597	2/23/1999	Sidewalk Construction
4	1998	3/2/2003	Resurface
5	985	7/23/2005	Seal
6	6228	8/3/2004	Line Painting

**Figure 2-19 Example Object Class Table**

## Features and Feature Classes

Features are objects with spatial characteristics that represent a real world object in a layer on a map, such as a land parcel polygon. Feature classes are collections of features with same type of feature geometry (e.g., Points, Lines or Polygons) and attributes. Again, the feature class might be all roads, a feature within that class would be a single street. We can also relate feature classes to other object or feature classes using relationships. A feature class is conceptually similar to a shapefile.

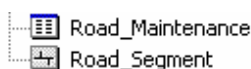
When viewed from within ArcMap, a feature class attribute table can be seen to contain a field named *Shape*. This indicates that features in this table have a spatial representation, and may be drawn on a map using point, line or polygon symbology. Figure 2-20 shows an example of a feature class attribute table as seen in ArcMap.



OBJECTID	Shape	FID_Vanisl	STREET	FROMLEFT	TOLEFT	FROMRIGHT	TORIGHT	PREDIR
1	Polyline	6334	CARMICHAEL RD	5410	5470	5415	5465	
2	Polyline	6335	GRIFFITH RD	0	0	5190	5190	
3	Polyline	6336	CASSIDY RD	5265	5311	5260	5260	
4	Polyline	6337	ORKNEY RD	5275	5275	5260	5260	
5	Polyline	6338	CEDAR RD	13270	13350	13285	13345	
6	Polyline	6339	TAKALA RD	5350	5358	5363	5363	
7	Polyline	6340	TAKALA RD	12800	12844	12863	12863	

**Figure 2-20 Example Feature Class Table**

In ArcCatalog, Object Classes and Feature Classes appear very different. For example, Figure 2-21 shows a non-spatial object class called *Road\_Maintenance*, and a line feature class *Road\_Segment*. Users may easily distinguish between feature and object classes by the icon used to represent it.



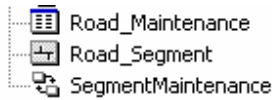
**Figure 2-21 Feature and Object Classes in ArcCatalog**

## Relationships and Relationship Classes

Relationships are an association between two or more objects in a geodatabase, and they may involve spatial objects (features) or non-spatial objects (rows in tables). They are the geodatabase equivalent of a class diagram association, and are implemented in a relationship class. Relationship classes serve as a permanent join between objects or features in our geodatabase, such as between a parcel of land and the parcel owner(s).

Like our class diagram associations, relationship classes have multiplicities, so you can indicate whether the relationship is 1:1, 1:M or M:M. For M:M relationships, the relationship class creates and maintains the new associative class which represents the relationship itself. Relationships may also trigger actions in the database, such as a cascading delete (remove all associated valves when a water pipe is deleted, for example), moves to follow a related move or custom behaviour.

Relationship classes appear as a separate element in the geodatabase, and its properties dictate the nature of the association, and its participating object and/or feature classes. Figure 2-22 shows a relationship class which associates road segments to maintenance performed there.



**Figure 2-22 Relationship Class in ArcCatalog**

The three elements we have so far discussed, object classes, feature classes and relationship classes, represent essentially the building blocks of our UML class diagrams. Table 2-4 summarizes the equivalency between relational terminology and geodatabase terminology.

**Table 2-4 Comparison of Geodatabase and Relational Concepts**

Geodatabase Element	Relational Equivalent
Attribute	Column, Field
Object	Row
Object Class	Table
Feature	Row with spatial attributes
Feature Class	Table with spatial attributes
Relationship	Association between two rows
Relationship Class	Association between two tables

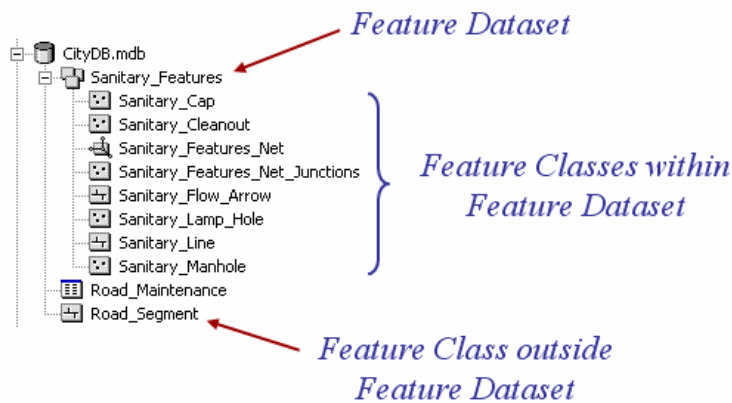
### Feature Datasets

A feature dataset is a “container” for related feature classes. They can be used to logically group feature classes that have some relationship to one another. This relationship could be a spatial or thematic relationship. Feature classes within a feature dataset must have the same spatial reference (coordinate system, extents, etc.).

Feature datasets are necessary if you wish to create a geodatabase terrain. Terrain datasets, as we have discussed previously, can contain several feature classes which participate in the representation of a 3-D TIN. For example, there might be point feature classes representing elevation spot heights and lines representing hard or soft breaklines. All of these feature classes together help define the resulting terrain. In order to make use of the terrain structure, all participating feature classes must reside in the same feature dataset. The same applies to topology rules, network datasets and geometric networks, in that all participating feature classes must be present in the same feature dataset.

Feature datasets may also be used to control permissions for data editing. If permissions are set for a feature dataset, all feature classes within it will share the same access privileges. Using this mechanism, geodatabase administrators may create groups of feature classes to be maintained by different user groups and aggregate them using feature datasets. This may reduce significantly the effort required to maintain several different sets of access rights.

In ArcCatalog, a feature dataset is clearly identified with a different icon, and all member feature classes are indented to indicate their membership in the feature dataset. Feature classes may be unaggregated in a geodatabase, or may be stored within a feature dataset. Figure 2-23 shows an example with both situations, as seen in ArcCatalog.



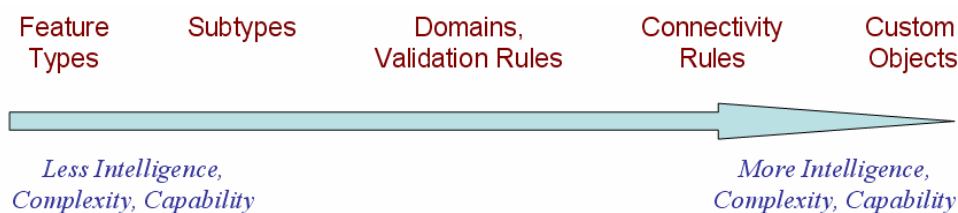
**Figure 2-23 Feature Dataset**

## Spatial References

A spatial reference is a means of relating objects in our geodatabase to real locations on the earth. In order to integrate all the geodatabase feature classes we must define for each the coordinate system. The spatial reference for a feature class includes the coordinate system plus the spatial domain. A spatial domain simply defines a spatial extent in x, y and z that all features must lie within. As discussed, all feature classes in a feature dataset must share the same spatial reference.

### 2.3.3 Behaviour Elements

Geodatabase behaviour provides a means of controlling data quality and speeding up the process of creating and editing data. There are a number of tools and elements available to assist in this, and they vary greatly in complexity and capabilities. The simplest way to control how a feature can exist in our database is by the way the feature class is defined, including its geometric type (point, line, polygon) and its available attributes. The most complex way of managing the way features are permitted to exist is to write programming code which can control in very specific ways how each feature is allowed to exist and relate to other objects. Managing behaviour can be thought of as a series of tools, each of which progressively adds intelligence to geodatabase objects, as shown in Figure 2-24.



**Figure 2-24 Continuum of Behaviour Capability**

### Subtypes

Subtypes are a categorization of feature classes that allows users to make distinctions between features, or to categorize features, without creating new feature classes. Conceptually, this is the geodatabase equivalent of the inheritance association we discussed in the UML Class Diagrams topic. For example, we might create a feature class *WaterPipes*, which would represent the superclass, and several subclasses to represent different types of water pipes, such as *Concrete*, *Steel* and *Plastic*. Each subtype might have different valid diameters or pressure ratings. The subtypes allow us to refine the type of pipe without having to have three separate feature classes.

There is a difference in the implementation, however, between the UML inheritance association and the geodatabase implementation of subtypes. The UML definition of a subclass allowed the subclass to inherit all the attributes and methods of the superclass, but define additional attributes and methods applicable only to the subclass. We discussed the example of a superclass *Birds*, having the attributes that they have feathers and a beak, and a subclass *Swans*, which have feathers and a beak, but also have the characteristics of being white and having webbed feet.

In a geodatabase, we cannot have different attribute (column) definitions for different subtypes. We can merely apply different default values and domains to the same set of attributes for all the subtypes of a feature class. In our *WaterPipes* example, all three subtypes would have the same attributes *Diameter* and *PressureRating*, but each might have different default values and domains to reflect characteristics of the different construction materials. A steel pipe, for example, might support a higher pressure or diameter than a plastic pipe.

At times a design decision must be made as to whether to create one feature class with two subtypes, or two separate feature classes. Perhaps we have two different types of roads: public roads and private roads (private roads might be roads built and maintained by forestry or mining companies). The primary motivation for using subtypes is performance. In very general terms, a geodatabase with a small number of feature classes and several subtypes will perform significantly better than a geodatabase with a large number of feature classes and no subtypes. If we can distinguish between public and private roads only by default values, attribute domains and connectivity rules, the appropriate choice would be to create a single feature class with subtypes.

If the two types of road had completely different attributes, or if they need to have different access privileges, then they need to be two distinct feature classes. For example, if public roads have address ranges and road names, but private roads have the name of the owning company and the status (active or inactive), we could not use subtypes. Also, if the road features themselves are digitized and maintained by two different user groups with different database access rights, then again we would need two separate feature classes.

When using a feature class with subtypes, subtypes appear as a single feature class. An attribute for the features defines which subclass it belongs to. The entire classification name is presented in a pull-down list of possible subtypes even if the geodatabase is actually storing a numeric code. This makes storage more efficient, since numbers such as 1, 2 and 3 are stored in the database,

but phrases or categories such as *Gravel Road* or *Paved Road* are presented to the user. Selecting from a pull-down list may be faster for data entry in many cases as well because of this.

Once a subtype is selected from the list, any other attributes which are controlled by the subtype domains and default values will be automatically set. For example, in Figure 2-25, the number of lanes is set to 2 and the default speed limit set to 50 kph when the subtype *Gravel Road* is selected.

OBJECTID*	SHAPE*	SurfaceCode	Lanes	SpeedLimit	SHAPE_Length
4	Polyline	Gravel Road ▼	2	50	1869.582964

**Figure 2-25 Attribute Table with Subtypes**

### Domains and Validation

The **Domain** of an attribute defines the range of possible values for that field. For example, we might limit the diameter of a pipe to be any value between 5cm and 20cm, or limit a land use classification to a finite list of values. Domains are a means of improving the integrity of our database. By predefining what values might reasonably be entered into a database field, then checking new values against these valid values, we can be far more confident of the data in our database. We cannot guarantee that the data are correct, but we can be sure that the values are not wildly wrong. For example, we cannot stop an operator from incorrectly classifying a pipe as 12cm diameter when it is actually 10cm, but we can eliminate typographic errors such as the entry of 100cm when we use a range domain such as described above.

There are two types of domains in a geodatabase:

- Range Domains: provide a range of possible values for continuous variables. For example, the entry of a pressure rating for a pipe might be restricted to a range of values between 40 and 100 pounds per square inch (PSI). The example cited above, restricting pipe diameter to any value between 5 and 20cm is also a range domain.
- Coded Value Domains: provide a list of values which may be entered for a given attribute. Coded value domains would be used for discrete variables such as land use classification, soil classification, or the material a pipe is made of. In these cases, there is a finite list of possible values, and we can define this list at the outset of the project.

Domains are created for the entire geodatabase and given a unique name, rather than being defined for individual classes or attributes. In this way, domains may be shared among several feature or object class attributes. When entering attribute values, a range domain will simply issue a warning message if a value outside the range is encountered. Coded value domains will appear in the attribute table as pull-down lists.

Domains validate attribute values, but they are just one of several tools or methods for validating data. Geodatabases can also validate using:

Connectivity Rules:	allow us to check that attributes of connected features in a network are valid. For example, we can ensure that a 10cm pipe does not connect directly to a 15cm pipe without a reducer fitting.
Relationship Rules:	allow us to constrain the multiplicity of features participating in a relationship class to 1:1, 1:M or M:M. In addition, relationship rules can supply very specific multiplicities. For example, a certain type of fitting might require that exactly three pipes connect to it, or we might wish to force parcels to be owned by one or two owners. Relationship rules can combine the very general multiplicities such as 1:M, or they can include the database constraints we discussed in Module 1 of this course, which can state a precise maximum or minimum multiplicity.
Custom Code:	For very specialized applications, a programmer can create software which can examine the attributes or structures of features and apply very specific criteria.

As a result of spatial edits to a feature, it is often the case that a single feature is split into two separate features, or that two features are merged into a single feature. The GIS software can easily perform the spatial manipulations necessary for these operations without further guidance from the user, but the effects of these operations on the attributes of the affected features are not necessarily predictable. For example, when two features with different attributes are combined, how should the attribute of the merged feature be derived? The rules governing the behaviour of an attribute's values when a feature is split or combined are called the **Split Policy** and the **Merge Policy** respectively. Every attribute domain in a geodatabase may be given both a split policy and a merge policy.

Split policies may be one of three settings, to control how a single attribute value should become two or more values when the feature is split. Split policy settings may be one of the following:

Default Value:	when new features are created by the division of existing features, attributes take on a default value, potentially unrelated to the attribute value of the original feature. The default value setting would be used if the vast majority of situations require a single value. For example, if the majority of the water pipes in a city were constructed of steel, then a divided pipe feature would produce two new steel pipes, regardless of the material of the original pipe.
Duplicate:	when new features are created by dividing an existing feature, the attribute value of the original feature is propagated to both of the newly created features. Thus a steel pipe feature divided into two new features would produce two steel pipe features.
Geometry Ratio:	where an attribute is numeric, the geometric ratio setting will multiply the original attribute value by the proportion of the original geometry present in the new feature. For example, there may exist a 10 hectare polygonal parcel feature, with an attribute <i>Value</i> which indicates the monetary value of the land. If the parcel were divided into two new parcels, each 3 ha and 7 ha in size, the <i>Value</i> attribute from the original parcel would be multiplied by 30% for one parcel and 70% for the other.

Merge policies may be one of three settings which control how the attributes of two distinct features should be combined to create an attribute value for a new merged feature. Merge policies may be one of the following settings:

Default Value:	when two or more features are combined to create one new feature, the resulting feature takes on a default value, potentially unrelated to the attributes of the component features. As with split policies, this setting is most helpful if there is a value which should apply in the vast majority of cases.
Sum Values:	For numeric attributes, the sum values setting will take the arithmetic sum of attribute values of the features being merged, and store the sum in the attribute of the resulting merged feature. As with the geometric ratio example above, we might use this setting to make a new, larger parcel have the <i>Value</i> which is the sum of the values of the parcels being combined.
Geometry Weighted:	For numeric attributes, the resulting attribute value is the weighted average of attribute values in the original features, weighted by the proportion of the geometry contributed by each feature. For example, we might have a polygonal feature class which stores areas of similar forest characteristics, perhaps considering the species and size of tree present in a polygon. As part of this feature class, there might be an attribute <i>VolumePerHa</i> , which identifies the estimated volume (in cubic metres) of timber per hectare of land. We might merge two polygons, one 3.0 ha in size with <i>VolumePerHa</i> of 250 m <sup>3</sup> /ha, and another 7.0 ha in size and having a <i>VolumePerHa</i> of 300 m <sup>3</sup> /ha. The resulting <i>VolumePerHa</i> of the new polygon would be $0.3 \times 250 + 0.7 \times 300$ , or 285 m <sup>3</sup> /ha.

While the use of domains and validation give us ways of controlling the values which can be entered in attributes and the ways we can structure our data, merge and split policies simply make working with our data faster and easier. They are used to make entering common attribute values automatic, and mean that operators may only need to alter attributes in unusual cases.

These tools to control behaviour can together create a very robust representation of how a feature class behaves in the real world. For example, Table 2-5 summarizes the ways in which the behaviour of road features can be closely controlled using these elements (adapted from Zeiler, 1999, p.89). It assumes that there is a single roads feature class with subtypes to indicate the construction material.

**Table 2-5 Example of Behaviour Elements**

Behaviour Element	Examples
<b>Default Values</b>	A new concrete road is given a default value of 4 lanes A new asphalt road is given a default width of 10 metres
<b>Attribute Domains</b>	For asphalt roads, valid widths are 10m, 12m or 15m, and valid number of lanes are

	1, 2 or 4. For gravel roads, valid widths are 5m, 8m and 10m, and valid suffixes are "Lane" and "Road".
<b>Split/Merge Policies</b>	A merged asphalt road takes a default value of 2 lanes. A split gravel road retains its width.
<b>Connectivity Rules</b>	A 2-lane asphalt road can only connect to another 2-lane road. A concrete road can connect to an asphalt road, but not a gravel road.
<b>Relationship Rules</b>	An asphalt road may be associated with tunnels or bridge features. A gravel road cannot have more than 4 roads at an intersection.

### Topology

Topology is a means to control the spatial relationships between features or feature classes in a geodatabase. While domains concentrate on the integrity of attribute data, topological rules help improve the integrity of the spatial data. There are three types of topology available in the geodatabase:

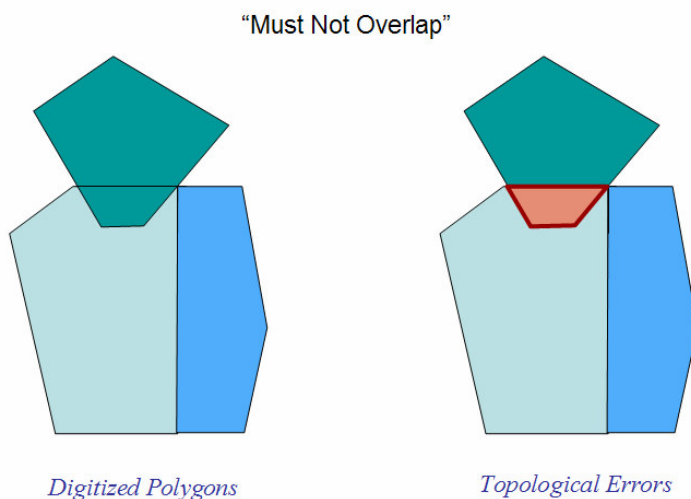
- Map Topology:** is the topology control imposed by the ArcGIS application during an edit session. As we have discussed earlier in the module when discussing Shapefiles, map topology can be used to ensure that shapefiles conform to topological rules, such as connecting lines at endpoints, or sharing boundaries for polygons. This type of topology is never stored with the finished data, but is simply a set of editing tools which allow users to better structure their data.
- Geodatabase Topology:** is a set of rules which define the spatial relationships involving one or more feature classes. This is an element stored within a geodatabase, and can be added to a view like other feature classes. Geodatabase topology allows ArcGIS to inspect features as they are added or edited to ensure they adhere to spatial rules.
- Geometric Network Topology:** is used to structure features forming a geometric network. This allows ArcMap to understand connectivity and flow in a network. Geometric networks include edge and junction features and model flow in stream, water, sewer and electrical and other networks.

Geodatabase topology rules may apply to features within the same feature class, or to features in different feature classes. For example, a topology rule may be defined to ensure that land parcels do not overlap each other, or to ensure that buildings lie entirely within parcel boundaries.

The general process with geodatabase topology is to define the rules which govern spatial behaviour of feature classes, then validate the feature geometry. For example, we might stipulate that land parcels cannot overlap each other. Once the rule is defined, we validate the parcels, and any areas which overlap are identified and presented to the user to remedy. In this manner, the topology of features (such as which parcels are adjacent or overlapping one another) is not

explicitly stored with the geodatabase as was the case with coverages, but it is discovered either at the time the rules are created or after features are added to a feature class.

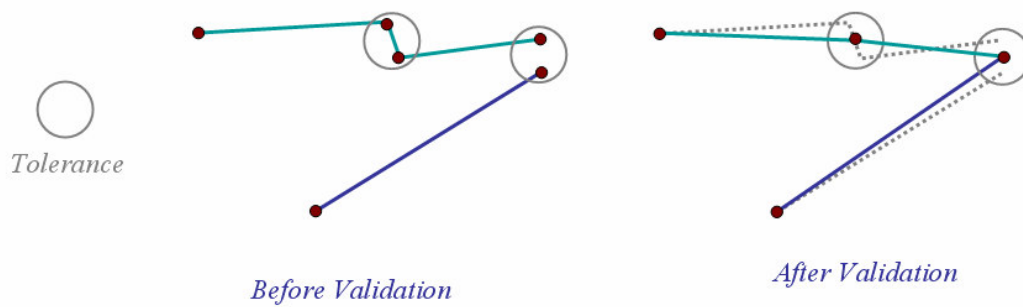
There are 26 predefined topological rules to choose from, each defining a different type of topological relationship that is acceptable (see the ArcGIS Help for a clear description of all available rules [http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm?TopicName=Topology\\_rules](http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm?TopicName=Topology_rules)). For example, the *Must Not Overlap* rule functions with a single polygonal feature class to ensure that no polygons overlap in space. Polygons may share boundaries or touch at a point, but they cannot overlap. Figure 2-26 shows an example of three features digitized such that two share a common vertical boundary, and a third feature overlaps partially with a polygon. When these three features are validated using the *Must Not Overlap* rule, the overlapping portions are identified as a topological error and highlighted for the user. Note that no errors result where polygons either touch at a point, or share a boundary.



**Figure 2-26 Topological Errors Resulting from the *Must Not Overlap* Rule**

As part of the topology rule definition, a **Cluster Tolerance** is specified. The cluster tolerance is a distance within which all points are considered to be a single point. Points within this tolerance are collapsed into a single point. This applies to nearby vertices within the same line or polygon feature, or to vertices of two nearby features. The cluster tolerance ensures that points that are intended to be coincident, but are perhaps merely very close to one another, are forced to be coincident prior to assessing which features are adjacent, overlapping or intersecting. Clustering is not intended as a feature generalisation tool.

For example, consider two linear features as shown in the Before Validation area of Figure 2-27. The cluster tolerance distance is indicated by the grey circle. Note that in two areas, more than one point lies within the tolerance. As part of validation, all of the points within the tolerance are collapsed into a single point at the mean centre of the original points. In our example below, the validation process will alter the shape of the light-blue line, and force the two lines to be connected at a common vertex.

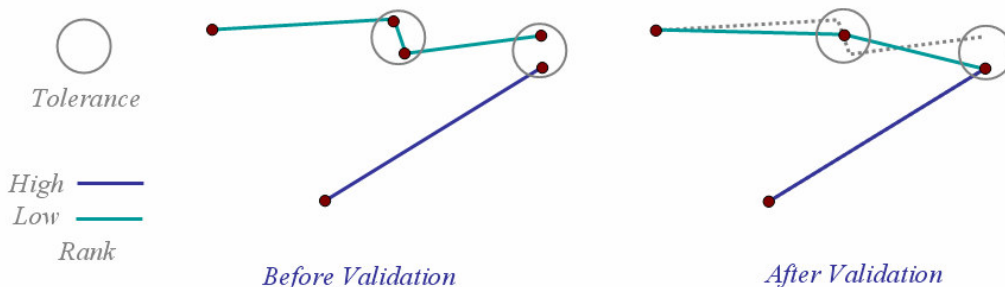


**Figure 2-27 Effects of Cluster Tolerance During Topology Validation**

Cluster tolerances should be very small. The default tolerance is 0.001 metres, which means that only points extremely close together will be moved. There is a trade-off when defining cluster tolerances: too small and features will not be integrated correctly, and too large and unwanted changes may be made to feature shapes. In general, the cluster tolerance should be significantly smaller than the spatial accuracy of the data. For example, if features are accurate to 2.0 metres, an order of magnitude smaller than that (0.2 metres) would be the largest cluster tolerance considered. It is common to use a tolerance several orders of magnitude smaller than the data accuracy (perhaps 0.02 or 0.002 metres).

Also during the definition of a topology rule, the user may specify a series of ranks for feature classes. Feature class **Rank** defines, in a relative measure, which feature classes have the most important spatial locations. It tells the ArcGIS application which features should be moved and which should not, where two features lie within the cluster tolerance of each other. Each feature class participating in the topology may be ranked, and where two vertices of the two feature classes are within the tolerance of each other, the lower-ranked feature is snapped to the higher-ranked feature.

For example, consider the same set of features as in the previous example, except with the features having different ranks. This situation is shown in Figure 2-28. The light blue line changes shape as before, but with feature class ranks defined, the connection of the two features at their endpoints is different. Because the dark blue line has a higher rank, the lower-ranked feature (the light blue line) moves and snaps to the original location of the dark blue line. Thus the higher ranked feature (dark blue) does not change its geometry at all.



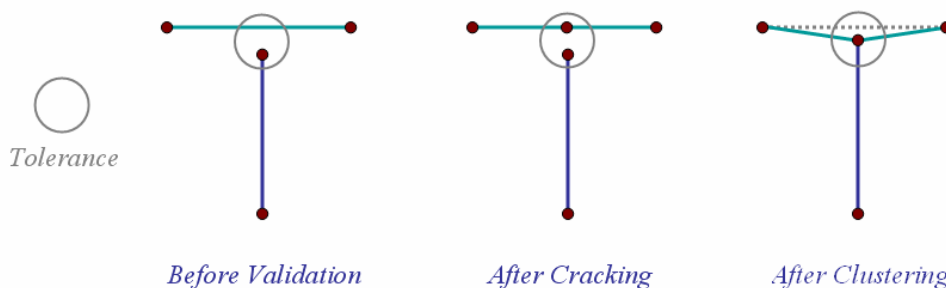
**Figure 2-28 Effects of Feature Class Rank on Topology Validation**

Feature class rank should be used if you are using topology rules involving feature classes with different levels of accuracy. Generally, the less-accurate feature class should be lower ranked than

the higher-ranked feature class, ensuring that the most accurate data remains, for the most part, intact.

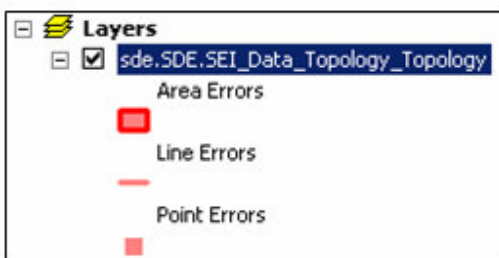
The validation process for topology rules follows a two-step process for altering feature geometry. During **Cracking**, vertices are inserted along a linear feature where points lie within the tolerance of the line, but no vertex existed previously. During **Clustering**, any points within the cluster tolerance are collapsed to create a single point.

For example, in Figure 2-29, in the *Before Validation* portion of the figure, we see two lines passing within the cluster tolerance of one another. However, the light blue line at the top has no vertex near the dark blue line. The cracking process inserts a new vertex in the light blue line at the point nearest the dark blue line. The clustering process then collapses the two vertices into a single point and snaps the two line features together at the mean centre point of the points.



**Figure 2-29 Cracking and Clustering in Topology Validation**

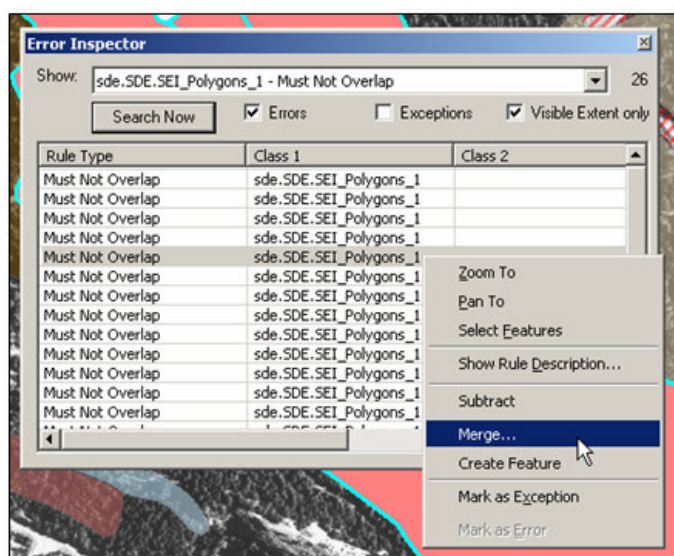
Once validated, the topology may be added to a view in ArcMap and any errors resulting from the validation process may be addressed individually by the user. Figure 2-30 shows a topology added to the Table of Contents in ArcMap. Note that each type of error (polygon, line, point) is represented with a different symbol.



