NATIONAL OPEN UNIVERSITY OF NIGERIA

COURSE CODE: CIT 844

COURSE TITLE:

ADVANCED DATABASE MANAGEMENT SYSTEM

COURSE GIUDE

ADVANCED DATABASE MANAGEMENT SYSTEM

Course Developer/Writer	Dr. Olusegun Folorunso Department of Computer Science University of Agriculture Abeokuta
Programme Leader	Prof. Afolabi Adebanjo National Open University of Nigeria
Course Coordinator	National Open University of Nigeria

NATIONAL OPEN UNIVERSITY OF NIGERIA

CIT 844 ADVANCED DATABASE MANAGEMENT SYSTEM

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Office
No. 5 Dar es Salaam Street
Off Aminu Kano Crescent
Wuse II, Abuja
Nigeria
e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published by
National Open University of Nigeria
Printed 2009
ISBN: XXX-XXX-XXX-X
All Rights Reserved

CONTENTS PAGE

Introduction

The course, Advanced Database Management System, is a core course for students studying towards acquiring the Master of Science in Information Technology. In this course we will study about the Database Management System as a key role in Information Management. Various principles of database management system (DBMS) as well as its advanced features are discussed in this course. This course also considers distributed databases and emerging trends in database system.

The overall aim of this course is to introduce you to various ways of designing and implementing database systems, features and distributed databases. In structuring this course, we commence with the basic design and implementation of relational databases.

There are four modules in this course, each module consists of units of topics that you are expected to complete in 2 hours. The four modules and their units are listed below.

What You Will Learn in this Course

The overall aims and objectives of this course provide guidance on what you should be achieving in the course of your studies. Each unit also has its own unit objectives which state specifically what you should be achieving in the corresponding unit. To evaluate your progress continuously, you are expected to refer to the overall course aims and objectives as well as the corresponding unit objectives upon the completion of each.

Course Aims

The overall aims and objectives of this course will help you to:

1. Develop your knowledge and understanding of the underlying principles of Relational Database Management System
2. Build up your capacity to learn DBMS advanced features
3. Develop your competence in enhancing database models using distributed databases
4. Build up your capacity to implement and maintain an efficient database system using emerging trends.

Course Objectives

Upon completion of the course, you should be able to:

1. Describe the basic concepts of Relational Database Design
2. Explain Database implementation and tools
3. Describe SQL and Database System catalog.
4. Describe the process of DB Query processing and evaluation.
5. Discuss the concepts of transaction management.
6. Explain the Database Security and Authorization.
7. Describe the design of Distributed Databases.
8. Know how to design with DB and XML.
9. Describe the basic concept of Data warehousing and Data mining
10. Discuss the emerging Database Models Technologies and Applications

Working through this Course

We designed this course in a systematic way, so you need to work through it from Module one, Unit 1 through to Module four, Unit 3.
This will enable you appreciate the course better.

Course Materials

Basically, we made use of textbooks and online materials. You are expected to I, search for more literature and web references for further understanding. Each unit has references and web references that were used to develop them.

Online Materials

Feel free to refer to the web sites provided for all the online reference materials required in this course.
The website is designed to integrate with the print-based course materials. The structure follows the structure of the units and all the reading and activity numbers are the same in both media.

Study Units

Unit 2: Technologies and Applications

Unit 3 PostgreSQL & Oracle

Module one describes Database Design and Implementation.
Module Two explains the DBMS advanced features.
Module Three discusses the Distributed Database.
Module Four discusses Emerging trends in DBMS including technologies and applications.

Equipment

In order to get the most from this course, it is essential that you make use of a computer system which has internet access.
Recommended System Specifications:
Processor
2.0 GHZ Intel compatible processor
1GB RAM
80 GB hard drive with 5 GB free disk
CD-RW drive.
3.5" Floppy Disk Drive
TCP/IP (installed)
Operating System
Windows XP Professional (Service Pack 2)
Microsoft office 2007
Norton Antivirus
Monitor*
19-inch
1024 X 768 Resolution
16-bit high color
*Non Standard resolutions (for example, some laptops) are not supported.
DBMS Tools
ORACLE
PostgreSQL
Hardware
Open Serial Port (for scanner)
120W Speakers
Mouse + pad

Windows keyboard
Laser printer

Hardware is constantly changing and improving, causing older technology to become obsolete. An investment in newer, more efficient technology will more than pay for itself in improved performance results.

If your system does not meet the recommended specifications, you may experience considerably slower processing when working in the application. Systems that exceed the recommended specifications will provide better handling of database files and faster processing time, thereby significantly increasing your productivity.

Assessment

The course, Advanced Database Management Systems entails attending a two-hour final examination which contributes 50% to your final grading. The final examination covers materials from all parts of the course with a style similar to the Tutor- marked assignments.

The examination aims at testing your ability to apply the knowledge you have learned throughout the course, rather than your ability to memorize the materials. In preparing for the examination, it is essential that you receive the activities and Tutor-marked assignments you have completed in each unit. The other 50% will account for all the TMA's at the end of each unit.

Tutor-Marked Assignment

About 20 hours of tutorials will be provided in support of this course.
You will be notified of the dates, time and location for these tutorials, together with the name and phone number of your tutor as soon as you are allotted a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail your TMAs to your tutor well before the due date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by phone, e-mail if you need help.

The following might be circumstances in which you would find help necessary. You can also contact your tutor if:

you do not understand any part of the study units or the assigned readings
you have difficulty with the TMAs
you have a question or problem with your tutor's comments on an assignment or with the grading of an assignment

You should try your best to attend tutorials, since it is the only opportunity to have an interaction with your tutor and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain maximum benefit from the course tutorials, you are advised to prepare a list of questions before attending the tutorial. You will learn a lot from participating in discussions actively.

Course Overview

This section proposes the number of weeks that you are expected to spend on the three modules comprising of 30 units and the assignments that follow each of the unit.

We recommend that each unit with its associated TMA is completed in one week, bringing your study period to a maximum of 30 weeks.

How to Get the Most from this Course

In order for you to learn various concepts in this course, it is essential to practice. Independent activities and case activities which are based on a particular scenario are presented in the units. The activities include open questions to promote discussion on the relevant topics, questions with standard answers and program demonstrations on the concepts. You may try to delve into each unit adopting the following steps:

1. Read the study unit
2. Read the textbook, printed or online references
3. Perform the activities
4. Participate in group discussions
5. Complete the tutor-marked assignments
6. Participate in online discussions

This course makes intensive use of materials on the world-wide web.

Specific web address will be given for your reference. There are also optional readings in the units. You may wish to read these to extend your knowledge beyond the required materials. They will not be assessed.

Summary

The course, Advanced Database Management Systems is intended to develop your understanding of the basic concepts of database systems, thus enabling you acquire skills in designing and implementing Database Management Systems. This course also provides you with practical knowledge and hands-on experience in implementing and maintaining a system. We hope that you will find the course enlightening and that you will find it both interesting and useful. In the longer term, we hope you will get acquainted with the National Open University of Nigeria and we wish you every success in your future.

ADVANCED DATABASE MANAGEMENT SYSTEM

MAIN COURSE

Course Developer/Writer	Dr. Olusegun Folorunso Department of Computer Science University of Agriculture Abeokuta
Programme Leader	Prof. Afolabi Adebanjo National Open University of Nigeria
Course Coordinator	National Open University of Nigeria

NATIONAL OPEN UNIVERSITY OF NIGERIA

CIT 844 ADVANCED DATABASE MANAGEMENT SYSTEM

National Open University of Nigeria

Headquarters

14/16 Ahmadu Bello Way

Victoria Island

Lagos

Abuja Office

No. 5 Dar es Salaam Street

Off Aminu Kano Crescent

Wuse II, Abuja

Nigeria

e-mail: centralinfo@nou.edu.ng

URL: www.nou.edu.ng

Published by

National Open University of Nigeria

Printed 2009

ISBN: XXX-XXX-XXX-X

All Rights Reserved

CONTENTS PAGE

Unit 2: The Major Application Domains

Unit 3 PostgreSQL

MODULE 1: DATABASE DESIGN & IMPLEMENTATION

UNIT 1: RELATIONAL DATABASE DESIGN

1.0 INTRODUCTION

This unit discusses extensively on relational data model. The model was first introduced by Ted Codd of IBM research in 1970 in a classic paper, and attracted immediate attention due to its *simplicity* and *mathematical foundations***.** The model uses the concept of mathematical relation, which looks somewhat like a table of values, as its basic building blocks, and has its theoretical basics in *set theory* and *first order predicate logic*. In this unit, we will discuss the basic characteristics for the relational model and its normalization processes. The model has been implemented in a large number of commercial systems over the last years.

1.1 OBJECTIVES.

To understand the relational database model.
To understand the concepts of normalization in database design

1.2 WHAT IS RELATIONAL DATABASE?

A **relational database** is a database that groups data using common attributes found in the data set. The resulting "clumps" of organized data are much easier for people to understand. For example, a data set containing all the real estate transactions in a town can be grouped by the year the transaction occurred; or it can be grouped by the sale price of the transaction; or it can be grouped by the buyer's last name; and so on. Such a grouping uses the relationalmodel (a technical term for this is schema). Hence such a database is called a "relational database”. The software used to do this grouping is called a relationaldatabasemanagementsystem. The term "relational database" often refers to this type of software. Relational databases are currently the predominant choice in storing financial records, manufacturing and logistical information, personnel data and much more.

1.2.1 Relational Database Model Concept.

The relational model represents the database as a collection of relations. Informally, each relation resembles a table of values or, to some extent a "**flat**" file of record. For example, the university database of files that was shown earlier is considered to be in the relational model. However, there are important differences between relations and files, as we shall soon see.

When a relation is thought of as a table of values, each row in the table represents a collection of related data values. In the relational model, each row in the table represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help in interpreting the meaning of the row which represents facts about a particular student entity. The column names - Name, Student, Number, Class, Major - specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a **tuple**, a column header is called an **attribute**, and the table is called a **relation**. The data type describing the types of values that can appear in each column is called a **domain**. We now define these terms – domain, tuple, attribute, and relation – more precisely.

Domains, Attributes, Tuples, & Relations

A domain D is a set of atomic values. By atomic we mean that each value in the domain is indivisible as far as the relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values. Some examples of domains follow:-

GSM phone-numbers: The set of 11-digit numbers valid in the Nigeria. E.g. 0803-5040- 707.

Local-phone-numbers: The set of 6-digit phone numbers valid within a particular area code in Nigeria. E.g. 245290

Names: The set of names of persons.

Grade-point-averages: Possible values of computed grade point averages; each must be a real (floating point) number between 0 and 5

Employee-ages: Possible ages of employees of a company, each must be a value between 18 and 65 years old

Academic-department-names: The set of academic department names, such a computer science, economics and physics, in a university.

Academic-department-codes: The set of academic department codes, such as CSC, ECON, and PHYS, in a university.

The preceding are called logical definitions of domains. A data type of format is also specified on each domain. For example, the data type for the domain GSM phone-numbers can be declared as a character string of the form (dddd)dddd-ddd, e.g. 0803-5640-707 where each d is a data type. For Employee-ages, the data type is an integer number between 18 and 65, for Academic-department-names, the data type is the set of all character strings that represent valid department names. A domain is thus given a name, data type, and format. Additional information for interpreting the values of a domain can also be given. For example, a numeric domain such as person-weights should have the units of measurement - kilograms.

A relations schema

, denoted by ([illegible]), is made up of a relation name R and a list of attributes [illegible]. Each **attributes** is the name of a role played by some domain

in the relation schema

.

is called the domain of

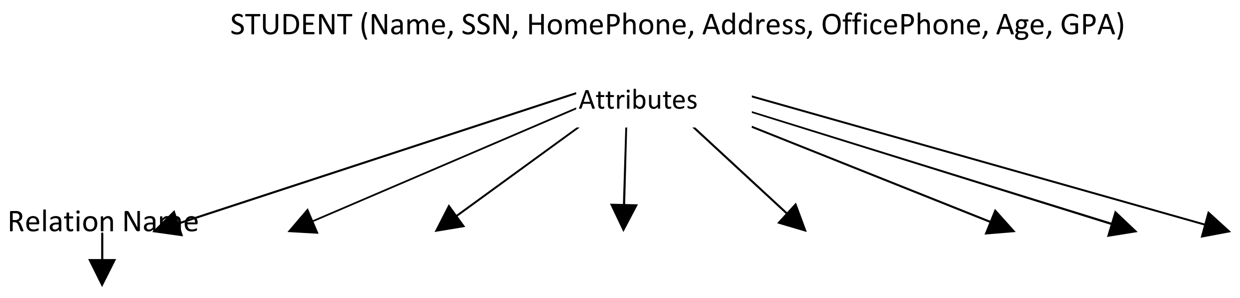
, and is denoted by (). A relation schema is used to describe a relation

is called the **name** of this relations. The **degree** of a relation is the number of attributes

of its relation schema.

An example of a relation schema for a relation of degree 7, which describes university students, is the following:

Table 1.0 The attributes and tuples of a relation STUDENT


STUDENT (Name, SSN, HomePhone, Address, OfficePhone, Age, GPA)
Attributes
Relation Name

Student	Name	Ssn	Home phone	Address	Office Phone	Age	Gpa
	Olumide Enoch	305-61-2435	080-3616	2, Kings way Rd.	null	19	4.21
	Adamson Femi	381-62-1245	080-4409	125, Allen Avenue	null	18	3.53
Tuples	Yisa Ojo	22-11-2320	null	34,Obantoko Road	749-1253	25	2.89
	Charles Olumo	489-22-1100	080-5821	256 Grammar School	749-6492	28	3.93
	Johnson Paul	533-69-1238	0804461	7384, Paul Job Road	null	19	3.25

For this relation schema, STUDENT is the name of the relation, which has seven attributes.

We can specify the following previously defined domains for some of the attributes of the STUDENT relations:

[illegible]([illegible]) = [illegible];
[illegible]([illegible]) = [illegible] – [illegible] – [illegible];
[illegible]([illegible]) = [illegible] – [illegible] – [illegible];
[illegible]([illegible]) = [illegible] – [illegible] – [illegible];

$$□□□ (□□□) = □□□□□ – □□□□□ – □□□□□□□□.$$

A relation (or relation state) $□$of the relation schema $□(□_1, □_2, \dots, □_□)$also denoted by

$□(□)$, is a set of n-tuples $□ = (□_1, □_2, \dots, □_□)$. Each n-tuple t is an ordered list of n values

$$□ = < □_1, □_2, \dots, □_□ >,$$

where each value $□_□$, $1 \le □ \le □$, is an element of $□□□(□_□)$ or is a special null value. The $□□□$ value in tuple t, which correspondence to the attribute $□_□$ is referred to as $□[□_□]$. The terms relation intension for the schema $□$and relation extension for a relation state $□(□)$ are also commonly used.

Table 1.0 shows an example of a STUDENT relation, which corresponds to the STUDENT schema specified above. Each tuple in the relation represents a particular student entity. We display the relation as a table, where each tuple is shown as a row and each attribute corresponds to a column header indicating a role or interpretation of the values in that column. Null values represent attributes whose values are unknown or do not exist for some individual STUDENT tuples.

The above definitions of a relation can be restated as follows. A relation □(□) is a

mathematical relation of degree n on the domains □□□(□1), □□□(□2), ..., □□□(□□),

which is a subset of the Cartesian product of the domains that define □:

$$() \subseteq ((_1) \times (_2) \times \ldots \times ()$$

The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the number of values or cardinality of a domain D by |D|, and assume that all domains are finite, the total number of tuples in the Cartesian product is:

$$|(_1)| \times |(_2)| \times \ldots \times |()|$$

Characteristics Of Relations

The earlier definition of relations implies certain characteristic that makes a relation different from a file or a table. We now discuss some of these characters.

Ordering of Tuples in a Relation: A relation is defined as a set of tuples. Mathematically, elements of a set have no order among them; hence tuples in a relation do not have any particular order. However in a file, records are graphically stored on disk so there always is an order among the records. This ordering indicates first, second, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.

Tuple ordering is no part of a relation definition, because a relation attempts to represent facts at a logical or abstract level. Many logical orders can be specified on a relation, for example, tuples in this STUDENT relation in Table 1.0 could be logically ordered by values of Name, SSN, Age, or some other attribute. The definition of a relation does not specify any order, there is no preference for one logical ordering over another. Hence, the relation displayed in Table 1.1 is considered identical to the one shown in Table 1.0. When a relation is implemented as a file, a physical ordering may be specified on the records of the file.

<u>Ordering of values within a tuple, and an alternative definition of a relation:</u> According to the preceding definition of a relation, an n-tuple is an ordered list of n values, so the ordering of values in a tuple and hence of attributes in a relation schema.

An alternative definition of relation can be given, making the ordering of value in a tuple unnecessary. In this definition, a relation schema R = (A_1,A_2,..., A_n) is a set of attributes and relation $()$ is a finite set of mappings r = [t_1,t_2,...,t_m], where each tuple is a mapping from R to D, and D is the union of attribute domains; that is, $= (_1) \cup (_2) \cup \ldots \cup ()$.

Table 1.1. The relation STUDENT from Table 1.0, with a different order of tuples.

Student	Name	Ssn	Home phone	Address	Office Phone	Age	Gpa
Tuples	Olumide Enoch	305-61-2435	080-3616	2, Kings way Rd.	Null	19	4.21
	Adamson Femi	381-62-1245	080-4409	125, Allen Avenue	null	18	3.53
	Yisa Ojo	22-11-2320	Null	34,Obantoko Road	749-1253	25	2.89
	Charles Olumo	489-22-1100	080-5821	256, Grammar School	749-6492	28	3.93
	Johnson Paul	533-69-1238	0804461	7384, Paul Job Road	null	19	3.25

In this definition, t(A_1) must be in dom(A_1) for $1 \leq \leq$ for each mapping t in r. Each mapping is called a tuple.

t=<(Name; Yisa Ojo), (SSN,422-112320), (HomePhone, null), (Address, 34 Obantoko Road). (Office Phone, 749-1253), (GPA, 2.89), (HomePhone, null) >

749-1253), (GPA, 2.89), (HomePhone; null)>

Figure 1.0 Two identical tuples when order of attributes and values in not part of the definition of relation.

According to this definition, a tuple can be considered as a set of (<attribute>, <value>) pairs, where each pair gives the value of the mapping form an attributes A_1 to a value v_1 from dom(A_1). The ordering of attributes is not important, because the attribute name appear with its value. By this definition, the two tuples shown in figure 1.0 are identical. This makes sense at an abstract or logical level, since they are really in no reason to prefer having one attribute value appear before another in a tuple.

When a relation is sense at an abstracts are physically ordered as fields within a records. We will use the first definition of relation, where the attributes and the values within tuples are ordered, because it simplifies much of the notation. However, the alternative definition given here is more general.

Values in the Tuples: Each value in a tuple is an atomic value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multi-valued attributes are not allowed. Much of the theory behind the relational model was developed with this assumption in mind, which is called the first normal form assumption. Multi-valued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes. Recent research in the relational model attempt to remove these restrictions by using the concepts of no first normal form or nested relations.

Interpretation of a Relation: The relation schema can be interpreted as a declaration or a type of assertion. For example, the schema of the STUDENT relation of Table 1.0 asserts that, in general, student entity has a Name, SSN, Home phone, Address, Office phone, Age, and GPA. Each tuple in the relation can then be interpreted as a fact or a particular instance of the assertion. For example, the first tuple in Table 1.0 asserts the fact that there is a STUDENT whose name is Olutunde Enoch, SSN is 305-61-2435, Age in 19 and so on.

Notice that some relations may represent facts about entities, whereas other relations may represent fact about relationship. For example, a relation schema MAJORS (Student SSN, Department Code) asserts that students major in academic department, a tuple in this relation relates a student to his or her major department. Hence, the relational model represents fact about both entities and relationship uniformly as relations.

1.2.2 Relational Constraints And Relational Database Schemas

In this unit, we discuss the various restrictions on data that can be specified on relational database schema in the form of constraints. These include domain constraints, key strains, called data dependencies (which include functional dependencies and multi-valued dependencies), are used mainly for database design by normalization.

Domain constraints

Domain constraints specify that the value of each attribute *A* must be an atomic value from the domain *dom(A)*. We have already discussed the ways in which domains can be specified above. The data types associated with domains typically include standard numeric data type for integers (such as short-integer, long-inter) and real number (float and double-precision float). Character, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money data types. A sub-range of values from a data type or an enumerated data type may describe other possible domains where all possible values are explicitly listed.

Key constraints and constraints on null

A relation is defined as a set of tuples. By definition, all elements of a set are distinct, hence, all tuples in relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes. Usually, there are other subsets of attributes of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two distinct tuples t_1 and t_2 in a relation sate r or R, we have the constraint that $t_1[SK]^1t_2[SK]$.

Any such set attributes SK is called a superkey of the relation schema R. A superkey SK specifies a uniqueness constraint that no two distinct tuples in a state r of R can have the same value for SK. Every relation has at least one default superkey the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a key, which has no redundancy. A key K of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K that is not a superkey of R. Hence, a key is a minimal superkey- that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold.

For example, consider the STUDENT relation of Table 1.0 the attribute set (SSN) is a key of STUDENT because no two student tuples can have the same value for SSN. Any set of attributes that include SSN for example, (SSN, Name, Age) is a superkey. However, the superkey (SS, Name, Age) is not a key for STUDENT, because removing Name or Age of both from the set still leaves us with a superkey.

The value of key attribute can be used to identify uniquely each tuples in the relation. For example, the SSN value 305-61-2435 identifies uniquely the tuples corresponding to Benjamin

Bayer in the STUDENT relation. Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on every relation state of the schema. A key is determined from the meaning of the attributes, and the property is time-invariant; it must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the student relation in Table 1.0 as a key, because there is no guarantee that two student with identical names will never exist.

In general, a relation schema may have more than one key. In this case, each of the keys is called a candidate key. For example, the CAR relation in Table 1.2 has candidate keys: License Number and Engine Serial Number. It is common to designate the candidate key as the primary key of the relation. This is the key whose value is used to identify tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined, as shown in Table 1.2. Notice that, when a relation schema has several candidate key, the choice of one to become primary key is arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. Another constraint on attributes specifies whether null values are or are not permitted. For example, if every student tuple must have a valid non-null value for the *Name* attribute, then *Name* of student is constrained to be NOT null.

Table 1.2 The CAR relation wish two candidate key: license Number and Engine Serial number.

CAR	License Number	Engine Serial Number	Make	Model	Year
	AB-419 KJA	A69352	BMW	800 series	96
	XA-893-AKM	B43696	Bluebird	Datsun	99
	AAB383 WDE	X83554	Datsun	Toyota	95
	LA 245 YYY	C43742	Golf	Volkswagen	93
	DE382 MNA	Y82935	Mercedes	190-D	98
	FE 107 EKY	U028365	Toyota	Toyota	98

Relational Databases And Relational Database Schemas

So far, we have discussed single relations and single relation schemas. A relational database usually contains may relations, with tuples relations that are related in various ways. In these units we define a relational database and a relational database schema

is a set of relation schema ■ $\{{}_1\ {}_2, \ldots, {}\}$ and a set of integrity constraints IC. A relational database state DB of a set of relation states $= \{ _1\ _2, \ldots, _m \}$ such that each is a state of and such that the relation states satisfy the integrity constraints specified in IC. Figure 1.1 shows a relational database schema

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUP	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORK ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT NAME	SEX	BDATE	RELATIONSHIP

Figure 1.1. Schema diagram for the company relational database schema; the primary keys are underlined.

1.2.3 Update Operations And Dealing With Constraint Violations

The operations of the relational model can be categorized into retrievals and updates. The relational algebra operations, which can be use to specify retrievals, are discussed in details in a later unit. In this unit, we concentrate on the update operations. There are three basic update operations on relations (1) **insert**, (2) **delete** and (3) **modify**. Insert is used to insert a new tuple or tuples in relation: Delete is used to delete tuple; and **Update** (or **modify**) is used to change the values of some attribute in existing tuple. Whenever update operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this unit we discuss the type of constraints that may be violated by each update operation and the types of actions that may be taken if an update does cause a violation.

THE INSERT OPERATION

The **insert** operation provides a list of attribute values for a new tuple that is to be inserted into a relation *R*. Insertion can violate any of the four types of constraints discussed in the previous unit. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain. Key constraints can be violating if a key value in the new tuple *t* already exists in another tuple in the relation (). Entity integrity can be violated if the primary key of the new tuple t is null. Referential integrity can be violated if the value of any foreign key in *t* refers to a tuple that does not exists in the referenced relation. Here are some example to illustrate this discussion.