**Figure 2-30 Topology in the ArcMap Table of Contents**

There are a variety of tools available to assist users in dealing with topological errors:

**Error Inspector:** A dialog which lists each error encountered, and allows users to search for and zoom to each error. A series of actions are presented to the user to quickly fix topological errors. Figure 2-31 shows the error inspector interface.



**Figure 2-31 Topology Error Inspector**

In this example, a specific error is highlighted from the list of errors, and when right-clicked, the inspector provides 3 possible solutions to overlapping polygons: *Subtract* will simply remove the overlapping area, *Merge* will add the overlapping portion to one of the overlapped polygons and remove the area from any other polygons sharing this area, and *Create Feature* will simply create a new polygon from the overlapping area and remove this area from all overlapping polygons. The suggested fixes will be different depending upon the nature of the topological error. Users are free to edit the features as necessary manually if the suggested options are unsatisfactory.

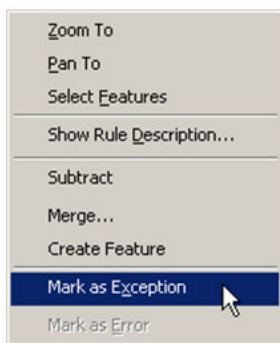
**Topology Toolbar:** Provides easy access to a number of tools relating to topology, including buttons to validate the topology and initiating the error inspector. This toolbar runs within the ArcMap interface.

**Error Reports:** Creates either an on-screen or permanent file report which summarizes the number of outstanding topology errors for a given set of topology rules. Such a report would be a helpful addition to data documentation upon completion of a project, demonstrating that all topological errors have been successfully processed. Figure 2-32 shows an example of an error report.

General   Source   Selection   Display   Symbology   Feature Classes   Rules   Errors		
Generate Summary		Export To File...
Rule	Errors	Exceptions
Must Be Larger Than Cluster Tolerance	0	0
Must Not Overlap sde.SDE.SEI_Polygons_1	82	3
Total	82	3

**Figure 2-32 Sample Topological Error Report**

While topology errors usually occur as a result of errors in the data, there are times where the spatial rules may be violated in reality. In these cases, we need to mark the offending feature as an **Exception**, meaning that for this specific instance we will explicitly allow the topological error to occur. For example, we might have a topology rule stating that buildings must lie entirely within a land parcel, and cannot cross the lot line and infringe on neighbouring properties. While this rule is observed in the vast majority of cases, there are on occasion situations where buildings were built across parcel boundaries in error, or perhaps where the same owner owns two neighbouring parcels and has constructed a building spanning both lots intentionally. In this small number of cases, we will want to mark these features as exceptions. While editing errors, the operator may simply right-click on an error and select Mark as Exception, as shown in Figure 2-33.



**Figure 2-33** Marking an Error as an Exception

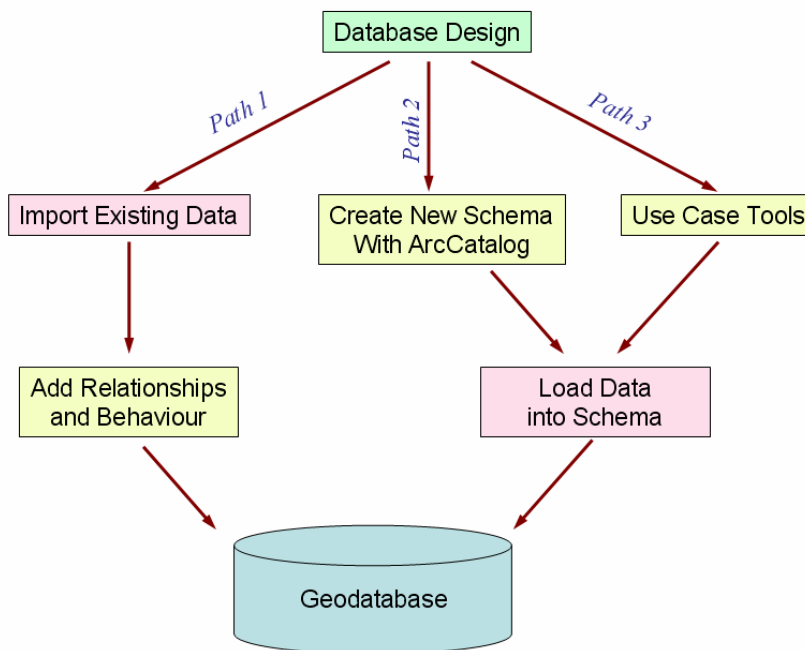
## 2.4 Creating a Geodatabase

### 2.4.1 Creation Workflow

There are several ways to create a new geodatabase. This process should always begin with a design phase, but the way a geodatabase moves from a design to a complete geodatabase with feature classes and domains may vary.

Fundamentally there are two ways to create a geodatabase: either we import existing data as they currently exist, and add relationships, domains, etc. afterward, or we begin by creating an empty database with all the necessary relationships and behaviour and then import or create the necessary data.

For example, the creation process can be defined as shown in Figure 2-34. One can see the process always begins with a design phase and ends with the completed geodatabase. There are several paths, though, between the start and end of the process.



**Figure 2-34 Geodatabase Creation Process**

The yellow boxes are tasks involving creating or modifying the structure, or schema, of the geodatabase. During these steps, we would define elements such as attributes, relationships, subtypes, domains and topology. The pink boxes correspond to tasks involving loading of data. One can see that all paths involve either a process of loading followed by schema refinement, or a process of defining the schema followed by a data loading phase.

The three major paths from database design to the completed geodatabase database are labelled as Path 1, Path 2 and Path 3 in Figure 2-34, and can be summarized as follows:

- Path 1: Import any existing data into a new geodatabase, then refine the geodatabase by adding relationships and behaviour elements using ArcCatalog.
- Path 2: Manually create the designed geodatabase using ArcCatalog to create feature classes, attributes, relationships and behaviour, then load or create data in the empty schema.
- Path 3: Use Computer Aided Software Engineering (CASE) tools to design the geodatabase structure, then use ESRI tools to automatically create the designed database directly from the design. In this path, the CASE software plays a significant part in the Design stage itself, but for notation purposes we have separated design and CASE implementation for clarity. As with Path 2, this path involves creating an empty geodatabase structure, and later adding data to it.

In practice, organisations may use some or all of these methods.

Most organisations have significant volumes of existing data in various formats which will be utilised in the completed geodatabase. These data might include coverages, shapefiles, computer-aided design (CAD) files, and non-spatial tables stored in dBase or Excel format. Importing existing data will almost always form a large part of the geodatabase creation process.

Tools exist within ArcCatalog and ArcToolbox to import such data, and allow simple-to-use wizards to perform such tasks as mapping attributes in an incoming table to perhaps differently-named attributes in the destination table. These tools also ensure that the data types or geometry types in one data format are mapped to an appropriate type in the new geodatabase. For example, we will need to change a dBase Number data type field in a source shapefile into either a short or long integer field in the geodatabase.

It is recommended that the tools supplied with the ESRI software be used to load tables and feature classes into a geodatabase, and not tools within the RDBMS. By using existing ArcMap tools, the new objects will be registered with the geodatabase so that they may participate in relationships, validation rules, and so on.

### **2.4.2 CASE Tools**

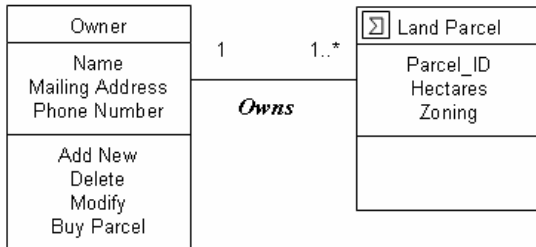
Computer-Aided Software Engineering (CASE) is the use of software tools to design and build software applications and databases. These software tools are often referred to as CASE Tools. These software applications typically include the ability to model data and behaviour of databases and systems, often using UML notation. For software applications, CASE tools can often create parts of the programming code. For database applications, the CASE tools can produce the database structure and may generate some code which will help with managing behaviour or constraints. There is usually the ability to help manage the development effort, including revision control. In a very general sense, software tools which assist in all aspects of the Systems Development Lifecycle could be considered CASE tools. Such tools include applications for project management, functional analysis, system design, translation tools, test software and applications to generate user documentation. For our purposes, we will be examining in detail tools which assist with the design of databases using UML class diagrams, and the process of converting such a design into a functioning geodatabase.

There are several advantages of using CASE tools for a geodatabase design:

- The design serves as documentation for your geodatabase, and they may be easily shared and reviewed by others.
- For large or complex geodatabase models, it may be faster or more clear to create the model using CASE tools.
- Updating your geodatabase structure is easy – changes to the design may be quickly made to the geodatabase even after data have been loaded.

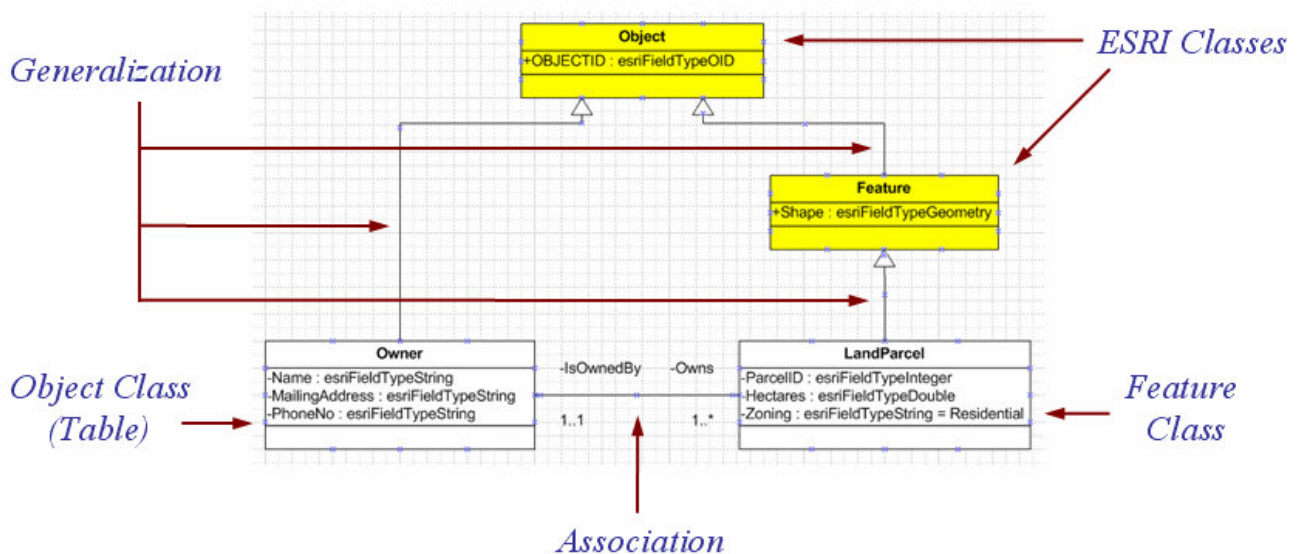
There are three components to creating a geodatabase using CASE: the design notation used to describe the database model, the design software in which the design is created, and the import mechanism which converts the design into a complete geodatabase which is ready to use. The design notation used will be the UML Class Diagram, which we discussed at length in Module 1. There are several design software packages available, and two of the leading applications (Rational Rose and Visio) are supported by ESRI. We will be using Microsoft Visio in our discussion and examples. Visio is a general-purpose drawing application which can be used to create flow charts, organisational charts or even office layout diagrams. It can also create a variety of UML diagrams, including Use Case Diagrams, Sequence Diagrams and Class Diagrams. The import mechanism is the Schema Wizard in ArcCatalog.

As an example of the process of creating a geodatabase using Visio, we will revisit an example from Module 1: that of the Owner-Parcel relationship. The Class Diagram for this relationship is repeated again in Figure 2-35 as a reminder. This is a simple relationship, involving a non-spatial table called *Owner*, and a polygonal feature class called *LandParcel*. Our two classes have a small number of attributes to illustrate how these will be implemented in Visio.



**Figure 2-35 Owner-Parcel Relationship**

Figure 2-36, below, shows the same class diagram as it would be drafted in Visio. The difference which is immediately obvious is the addition of the two yellow classes *Object* and *Feature*. These represent the ESRI ArcObjects classes which will be implemented to create our two user-defined classes. The class *LandParcel*, for example, is connected to the *Feature* class by an inheritance or generalization relationship. You may recall that this is often called the “is-a” association, so a *LandParcel* is-a *Feature*. This allows ArcGIS to understand that the class *LandParcel* needs to be created in the new geodatabase as a polygonal feature class. Without these ESRI object definitions, ArcGIS would not be able to convert a class from Visio into a geodatabase element in the new geodatabase. This diagram also connects the *Owner* table to the *Object* class, so that ArcGIS understands that *Owner* is a non-spatial table, and connects *Feature* to *Object* to indicate that a feature class is simply a refinement of an object class, in that it has a spatial component.

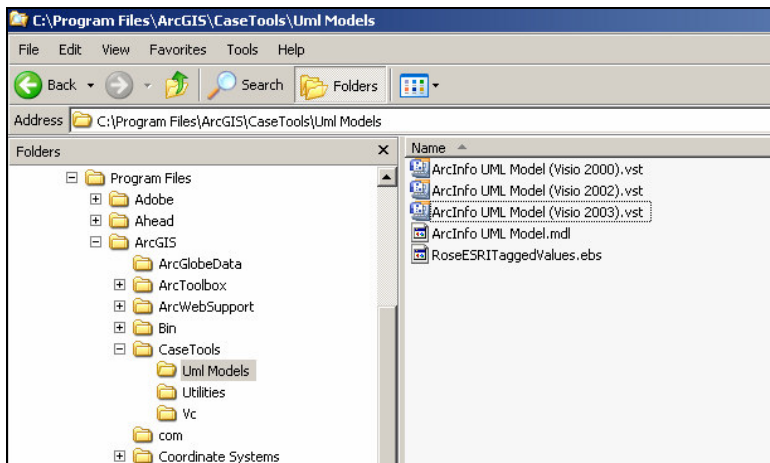


**Figure 2-36 Example Visio Design**

Also note that the attributes have been defined in a more detailed manner. Each attribute is given a data type, which again must correspond to an ESRI data type. This is why attributes are defined as `esriFieldTypeDouble` and `esriFieldTypeString`. As with the *Feature* and *Object* classes, these ESRI data types allow the creation of a geodatabase which can support the correct data. In addition, the *Zoning* attribute of the *LandParcel* class is given the default value of “Residential”. Visio allows us to define most of the geodatabase elements we have previously discussed. We can create subtypes, default values and domains as part of our designs. Some elements, such as topology, spatial reference and annotation, cannot be modeled in Visio and must be added to the data model using ArcCatalog after the geodatabase has been created.

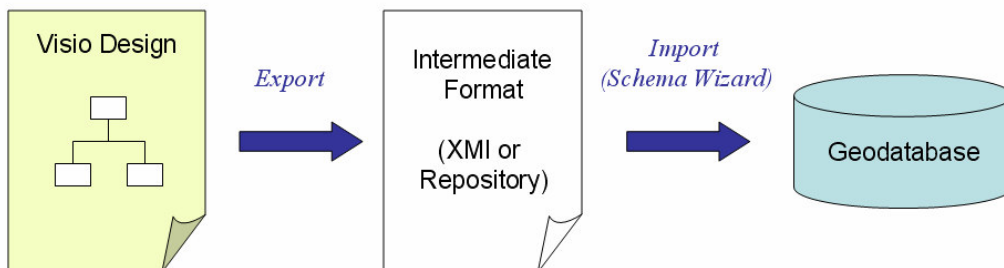
The ESRI objects such as the *Feature* class and the field data types are defined in a Visio template file which is installed with ArcGIS on your computer. Before beginning a design, you must load this template in order to add these custom objects to Visio. The ArcObjects elements in the template file include the *Object* and *Feature* classes, as well as geodatabase field data types, the geometry types, and specialized classes such as edges and junctions for network feature classes. These are the blueprint for how the Visio design should be translated into a finished geodatabase.

There are separate templates for different versions of Visio or Rational Rose. The files are in a subfolder of the ArcGIS installation folder. Figure 2-37 shows the template files supplied with ArcGIS 9.2, and their location in a typical ArcGIS installation.



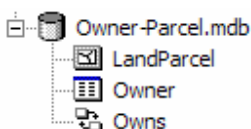
**Figure 2-37 ESRI Template Files**

As discussed previously, the import mechanism which takes a finished design and creates the necessary geodatabase elements is called the Schema Wizard in ArcCatalog. The schema wizard cannot directly read the Visio design, but requires an intermediate format to be produced. This may be an XML metadata interchange (XMI), or Microsoft Repository MDB. As a result, the transition from design to geodatabase will appear as shown in Figure 2-38.



**Figure 2-38 CASE Tool Process**

The schema wizard will read the XML or repository database to determine what objects, relationships and behaviour elements must be created in the new geodatabase. Users may set some properties during the import process, such as setting the spatial reference for a feature class or feature dataset. The result of the import process on our small design would produce a simple geodatabase as shown in Figure 2-39. The finished geodatabase contains a single polygonal feature class called *LandParcel*, a non-spatial table called *Owner* and a relationship class called *Owns* which associates *LandParcel* and *Owner*. All the attributes and multiplicities of these elements is created as part of the import process.



**Figure 2-39 Finished Geodatabase**

For many GIS user groups, such as forestry, geology, groundwater or transportation, data models have previously been created. ESRI maintains a series of web pages devoted to these models,

and promotes their use and refinement. These data models may serve as a starting point for future development in your area of use, or they may serve to illustrate alternative ways of representing things in a geodatabase.

## References

- Blaha, M., and Premerlani, W. *Object-Oriented Modeling and Design for Database Applications*. Prentice-Hall, 1998.
- ESRI. *Building a Geodatabase*. ESRI Press, 2005.
- ESRI. *Introduction to CASE Tools*. ESRI Press, 2004.
- ESRI. *Working with Geodatabase Topology*. ESRI Press, 2003.
- Kroenke, D.M. *Database Processing Fundamentals, Design and Implementation*. Prentice-Hall, 1998.
- Shekhar, S., and Chawla, S. *Spatial Databases: A Tour*. Prentice-Hall, 2003.
- Zeiler, M. *Modeling Our World*. Redlands, CA: ESRI Press, 1999.

### 3 Multi-user Geodatabases

Allowing several users to simultaneously access a database can introduce new problems to the management of a database. This module seeks to provide an overview of the general process of allowing multi-user access to data, and the manner in which the ESRI suite of applications resolves these challenges.

Topic 1 of this module provides an overview of the theoretical basis of multi-user databases, including a description of the problems one may encounter, and the possible approaches to mitigating such problems. Topic 2 discusses how the ESRI suite of applications handles multi-user access. It describes the ArcSDE (Spatial Database Engine) application and the mechanism it uses to support several concurrent editors (versioning). The final topic looks at administration of geodatabases. While this final topic does not specifically relate to multi-user support, in general much of the administration of a geodatabase arises from support for multiple users and the problems inherent with large numbers of changes made to geographic data.

#### Module Outline

- Topic 1: Multiuser Databases
- Topic 2: ArcSDE and Versioning
- Topic 3: Geodatabase Administration

## 3.1 *Multi-user Databases*

### 3.1.1 Introduction

Multi-user database processing arises when more than one person uses the database at the same time. Almost all large organisations making use of GIS technology will wish to allow several users to access the same data at the same time. This situation can create some significant differences from single-user database systems, however. Foremost is that there arises the possibility that the work one person does will conflict with or contradict work that another user does. For example, two salespeople might sell the same item in inventory if there were no controls in place to manage multiple accesses to the same record of a table. In addition, there will now be potentially many users, with differing information needs and operating skills accessing the database, which may necessitate administration functions such as security. In larger organisations, there may be a need for additional staff simply to perform database administration (DBA) tasks.

The term **concurrency** refers to methods associated with allowing many users to access the same database at the same time. The procedures within the DBMS which ensure that concurrent transactions in the database do not interfere with each other are sometimes referred to as concurrency control mechanisms. Such mechanisms attempt to present the results of a single user's work within a multi-user environment as if that user were the only user of the database. Reliability in a multi-user database uses concurrency control mechanisms and database **recovery** functionality to repair or reconstruct data in the event of system failures.

Topics such as concurrency and recovery are large subjects; the goal of this topic is to introduce students to the terminology and the most important concepts in the topic.

### 3.1.2 Transaction Processing

In most uses of databases, users process work in **transactions**, which are logical units of work which must be either completed or aborted; intermediate results are unacceptable. Transactions may be composed of one or more database access operations, such as insertions, deletions, modifications or retrievals. For example, consider the situation where a user of a banking database wishes to transfer funds from one bank account to another. In this situation, the transaction consists of two updates: one update to reduce the balance of the source account, and a second update to increase the balance of the destination account. It would be unacceptable if only one of these updates were performed, thus the entire transaction must either be completed in its entirety or aborted if problems are encountered. Incomplete transactions can have a destructive effect on database integrity.

Transactions are important during in a multi-user application, but they are also significant in single-user cases as well. Unfortunately, there is always a chance that the DBMS will encounter errors while attempting to process a transaction. Such failures might include storage media failures and power supply failures. A system crash between our two updates in the banking transaction example above would leave the database in an inconsistent state. Transaction management on the part of the DBMS ensures that if the transaction fails to complete in its entirety, any operations forming part of the transaction will be undone.

The **commit** and **rollback** operations are central to transaction processing. The commit operation signals the successful completion of a transaction. Upon encountering a "commit point" in a series

of database operations, the database will be in a consistent state and any updates may be made permanent. A commit indicates the end of a transaction. A rollback operation will undo any changes made since the last commit point. System failures will trigger a rollback operation, or rollbacks may be issued in programming code as part of a recovery routine. Commit and rollback operations are particularly important in relational systems, since the information stored in a database tends to be split into many related tables, so that a single change may require updates to several tables.

Consider once again the two updates necessary to transfer funds from one bank account to another. If the first update deducts the amount from the source account and an error of some sort is encountered, that first update would be rolled back, restoring both accounts to their pre-transfer state. Alternately, if the first update to deduct the funds, and the second update to add the funds to the destination account both complete successfully, a commit point is reached and both updates are made permanent. In this manner, we ensure that funds are not simply “lost” due to poor transaction management.

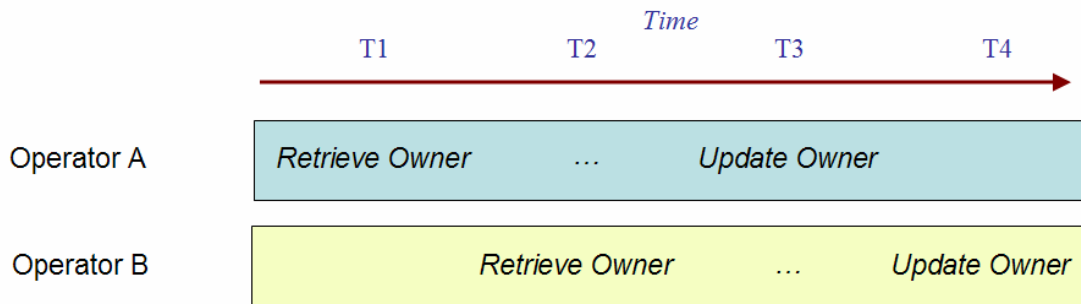
Transactions must have several characteristics to ensure database integrity during concurrent processing:

Atomicity	Transactions are sometimes called <b>Atomic</b> , since they must be performed as a unit. Transactions are treated as a single, indivisible unit of work.
Durability	Databases must be left in a consistent state. When a transaction is completed, the database reaches a consistent state and that state cannot be lost even in the event of system failure.
Serializability	Describes the result of concurrent execution of transactions from several users. Transactions cannot be <b>interleaved</b> , or executed in pieces with other transactions, but rather each transaction should be executed serially in order.
Isolation	Data used by one transaction cannot be used by a second transaction until the first one is completed.

Single-user databases always ensure both serializability and isolation, because with only one user, transactions will be executed one at a time. In a multi-user situation, the DBMS must implement controls to ensure these four characteristics of transactions are preserved. There are essentially three ways in which problems can be encountered during concurrent processing. Terminology may vary between references, but fundamentally they describe the same database integrity problems. We will use the terminology adopted by Date (2000), but other terminology will be noted where relevant.

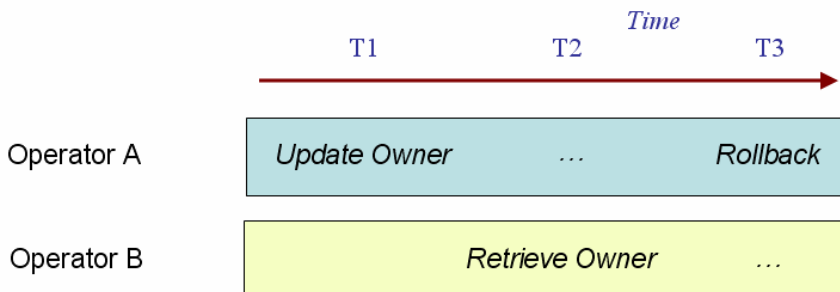
The *Lost Update Problem* may occur any time two separate users are accessing and changing the same records in a database. For example, we might have two GIS operators updating owners of a land parcel database. If they were both to edit the owner of a parcel, the result can be as shown in Figure 3-1. This figure shows the two operators on the left side, and time reading from left to right. At time T1, operator A reads the owner of a given parcel of land. For the sake of discussion, let us assume the database responds with the name “Mr. Smith”. At time T2, Operator B also performs the same database read, and would receive the same response. At time T3, Operator A updates the parcel, changing the owner’s name to “Mr. Jones”. At T4, Operator B updates the owner to become “Mr. Thomas”. The result of this set of operations is that the parcel owner is set to “Mr.

Thomas". The value "Mr. Jones" which was applied by Operator A at T3 is lost. Since the value written at T3 lies between Operator B's retrieval at T2 and update at T4, Operator B never even sees the value that was applied at T3 and simply overwrites it at T4.



**Figure 3-1 The Lost Update Problem**

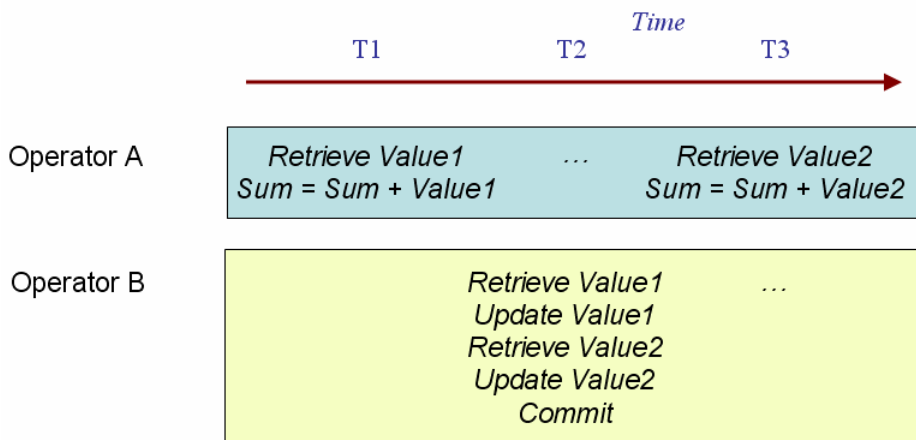
The *Uncommitted Dependency Problem* (also called the Temporary Update Problem) may occur where one transaction updates a database item but then the transaction fails for some reason and the update is rolled back. Figure 3-2 shows such a situation, and we can assume again that the original parcel owner is "Mr. Smith". Here, Operator A changes the parcel owner to "Mr. Jones" at time T1. At time T2, Operator B retrieves the owner, and is given the value "Mr. Jones". At time T3, Operator A's transaction encounters a problem, and the entire transaction is rolled back, functionally changing the owner back to "Mr. Smith". Operator B, however is still working on the assumption that the owner is "Mr. Jones", a value which in a sense never existed in the database in any permanent way.



**Figure 3-2 The Uncommitted Dependency Problem**

The *Inconsistent Analysis Problem* (also called the Incorrect Summary Problem) may occur if a transaction calculates summary values, such as a sum or average, while other transactions are updating the values. In this situation, the summary transaction might read some values from before the values are changed, and other values after the transaction. Figure 3-3 shows such a situation. In this case, we will assume the two operators are working with two parcels of land, the first worth \$50,000 and the second worth \$70,000. Operator A begins a process of determining the total value of two land parcels at time T1 by reading the value of the first property. The database returns the value \$50,000. The transaction for Operator A adds this value (\$50,000) to a *Sum* variable which will "hold" the calculated sum. At time T2, Operator B makes several updates. They change the value of the first parcel to \$60,000 and the value of the second parcel to \$80,000. Operator B then commits both of these updates. Operator A then continues their sum calculation at time T3 by retrieving the value of the second parcel. The database responds with \$80,000.

That value is then added to the existing Sum variable, producing a total for the two land parcels of \$130,000.



**Figure 3-3 The Inconsistent Analysis Problem**

The problem here is that the sum of the two parcel values before Operator B made the changes was \$120,000. After Operator B made all necessary changes, the total value for the two lots was \$140,000. Operator A has calculated a value of \$130,000, which is neither the old nor the new value, but an inconsistent sum which included one old value and one new value.

In all of the cases described above, problems arise only because two or more transactions require access to the same database information, and at least one of the transactions has a “write” operation. Two simultaneous “read” operations do not cause database inconsistencies.

### 3.1.3 Resource Locking

All three of these problems are a direct result of concurrent database access. Each transaction is correct in isolation, but with interference from other transactions they can produce incorrect values. A method of managing concurrent processing is called **locking**. The basic premise is that if a transaction requires exclusive use of a resource, like a database record, it “locks out” all other transactions so that they cannot change the resource until the locking transaction completes. Any subsequent transactions wishing access to the same database resource must wait until the lock is released before proceeding. In this manner a given transaction can carry out its necessary actions in the knowledge that the objects of interest will not be altered by other transactions until the lock is released. Locks may be acquired at several database levels: database, table, page, row or column. The level of locking, or the size of the object being locked, is called **lock granularity**.

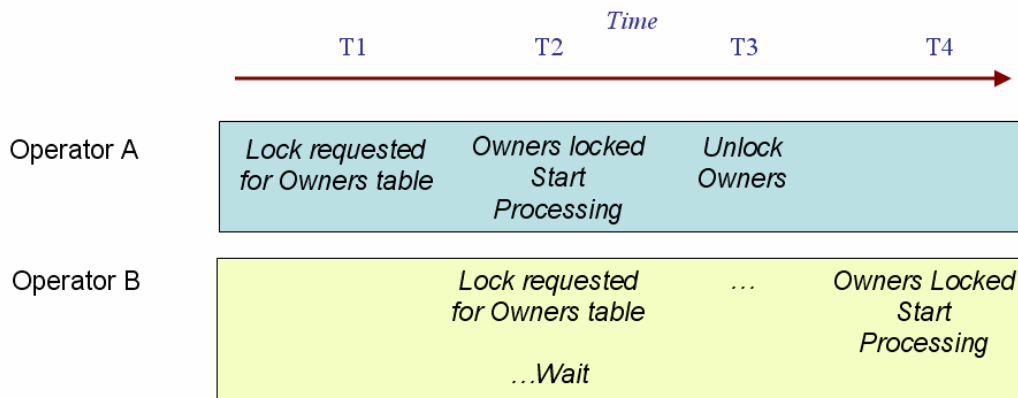
#### Database Locks

In a database-level lock, the entire database is locked, preventing the use of any tables until the locking transaction completes. This is an unusual approach, and is really only acceptable in single-user implementations, or for large batch processes.

#### Table Locks.

In a table-level lock, an entire table is locked while a given transaction performs actions with it. In many cases, changes must be made to several tables necessitating all applicable tables to be locked. Two transactions may operate on the same database if they are accessing different tables.

Figure 3-4 illustrates table-level locking. In this example, we can see that once Operator A has begun their transaction, the *Owners* table is locked such that no other transaction may alter this table. At time T2, Operator B tries to initiate a transaction requiring a change to the *Owners* table, but due to the lock acquired by Operator A, must wait until the lock is released. At time T3, Operator A's transaction is complete, and the lock is released from the *Owners* table. At time T4, Operator B is granted a lock on the *Owners* table, and may begin operations with that table. In this manner, transactions are restricted to serial access on the *Owners* table.



**Figure 3-4 Table Locking**

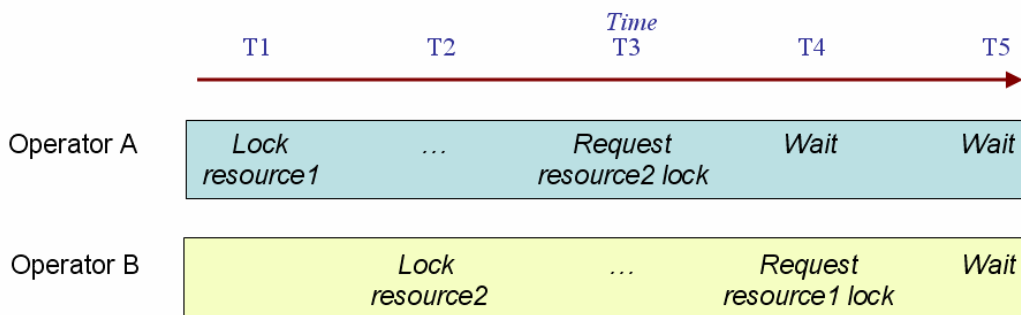
Page Locks	Page-level locks restrict access to the storage media itself. A page, or diskpage, is a section of disk. A page has a fixed size, such as 8K, 16K, 32K, and so on. Since a given page is simply a portion of a storage disk, a page can contain part of one table, an entire table, or parts of several tables. If a transaction requires a lock to an object in the database, the entire media page is locked, which may affect several tables or it may affect only a small portion of one table, depending upon the size of each table and how each is divided into pages on the media. Page-level locking is one of the most frequently used DBMS locking mechanisms.
Row Locks	Row-level locking is far less restrictive, in that it can allow other transactions to access the same table, but not the same row that is locked by a previous transaction. This mechanism can vastly improve data availability, since locks are now only granted for specific rows being accessed. However, the overhead of managing a large number of row-level locks can be high.
Field Locks	Field-level locking allows concurrent transactions to access the same row, as long as they require the use of different fields within the table. As with row-level locking, however, the overhead of managing as many locks as there are database fields outweighs the gain in flexibility to transaction access.

Locks can also vary by their type. The lock status of a given object (table, or row, for example) can be one of three values:

Unlocked	No other transaction is currently active using the object, and a new lock may be granted for access.
Shared Lock	A shared lock is really a “read” lock. It means that a transaction is currently reading the object. As a result, other transactions may concurrently read the same object without unwanted results, but no transaction will be granted permission to alter the object. Several transactions which require read access to an object may concurrently be granted a shared lock on an object. Transactions which seek to alter the object must wait until a shared lock is released.
Exclusive Lock	An exclusive lock is a “write” lock. If a transaction will alter a value in the database, an exclusive lock must be placed on the object so that no other transactions have access of any kind. As we saw with the three concurrency problems described above, it is the “write” accesses which cause problems in concurrency.

### 3.1.4 Deadlock

While locking resolves the problems introduced by concurrent database access, it can also introduce a new problem. **Deadlock** is a situation where two or more transactions are in a simultaneous wait state, each waiting for the other to release a resource lock. Figure 3-5 shows an example which will cause a deadlock situation.



**Figure 3-5 Deadlock**

In this situation, Operator A initiates a transaction which requires an exclusive lock on a resource, noted as resource1. At T2, Operator B initiates a transaction requiring an exclusive lock on a second resource. At T3, Operator A's transaction requests a lock on resource2. Since Operator B already has a lock on that resource, Operator A's transaction must wait until the lock is released. At time T4, Operator B's transaction requests a lock on resource1. Since this resource is already locked exclusively, Operator B's transaction must wait as well. By time T5, both transactions are waiting for the other to release locks, and neither transaction may proceed.

One method of managing deadlock is to ensure that a deadlock situation never arises. The most basic means of doing this is to force a transaction to obtain all the locks it will need before the transaction begins executing. This method avoids conflicting transactions because locks must be allocated in blocks, in succession. This method is effective, but it does limit concurrency in that it locks far more resources at once. A second way of avoiding deadlock is to order every item in the database and make sure that a transaction requiring several locks acquires those locks in the order

specified. For example, in a table-locking system, each table would be assigned a position in a sequence and locks on multiple tables would be acquired in that predefined order. By forcing resources to be locked in a specific order, we remove the possibility of deadlock. However, this scheme requires that programmers, and the DBMS itself, be aware of and correctly utilise the assigned order of resources.

There are several other methods of ensuring deadlock conditions do not arise, and all have similar characteristics. There is a significant reduction in performance or limitation of concurrency with all methods, because of the restrictions in how locks are allocated to transactions.

An alternative way of handling deadlock is to allow locks to be allocated in the most efficient manner possible (but allowing the possibility of deadlock) and to detect a deadlock state if one arises. Once detected, one of the deadlocked transactions is aborted and restarted once the other deadlocked transaction(s) have completed. This method is particularly useful in situations where transactions are typically short, and it is unlikely that concurrent transactions will access the same database items.

An elementary way some DBMS detect deadlock is to merely monitor transactions, and any transaction which has done no work for a predefined period of time is assumed to be in a deadlock state. The more complex way of detecting deadlock is to maintain a **wait-for-graph**. This graph stores a connection between transaction  $T_i$  and  $T_j$  if transaction  $T_i$  is currently waiting for  $T_j$  to release a locked resource. If a cycle exists in the graph, a deadlock state exists and one of the transactions in the cycle must be terminated to break the deadlock.

Once deadlock has been detected, a **victim** transaction is selected. There are many victim selection algorithms, but in general it is undesirable to select as a victim a transaction which involves many operations and has already been running for a long time. The most efficient choice of victim would be a short transaction which has just begun execution. Once selected, the victim transaction is then terminated and rolled back. In some DBMS, the victim transaction is then restarted from the beginning, assuming that the conditions which caused the deadlock will have changed and the transaction will complete successfully the second time. Other systems simply issue an exception code which informs the user or calling program that the transaction was a deadlock victim and did not execute. Such systems rely on the users or programs to handle this situation in an appropriate way.

### 3.1.5 Concurrency in GIS

The majority of database applications involve transactions which can be accomplished in a very short time. An airlines reservation system, or a banking application, for example, might involve edits which can be accomplished in seconds. Such transactions are called **short transactions**. The locking mechanisms we have discussed here are effective in these cases, and they ensure that a consistent view of the database is always presented to users, and that changes are made in a well-considered order. Many GIS operations, such as changing the name of a land parcel owner, are also short transactions and can benefit from concurrency management methods previously discussed.

There are many transactions in a GIS, however, which are far more time-consuming, perhaps involving many interrelated records. Editing spatial data can be complex; creating a new subdivision plan or preparing a mine reclamation plan, for example, require edits to many feature

classes which may be spatially related. Such operations are called **long transactions**, and they may take place over several hours or several weeks. They may involve several short transactions, but together they form a cohesive set of operations in a single, long transaction. Long transactions do not lend themselves well to the standard resource-locking mechanisms because they tend to tie up large geographic areas with many feature classes for long periods of time.

Early GIS applications tended to be based on CAD systems, where spatial data may have been stored as a series of mapsheet drawings. In this arrangement, users would have exclusive use of a single mapsheet for the course of the transaction. In this manner, there was no need for concurrency controls because each sheet became functionally a single-user database. The drawback, of course, is that handling any object which spans a mapsheet boundary becomes difficult.

As GIS applications increasingly moved to commercial RDBMS, they gained the ability to store large, seamless databases. The RDBMS provided significant performance, security and data integrity capabilities, but still did not provide support for the long transactions typically used in many applications of GIS. Some spatial data formats, such as the ESRI shapefile or coverage, are functionally using table-level locking, since only one user may edit a given feature class at one time.

Other GIS vendors chose to implement long transactions using what is sometimes called Check-Out/Check-In. This mechanism allowed users to define a spatial extent and a set of one or more feature classes to check out, or make a personal copy of. Users could then update their subdivision plan, or redesign a road interchange, in their checked out copy in a single-user mode. Other users are locked out of the area in question for the feature classes which were checked out. Once edits are complete, the changes are checked in and updates are made to the master database. Functionally, this mechanism is a combination of mapsheet locking and row-level locking.

Check Out/Check In has the benefit that the checked out databases may reside locally on technician machines, taking the processing load away from the server. The only work that the server is required to perform is the checking in and out processing. The problems with this mechanism are that the check out and check in processes can be very time-consuming, and it can be difficult to manage relationships between feature classes to ensure that all affected features are checked out correctly. This method also can lock users out of large geographic areas for significant periods of time.

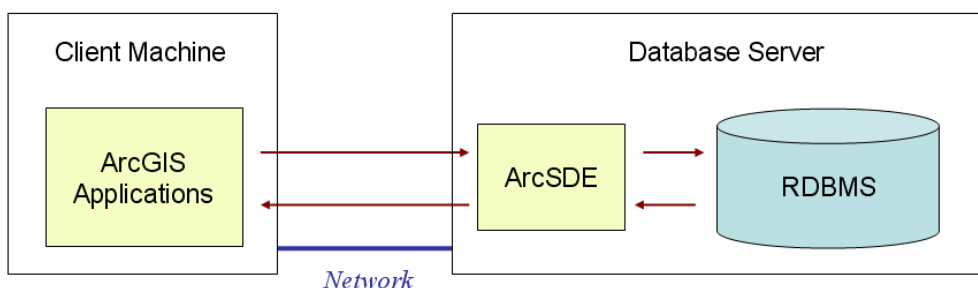
The alternative approach chosen by others (Smallworld, ArcSDE) is often called versioning. In versioning, concurrency is managed by isolating each transaction and reconciling each after its completion. Conflicts are detected and user interaction may be necessary to resolve problems. We will discuss ArcSDE's method of managing concurrency in the topics which follow.

### 3.2 The ESRI Multi-User Geodatabase

This topic explores the multi-user implementation used by ESRI. The ArcGIS suite of applications implement concurrency using software called ArcSDE (Arc Spatial Database Engine), using a technique called versioning. This topic will explore both of these elements in order.

#### 3.2.1 What is SDE?

In module 2 we introduced ArcSDE geodatabases, stating that they are a type of geodatabase that store their information in an industry-standard RDBMS such as DB2, Oracle, SQL Server or Informix. The phrase “ArcSDE Geodatabase” implies that our application data will be stored within ArcSDE, which is misleading. ArcSDE is software, and it does not store, nor manage data. It allows our ArcGIS applications such as ArcMap and ArcCatalog to communicate with the RDBMS in which the GIS data are physically stored. Figure 3-6 shows a simple illustration of how ArcSDE functions.



**Figure 3-6 ArcSDE Communication**

When a client application, such as ArcMap, requires access to data stored in an SDE geodatabase, it communicates with SDE. SDE then converts this request into the necessary SQL for the RDBMS in use (e.g., Oracle) and forwards the request to the RDBMS. SDE then takes the returned data from the RDBMS and passes it to the client application for display or analysis. By using the GIS application to manage display and analytical functions, and the RDBMS to manage the storage, retrieval and maintenance functions, ArcSDE takes advantage of the strengths of both software applications.

ArcSDE is the conduit for communication between ArcGIS and the spatial database, so it handles any differences between the supported RDBMS's. There can be significant differences between these database server applications, including data types, support for spatial data types, indexing and database constraint implementation. SDE ensures that regardless whether the spatial data are being stored in Oracle or DB2, there is a consistent interface to the data. A GIS application built to access data stored in SQL Server using ArcSDE will function the same if used with Oracle, DB2 or Informix.

Since the data are being stored in a powerful, 3<sup>rd</sup> party RDBMS environment, we gain some advantages over the file-based formats such as coverages or the less complex personal or file geodatabases. The primary advantage is data protection; the RDBMS handles the standard data maintenance functions such as backup and recovery, so the data are far better protected. In addition, there is the ability to store massive amounts of data, yet deliver it to the GIS user without

significant performance loss. As discussed in the previous topic, we also gain the ability to handle many concurrent users.

There are three levels of ArcSDE.

- |                   |  |
|-------------------|--|
| Personal ArcSDE   | This basic implementation of ArcSDE is included with ArcGIS. It relies on the SQL Server Express RDBMS, which is supplied free of charge. SQL Server Express imposes several limitations, including that the database be limited to 4GB. Personal ArcSDE does not support concurrency, only allowing one editor at any given time, but will allow a small number of concurrent read-only users. Users can administer Personal ArcSDE using ArcCatalog, so there is no need for additional RDBMS administration skills or experience.   |
| Workgroup ArcSDE  | As with Personal ArcSDE, Workgroup ArcSDE relies on the freely distributed SQL Server Express and may be administered with ArcCatalog. Again, the 4GB limit on database size is in place, but this level of ArcSDE supports full concurrency for up to 10 simultaneous users.  |
| Enterprise ArcSDE | This is the full-function ArcSDE which runs on Oracle, SQL Server, DB2 and Informix. This level of SDE can support any size of database, and any number of simultaneous users. With Enterprise ArcSDE, users supply their own RDBMS license, and must manage and administer the database using the tools supplied with the DBMS. Enterprise ArcSDE implementations often require a database administrator (DBA). Due to the additional costs of an RDBMS and the likelihood of requiring a DBA to manage the geodatabase, the enterprise SDE level can be significantly more expensive than the SQL Sever Express-based options. |

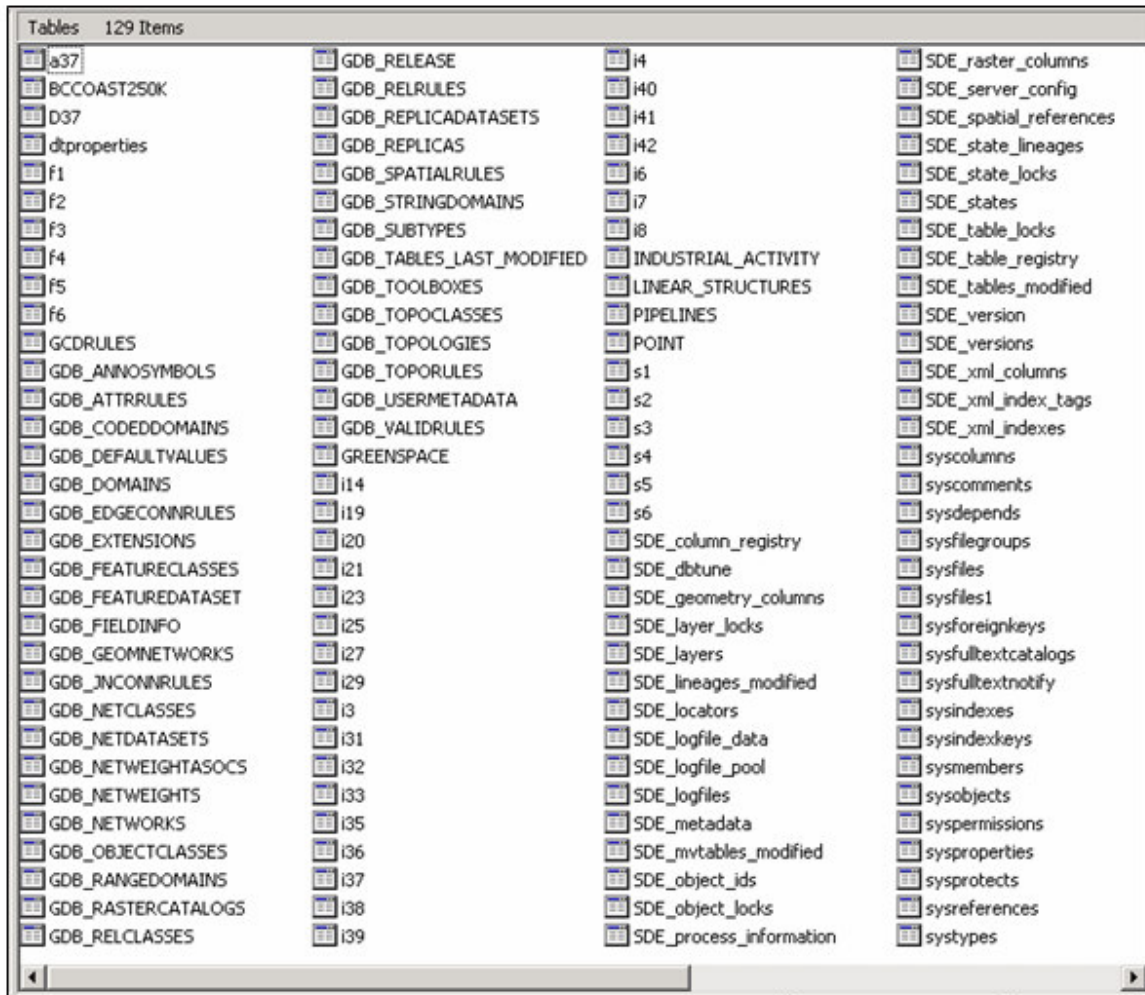
In the topics which follow, we will be addressing concurrency specifically, so it is relevant to Workgroup and Enterprise implementations only.

ArcSDE works by adding several tables to the geodatabase. An SDE geodatabase stored in SQL Server, for example, will include the user-defined tables to represent feature classes and non-spatial tables, the geodatabase tables which are used to manage geodatabase elements such as domains and subtypes, SDE tables which are used to manage SDE and versioning functions, and tables required by SQL Server itself to manage such things as users, permissions and indexes.

For example, Figure 3-7 shows the tables present in a small SDE geodatabase stored in SQL Server. Tables such as *INDUSTRIAL\_ACTIVITY* and *PIPELINES* are the user-defined tables used for two feature classes. Tables prefixed with *GDB\_*, such as *GDB\_Domains*, are used to manage geodatabase elements. Tables prefixed with *SDE\_* are used strictly by SDE, and tables prefixed with *sys* are internal tables used by SQL Server. In addition, the tables with very short, cryptic names such as *f1*, *d37*, and so on, are internal SDE tables. These tables serve a variety of functions:

- |                |   |
|----------------|---|
| Feature tables | Prefixed with the letter “f”. These tables store the feature geometry and other metadata about each feature class. Each numbered f table, such as <i>f5</i> , corresponds to a different feature class. |
|----------------|---|

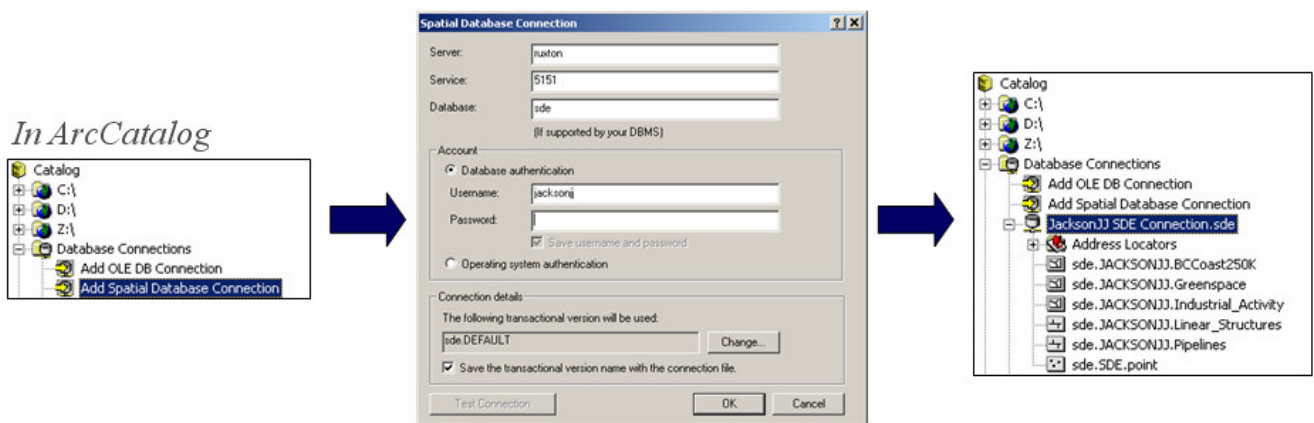
Spatial indexes	Prefixed with the letter “s”. Stores the spatial index for the feature class. As with feature tables, each s table applies to a different feature class.
Delta tables	Prefixed with an “a” or a “d”. For versioned data, these tables store additions or deletions, respectively, for each feature class.



**Figure 3-7 SDE Geodatabase Tables in SQL Server**

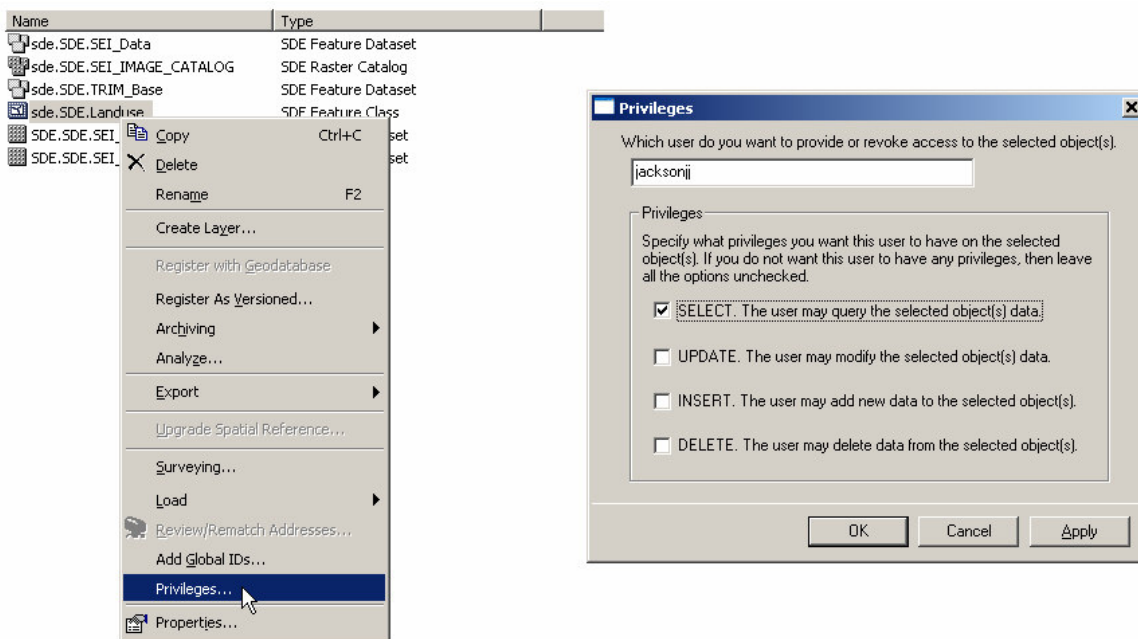
These tables are created and managed by SDE or the RDBMS itself, and users should avoid manipulating anything in a geodatabase using the DBMS directly. Doing so may cause problems with one or more feature classes in the geodatabase.

Security for an ArcSDE geodatabase is managed by the RDBMS, so to use an SDE geodatabase, users must connect to, or log into the database by supplying a username and password, or indirectly by using the same username and password as is used to log into the operating system. Once connected to the database, users may interact with the feature classes and objects within it as they would any other geodatabase. ArcCatalog may be used to connect to an SDE geodatabase, as shown in Figure 3-8.



**Figure 3-8 Creating a Database Connection using ArcCatalog**

Each feature class in an SDE geodatabase may be given different access permissions. By default, when a new feature class is created in a geodatabase, the user that created the feature class will have full privileges (i.e., both read and write permissions) and all other users of the geodatabase will have no privileges. Unless explicitly granted permissions, other users cannot even “see” new feature classes. For each user which should have permission to access a feature class, the creator of the feature class must grant relevant permissions. Figure 3-9 shows the use of ArcCatalog to grant such permissions.