1. **Insert** < 'Cecilia', 'F', 'Komolafe', null, '1960-04-05', '6357 Adetutu, Lane, Abeokuta, OG', F 28000, null 4> into EMPLOYEE. This insertion violates the entity integrity constraint (null for the primary key SSN), so it is rejected.
2. **Insert** < 'Alice', 'J', 'Zachariah', '999887777', '1960-04-05', '6357 Adetutu Lane, Abeokuta OG', F,28000, '987654321', 4> into EMPLOYEE. This insertion violates the key constraint because another tuple with the same SSN value already exists in the EMPLOYEE relation, and so it is rejected.
3. **Insert** < 'Folorunso', 'O','Olusegun','677678989', '1960-04-05' '6357 Obasanjo Aremu OG', F,28000, '987654321', 7 > into EMPLOYEE. This insertion violates the referential integrity constraint specified on KNO because no DEPARTMENT tuples exists with DNUMBERS = 7.
4. **Insert** < 'Folorunso', 'O', 'Olusegun', '677678989', '1960-04-05', '6357 Obasanjo Lane, Aremu, OG', F, 28000, null 4 > into EMPLOYEE. This insertion satisfies all constraints, so it is acceptable.

If an insertion violates one or more constraints, the default option is to reject the insertion. In this case, it would be useful if the DBMS could explain to the user why the insertion was rejected. Another option is to attempt to correct the reason for rejecting the insertion, but this is typically not used for violations caused by insert; rather, it is used more often in correcting violations for **Delete** and **Update**. The following examples illustrate how this option may be used for insert violations. In operation 1 above, the DBMS could ask the user to provide a value for SSN and could accept the insertion if a valid SSN value was provided. In operation 3, the DBMS could either, ask the user change the value of DNO to some valid values (or set it to null), or it could ask the user to insert a DEPARTMENT tuple with DNUMBER =7 and could accept the insertion only after such an operation was accepted. Notice that in the latter case the insertion can cascade back to the EMPLOYEE relation if the user attempt to insert a tuple for department 7 with a value off MGRSS that does not exist in the EMPLOYEE relation.

THE DELETE OPERATION

The Delete operation can violate only referential integrity; If the foreign keys reference the tuple being deleted from other tuples in the database. To specify deletion, a condition on the attributes of the relation select the tuples (or tuples) to be deleted. Here are some examples.

1. Delete the WORKS-ON tuples with ESS = 999887777 and KPNO = 10. This deletion is acceptable.
2. Delete the employee tuple with SSN = 999887777. This deletion is not acceptable, because tuples in works-on refer to this tuple. Hence, if the tuple is deleted, referential integrity violation will result.

3. Delete the employee tuple with SSN = 333445555. This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the employee, department, works-on, and department relations.

Three options are available if a deletion operation causes a violation,.

The first option is to reject the deletion.

The second opting is to attempt to cascade (or propagate) the deletion by deleting tuples that reference the tuple that is being deleted. For example in operation 2, the DBMS could automatically delete the offending tuples form works-on with ESSN= 999887777.

A third option is a to modify the referencing attribute that cause the valid tuple. Notice that, if a referencing attribute that cause a violation is part of the primary key, it cannot be set to null otherwise, it would violate entity integrity.

Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, DBMS may automatically delete all tuples from works –on and department with ESSN=333445555. Tuples in employee with super SSN = 33445555 and changed to other valid values or to null. Although it may makes sense to delete automatically the works-on and department tuples that refer to an employee tuples, it may not make sense to delete other employee tuples or a department tuple. In general, when a referential integrity constraint is specified, the DBMS should allow the user to specify which of the three options applies in case of a violation of the constraint.

THE UPDATE OPERATION

The update operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation r. It is necessary to specify a condition on the attribute of the relation to select the tuple (or tuples) to be modified. Here are some examples.

1. Update the salary of the Employee tuple with SSN 999887777 to 28000 - Acceptable
2. Update the DNO of the employee tuple with SSN = 999887777 to 1 - Acceptable.
3. Update the DNO of the employee tuple with SSN = 999887777 to 7 - Unacceptable, because it violates referential integrity.
4. Update the SSN of the employee tuple with SSN = 999887777 to 987654321 - Unacceptable, because it violates primary key and referential integrity constraints.

Updating an attribute that is neither a primary key nor a foreign key usually cause no problems. The DBMS need only check to confirm that the new values as of the correct data type and domain. Modifying a primary key value is similar to deleting one tuple and the issues discussed earlier under both **insert** and **delete** comes into play. If a foreign key attribute is modified, the

DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is null).

1.2.4 Basic Relational Algebra Operations

In addition to defining the database structure and constraints, a data model must include a set of operations to manipulate the data. A basic set of relational model operations constitutes the relational algebra. These operations enable the user to specify basic retrieval requests. The result of a retrieval is a new relation, which may have been formed from one or more relations. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a relation algebra expression, whose result will also be a relation.

The relational algebra operations are usually divided into two groups. One group includes set operations from mathematical set theory, these are applicable because each relation is defined to be a set of tuples. Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT. The other group consists of operations developed specifically for relational database; these include *select* , *project*, and *join* among others. The *select* and *project* operations are discussed first, because they are the simplest, then we discuss set operations. Finally, we discuss *join* and other complex operations. The relational database shown in Table 1.3 is used for our examples.

Some common database requests cannot be performed with the basic relational algebra operations, so additional operations are needed to express these requests. Some of these additional operations are described later.

THE SELECT OPERATIONS

The select operation is used to select a subset of the tuple from a relation that satisfy a selection condition. One can consider the select operation to be a filter that keeps only those tuples that satisfy a qualifying condition. For example, to select the employee tuples whose departments is 4, or those whose salary is greater than ₦30,000, we can individually specify each of these two conditions with a select operation as follows:

$_{=4}\ ()$

$_{>30000}()$

In general, the SELECT operation is denoted by

$_{<>}()$

Where the symbol

(sigma) is used to denote the SELECT operator, and the selection condition is a Boolean expression specified on the attributes of relation

. Notice that

is generally a relational algebra expression whose result is a relation; the simplest expression is just the name of database relation. The relation resulting from the select operations has the same attributes as

. The Boolean expression specified in <selection condition> is made up of number of clauses of the form

<attribute name> <comparison op><constant value>

or

<attribute name><comparison op><attribute name>

where <attribute name> is the name of an attribute of < > is normally one of the operators {=<,>,e, ≠} and <constant value> is a constant value from the attribute domain. Clauses can be arbitrarily connected by the Boolean operators AND OR and NOT or form a general selection. For example, to select the tuples for all employees who either work in department 4 and make over ₦25,000 per year, or work in department 5 and make over ₦30,0000 we can specify the following select operation:

Table 1.3 Result of select and project operations.

(a)

FNAME	MINT	LNAME	SSN	BDATE	ADD	SEX	SAL	SUPERSSN	DNO
Akin	G	Adeosun	123456789	1965-01-09	75 Fontren, Housing OG	F	5000	333445555	6
Ayo	T	Adeluola	33344555	1955-12-08	291,Itoko, Abeokuta	M	7000	88866555	5
Daniel	O	Olayinka	453453453	1972-07-31	34,Offmobil Road	M	3400	333445555	5

(b)

LNAME	FNAME	SALARY
Adeosun	Akin	50000

Adeluola	Ayo	70000
Olayinka	Daniel	34000
Daunsi	Jumoke	42000
Adjumo	Kunle	50000
Bamidele	Segun	40000
Oduntan	Segun	45000
Olamide	Kunle	80000

(c)

SEX	SALARY
F	50000
M	70000
M	34000
F	42000
M	50000
M	40000
M	45000
M	80000

(a) $(=4 > 25000)\ (=5 > 30000)()$
(b) $\Pi_{(,)}()$
(c) $\Pi_{()}()$

The result is shown in Table 1.5(a). Notice that the comparison operators in the set {=,<,>, ≠} apply to attribute whose domains are ordered values, such as numeric or date domains. Domains of strings of characters are considered ordered based on the collating sequence of the characters. If the domain of an attribute is a set of unordered values, then only the comparison operations in the set {=""} can be used. An example of an unordered domain is the domain colour = {red, blue, green, white, yellow} where no order is specified among the various colours. Some domains allow additional types of comparison operations; for example, a domain allows additional types of comparison operators. For example, a domain of character strings may allow the comparison operator substring *of*.

In general the result of a SELECT operation can be determined as follows. The selection condition is applied independently to each tuple

in

. this is done by substituting each occurrence of an attribute $_1$ in the selection condition with its value in the tuples $[_1]$. if the condition evaluates to true, then tuple

is **selected**. All the selected tuples appear in the result of the select operation. The Boolean conditions *AND, OR* and *NOT* have their normal interpretation as follows:

(cond1 AND cond2) is true if both *(cond1)* and *(cond2)* are true. Otherwise, it is false

(cond1 OR cond2) is true if either *(cond1)* or *(cond2)* is true or both are true. Otherwise, it is false.

(NOT cond) is true if *cond* is false. Otherwise, it is false.

The SELECT operator is unary, that is, it is applied to a single relation. Moreover, the selection operation is applied to each tuple individually. Hence, selection conditions cannot involve more than one tuple. The **degree** of the tuples in the resulting relation is always less than or equal to the number to tuples in

. That is $|()| \leq ||$ for any condition

. The fraction of tuples selected by a selection condition is referred to as the selectivity of the conditions.

Notice that the select operation is commutative; that is,

$$<_1> (<_2> ()) = <_2> (<_1> ())$$

Hence, a sequence of SELECTs can be allied in any order. In addition, we can always combine a **cascade** of SELECT operations into a single SELECT operation with a conjunctive AND condition, that is,

$$<_1> (<_2> (\dots(< > ())\dots)) = <_1> _2 \dots$$

THE PROJECT OPERATION

If we think of a relation as a table, the SELECT operation selects some of the rows from the table while discarding other rows. The PROJECT operation on the other hand, selects certain attributes of a relation, we use the PROJECT operation to project the relation over these attributes only. For example, to list each employees, first and last name and salary, we use the PROJECT operation as follows:

LNAME, FNAME, SALARY (EMPLOYEE)

The result relation is shown in Table 1.3(b). The general form of the PROJECT operation is

<attribute list > (R)

Where (Pi) is the symbol used to represent the PROJECT operation and < attribute list> is a list of attributes from the attributes of relation R. Again, notice that

is , in general, a relational algebra expression whose result is a relation, which in the simplest case is just the name of a database relation. The result of the PROJECT operation has only the attribute specified in attribute list and in the same order as they appear in the list. Hence, its degree is equal to the number of attribute in attribute list. If the attribute list include only non key attributes of

, duplicate tuple are likely to occur, the PROJECT operation removes any duplicate tuples. So the result of the project operation is a set of tuples and hence a valid relation. This is known as duplicate elimination. For example, consider the following PROJECT operation.

$,\,()$

The result it shown in Table 1.3 (c). Notice that the tuple <F,25000> appears only once in TABLE 1.3(c) even though this combination of values appears twice in the employee relation.

The number of tuples in relation resulting from a PROJECT operations is always less than or equal to the number of tuples in

. If the projection list is a superkey of —that is, it includes some keys of *R* – the resulting relation has the same number of tuples as *R*. Moreover as long as

$$<_1>(<_2>()) = <_1>()$$

$<_2>$ contains the attribute in <list>. Otherwise, the left-hand side is an incorrect expression. It is also noteworthy that commutativity does not hold on PROJECT.

SEQUENCES OF OPERATIONS AND THE RENAME OPERATION

The relations shown in Table 1.3 do not have any names. In general, we may want to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can name the first name, last name, and salary of all employee who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression as follows:

$$_{,,}(_{=5}())$$

Table 1.4(a) shows the result of this relational algebra expression. Alternatively we can explicitly show the sequence of operations, giving a name to each intermediate relation:

$$5 - \;_{=5}()$$

$$\Pi_{,,}(5 -)$$

It is often simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression. We can also use this technique to rename the attributes in the intermediate and result relations. This can be useful in connection with more complex operations such as union and join as we shall see. To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

$$\Pi_{=5}\ ()$$

$$(,,)\Pi_{,}()$$

The two operations above are illustrated in figures 1.6(b). If no renaming is applied the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order. For a project OPERATION with no renaming, the resulting relation has the same attribute names as those in the project list and in the same order in which they appear in the list.

We can also define a RENAME operation which can rename either the relation name, or the attribute name, or both in a manner similar to the way we defined *select*.

$$\Pi_{,,}\ (_{=5}())$$

Table 1.3. Results of a relational algebra expression.

(a)

FNAME	LNAME	SALARY
Akin	Adeosun	50000
Ayo	Adeluola	70000
Daniel	Olayinka	34000

Jumoke	Daunsi	42000

(b)

FNAME	NINT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPER	DN
Akin	G	Adeosun	123456789	1965-01-09	75	F	50000	33445555	6
Ayo	T	Adeluola	3344555	1955-12-08	291,Itoko Abeokuta	M	70000	88866555	5
Daniel	O	Olayinka	453453453	1972-07-31	34 Off Mobil Road	M	34000	33445555	5
Jumoke	O	Daunsi	666884444	1964-09-15	23 Panseke Street	F	342000	333445555	5

(c)

FNAME	LNAME	SALARY
Akin	Adeosun	50000
Ayo	Adeluola	70000
Daniel	Olayinka	34000
Jumoke	Daunsi	42000

The same expression using intermediate relation and renaming of attribute and PROJECT. The general RENAME operation when applied to a relation

of degree

is denoted by $(_{1},_{2},\ldots\ )()$ or $(_{1},_{2},\ldots,)()$. Where the symbol

(rho) is used to denote the rename operator,

is the new relation name, and $_{1}\ _{2}$ are the new attribute names. The first expression rename both the relation and its attributes; the second renames the relation only; and the third renames the attributes only. If the attribute of are $(_{1}, _{2}, \ldots,)$ in that order, then each is renamed as .

SET THEORETIC OPERATIONS

The next group of relational algebra operations is the standard mathematical operations on sets. For example to retrieve security numbers of a employees who either work in *department 5* or directly supervise an employee who works in *department 5*, we can use the union operation as follows:

$5\_ \leftarrow ()$

$1 \leftarrow \prod(5\_)$

$2 \leftarrow \prod(5\_5)$

$\leftarrow 1\ 2$

Table 1.5 Query Result after UNION Operation: RESULT$\prod$ RESULT1 $\leftarrow$RESULT2

RESULT1	SSN		RESULT2	SSN		RESULT	SSN
	123456789			33344555			123456789
	333445555			888665555			333445555
	666884444						666884444
	453453453						453453453
							888665555

The relation *RESULT1* has the social security numbers of all employee who work in *department* 5, whereas *RESULT2* has the social security numbers of all employees who directly supervise an employee who works in *department* 5. The union operation produces the tuples that are in either *result1* or *result2or* both. See Table 1.5.

Several set theoretic operations are used to merge the elements of two sets in various ways, including UNION, INTERSECTION and SET DIFFERENCE. These are binary operators; that is, each is applied to two sets. When these operations are adapted to relational database, the two relations on which any of the above three operations are applied must database, the two relations on which any of the above three operations are applied must have the same types of tuples: this condition is called union compatibility. Two relations $(_{1}, _{2}, \ldots,)$ $(_{1}, _{2}, \ldots,)$ are said to be *union compatible* if they have the same degree

, and if $() = ()$ for $1 \leq \leq$. This means that the row relations have the same number of attributes and that each pair of corresponding attributes have the same domain.

We can define the three operations UNION, INTERSECTION, and DIFFERENCE on two *union-compatible* relations

and

as follows:

UNION: The result of this operation, denoted by $\cup$, is a relation that includes all tuples that are either in

or in

or in both . duplicating tuples are eliminated.

INTERSECTION: The result of this operation denoted by $\cap$, is a relation that includes all tuples that are in both .

SET DIFFERENCE: The result of this operation, denoted by $-$ is a relation that includes all tuples that are in .

We will adopt the convention that the resulting relation has the same attribute name as the first relation

. FIGURE 1.2 illustrates the three operations. The relation student and instructor in Figure 1.2(a) are union compatible, and their tuples represent the names of students and instructors, respectively. The result of the union operation in appear in Figure 1.2 (b) shows names of all student and instruction. The result of the intersection operator Figure 1.2(c) include only those who are both students and instructions. Notice that both union and intersection are commutative operations; that is

$$\cup = \cup \quad \cap = \cap$$

Both union and intersection can be treated as n–ary operations applicable to any number of relations as both are associative operations; that is

$$\cup(\cup) = (\cup)\cup \quad (\cap)\cap = \cap(\cap)$$

a. Two union compatible relations
b. STUDENT $\cup$ INSTRUCTOR
c. STUDENT $\cap$ INSTRUCTOR,
d. STUDENT – INSTRUCTOR
e. INSTRUCTOR – STUDENT

Figure 1.2 Illustrating the set operations UNION, INTERSECTION and DIFFERENCE

The DIFFERENCE operation is not commutative: that is, in general R – S ≠ S – R

Next we discuss the CARTESIAN PRODUCT operation – also known as CROSS PRODUCT or CROSS JOIN – denoted by $\times$, which is also a binary set operation, but the relations on which it is applied do not have to be union compatible. This operation is used to combine tuples form two relations in a combinational fashion. In general, the result of

$(_{1},_{2},\dots, \times (_{1},_{2},\dots,)$

is relation

with $*$ attribute

$(_{1},_{2},\dots,,_{1},_{2},\dots,)$

in that other. The resulting relation

has one tuple for each combination of tuples – one from R and one from

. Hence if

has tuples and

has n tuples, then $\times$ will have $*$ tuples. It is useful when followed by a selection that matches values of attributes coming from the component relations. For example, suppose that we want to retrieve for each female employee a list of the names of her dependents, we can do this as follows:

$$- \leftarrow = ()$$

$$\leftarrow _{,,}(-)$$

$$- \leftarrow \times$$

$$- \leftarrow = (-)$$

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can then SELECT only related tuples form the two relations by specifying an appropriate selection condition, as we did in the preceding example. Because this sequence of CARTESIAN PRODUCT followed by SELECT is used quite commonly to identify and select related tuples from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation.

THE JOIN OPERATION

The JOIN operation denoted by $\bowtie$ is used to combine related tuples from two relations into single tuples. This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations. To illustrate *join*, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each *department* tuple with *employee* tuple whose *SSN* value matches the *MGRSSN* value in the *department* tuple. We do this by using the JOIN operation, and then projecting the result over the necessary attribute, as follows:

A Complete Set Of Relational Algebra Operations

It has been shown that the set of relational algebra operation $\{, -, \times, \Pi$ is a **complete** set. Any of the other relational algebra operations can be expressed as a sequence of operations from this set. For example, the INTERSECTION operation can be expressed by UNION and DIFFERENCE as follows:

$$\cap \; - (\cup) - (-) \cup (-)$$

Although, strictly speaking, INTERSECTION is not required, it is inconvenient to specify this complex expression every time we wish to specify an intersection. As another example, a JOIN operation can be specified as a CARTSIAN PRODUCT followed by a SELECT operation as we discussed:

[illegible] = [illegible] (X)

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT proceeded by RENAME and followed by SELECT and PROJECT operations. Hence, the various JOIN operations are also not strictly necessary for the expressive power of the relational algebra. However, they are very important because they are convenient to use and are commonly applied in database applications. Other operations have been included in the relational algebra for convenience rather than necessity.

THE DIVISION OPERATION.

The DIVISION operation is useful for a special kind of query that sometimes occurs in database applications. An example is "retrieve the names of employee who work on all the project that 'Daniel Olayinka works on". To express this query using the DIVISION operation, the intermediate relation *Daniel_PNOS*:

$\leftarrow{}_{\blacksquare\,\blacksquare}()$

$\_\leftarrow \Pi(- \;\infty_{=})$

Next, create a relation that includes a tuple < , >. Whenever the employee whose social security number is ESSN works on the project whose number is PNO in the intermediate relation SSN – PNOS

$- \;\leftarrow_{,}(-)$

Finally, apply the DIVISION operation to the two relations, which gives the desired employee social security numbers:

$$() \leftarrow - \;+\; -$$

$\leftarrow \Pi_{,}(*)$

ADDITIONAL RELATIONAL OPERATIONS

Some common database requests-which are needed in commercial query languages for relational DBMSs-cannot be performed with the basic relational algebra operations described earlier.

1.3 RELATIONAL DATABASE DESIGN

Here, we consider design issues regarding relational databases. In general, the goal of a relational database design is to generate a set of relation schemes that allow us to store information without unnecessary redundancy, yet allowing us to retrieve information easily. One approach is to design schemes that are in an appropriate normal form. In order to determine whether a relation scheme is in one of the normal forms, we shall need additional information about the "real–world" enterprise that we are modelling with the database. This additional information is given by a collection of constraints called data dependencies.

1.3.1 The Entity – Relationship Model

Designing a successful database application requires a good conceptual model. Generally the term database application refers to a particular database, for example an XYZ bank database that keeps track of customer accounts, and the associated programs that implement database updates corresponding to customers making deposits and withdrawals. These programs often provide user–friendly graphical user interfaces (GUI) utilizing forms and menus. Hence, part of the database application will require the design, implementation and testing of these application programs. This unit discusses ER-model extensively.

USING HIGH-LEVEL CONCEPTUAL DATA MODELS FOR DATABASE DESIGN

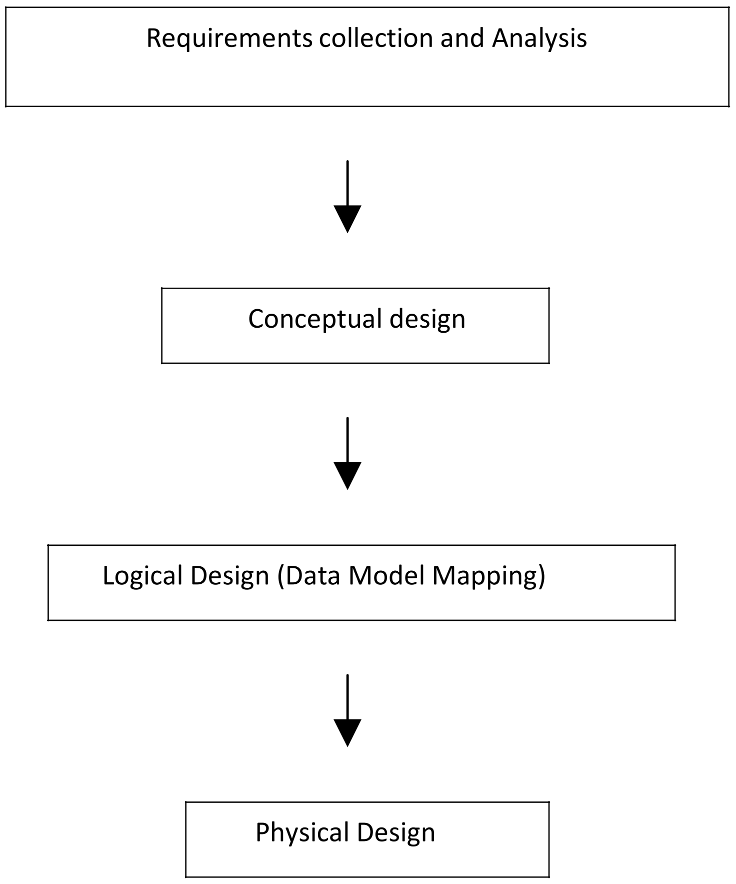

Requirements collection and Analysis
Conceptual design
Logical Design (Data Model Mapping)
Physical Design

Figure 1.9 A simplified description of the database design process.

STEPS INVOLVED IN DATABASE DESIGN

1. Requirements, Collection and Analysis.

During this step, the database designers interview prospective database users to understand and document their data requirements. The result of this step is concisely written as a set of users' requirements. These requirements should be specified as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify the known **functional requirements** of the application. These consist of the user–defined **operations** (or **transactions**) that will be applied to the database and they include both retrievals and updates. In software design, it is common to use data flow diagrams, sequence diagrams, scenarios, and other techniques for specifying functional requirements.

2. Conceptual Schema

Once all the requirements have been collected and analyzed, the next steps is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called conceptual design. The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with non-technical users. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not include conflicts. This approach enables the database designers to concentrate on specifying the properties of the data, without being conceptual with storage details. Consequently, it is easier for them to come up with a good conceptual database design.

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified in the initial schema.

3. Logical Design

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model, such as the relational or the object database model, so the conceptual schema is transformed from the high-level data models into the implementation data model. This step is called **logical design** or **data model mapping**, and its result is a database schema in the implementation data model of the DBMS.

4. Physical Design

Finally, the last step is the physical design phase, during which the internal storage structures access paths, and file organizations for the database files are specified. In parallel with these activities, application programs are designed and implemented as datasets transactions, corresponding to the high – level transaction specifications.