**Figure 3-9 Setting Feature Class Permissions in ArcCatalog**

### 3.2.2 Versioning

We have discussed how concurrency is managed for short transactions using resource locking. However, this sort of solution is ineffective for the long transactions typically found in GIS

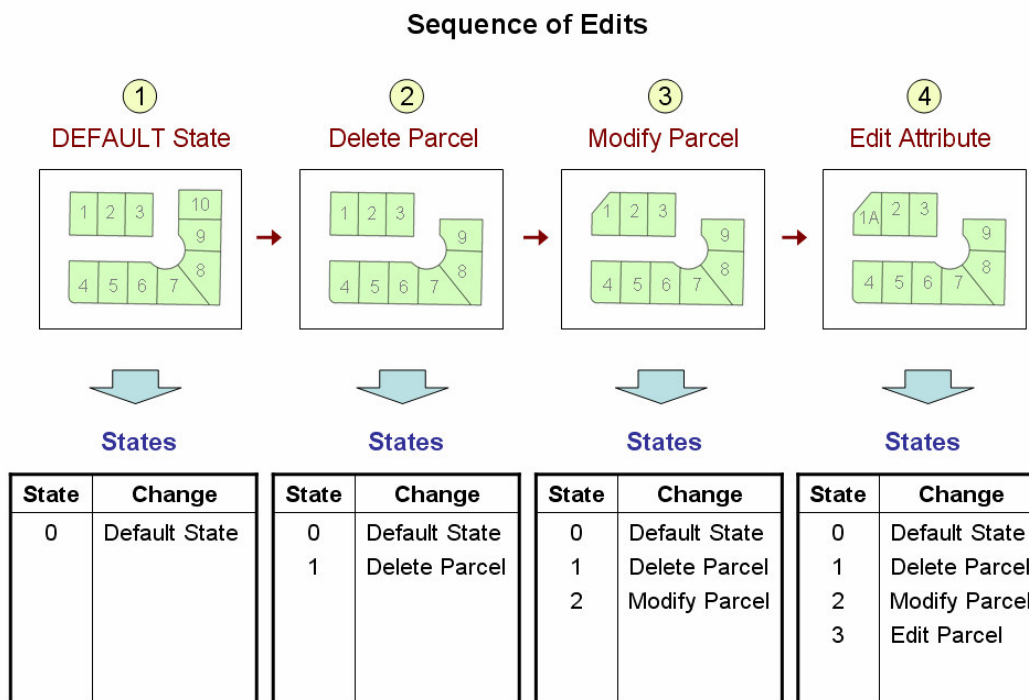
operations because it would require significant resources to be locked for very long periods of time. The alternative approach chosen by ESRI is called **versioning**. Versioning supports concurrency by creating a “version” of the geodatabase which stores an independent view of the database. This version incorporates changes to the database without creating separate copies of the data. Several versions can exist at any one time, and may support multiple users.

This approach can be thought of as an **optimistic** concurrency management method. Resource locking is a pessimistic approach – it assumes that conflicts will occur and takes steps to manage data integrity problems before they occur by stopping a second user from accessing affected resources. Versioning is optimistic because it does not apply resource locks. As a result it does introduce the possibility of editing conflicts, but it does not restrict the use of significant portions of the database while a long transaction takes place. In the event of editing conflicts, tools are provided to find and manage them when the transaction completes.

Rather than storing separate copies of the database, SDE retains the original data and stores each change made to the original data. The changes are managed using what are called database **states**. States are discrete snapshots of the data as each change is made. Each edit operation, such as an addition, deletion or modification will produce a new state. The SDE database has a DEFAULT version, which represents the state of the database at the time versioning was enabled for the data. This constitutes the base state for the database, and all changes to the data refer back to this DEFAULT state. Each change thereafter is stored in the database as a state.

To manage database states, SDE maintains a variety of tables, among them a *States* table. These tables define each change as it is made, and also store a timestamp indicating when the change was made and the user that made the change. The actual changes are stored in a series of system tables in the RDBMS called **delta tables**. An example of a database with delta tables visible was shown in Figure 3-7.

Figure 3-10 shows a series of edits to a parcel database, and the resulting database states. For the sake of clarity the change at each state is summarized by a single *States* table entry, but in reality the information stored about the change at each state is far more comprehensive and involves several tables.



**Figure 3-10 States During Database Edits**

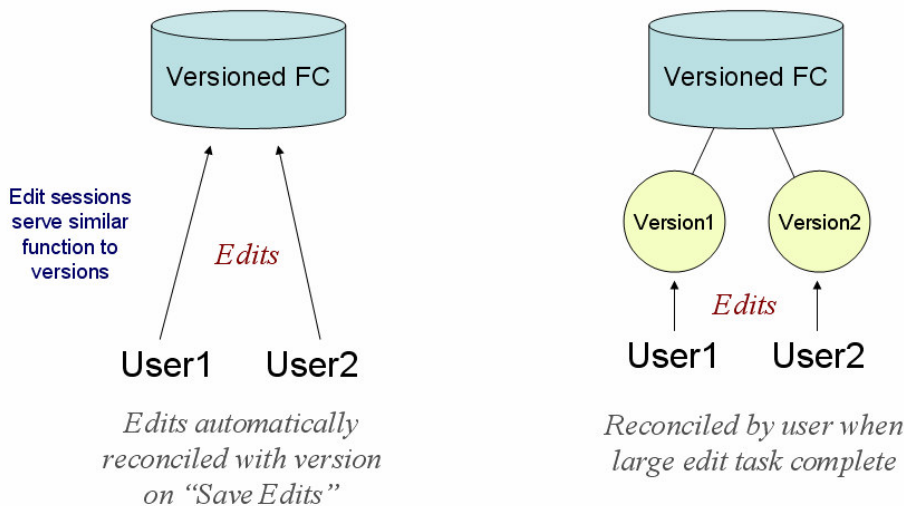
The top left image shows the default state, with 10 land parcels. The *States* table for this default state need not articulate a feature change, so it simply refers to the DEFAULT state. In steps 2 through 4, as each change is made, entries are added to the state and delta tables to reflect each change. After making three changes, there are three states plus the starting DEFAULT state.

The resulting DEFAULT version, and all states for that version, dictate how the database will be represented. For ArcMap to draw the current state of the version, it must query the base state (the DEFAULT version) and each state thereafter in order to make an up-to-date representation of the database.

In order to use versioning to allow multi-user editing with a given feature class, the feature class must be *registered as versioned*. This registration process creates any necessary system tables in the SDE database so that changes can be managed using versions. The registration process also prepares the DEFAULT version. Subsequently, users may create new versions from this DEFAULT version, and work on this version of the database while the original DEFAULT version remains intact and visible to all users. For example, a road designer might create a new version to draft a new highway interchange. The operator might keep this version separate from the DEFAULT version for a period of days or weeks as the interchange is designed. Once the design is complete, or once construction is complete, the design or as-built version may be committed to the DEFAULT database for all users to see and use, overwriting the previous representation of the interchange. The process of writing version changes back to DEFAULT is called **posting** a version. Versions can also be created from other versions, so different design options might be explored in separate versions without affecting the master database.

Versioning is transparent to the user; when editing spatial data stored in ArcSDE, feature classes behave in the same way whether they are versioned or not. With a versioned feature class,

changes made by different users will be **reconciled** when the editing is done. Reconciling edits is the process of checking recently completed edits to ensure that no conflict exists due to edits made by another user to the same features. For short transactions, users may edit a versioned feature class directly in the master database (the DEFAULT version). For longer transactions, a new version of the feature class may be created and worked with for a longer period. These two approaches are illustrated in Figure 3-11.



**Figure 3-11 Multi-User Editing in SDE**

The left side of this figure represents the short transaction workflow. Here, several users simply edit the DEFAULT version of a feature class directly. Each user simply uses an edit session to manage the length of the transaction. The edit session serves a similar function to a version, by isolating the edits from other users until the edit session is stopped and changes are committed to the DEFAULT version. When the user saves any edits in the edit session, a reconcile is automatically done to ensure that no conflicts with other users' edits has occurred.

The right side of Figure 3-11 shows the situation where two users make two separate versions to work with. In this case, users may start and stop edit sessions, perhaps over a period of several days or more, without the changes being made to the DEFAULT database. Saved edits in this case are merely committing changes to the version being worked on. When the long transaction is complete, each user will explicitly request that all changes to their version be reconciled against the DEFAULT version.

Users may create their own versions of feature classes they have permission to access, or an administrator may create a version for them of a feature class the user does not have access to. As previously stated, versions may be created from the DEFAULT version, or from other versions. As with feature classes, versions are given permissions which reflect how other users may interact with the version. Unlike feature classes, however, permissions are not set for specific users, but for all users at once. Permissions for a new version may be set to one of three possible values:

- Private      No other users may access the new version.
- Protected    All users may view the version, but cannot make changes to it.
- Public        All users have full access (both read and write) to the version.

Figure 3-12 shows the process of creating a new version, and applying permissions to it.



**Figure 3-12 Creating a Version**

### Workflow Examples

There are a number of different workflows possible using SDE and versions. The following section identifies a few scenarios, and the way in which SDE might be used to accommodate the workflow requirements.

#### **Scenario 1:**

- Multiple concurrent users making simple database modifications – inserting, updating, removing.

Solution:

- Multiple users using separate edit sessions on the same DEFAULT version. If two users edit the same feature, the second one to save is forced to reconcile the changes.

#### **Scenario 2:**

- A user wishes to make significant changes to the data, spanning several edit sessions and perhaps several days.

Solution:

- User creates and switches to a new version derived from the DEFAULT version.
- User modifies features and saves as required.
- When the changes are complete, user must reconcile changes. If conflicts are detected, the user can resolve the differences and complete the transaction by posting their changes to the DEFAULT version.
- The user can then delete the version.

### Scenario 3:

- An organization wishes to embark on a large development project, but needs to keep the existing database intact and available during design and construction

#### Solution:

- Create a version derived from the DEFAULT version, and in that version a design is developed
- Several versions can be developed (from each preceding version) to reflect supervisor changes, changes encountered during construction, etc.
- Final, as-built, version can be posted to DEFAULT and conflicts resolved.
- Each stage of the process can be preserved as historical record, or versions may be deleted at the completion of the project.

### Scenario 4:

- An organization wishes to control write access to a database

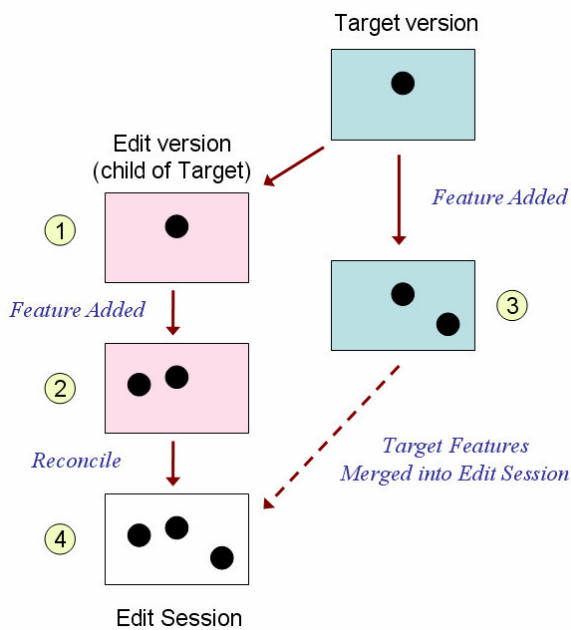
#### Solution:

- The DEFAULT version is write-protected.
- If changes are to be made, an administrator creates a version for editing, in which the necessary users have write permission
- Operators edit this new version as required.
- An Administrator reviews, edits and posts changes back to DEFAULT.
- The version can be deleted at the end of the process.

### 3.2.3 Reconciliation

As stated, versioning places no locks on the SDE database, so there will always be the possibility that two users will make conflicting edits. As a result, once edits are complete all edits made by a user to a version must be reconciled against the parent version (the version the edited version is based on; usually DEFAULT, but not always). The parent version in the reconcile process is called the reconcile **target**, because it is where the new changes are to be placed.

Reconciling must always be performed during an edit session. With the current edit version being edited in an edit session, users may begin the reconcile process. The first stage of the reconcile process is to merge all the features and objects from the target version into the edit session so that the edit and target versions may be viewed together. The vast majority of objects will probably be identical in both versions, so much of the merge process will not affect the users view of the feature class. However, if for example, a new feature were added to the target version since the edit version was created, this new feature would be added to the edit session when reconciling is begun. Figure 3-13 shows this initial merge process. The yellow-circled numbers represent the sequence of events leading up to the reconcile.



**Figure 3-13 Merge Stage of a Reconcile**

At time 1, a new edit version has been created, with the target version as its parent. Before beginning edits, the edit version and the target version are identical. At point 2, a new feature has been added to the edit version. At some point after the edit version was created, but before a reconcile is initiated (shown as point 3), another user has added a third feature to the target version. When the edit version is reconciled, the merge process brings the new feature from the target version into the edit session and presents it together with the current edit version. This we can see at time 4, where all three points are presented together. Deletions and modifications would appear in the reconcile edit session in the same manner. The operator doing the reconcile operation now has the opportunity to review all changes that have been merged into the edit session. Where an object has been edited in both versions, the conflict will be identified for the editor.

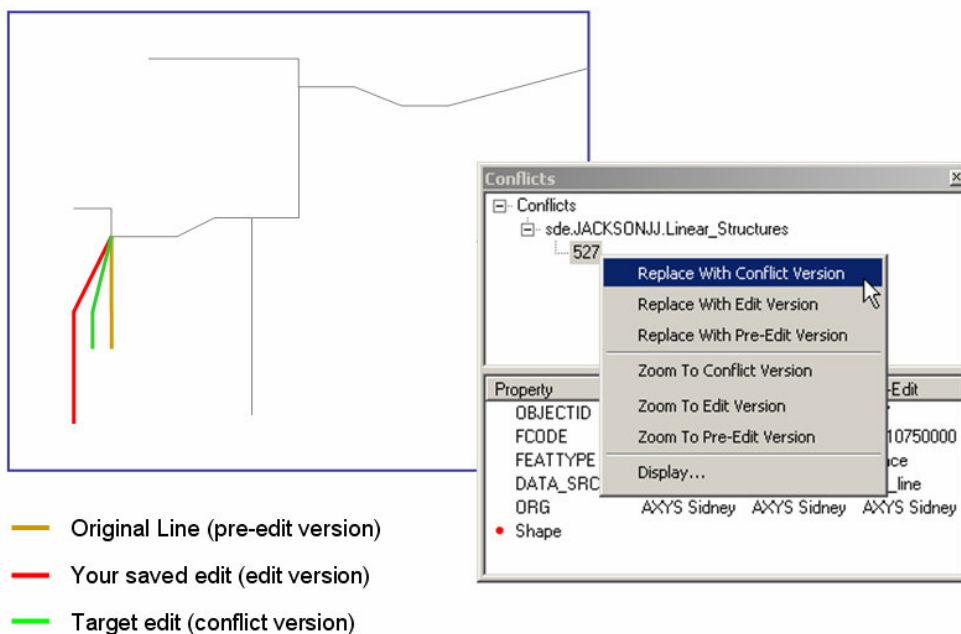
Where no changes have been made to the target version since the creation of the edit version, the edit session would show simply the edit version. Where changes are made to the target version, but no changes have been made to the edit version, the edit session will simply show the target version, since all target changes will be integrated into the edit session. Where changes are made to both the target and the edit versions, both sets of changes will be incorporated together in the edit session (the case indicated in Figure 3-13). Where changes have been made to the same object in both versions, a conflict will be indicated to the editor in the edit session.

The exception to the optimistic concurrency management used by versioning occurs during the reconcile process. During a reconcile, and for the duration of the reconcile until a *save edits* or *version post* is performed, an exclusive lock is placed on the version. While locked, other users may not view or edit the version while the reconcile is taking place. A version that is being edited by one user cannot be reconciled by another user, since an exclusive lock cannot be granted in that case. In addition, a shared lock is placed on the target version during a reconcile. This ensures that while other users can view the target version, two concurrent reconcile processes cannot be performed.

### 3.2.4 Conflicts and Conflict Resolution

The initial merge stage of reconciliation handles trivial changes, where a change is made in one version and no change was present in the other. This includes situations where a feature is deleted or modified in one version and left intact in the other. True conflicts, where an object is explicitly modified in different ways by two users, must be handled manually. In a manner similar to the process of fixing topological problems, each conflict may be visited in turn and a decision made about which change should be retained in the finished, reconciled database. Tools are provided to move between conflicts, and to select the edit to be retained.

For example, suppose you create a version in which to make edits to a linear pipeline feature class. You edit a pipeline feature, and change its shape slightly. While you are doing this, a second operator also edits this same feature, changing its shape in a different way. This second edit may have occurred directly in the DEFAULT version, or another version may have been created, edited, validated and posted before you validate your changes. Once you complete your edits and perform a validation, the changes made to the target (DEFAULT) version since you created your edit version will be merged into your edit session, and a conflict will be detected. Using an interface for managing conflicts, you can zoom to each and assess the problem. Figure 3-14 shows how the interface might present the conflicting pipeline edit described above.



**Figure 3-14 Resolving Conflicts**

The map view shows all three of the conflicting representations of the same pipeline feature. It shows the way the pipeline was originally drawn before you created your edit version (the pre-edit version), the change you made (the edit version) and the change made by the other user (the conflict version). You must now decide which representation of this pipeline feature is correct using the conflicts dialog. The default choice, if you do not tell the application otherwise, is that the conflict version takes precedence. This means that unless you tell ArcMap otherwise, your change will be discarded in favour of the edit made by the other user and previously reconciled and posted to the DEFAULT version. You must explicitly visit each conflict, and choose the correct version of

each edit. This will include spatial conflicts as discussed in this example, as well as attribute conflicts, such as two editors changing the same parcel owner to different names.

One can see that the order of edits and validation makes a significant difference to how the validation process happens. Where two editors make changes to the same set of features, the first editor to validate and post will encounter no conflicts. It is the second editor to validate who must review all conflicts and make decisions regarding the correct representation for features. To a certain degree, conflicts can be managed in the workflow of editors. Managers can vastly reduce the likelihood of conflicts by ensuring that operators seldom work on the same geographic extent, or with the same feature classes. In addition, managers can control how conflicts are handled by directing the order of edit posting such that perhaps a more senior person is the last person to validate changes, thus theoretically improving the data quality by having someone with more experience resolve conflicts.

There are primarily two kinds of conflict when validating:

- The same feature is modified in both versions. Either the spatial representation or an attribute value is explicitly edited by two different operators, with two different results.
- The same feature is modified in one version and deleted in another. One operator explicitly changes either an attribute value or the spatial representation of a feature, and the other operator deletes the feature. The act of changing an attribute or shape of a feature is an implicit way of stating that the feature should be present in the database, which means another operator deleting that feature is a conflict.

This differs from the case where one operator deletes a feature, and the other operator does not delete it (but doesn't change it, either). In this latter case, the operator who leaves the feature intact hasn't indicated in any way that the feature *shouldn't* be deleted, so it is treated as a trivial change and no conflict is presented. This is perhaps a limitation of the validation process, but if operators are aware of this behaviour, they may review deletions during the merge process of reconciliation.

The conflicts discussed so far constitute problems with simple features; features that are changed in isolation and do not affect other related features. In a geodatabase, there are many ways features may relate to other features, and these relationships mean that edits to one feature can affect other features not explicitly edited. Feature relationships which can affect conflict resolution are:

- Feature-linked annotation
- Relationship classes
- Geometric networks
- Topology rules

### Feature-Linked Annotation

Feature-linked annotation is a method of having text labels based on feature attributes in a map be directly tied to the geometry of the feature. Annotation text will be updated or moved if the attribute value is changed or the feature is moved, for example. This presents added complexity to the version reconciliation. If one operator changes a feature's position (which will trigger a

commensurate annotation move) and another operator moves the annotation itself, but leaves the feature intact, this triggers a reconcile conflict. In this situation, both the feature position and the annotation position will be flagged as a conflict. It is not possible to accept the feature move from one version and the annotation move from the other version. In some situations, you may need to reposition annotation after resolving a conflict.

### Relationship Classes

Relationship classes produce problems in reconciliation that are similar to feature-linked annotation. The primary source of concern is the effect the source feature in a relationship can have on destination features, in the same way as a feature change can affect a feature-linked annotation. If a source feature is deleted it may trigger a *cascading delete* which removes a number of related features. You might delete or move a water main, and have all the related valve features also be removed or repositioned. A conflict would arise if one operator deleted the water main (and thus all the valves), while another operator altered the attributes of one valve related to that water main.

### Geometric Networks

Geometric networks provide a way to model resource flows through networks, such as water distribution networks, stream networks, or electrical networks. They use edges, junctions and connectivity rules to represent and model network behaviour, and maintain a logical network which defines connectivity between network features. The connectivity maintained in this logical network is what can affect reconcile conflicts.

For example, if a user adds a water pipe lateral (which branches off from a main) to a water main feature, the main is not physically changed; that is, the main feature is not broken into two pieces where the new pipe joins. However, the logical network representing this water main is split into two pieces, indicating the connectivity where the new lateral was placed. Because the logical representation of the water main has changed, any change to this main feature in another version, such as an attribute or spatial change, will produce a conflict.

### Topology

In module 2 of this course, we discussed topological rules, and their validation. As part of topology creation, you will recall, the cracking and clustering process ensures that features lying very near each other are restructured to share precise locations. This sharing of geometry between feature classes that participate in a topology means that spatial changes to one feature may trigger changes to features which share that geometry. These new changes may occur to other features in the same feature class, or to features in other feature classes.

As a means of managing these triggered changes, ArcMap records edits to features participating in a topology, and creates *dirty areas*, or areas for which geometry has been changed but topology has not been validated. Even where topology in each version has been validated, the reconcile process after editing may introduce new topology problems. As a result, during the reconcile process all topology dirty areas are reinstated so that topology may be revalidated for these areas to ensure that no new topological errors have resulted from edits made to the two versions. Where edits are made to feature classes participating in a topology, topology validation should always form part of the reconcile process before posting changes to the target version.

## Posting

Once all conflicts have been reviewed and appropriate decisions made about each conflict, the version may be posted to the target. Once posted, all the trivial edits (i.e., edits with no conflict), and all edits retained during the conflict resolution process, will be permanently written to the target version (often the DEFAULT version). Once changes are posted, users may choose to delete their edit version, or retain the version as a historical reference.

### **3.2.5 Non-Versioned Multi-User Editing**

In version 9.2 of ArcMap, support has been added to provide multi-user editing without the concurrency control of versioning. As mentioned, versioning does not create a separate copy of the version, but merely refers to the parent version and then stores any subsequent changes to that parent version. As a result, the true representation of a version is never truly represented in a concrete way in the database. If a 3<sup>rd</sup> party application which was not written with the intention of interfacing with database states and delta tables wished to access data which is stored in a version, it would never see the true current state of the version. Versions can also add unnecessary overhead to a database where only small, simple changes to the data are necessary. Where feature classes do not participate in topology or geometric networks, it is possible to use a short transaction editing model, which makes use of the locking mechanism used by the underlying DBMS.

### 3.3 Geodatabase Administration

#### 3.3.1 Introduction

One should always devote energy to administration of GIS data. With some data formats such as coverages or shapefiles, the administration is fairly limited, but it should always involve managing backups and ensuring that metadata is current. With geodatabases, however, there can be more significant work involved in managing geographic data.

Backup and metadata maintenance functions are universal, so we will not address them here beyond identifying them as necessary tasks. In this topic, we will discuss tasks which are necessary specifically to maintain the geodatabase structure. The more users we have, and the more large and complex your geodatabase, the more administration work is necessary. Like any other function in an organisation, when we reach a certain size, administrative tasks become significant.

Table 3-1 summarizes the different tasks which must be performed by a geodatabase administrator. The different types of geodatabase are presented across the top of the table, with the size and complexity of the geodatabase implementation increasing from left to right. One can see fairly quickly that even simply the number of tasks to be performed for the enterprise geodatabase is significantly larger than the number of tasks for a personal geodatabase. In addition, the complexity of the maintenance tasks is often greater for enterprise databases than for the more basic implementations.

**Table 3-1 Summary of Administration Tasks for Geodatabase Types**

Task	Personal/File Geodb	Personal SDE	Workgroup SDE	Enterprise SDE
Installation	N	Y	Y	Y
Configuration	N	N	N	Y
Compression	Y (feature class compression)	Y (version compression)	Y (version compression)	Y (version compression)
Compaction/Shrink	Y	Y	Y	Y
User Accounts & Permissions	N	Y (limited to 3 users)	Y	Y
Backup and recovery	Y	Y	Y	Y
Rebuild indexes	N	Y	Y	Y
Rebuild statistics	N	Y	Y	Y
Tuning	N	N	N	Y

The tasks in this table are defined in detail in the sections which follow.

### 3.3.2 Installation and Configuration

These tasks relate to work which must be done prior to using a geodatabase. This will involve installation of additional software (beyond the ArcGIS Desktop applications) and preparing any additional settings required by ArcSDE. These tasks would typically need to be done only when new software is installed.

Personal and file geodatabases require no installation or configuration. Personal and Workgroup SDE geodatabases require that SQL Server Express be installed, but no further configuration is necessary.

For enterprise SDE, the installation process is more complex, involving three steps. The steps are necessarily different, depending upon which RDBMS your enterprise SDE will be based upon. The basic steps, however, are similar. First, the underlying RDBMS (e.g., DB2) must be installed, environment variables set, and in some cases a new, empty database must be created. The second step simply involves installing ArcSDE, following the installation guide for the correct DBMS and operating system. During the third step, post-installation, the geodatabase schema is created, an SDE administrator account is prepared and granted appropriate permissions, and an SDE service is started. The service is a process which “listens” for users requesting a connection to the geodatabase.

As part of the configuration process, a variety of operating parameters will be set which dictate things such as the location of tables and the size of the temporary space, and how data should be stored and accessed. For example, in some DBMS you can control how physical storage of spatial objects is handled. In all systems you can utilize compressed binary formats (e.g., Binary Large Objects; BLOB data type), and in some systems you may use data types specifically designed to store spatial data (e.g., ST\_SPATIAL data types from Oracle Spatial or Informix Spatial DataBlade). ESRI supplies installation and configuration documents for each RDBMS, and the process will vary greatly depending upon which database application is installed. In general, the default configuration settings are sufficient to begin using SDE, but as patterns of use and database size begin to become clear refinements can be made to the configuration which may improve performance. Such modifications will be discussed further in the Tuning section which follows.

### 3.3.3 User Accounts

The creation of usernames and passwords for the users of a geodatabase allows control over what type of access users have to the datasets within the geodatabase. The process of identifying a user to the geodatabase prior to gaining access to its data is called **authentication**. Authentication may be handled in two ways with geodatabases, using operating system authentication, or database authentication.

With operating system (OS) authentication, users log into their computer using credentials (username, password) supplied to the operating system. When they use a geodatabase, the OS credentials are passed to the database and they are not required to supply authentication a second time. For administrators, this means that the existing OS usernames must be added to the database, and the database must be configured to accept the OS authentication.

With database authentication, users log into their computer as usual, but they must log in separately to the geodatabase when they wish to use its data. In Oracle and SQL Server, these accounts must be created and maintained using the RDBMS.

Once accounts have been prepared, permissions may be defined for these users. As discussed in the previous topic, each user can be given different access rights to each feature class or other database object. For example, one user may be given read-only rights for streets, but full write permissions on the land parcels. Another user might not even be able to see the existence of the land parcels, but would have full permissions on the street data. Using this mechanism, administrators can control which of many users accessing the data is to maintain a specific dataset, and which may be a user of the data.

In addition, administrators may create **groups** of users to make applying permissions easier. Rather than giving each user permission to read or write every feature class or table, a set of group permissions may be created and the same permissions easily applied to a number of users. For example, two groups could be defined, such as “Researcher” and “Editor”. A Researcher might have read-only permission to only a few database objects, while an Editor might have full write permission on all objects. Once these groups are defined, and their permissions for each database object set, new users may simply be added or removed from these groups to have them receive the appropriate privileges.

Personal and file geodatabases require no user account administration, because they do not support concurrency. As discussed in module 2 of this course, personal geodatabases functionally implement database locking, since at most one user may edit a geodatabase. File geodatabases improve concurrency support slightly, allowing one editor per feature dataset in a geodatabase. In both file and personal geodatabases there is no support for setting separate permissions for feature classes. Users are not authenticated in any way.

Personal SDE geodatabases require a small amount of user administration, since only a small number of users may concurrently edit the geodatabase. This type of SDE database would only be appropriate for very small organisations or project teams, so in all likelihood only a small amount of work would be necessary to define user accounts and permissions. Workgroup and Enterprise SDE installations will require significant oversight, since they can support very large organisations with many users having different types of access rights.

### 3.3.4 Compression

Vector feature classes and non-spatial tables stored in a file geodatabase may be compressed. This is an optional process that simply allows these objects to occupy less storage space. When compressed, data are in a read-only state. They appear as they normally would in ArcCatalog or ArcMap and retain approximately the same display and query performance, but they cannot be edited while compressed. Compression of data can vary from a negligible change in storage space to an improvement of better than 4:1, depending upon the nature of the data being compressed. Compression is an optional administration task, and feature classes may be uncompressed at any time to allow editing.

In an SDE database, compression performs a different function. With all ArcSDE geodatabase types (personal, workgroup or enterprise) compression will reorganise data within the geodatabase that relate to versioning. As users make changes to an SDE geodatabase using versioning, remember that the changes are not actually made to the database objects; all changes made to

every version become database states, and are stored in the delta tables. If large numbers of edits are made, the number of states and rows in the delta tables can grow to the point that database performance suffers.

An SDE compress collapses any set of states into one state that will not affect the representation of the version, and where possible it moves rows from the delta tables to the base tables. The best possible gain from a compress operation is if all versions have been reconciled and posted, and the versions themselves deleted. In such a case, the delta tables may be emptied completely, and only one state (the DEFAULT state) need remain. In practice, several versions may still be active, but a compress operation will still be able to improve database performance.

An SDE compress should be performed as frequently as daily, in working environments where significant edits are made to the data on a regular basis. ESRI recommends that even for average to low edit volumes, a compress should be performed weekly.

### 3.3.5 Compaction

Compaction, though similar conceptually to compression, is administratively very different. As you add and delete data from geodatabases, data elements get out of order and empty space develops in the file system. The longer this process continues, the slower data retrieval becomes, and the more space the geodatabase will occupy. Compaction will reorder data in the geodatabase to place data objects in a more efficient order and remove references to unused space very much like the disk defragmentation process you might perform using your computer's operating system. For databases which are frequently modified, compaction should be performed monthly.

In the file and personal geodatabase, this process is called compaction. In personal and workgroup geodatabases, the corresponding function is called **shrinking** the database. In an enterprise geodatabase, this function is controlled by the underlying DBMS.

### 3.3.6 Database Tuning

Tuning a database is the process of improving the efficiency of the system by maximizing the use of system resources. The larger the user community for a given geodatabase, the more tuning the administrator must perform, and the greater the improvement in performance that will be felt by effective database tuning. In personal and file geodatabases, as well as personal and workgroup SDE geodatabases, there is very little an administrator can do to improve performance. There are certainly maintenance tasks which can be effective (e.g., compression), but these are primarily maintenance functions and not tuning in the true sense. Because an enterprise geodatabase makes use of a full-function RDBMS, however, there are a significant number of ways to alter the way the DBMS works.

The methods and tasks involved in tuning can vary greatly between the different supported DBMS's, but in general, the following tasks fall under the general category of tuning:

- |              |  |
|--------------|--|
| Input/Output | When many users are attempting to access data files which reside in the same physical location on disk, these users are competing for the same resource. This is called I/O contention. We can reduce contention by storing commonly used files on different disk locations. |
| Memory       | Many functions in a DBMS involve the use of memory, including the use of virtual memory, buffering of data and the use of temporary space. In many   |

	cases, the size of these units of memory may not be effective when the default DBMS values are used.
Indexes	Both spatial and non-spatial indexes can greatly influence the efficiency of any data retrievals. We will discuss spatial indexes during the next module of this course, but essentially it is a way of quickly finding features based on their location, much as you can quickly find topics in a book by using the index to find the relevant page numbers. The way the spatial index is defined can greatly impact the speed of geographic query and processing.
Views	A view is a stored query which may take information from several tables or objects and combine them for use at a later date. Where users find themselves performing similar queries frequently, it may be helpful for the administrator to prepare database views.

Tuning can be a complex topic, and administrators should refer to the SDE Tuning Guide for the appropriate RDBMS in use.

### **3.3.7 Updating Statistics and Indexes**

Database applications expend significant efforts to ensure that the execution of queries is very fast. One of the things which the DBMS considers when deciding how to execute a given query is statistics. Statistics will include a variety of information about the database itself, such as the number of records in a given table, the number of data pages in each table, the physical location on disk of each table and whether there are indexes for a table. Based on these statistics, the DBMS can optimize the internal strategy for dealing with a given query. If these statistics are not up-to-date, the query optimizer cannot effectively do its job.

You should update database statistics before and after a compress operation, after you add or remove topology rules, and after you have finished importing, loading, or copying data into an ArcSDE geodatabase. Statistics must be updated for all SDE databases, but is not possible for personal or file geodatabases.

Adding and deleting data, and database compression can fragment indexes used by the DBMS to quickly find records. Although the performance loss due to fragmented indexes is not large, administrators may wish to periodically update them. With personal and workgroup SDE databases, this may be performed using ArcCatalog. Users should consult the documentation for the appropriate DBMS for further information on updating enterprise SDE indexes.

## **Module Self-Study Questions:**

1. Why do RDBMS's not simply allow uncontrolled access to a database by several users? What sorts of problems might be encountered?
2. What advantages are there of making use of existing, industry-standard database applications (e.g., Oracle, SQL Server) as a storage method for GIS data?
3. Describe what versioning is, and what purpose it serves.
4. Why is versioning called an Optimistic concurrency management method?
5. If you were the administrator of a Workgroup ArcSDE geodatabase, what administration tasks would be necessary, and how frequently might you perform them. Assume you have a fairly small user-community with a low volume of database edits.

## References

Date, C.J. *An Introduction to Database Systems*. Addison-Wesley, 2000.

Elmasri, R., and Navathe, S.B. *Fundamentals of Database Systems*. Addison-Wesley, 2000.

ESRI. *Building a Geodatabase*. ESRI Press, 2005.

ESRI. *Versioning*. ESRI Technical Paper, 2004. Viewed July, 2007.  
[http://downloads.esri.com/support/whitepapers/ao\\_/Versioning\\_2.pdf](http://downloads.esri.com/support/whitepapers/ao_/Versioning_2.pdf).

ESRI. *Understanding SDE*. ESRI Press, 2005.

ESRI. *ArcGIS On-line Help*. Viewed July, 2007.  
<http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm>

Kroenke, D.M. *Database Processing Fundamentals, Design and Implementation*. Prentice-Hall, 1998.

Zeiler, M. *Modeling Our World*. Redlands, CA: ESRI Press, 1999.

## 4 Additional Topics

Previous modules have followed a path from the theoretical to the applied, beginning with basic database theory, and ending with very specific material regarding the use and administration of specific software applications and databases. This module deviates somewhat from this approach, in that it simply presents a number of topics relating to geographic databases that have not yet been covered, but which require a certain level of knowledge to address.

This module begins with a discussion of the structured query language (SQL), which is central to any use of databases. Topic 2 provides an overview of methods for speeding up access to large geographic databases. Topic 3 investigates the concept of including time in our geographic databases, including several examples of solutions to this problem. Topic 4 discusses some alternatives to the standard 1-server, multiple-clients approach to geodatabase configurations.

### Module Outline

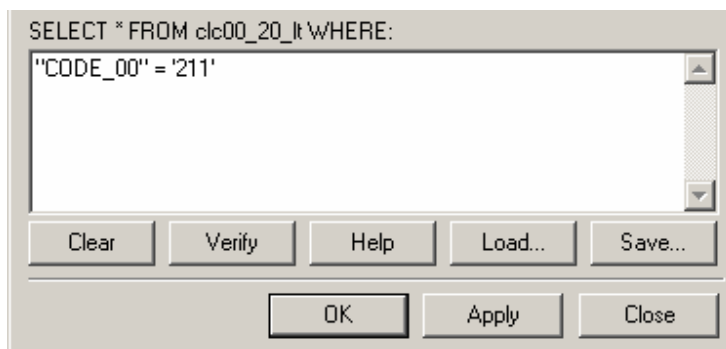
- Topic 1:      Structured Query Language
- Topic 2:      Spatial Indexing
- Topic 3:      Temporal Data Storage
- Topic 4:      Distributed DBMS

## 4.1 Structured Query Language (SQL)

### 4.1.1 Introduction

Structured Query Language (SQL) is the relational model's standard language. It allows users to create database structures, perform data manipulation such as updates, and perform queries to extract data from the tables. Almost all RDBMS software support SQL, and in some cases they have extended that standard functionality with their own language extensions. SQL was developed in the mid-1970's by IBM as the data manipulation language for their relational database management system System R. It has since become accepted by both the American National Standards Institute (ANSI) and the International Organisation for Standardisation (ISO).

Users of GIS applications and programmers of GIS applications will make significant use of SQL. One of the most elementary procedures in ArcMap, the *Select by Attributes* dialog, makes use of SQL to retrieve records from a feature class. Figure 4-1 shows part of this dialog. Programmers who use ArcObjects to create new GIS applications or extensions to existing ArcMap functionality will often use SQL to manipulate data stored within a geodatabase.



**Figure 4-1 ArcMap Select by Attributes Dialog**

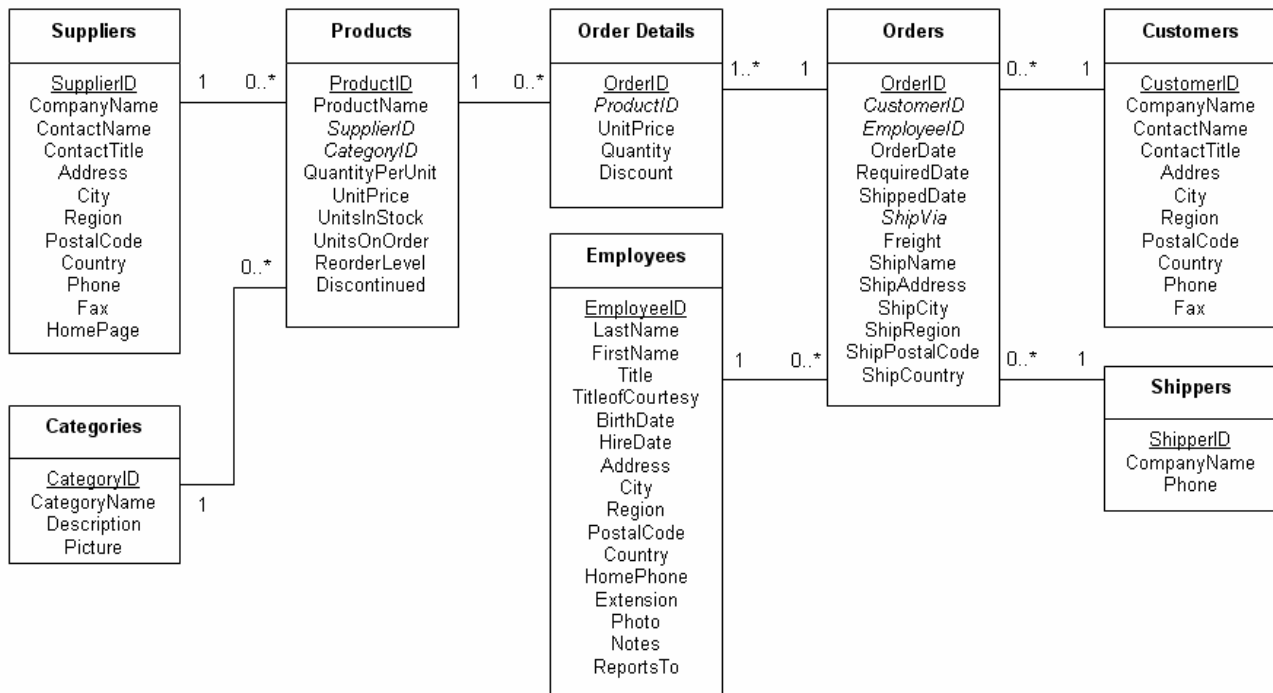
SQL has several characteristics:

- Allows the creation of the database and table structures, using **Data Definition** commands
- Allows us to perform add, delete and modify functions, using **Data Manipulation** commands
- Allows complex queries to transform the raw data into useful information, also using Data Manipulation language
- Easy to use and learn; SQL has a vocabulary of about 100 words.
- Portable, so that the same commands work in all RDBMS. This is mostly true – there are some minor variations between implementations of SQL, producing several **SQL Dialects**. Some commands which will run in Oracle, might require some minor changes to run in SQL Server, for example.

### 4.1.2 Case Study Database

The database used for all of the examples which follow will be Microsoft's Northwind database. This database is supposed to represent a simple database design for a fictional sales organisation.

It includes such objects as *Orders*, *Customers* and *Employees*. Figure 4-2 shows a Class Diagram of the Northwind database.



**Figure 4-2 Microsoft's Northwind Database**

Having completed Module 1 of this course, students should be familiar with the notation used here. This says, for example, that a Customer can have zero or more orders, but an order must be associated with one and only one customer. This diagram also includes the primary and foreign keys, using the same notation used previously with primary keys underlined and foreign keys in *italics*. *OrderID* is the primary key of the Orders table, and the attribute *CustomerID* in the Orders table is a foreign key which allows us to associate the correct customer to each order.

Although this database does not contain geographic objects, and will thus not be helpful when we begin discussing spatial queries, it is useful for the purpose of learning basic SQL for a variety of reasons:

- It is the database used for learning Microsoft database products such as Access and SQL Server. This database can be installed as part of the database application, so many students will have seen this example database before.
- This example database has been in use for at least 10 years, so there are many tutorials and other references which make use of Northwind both in print and on the internet.
- This is a well-considered database, including a wide variety of attribute data types, association types and naming conventions.
- It has been populated with a good collection clear, understandable data, so that queries can respond with reasonable answers.

We will likely only explore a small part of this database during presentation of the theoretical material, but this database lends itself well to practical exercises as well.

There are two types of commands in SQL: data definition commands which allow users to define the database structure, and data manipulation language commands which allow users to insert, delete, update and select data.

#### 4.1.3 Data Definition Commands

The data definition commands allow users to create databases and tables, and are often called Data Definition Language (DDL) commands. This is probably the part of SQL which varies most from one implementation to another, simply because creating tables involves attribute data types and these are often slightly different between RDBMS's. In the examples that follow, we will use the SQL Server data types, so the example commands shown below may not function correctly with other database systems, such as Oracle. However, the differences are minor, and users should be able to adapt the material here to other RDBMS applications with ease. Refer to the SQL documentation for your RDBMS for details on your SQL dialect.

The most straightforward SQL command is CREATE DATABASE. This command creates an empty database which is ready to start creating tables in. It has the syntax:

```
CREATE DATABASE <database name>
```

e.g., CREATE DATABASE MyNewDatabase

The other main data definition command is the CREATE TABLE command. It creates each table, including all attributes and their data types, and can define primary and foreign keys. Its basic syntax is as follows:

```
CREATE TABLE <table name> (  
    <attribute1 name and characteristics>,  
    <attribute2 name and characteristics>,  
    <primary key designation>,  
    <foreign key designations>)
```

For example, to create the Customers table from Figure 4-2, the command would appear as follows.

```
CREATE TABLE Customers (  
    CustomerID      nchar (5)          NOT NULL,  
    CompanyName     nvarchar(40)       NOT NULL,  
    ContactName     nvarchar(30)       NULL,  
    ContactTitle    nvarchar(30)       NULL,  
    Address         nvarchar(60)       NULL,  
    City            nvarchar(15)       NULL,  
    Region          nvarchar(15)       NULL,  
    PostalCode      nvarchar(10)       NULL,  
    Country         nvarchar(15)       NULL,  
    Phone           nvarchar(24)       NULL,  
    Fax             nvarchar(24)       NULL,  
    CONSTRAINT PK_Customers PRIMARY KEY (CustomerID)  
)
```

This can be written all on one line, but programmers typically place one attribute per line to improve readability. In addition, language keywords like `CREATE` are capitalized here to increase readability. We will use these conventions throughout this document.

This table uses only two data types:

<code>nchar</code>	Fixed-length character field, length as specified up to 255. If you store strings that are less than the specified length, the remaining characters are filled with blanks.
<code>nvarchar</code>	Variable-length character field.

As mentioned, the data types used in these examples are SQL Server compliant, and the code presented here may not work correctly in Oracle, for example. In Oracle, `nvarchar` would be replaced by `varchar` or `varchar2`.

The `NOT NULL` specification is a means of forcing the entry of a value for that attribute. A `NOT NULL` specification means that a user cannot enter a new record and leave this attribute empty. The `NULL` specification means that the attribute is optional and may be omitted as necessary.

Some RDBMS, such as MS Access, do not support the `PRIMARY KEY` designation. Simply omit this clause from the command in such cases.

If a table were required to have a composite primary key, this is denoted by separating member attributes with commas, such as `PRIMARY KEY (OrderID, CustomerID)`.

Because the `CustomerID` appears in the `Orders` table as a foreign key, we create `Customers` table before the `Orders` table. The DDL for creating the `Orders` table illustrates the use of foreign key constraints, and would appear as follows:

```
CREATE TABLE Orders (
    OrderID          int          NOT NULL,
    CustomerID       nchar (5)    NULL,
    EmployeeID       int          NULL,
    OrderDate        datetime     NULL,
    RequiredDate     datetime     NULL,
    ShippedDate      datetime     NULL,
    ShipVia          int          NULL,
    Freight          money        NULL,
    ShipName         nvarchar (40) NULL,
    ShipAddress      nvarchar (60) NULL,
    ShipCity         nvarchar (15) NULL,
    ShipRegion       nvarchar (15) NULL,
    ShipPostalCode   nvarchar (10) NULL,
    ShipCountry      nvarchar (15) NULL,
    CONSTRAINT PK_Orders PRIMARY KEY (OrderID),
    CONSTRAINT FK_Orders_Customers FOREIGN KEY (CustomerID)
        REFERENCES Customers (CustomerID),
```

```
        CONSTRAINT FK_Orders_Employees FOREIGN KEY (EmployeeID)
            REFERENCES Employees (EmployeeID),
        CONSTRAINT FK_Orders_Shippers FOREIGN KEY (ShipVia)
            REFERENCES Shippers (ShipperID)
    )
```

This table uses a number of additional data types:

Int	Integer (whole) numbers from $-2^{31}$ to $+2^{31} - 1$ .
Datetime	Date and time data.
Money	Monetary values from $-2^{63}$ to $+2^{63} - 1$

In addition, in some database implementations, we may wish to use the ON DELETE or ON UPDATE clauses to further control the foreign key constraints.

```
        CONSTRAINT FK_Orders_Customers FOREIGN KEY (CustomerID)
            REFERENCES Customers (CustomerID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
```

These two additional clauses are called referential integrity constraints, and they control how related objects are managed. The ON DELETE clause tells SQL Server that if a customer is deleted from the database, the delete should be cascaded to all related orders, so any orders for a deleted customer will be deleted as well. The ON UPDATE clause tells SQL Server that if we change the CustomerID for a customer in the Customers table, that new ID should be propagated to all orders for that customer. These two clauses together ensure that we do not create “orphan” order records when we delete or change a customer.

There are a small number of additional data definition commands. Among them is the DROP command, which deletes a table from the database, and the ALTER command, which can change the structure of a table after it has been defined.

The most commonly-used set of commands in SQL are classified as data manipulation commands, or data manipulation language (DML). These primarily are composed of:

- Select
- Insert
- Delete
- Update

We will study the SELECT command in detail, as it is the most complex of the data manipulation commands and its syntax elements are used in the other commands.

#### **4.1.4 Data Manipulation: SELECT**

The SELECT command is used to retrieve data from one or more tables. There are several forms of this command, ranging from fairly straightforward to very complex.

##### Basic Selects

The most basic form of the select command simply selects an entire table, including all rows and columns. It has the form:

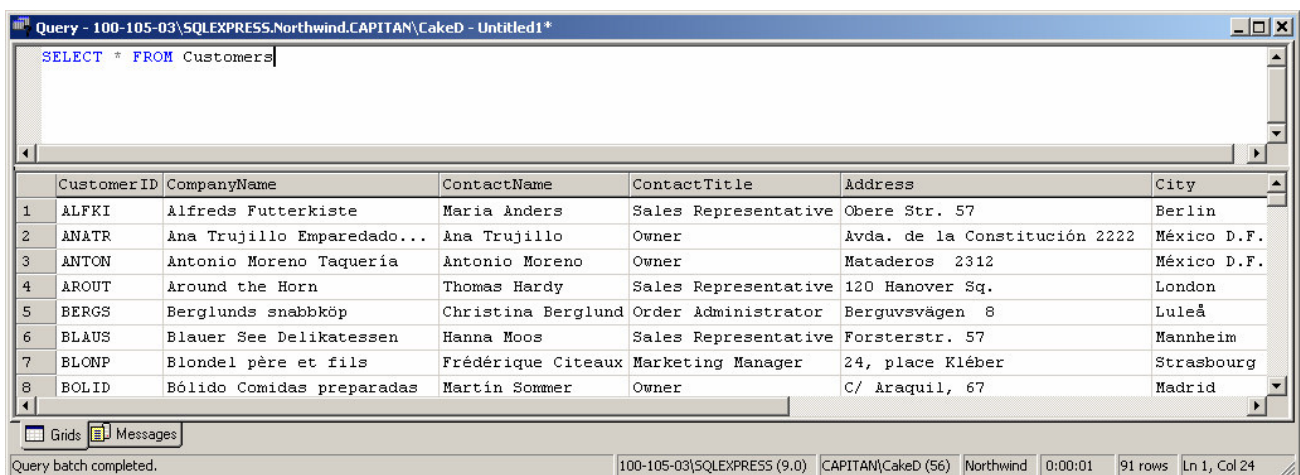
```
SELECT <expression> FROM <table>
```

- <table> is the name of a table in the database, from our case study database, it might be Customers, Products or Orders
- <expression> in the most basic version of the SELECT merely uses "\*" as the expression. This is the wildcard character, and in this context means "everything".

For example, to select all the records from the Customers table, the command would be:

```
SELECT * FROM Customers
```

The results of this query, as seen in the SQL Server Query Analyser application, would appear as shown in Figure 4-3. The result of this query is the entire Customer table, including all rows and columns.



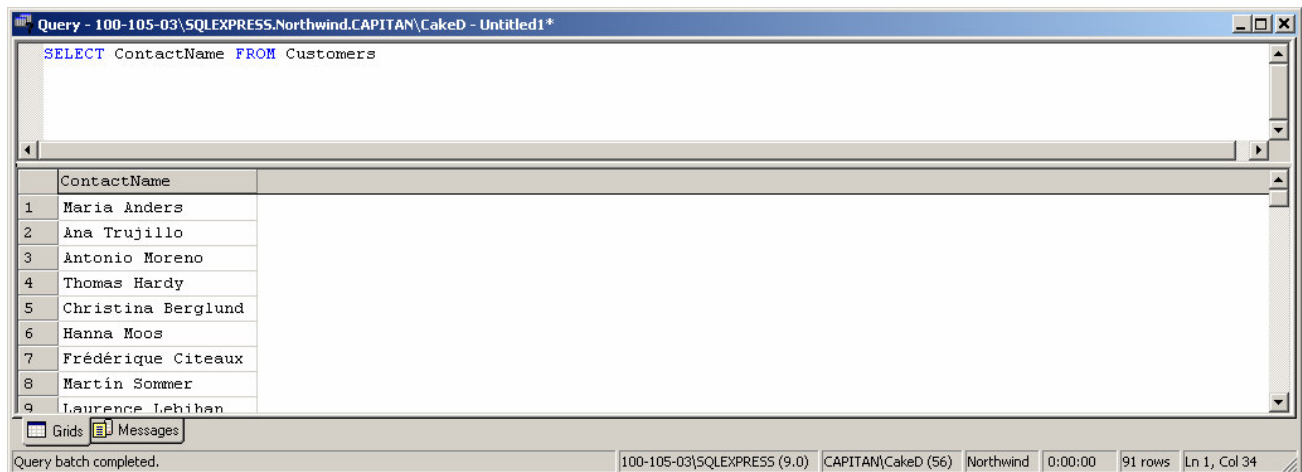
	CustomerID	CompanyName	ContactName	ContactTitle	Address	City
1	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin
2	ANATR	Ana Trujillo Emparedado...	Ana Trujillo	Owner	Avda. de la Constitución 2222	México D.F.
3	ANTON	Antonio Moreno Taqueria	Antonio Moreno	Owner	Mataderos 2312	México D.F.
4	AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London
5	BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	Luleå
6	BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim
7	BLONP	Blondel père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbourg
8	BOLID	Bólido Comidas preparadas	Martin Sommer	Owner	C/ Araquil, 67	Madrid

**Figure 4-3 SELECT Query Results in SQL Server**

We can adapt this basic form to examine many different things. If we change the \* expression to another value, we can examine just the ContactNames from the Customers table, as follows:

```
SELECT ContactName FROM Customers
```

This command still gives us all the rows in the table, but in this case it gives us just the single column ContactName. Figure 4-4 shows the results of this query.



**Figure 4-4 Retrieving a Single Column with the SELECT Command**

SELECT Expressions can give us a variety of things, including all columns, selected columns, the number of rows in the table, arithmetic calculations, and summary values. These expressions, and examples of SELECT commands making use of these expressions, are summarised in Table 4-1, below. All of these expressions return or examine all rows in the table.

**Table 4-1 Expressions in the SELECT Command**

<expression>	Meaning	Example(s)
*	Returns the whole table	Select * from Customers
count(*)	Returns the number of records in a table	Select count(*) from Customers
Fieldname(s)	Returns requested field values	Select Address from Suppliers Select ContactName, Phone from Customers
Calculations	Calculates on numeric fields	Select UnitPrice + 1.14 from Products
Summary Values	Calculates the sum, average, min or max of a given field, for all records in the database	Select avg(UnitPrice) from Products Select sum(UnitPrice) from Products Select max(UnitPrice) from Products Select min(UnitPrice) from Products

## Restricting Rows

So far we have been able to examine specific columns in a table, but the basic select cannot restrict the results to rows of interest. To do this, we introduce a new clause to the SELECT command, the WHERE clause. A SELECT with a WHERE clause uses the following syntax:

```
SELECT <expression> FROM <table>
      WHERE <condition>
```

<table>	is the name of a table in the database, from our case study database, it might be Customers, Products or Orders
<expression>	can be one of many things, as defined in Table 4-1 above
<condition>	some condition which limits the rows examined. Where the condition evaluates to true for a given record, that record is selected.

For example, if we only wanted to retrieve those orders placed on January 1, 1997, we might issue a query such as this:

```
SELECT * FROM Orders WHERE OrderDate = 'Jan 1, 1997'
```

In this case, the \* expression indicates that we would like to see all the columns in the Orders table, but the WHERE clause restricts the rows to be displayed. WHERE clauses can be very complex, involving several criteria and comparing values in many fields.

Note that the date 'Jan 1, 1997' is enclosed in single quotation marks. These are called **delimiters**, and values are delimited differently depending upon the data type of the expression. Numbers are not enclosed in any delimiters, as in the expression `Amount = 10`. Character strings and Dates, as indicated above, are enclosed in single quotes. A character string expression might look like: `ContactName = 'Tom Jones'`. As the delimiters relate once again to data types which may vary between RDBMS applications, this is another point of variability between SQL dialects. In MS Access, for example, strings should be enclosed in double quotes ("Tom Jones"), and dates are enclosed in hash marks (#Jan 1, 1997#).