1.3.2 An Illustration Of A Company Database Applications

In this unit, we describe an example database application, called COMPANY, which serves to illustrate the ER model concepts and their use in schema design. The COMPANY database keeps track of a company's employees, departments, and projects. Suppose that, after the requirements, collection and analysis phase, the database designers stated the following description of the "mini world" – the part of the company to be represented in the database:

The company is organized into departments. Each departments has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
A departments controls a number of projects, each of which has a unique name, a unique number, and a single location.
We store each employee's name, social security number, address, salary, sex, and birth date. An employee is assigned to one department but may work on several project, which are not necessarily controlled, by the same department. We keep track of the number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee for insurance purposes. We keep each dependent's first name, sex. birth date, and relationship to the employee.

1.3.3 Entity Types, Entity Sets, Attributes And Keys

The ER model describes data as entities, relationships, and attributes.

ENTITIES AND ATTRIBUTES

The basic objects that ER model represents is an entity, which is a "thing" in the real world with an independent existence. An entity may be an object with a physical existence – a particular person, car, house, or employee, or it may be an object with a conceptual existence – a company, a job, or a university course.

Each entity has attributes – the particular properties that describes it. For example, the employee's name, age, address, salary, and job may describe an employee entity. A particular entity will have a value for each of its attributes. The attribute Values that describe each entity become a major part of the data stored in the database e.g.

EMPLOYEE {Name, Address, Age, and Home Phone}

Composite Versus Simple (Atomic) Attributes

Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the employee entity can be subdivided into Street Address, City, State, and Zip, with the values "8 Housing Estate", "Abeokuta", Ogun", and "110001". Attributes that are not divisible are called *simple* or *atomic*, subdivided into three simple attributes, *Number*, *Street*, and *ApartmentNumber*, as of its constituent simple attributes.

Composite attributes are useful to model situations in which a user sometimes refer to the composite attribute as a unit but at other times refers specially to its components. For example, if there is no need to refer to the individual components of an address (Zip, Street, and so on), then the whole address is designated as a simple attribute.

Single–valued Verses Multivalued Attributes

Most attributes have a single value for a particular entity; such attribute are called single–valued. For example, Age is a single–valued attribute of person. In some cars, or a College Degrees attribute for a person. Cars with one colour have a single value, whereas two–tone cars have two values for colour. Similarly, one person may not have a College Degree, another person may have one, and a third person may have two or more degrees; so different person can have different numbers of values of the College Degrees attributes.

Such attributes are called multivalued. A multivalued attribute may have lower and upper

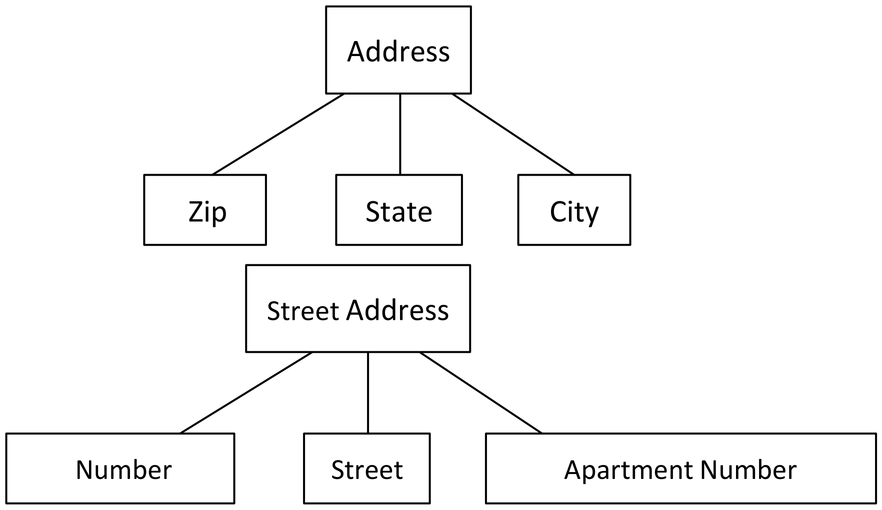


Figure 1.4 A hierarchy of composite attributes; the Street Address component of an Address is further composed of number, s

bounds on the number of values allowed for each individual entity. For example, the colour attribute of *car* may have between one and three values, if we assume that a car can have at most three colours.

Stored Versus Derived Attribute. In some cases, two (or more) attribute values are stated, for example, the *Age* and *Birth Date* attributes of a person. For a particular person entity, the value of *Age* can be determined from the current (today's) date and the value of that person's *Birth Date*. The *Age* attribute is hence called a *derived attribute* and is said to be derivable from the *Birth Date* attribute, which is called a *stored attribute*. Some attributes values can be derived from related entities, for example, an attribute *Number of Employees* of a department entity can be derived by counting the number of employees related to (working for) that department.

Null Values. In some case a particular entity may not have applicable value for an attribute. For example, the Apartment Number attribute of an address applies only to address that are in apartment buildings and not to other types of residences, such as single degrees. For such situations, a special value called null is created. An address of a single–family home would have null for its apartment Number attribute, and a person with no college degree would have null for college degrees.

Complex Attributes. Notice that composite and multivalued attributes can be nested in an arbitrary way. We can represent arbitrary nesting by grouping components of composite attribute between parentheses. Such attributes are called complex attributes. For example, if a person can have more than one residence and each residence can have multiple phones, an attribute Address Phone for a PERSON entity type can be specified as shown in Figure 1.5.

1.3.4 Entity Types, Entity Sets, Keys & Value Sets

{Address Phone] [Phone {Area Code, Phone Number}, Address

(Street Address (Number, Street, Apartment Number) City,

State, Zip)

Figure 1.5 A complex attribute address phone with value and composition.

ENTITY Types and Entity Sets. A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has its own values for each attribute. An entity type defines a collection (or set) of entities that have same attributes. The collection of all entities of a particular entity type in the database at any point in the time is called an entity set; the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a type of entity as well as the current set of all employee entities in the database.

An entity type describes the schemas or intension for a set of entities that have the same structures. The collection of entities of a particular entity type is grouped into an entity set, which is also called the *extension* of the entity type.

Key Attributes of an Entity Type. An important constraint on the entities of an entity type is the key or uniqueness constraint on attributes. An entity type usually has an attribute whose values are distinct for each individual entity in the collection is called a key attribute, and its values can be used to identify each entity uniquely, for example, the Name attributes is a key of the COMPANY entity type, because no two companies are allowed to have same name. For the PERSON entity type a typical key attribute is Identification Number. Sometimes, several attributes together form a key, meaning that the combination of the attribute values must be distinct for each entity. If a set of attributes possesses this property, we can define a composite attribute that becomes a key attribute of the entity type. Notice that a composite key must be minimal, that is, all component attributes must be included in the composite attribute to have the uniqueness property.

Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for every extension of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attributes at the same time. It is not the property of a particular extension, rather, it is a constraint on all extensions of the entity type. This key constraint (and other constraints we discuss later) is derived from the constraints of the mini world that the database represents.

Value Sets (Domains) of Attributes. Each simple attribute of an entity type is associated with a value set (or domain of values), which specifies the set of a values that may be assigned to that attributes of each individual entity. E.g., if the range of ages allowed for employees is between 26 and 65, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer number between 26 and 65. Similarly, we can specify the value set for the name attribute as being the set of strings of alphabetic characters separated by blank characters and so on. Value sets are not displayed in ER diagrams.

Mathematically, an attribute A of entity E whose value set is V can be defined as a function from E to the power set $0 : \quad \rightarrow 0$

1.3.5 Pitfalls In Relational Database Design

Before we begin our discussion of normal forms and data dependencies, let us look at what

can go wrong in a bad database design. Among the undesirable properties that a bad design

may have are:

Repetition of information
Inability to represent certain information
Loss of information.

Below, we discuss these in greater detail using a banking example, with the following two relation schemes:

Branch-scheme = (branch-name, assets, branch-city)

Borrow-scheme =(branch-name, loan-number, customer-name, amount)

Tables 1.6 and *1.7* show an instance of the relations branch (Branch-scheme) and borrow (Borrow-scheme).

Repetition of Information

Consider an alternative design for the bank database in which we replace *branch-scheme* and *borrow-scheme* with the single scheme

lending-scheme = (branch-name, assets, branch-city, loan number, customer-name, amount)

Table 1.6 Sample Branch Relation

branch-name	Assets	branch-city
Downtown	9000000	Brooklyn
Redwood	2100000	Palo Alto
Perryridge	1700000	Horseneck
Mianus	400000	Horseneck

Round hill	8000000	Horseneck
Pownal	300000	Bennington
North Town	3700000	Rye
Brighton	7100000	Brooklyn

Table 1.8 shows an instance of the relation lending(Lending-scheme) produced by taking the natural join of the branch and borrow instances of Tables 1.6 and 1.7. A tuple

in the lending relation has the following intuitive meaning:

Table 1.17 Sample Borrow Relation

branch-name	loan-number	customer-name	amount
Downtown	17	Jones	1000
Redwood	23	Smith	2000
Perryridge	15	Hayes	1500
Downtown	14	Jackson	1500
Mianus	93	Curry	500
Round hill	11	Turner	900
Pownal	29	Williams	1200
North town	16	Adams	1300
Downtown	18	Johnson	2000
Perryridge	25	Glenn	2500
Brighton	10	Brooks	2200

t (assets) is the asset figure for the branch named (branch-name)
t (branch-city) is the city in which the branch named t (branch-name) is located

t (loan-number) is the number assigned to a loan made by the branch named t(branch-name) to the customer named t (customer-name)
t (amount) is the amount of the loan whose number is t(loan-number])

Suppose we wish to add a new loan to our database. Assume the loan is made by the Perryridge branch to Turner in the amount of ₦1,500. Let the number of the loan be 31. In our original design, we would add the tuple

[Perryridge, 31, Turner, 1500]

to the borrow relation. Under the alternative design, we need a tuple with values on all the attributes of lending-scheme. Thus, we must repeat the asset and city data for the Perryridge branch and add a tuple

[Perryridge, 17000000, Horseneck, 31, Turner, 1500]

to the lending relation. In general, the asset and city data for a branch must appear once for each loan made by that branch.

The repetition of information required by the use of our alternative design is undesirable. Repeating information wastes space. Furthermore, the repetition of information complicates updating the database.

Table 1.8 branch ⋈ borrow

branch-name	assets	branch-city	loan-number	customer-name	amount
Downtown	9000000	Brooklyn	17	Jones	1000
Redwood	2100000	Palo alto	23	Smith	2000
Perryridge	1700000	Horseneck	15	Hayes	1500
Downtown	9000000	Brooklyn	14	Jackson	1500
Mianus	400000	Horseneck	93	Curry	500
Round hill	8000000	Horseneck	11	Turner	900
Pownal	30000	Bennington	29	Williams	1200
North town	3700000	Rye	16	Adams	1300
Downtown	9000000	Brooklyn	18	Johnson	2000

Perryridge	1700000	Horseneck	25	Glenn	2500
Brighton	7100000	Brooklyn	10	Brooks	2200

Suppose, for example, that the Perryridge branch moves from Horseneck to Newtown. Under our original design, one tuple of the branch relation needs to be changed. Under our alternative design, many tuples of the lending relation needs to be changed. Thus, updates are more costly under the alternative design than under the original design. When we perform the update in the alternative database, we must ensure that every tuple pertaining to the Perryridge branch is updated, lest our database shows two cities for the Perryridge branch.

The above observation is central to understanding why the alternative design is bad. We know that a bank branch is located in exactly one city. On the other hand, we know that a branch may make many loans. These are independent facts and, as we have seen, these facts are best represented in separate relations. We shall see that the fact that for every branch there is exactly one city can be stated formally, and this fact can be used to improve our database design.

Representation of Information

Let us consider a second alternative to our original design. This time, we attempt to combine the following two schemes:

Borrow-scheme = [branch-name, loan-number, amount]

Deposit-scheme = [branch-name, account-number, customer-name, balance]

To create a new scheme:

BD-scheme = [branch-name, loan-number, amount, account-number, balance, customer-name]

Let

be a relation on scheme –. A tuple

in this relation has the following intuitive meaning:

[loan-number] is the number assigned to a loan made by the branch named [branch-name]to the customer named

[customer-name]

[amount] is the amount of the loan whose number is t[loan-number]

[amount-number] is the number assigned to an account at the branch named

[branch-name] and belonging to the customer named t[customer-name]
t[balance] is the balance in the account whose number is t[account-number].

Consider a customer who has an account at the Perryridge branch but does not have a loan from that branch. We cannot represent this information directly by a tuple in the BD relation since tuples in k the BD relation require values for loan-number and amount.

One solution to this problem is to introduce null k values, as we did above to handle updates through view. Recall, however, that null values are difficult to deal with. If we are not willing to deal with null values, each branch must force every depositor to take a loan at that branch and every borrower to open an account at that branch. Presumably, the teller would inform the customer of this by saying, “we are sorry, but the computer forces us to do this”. Clearly, this excuse is untrue, since under our original Database design, we could have depositors with no loan and borrowers with no account, and we could do so without resorting to the use of null values.

Loss of Information

The above examples of bad designs suggest that we should decompose relation schemes with many attributes into several schemes with fewer attributes. Careless decomposition, however, may lead to another form of bad design.

Consider yet another alternative design in which Borrow-scheme is decomposed into two schemes, *amt-scheme* and *loan-scheme*, as follows:

Amt-scheme = (amount, customer-name)

Loan-scheme =(branch-name, loan-number, amount)

Using the borrow relation of *Table 1.9* we construct our new relations *Amt(Amt-scheme)* and *loan (Loan-scheme)* as follows:

$$- \Pi_{-} ()$$

$= \Pi_{-,-,}()$

We show the resulting

and

relations in *Table 1.9*

There are cases in which we need to reconstruct the borrow relation. For example, suppose that we wish to find those branches from which Jones has a loan. None of the relations in our alternative database contains this data. We need to reconstruct the borrow relation. It appears that we can do this by writing:

$\bowtie$

Table 1.10 shows the result of computing $\bowtie$. When we compare this relation and the borrow relation with which we started (Table 1.7), we notice some difference.

Although every tuple that appears in

appears in $\bowtie$, there are tuples in $\bowtie$ that are not in borrow. In our example, $\bowtie$ has the following additional tuples:

(Downtown, 14 Hayes, 1500)

(Perryridge, 15 Jackson, 1500)

(Redwood, 23, Johnson, 2000)

(Downtown, 18, smith, 2000)

Table 1.9 The relations amt and loan

branch-name	loan-number	amount
Downtown	17	1000
Redwood	23	2000
Perryridge	15	1500
Downtown	14	1500
Mianus	93	500
Round hill	11	900
Pownal	29	1200
North Town	16	1300
Downtown	18	2000
Perryridge	25	2500
Brighton	10	2200

amount	customer-name
1000	Jones
2000	Smith
1500	Hayes
1500	Jackson
500	Curry
900	Turner
1200	Williams
1300	Adams
2000	Johnson
2500	Glenn
2200	Brooks

Consider the query, "find those branches from which Hayes has a loan." If we look back at Table 1.7 we see that Hayes has only one loan, and that loan is from the Perryridge branch. However, when we apply the expression

$$\Pi_{[illegible]} (\sigma_{[illegible]} (\bowtie))$$

we obtain two branch names: Perryridge and Downtown.

Let us examine this example more closely. If several loans happen to be the same amount, we cannot tell which customer has which

. Thus, when we join

and

we obtain not only the tuples we had originally in borrow, but also several additional tuples. Although we have more tuples in $\bowtie$, we actually have less information.

We are no longer able, in general, to represent in the database which customers are borrowers from which branch. Because of this loss of information, we call the decomposition of borrow-scheme into amt-scheme and loan-scheme a lossy decomposition, or a lossy-join decomposition. A decomposition that is not a lossy-join decomposition is referred to as a lossless-join decomposition. It should be clear from our example that a lossy-join decomposition is, in general, a bad database design.

Table 1.10 The relation $\bowtie$

branch-name	**loan-number**	**customer-name**	**Amount**
Downtown	17	Jones	1000
Redwood	23	Smith	2000
Perryridge	15	Hayes	1500
Downtown	14	Jackson	1500
Mianus	93	Curry	500
Round Hill	11	Turner	900
Pownal	29	Williams	1200
North town	16	Adams	1300
Downtown	18	Johnson	2000
Perryridge	25	Glenn	2500
Brighton	10	Brooks	2200
Downtown	14	Hayes	1500
Perryridge	15	Jackson	1500
Redwood	23	Johnson	2000
Downtown	18	Smith	2000

Let us examine the decomposition more closely to see why it is lossy. There is one attribute in common between loan-scheme and amt-scheme:

$- \cap - = \{\}$

The only way we can represent a relationship between branch-name and customer-name is through amount. This is not adequate because many customers may happen to have loans in the same amount, yet they do not necessarily have these loan from the same branches. Similarly, many customers may happen to have loan from the same branch, yet the amounts of their loan may be unrelated to each other.

Contrast this with *Lending-scheme*, which we discussed earlier. We argued that a better design would result if we decompose *Lending-scheme* into *Borrow-scheme* and *Branch-scheme*.

$- \cap - = \{-\}$

Thus, the only way we can represent a relationship between for example, *customer-name* and assets is through *branch-name*. The difference between this example and the example above is that the assets of a branch are the same regardless of the customer to which we are referring, while the lending branch associated with a certain loan amount does depend on the customer to which we are referring. For a given *branch-name,* there is exactly one assets value and exactly one *branch-city*, while a similar statement cannot be made for amount. Note that we have never stated formally in our design that a branch has a unique assets value and branch-city value. We know this from our intuition of the banking enterprise. We represented this earlier by the notion of keys. In the next unit, we shall see how to express these facts formally. The notion of lossless joins is central to much of relational database design. Therefore, we restate the above examples below more concisely and more formally: Let

be a relation scheme, a set of relation schemes $\{_1, _2, \dots, \}$ is a decomposition of U if:

$$\bigcup_{=1} =$$

That is, $\{_1, _2, \dots, \}$ is a decomposition of

if every attribute in

appears in at least one , for $1 \leq \ \leq$. Let

be a relation on scheme

, and let $= \Pi()$ for $1 \leq \ \leq$. That is $\{_1, _2, \dots, \}$ is the database that results from decomposing

into $\{_1, _2, \dots, \}$. It is always the case that:

$$\subseteq \bowtie_{=1}$$

To see this, consider a tuple

in relation

. When we computer $_{1, 2, \ldots,}$, the tuple gives rise to one tuple , in each , $1 \leq \quad \leq$. These

tuples combine to regenerate

when we compute $\bowtie_{=1}$ = 1 . The details are left as an exercise to the reader. Therefore, every tuple in

appears in $\bowtie_{=1}$.

In general, $\neq \bowtie_{=1}$. To illustrate this, consider our earlier example in which:

$= 2$

$= -$

$_1 = -$

$_2 = -$

$= \quad 1.7$

$_1\ _2 =$ Table 1.9

$_1 \bowtie\ _2 = \quad 1.10$

Note that the relations in Tables 1.7 and 1.10 are not the same.

In order to have a lossless-join decomposition, we need to impose some constraints on the set of possible relations. We found that decomposing *Lending-scheme* into *Borrow-scheme* and *Branch-scheme* is lossless because of the rule that “for every branch-name, there is a unique asset value and a unique branch-city value”. We say that a relation is legal if it satisfies all rules, or constraints, that we impose on our database.

Let

represent a set of constraints on the database. A decomposition $\{_1, _2, \ldots,\}$ of a relation scheme

is a lossless-join decomposition for

if for all relations u on scheme

that are legal under

:

$$- \bowtie_{\blacksquare 1} \Pi ()$$

We shall show how to test whether decomposition is a lossless-join decomposition in the next unit. A major part of this subunit is concerned with the question of how to specify constraints on the database and how to obtain lossless-join decomposition that avoid the pitfalls represented by the examples of bad database designs that we have seen in this unit.

1.4 NORMALIZATION

Normalization Using Functional Dependencies

In this unit, we focus on a particular kind of constraint called a functional dependency. Functional dependencies are important because they lead to several highly desirable normal forms for relational databases. The notion of a functional dependency is a generalization of the notion of key, as discussed above. In general, it may not be a simple manner to determine keys from a given set of functional dependencies. In order to find key, we need to study the properties of functional dependencies.

First, we need to determine all the functional dependencies that hold. Second, once we have chosen a particular decomposition of a relation scheme, we need to determine those functional dependencies that hold on the decomposed schemes. We need general techniques if we are to deal with large, real-world databases.

Furthermore, we need to guard against decompositions that have anomalous behaviour similar to those discussed earlier.

Functional Dependencies

Functional dependencies are a constraint of the set of legal relations. They allow us to express facts about the enterprise that we are modelling with our database.

Earlier, we defined the notion of a superkey as follows: Let

be a relation scheme. A subset

of

is a superkey of

if, in any legal relation $()$, for all pair $_1$ and $_2$ of tuples in such that :

$_1 \neq {}_2, {}_1() \neq {}_2().$

That is no two tuples in any legal relation $()$ may have the same value on attribute set

. The notion of functional dependency generalizes the notion of superkey. Let $\subseteq$ and $\subseteq$. The *functional dependency*

$\rightarrow$

Holds on

if in any legal relation $()$, for all pairs of tuples $_1$ and $_2$ in

such that

$_1() = {}_2()$, it is also the case that $_1() = {}_2()$

Using the functional dependency notation, we say that

is a superkey of

if $\rightarrow$. That is. K is a superkey if whenever $_1[\,] = {}_2[\,], {}_1[\,] = {}_2[\,]\ ({}_{\ldots 1} = {}_2).$

Functional dependencies allow us to express constraints that cannot be expressed using superkey. Consider the scheme lending-scheme that we used in one of our alternative designs above, we would not expect the attribute branch-name to be a superkey because we know that a branch may have many loans to many customers. However, we do expect the functional dependency:

$$- \rightarrow -$$

to hold, since we know that a branch may be located in exactly one city.

We shall use functional dependencies in two ways:

1. To test relations to see if they are legal under a given set of functional dependencies. If a relation is legal under a set of functional dependencies, we say that

satisfies

.

2. To specify constraints on the set of legal relations. We shall thus concern ourselves only with relations that satisfy a given set of functional dependencies.

Let us consider the relation

of *Table 1.11* and see which functional dependencies are satisfied. Observe that $\rightarrow$ is satisfied. There are two tuples that have an

value of $_1$. These tuples have the same

value, namely, $_1$. Similarly, the two tuples with an

value of $_2$ have the same

value, $_2$. There are no other pairs of distinct tuples that have the same

value. The functional dependency $\rightarrow$ is not satisfied, however. To see this, consider the tuples $_1 = (_{2}, _{3}, _{2}, _{3})$ $_2 = (_{3}, _{3}, _{2}, _{3})$. These two tuples have the same

value $_2$ but they have different

value, $_2$ and $_3$, respectively. Thus, we have found a pair of tuples $_1$ and $_2$ such that $_1() = {}_2()$ but $_1() \neq {}_2()$.

Table 1.11. sample relation

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2

a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Many other functional dependencies are satisfied by r, including, for example, the functional dependency $\rightarrow$. Note that we use "AB" as a shorthand for {AB}, to confirm with standard practice. The verification that $\rightarrow$ is satisfied is left as an exercise. Some functional dependencies are said to be trivial, because they are satisfied by all relations. For example, *A* is satisfied by all relations involving attribute *A*.

Reading the definition of functional dependency literally, we see that for all tuples $_1$ and $_2$ such that $_1[]$ = $_2[]$, it is also the case that $_1[]$ = $_2[]$. In general, a functional dependency of the form $\rightarrow$ is trivial if $\subseteq$.

Let us return to the banking example. If we consider the customer relation as shown in *Table 1.12*, we see that $\rightarrow$ – y is satisfied. However, we do believe that, in the real world, two cites can have streets with the same name. Thus, it is possible at some time to have an instance of the

relation in which $\rightarrow$ – is not satisfied. Thus, we would not include $\rightarrow$ – in the set of functional dependencies that hold on – .

In the borrow relation of *Table 1.7*, we see that – $\rightarrow$ is satisfied. Unlike the case of customer-city and street, we do not believe that the real-world enterprise that we are modelling allows loans to have several balances. Therefore, we want to require that – $\rightarrow$ be satisfied by the borrow relation at all times. In order words, we impose the constraint that – $\rightarrow$ hold on borrow-scheme.

In what follows, we assume that when we design a relational database, we first list those functional dependencies that must always hold. In the banking example, our list of dependencies includes the following

On Branch-scheme

– $\rightarrow$ –

– $\rightarrow$

On Customer-scheme

– $\rightarrow$ –

– →

On borrow-scheme

– →

– → –

On deposit-scheme

– →

– → –

Table 1.12 Customer Relation

customer-name	street	customer-city
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto

Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

Theory of functional Dependencies

A large body of formal theory has been developed for functional dependencies. We shall focus on those algorithms that are most useful in applying this theory to the design BCNF and 3NF relational databases.

In general, if we are given a set of functional dependencies, these functional dependencies may imply that other functional dependencies hold also, For example, suppose we are given a relation scheme $= (,,,,,)$ and a set of functional dependencies:

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

It is not sufficient to consider the given set of functional dependencies. Rather, we need to consider all functional dependencies that hold. We shall see that, given a set

of functional dependencies, we can prove that certain other functional dependencies hold. We say that such functional dependencies are logically implied by

.

If we are given the set of functional dependencies that appear above, the functional dependency

$\rightarrow$

is logically implied. That is, we can show that whenever our given set of functional dependencies hold, $\rightarrow$ must hold also. Suppose that $_1$ $_2$ are tuples such that

$$_1[] = _2[]$$

then, since we are given that $\rightarrow$, it follows from the definition of functional dependency that

$${}_1[\ ] = {}_2[\ ]$$

then, since we are given that $\rightarrow$, it follows from the definition of functional dependency that

$${}_1[\ ] = {}_2[\ ]$$

Therefore, we have shown that whenever ${}_1\ {}_2$ are tuples such that ${}_1[\ ] = {}_2[\ ]$, it must be that ${}_1[\ ] = {}_2[\ ]$ but that is exactly the definition of $\rightarrow$.

Let

be a set of functional dependencies. We denote the *closure* of

by $^+$. The *closure* of

is the set of all functional dependencies logically implied by

. Given F, we can compute $^+$ directly from the formal definition of functional dependency. If

is large, this process would be lengthy and difficult. Such a computation of $^+$ requires arguments of the type given above to show that $\rightarrow$ is in the *closure* of our example set of dependencies. We now present a simpler techniques for reasoning about functional dependencies.

The first technique we present is based on three *axioms* or rules of inference for functional dependencies. By applying these rules repeatedly, we can find all of $^+$ given

.

1. **Reflexivity rule.** If

is a set of attributes and $\subseteq$, then $\rightarrow$ holds

2 **Augmentation rule.** If $\rightarrow$ holds and

is a set of attributes, then $\rightarrow$ holds

3 **Transitivity rule.** If $\rightarrow$ holds, and $\rightarrow$ holds, then $\rightarrow$ holds.

4. **Union rule.** If $\rightarrow$ holds and $\rightarrow$ holds, then $\rightarrow$ holds

5 **Decomposition rule.** If $\rightarrow$ holds, then $\rightarrow$ holds and $\rightarrow$ holds

6 **Pseudo transitivity rule.** If $\rightarrow$ holds and $\rightarrow$ holds, then $\rightarrow$ holds

These rules are said to be *sound* because they do not generate any incorrect functional dependencies. The rules are complete because given a set

of functional dependencies, they allow us to find all of $^{+}$. This collection of rules is called *Armstrong's axioms* in honour of the person who first proposed them.

Although Armstrong's axioms are complete, it is tiresome to use them directly for the computation of $^{+}$To simplify matters further, we list some additional rules. It is possible to use Armstrong's axioms to prove that these rules are correct .

Let us apply our rules to the example we presented earlier of scheme $= (,,,,,)$ and the set

of functional dependencies $\{\rightarrow, \rightarrow, \rightarrow, \rightarrow, \rightarrow\}$. We list some members of $^{+}$ below:

$\rightarrow$. Since $\rightarrow$ and $\rightarrow$ holds, we apply rule 3 (transitivity). Observed that it was much easier to use Armstrong's axioms to show that $\rightarrow$ holds than it was to argue directly from the definitions as we did earlier.

$\rightarrow$. Since $\rightarrow$ and $\rightarrow$, the union rule (rule 4) implies that $\rightarrow$

$\rightarrow$. We need several steps to show $\rightarrow$. First, observe that $\rightarrow$ holds, using the augmentation rule (rule 2) we see that $\rightarrow$ We are given that $\rightarrow$, so by the transitivity rule (rule 3) $\rightarrow$ holds.

Frequently, we are interested in finding those attributes functionally determined by a given set of attributes. One way to find this set is to compute the closure of the given set of functional dependencies by an exhaustive application of our inference rules. Fortunately, there is a more efficient approach.

Let

be a set of attributes. We call the set of all attributes functionally determined by

under a set

of functional dependencies the closure of

under

and denote it by $^{+}$. *Figure 1.6* shows an algorithm, written in pseudo Pascal, to compute $^{+}$. The input is a set F of functional dependencies and the set of attributes. The output is store in the variable result.

To illustrate how the algorithm of *Figure 1.6* works, let us use the algorithm to compute $()^+$ with the functional dependencies defined above. We start with $=$. The first time we execute the **while** loop to test each functional dependency we find that:

$\rightarrow$ Causes us to include

in

. To see this, observe that $\rightarrow$ is in

, $\subseteq$ (which is

). So $:= \cup$.

$\rightarrow$ causes

to become ABCG

$\rightarrow$ causes

to become ABCGH

$\rightarrow$ causes

to become ABCGHI

The second time we execute the **while** loop, no new attributes are added to result and the algorithm terminates.

Let us see why the algorithm of *Figure 1.6* is correct.

The first step is correct since $\rightarrow$ always holds (by the reflexivity rule). We claim that for any subset

```
:= ;
while (change to
```

of

, it is the case that $\rightarrow$. Since we start the **while** loop with $\rightarrow$ being true, the only way we can add Z to result is if $\subseteq$ and $\rightarrow$. But

FIGURE 1.6 Algorithm to compute X^+, the closure of X under F

then $\rightarrow$ by the reflexivity rule, so $\rightarrow$ by transitivity. Another application of transitivity shows that X $\rightarrow$ Z (using $\rightarrow$ and $\rightarrow$). The union rule implies that $\rightarrow$ $\cup$, so

functionally determines any new result generated in the **while** loop. Thus any attribute returned by the algorithm is in $^+$.

It is easy to see that the algorithm finds all of $^+$. If there is an attribute in $^+$ not yet in result, then there must be a functional dependency $\rightarrow$ for which $\rightarrow$ result and at least one attribute in F is not in result. It turns out that in the worst case this algorithm may take exponential time in the size of

. There is a faster [though slightly more complex] algorithm that runs in time linear in the size of

.

Desirable Properties of Decomposition

A given set of functional dependencies can be used in designing a relational database in which most of the undesirable properties discussed in the unit on pitfalls in relational database design do not occur. In designing such systems, it may become necessary to decompose a relation to a number of smaller relations.

Here, we shall illustrate our concepts by considering the *lending-scheme* scheme again.

Lending-scheme =(branch-name, assets, branch-city, loan number, customer-name, amount)

The set

of functional dependencies that we required to hold on lending-scheme are:

$- \rightarrow$

$- \rightarrow -$

$- \rightarrow$

$- \rightarrow -$

As discussed in earlier, the $-$ is an example of a bad database design. Assume that we decompose it to the following five relations:

$_1 = [-,]$

$_2 = [-,-]$

$_3 = [-,]$

$_4 = [-,-]$

$_5 = [-,-]$

We claim that this decomposition has several desirable properties, which we discuss below.

Lossless-join decomposition

We have argued that it is crucial when decomposing a relation into a number of smaller relations that the decomposition be lossless. We claim that the above decomposition is indeed lossless. To demonstrate this, we must first present a criterion for determining whether a decomposition is lossy.

Let R be a relation scheme and F a set of functional dependencies on R. let $_1$ and $_2$ form a decomposition of

. This decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies are in $^+$.

$_1 \cap _2 \rightarrow _1$

$_1 \cap _2 \rightarrow _2$

We now show that our decompostition is a lossless-join decomposition by showing a sequence of steps that generate the decomposition. We begin by decomposing lending-scheme into two schemes:

$_1 = (-,)$

$_1 = (-,-,-,-,)$

Since $_1 \cap _1 = \{-\}$ and $- \rightarrow$, it follows (by augmentation) that

$_1 \cap = -$

$- \quad - \rightarrow$

$- \rightarrow = _1$

So our initial decomposition is a lossless-join decomposition

Next, we decompose $_1$ into:

$_2 = (-, -)$

$_2 = (-, -, -,)$

This step result in a lossless-join decomposition since $- \rightarrow -$. We then decompose $_2$ into:

$_3 = (-,)$

$_3 = (-, -, -)$

Then $_3$ is decomposed into:

$_4 = (-, -)$

$_5 = (-, -)$

It is simple, using the set

of functional dependencies on $-$, to verify that both of these final steps generate a lossless-join decomposition.

Dependency Preservation

There is another goal in relational database design to be considered: *dependency preservation.* When an update is made to the database, the system should be able to check that the update will not create an illegal relation; that is one that does not satisfy all of the given functional dependencies. In order to check updates efficiently, it is desirable to design relational database schemes that allow update validation without the computation of joins.

In order to determine whether we need to compute joins, we need to determine what functional dependencies may be tested by checking only one relation. Let

be a set of functional dependencies on a scheme

and let $_1, _2, \ldots,$ be a decomposition of

. The restriction of

to is the set of all functional dependencies in $^+$ that include only attributes of . Since all functional dependencies in a restriction involve attributes of only one relation scheme, it is possible to test satisfaction of such a dependency by checking only one relation.

The set of restrictions $_{1, 2, \ldots,}$ is the set of dependencies that can be checked efficiently. We now must ask whether testing only the restrictions is sufficient. Let

be the union of all the restrictions, $\cup_{1}$. is a set of functional dependencies on scheme

, but, in general, $\neq$. However, even if $\neq$, it may be that $^+ = ^+$. if this is true, then every dependency in

is logically implied by

and if we verify that

is satisfied, we have verified that

is satisfied. We say that a decomposition having this property is a *dependency-preserving* decomposition. *Figure 1.7* shows an algorithm for testing dependency preservation. The input is a set D $=$ $\{_{1, 2, \ldots,}\}$ of decomposed relation schemes, and a set F of functional dependencies.

We can now show that our decomposition *Lending-scheme* is dependency preserving. To see this, we consider each member of the set

of functional dependencies that we require to hold on *Lending-scheme* and show that each one can be tested in a at least one relation in the decomposition.

- $- \to$ can be tested using $_1 = (- ,)$
- $- \to -$ can be tested using $_3 = (- , -)$
- $- \to$ can be tested using $_2 = (- ,)$
- $- \to -$

can be tested using

$_7 = (- , -)$

As the above example shows, it is often easier not to apply the algorithm of *Figure 1.7* to test dependency preservation, since the first step, computation of $^+$, takes exponential time.

Repetition of Information

The decomposition of *lending-scheme* does not suffer from the problem of repletion of information. In *lending-scheme*, it was necessary to repeat the city and assets of a branch for each loan. The decomposition separates branch and loan data into distinct relations, thereby eliminating this redundancy. Similarly, observe that if a single loan is made to several customer, we must repeat the amount of the loan once for each customer (as well as the city and assets of the branch). In the decomposition, the relation on scheme $_5$ contains the *loan-number, customer-name* relationship, and no other scheme does. Therefore, we have one tuple for each customer for a loan only in the relation on $_5$. in the other relations involving *loan-number* (those on schemes $_3$, and $_4$), only one tuple per loan need appear.

```
compute $^+$
for each scheme  in
do
begin
        = the restriction of $^+$ to ;
```

Figure 1.7 Testing dependency preservation.

Clearly, the lack of redundancy exhibited by our decomposition is desirable. The degrees to which can achieve this lack of redundancy is represented by several *normal forms*, which we shall discuss in the remainder of this cheaper.

Boyce-Codd Normal Form

We now examine how we can use a given set of functional dependencies to design a relational database. Using functional dependencies, we can define several normal forms which represent “good” database designs. There are large number of normal forms. Of these, one of the more desirable normal forms we can obtain is Boyce-Codd normal form (BCNF).

A relation scheme

is in BCNF if for all functional dependencies that hold on of the form $\rightarrow$, where $\subseteq$ and , at least one of the following holds:

$\rightarrow$ is a trivial functional dependency (that is $\rightarrow$)

is a superkey for scheme

A database design is in BCNF if each member of the set of relation schemes comprising the design is in BCNF

In our banking example, the relation scheme *Customer-scheme* is in BCNF. To see this, note that a candidate key for the relation is *Customer-name*. The only nontrivial functional dependencies that hold on all legal instances of the customer relation have *Customer-name* on the left side of the arrow. Since *Customer-name* is a candidate key, functional dependencies with *Customer-name* on the left side do not violate the definition of BCNF.

The scheme, *Borrow-scheme*, however, is not in BCNF. First, note that loan-number is not a superkey for *Borrow-scheme*. Although

$- \rightarrow -$

Happens to hold on the borrow relation of *Table 1.7,* we could have a pair of tuples representing a single loan made to two people:

[Downtown, 44, Mr. Bilk, 1000]

[Downtown, 44, Mrs. Bill, 1000]

Because we did not list functional dependencies that rule out the above case, loan-number is not a candidate key. However, the functional dependency $- \rightarrow$ is nontrivial. Therefore, borrow-scheme does not satisfy the definition of BCNF.

We claim that borrow-scheme is not in a desirable form since it suffers from the repetition of information problem described earlier. To illustrates this, observe that if there are several *Customer-name* associated with a loan, in a relation on borrow-scheme, then we are forced to repeat the branch name and the amount once for each customer. We can eliminate this redundancy by redesigning our database so that all schemes are in BCNF. One approach to this problem is to take the existing non BCNF design as a starting point and decompose those schemes that are not in BCNF. Consider the decomposition of *Borrow-scheme* into two schemes:

Loan-info-scheme = (branch-name, loan-number, amount)

Loan-customer-scheme= (loan-number, customer-name)

To determine whether these schemes are in BCNF, we need to determined what functional dependencies apply to the schemes. In this examples, it is easy to see that

$- \rightarrow$

$- \rightarrow -$

Apply to loan-info-scheme, and that only trivial functional dependencies apply to loan-customer-scheme. Although loan-number is not a superkey for borrow-scheme, it is a candidate key for loan-info-scheme. Thus, both schemes of our decomposition are in BCNF.

It is now possible to avoid redundancy in the case where there are several customers associated with a loan. There is exactly one tuple for each loan in the relation on loan-info-scheme, and one tuple for each customer of each loan in the relation on loan-customer-scheme. Thus, we do not have to repeat the branch name and the amount once for each customer associated with a loan. The decomposition is a lossless-join decomposition. We shall see how to determine this in the next unit.

In order for the entire design for the bank example to be in BCNF, we must decompose Deposit-scheme in a manner similar to OUR decomposition of borrow-scheme. When we do this, we obtain two schemes:

Account-info-scheme = (branch-name, account-number, balance)

Account-customer-scheme = (account-number, customer)

We are now able to state a general method to generate a collection of BCNF schemes. Let

be a given relation scheme and let

be given set of functional dependencies on

. Recall that

is in BCNF if for all functional dependencies $\rightarrow$ in $^+$ either $\rightarrow$ is trivial or is a superkey for $(, \rightarrow \quad ^{+})$. If

```
    := {};
    := ;
    compute +;
    while (not done) do
    if (there is a scheme , in result that is not in BCNF)
```

is not in BCNF, we can decompose

into

FIGURE 1.8 BCNF decomposition algorithm.

a collection of BCNF schemes $_{1}, _{2}, \ldots,$ using the algorithm of *Figure 1.8* which generates not only a BCNF decomposition but also a lossless-join decomposition. To see why our algorithm generates only lossless-join decomposition, notice

that when we replace a scheme with $-$ and $(,)$, the dependency $\rightarrow$ holds and $(-) \cup () = .$

Let us apply the BCNF decomposition algorithm to the *Lending-scheme* scheme that we used earlier as an example of a bad database design.

Lending-scheme = (branch-name, assets, branch-city, loan-number, customer-name, amount).

The set F of functional dependences that we require to hold on lending-scheme are:

$- \rightarrow$

$- \rightarrow -$

$- \rightarrow$

$- \rightarrow -$

We can apply the algorithm of *Figure 1.8* as follows:-

The functional dependency:

$$- \rightarrow$$

holds on *Lending-scheme*, but *branch-name* is not a superkey. Thus, *Lending-scheme* is not in BCNF. We replace *Lending-scheme* by

$_1 = (-,)$

$_2 = (-, -, -,$

$-,)$

$_1$ is in BCNF since branch-name is a key for $_1$

The functional dependency:

$$- \rightarrow -$$

Holds on $_2$, but branch-name is not a key for $_2$. We replace $_2$ by

$_3 = (-, -)$

$_4 = (-, -, -,)$

$_3$ is in BCNF

The functional dependency:

$$- \rightarrow$$

Causes us to decompose $_4$ into

$_5 = (-,)$

$_6 = (-, -, -)$

$_5$ is in BCNF

The functional dependency:

$$- \rightarrow -$$

Causes us to decompose $_6$ into

$_7 = (-, -)$

$_8 = (-, -)$

$_7$ and $_8$ are in BCNF.

Thus, the decomposition of lending-scheme results in the five relation scheme $_1, _2, _5, _7$, and $_8$ each of which is in BCNF. These relation schemes are the same as those used in above (except for naming for the schemes). We have demonstrated in that unit that the resulting decomposition is both a lossless-join decomposition and a dependency-preserving decomposition.

Not every BCNF decomposition is dependency-preserving.

To illustrate this, consider the BD-scheme defined in earlier, that we used as an example of a bad database design.

BD-scheme = (branch-name, loan-number, amount, account-number, balance, customer-name)

The set

of functional dependencies that we require to hold on the *BD-scheme* is

$- \rightarrow$

$- \rightarrow -$

$- \rightarrow$

$- \rightarrow -$

If we apply the algorithm of *Figure 1.8*, we may obtain the following BCNF decomposition

$_1 = (-,)$

$_2 = (-, -)$

$_3 = (-,)$

$_4 = (-, -, -)$

Using our relation in scheme $_1$, we can esnsure that the functional dependency

$\rightarrow$ is not violated by an update.

Similarly, scheme $_2$ allows us to ensure that $- \rightarrow$ is satisfied, and $_3$ serves this purpose for account-number. However, the dependency:

$$- \rightarrow -$$

is not preserved. That is it possible to insert two branch names for the same account number (violating $- \rightarrow -$). the violation of this dependency cannot be detected unless a join is computed. To see why the decomposition of *BD-scheme* into $_{1,2,3}$ and $_4$ is not dependency-preserving, apply the algorithm of *Figure 1.7*. We find that the restriction of

to each scheme is as follows (trivial dependencies are omitted);

$_1$: $- \rightarrow$

$_2$: $- \rightarrow -$

$_3$: $- \rightarrow$

$_4$: only trivial dependencies hold.

Thus the set

is

$- \rightarrow$

$- \rightarrow -$

$- \rightarrow$

It is easy to see that the functional dependency

$$- \rightarrow -$$

Is not in $^+$ even though it is in $^+$. therefore, $^+ \neq ^+$ and the decomposition is not dependency-preserving.

The above example demonstrates that not every BCNF decomposition is dependency-preserving. We shall now show that it is not always possible to satisfy all three design goals:

BCNF
lossless join
dependency preservation

Consider the scheme $= (,,)$ with a given set of functional dependencies

$\rightarrow$

$\rightarrow$

Clearly,

is not in BCNF since $\rightarrow$ and

is not a superkey. However, every BCNF decomposition of

must fail to preserve $\rightarrow$. The algorithm of *Figure 1.8* generates the decomposition $\{(,)(,)\}$. The decomposed schemes preserve only $\rightarrow$ (and trivial dependencies) but the closure of $\{\rightarrow\}$ does not include $\rightarrow$

Third Normal Form

In those cases where we cannot meet all three design criteria, we abandon BCNF and accept a weaker normal form called third normal form (3NF). We shall see that it is always possible to find a lossless-join, dependency-preserving decomposition that is in 3NF.

BCNF requires that all nontrivial dependencies be of the form $\rightarrow$ where X is a superkey. 3NF relaxes this constraint slightly by allowing nontrivial functional dependencies whose left side is not a superkey.

A relation scheme

is in 3NF if for all functional dependencies that hold on

of the form $\rightarrow$, where $\subseteq$ and , at least one of the following holds:

$\rightarrow$ is a trivial function dependency

is a superkey for

is contained in a candidate key for

The definition of 3NF allows certain functional dependencies that are not allowed in BCNF. A dependency $\rightarrow$ that satisfies only the third condition of the 3NF definition is not allowed in BCNF though it is allowed in 3NF. These dependencies are called transitive dependencies.

Observe that if a relation scheme is in BCNF, then all functional dependencies are of the form "superkey" determines a set of attributes" {or the dependency is trivial}. Thus, a BCNF scheme cannot have any transitive dependencies at all. As a result, every BCNF scheme is also in 3NF, and BCNF is therefore a more restrictive constraint than 3NF .

Let us return to our simple example of a scheme that did not have a dependency-preserving, lossless-join decomposition into BCNF.

Let $=$ $(,,)$ with $=$ $\{\to , \to\}$. although scheme

is not in BCNF, it is in 3NF. To see that this is so, note that

is a candidate key for

, so the only attribute not contained in a candidate key for

is

. The only non trivial functional dependency of the form $\to$ is $\to$. Since

is a candidate key, this dependency does not violate the definition of 3NF.

Figure 1.9 shows an algorithm for finding a dependency-preserving, lossless-join decomposition in 3NF. It ensures preservation of dependencies by building explicitly a schemed for each given dependency. It ensure that the decomposition is a lossless-join decomposition by ensuring that at least one scheme contains a candidate key for the scheme being decomposed. The exercise provide some insight into the proof that this success to guarantee a lossless join.

We illustrate the algorithm of *Figure 1.9*by using it to generates a 3NF decomposition of BD-scheme. The **for** loop causes us to include the following schemes in our decomposition:

(loan-number, account)

(loan-number, branch-name)

(account-number, balance)

```
:= 0
for each functional dependency → in
do
    begin
        := + ;
        := ;
    end
if none of the schemes , ≤ ≤ contains a candidate key for

    then begin
        := + ;
```

(account-number, branch-name)

Since none of the above schemes contains a candidate key for *BD-scheme*, we add a scheme consisting of a candidate key

(customer-name, loan-number, account-number)

The final **if** statement does not cause us to add any more schemes to our decomposition since all attributes of BD-scheme already appear in some attribute of the decomposition.

Comparison of BCNF and 3NF

We have seen two normal forms for relational database schemes: 3NF and BCNF. There is an advantage to 3NF in that we know that it is always possible to obtain a 3NF design without sacrificing a lossless join or dependency preservation. Nevertheless, there is a disadvantages to 3NF. If we do not eliminate all transitive dependencies, it may be necessary to use null values to represent some of the possible meaningful relationships among data items. To illustrate this, consider the scheme = (,,) with = { → , → }. Since → , we may want to represent relationships between values for L and values for K in our database. However, in order to do so, either there must be a corresponding value for J, or we must use a null value for the attribute J.

If we are forced to choose between BCNF and dependency preservation with 3NF, it is generally preferable to opt for 3NF. If we cannot test for dependency preservation efficiently, we either pay a high penalty in system performance or risk the integrity of the data in our database.

$\twoheadrightarrow$

Neither of these alternatives is attractive. With such alternatives, the limited amount of redundancy imposed by transitive dependencies allowed under 3NF is the lesser evil. Thus we normally choose to retain dependency preservation and sacrifice BCNF.

To summarize the above discussion, we note that our goal for a relational database design is:

BCNF
lossless join
dependency preservation

If we cannot achieve this, we accept:

3NF
lossless join
dependency preservation

Normalization Using Multivalued Dependencies

Until now, the only form of constraint we have allowed on the set of legal relations is the functional dependency. We now define another form of constraint, called a multivalued dependency. As we did for functional dependencies, we shall use multivalued dependencies to define a normal form for relation schemes. This normal form, called fourth normal form (4NF), is more restrictive than BCNF. We shall see that every 4NF scheme is also in BCNF, but there are BCNF schemes that are not in 4NF.

Multivalued Dependencies

Functional dependencies rule out certain tuples from being in a relation. If $\rightarrow$, then we cannot have two tuples with the same

value but different

values. Multivalued dependencies do not rule out the existence of certain tuples. Instead, they require that other tuples of a certain form sometimes are referred to as "equality-generating" dependencies.