This particular query has used the equals (=) **comparison operator** to compare the values found in the *OrderDate* field to the date expression 'Jan 1, 1997'. Comparison operators simply allow the comparison of two values. Supported comparison operators are summarised in Table 4-2.

**Table 4-2 Comparison Operators**

Symbol	Meaning
=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<>	Not equal to (some implementations use !=)

These operators make intuitive sense when comparing numeric values, and the equal to and not equal to comparisons perform as one would expect with string values and dates as well. With date values, less than is interpreted as “prior to” and greater than is interpreted as “after”. For example, the following query would find all Orders made prior to January 1, 1997.

```
SELECT * FROM Orders WHERE OrderDate < 'Jan 1, 1997'
```

With strings, less than and greater than may not perform as one would expect. String comparisons using <, >, >= and <= use the American Standard Code for Information Interchange (ASCII) code values for characters in the string to resolve the comparison. Each character has a standard numeric code, and it is these numeric codes which are compared. Generally the ASCII codes are sequential, so that “A” is 65 and “B” is 66, but upper case and lower case letters are coded differently (“a” is 97). The result is that “BBB” comes before “aaa” in ASCII order.

For numeric and date values, there is an additional comparison operator, the BETWEEN operator. This allows queries to search for values between two numeric constants. This is logically the same as if you use the arithmetic comparisons above, but makes a shorter, more readable query. For example, one might wish to find all the products which have a Price between 5 and 10 dollars. Both of the following queries could be used in this case:

```
SELECT * FROM Products WHERE UnitPrice BETWEEN 5 and 10
```

```
SELECT * FROM Products WHERE UnitPrice >=5 AND UnitPrice <=10
```

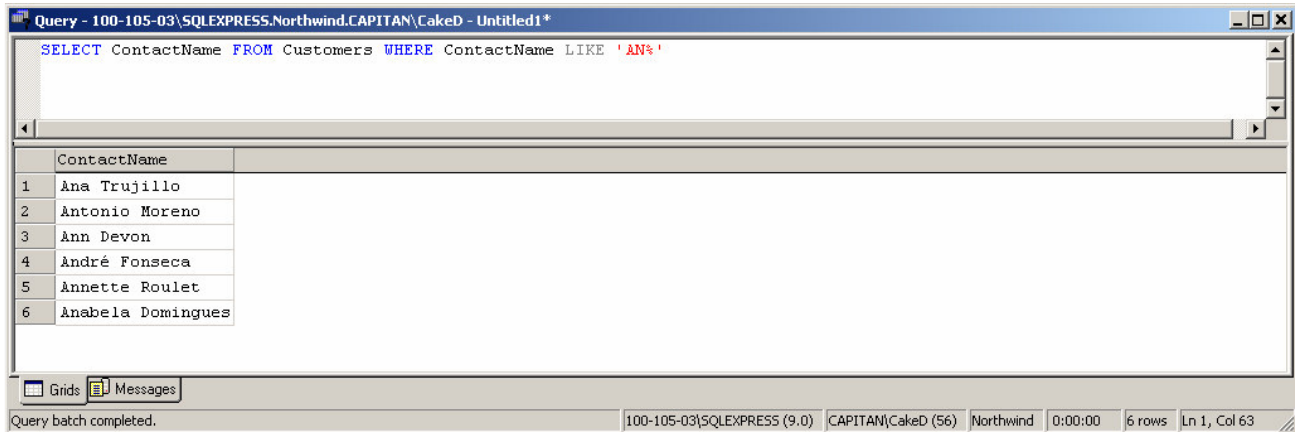
Note in the latter example, that two comparisons are combined using the **logical operator** AND. This means that both of these comparisons must evaluate to True for a record to be retrieved. There are three logical operators: AND, OR and NOT.

For string values, there is an additional comparison operator which allows string comparisons using wildcard characters. The LIKE operator uses a wildcard character “%” to find patterns in strings. This wildcard translates to “zero or more characters”. ‘%Thomas’ would match any number of

characters followed by the string 'Thomas'. It would thus match 'Thomas' or 'Robert Thomas' but not 'Thomas Hardy'. Note that the LIKE operator is not case sensitive, so comparison with '%Thomas%' is the same as comparing with '%thomas%'. The following is an example of a query using the LIKE operator:

```
SELECT * FROM Customers WHERE ContactName LIKE 'AN%'
```

The results of this query are shown in Figure 4-5, below.



	ContactName
1	Ana Trujillo
2	Antonio Moreno
3	Ann Devon
4	André Fonseca
5	Annette Roulet
6	Anabela Domingues

**Figure 4-5 Results of the LIKE Operator**

## Joins

A join is a connection between records of two separate tables. For those students who have been using ArcMap, you may have used joins to connect records in a non-spatial table to records in a feature class based on common field values in the two tables. This is the Primary Key – Foreign Key relationship we discussed in module 2 during the implementation topic.

We will use a join to extract data from more than one table at the same time. For example, in the class diagram in Figure 4-2, there is a connection between Products and Suppliers. This 1:M association is implemented by placing the primary key of the Suppliers table into the Products table as a Foreign Key. To extract information from both of these tables at the same time, we need to ensure that we correctly connect the products with their correct supplier using this foreign key mechanism.

We “connect” these two tables by adding a condition to the WHERE clause which states that the value in the two fields (primary and foreign key fields) are equal. A SELECT command using a join uses the following syntax:

```
SELECT <expression>
      FROM <table1>, <table2>, <table3>
      where <condition>
```

Note that there are now two or more tables as part of the FROM statement. In addition, the WHERE clause must include the condition that the joined fields are equal. For example, if we wanted to extract all products and the supplier for each product, we would issue the command:

```
SELECT * FROM Products, Suppliers
WHERE Suppliers.SupplierID = Products.SupplierID
```

The clause “WHERE Suppliers.SupplierID = Products.SupplierID” ensures that the correct row in the Suppliers table is displayed with the correct row from the Products table. Note that there is a new notation shown here for database fields: Suppliers.SupplierID. This tells the database application that we are looking for the *SupplierID* field in the *Suppliers* table. We need this “dot” notation because the field *SupplierID* is present in both tables. Foreign keys do not need to be named the same as the primary key (in which case the dot notation would be unnecessary because the field names would be unique), but it is common to find databases designed in this way, with both fields identically named.

We can also include other criteria in the WHERE clause to restrict the rows returned and ensure that the join is formed correctly. As before, we will use the logical operators AND and OR to combine WHERE conditions. The following is a fairly complex query, and we will discuss each line in turn. The line numbers are not part of the query, but merely for reference during the discussion which follows.

```
1  SELECT FirstName, Lastname, UnitPrice * Quantity, OrderDate
2      FROM Employees, Orders, [Order Details]
3      WHERE Employees.EmployeeID = Orders.EmployeeID
4            AND Orders.OrderID = [Order Details].OrderID
5            AND UnitPrice * Quantity > 5000
```

Line 1 directs that only a small number of selected columns should be retrieved, in this case 4 columns. This line also shows an example of an arithmetic calculation “UnitPrice \* Quantity”. Here, the \* indicates multiplication, so this calculates the price of a single unit times the number ordered. This expression might be considered the total value of an order. Arithmetic operators available are as follows:

Symbol	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (the remainder of one number divided by another, e.g., 5 % 2 = 1)
^	Exponent (e.g., 2 ^ 3 means 2 <sup>3</sup> )

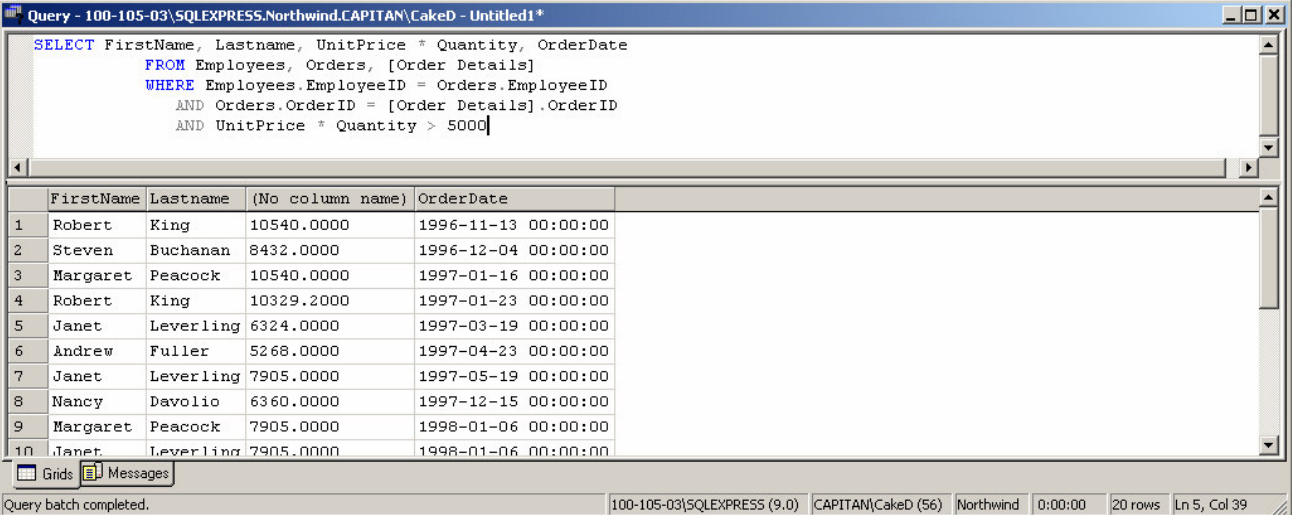
Line 2 of the query indicates that the necessary information will come from three separate tables. As a consequence of including 3 tables, we will need at least two clauses in the WHERE portion of the query, one to ensure that Employee and Order is joined properly, and another to ensure that Orders and [Order Details] are joined correctly. Note also that in this line the table [Order Details] is enclosed in square brackets. This is necessary because the table name has a blank character in

it. The square brackets ensure that the RDBMS interprets both words (“Order” and “Details”) as a single table name, rather than as two words. Users may place square brackets around all table names for completeness, but they are only really necessary where the table name is a reserved word (like “SELECT”) which means something else to the RDBMS, or where the table name has blanks in it. The Northwind database includes this table name with a blank to expose students to the square bracket notation.

Lines 3 and 4 of this query are the restrictions which ensure that the two joins are correctly made. These connect the foreign keys to primary keys, and again make use of “dot” notation because the field names are not unique.

Line 5 is an additional clause which restricts the rows to be examined. Here, we only want to see rows where the total value of the order is more than \$5000.

The result of this entire query is shown below, in Figure 4-6.



Query - 100-105-03\SQLEXPRESS.Northwind\CAPITAN\CakeD - Untitled1\*

```

SELECT FirstName, Lastname, UnitPrice * Quantity, OrderDate
FROM Employees, Orders, [Order Details]
WHERE Employees.EmployeeID = Orders.EmployeeID
AND Orders.OrderID = [Order Details].OrderID
AND UnitPrice * Quantity > 5000

```

	FirstName	Lastname	(No column name)	OrderDate
1	Robert	King	10540.0000	1996-11-13 00:00:00
2	Steven	Buchanan	8432.0000	1996-12-04 00:00:00
3	Margaret	Peacock	10540.0000	1997-01-16 00:00:00
4	Robert	King	10329.2000	1997-01-23 00:00:00
5	Janet	Leverling	6324.0000	1997-03-19 00:00:00
6	Andrew	Fuller	5268.0000	1997-04-23 00:00:00
7	Janet	Leverling	7905.0000	1997-05-19 00:00:00
8	Nancy	Davolio	6360.0000	1997-12-15 00:00:00
9	Margaret	Peacock	7905.0000	1998-01-06 00:00:00
10	Janet	Leverling	7905.0000	1998-01-06 00:00:00

Query batch completed. 100-105-03\SQLEXPRESS (9.0) | CAPITAN\CakeD (56) | Northwind | 0:00:00 | 20 rows | Ln 5, Col 39

**Figure 4-6 Results of the SELECT Query with Two Joins**

There are a number of additional refinements to the SELECT command, such as sorting, aggregating and creating pivot tables. These are increasingly complex, however, and without having practical experience with the more basic SELECT commands, we will leave these refinements for self-study.

#### 4.1.5 Data Manipulation: Insert, Update and Delete

##### Insert

The INSERT command allows users to add data to existing tables in the database. This command takes the following form:

```

INSERT INTO <table>
VALUES (<attribute1>, <attribute2>, ..., <attributeN>)

```

For example,

```
INSERT INTO Shippers
VALUES (4, 'Super Delivery', '(555) 241-2384')
```

Values must be enclosed in parentheses, and must be delimited correctly with quotes for strings and dates, and numbers without delimiters. Each attribute in the row being added is separated with commas.

### Update

```
UPDATE <table>
SET <attribute1> = <value>,
    <attribute2> = <value>, etc.
[WHERE <where clause>]
```

For example,

```
UPDATE Customers
SET Phone = '(250) 555-1234',
    ContactName = 'Wayne Gretzky'
WHERE CustomerID = 12
```

This query simply updates the contact name and phone number for one record in the *Customers* table. If more than one attribute is being updated, the update clauses are separated by commas. Note that the WHERE clause is optional in the command syntax. If this is omitted, the update is performed on every record in the table.

### Delete

```
DELETE FROM <table>
WHERE <condition>
```

For example,

```
DELETE FROM Orders
WHERE OrderDate < 'Jan 1, 1980'
```

This query simply deletes the old order records, where they were placed prior to January 1, 1980.

#### **4.1.6 Spatial Extensions to SQL**

While the commands available in SQL are powerful, they are limited in that they only support simple data types such as numbers, strings and dates. Spatial databases must have the ability to retrieve information using more complex data types such as points, lines and polygons and more complex interrelationships such as adjacency and proximity.

Recognizing the pervasiveness of spatial data and GIS applications, the international standards for SQL and the major RDBMS producers have implemented extensions to standard SQL to enable management and query of spatial data. Both the OGC and ISO have standardized the use of spatial data in an RDBMS. The OGC reference is *OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option*. The ISO reference is

*ISO/IEC 13249-3 SQL multimedia and application packages - Part 3: Spatial.* The ISO SQL multimedia standard is commonly referred to as SQL/MM. RDBMS spatial implementations include Oracle Spatial, Informix Spatial Datblade and DB2 Spatial Extender. SQL Server is scheduled to include full spatial support with their 2008 release.

Both the georelational and the object-relational structures have limitations relating to the use of an underlying RDBMS. The Georelational structure, you may recall, stores the feature attributes in a relational table and the geometry in a separate binary file. Most object-relational implementations store all GIS data in a single relational database, but they store the geometry in BLOB (binary large object) data types in an encoded form. Both of these implementations, because they rely on different, proprietary ways of representing spatial data, rely heavily on the GIS application software to manage and manipulate the data, since the RDBMS itself has no knowledge of the semantics nor the encoding of the spatial objects.

Standardizing the way spatial data is stored is the foundation of establishing spatial standards for databases and SQL. Doing so eliminates the need for handling spatial relationships at the application level. Many geographic processing functions and much of the data integrity control can be moved to the RDBMS itself, greatly reducing the requirements of the GIS software. In addition, a standardized storage method allows different GIS applications to access the same data without the need for translation routines.

As with basic SQL, there are dialects with Spatial SQL, because implementations vary slightly between the major RDBMS vendors. The following examples use a generic dialect based on the 1999 OGIS specification and used by several references (e.g., Shekhar, 2003). Users may refer to documentation regarding their RDBMS to determine the precise dialect subtleties.

The spatial DDL is very similar to standard SQL when using spatial types:

```
CREATE TABLE Cities (  
    Name          varchar(30),  
    Population    int,  
    Shape         POINT  
)  
  
INSERT INTO Cities VALUES (  
    'Some Little Town',  
    10784,  
    NEW POINT (493940.1, 5363403.2)  
)
```

Note the use of the spatial data type POINT. Other data types exist for linear and polygonal features as well. These spatial data types must be added to the data definition language portions of SQL, and the spatial calculations and relationships must be included in the data manipulation commands.

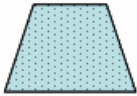
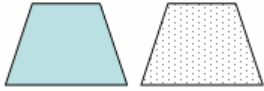
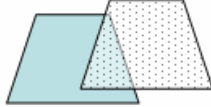
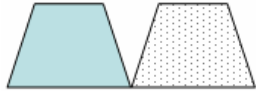

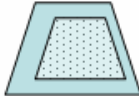
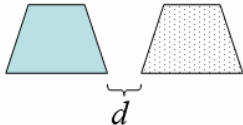
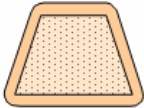
For example, where standard SQL includes operators for arithmetic operations such as addition, subtraction, multiplication and division, SQL/MM must include operators to accommodate spatial

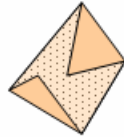
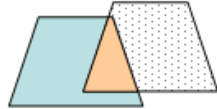

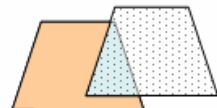
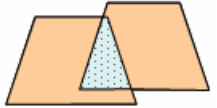
calculations. New operators are added to SQL to allow such calculations as boundary or line length, interior area and linear direction. For example, the following query uses an AREA operator in conjunction with a spatial field *Shape* to determine the area of each country.

```
SELECT Name, AREA (Shape)
FROM Country
```

Where standard SQL allows comparison of values using comparison operators such as greater than, equal to, or not equal to, SQL/MM must include a number of new comparison operators to accommodate spatial comparisons. New operators will include touch, cross, overlap, inside, covers, covered by, equal and disjoint. Table 4-3 summarizes these operators.

**Table 4-3 Topological Operators for Spatial SQL**

Operator Type	Operator	Description	Example
Topological Operators	Equal	Returns true if the two geometries are the same	
	Disjoint	Returns true if there is no intersection between the two geometries (i.e., the geometries are separated in space)	
	Overlap	Returns true if the interiors of the two geometry intersect	
	Touch	Returns true if the boundaries of the two geometries intersect, but the interiors do not	
	Cross	Returns true if the interior of a surface intersects with a curve	
	Contains	Returns true if an entire geometry lies within the interior of another geometry.	
	Intersect	Returns true if the two geometries are not disjoint	Either an overlap or a touch.
Spatial Analysis	Distance	Returns the shortest distance between two geometries	
	Buffer	Returns a geometry that consists of the area around the geometry of a given distance	

	ConvexHull	Returns the smallest convex geometric set enclosing the geometry	
	Intersection	Returns the intersection of the two geometries	
	Union	Returns the union of the two geometries	
	Difference	Returns the portion of one geometry that does not intersect with the second geometry	
	SymDiff	Returns the portions of both geometries which do not intersect	

For example, consider three feature classes, City, River and Country. We might issue a variety of queries to examine the spatial relationships between features.

For all the rivers in the *River* table, find the countries they pass through:

```
SELECT Country.Name, River.Name
FROM Country, River
WHERE CROSS (River.Shape, Country.Shape)
```

Find all the cities which lie within 300m of the Rhine river:

```
SELECT City.Name
FROM City, River
WHERE OVERLAP (City.Shape, BUFFER (River.Shape, 300))
AND River.Name = 'Rhine'
```

Find the length of the rivers within each country they pass through:

```
SELECT      River.Name,
            Country.Name,
            LENGTH ( INTERSECTION (River.Shape, Country.Shape))
FROM River, Country
WHERE CROSS (River.Shape, Country.Shape)
```

Extensions to SQL for spatial data are fairly new (adopted for the 1999 SQL standard). Changes are happening very quickly, witnessed by the fact that SQL Server is just now beginning support for spatial types and spatial functionality. It is expected that although there are currently limitations

with spatial SQL, such as lack of support for such objects as rasters, 3-dimensional terrain surfaces, metadata and networks, support for spatial data and spatial data processing will become a larger part of mainstream RDBMS technology.

With some supported RDBMS, ArcGIS users may choose between BLOB or standardised geometry data types for representation of spatial features. Using the standardized spatial data types will mean that custom applications not using the ArcObjects framework may still work with the underlying database, and third-party applications may access the data. In all ArcMap versions prior to 9.2, users were forced to use BLOB fields to encode binary representations of feature geometry. This is a significant move toward a more open GIS architecture, and an increased role for the underlying DBMS in the functionality of the GIS application.

The GIS application software is still relied upon significantly to do much of the work with a geodatabase. If SQL is used to manipulate geodatabase records directly, care should be taken, as data may become unusable by the GIS application. ESRI suggests that users of SQL adhere to the following guidelines when modifying information in a geodatabase:

- Avoid making changes to feature classes which have been versioned (SDE geodatabases only).
- Issue a commit or rollback statement after any SQL modifications, since failing to do so can cause problems. For example, a database compress operation may wait indefinitely for the transaction to be either committed or rolled back.
- Avoid modifying any attributes that, through geodatabase behavior, affect other objects in the database (e.g., feature-linked annotation or relationship classes).
- Avoid using SQL to modify the geometries of feature class that participate in any advanced geodatabase objects or functionality such as geometric networks, topologies, and relationships.

## 4.2 Spatial Indexing

### 4.2.1 Introduction

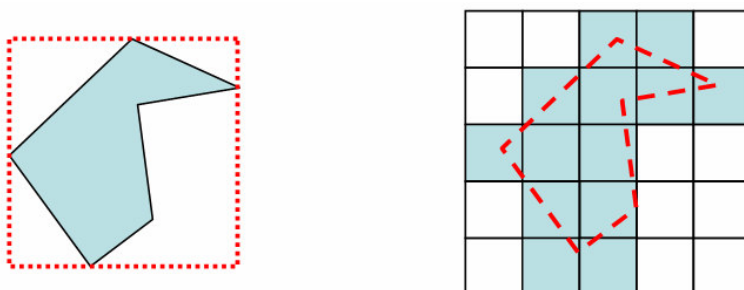
An **index** is a data structure that allows for improved access of data objects. It functions in the same way as an index in a book. If you were looking for all relevant discussion of SQL in a database textbook, for example, you would not read the entire book scanning for the text “SQL”. You would look up SQL in the index, find the relevant page(s), and then go directly to those pages. Indexes in a non-spatial database can provide extremely fast response times even when dealing with very large data volumes.

A book index, or an index to help find bank account numbers in a banking database are both examples of 1-dimensional indexes. They are one dimensional because they represent values on a single axis, such as words sorted from A to Z, or bank accounts sorted in numeric order. The index takes a value along that single-axis continuum and finds it quickly in the database. There are many techniques to help speed up 1-dimensional searches.

Spatial data must use 2 or 3-dimensional indexes, since locations are necessarily represented with x, y coordinates, or even x, y, z coordinates. As a result, we must use different techniques either to represent 2-dimensional space in a linear fashion in conjunction with existing 1-dimensional techniques, or derive new methods for indexing 2 or 3-dimensional space. We will explore the latter situation in this topic, since these new methods are used in the RDBMS and GIS applications we have been discussing previously.

A spatial index works in much the same way as a 1-dimensional index, in that it stores information about the spatial extent of features to more quickly find and use spatial objects. The underlying idea for spatial indexing is to store an approximation of an object's geometry. This simplified geometry serves as a spatial key for retrieval of the true feature geometry.

One type of approximation is called a **continuous approximation**. Such approximations are based on the coordinates of the objects themselves. The most common of these is the bounding box, or minimum bounding rectangle (MBR), which is the smallest rectangle, parallel to the coordinate system axes, fully enclosing the true feature geometry. The other type of approximation is the **grid approximation**. In this case, space is divided into cells by a regular grid, and the geometry of the feature is represented by the set of cells which the feature intersects. Figure 4-7 shows examples of both kinds of approximation.



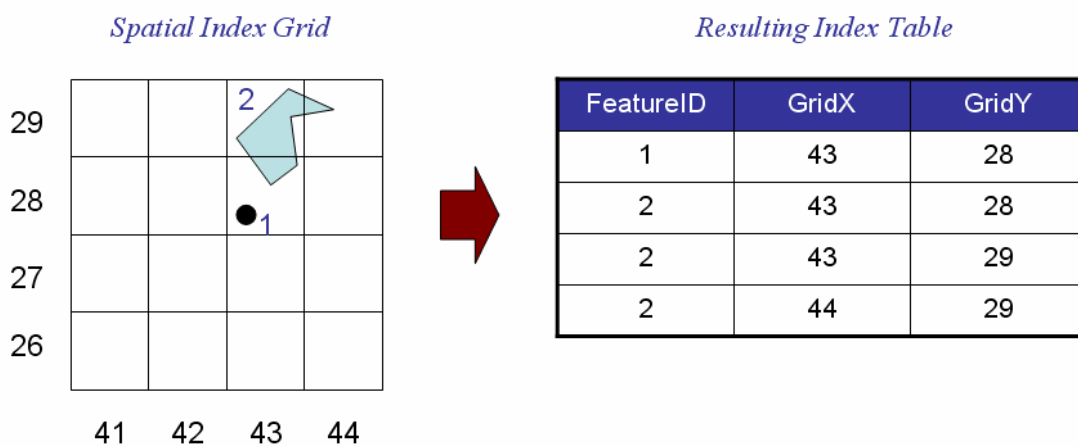
**Figure 4-7 MBR and Grid Approximations of the Same Feature Geometry**

Using approximations as a spatial key leads to query processing which uses a two-step process for finding features by their locations. The first step seeks to reduce the number of candidate features which must be examined in detail, and is sometimes called the **filter** stage. Hopefully, the filter step vastly reduces the number of features which might be of interest by examining only the approximated geometry. The remaining possible candidates which pass the filter stage are examined in detail during the **refinement** step of query processing, during which the precise coordinates of the true geometry are checked. In this manner the vast majority of features in the database are examined using only their generalized geometry, which will in most cases improve performance significantly. We will explore two methods of spatial indexing, grid indexes and R-tree indexes, both of which take advantage of these approximation techniques.

#### 4.2.2 The Grid Index

A **grid index** can be simplistically considered the same as a road map index. Rather than scanning the entire city map looking for 1<sup>st</sup> Avenue, a reader might look up 1<sup>st</sup> Avenue in the index. The entry there might tell the user that 1<sup>st</sup> Avenue lies in or near the grid coordinates A3. The reader then scans the row and column labels, perhaps rows are labelled A, B, C, and so on, and the columns are labelled 1, 2, 3, etc. Where the row A intersects column 3, there should be a grid cell in which the reader will find 1<sup>st</sup> Avenue.

Functionally, this is how a grid index works in a GIS application, and it is probably the simplest method of quickly filtering candidate features. To use this technique, the application creates a grid, where cells are referenced by row and column indices, as shown in the image on the left of Figure 4-8. Then, a table is created which summarizes the interaction of features in the database with this grid. This table is shown at the right of Figure 4-8. For example, the point labelled “1” in the spatial representation lies entirely within the grid cell referenced by 43, 28. The entry in the index table 1, 43, 28 indicates that feature 1 lies in the cell 43, 28. The polygon labelled “2” lies within 3 cells, so three rows are created in the index table to indicate that feature 2 intersects 3 grid cells.



**Figure 4-8 Simple Grid Index Mechanism**

For the sake of discussion, let us imagine the user clicks a point location in a map window, and wishes to determine which feature they have clicked on. The set of steps used by the GIS application might be something like the following:

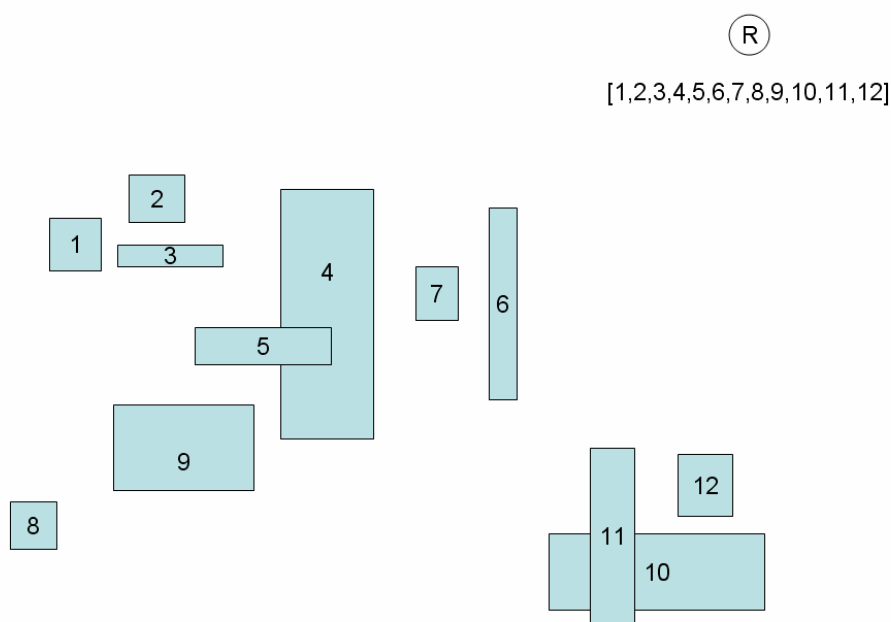
1. Determine which grid cell the x, y coordinates clicked by the user fall within. Let us assume that the user clicks in cell 43, 28.