Let

be a relation scheme and let $\subseteq$ and $\subseteq$

The multivalued dependency:

$\twoheadrightarrow$

holds on

if in any legal relation (), for all pairs of tuples $_1$ and $_2$ in

such that

$_1[] = _2[]$, there exist tuples $_3$ and $_4$ in

such that:

Table 1.13 Tabular representation of $\rightarrow$

			$- \; -$
$_1$	$_1 \cdots$	$_{+1} \cdots$	$_{+1} \cdots$
$_2$	$_1 \cdots$	$_{+1} \cdots$	$_{+1} \cdots$
$_3$	$_1 \cdots$	$_{+1} \cdots$	$_{+1} \cdots$
$_4$	$_1 \cdots$	$_{+1} \cdots$	$_{+1} \cdots$

$$_1[] = _2[] = _3[] = _4[]$$

$$_3[] = _1[]$$

$$_3[-] = _2[-]$$

$$_4[] = _2[]$$

$$_4[-] = _1[-]$$

This definition is less complicated than it appears. In *Tablee 1.13*, we give a tabular picture of $_1, _2, _3$, and $_4$. intuitively, the multivalued dependency $\rightarrow$ says that the relationship between

and

is independent of the relationship between and $-$. If the multivalued dependency $\rightarrow$ is satisfied by all relations on scheme

, then $\rightarrow$ is a trivial multivalue dependency on scheme

. Thus $\rightarrow$ is trivial if $\subseteq$ or $\cup$ $=$.

To illustrate the difference between functional and multivalued dependencies, consider again our banking example. Let us assume that instead of the schemes borrow-scheme and customer-scheme, we have the single scheme:

BC-scheme =(loan-number, customer-name, street, customer-city)

The astute reader will recognize this as a non–BCNF scheme because of the functional dependency *customer-name→street customer-city* that we asserted earlier, and the fact that *customer-name* is not a key for *BC-scheme*. However, let us assume that our bank is attracting wealthy customers who have several addresses (say, a winter home and a summer home). Then, we no longer wish to enforce the functional dependency *customer-name→street customer-city*. If we remove this functional dependency, we find *BC-scheme* to be in BCNF with respect to our modified set of functional dependencies. Despite the fact the *BC-scheme* is now in BCNF, we still have the problem of repetition of information that we had earlier.

Consider the relation BC(BC-scheme) of Table *1.14*, we must repeat the, loan-number once for each address a customer has, and we must repeat the address for each loan a customer has. This repetition is unnecessary since the relationship between a customer and his address is independent of the relationship between that customer and a loan. If a customer, say Smith,

Table 1.14 Relation bc*, an example of redundancy in a BCNF relation.*

loan-number	customer-name	street	customer-city
23	Smith	North	Rye
23	Smith	Main	Manchester
93	Curry	North	Rye
93	Curry	Main	Manchester

has a loan, say loan number 23, we want that loan to be associated with all of Smith's addresses. Thus, the relation of *Table 1.15* is illegal. To make this relation legal, we need to add the tuples (23, Smith, MAIN, Manchester) and (27, Smith, North, Rye) to the BC relation of *Table 1.15*. Comparing the above example with our definition of multivalued dependency, we see that we want the multivalued dependency:

customer-name ↠ street customer-city

to hold. (The multivalued dependency *customer-name ↠loan-number* will do as well. We shall soon see that they are equivalent).

As was the case for functional dependencies, we shall use multivalued dependencies in two ways:

1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies.

2 To specify constraints on the set of legal relations. We shall thus concern ourselves only with relations that satisfy a given set of functional and multivalued dependencies.

Note that if a relation

fails to satisfy a given multivalued dependency, we can construct a relation

that does satisfy the multivalued dependency by adding tuples to

.

Table 1.15 An illegal BC relation

loan-number	*customer-name*	*street*	*customer-city*
23	Smith	North	Rye
27	Smith	Main	Manchester

Theory of Multivalued Dependencies

As was the case for functional dependencies and 3NF and BCNF, we shall need to determine all the multivalued dependencies that are logically implied by a given set of multivalued dependencies.

We take the same approach here that we did earlier or functional dependencies. Let

denote a set of functional and multivalued dependencies. The closure $^+$, of

is the set of all functional and multivalued dependencies. We can compute $^+$ from

using the formal definitions of functional dependencies and multivalued dependencies. However, it is usually earlier to reason about set of dependencies using a system of inference rules.

The following list of inference rules for functional and multivalued dependencies is sound and complete. Recall that soundness means that the rules do not generate any dependencies that are not logically implied by

. Completeness means that the rules allow us to generate all dependencies in $^+$. The first three rules are Armstrong's axioms, which we saw earlier.

1. Reflexivity rule. If is a set of attributes and $\subseteq$, then $\rightarrow$ holds.
2. Augmentation rule. If $\rightarrow$ holds and

   is a sset of attributes, then $\rightarrow$ holds
3. Transitivity rule. If $\rightarrow$ holds and $\rightarrow$ holds, then $\rightarrow$ holds
4. Complementation rule. If $\twoheadrightarrow$ holds then $\twoheadrightarrow - -$ holds.
5. Multivalued augmentation rule. If $\twoheadrightarrow$ holds and $\subseteq$ and $\subseteq$, then $\twoheadrightarrow$ holds.
6. Multivalued transitivity rule. If $\rightarrow$ holds and $\rightarrow$ holds, then $\rightarrow -$ holds.
7. Replication rule. If $\rightarrow$ holds, then $\rightarrow$
8. Coalescence rule. If $\rightarrow$ holds and $\subseteq$ and there is a

   such that $\subseteq$ and $\cap$ = and $\rightarrow$, then $\rightarrow$ holds.

The complementation rules states that if $\rightarrow$, then $\rightarrow$. Observe that $_3$ and $_4$ satisfy the definition of $\rightarrow$ if we simply change the subscripts.

We can provide similar justification for rules 5 and 6 using the definition of multivalued dependencies.

Rule 7, the replication rule, involves functional and multivalued dependencies. Suppose that $\rightarrow$ holds on

. then if $_1[] = _2[], _1[] = _2[]$, and $_1$ and $_2$ themselves serve as the tuples $_3$ and $_4$ required by the definition of the multivalued dependency $\rightarrow$.

Rule 8, the coalescence rule, is the most difficult of the eight rules to verify

We can simplify the computation of the closure of

by using the following rules which can be proved using rules 1 to 8

Multivalued union rule. If $\rightarrow$ holds and $\rightarrow$ holds, then $\rightarrow$ holds.
Intersection rule. If $\rightarrow$ holds and $\rightarrow$ holds, then $\rightarrow$ $\cap$ holds
Differences rule. If $\rightarrow$ holds and $\rightarrow$ holds, then $\rightarrow -$ holds and $\rightarrow -$ holds.

Let us apply our rules to the following example. Let $= (,,,,)$ with the following set of dependencies

given:

$\rightarrow$

$\rightarrow$

$\rightarrow$

We list some members of $+$ below:

$\rightarrow$: Since $\rightarrow$, the complementation rule (rule 4)implies that $\rightarrow - - . - - =$, so $\rightarrow$

$\rightarrow$: Since $\rightarrow$ and $\rightarrow$, the multivalued transitivity rule (rule 6) implies that $\rightarrow -$. Since $- =$, $\rightarrow$

$\rightarrow$: To show this fact, we need to apply the coalescence rule (rule 8). $\rightarrow$ holds since $\subset$ and $\rightarrow$ and $\cap =$. We satisfy the statement of the coalescence rule with

being

,

being

,

being

, and

being

. We conclude that $\rightarrow$.

$\rightarrow$: We already know that $\rightarrow$ and $\rightarrow$. By the difference rule, $\rightarrow -$. Since $- = \therefore \rightarrow$.

Let us return to our BC-scheme example in which customer-name$\rightarrow$street and customer-city holds, but no nontrivial functional dependencies hold. We saw earlier that, although BC-scheme is in BCNF, it is not an ideal design since we must repeat a customer's address is information for each loan. We shall see that we can use the given multivalued dependency to improve the database design, by decomposing BC-scheme into a fourth normal form (4NF) decomposition.

A relation scheme is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in $^{+}$ of the form $\twoheadrightarrow$, where $\subseteq$ and $\subseteq$, at least one of the following hold:

$\rightarrow$ is trivial multivalued dependency

is a superkey for scheme

A database design is in 4NF if each member of the set of relation schemes comprising the design is in 4NF.

Note that the definition of 4NF differs from the definition of BCNF only in the use of multivalued dependencies instead of functional dependencies. Every 4NF scheme is in BCNF. To see that this is so, note that if a scheme

is not in BCNF, then there is a nontrivial functional dependency $\rightarrow$ holding on

, where

is not a key. Since $\rightarrow$ implies $\twoheadrightarrow$ (by the replication rules),

cannot be in 4NF.

The analogy between 4NF and BCNF applies to the algorithm for decomposing a scheme into 4NF. *Figure 1.10* shows the 4NF decomposition algorithm. It is identical to the BCNF decomposition algorithm of *Figure 1.8* except for the use of multivalued instead of functional dependencies.

If we apply the algorithm of *figure 1.10* to *BC-scheme*, we find that $- \twoheadrightarrow -$ is a nontrivial multivalued dependency and customer-name is not a key for BC-scheme. Following the algorithm, we replace BC-scheme by two schemes:

(customer-name, loan number)

(customer-name, street, customer-city)

This pair of schemes which are in 4NF eliminates the problem we have encountered with redundancy of BC-scheme.

As was the case when we were dealing solely with functional dependencies, we are interested also in decompositions that are lossless-join decompositions and that preserve dependencies. The following fact about multivalued dependencies and lossless joins shows that the algorithm of *figure 1.10* generates only lossless-join decompositions:

```
Let

$:= \{\};$

done := false;
while (not done) do
    if (there is a scheme   result that is not in 4NF)
        then begin
            let $\rightarrow$ be a nontrivial multivalued dependency that holds on  such that is not in $^+$
            and $\cap$   $=$
            $:= ( - ) \cup (\ \ ) \cup ();$
```

be a relation scheme and a set of functional and multivalued dependencies on . Let $_1$ and $_2$ form a decomposition of This decomposition is a lossless-join decomposition of if and only if at least one of the following multivalued dependencies is in $^+$:

$$_1 \cap _2 \twoheadrightarrow _1$$

$$_1 \cap _2 \twoheadrightarrow _2$$

Recall that we stated earlier that if $_1 \cap _2 \rightarrow _1$ or $_1 \cap _2 \rightarrow _2$, then $_1$ and $_2$ are a lossless-join decomposition of

. The above fact regarding multivalued dependencies is a more general statement about lossless joins. It says that for every lossless-join decomposition of into two schemes $_1$ and $_2$,, one of the two dependencies $_1 \cap _2 \twoheadrightarrow _1$ or $_1 \cap _2 \twoheadrightarrow _2$ must hold.

The question of dependency preservation when we have multivalued dependencies is not as simple as for the case in which we have only functional dependencies. Let R be a relation scheme and let $_1, _2, \ldots,$ be a decomposition of

. recall that for a set

of functional dependencies, the restriction of

to is all functional dependencies in $^+$ that include only attributes of . Now consider a set

of both functional and multivalued dependencies. The restriction of

to is the set consisting of:

All functional dependencies in $^+$ that include only attributes of

All multivalued dependencies of the form:

$$\twoheadrightarrow \cap$$

where $\subseteq$ and $\twoheadrightarrow$ is in $^+$

A decomposition of scheme

into schemes $_1, _2, \ldots,$ is a dependency-preserving decomposition with respect to a set

of functional and multivalued dependencies if every set of relations $_1(_1), _2(_2), \ldots ()$ such that all, , satisfies , there exists a relation $()$ that satisfied

and for which $= \Pi()$ for all

.

Let us apply the 4NF decomposition algorithm of *figure 1.10* to our example of $= (,,,,)$ with $= \{ \twoheadrightarrow , \twoheadrightarrow , \rightarrow \}$. We shall then test the resulting decomposition for dependency preservation

is not in 4NF. Observe that $\twoheadrightarrow$ is not trivial, yet

is not a key. Using $\twoheadrightarrow$ in the first iteration of the **while** loop, we replace

with two schemes $(,)$ and $(,,,)$. it is easy to see that $(,)$ is in 4NF since all multivalued dependencies that hold on $(,)$ are trivial. However, the scheme $(,,,)$ is not in 4NF. Applying the multivalued dependency $\twoheadrightarrow$ (which follows from the given functional dependency $\rightarrow$ by the replication rule), we replace $(,,,,)$ by the two schemes $(,,)$ and $(,,,)$. Scheme $(,,)$ is in 4NF, but scheme $(,,,)$ is not.

To see that $(,,,)$ is not in 4NF recall that we showed earlier that $\twoheadrightarrow$ is in $^{+}$. therefore $\twoheadrightarrow$ is in the restriction of

to $(,,,)$. Thus in a third iteration of the **while** loop, we replace $(,,,)$ by two schemes $(,)$ and $(,,)$. the resulting 4NF decomposition is $(,)$ $(,,)$ $(,)$, $(,,)$.

This 4NF decomposition is not dependency-preserving since it fails to preserve the multivalued dependency $\twoheadrightarrow$. Consider the relations of *Figure 1.11*. It shows the four relations that may result from the projection of a relation on $(,,,,,)$ onto the four schemes of our decomposition. The restriction of

to $(,)$ is $\twoheadrightarrow$ and some trivial dependencies. It is easy to see that $_1$ satisfies $\rightarrow$ because there is no pair of tuples with the same

value. Observe that $_2$ satisfies all functional and multivalued dependencies since no two tuples in $_2$ have the same value on any attribute.

$_1$:

$_1$	$_1$
$_2$	$_2$

$_2$:

$_1$	$_1$	$_1$
$_2$	$_2$	$_2$

$_3$:

$_1$	$_1$
$_2$	$_2$

$_4$:

$_1$	$_1$	$_1$
$_2$	$_2$	$_2$

Figure 1.11 Projection of r unto a 4NF decomposition of R

A similar statement can be made for $_3$ and $_4$. Therefore, the decomposed version of our database satisfies all the dependencies in the restriction of

. however, there is no relation

on $(, , , , ,)$ that satisfieds and decomposes into $_1, _2, _3$, and $_4$. To see this, observe that $= \bowtie^{4}_{=1}$ is as shown in *Figure 1.12*. Relation

does not satisfy $\twoheadrightarrow$. Any relation

containing

and satisfying $\twoheadrightarrow$ must include the tuple $(_2, _1, _2, _2, _2, _1)$. However $\Pi()$ includes a tuple $(_2, _2,)$ that is not in $_2$. Thus, our decomposition fails to detect a violation of $\twoheadrightarrow$

$_1$	$_1$	$_1$	$_1$	$_1$	$_1$
$_2$	$_2$	$_2$	$_2$	$_1$	$_2$

Figure 1.12 A relation $()$ that does not satisfy $\twoheadrightarrow$

We have seen that if we are given a set of multivalued and functional dependencies, it is advantageous to find a database design that meets the three criteria of:

4NF
dependency preservation
lossless join.

If all we have are functional dependencies, the first criteria is just BCNF. We have seen also that it is not always possible to achieve all three of these criteria. We succeeded in finding such a decomposition for the bank example, but failed for the example of scheme $= (,,,,)$. When we cannot achieve our three goals, compromise on 4NF, and accept BCNF or even 3NF, if necessary to ensure dependency preservation.

Normalization Using Join Dependencies

We have seen that the lossless-join property is one of several properties of a good database design. Indeed, this property is essential since, without it, information is lost. When we restrict the set of legal relations to those satisfying a set of functional and multivalued dependencies, we are able to use these dependencies to show that certain decompositions are lossless-join decompositions. Because of the importance of the concept of lossless join, it is useful to be able to constrain the set of legal relations over a scheme

to those relations for which a given decomposition is a lossless-join decomposition.

In this unit, we define such a constraint, called a join dependency. As has been the case for other types of dependency, join dependencies will lead to another normal form called project-join normal form (PJNF).

Join Dependencies

Let

be a relation scheme and $_{1,2,...,}$ be a decomposition of R, the join-dependency $(_{1,2,...,})$ is used to restrict the set of legal relations to those for which $_{1,2,...,}$ is a lossless-join decomposition of

. Formally, if $= \quad = 1$, we say that a relation $()$ satisfies the join dependency $(_{1,2,...,})$ if:

$\bowtie_{=1} (\Pi()) =$

A *join dependency* is trivial if one of the is

itself.

Consider the join dependency $({}_1\ {}_2)$ on scheme

. This dependency requires that for all legal $()$

$$r = \Pi_{R_1}() \bowtie \Pi_{R_2}()$$

Let

contain the two tuples ${}_1$ and ${}_2$ defined as follows:

$${}_1[{}_1 - {}_2] = ({}_1, {}_2, \dots, {}_i) \qquad {}_2[{}_1 - {}_2] = ({}_1, {}_2, \dots, {}_i)$$

$${}_1[{}_1 \cap {}_2] = ({}_{+1}, \dots,) \qquad {}_2[{}_1 \cap {}_2] = ({}_{+1}, \dots,)$$

$${}_1[{}_2 - {}_1] = ({}_{+1}, \dots,) \qquad {}_2[{}_1 - {}_2] = ({}_{+1}, \dots,)$$

Thus, ${}_1[{}_1 \cap {}_2] = {}_2[{}_1 \cap {}_2]$, but ${}_1$ and ${}_2$ have different values on all other attributes. Let us compute $\Pi_{R_1}() \bowtie \Pi_{R_2}()$. *Figure 1.13* shows $\Pi_1()\ \Pi_2()$. When we compute the join, we get two additional tuples besides ${}_1$ and ${}_2$, as shown by ${}_3$ and ${}_4$ in *Figure 1.14*

If $*({}_1, {}_2)$ holds, whenever we have tuple ${}_1$ and ${}_2$ we must also have ${}_3$ and ${}_4$. Thus *Figure 1.14* shows a tabular representation of the join dependency $*({}_1, {}_2)$. Compare *Figure 1.14* to Figure 1.13, in which we gave a tabular representation of $\twoheadrightarrow$. If we let $= {}_1 \cap {}_2$ and $= {}_1$, then we can see that the two tabular representations in these figures are the same. Indeed $*({}_1, {}_2)$ is just another way of stating ${}_1 \cap {}_2 \twoheadrightarrow {}_1$. Using the complementation and augmentation rules for multivalued dependencies, we see that ${}_1 \cap {}_2 \twoheadrightarrow {}_1$ implies ${}_1 \cap {}_2 \twoheadrightarrow {}_2$. Thus, $*({}_1, {}_2)$ is equivalent to ${}_1 \cap {}_2 \twoheadrightarrow {}_2$. This observation is not surprising in light of the fact we noted earlier that ${}_1$ and ${}_2$ form a lossless-join decomposition of

if and only if ${}_1 \cap {}_2 \twoheadrightarrow {}_2$ or ${}_1 \cap {}_2 \twoheadrightarrow {}_1$.

	${}_1 - {}_2$	${}_1 \cap {}_2$
$\Pi_1({}_1)$	${}_1 \dots$	${}_{+1} \dots$
$\Pi_1({}_2)$	${}_1 \dots$	${}_{+1} \dots$

	${}_1 \cap {}_2$	${}_2 - {}_1$

$\Pi_2(_1)$	$+1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$
$\Pi_1(_2)$	$+1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$

Figure 1.13 $\Pi_1() \; \Pi_2()$

	$_1 - _2$	$_1 \cap _2$	$_2 - _1$
1	$1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$
2	$1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$
3	$1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$
4	$1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$	$+1 \cdot \cdot \cdot$

Figure 1.14 Tabular representation of $*(_1, _2)$.

Every join dependency of the form $*(_1, _2)$ is therefore equivalent to a multivalued dependency. However, there are join dependencies that are not equivalent to any multivalued dependency. The simplest example of such a dependency is on scheme $= (,,)$. The join dependency $(,) \; (,), (,)$ is not equivalent to any collection of multivalued dependencies. Table 1.16 shows a tabular representation of this join dependency. To see that no set of multivalued dependencies logically implies $(,),(,),(,)$, consider *Table 1.16* as a relation $(, <)$ as shown in *Table 1.15* relation

satisfies $(,),(,),(,)$, as can be verified by computing:

$\Pi() \bowtie \Pi() \bowtie \Pi()$

and showing that the result is exactly

. However,

does not satisfy any nontrivial multivalued dependency. To see this, verify that r fails to satisfy any of $\twoheadrightarrow$, $\twoheadrightarrow$, $\twoheadrightarrow$, $\twoheadrightarrow$, $\twoheadrightarrow$, $\twoheadrightarrow$.

Just as a multivalued dependency is a way of stating the independence of a pair of relationship, a join dependency is a way of stating that a set of relationships are all independent. This notion of independence of relationships is a natural consequence of the way we generally define a relation.

*Table 1.16 Tabular representation of *((A,B), (B,C),(A,C))*

1	1	2
2	1	1
1	2	1
1	1	1

Consider, for example:

Borrow-scheme = (branch-name, loan-number, customer-name, amount)

From our banking example. We can define a relation borrow (borrow-scheme) as the set of all tuples on borrow-scheme such that

the loan represented by *loan-number* is made by the branch named *branch-name*
the loan represented by *loan-number* is made to the customer named *customer-name*.
the loan represented by *loan-number* is in the amount given by *amoun*t

The above definition of the borrow relations is a conjunction of three predicates: one on loan-number and branch-name, one on loan-number and customer-name, and one on loan-number and amount. Surprisingly, it can be shown that the above intuitive definition of borrow logically implies the join dependency **((loan-number, branch-name)(loan-number, customer-name)(loan-number amount))*

Thus, join dependencies have an intuitive appeal and correspond to one of our three criteria for a good database design.

For functional and multivalued dependencies, we were able to give a system of inference rules that are sound and complete. Unfortunately, no such set of rules is known for join dependencies. It appears to be necessary to consider more general classes of dependencies than join dependencies to contrast a sound and complete set of inference rules. The bibliographic notes contain references to recent research in this area.

Project – Join Normal Form

Project-join normal form is defined in a manner similar to BCNF and 4NF, except that join dependencies are used. A relation scheme R is in *project-join normal form* (PJNF) with respect to a set D of functional, multivalued, and join dependencies if for all join dependencies in $^+$ of the form $*(_1, _2, \ldots,)$ where each $\subseteq$ and $\bigcup_{=1}$ $=$, at least one of the following holds.

$*(_1, _2, \ldots,)$ is a trivial join dependency

Every is a superkey for

A database design is in PJNF if each member of the set of relation schemes comprising the design is in PJNF. PJNF is called fifth normal form (5NF) in some of the literature on database normalization.

Let us return to our banking example. Given the join dependency **((loan-number, branch-name, (loan-number, customer-name), (loan-number, amount,)) borrow-scheme* is not in PJNF. To put borrow-scheme into PJNF, we must decompose it into the three schemes specified by the join dependency: *(loan-number, branch-name,) (loan-number, customer-name) and (loan-number, amount).*

Because every multivalued dependency is also a join dependency, it is easy to see that every PJNF scheme is also in 4NF. Thus, in general, we may not be able to find a dependency-preserving decomposition for a given scheme into PJNF.

Domain-key Normal Form

The approach we have taken toward normalization is to define a form constraint (functional, multivalued, or join dependency) and then use that form of constraint to define a normal form. Domain-Key normal form is based on three notions.

Domain declaration. Let be an attribute, and let be a set of values. The domain declaration $\subseteq$ requires that the value of all tuples be values in .

Key declaration. Let

be a relation scheme with $\subseteq$. The key declaration key () requires that

be a superkey for

, that is $\rightarrow$. Note that all key declarations are functional dependencies are key declarations.

General constraint. A general constraint is a predicate on the set of all relations on give scheme. The dependencies we have studied in this unit are examples of a general constraint. In general, a general constraint is a predicate expressed in some agreed-upon form, such as first-order logic.

We now give an example of a general constraint that is not functional, multivalued, or join dependency. Suppose that all accounts whose number begins with 9 are special high-interest accounts with a minimum balance of ₦2500. then we would include as a general constraint "if the first digit of t[account-number] is 9, then t [balance] $\geq$2500"

Domain declarations and key declarations are easy to test in a practical database system. General constraints, however, may be extremely costly (in time and space) to test. The purpose of a domain-key normal form database design to allow the general constraints to be tested using only domain and key constraints.

Formally, let

be a set of domain constraints and let

be a set of key constraints implies that our database design is not in DKNF. To create a DKNF design, we need two schemes in place of account-info-scheme.

Regular-acct-scheme =(branch-name, account-number, balance)

Special-acct-scheme = (branch-name, account-number, balance)

We retain all the dependencies we had on account-info-scheme as general constraints. The domain constraints for special-acct-scheme require that:

the account number begins with 9
the balance is greater than 2500

The domain constraints for regular-acct-scheme require that the account number not begin with 9. the resulting design is in DKNF although the proof of this fact is beyond the scope of this text.

Let us compare DKNF to the other normal forms we have studied. Under the other normal forms, we did not take into consideration domain constraints. We assumed (implicitly) that the domain of each attribute was some infinite domain such as the set of all integers or the set of all character strings. We allowed key constraints (indeed, we allowed functional dependencies). For each normal form, we allowed a restricted form of general constraint (a set of functional, multivalued, or join dependencies.) Thus, we can rewrite the definitions of PJNF, 4NF, BCNF, and 3NF in a manner which shows them to be special cases of DKNF.

The following is a DKNF-inspired rephrasing of our definition of PJNF. Let $= (_{1,2},\ldots,)$ be a relation scheme. Let $()$ denote the domain of attribute , and let all these domains be infinite. Then all domain constraints

are of the form $\subseteq ()$. Let the general constraints be a set

of functional, multivalued, or join dependencies. If

is the set of functional dependencies in

, let the set

of key constraints be those nontrivial functional dependencies in $^+$ of the form $\rightarrow$. Scheme

is in PJNF if and only if it is in DKNF with respect to , , .

A consequence of DKNF is that insertion and deletion anomalies are eliminated.

DKNF represents an "ultimate" normal form because it allows arbitrary constraints rather than dependencies, yet it allows efficient testing of these constraints. Of course, if a scheme is not in DKNF we may be able to achieve DKNF via decomposition, but such decomposition, as we have seen, are not always dependency-preserving decompositions. Thus, while DKNF is a goal of a database designer, it may have to be sacrificed in a practical design.

Atomic Values

Conspicuous by their absence from our discussion in 3NF, 4NF, etc are first and second normal forms. The reasons for this is that second normal form is of historical interest only. Although we have not mentioned first normal form, we have assumed it since we introduced the relational model. We now, finally, make our assumption explicit.

A relation scheme

is in first normal form (INF) if the domains of all attributes of are atomic. A domain is atomic if elements of the domain are considered to be individisble units.

For example, the set of integers is an atomic domain, but the set of all sets of integers is a non atomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider set of integers to have subparts namely, the integers comprising the set. The important issue in INF is not the domain itself, but the way we use domain elements in our database. The domain of all integers would be non atomic if we considered each integer to be an ordered list of digits.

In all our examples, we have assumed atomic domains. We have made this assumptions not only for the relational model but also for the entity-relationship, network and hierarchical model. Because this assumption is so natural, we have chosen not to draw attention to it until now.

The reason we mention INF at all is that recent development in database theory and practice have called into question the legitimacy of assuming INF. Let us consider a document retrieval system. For each document, we store the following information.

Document title
Author list.
Date
Keywords lists

Table 1.17 Non 1NF documents relation

title	*author-list*	*date*			*keywords*
		day	*month*	*year*	
sales plan	{Smith, Jones}	1	April	79	{profit, strategy}
status plan	{Jones, Frick}	17	June	85	{profit, personnel}

We can see that if we define a relation for the above information, several domains will be non atomic.

Authors. A documents may have a set of authors. Nevertheless, we may want to find all documents of which Jones was one of the authors. Thus we are interested in a subpart of the domain element "set of authors"

Keywords. If we store a set of keywords for a document, we expect to be able to retrieve all documents whose keywords include one or more keywords. Thus, we view the domain of keyword list as non atomic.

Date. Unlike keywords and authors, date does not have a set-valued domain. However, we may view date as consisting of the subfields day, month, and year. This makes the domain of date monatomic.

Table 1.17 shows an example document relation. This relation can be represented in INF, but the resulting relation is awkward (see *Table 1.18*). Since we must have atomic domains in INF, yet want access to individual authors and to individual keywords, we need one tuple for each (keyword, author) pair. The date attribute is replaced in the INF version by three attributes; one for each subfield of date.

title	*author*	*day*	*month*	*year*	*keyword*
salesplan	Smith	1	April	79	profit
salesplan	Jones	1	April	79	profit
salesplan	Smith	1	April	79	strategy
salesplan	Jones	1	April	79	strategy
status report	Jones	17	June	79	profit
status report	Frick	17	June	79	profit
status report	Jones	17	June	79	personnel
status report	Frick	17	June	79	personnel

Table 1.18 INF version of non-INF relation in Table 1.17

Much of the awkwardness of the relation in *Table 1.18* is removed if we assumed that:

title $\twoheadrightarrow$ *author*

title $\twoheadrightarrow$ *keyword*

title $\twoheadrightarrow$ *day month year*

Then, we can decompose the relation into 4NF using the schemes:

(title, author)
(title, keyword)
(title, day, month, year)

Table 1.19 shows the projection of the relation of *Table 1.18* onto the above decomposition.

Although INF can represent our example document database adequately, the non-1NF representation may be an easier-to-understand model for the typical user of a document retrieval system.

Table 1.19 4NF version of 1NF relation in Table 1.18

title	*author*
salesplan	Smith
salesplan	Jones
status report	Jones
status report	Frick

title	*keyword*
salesplan	Smith
salesplan	Jones
status report	Jones
status report	Frick

title	*day*	*mont h*	*yea r*
salesplan	1	April	79
status report	17	June	85

Users of such system think of the database in terms of our non-1NF design. The 4NF design would require users to include joins in their queries, there by complicating interaction with the system. We could define a view that eliminates the need for users to write joins in their query.

However in such a view, we lose the one-to-one correspondence between tuples and documents.

Once we realize that there are new applications of database systems that benefit from a non-1NF representation, we need to define new normal forms and criteria for goodness of a design.

Alternative Approaches to Database Design

In this unit, we re-examine normalization of relation schemes with an emphasis on the impact of normalization on the design of practical database systems.

We have taken the approach of starting with a single relation scheme and decomposing it. One of our goals in choosing a decomposition was that the decomposition be a lossless-join decomposition. In order to consider losslessness, we assumed that it is valid to talk about the join of all the relations of the decomposed database.

Consider the database of *Figure 1.16* showing a borrow relation decomposed in PJNF. In *Figure 1.16*, we represent a situation in which we have not yet determined the amount of loan 58, but wish to record the remainder of the data on the loan. If we compute the natural join of these relations we discover that all tuples referring to loan 58 disappear. In other words, there is no borrow relation corresponding to the relations of *Figure 1.16*. We refer to the tuples that "disappear" in computing the join as dangling tuples.

branch-name	*loan-number*
Round Hill	58

loan-number	*amount*

loan number	*customer-number*
58	Johnson

Figure 1.16 Decomposition in PJNF

Formally, let $_1(_1),_2(_2),\ldots,()$ be a set of relations. A tuple of relation is a dangling tuple if it is not in the relation:

$$\Pi(\bowtie_{=1}\ )$$

Dangling tuples may occur in practical database applications. They represent incomplete information, as in our example where we wish to store data about a loan still in the process of being negotiated. The relation $\bowtie_{=1}$, is called a *universal relation* since it involves all the attributes in the "universe" defined by $\cup_{=1}$.

The only way we can write a universal relation for the examples of Figure 1.16 is to include *null values* in the universal relation. Ongoing research regarding null values and universal relations is discussed in the bibliographic notes. Because of the difficulty of managing null values, it may be desirable to view the relations of the "decomposed" design as representing "the" database, rather than the universal relation whose scheme we decomposed during the normalization process.

Note that not all incomplete information can be entered into the database of *Figure 1.16* without resorting to the use of null values. For example, we cannot enter a loan number unless we know at least one of the following:

The customer name
The branch name
The amount of the loan.

Thus, a particular decomposition defines a restricted form of incomplete information that is acceptable in our database.

The normal forms that we have defined generate good database designs from the point of view of representation of incomplete information. Returning again to the example of *Figure 1.16*, we would not want to allow the storage of the following fact "there is a loan (whose number is unknown) to Jones in the amount of ₦100". Since *loan-number* $\rightarrow$ *customer-name amount* is the only way we can relate *customer-name* and *amount* through *loan-number*. If we do not know the loan number, we cannot distinguish this loan from other loan with unknown numbers.

In other words, we do not want to store data for which the key attributes are unknown. Observed that the normal forms we have defined do not allow us to store that type of information unless we use null values. Thus, our normal forms allow representation of acceptable incomplete information via dangling tuples while prohibiting the storage undesirable incomplete information. If we allow dangling tuples in our database, we may prefer to take an alternative view of the database design process. Instead of decomposing a universal relation, we may *synthesize* a collection of normal form schemes from a given set of attributes. We are interested in the same normal forms regardless of whether we use decomposition or synthesis. The decomposition approach is better understood and more widely used.

Another consequence of our approach to the database design is that the attribute names must be unique in the universal relation. We cannot use name to refer to both customer-name and to branch-name. It is generally preferable to use unique names, as we have done. Nevertheless, if we defined our relation schemes directly rather that in terms of a universal relation, we could obtain relations on schemes such as the following for out banking example:

branch-loan(name, number)

loan-number(name, number)

loan(number, amount)

Observe that with the above relation, such expression as *branch-name* ⋈ *loan-customer* are meaningless. Indeed, the expression *branch-loan* ⋈ *loan-customer* finds loan made by branches to customers with the same name as the branch.

However, in a language like SQL, there is natural join operation, so in a query involving *branch-loan* and *loan-customer,* references to name must be disambiguated by prefixing the relation name. In such environment, the multiple roles for name (as branch name and customer name) are less troublesome and may be simpler for some users.

We feel that the unique role assumption that each attribute name has a unique meaning in the database, is generally preferable to the use of the same name in multiple roles. When the unique role assumption is not made, the database designer must be especially careful when constructing a normalised relational database design.

1.5 CONCLUSION

A database management system provides us with mechanisms for storing and organizing data to facilitate easy access to and manipulation of the data. Today's most popular database systems are relational databases.

1.6 SUMMARY

In this unit we defined additional operations to express the requests. These operations enhance the expressive power of the relational algebra. We have presented criteria for a good database design:

Lossless join
Dependency preservation
PJNF, BCNF, 4NF, or 3NF

We have shown how to achieve these goals and how to find a good compromise when not all the criteria can be achieved.

In order to represent these criteria, we defined several types of data dependencies

Functional dependencies
Multivalued dependencies
Join dependences

We studied the properties of these dependencies with emphasis on what dependencies are logically implied by a set of dependencies.

DKNF is an idealized normal form that may be difficult to achieve in practice. Yet, DKNF has desirable properties that should be included to the extent possible in a good database design.

In reviewing the issues we have discussed in this unit, it is worthwhile to observe that the reason we could define rigorous approaches to relational database design is that the relational data model rest on a firm mathematical foundation

1.7 TUTOR-MARKEDASSIGNMENT(TMA).

1. Define the following terms: domain, attribute, n-tuple, relation schema, relation state

2. Why are tuples in a relation not ordered

3 Why are duplicate tuples not allowed in a relation

4 What is the difference between a key and a upper key

5 List the operations of relational algebra and the purpose of each

6 What is union compatibility ? Why do the UNION, INTERSECTION, and DIFFERENCE operations require that the relations on which they are applied be union compatible.

7. Consider the following relations for a database that keeps track of student enrolment in courses and the books adopted for each course:

   STUDENT (SSN, Name, Major, Bdate)

   COURSE (course#, Cname, Dept)

   ENROLL(SSN, course#, Quarter, Grade)

   BOOK-ADOPTION (course#, QUARTER, BOOK-ISBN)

   TEXT (BOOK-ISBN, Book-Title, Publisher, Author)

Specify the foreign keys for the above schema, stating any assumptions you make. Then specify the following queries in relational algebra:

(a) List the number of courses taken by all students named 'Daniel Olayinka' in December 1991 (i.e. Quarter = w99)
(b) Produce a list of textbooks (includes course#, Book-ISBN, Book – Title) for course offered by the 'cs' department that have used more than two books.

1.8 FURTHERREADINGS

Atzeni, P., and De Antonellis, V. [1993] Relational Database Theory Benjamin / Cummings, 1993.

Batini., C., Ceri, S., and Navathe, S. [1992] Database Design: An Entity – Relationship Approach, Benjamin/Cummings, 1992.

Bernstein, P. [1976] "Synthesizing Third Normal Form Relations from Functional Dependencies", TODS, 1:4, December 1976.

Codd, E [1970] "A Relational Model for Large Shared Data BANks" CACM, 136, June 1970.

Maier, D. [1983] The Theory of Relational Databases, Computer Science Press, 1983.

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Schmidt., J., and Swenson, J., [1975] "On the Semantics of the Relational Model" in SIGMOD [1975].

Silberwschatz., A., and Korth, H.,[1986]. "Database System Concepts", McGraw – hill, 1986.

MODULE 1: DATABASE DESIGN AND IMPLEMENTATION

UNIT 2 : DATABASE IMPLEMENTATION

2.0 INTRODUCTION

Database design is a part of a larger picture called an information system more like a relational database management system.. Within the system, we not only collect, store and retrieve data, but also transform that raw data into useful information. The system does not just happen, they are a product of a carefully staged development process. To determine the need for an information system and to establish its limit, we use a process known as system analysis to develop the system. This continuous process of creation, maintenance, enhancement, and replacement constitute the Systems Development Life Cycle. The information contained in the database, to a large extent, is the product of interviews with the end users. These people are the system's main beneficiaries and must be identified carefully. The key users of the University Computer Lab application developed in this unit are:

- The lab director, who is charged with the Lab's operational management.
- The lab assistants LA, who are charged with the Lab's daily operations.
- The secretary of the Computer Information Systems (CIS) department, who assists in the Lab's general administrative functions.

In the interest of brevity, we will show only a few excerpts of the numerous interviews that were performed for this project.

2.1 OBJECTIVES

To understand the stages involved in the database design

To see important features of the implementation stage

To consider various operations involved during design and implementation
To develop a University Computer Lab application

2.2 CONCEPTUAL DESIGN

The conceptual database blueprint development helps us define the basic characteristic of the database environment. Using an analogy, if an architect's blueprint show a wall, it is important you know if that wall is to be made of board, brick, block or poured concrete and whether that wall will bear a load or merely act as a partition. In short, *detail matters*.

To develop a good conceptual design we must be able to gather information that lets us accurately identify the entities and describe their attributes and relationships must accurately reflect real-world relationships.

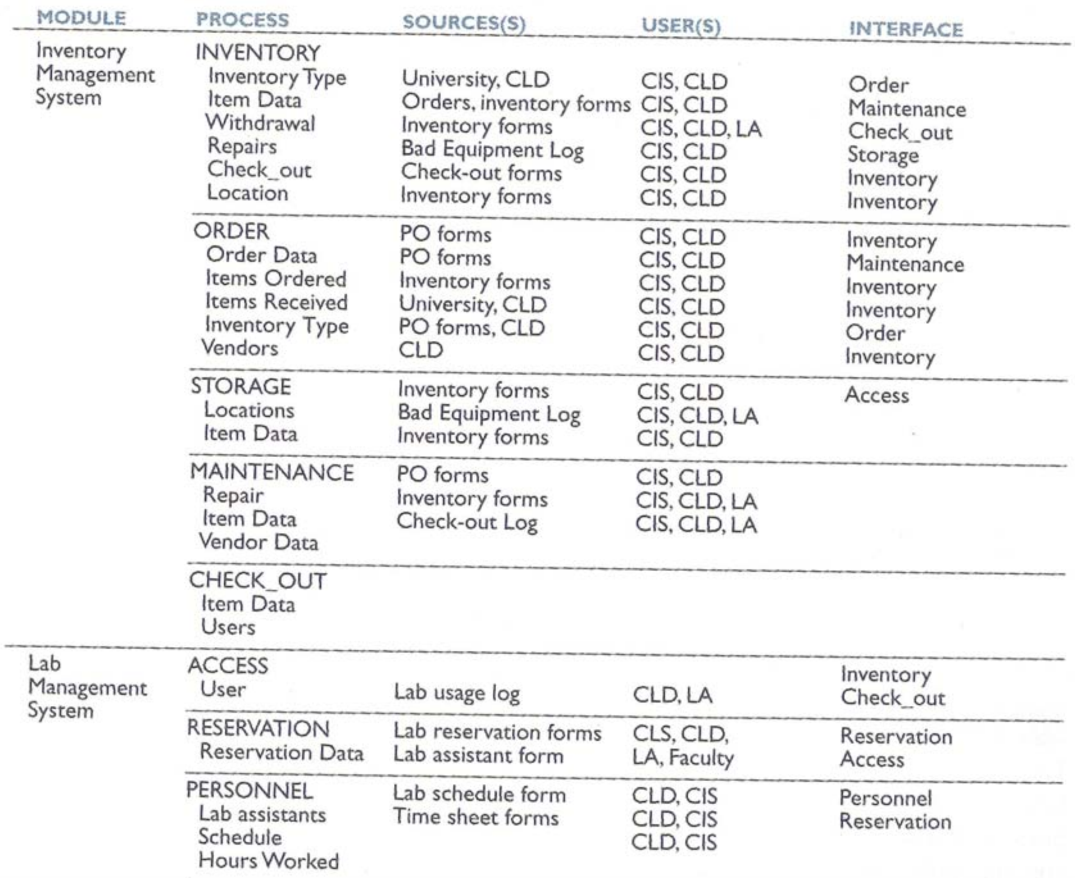
2.2.1 Information Sources And Users

We must begin the conceptual design phase by confirming good information sources. The confirmation process recertifies key users and carefully catalogues all current and prospective end users. In addition, the confirmation process targets the current system's paper flow and documentation , including data and report forms. For the UCL the following are necessary:

Computer Lab director (CLD).
Computer information Systems (CIS) department secretary and chair.
Computer Lab assistant (LA).
Students, faculty, and staff who use the Lab's resources.
All currently used computer lab forms, file, and report forms.
Not surprisingly, a list of prospective system user tends to duplicate at least a portion of the list of information sources.

The CLD (who is also the UCL system administrator) will manage the system, enter data into the database, and define reporting requirements.
The LAs are the primary UCL system users and will enter data into the database.
The CIS secretary is a UCL system user and will query the database.

We suggest that you create a summary table to identify all system sources and users. Such a table can be used for cross-checking purposes, thereby enabling you to audit sources and users more easily. The UCL system summary is shown in Table 2.1.



MODULE	PROCESS	SOURCES(S)	USER(S)	INTERFACE
Inventory Management System	INVENTORY			
	Inventory Type	University, CLD	CIS, CLD	Order
	Item Data	Orders, inventory forms	CIS, CLD	Maintenance
	Withdrawal	Inventory forms	CIS, CLD, LA	Check_out
	Repairs	Bad Equipment Log	CIS, CLD	Storage
	Check_out	Check-out forms	CIS, CLD	Inventory
	Location	Inventory forms	CIS, CLD	Inventory
	ORDER	PO forms	CIS, CLD	Inventory
	Order Data	PO forms	CIS, CLD	Maintenance
	Items Ordered	Inventory forms	CIS, CLD	Inventory
	Items Received	University, CLD	CIS, CLD	Inventory
	Inventory Type	PO forms, CLD	CIS, CLD	Order
	Vendors	CLD	CIS, CLD	Inventory
	STORAGE	Inventory forms	CIS, CLD	Access
	Locations	Bad Equipment Log	CIS, CLD, LA	
	Item Data	Inventory forms	CIS, CLD	
	MAINTENANCE	PO forms	CIS, CLD	
	Repair	Inventory forms	CIS, CLD, LA	
	Item Data	Check-out Log	CIS, CLD, LA	
	Vendor Data			
	CHECK_OUT			
	Item Data			
	Users			
Lab Management System	ACCESS			Inventory
	User	Lab usage log	CLD, LA	Check_out
	RESERVATION	Lab reservation forms	CLS, CLD,	Reservation
	Reservation Data	Lab assistant form	LA, Faculty	Access
	PERSONNEL	Lab schedule form	CLD, CIS	Personnel
	Lab assistants	Time sheet forms	CLD, CIS	Reservation
	Schedule		CLD, CIS	
	Hours Worked			

Table 2.1 Data Sources and Users

Source: Database System Design, Implementation and Management Rob Coronel

2.3 E-R MODEL VERIFICATION

At this point we would have identified:

Entity sets, attributes, and domains.

Composite attributed. Such attributes may be (and usually are) decomposed into several independent attributes.

Multivalued attributes. We implemented them in a new entity set in a I:M relationship with the original entity set.

Primary keys. We ensured primary key integrity.

Foreign keys. We ensured referential integrity through the foreign keys.

Derived attributes. We ensured the ability to computer their values.

Composite entities. We implemented them with 1:M relations.

Although we seem to have made considerable progress, much remains to be done before the model can be implemented.

To complete the UCL conceptual database design we must verify the model. Verification represents the link between the database modelling and design activities, database implementation, and database application design. Therefore, the verification process is used to establish that:

The design properly reflects the end user or application views of the database.

All database transaction - inserts, updates, deletes – are defined and modelled to ensure that the implementation of the design will support all transaction – processing requirements.

The database design is capable of meeting all output requirements, such as query screens, forms, and report formats. (Remember that information requirements may drive part of the design process).

All required input screens and data entry forms are supported.

The design is sufficiently flexible to support expected enhancements and modifications. In spite of the fact we were forces to revise the E-R diagram depicted in Figure 2.1, it is still possible that:

Some of the relationship are not clearly identified and may even have been misidentified.

The model contains design redundancies (consider the similarity between the WTIHDRAW and CHECK-OUT entities).

The model can be enhanced to improve semantic precision and better represent the operations in the real world.

The model will demonstrate the requirements (such as processing performance or security).

In the following few paragraphs, we will demonstrate the verification process for some of the application views in the inventory Management module. (This verification process should be repeated for all the system's modules.)

Identifying the Central Entity. Although the satisfaction of the UCL's end users is vital, inventory management has the top priority from an administrative point of view. The reason for this priority rating is simple: state auditors target the Lab's considerable and costly inventory to ensure accountability to the state's taxpayers. Failure to track items properly may have serious consequences and, therefore, ITEM becomes the UCL's central entity.

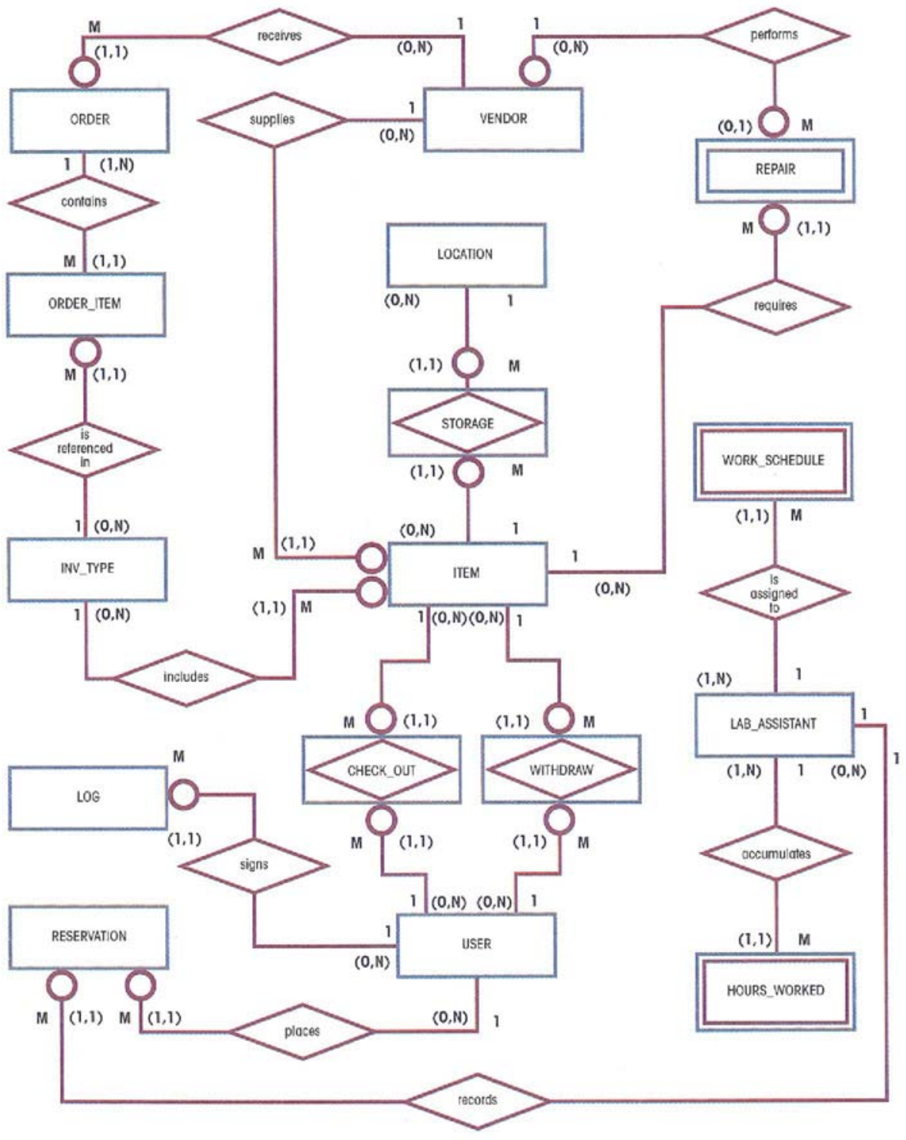


Figure 2.1 The UCL Management System's Initial ERD

Source: Database System Design, Implementation and Management. Rob Coronel

Identifying each module and its components. It is important to "connect" the components by using share entities. For example, although USER is classified as belonging to the Lab management module and ITEM is classified as belonging to the inventory Management module, the USER and ITEM entities interface with both. For example, the USER is written into the LOG in the Lab Management module. USER also features prominently in the inventory Management module's withdrawal of supplies and in the check-out-in processes.