2. Look within the index table to find all entries for grid cell 43, 28. This table tells us that features 1 and 2 intersect grid cell 43, 28. This is the filter step, since we have quickly determined that of all the features in the database, there are two candidate features which the user *might* have clicked on.
3. Finally, the application must use a refinement step to look at the true geometry of features 1 and 2, and the actual x, y location clicked to determine which feature(s) the user actually clicked on.

The performance gain in this process lies in the fact that the GIS application firstly does not look at all of the features in the database to determine which feature was selected, and secondly that the actual geometry of the features is not examined for large numbers of features. The efficiency of the grid is dictated by the size of the grid cells, which is a trade-off between the efficiency of the filter and the size of the index table. Large grid cells mean that many features might lie within a given cell, so the filter stage might still leave a large number of candidate features to pass on to the refinement stage. A large grid cell size also means that the index table is small, so it is very fast to read from. Conversely, a small grid cell size means that the filter stage is extremely efficient, and can reduce the number of candidate objects dramatically before beginning the refinement stage. However, the index file may be very large, and retrievals from this table might be relatively slow.

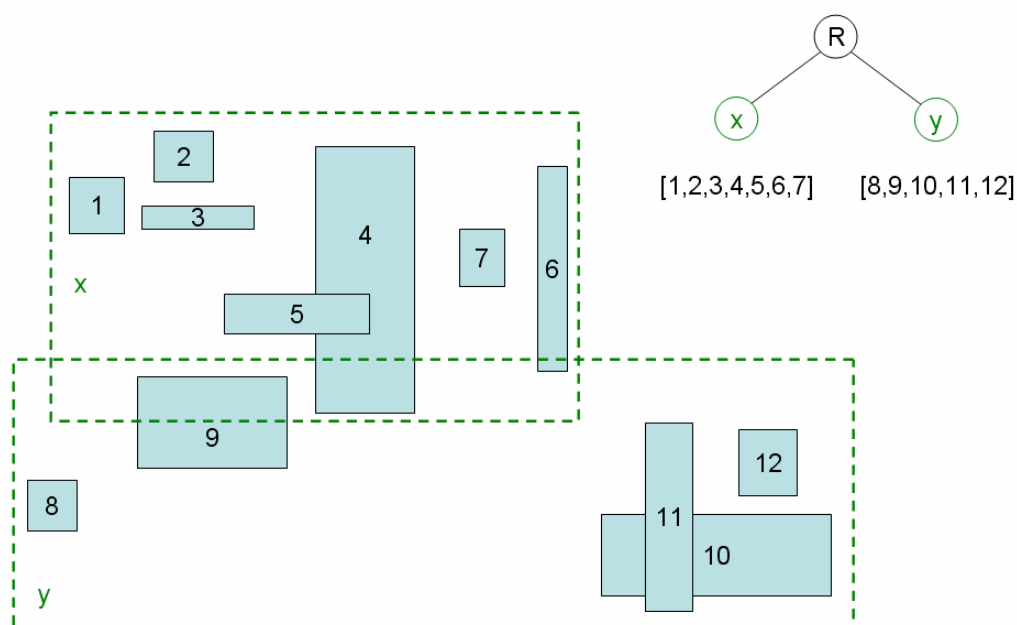
#### 4.2.3 The R-Tree Index

An **R-tree index** makes use of nested layers of minimum bounding rectangles. The basis is to store an MBR for every feature in the database. Polygon and line features are represented by a rectangle, while the MBR for a point is simply the point. We will use examples with rectangular MBR's, but the same R-tree process also works with points. Figure 4-9 shows a series of numbered rectangles, each representing an MBR of a feature in the database. The "R" and numbers at the top right shows the R-tree representation for this level of decomposition. Here, this might be the equivalent of a very large grid cell in the grid indexing method, and all MBR's currently are referenced by the **root** of the R-tree, so no spatial search efficiency is realized. The root is the starting point for the tree structure. The notation [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] refers to the fact that all feature rectangles are contained in, or referenced by, the undivided root coordinate space.



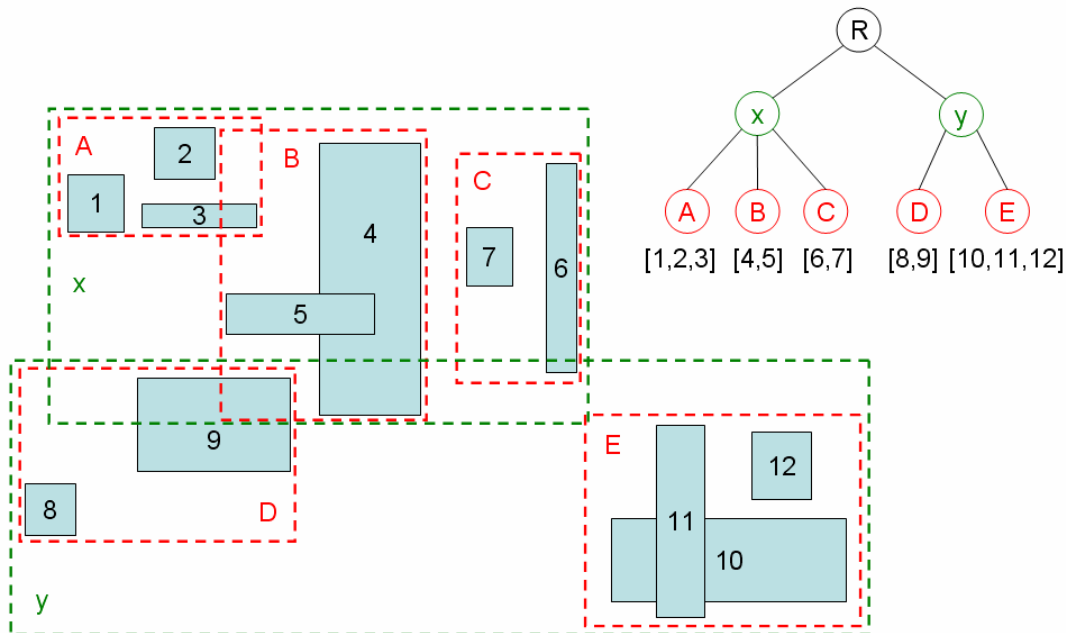
**Figure 4-9 Feature MBR's to be Represented using R-tree Indexing**

We then decompose 2-dimensional space into another set of MBR's. In this case, the new MBR's are the minimum rectangle which can encompass a subset of the feature MBR's. In Figure 4-10, for example, all feature MBR's are grouped into two new MBR's which enclose the feature MBR's. This functionally decomposes the 2-dimensional space of the database into two MBR's, labelled x and y.



**Figure 4-10 Partial R-tree Decomposition**

The tree figure at the top right now shows that the root, or entire database coordinate space is divided into two MBR's. MBR y, for example, now contains 5 features (8 through 12). The MBR's x and y can be further subdivided into smaller MBR's, as shown in Figure 4-11.



**Figure 4-11 Full R-tree Decomposition**

Here, for example, MBR Y is decomposed into two smaller rectangles, labelled D and E. In this way, the space occupied by each rectangle is decomposed into smaller rectangles which encompass the feature MBR's which lie within it. The tree structure at the top right now has three levels, indicating the decomposition of space.

To find a given feature, it is a fairly simple matter to traverse the tree structure to find the relevant feature(s). For example, suppose the user clicked on the map at a location that lay within the feature MBR labelled "2", near the top left. The search process begins at the root of the R-tree structure. It then compares the search x, y with the two child rectangles of the root, labelled x and y. The point clicked lies within x, but not within y, so the entire subtree beneath y may be disregarded for this search. We then compare the clicked point with the child rectangles of x, rectangles A, B and C. We find that the clicked point lies only within MBR A. We can then discard the B and C MBR's. Rectangle A tells us that three feature rectangles lie within it, labelled 1, 2, and 3. We can then compare these feature MBR's to the clicked point. Doing so tells us that one feature, feature 2, might have been clicked on. To know whether the clicked point touches the true geometry of feature 2, the detailed comparison must be made as part of the refinement process. The MBR for feature 2 will also store a reference, or key, to find the true feature in the database so that the detailed geometry comparison may be performed.

The performance gain from this type of indexing lies in that large portions of the database can be removed from the list of potential candidates (the filter process), because only the relevant subtrees of the R-tree structure are traversed during the search. Additionally, a very small number of features must have their true geometry examined; in our example, only one feature. It is

possible that in the case of overlapping MBR's (e.g., a portion of area is common to both x and y), multiple subtrees must be traversed.

The R-tree structure must be a height-balanced structure, so the feature MBR's are always the same depth, or number of levels removed from the root. All feature MBR's appear at the same level of the tree. R-trees are typically quite shallow and wide. For example, an R-tree indexing 100 million features may have a depth of 5, and over 100 children for each non-leaf node (Shekhar and Chawla, 2003. p 101). A non-leaf node is one at an interior level of the tree, such as x and y.

Performance in the R-tree depends upon two factors:

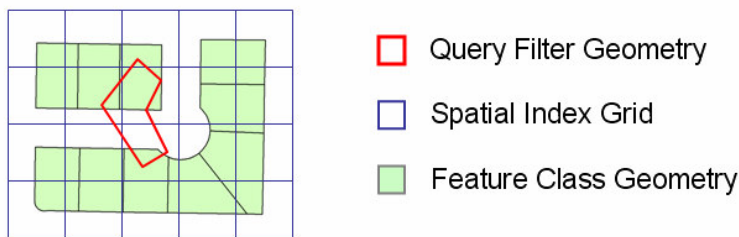
- |          |  |
|----------|--|
| Coverage | Coverage is the proportion of the area at a given level of the tree which is included by the child MBR's. This is an indirect measure of "dead space" at each level, or the portion of the parent that is not covered by a child. For example, the child rectangles D and E actually cover only a small portion of the area covered by the parent rectangle y. For this subtree, there is fairly low coverage. The coverage for rectangle x, by comparison, is fairly high since the proportion of x covered by A, B and C is fairly high. For higher performance from the R-tree structure, we wish to minimize coverage. Low coverage means that large areas of space can be discarded as each level of the tree is traversed. |
| Overlap  | Overlap is a measure of how much rectangles at a given level overlap each other. For example, rectangles x and y overlap for perhaps 10% of the area covered by the two rectangles. Rectangles A through E likely overlap each other even less than 10%. From a performance perspective, we wish to minimize overlap. Where two rectangles overlap, there is the possibility that both subtrees must be traversed in a search.   |

Overlap minimization is more critical than coverage to R-tree performance. As a result, there have been many attempts to refine the R-tree structure to reduce overlap. Examples include the R\*-tree, R+-tree and packed R-tree structures.

#### 4.2.4 The ESRI Spatial Index

Spatial indexes in ArcMap make use of a different mechanism depending on the data source. Personal geodatabases make use of a single grid index, while file geodatabases and ArcSDE geodatabases in Oracle, SQL Server, and DB2 use a system of up to three grids as the spatial index. Spatial data stored in Oracle Spatial and Informix ArcSDE geodatabases use an R-tree index which is managed directly by the RDBMS. The effectiveness of the grid index is dictated by the grid size used. This size may be defined by the user, so in the interest of more effectively managing spatial indexes in ArcMap, we will briefly define how the ArcMap grid index functions.

For the purposes of discussion, we will assume the user is trying to determine which land parcels are intersected by an arbitrary polygonal feature. We will call the polygons being searched (the land parcels) the *feature class geometry* and the polygon we are using to query with the *filter geometry*. Such a situation might resemble that illustrated in Figure 4-12 Spatial Query Example.

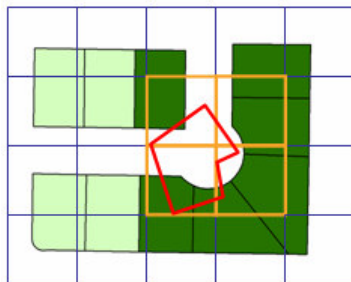


**Figure 4-12 Spatial Query Example**

The ESRI index functions in a four-stage filter and refinement process, as defined below.

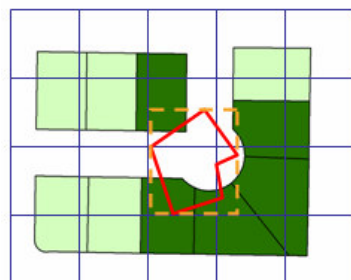
1. **Grid tile on grid tile filtering.** This step finds all feature class records where the grid cells intersected by the filter geometry match the grid cells which intersected by feature class geometries. Figure 4-13 shows in orange the grid cells intersected by the filter geometry. These cells then intersect a number of parcels. The candidate parcels are shown in dark green.

This first step in the filter process should result in the greatest reduction of candidate features. This example obviously does not illustrate the vast majority of the features not being selected, since we have purposely zoomed in on a small area of interest. The remaining candidate features are passed on to step 2.



**Figure 4-13 Remaining Candidates after Step 1**

2. **MBR on MBR filtering.** The second step takes the MBR (or “envelope” in ESRI documentation) of the filter geometry and compares it to the MBR of all candidate features. Those whose MBR’s intersect the filter geometry MBR remain candidates and are passed on to step 3. Figure 4-14 shows the MBR of the filter geometry in relation to the land parcels, and the resulting remaining candidate features. This step makes a small reduction in the number of candidates.



**Figure 4-14 Remaining Candidates after Step 2**

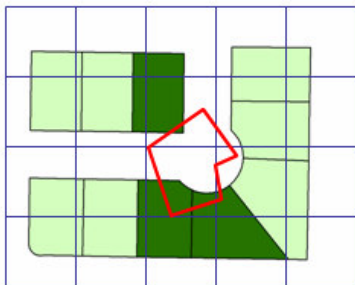
Both of the preceding steps are done at the RDBMS level, so they are performed very quickly. The resulting candidates are passed on to SDE to complete the query processing.

3. **Filter geometry on feature class envelope filtering.** The final filter, prior to full comparison with the feature class geometry, compares the geometry of the spatial filter to the MBR's of the remaining candidates. Figure 4-15 shows the remaining candidates after step 3 completes. The orange MBR on the one parcel which has a concave surface shows why it remains a candidate at this stage.



**Figure 4-15 Remaining Candidates after Step 3**

4. **Filter geometry on feature class geometry refinement.** This final stage is the most intensive calculation. It compares the filter geometry to the full geometry of each of the remaining candidates. Figure 4-16 shows the final result of the intersect operation, after all four processing steps have completed.



**Figure 4-16 Final Intersect Result**

## 4.3 Temporal Data Storage

### 4.3.1 Introduction

GIS has developed into an efficient tool for managing and analysing space. However, there has been very limited progress in the ability to examine changes over time to spatial and non-spatial objects. Virtually anything which can appear on a map is subject to change over time, particularly entities such as political boundaries, vegetation units and land ownership. The element of time is even more important in those disciplines where changes through time are not simply a representational challenge to managing data (such as with land ownership), but are the fundamental focus of study. Such areas of study include criminology, disease epidemiology and hydrological dynamics.

GIS technology has its roots in cartography and the automation of static map production. However, the expansion of GIS into many different application areas, growth in the availability of spatial data and the increased storage and analysis capacity of computer technology have all contributed to an increased focus on time as a necessary dimension to geographic analysis. Geographers have long considered that geographic information is based on the three major components of location, attribute and time, but it was not until the 1990's that research into managing the element of time in GIS began in earnest.

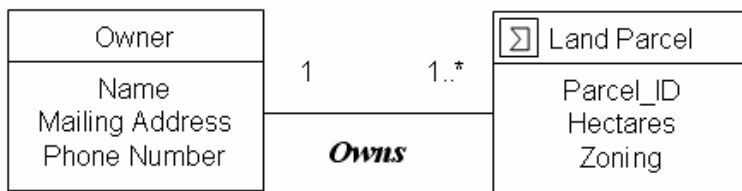
A number of terms can be used to describe the nature of change, and it is these characteristics of change that we wish to capture and describe with a GIS. The manner of a change may be described as:

Continuous	Change occurring at a fairly constant rate throughout some interval of time (e.g., geological or hydrological processes)
Majorative	Change occurring through most of some interval of time, including periods with no change
Sporadic	Change occurring intermittently over some interval of time, while remaining static much of the time.
Unique	Change occurring only once.

### 4.3.2 Time in a Database

Non-spatial databases can deal with time dependence of records by adding attributes which define the time period for which a given record is valid. Many temporal phenomena may be captured with this simple structure, by adding two attributes which store *FromDate* and *ToDate*. These two attributes indicate the time period during which the object or database row is valid. This approach is particularly effective for phenomena which are short-lived with very specific creation and destruction dates. An example might be an Employee table, where we store the date a person was hired, and the date they ceased to be employed. By querying using these two fields, we can gain an understanding of who worked for us at any given point in time.

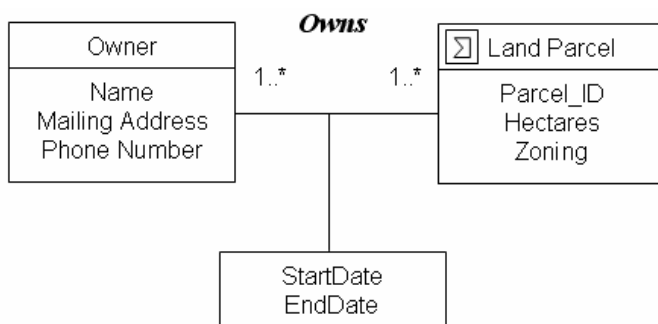
We can extend this concept further to place temporal fields on associations within our database as well. Consider the Owner-Parcel class diagram we have discussed at several points in this course. This diagram is presented once again in Figure 4-17.



**Figure 4-17 Owner-Parcel Class Diagram – no Temporal Element**

This diagram models a static view of the association. Here, a given parcel of land may have one and only one owner, so we cannot model a history of ownership - ownership through time. This model is a temporally static model, which can represent only a single point in time.

If, on the other hand, we alter the model to allow many owners, we change the 1:M association into a M:M association. In addition, we turn the simple M:M association into an *association class*, so that we can then store a start and end time with the association itself. In this way we can store several owners for a given parcel, and the time period during which they owned the land. Figure 4-18 shows such a model.



**Figure 4-18 Owner-Parcel with Ownership History**

Another method of tracking change in a non-spatial database is the use of a transaction log. This might take the form of a single table which summarizes each change as it is made, and records the time when the change occurred. A transaction table might include the following information:

```

CREATE TABLE TransactLog (
    TransID          Number (10)      NOT NULL,
    TableName        Varchar (30)     NOT NULL,
    RowID            VarChar (200)    NOT NULL,
    Description       Varchar (2000)  NOT NULL,
    TransDate        Date              NOT NULL
)
  
```

This table structure allows the extraction of every change made to the database, so it can be very helpful in describing the types of changes made and the precise timing of events. It does not, however, show a complete picture of what the database looked like at any given time. For example, if a database record is deleted, we simply know when the deletion took place, and which record was deleted. We do not know what information was stored in the deleted record, so this method can be limited. The transaction log approach is very effective, however, for databases with a small number of simple transaction types. For example, the balance of a bank account might be

affected by three transaction types: transfer, deposit and withdrawal. Using a simple transaction log to store a history of these simple transactions, one could reconstruct the account balance at any point in time.

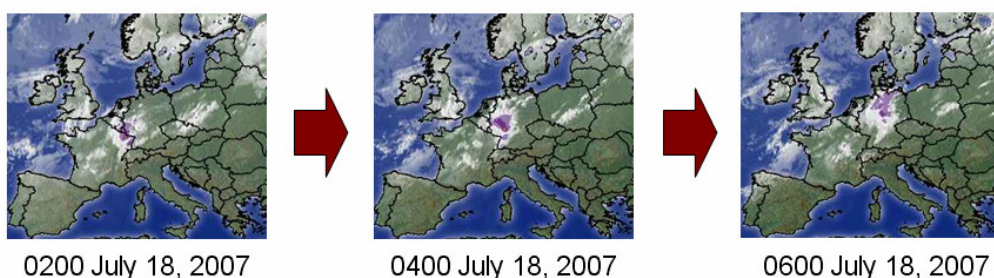
Geographical cases are far more complex; objects may have limited existence (which lends itself well to adding *FromDate* and *ToDate* attributes), but may also move, change shape, and change attributes as well. For example, land parcels may be aggregated or subdivided over time, but they could also simply change ownership. The challenges for representing time in a GIS involve:

- Methods for data storage, or data structures which can accommodate data which change through time.
- Methods for querying temporal information. Not only must the query mechanisms be efficient, but they need to accommodate questions such as “where has change occurred?”, “what direction is this phenomenon moving?”, and “at what rate is this change occurring?” – questions which are difficult to accommodate using current GIS techniques
- Methods for effective and efficient display of temporal data.
- Methods for interpolating or generalizing time. Although methods exist for storing and examining several discrete views of data through time, the means to effectively examine data between discrete time views does not exist.

Many data models have been proposed for recording changes in spatial objects. We will explore some of these options as a means of illustrating potential solutions to temporal data management.

#### 4.3.3 The Snapshot Model

The most common approach to incorporating time is often called the **snapshot model**. The snapshot model stores information in a series of map layers depicting the same phenomenon through time. Each snapshot, or “time slice”, represents the entire spatial extent of interest and simply depicts a given state of the geographic data for a given point in time. This method derives its name from the idea of taking a photograph, or snapshot, of something at a series of points in time. Figure 4-19, below, shows an example of snapshots of weather patterns changing through time.



**Figure 4-19 Snapshot Approach to Modeling Time**

This method lends itself particularly well to raster datasets, and indeed this representation is inherent in remote sensing, where images might be acquired at intervals each time a satellite passes over a geographic area. It can also be used with vector representations, and in either case (raster or vector) this method is conceptually simple and easy to implement.

There are several major shortcomings of the snapshot model. Firstly, since each snapshot represents the entire dataset as a standalone copy, there is significant redundancy in this model, particularly where limited change has occurred. In addition, there is no logical connection between the features stored in different snapshots. For example, it is difficult to answer questions of a landuse dataset such as “which areas have been actively cultivated at least 5 of the past 10 years?” Thirdly, the nature of a snapshot model (storing discrete observations at intervals through time), means that what happens *between* the snapshots is not recorded. We cannot know the precise timing of events, only the period during which the change occurs.

The length of time between snapshots determines how well we understand the nature of the change. As mentioned previously, we cannot know precisely when, between snapshots, something happens. We can only see that something was there in one snapshot, for example, and not there in a subsequent snapshot. The closer together, in time, the snapshots are taken, the more precisely we can state when the change occurred. We call this **time granularity** (sometimes temporal granularity), or the smallest interval of time which can be represented in a temporal GIS. The time granularity of a database depends on the needs of the specific GIS application. Some applications, such as traffic monitoring or weather observation, may seek extremely fine granularity of minutes or seconds, while community planning applications may require only months or years between observations.

Snapshots may be managed in ArcGIS using SDE versions. In this way, we can keep a series of snapshots of our data through time, and may also maintain snapshots in a non-linear fashion. For example, we could explore several planning scenarios which represent alternative branches of time, rather than snapshots along a linear temporal path. As we have discussed, versioning does not have the redundancy of a full copy of the data for each version, so the storage costs of this method may be lower. However, maintaining many complex versions of a feature class can produce a large overhead in the delta tables and can result in significant performance reductions. Versioning is likely not a good solution for an organisation wishing to maintain a long-term history of a volatile feature class, but can be effective for maintaining a small number of snapshots, perhaps only required for a short-term planning process.

#### 4.3.4 The Region-Entity Model

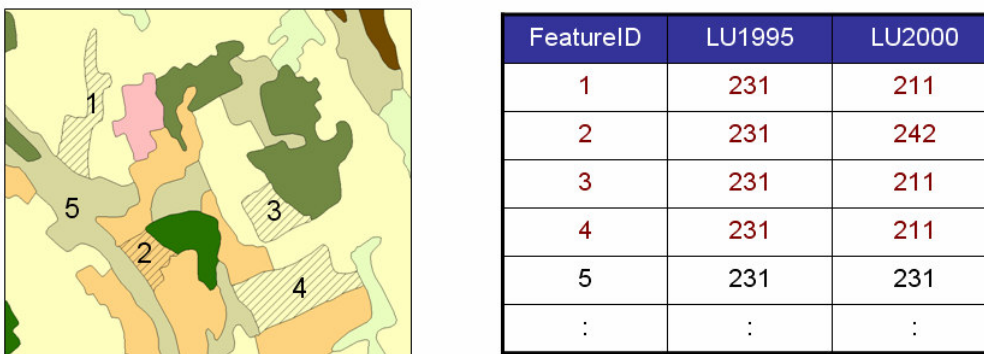
Another approach to incorporating time in GIS is to map every line that ever existed and use some mechanism, such as attributes, to indicate the boundaries of a given feature or attribute value at a given time. This approach is fairly difficult to describe in text, but is quite clear when one sees the results, so we will use several examples to illustrate how this model represents time.

To illustrate the simplest implementation of this model, consider a small portion of the Lithuanian CORINE land use dataset. This dataset maps land use at 1:10,000 for two time periods, 1995 and 2000. Figure 4-20 shows a small area of this dataset, using a snapshot approach to indicate change in the period 1995 - 2000. The areas numbered in both images are areas where the land use has changed between 1995 and 2000. All numbered areas were coded as land use 231 (Abandoned Pasture, the light brown colour) in 1995. In 2000, areas 1, 3 and 4 below were changed to class 211 (Non-Irrigated Arable, pale yellow) and area 2 changed to class 242 (Complex Cultivation, pale orange).



**Figure 4-20 CORINE Land Use as Snapshots**

The common snapshot approach to handling this change would be to maintain two land use datasets: one for 1995 and one for 2000. Both would be complete land use datasets and could function as standalone data. An alternative using the region-entity model is to represent both temporal views in a single spatial dataset, and use attributes to manage which land use was valid for which snapshot. Figure 4-21 shows one way of accomplishing this.



**Figure 4-21 Region-Entity Representation of CORINE Land Use Change**

The first requirement for this representation, is that the 1995 and 2000 land use polygons must be merged into a single set of polygons, functionally intersecting them. The map on the left above shows the merged polygons, with those areas with changed land use highlighted using a cross-hatched pattern. The polygon colours are drawn using 2000 colours, so students can see what land use the Abandoned Pasture became in 2000. The table on the right shows what the polygon attributes would look like, including both 1995 and 2000 land use attributes (stored in the fields *LU1995* and *LU2000*, respectively).

Polygons which do not change classification between 1995 and 2000 appear as polygon 5 in Figure 4-21, with the 1995 and 2000 land use values remaining the same. Polygons which changed their classification would appear as FeatureID's 1 through 4 in the table. Polygon 1, for example, was coded 231 in 1995 and 211 in 2000. Functionally we are still using the snapshot model, since we can only see the land use in 1995 and 2000, but we have mitigated the significant problem of redundancy by integrating both sets of polygons into a single attribute table. This method may be adapted to include *FromDate* and *ToDate* attributes to accommodate a wider variety of time resolutions, rather than the specific, discrete snapshots in 1995 and 2000.

To map or query the land use in 1995, one would use only the *LU1995* column. To map or query the land use in 2000, one would use only the *LU2000* column. This representation facilitates calculation of change statistics as well, since the attribute table clearly identifies those polygons which have changed land use.

One of the major limitations of this representation is that it does not indicate aggregation of polygons. For example, the region-entity representation of the polygon labelled as “1” in Figure 4-21 does not indicate that this polygon was part of polygon 5 in 1995, but part of the larger surrounding yellow polygon in 2000. While this limitation is not a problem for an application such as land use, where the main goal is to simply classify a given area, it is a significant issue for applications such as land ownership. In this latter application, we need to understand that two or more discrete polygons are logically a single parcel of land for purposes of ownership and taxation.