Identifying each module transaction requirement. We will focus our attention on one of the INVENTORY module's reporting requirements. Then, validate these requirement against UCL database for all system modules.

A sample inventory Management module's reporting requirement uncovers the following problems:

The inventory module generates three reports, one of which is an inventory movement report. But the inventory movements are spread across several different entities (CHECK-OUT and WITHDRAW and ORDER). Such a spread makes it difficult to generate the output and reduces system performance.

An item's quantity on hand is updated with an inventory that can represent a purchase, withdrawal, check-out, check-in, or inventory adjustment. Yet only the withdrawal and check-outs are represented in the model.

The solution to these problems is described by the database designer:-

What the inventory Management module needs is a common entry point for all movements. In other words, the system must track all inputs to and withdrawals from inventory. To accomplish this task, we must create a new entity to record all inventory movements; that is, we need an inventory transaction entity. We will name this entity INV-TRANS.

The creation of a common entry point serves two purposes:

1. It standardizes the inventory module's interface with other (external) modules. Any inventory (whether it adds or withdraws) will generate an inventory transaction entry.
2. It facilities control and generation of required outputs, such as the inventory movement report.

Figure 2.2 illustrates a solution to the problems we have just described.

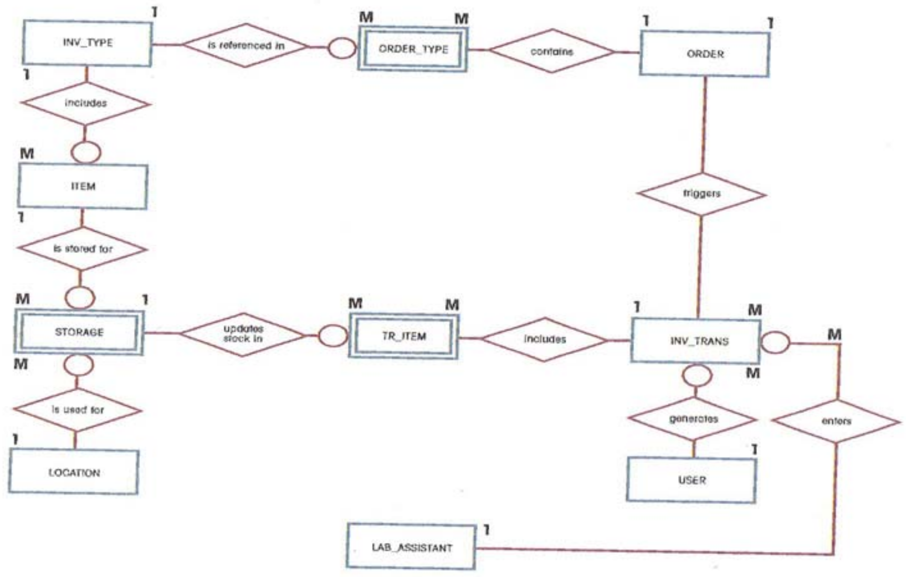


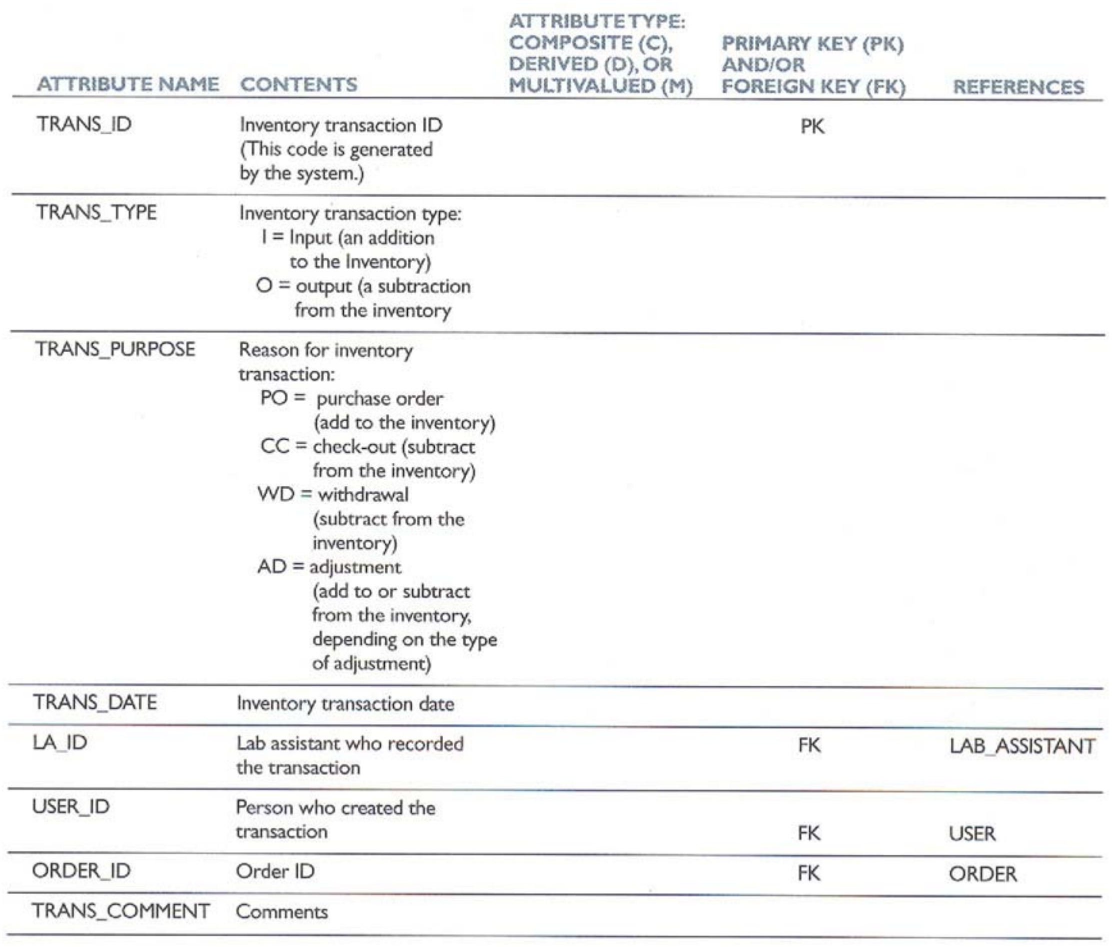
Figure 2.2 The Inventory Transaction Process

Source: Database System Design, Implementation and Management. Rob Coronel

The INV-TRANS entity in Figure 2.2 is a simple inventory transaction log, and it will contain any inventory 1/0 movement. Each INV-TRANS entry represents one of two types of movement: input (+) or output (-). Each INV-TRANS entry must contain a line TR-ITEM for each item that is added. Withdrawn, checked in, or checked out.

The INV-TRANS entity's existence also enables us to build additional 1/0 movement efficiently. For example, when an ordered item is received, an inventory transaction entry (+) is generated. This INV-TRANS entry will update the quantity received (01-QTY-RECVD) attribute of the ORDER-ITEM entity in the inventory management module. The inventory management module will generate an inventory transaction entry (+) to register the items checked in by a user. The withdrawal of items (supplies) will also generate an inventory transaction entry (-) to register the items that are being withdrawn. These relationship are depicted in Figure 2.2.

The new INV-TRANS entity's attributes are shown in Table 2.2.



ATTRIBUTE NAME	CONTENTS	ATTRIBUTE TYPE: COMPOSITE (C), DERIVED (D), OR MULTIVALUED (M)	PRIMARY KEY (PK) AND/OR FOREIGN KEY (FK)	REFERENCES
TRANS_ID	Inventory transaction ID (This code is generated by the system.)		PK	
TRANS_TYPE	Inventory transaction type: I = Input (an addition to the Inventory) O = output (a subtraction from the inventory			
TRANS_PURPOSE	Reason for inventory transaction: PO = purchase order (add to the inventory) CC = check-out (subtract from the inventory) WD = withdrawal (subtract from the inventory) AD = adjustment (add to or subtract from the inventory, depending on the type of adjustment)			
TRANS_DATE	Inventory transaction date			
LA_ID	Lab assistant who recorded the transaction		FK	LAB_ASSISTANT
USER_ID	Person who created the transaction		FK	USER
ORDER_ID	Order ID		FK	ORDER
TRANS_COMMENT	Comments			

Table 2.2 The INV_TRANS Entity

Source: Database System Design, Implementation and Management. Rob Coronel

can be accessed through STORAGE. Because INV-TRANS is relate to both LAB-ASSISTANT and USER, we know who recorded the transaction and who generated it.

Figure 2.3 The INV_TRANS Sample Data

Source: Database System Design, Implementation and Management. Rob Coronel

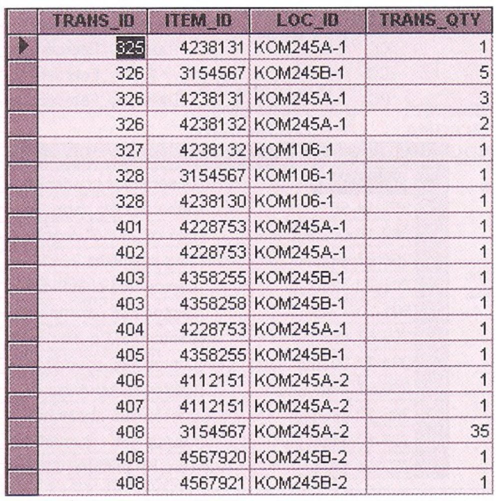
For example, the first INV-TRANS row lets us know that on Thursday, February 4, 1999 a laser printer cartridge was withdrawn from inventory by user 299-87-9234. The transaction was recorded by LA 387-99-9565, and the transaction decreased (TRANS-TYPE = 0) the stock in inventory.

The transaction details in Figure 2.3 are stored in TR-ITEM, so before we can examine these details we must examine the TR-ITEM structure in Table 2.3.

Table 2.3 The TR_ITEM(Weak) Entity

Source: Database System Design, Implementation and Management. Rob Coronel

We can trace some transaction details in Figure 2.3.



TRANS_ID	ITEM_ID	LOC_ID	TRANS_QTY
325	4238131	KOM245A-1	1
326	3154567	KOM245B-1	5
326	4238131	KOM245A-1	3
326	4238132	KOM245A-1	2
327	4238132	KOM106-1	1
328	3154567	KOM106-1	1
328	4238130	KOM106-1	1
401	4228753	KOM245A-1	1
402	4228753	KOM245A-1	1
403	4358255	KOM245B-1	1
403	4358258	KOM245B-1	1
404	4228753	KOM245A-1	1
405	4358255	KOM245B-1	1
406	4112151	KOM245A-2	1
407	4112151	KOM245A-2	1
408	3154567	KOM245A-2	35
408	4567920	KOM245B-2	1
408	4567921	KOM245B-2	1

Figure 2.4 The TR_ITEM Sample Data

Source: Database System Design, Implementation and Management. Rob Coronel

For example, note that the first INV-TRANS row's TRANS-ID = 325 entry (see Figure 2.3) now points to the TR-ITEM's TRAND-ID = 325 entry shown. Thus allowing us to conclude that this transaction involved the withdrawal of a single unit of item 4238131, a laser printer cartridge. (We can conclude that item 4238131, is a laser printer by examining the ITEM and TY-GROUP =

SUCALPXX.) Transaction 362 involved three items, so the TR-ITEM table contains three detail lines for this transaction.

Examine how checks-out and check-in are handled. In Figure 2.3, INV TRANS transaction 401 records TRANS PURPOSE= CC and TRANS TYPES= O, indicating that a check was made. This transaction recorded the following: check-out of a 486 laptop, ITEM ID= 4228753 on Tuesday, February 2, 1999 by an accounting faculty member, USERID= 301-23-4245. The laptop was returned on Wednesday, February 3, 1999, and this transaction was recorded as TRANS ID=402, whose TRANS PURPOSE= CC and TRANS TYPES=1, indicating that this particular 486 laptop was returned to the available inventory. Indicating, because the department owns several laptops, Faculty members need not wait for one to be returned before checking out a laptop, as long as there are laptops in inventory, However, if no additional laptops are available, the system can trace who has them and when they were checking out. If the CLD wants to place restriction on the length of time an item can be check out, this design makes it easy to notify users to return the items in question.

The Final entity relationship diagram reflects the changes that we have made, Although the original E-R diagram is easier to understand from the users point of v, the revised E_R diagram has more meaning from the procedural point of view. For example, the changes we have made are totally transparent (invisible) to the user, because the user never seen the INV-TRANS entity. The final E-R diagram is shown in Figure 2.4.

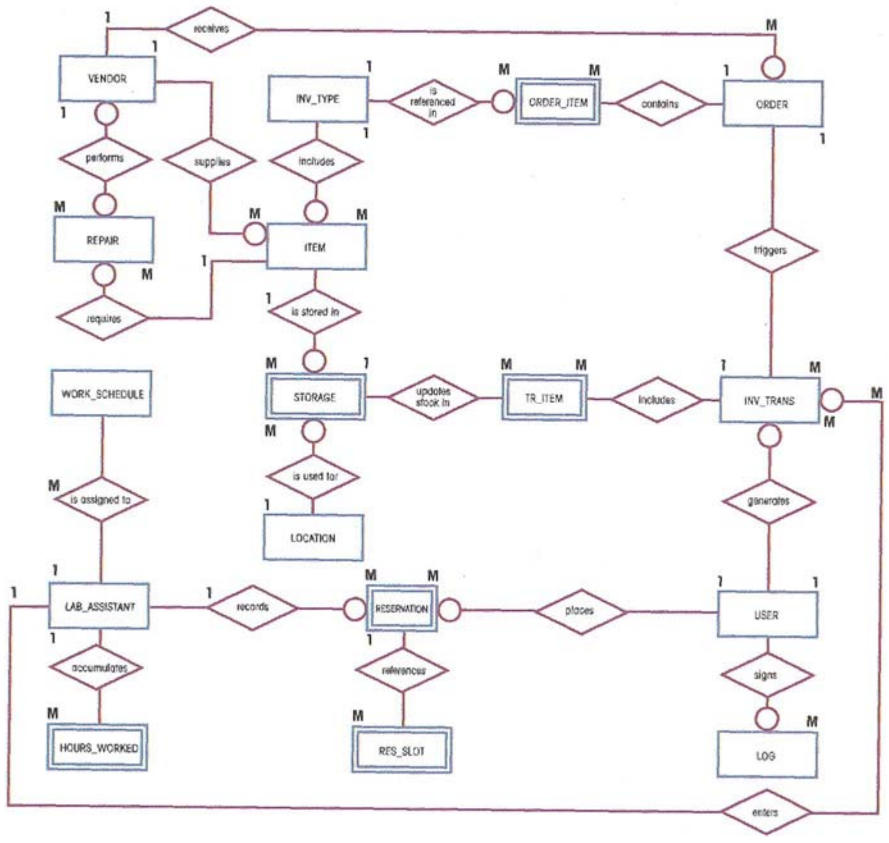


Figure 2.4 The Revised University Computer Lab ERD

Source: Database System Design, Implementation and Management. Rob Coronel

2.4 LOGICAL DESIGN

When the conceptual design phase is completed, the ERD reflects – at the conceptual level-the business rules that , turn, define the entities, relationship, optional ties, connectives, cardinalities, and constraints. (Remember that some of the design elements cannot be modelled and are, therefore, enforced at the application level;. For example, the constraint, “a checked out item must be returned within five days” cannot be reflected in the ERD.) In addition, the conceptual model includes the definition of the attributes that describe that describe each of the entities and are required to meet information requirements.

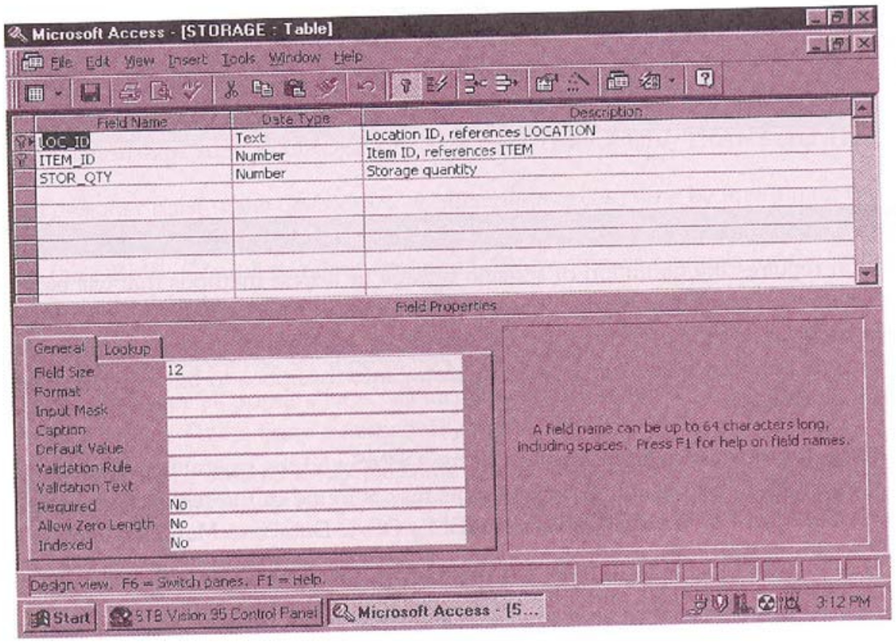
Keep in mind that the conceptual model’s entries must be normalized before they can be properly implemented. The normalization process may yield additional entities and relationships, thus requiring the modification of the initial ERD. We made sure that the model we verified in this unit meets the requisite normalization requirements. In short, we used design and normalization reflects real-world practice. However, you should remember that the logical design process is used to translate the conceptual design into the internal model for the selected DBMS. To the extent that normalization helps establish the appropriate attributes, their characteristics, and their domains, normalization moves you to the logical design phase. Nevertheless, because the conceptual modelling process does preclude the definition of attributes, you can reasonably argue that normalization occasionally straddles the line between conceptual and logical modelling.

The following few examples illustrate the design of the logical model using SQL. (Make sure that the table conform to the E-R model's structure and that they obey the foreign rules if your DBMS software allows you to specify foreign keys.)

Using SQL, you can create the table structures within the database you have designated. For example, the STORAGE table would created with:

```
CREATE TABLE STORAGE(
        LOC_ID          CHAR(12)                NOT NULL,
        ITEM_ID                 CHAR(10)                NOT NULL,
        STOR_QTY                NUMBER,
        PRIMARY KEY(LOC_ID,ITEM_ID),
        FOREIGN KEY(LOC_ID) REFERENCES LOCATION
                ON DELETE RESTRICT
                ON UPDATE RESTRICT,
        FOREIGN KEY(ITEM_ID) REFERENCES ITEM
                ON DELETE CASCADE
                ON UPDATE CASCADE);
```

Most DBMSs now use interfaces that allow you to type the attribute names into a template and to select the attribute characteristics you want from pick lists. You can even insert comments that will be reproduced on the screen to prompt the user for input. For example, the preceding STORAGE table structure might be created in a


Microsoft Access - [STORAGE : Table]
File Edit View Insert Tools Window Help
Field Properties
General
Lookup
Start
STB Vision 95 Control Panel
Microsoft Access - [S...
3:12 PM

Field Name	Data Type	Description
LOC_ID	Text	Location ID, references LOCATION
ITEM_ID	Number	Item ID, references ITEM
STOR_QTY	Number	Storage quantity

Field Size	12
Format	
Input Mask	
Caption	
Default Value	
Validation Rule	
Validation Text	
Required	No
Allow Zero Length	No
Indexed	No

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

Design view. F6 = Switch panes. F1 = Help.

Figure 2.5 The STORAGE Tables Structure Defined In Microsoft Access Source:

Database System Design, Implementation and Management. Rob Coronel

Microsoft Access template as shown in Figure 2.5.

2.4.1 Index And Views

In the logical design phase, the designer can specify appropriate indexes to enhance operational speed. Indexes also enable us to produce logically ordered output sequences. For example, if you want to generate a LA schedule , you need data from two tables. LAB-ASSISTANT and WORK-SCHEDULE. Because the report output is ordered by semester, LA, weekday, and time, indexes must be available for the primary key fields in each table. Using SQL, we would type:

```
CREATE UNIQUE INDEX LAS_DEX
        ON LAB_ASSISTANT(LA_ID)
```

and

```
CREATE UNIQUE INDEX WS_DEX
```

```
        ON WORK SCHEDULE(SCHED_SEMESTER, LA_ID, SCHED_WEEKDAY,

        SCHED_TIME_IN);
```

Most modern DBMSs automatically index on the primary key components.

Views (see Unit 3, "Advanced Structured Query Language") are often for security purposes. However, views are also used to streamline the system's processing requirements. For example, output limits may be defined efficiently appropriate views necessary for the LA schedule report for the fall semester of 1999, we use the CREAT VIEW command:

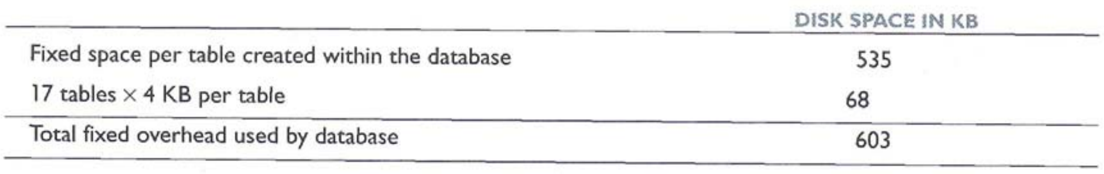
```
    CREATE VIEW LA_SCHED AS

        SELECT LA_ID, LA_NAME, SCHED_WEEKDAY, SCHED_TIME_IN

        SCHED_TIME_OUT

            WHERE SCHED_SEMESTER = 'FALL99';
```

The designer creates the view necessary for each database output operation.

2.5 PHYSICAL DESIGN

Physical design requires the definition of specific storage or access methods that will be used by the database. Within the DBMS's confine, the physical design must include as estimate to the space required to store the database. The required space estimate is translated into the space to be reserved within the available storage devices.

Physical storage characteristics are a function of the DBMS and the operating systems being used. Most of the information necessary to perform this task is found in the technical manuals of the software you are using. For example, if you use IBM's OS/2 Database Manager Version 1.2, an estimate of the physical storage required for database creation (empty database) would be provided by a table such as the shown in Table 2.4.

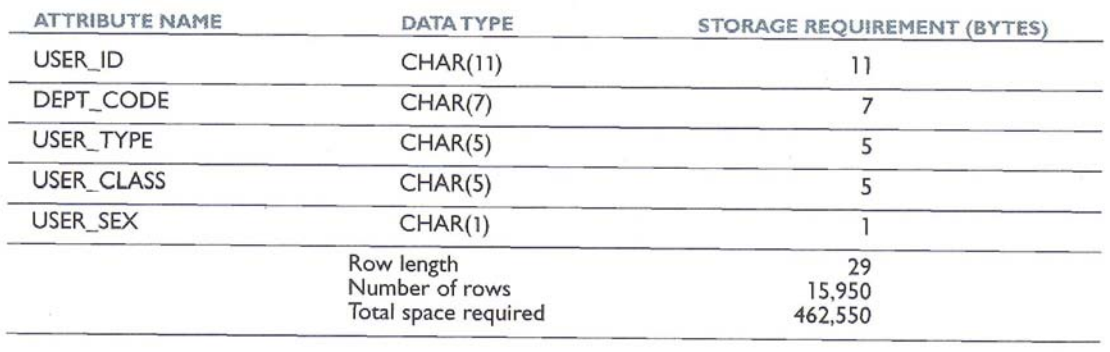


	DISK SPACE IN KB
Fixed space per table created within the database	535
17 tables × 4 KB per table	68
Total fixed overhead used by database	603

Table 2.4 Fixed Space Claimed by OS/2 DBM V1.2 Per Database

Source: Database System Design, Implementation and Management. Rob Coronel

Next we need to estimate the data storage requirement for each table. Table 2.5 shows this calculation for the USERS table only



ATTRIBUTE NAME	DATA TYPE	STORAGE REQUIREMENT (BYTES)
USER_ID	CHAR(11)	11
DEPT_CODE	CHAR(7)	7
USER_TYPE	CHAR(5)	5
USER_CLASS	CHAR(5)	5
USER_SEX	CHAR(1)	1
	Row length	29
	Number of rows	15,950
	Total space required	462,550

Table 2.5 Physical Storage Requirements: The USERS Table

Source: Database System Design, Implementation and Management. Rob Coronel

If the DBMS does not automate the process of determining storage locations and data access paths, physical design requires well-developed technical skills and a precise knowledge of the physical-level details of the database, operating system, and hardware used by the database, fortunately, the more recent versions of relational DBMS software hide most of the complexities inherent the physical design phase.