To handle this situation, we will use an additional table to manage higher level objects which combine the simple fragmented polygons into logical aggregations. This is true region-entity modeling, and it gets its name from the *region* feature class in the coverage data structure (as discussed in module 2 of this course), where we can logically group simple features to form more complex objects.

To illustrate this concept, we will look at a simple parcel subdivision, as shown in Figure 4-22.



**Figure 4-22 Simple Subdivision Example**

In this simple example, the image on the left shows the state of two parcels of land during the period 1980 to 1992. At the beginning of 1993, parcel A is subdivided to create two new parcels, C and D. To accommodate this change, we will again merge both spatial representations, to store all possible polygon representations in one dataset. We will consider all the resulting polygons as simple features, or polygonal primitives, which will be used as building-blocks for the parcels themselves, which will be represented as regions. The polygonal primitives and resulting region-entity table are shown in Figure 4-23.

1	2	3

FeatureID	Polygon	FromDate	ToDate
A	1	1980	1992
A	2	1980	1992
B	3	1980	2007
C	1	1993	2007
D	2	1993	2007
:		:	:

**Figure 4-23 Region-Entity Representation of the Subdivision Example**

The table on the right allows us to represent our land parcels (A, B, C and D) using aggregations of the polygonal primitives 1, 2 and 3. For example, parcel A was originally composed of the two smaller primitives 1 and 2. To represent this, two records are added to the table to show that A is composed of these two primitives. Each record has a *ToDate* of 1992 indicating that it ceased to exist after 1992. Parcels which do not change, such as parcel B, simply have a single record, indicating that they exist from 1980 through 2007. The new parcels, C and D, have records with *FromDate* 1993 to indicate that they were created at that time.

The problem with this approach is that querying and display can be difficult. No built-in functionality for this purpose is supported by the existing GIS application interface, so a complex series of definition queries and symbology may need to be developed for users to be able to view the database as of a given time period. This method can also produce significant fragmentation and overhead when trying to manage many time slices.

#### 4.3.5 Other Alternatives

Although the vast majority of temporal implementations rely on the snapshot or region-entity models or some variation on these, there are a number of alternative ways to represent time in a GIS. Several of these are summarized below.

The **update model** stores only one full version of a data set, with new information added as updates whenever changes occur. The updates do not store a full copy of the database, but instead store just the changes to the database, in manner similar to the transaction log concept we discussed earlier.

In practice, this method can be implemented using ArcMap's **archiving** functionality. Archiving takes advantage of the very simplest method for managing time that we have discussed, that of adding *FromDate* and *ToDate* attributes to features to indicate the time period during which an object is valid. A feature class which has archiving enabled, will create new records for a database entity, such as a land parcel, each time changes are made to the feature.

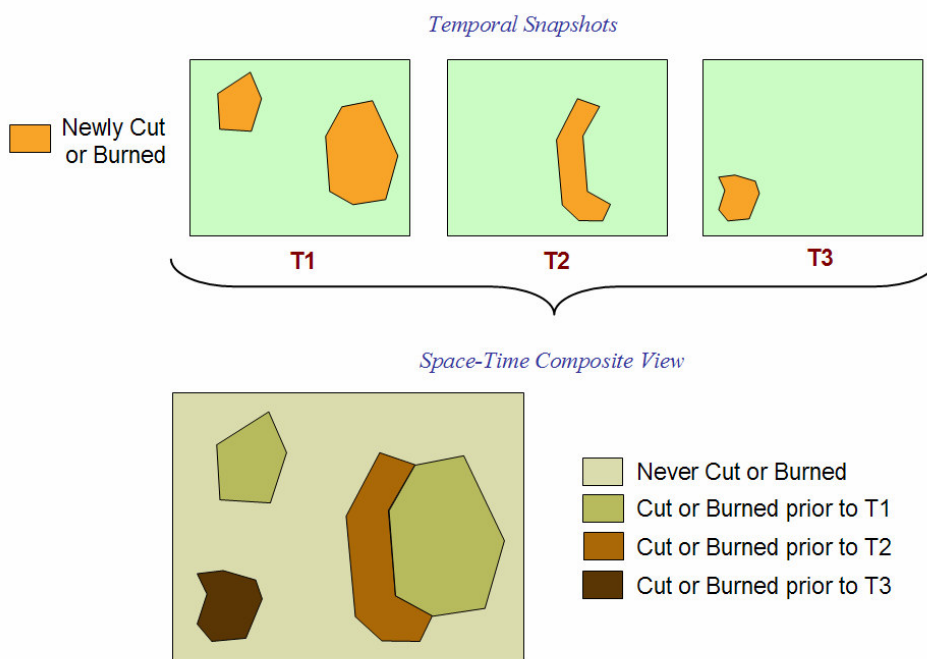
For example, if a land parcel were split into two new parcels today, the *ToDate* of the original parcel would be set to today to show that after today that parcel is no longer valid, and two new parcel features would be created with *FromDate* set to today to show that prior to today, these parcels did not exist. To effectively use these time-stamped records, a number of application tools are provided to view and query changes through time. There is some redundancy in this model, since this one change to a parcel produces three records in the archived feature class tables, but

the key difference from the snapshot model is that an unchanged feature appears in the database only once.

The **space-time composite model** is similar to the region-entity model, in that it stores both past and present data in the same layer. Here, the temporal snapshots are geoprocessed to bring all changes into a single layer, as was done with the region-entity method. The exact representation of the database, though, will be different depending upon the application. The result could be a single attribute indicating the time a given activity has taken place, or the time since an event. A very simple example might involve storing volcanic eruptions. A simple space-time composite model of eruptions might store the point locations of all eruptions, and an attribute indicating how long ago the eruption occurred. We could then use symbology or query clauses to alter the way we view or analyze such a feature class.

We might also use this approach to manage forests. Figure 4-24 shows a space-time composite showing when deforestation events (harvest or wildfire, for example) occurred. The images at the top show harvest or burn events occurring at three points in time. The composite version, at the bottom of the figure, shows how this might be represented in a single feature class, perhaps with an attribute indicating the length of time since the forest was harvested. Using this method, we might be able to judge the approximate age of each area of forest, or to visually understand how the forest has changed over time.

This model can be effective for thematically simple data, or data which have a small number of attributes with a small number of values. For example, Figure 4-24 is functionally representing a binary relationship: forested and deforested. Trying to show change in the CORINE land use, with many different land use attribute values, or land ownership with many changes to land parcel extents, will be cumbersome at best using this method. In addition, as changes are made to the data, snapshots must be integrated into the composite layer and attributes must be recalculated.



**Figure 4-24 Space-Time Composite Model**

ArcMap has one other mechanism for handling temporal data, an extension called the **Tracking Analyst**. Tracking analyst is a reasonable solution for managing objects which move or change status through time. Examples of such situations include moving objects such as vehicles or animals, or stationary objects which change status, such as manufacturing machinery or a remote weather sensor. Tracking analyst can work with real-time data, or archived temporal data, and provides a set of tools which allow users to view and query objects through time, and includes the ability to create animations showing movements or status changes.

## 4.4 Distributed DBMS

### 4.4.1 Introduction

We have for the most part, so far avoided discussion of the underlying architecture of the DBMS. During discussion of concurrency, we discussed many examples of several users accessing the same central database, which is an example of the client-server configuration. This is probably the single most common approach to providing access to data for a number of users.

The simplest method of data access, where only one GIS user exists, is to place the application software, such as ArcMap, and the geodatabase, such as a file geodatabase, on the same physical machine. This approach means that there is no network delay or competition between users for data, so it is very efficient. However, where more than one user exists in one organisation, this method precludes sharing of data between workstations, so it is an unlikely solution for anything but the smallest office.

The file server approach involves placing shared files on a central server, and having several users access these files as necessary, most likely from machines joined together in a network. This approach now allows users to share data, but they cannot work on a given data file at the same time, and responsibility for security, backup and recovery lies with the users. This model was very common in the early 1990's, where many data structures were single-user in nature, such as the shapefile.

As a result of the limitations of these simpler approaches to data storage and access, and the advances made in network and server technology, the client-server has become typical. It allows several users to access data being served by a DBMS which can control concurrency, security, data integrity, backup and recovery. The client-server configuration is not the only possible way to facilitate data access, however. This topic will explore some alternative configurations to the standard 1-server, multiple-client model.

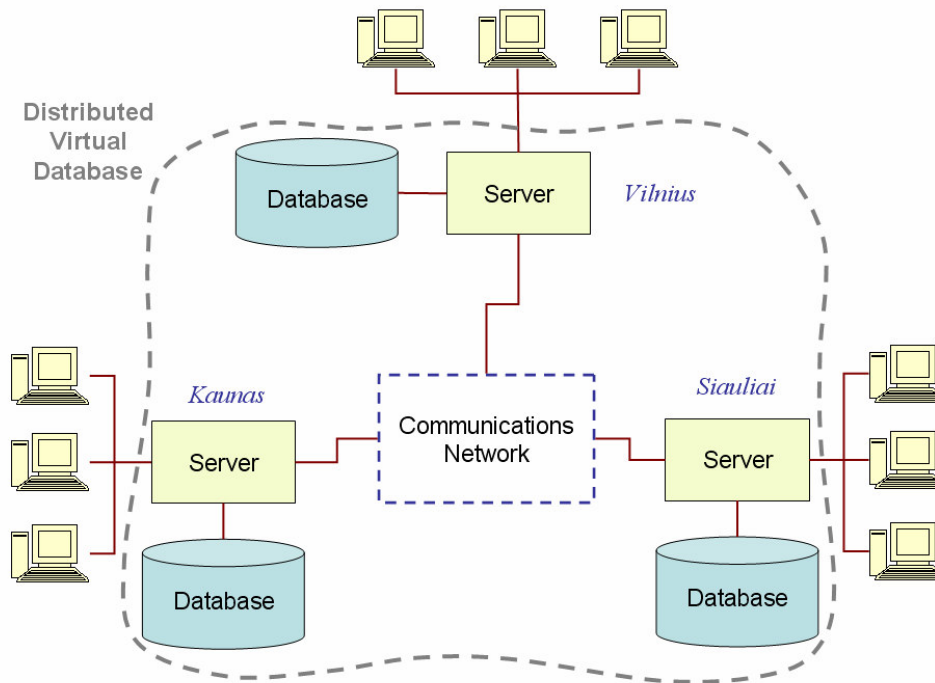
### 4.4.2 The Distributed Configuration

In **distributed processing**, "distinct machines can be connected together into some kind of a communications network in such a way that a single data-processing task can be spread across several machines in the network" (Date, 2000, p.50). The general concept is to break up a single, large database into several smaller, more manageable databases, and to store them on different computers which are connected by a network. In addition, processing can be divided among several different connected computers. Interest in distributed database management systems grew as a result of advances in network communication, and increased dispersion of business operations.

We can define a **distributed database** (DDB) as "a collection of multiple logically interrelated databases distributed over a computer network", and a **distributed database management system** (DDBMS) as "a software system which manages a distributed database while making the distribution transparent to the user" (Elmasri and Navathe, 2000, p.766). A DDBMS is theoretically capable of flexibility and power not found in centralized system.

Figure 4-25 shows an example of a distributed database configuration. Here, we have several geographically separated sites, and each site is valid, standalone database system. There are a set of discrete physical databases, which in effect become one, large *virtual* database when they

function as a unit. Each local database likely stores data most relevant to, and accessed by, the local users, but remote data may be accessed when necessary.



**Figure 4-25 Distributed Database Configuration**

Distributed databases have several advantages over the more common centralised database concept:

Increased Availability	Databases are more reliable, since there is less reliance on a single, central machine. A distributed system is able to shift operations if one computer fails. Also, because parts of the database are stored in separate locations, there is less likely to be locking or contention issues to hinder access.
Improved Access Speeds	When a large database is distributed over several sites, each component database is smaller, resulting in better response times. In addition, fewer users will be making requests of each small database than would be the case if all users accessed a single, large database.
Expansion facilitation	Adding more processors or increasing the database size are more easily accomplished in a distributed system.

Of course, these improvements come at a cost. Distributed databases also have several disadvantages:

Increased Complexity	Management of a distributed system is far more complex than a centralized system. The DBMS must pull together data from several sites, synchronize duplicated data, and balance processing loads among contributing nodes. In addition,
----------------------	---

	procedural complexity increases, since there may now be several database administrators at several discrete sites.
Increased Storage	Some distributed models (see replication below) require duplication of some data.

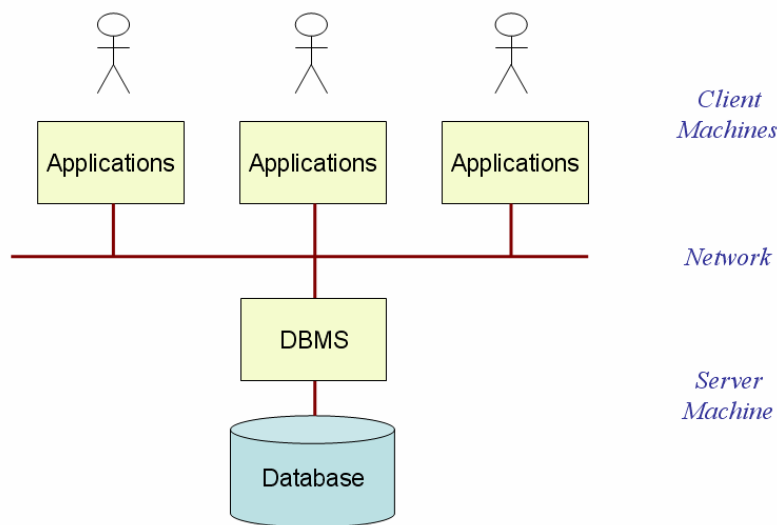
There are two aspects to a distributed database, the distribution of processing and the distribution of data storage. **Distributed processing** shares the databases processing among two or more nodes in the network. For example, distributed processing might allow the querying and data validation to be performed on one machine, and the report generation which uses the results to be performed on another machine. **Distributed data storage** involves the storage of enterprise data on several logically connected sites. To do this, data must be divided into a number of database fragments, which are then distributed among the participating sites. These fragments may or may not involve duplication of data, depending upon the configuration.

#### 4.4.3 The Client-Server Configuration

From a generalized point of view, virtually all current database systems are constructed in a client-server configuration. This is a very simple, two-part structure, where the DBMS application, or server, manages the database, and a number of client applications make requests of the server. Client applications may be tools which are provided with the DBMS to manage or otherwise work with the database, or they may be 3<sup>rd</sup> party applications which interact with the DBMS. The client applications are responsible for the user interface and any processing required prior to presenting results to the user. The server application is responsible for low-level interaction with the database, including database storage and access. Looking at an example that will be familiar to GIS students, the server might be the SQL Server RDBMS in which our geographic data reside, and the client applications would be ArcCatalog or ArcMap, which make data requests to SQL Server and display the results to the user. All data storage, integrity and access functions are performed by SQL Server.

This configuration is distinct from the file server architecture. With file servers, the entire file is transferred to the client, each client must have the entire DBMS application, and data integrity must be managed by the client applications. For example, an organisation which stores a series of shapefiles on a central file server which is accessed by several users is making use of a file server configuration. In this situation, the server is simply serving the shapefile contents to the ArcMap clients. The ArcMap software must be responsible for data integrity. In contrast, the client-server database relies on server software to process requests and return only the requested information to the client, dramatically reducing the network load.

Client and server applications may physically reside on the same machine, or on separate machines connected by a network, but the most common type of client-server configuration is shown in Figure 4-26. Here, several machines running client applications communicate with a single machine running the server application. Although it is not technically the case, the term client-server has come to apply almost exclusively to this situation.



**Figure 4-26 Common One-Server, Many-Client Configuration**

This type of configuration is a type of distributed system, since processing has been divided between two machines, so we might consider this architecture simply a special case of distributed systems in general. The client machine can process results and present them to a user, while the server machine is accessing the data and passing results to the client application. This is the primary advantage of the client-server architecture: that it allows the separation of processing which can result in improved performance. The other attractive thing about client-server configurations is that this separation of client applications allows the use of less powerful, more commonly-used personal computers (PCs) as clients. These machines tend to be less expensive than the more powerful database servers, and staff are more likely to have existing PC skills.

There are a number of concerns about the client-server configuration, however, particularly when it involves physically separated client and server machines. Firstly, the reliance on network connectivity means that performance will degrade as more users are added, or when distant users are accessing the database. This arrangement also means that there may be significant cost associated with operating and maintaining a large, central database server, and the reliance on this central machine raises concerns regarding reliability.

#### 4.4.4 Replication

The effectiveness of a DDBMS depends upon its ability to spread data among its member sites in a way which facilitates efficient retrieval. One of the most basic ways of allowing local access to distributed data is to allow copies to be made and placed at more than one physical location. This is primarily beneficial from a performance standpoint, since a local database does not experience significant network time delays, and it avoids competing with other users for data. Simply creating copies and circulating them to other staff members or offices, however, creates a significant amount of work trying to integrate changes made to each copy into a single, current database view. It also introduces opportunities for errors and inconsistencies to enter the database.

Most RDBMS applications have the ability to create duplicate copies of some or all of the database and integrate changes to the copy at a later date. The copies are called replicas, and the process of creating and managing replicas is called **replication**.

Replication is a two-stage process. The first stage is to create one or more replica databases from the master database. These replicas may then be distributed and used. The second stage is the process of integrating these separated versions after edits are complete, or when connection is possible between the replicas. This process is functionally the same as the version reconciliation process described in module 3, and is sometimes called **synchronization**. As with version reconciliation, it is only those situations where two different changes are made to the same entity that cause difficulty. All other changes may be integrated in an automated way.

Replication may be attractive in many situations. Some of these are summarised below:

Poor Communication	Where discrete offices or installations are separated by either a poor or non-existent communications network, it may be necessary to use replication. A common example is the staff member who must travel to remote locations as part of their work, perhaps to perform remote field data collection. In such a case, a replica could be taken to the field on a portable computer and synchronised when returned. A replica may be disconnected entirely from the network for a period of time.
Data Access	In cases where users at times cannot access data due to resource locks, replication may allow several users to continue working until a time of less activity, when the synchronisation can be performed.
Load Balancing	During times of extreme resource use by many users, replication may allow users to make use of local resources (e.g., on their own machine) rather than the resources of the central DBMS. Separating resource-intensive tasks to separate machines may increase the overall performance of the system.
Quality Assurance	Replication may be used to isolate part or all of a database where there exist data quality concerns. Once processing is complete on the replica, it may be thoroughly reviewed before being integrated into the master database. For example, a contractor or consultant may be given a replica to work with, and it may be formally reviewed and approved prior to synchronization with the master database.

#### 4.4.5 Fragmentation

An alternative to replication as a means of data distribution is called **data fragmentation**. Fragmentation is a process where a single data object, such as a database table, is broken into two or more fragments which are then stored at separate sites. Each site maintains a **distributed data catalog**, which identifies the location and nature of each fragment, to help determine how to gather the necessary fragments for a given query or operation. There are three basic strategies for dividing data tables: horizontal, vertical and mixed fragmentations.

To illustrate these fragmentation methods, we will use a simple table showing cities in Lithuania, with attributes including the city name, the county it lies in, the population in 2006, and the X and Y coordinates of the city centre. Figure 4-27 shows this table.

CityID	Name	County	Pop2006	XCoord	YCoord
1	Alytus	Alytus	69,145	502600	6028900
2	Druskininkai	Alytus	16,641	492400	5986000
3	Vilnius	Vilnius	541,824	582600	6061500
4	Grigiskes	Vilnius	11,567	567000	6059200
5	Lentvaris	Vilnius	11,838	561600	6045800

**Figure 4-27 Example CITIES Table**

Horizontal fragmentation of tables involves breaking a table into subsets (fragments) of rows. Each fragment is stored at a different site, and each fragment has unique rows. All fragments have the same attributes. When horizontally fragmenting our CITIES table, we might wish to distribute this table to two counties, each responsible for the cities within its boundaries. It is logical, then to break this table according to which county each city lies within, each fragment containing all attributes of the relevant cities. This is a horizontal fragmentation, since we are dividing the table rows into fragments. The resulting fragmentation appears in Figure 4-28.

CityID	Name	County	Pop2006	XCoord	YCoord
1	Alytus	Alytus	69,145	502600	6028900
2	Druskininkai	Alytus	16,641	492400	5986000

CityID	Name	County	Pop2006	XCoord	YCoord
3	Vilnius	Vilnius	541,824	582600	6061500
4	Grigiskes	Vilnius	11,567	567000	6059200
5	Lentvaris	Vilnius	11,838	561600	6045800

**Figure 4-28 Horizontal Fragmentation of the CITIES Table**

By dividing the table in this manner, the vast majority of work being performed in Vilnius, for example, will be performed against a local database table. If a user requires records for all cities in Lithuania for an operation, the DDBMS must detect this and perform a union of the fragments prior to returning the results. Thus the DDBMS must know the fragmentation criteria and be able to

reconstruct larger fragments or the entire table from remote fragments when necessary. This process can add significant complexity to the DDBMS.

Vertical fragmentation of tables involves breaking the table into groups of attributes, or columns. This process is similar to the normalization process, where columns corresponding to separate “themes” are broken into separate tables and joined together as necessary during querying. Here, however, the tables are split by business function, or by logical groupings according to how user groups in separate sites will use the information.

For example, we might have separate database sites for two government agencies. One agency is interested in the populations of cities, and the other is interested only in the geographic location of cities. We could then vertically fragment our CITIES table as shown in Figure 4-29. Again, the vast majority of queries will likely come from the local fragment, but where columns are required from a remote fragment, the DDBMS must perform a join operation to bring the two sets of columns together prior to data presentation.

CityID	Name	County	Pop2006
1	Alytus	Alytus	69,145
2	Druskininkai	Alytus	16,641
3	Vilnius	Vilnius	541,824
4	Grigiskes	Vilnius	11,567
5	Lentvaris	Vilnius	11,838

CityID	Name	XCoord	YCoord
1	Alytus	502600	6028900
2	Druskininkai	492400	5986000
3	Vilnius	582600	6061500
4	Grigiskes	567000	6059200
5	Lentvaris	561600	6045800

**Figure 4-29 Vertical Fragmentation of the CITIES Table**

Mixed fragmentation combines the horizontal and vertical strategies, such that each fragment may have a subset of both rows and columns. This might be effective in our CITIES example, if both counties had two departments. In such a case, we might fragment the table into four tables, as shown in Figure 4-30.

CityID	Name	County	Pop2006
1	Alytus	Alytus	69,145
2	Druskininkai	Alytus	16,641

CityID	Name	XCoord	YCoord
1	Alytus	502600	6028900
2	Druskininkai	492400	5986000

CityID	Name	XCoord	YCoord
3	Vilnius	582600	6061500
4	Grigiskes	567000	6059200
5	Lentvaris	561600	6045800

CityID	Name	County	Pop2006
3	Vilnius	Vilnius	541,824
4	Grigiskes	Vilnius	11,567
5	Lentvaris	Vilnius	11,838

**Figure 4-30 Mixed Fragmentation of the CITIES Table**

In all fragmentation methods, there will be significant overhead in pre-processing queries to allow for a join or union operation as part of the query. This overhead is hopefully offset by the fact that in most cases, queries are performed against the local fragments only, reducing the response time for the majority of queries.

#### 4.4.6 Distributed DBMS and GIS

We have discussed, at several points in this course, how the GIS application manages the behaviour of a geodatabase, and the underlying DBMS manages the data. In the past, as with shapefiles for example, almost all of the processing was done purely by the GIS software, with very little being done by the underlying database. The geodatabase model and ArcSDE moved much of the functionality to the DBMS, so that functions relating to retrieval, backup and recovery, security and some data integrity constraints were no longer handled by the GIS application, or the client software. More recently, with the adoption of standard geometric data types (rather than binary-encoded BLOB fields) and spatial query extensions to SQL, we are seeing the potential for much more of GIS work to be handled by the DBMS itself.

We still see the situation, however, where the DBMS does not “understand” the meaning, or the purpose of many objects stored within it. For example, the DBMS (or other 3<sup>rd</sup> party applications, for that matter) cannot use spatial SQL commands to view versioned features, since they have no understanding of the DEFAULT version and the delta tables which store changes to DEFAULT since the version was created. We need to use client applications, such as ArcMap, to view these versioned feature classes.

This is also the case with distributed processing and data storage, as they relate to GIS. For example, virtually all commercial database programs support replication, but we cannot use the DBMS replication mechanisms since doing so will compromise complex objects like topology,

relationship classes and geometric networks. Instead, we must use the replication mechanisms provided by the client applications. So in a sense, we can use some of the mechanisms of a DDBMS, but we cannot move to a full DDBMS because so much of the necessary data manipulation lies with the client application.

We see a similar situation with the use of distributed processing and distributed data storage using fragmentation (rather than replication). Several commercial database systems, including Ingres, Oracle and DB2 have begun moving toward the concept of **grid computing**. The concept underlying grid computing is that all of the resources of a network, including processors, storage, data and memory should be pooled into a single logical entity. Demands placed on a grid computing database can be balanced by making use of any of the discrete hardware or software components in the system, as required. The resulting system is theoretically faster, cheaper (since we no longer need massive, powerful database servers) and more reliable (since we no longer rely on that central database server exclusively). This is fairly new technology, with Oracle's first grid computing version (10g) being released about four years ago.

While ArcMap supports the grid computing versions of Oracle, for example, there will be limited gains from using it. The underlying DBMS, like Oracle, cannot effectively distribute database fragments if it cannot interpret the contents of complex table structures such as those used in versioning, geometric networks and topology. In addition, because so much of the processing is still performed by the GIS client applications, the ability of a grid computing DBMS to control the allocation of processing power is severely limited.

## **Module Self-Study Questions:**

6. What elements of standard SQL are likely to be different from one SQL dialect to another?
7. Describe the distinction between data definition commands and data manipulation commands in SQL.
8. Describe the trade-off in performance as a result of changing the grid size used by a grid index.
9. What are the limitations or disadvantages of the snapshot model for incorporating time into a GIS database?
10. Describe the difference between replication and fragmentation as a means of distributing data storage in a distributed database.

## References

Date, C.J. *An Introduction to Database Systems*. Addison-Wesley, 2000.

Elmasri, R., and Navathe, S.B. *Fundamentals of Database Systems*. Addison-Wesley, 2000.

ESRI. *Building a Geodatabase*. ESRI Press, 2005.

ESRI. *ArcGIS On-line Help*. Viewed July, 2007.  
<http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm>

ESRI. *Working with the Geodatabase Using SQL*. ESRI Technical Paper, 2004. Viewed July, 2007.  
[http://downloads.esri.com/support/whitepapers/sde\\_/GeodatabaseUsingSQL\\_2\\_30mar06.pdf](http://downloads.esri.com/support/whitepapers/sde_/GeodatabaseUsingSQL_2_30mar06.pdf).

McBride, S., Ma, D. and Escobar, F. *Management and Visualisation of Spatiotemporal information in GIS*. Presented at SIRC 2002 – The 14th Annual Colloquium of the Spatial Information Research Centre, University of Otago, Dunedin, New Zealand. Viewed July, 2007.  
[http://www.business.otago.ac.nz/sirc/conferences/2002\\_SIRC/06\\_McBride.pdf](http://www.business.otago.ac.nz/sirc/conferences/2002_SIRC/06_McBride.pdf)

Oracle Corp. *Grid Computing with Oracle*. An Oracle Technical White Paper. March, 2005. Viewed July 2007.  
[http://www.oracle.com/technology/tech/grid/pdf/gridtechwhitepaper\\_0305.pdf](http://www.oracle.com/technology/tech/grid/pdf/gridtechwhitepaper_0305.pdf)

Rob, P., and Coronel, C. *Database Systems: Design, Implementation and Management*. Thomson Learning, 2000.

Shekhar, S., and Chawla, S. *Spatial Databases: A Tour*. Prentice-Hall, 2003.

Zeiler, M. *Modeling Our World*. Redlands, CA: ESRI Press, 1999.