You might store the database within a single volume of permanent storage space; or you can use several volumes, distributing the data in order to decrease data-retrieval time. Some DBMSs also allow you to create cluster tables and indexes. Cluster tables store rows of different tables together, in consecutive disk locations. This arrangement speeds up data access; it is mainly used in master / detail relationship such as ORDER and ORDER-ITEM or INV-TRANS and TR-ITEM.

The database designer must make decisions that affect data access time by fine-tuning the buffer pool size, the page frame size, and so on. These decisions will be based on the selected hardware platform and the DBMS.

Indexes created for all primary keys will increase access speed when you use foreign key references in tables. This is done automatically by the DBMS.

Indexes can also be created for all alternate search keys. For example, if we want to search the LAB-ASSITANCE table by user name, we should create an index for the LA-LNAME attribute. Example:
CERATE INDEX LA001 ON LAB-ASSISTANT)LA-LNAME);

Indexes can be created for all secondary access keys used in report or queries. For example, an inventory movement report is likely to be ordered by inventory type and item ID. Therefore, an index is create for the ITEM table:
CREATE INDEX INV002 ON ITEM(TY-GROUP, ITEM-ID);

Indexes can be created for all columns used in the WHERE, ORDER BY, and GROUP BY clauses of a SELECT statement

2.6 IMPLEMENTATION

One of the significant advantages of using databases of using databases is that a database enables users to share data. When data are held in common, rather than being "Owner" by the various organizational divisions, data management becomes a much more specialized task. The database environment thus favours the creation of a new organizational structure designed to manage the database resources. Database management functions are controlled by the database administrator (DBA).

Once the data base designer has completed the conceptual, logical, and physical design phases, the DBA adopts an appropriate implementation plan. The plan includes formal definitions of the processes and standards to be followed, the chronology of the required activities (creation, loading, testing, and evaluation), the development of adequate documentation standards, the specific documentation needed, and the precise identification of responsibilities for continued development and maintenance.

2.6.1. Database Creation

All the tables, indexes, and views that were defined during the logical design phase are created during this phase. We also issue the commands or use utility programs at this time to create storage space and the access methods that were defined by the physical design.

2.6.2. Database Loading And Conversion

The newly created database contains the (still empty!) table structures. These table structures can be filed by entering (typing) the data one table at a time or by copying the data from existing databases or files. If the new table structures and formats are incompatible with those used the original DBMS or file system software, the copy procedure requires these of special loading or conversion utilities.

Because many process require a precise sequencing of data activities, data are loaded in a specific order. Because of foreign key requirement, the University Computer Lab database must be initially loaded in the following order:

1. User, vendor, and location data
2. Lab assistant and work schedule data
3. inventory type data
4. Item data

After these main entities have been loaded, the system is ready for testing and evaluation.

2.6.3 System Procedures

System procedures describe the steps been required to manage, access, and maintain the database system. The development of these procedures constitutes a concurrent and parallel activity that started in the early stages of the system's design.

A well-run database environment requires and enforces strict standards and procedures to ensure that the data repository is managed in an organized manner. Although operational and management activities are logically connected, it important to define distinct operations and management procedures.

IN the case of the University Computer Lab Management System, procedure must be established to:

Test and evaluate the database.
Fine-tune the security and integrity.
Back up and recover the database.
Access and the database system.

Several database may exist within a database environment. Each database must have its own set of system producers and must be evaluated in terms of how it fits into the organization's information system.

2.7 TESTING AND EVALUATION

The purpose of testing and evaluation is to determine how the database meets its goals. Although testing and evaluation constitute a district database life cycle (DBLC) phase, the implementation, testing and evaluation of the database are concurrent and related. Database testing and evaluation should be ongoing. A database that is acceptable today may not be acceptable a few years from now because of rapidly changing information needs. In fact, one of the important reasons for the relational database's growing dominances is its flexibility (Because relational database tables are independent, changes can be made relatively quickly and efficiently.)

2.7.1 Performance measures

Performance refers to Ability to retrieve information within a responsible time and at a responsible cost. A database system's performance can be affected by such fact ors as communication speeds, number of concurrent users, resources limits, and so on. Performance, primarily measured in terms of database query response time, is generally evaluated BY Computing the number of transaction per second .

Performance of the UCL database will be affected by the types of processor used by the database server computer, the available amount of memory in the server and in the workstation computers, the communication Network speed, and the number of concurrent users.

Several measures may enhance the UCL system's performance, keep in mind that, at the physical levels, each DBMS/hardware combination has its own specific advantages, For example, IBM's OS/2 DBM allows us to use the following techniques:

Define system configuration parameters to enhance system performance For instance, we can:

- o Change the maximum number of concurrently active databases by settings the NUMDB parameter.
- o Increase the number of the maximum shared memory segments that OS/2DBM uses to open a database, by setting the SQLNSEG parameters.

Adjust database configuration parameters to optimize the database, For instance, we can:

- o Increase the buffer pool manager size through the BUFFPAGE parameters.
- o Increase the shared segment used to sort data through SORTHEAP. The selection of this parameter affects the amount of work space that is used by the ORDER BY clause in a SELECT statement.

Enhanced system performance through the reorganization of the database with REORG and RUNSTATS commands.

Enhance system performance by fine-tuning the local area parameters and increasing the amount of RAM used in each of the network's computers.

The designer must fine-tune the database and the system itself in order to achieve acceptable performance. Make sure to distinguish between acceptable and optimum performance.

Performance is not the only important factors in database, nor is it necessarily the most important factors. In fact, requirement for proper concurrency control, data integrity, security, and backup/recovery may outweigh strict performance standards.

2.8 SECURITY MEASURES

Security measures seek to ensure that data safely stored in the database. Security is especially critical in a multiuser database environment, in which someone might deliberately enter inconsistent data. The DBA must define (with the help of end users) a company information policy that specifies how data are stored, accessed, and managed within a data security and privacy framework.

Access may be limited by using access rights or access authorizations. Such rights are assigned on the basics of the user's need to known or the user's system responsibilities, IN the case of the UCL database, access rights must be assigned to Las, the CLD and the CIS secretary. But such rights are limited For example, the Las may read their work schedules, but they are able to modify the data stored in LAB ASSISTANT or HOURS WORKED tables.

The database administrator may, for example, grant the use of a previously created LA_SCHED view to the lab assistant Anne Smith by using the following (SQL) syntax:

GRANT SELECT ON LA_SCHED TO LA_ASMITH;

In this case, only the LA_ASMITH may use the view LA_SCHED to check the LA schedules. A similar procedure is used to enable other Las to check the Lab schedules.

Backup And Recovery Procedures

Database backup is crucial to the continued availability of the database system after a database or hardware failure has occurred. Backup must be a formal and frequent procedure, and backup files should be located in predetermined sites. Recovery procedures must delineate the steps to ensure full recovery of the system after a system failure or physical disaster.

2.9 OPERATION

2.91 Database is Operational

An operational database provides all necessary support for the system's daily operations and maintains all appropriate operational procedures. If the database is properly designed and implemented, its existence not only enhances management's ability to obtain relevant information quickly, it also strengthens the entire organization's operational structure.

Operational Procedures

Database operational procedures are written documents in which the activities of the daily database operations are described. The operational procedures delineate the steps required to carry on specific tasks, such as data entry, database maintenance, backup and recovery procedures, and security measures.

2.11 CONCLUSION

From the study, the database designer prepares for the conceptual design by carefully establishing the organisation's information from sources and users. Good design must take into account the end user requirement for both as hoc queries and formail results. Report contents and formats are analysed carefully to provide clues about the entities that must be contained within the design.

2.11 SUMMARY

An examination of sample data makes it easier to evaluate the adequacy of the entities and their attributes against end user requirements. The designer uses normalization checks to ensure that data redundancies are kept to an absolute minimum, subject to performance requirements.

The E-R model's verification process also requires a careful examination of the propose system modules. Therefore, we must review and verify:

The central entity within the system.
All modules and their components.
All module processes.

The verification process often leads to the creation of few entities and attributes, or it might demonstrate that one or more existing may be deleted from design. For example, we discovered that the UCL's WITHDRAW entity could be implemented as shown in the initial E-R diagram. The addition of a new INV-TRANS entity and a set of relationship based on this entity closed the implementation gap.

Following the verification of the conceptual model based on the E-R diagram the designer addresses the logical design. The logical yields all appropriate table structures, indexes, and views and is, for this reason, software-dependent.

The completion of the logical design lets the designer address the database's physical design. The physical design addresses physical storage requirements, memory allocation, and access methods. Because these features are a function of the computer's operating system and its physical configuration, physical design is both software – and hardware-dependent.

The completion of the physical design allows the designer to being the database implementation process. The data are entered into the table structures created during the logical design phase, using appropriate loading and / or conversion utilities. The order in which data are loaded is a function of the foreign key requirements. The designer must also address the implementation of appropriate systems operational and management procedures.

Before the database design is accepted, it must be subjected to testing and evaluation. Actually, testing and evaluation are continuous database management activities, but they are especially important when the database is about to come on-line. Although database performance (speed and efficiency) testing and evaluation criterion. Testing and evaluation must ensure that database security and integrity are maintained hat backup and recovery procedure are in place, and that database access and use are properly secured.

When all appropriate operational and management issues have been addressed and implemented, the database is finally operational. The database's existence not only ensures the availability of information on which decision making is based, it also sets the stage for the evaluation of data as a corporate resource and tightens operational structure throughout an organization.

2.12 TUTOR MARKED ASSIGNMENT (TMA)

1. Why must a conceptual be verified? What steps are involved in the verification process?

2. what steps must be completed before the database design is fully implemented? (Make sure you follow the correct sequence and discuses each briefly.)
3. What major factors should be addressed when database system performance is evaluated? Discuss each factor briefly.
4. How would you verify the E-R diagram shown in Figure Q8.4? Make specific recommendations.
5. Described and discuss the E-R model's treatment of the UCL's inventory / order hierarchy:
   a. Category
   b. Class
   c. Type
   d. Subtype

6. You read in this unit that:
   An examination of the UCL's inventory Management module reporting requirements uncovered the following problems:

   The inventory module generates three reports, one of which is an inventory movement report. But the inventory movement are spared across two different entities (CHECK-OUT and WITHDRAW). Such a spread makes it different to generate the output and reduces the system's performance.
   An item's quantity on hand is updated with an inventory movement that can represent a purchase, with drawl, check-out, check-in, or inventory adjustment. Yet only the withdrawals and check-outs are represented in the system.

What solution was proposed for this set of problems? How would such a solution be useful in other sups of inventory environments?

2.13.FURTHERREADINGS

Boyce, R., Chamberlin, D., King, W., and Hammer, M. [1975] "Specifying Queries as Relational Expressions", CACM, 18 : 11, November 1975.

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[1986]. "Database System Concepts", McGraw – hill, 2000.

www.w3schools.com

Zobel, J., Moffat, A., and Sacks–Davis, R., [1992]. "An Efficient Indexing Technique for Full – Text Database Systems," in VLDB [1992.

MODULE 1: DATABASE DESIGN & IMPLEMENTATION

UNIT 3: ADVANCED SQL

3.0. INTRODUCTION

SQL - Structured Query Language is a standard language for accessing and manipulating databases. SQL lets you access and manipulate databases. It is used for defining tables and integrity constraints and for accessing and manipulating data. SQL (pronounced “S-Q-L” or “sequel”). This unit explains how to use SQL to access and manipulate data from database systems like MySQL, SQL Server, MS Access, Oracle, Sybase, DB2, and others.

3.1. OBJECTIVES

To understand the SQL statements
To be able to query a database using SQL queries

3.2. WHAT IS SQL?

SQL - Structured Query Language is a standard language for accessing and manipulating databases. SQL lets you access and manipulate databases. It is used for defining tables and integrity constraints and for accessing and manipulating data. SQL (pronounced “S-Q-L” or “sequel”). This unit explains how to use SQL to access and manipulate data from database systems like MySQL, SQL Server, MS Access, Oracle, Sybase, DB2, and others. Application programs may allow users to access a database without directly using SQL, but these applications themselves must use SQL to access the database.

Although SQL is an ANSI (American National Standards Institute) standard, there are many different versions of the SQL language. However, to be compliant with the ANSI standard, they all support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, WHERE) in a similar manner. Most of the SQL database programs also have their own proprietary extensions in addition to the SQL standard.

The Structured Query Language can be used to execute queries against a database, retrieve data from a database, insert records in a database, update records in a database, delete records from a database, create new databases, create new tables in a database, create stored procedures in a database, create views in a database, and set permissions on tables, procedures, and views among other things. It could also be used when creating a web application i.e. a build web site that shows some data from a database. Some other tools including SQL would be required for this.

Namely:

An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
A server-side scripting language, like PHP or ASP

HTML / CSS

RDBMS

RDBMS stands for Relational Database Management System. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access. The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

Database Tables

A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"). Tables contain records (rows) with data. Below is an example of a table called "Persons":

Table 3.0: Table Persons

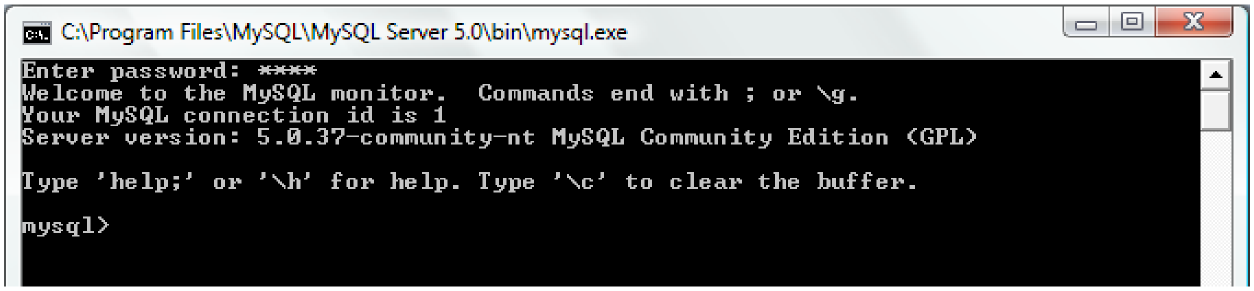
P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

The table above contains three records (one for each person) and five columns (P_Id, LastName, FirstName, Address, and City).

3.3. USING SQL ON A RELATIONAL DATABASE

SQL can be used on MySQL, Oracle, Sybase, IBM DB2, IBM Informix, Borland Interbase, MS Access, or any other relational database system. This unit uses MySQL to demonstrate SQL and uses MySQL, Access, and Oracle to demonstrate JDBC programming.

Assume that you have installed MySQL with the default configuration, you can access MySQL from the DOS command prompt using command MySQL from c./MySQL/bin directory, as shown in figure below.



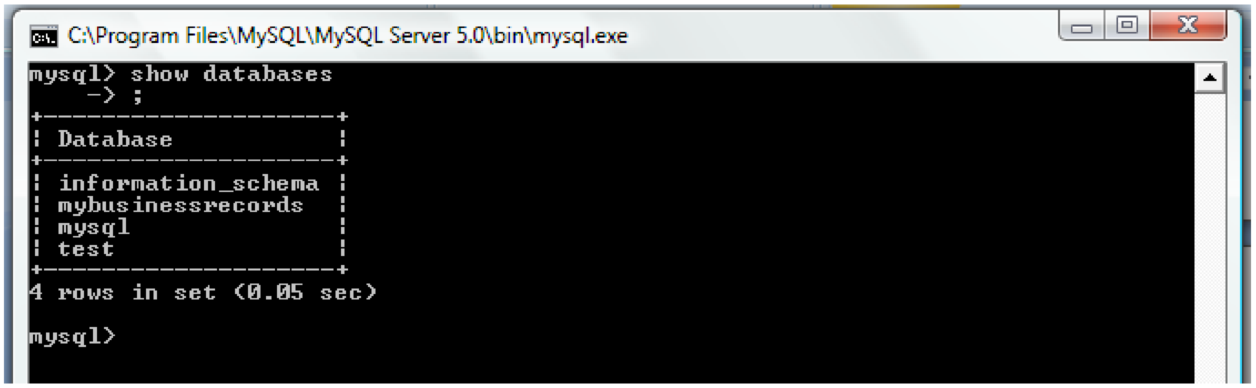
C:\Program Files\MySQL\MySQL Server 5.0\bin\mysql.exe

```
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Figure 3.1 You can access a MySQL database server from the command window.

By default, the server contains two databases named MySQL and test. You can see these two database displayed in figure using the command “show databases”.

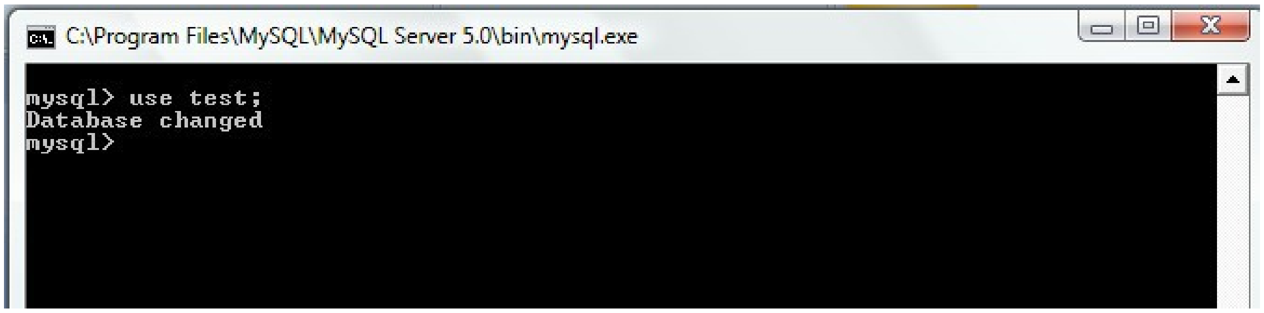
(a)



C:\Program Files\MySQL\MySQL Server 5.0\bin\mysql.exe

```
mysql> show databases
    -> ;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mybusinessrecords  |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.05 sec)

mysql>
```

(b)



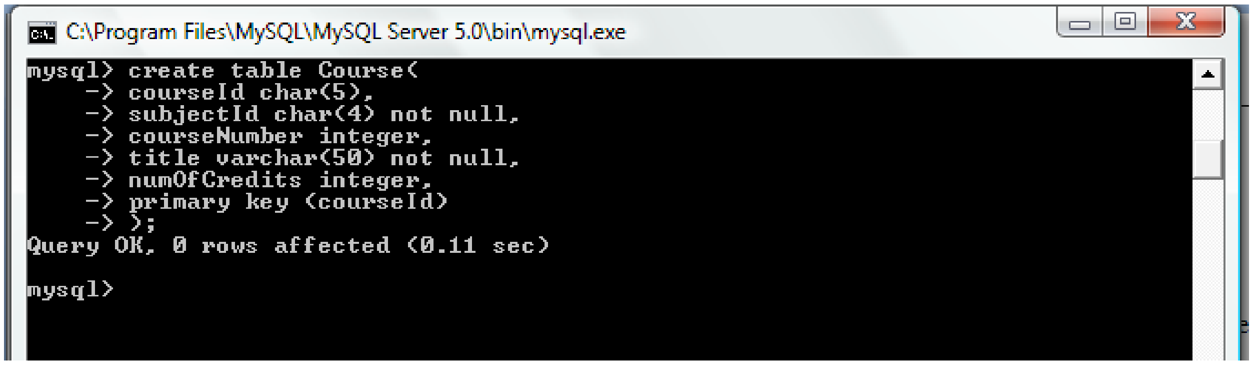
C:\Program Files\MySQL\MySQL Server 5.0\bin\mysql.exe

```
mysql> use test;
Database changed
mysql>
```

Figure 3.2 (a) The show database command display all available databases in the MySQL database server; (b) The use test command selects the test database.

The MySQL database contains the tables that store information about the server and its users. This database is intended for the server administrator to use. For example, the administrator can use it to create users and grant or revoke user privileges. Since you care the owner of the server installed on your system, you have full access to the MySQL database. However, you should not create user tables in the MySQL database. You can use the test database to store data or create new databases. You can also create a new database using the command *create database <database name>* or drop an existing database using the command *drop database <database name>.*

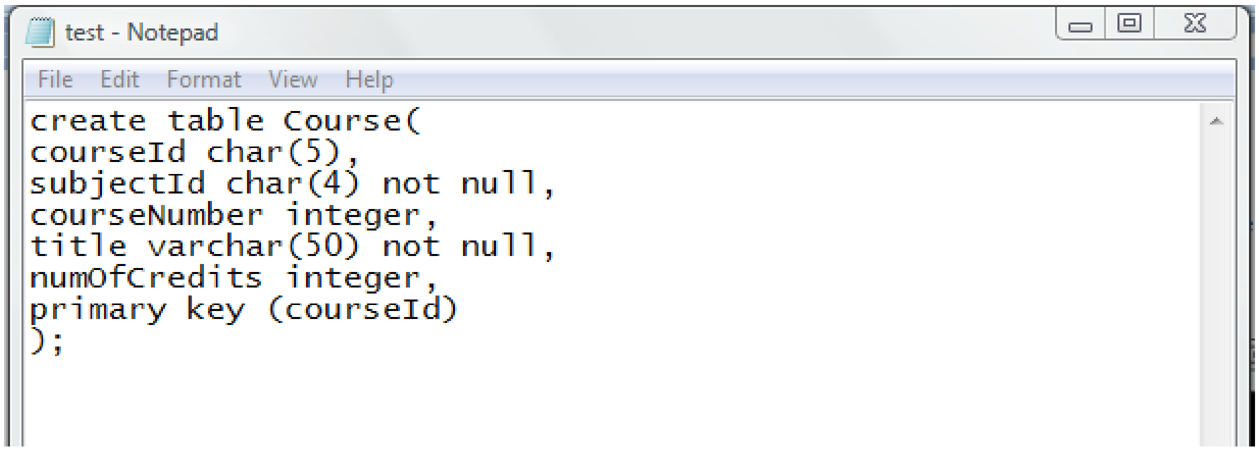
To select a database for use, type *use database* command. Since the test database is created by default in every MySQL database, let use it to demonstrate SQL commands. As shown in the figure above, the test database is selected. Enter the statement to create the course table as shown in figure below:



```
mysql> create table Course(
    -> courseId char(5),
    -> subjectId char(4) not null,
    -> courseNumber integer,
    -> title varchar(50) not null,
    -> numOfCredits integer,
    -> primary key (courseId)
    -> );
Query OK, 0 rows affected (0.11 sec)

mysql>
```

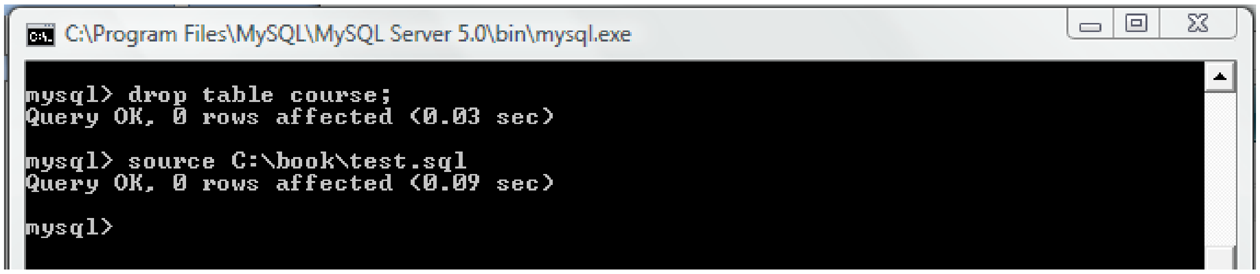
Figure 3.3 The execution result of the SQL statements is displayed in the MSQL monitor

If you make typing errors, you have to retype the whole command. To avoid retyping the whole command, you can save the command in a file, and then run the command from the file. To do so, create a text file to contain the commands, named, for example, test.sql. You can create the text file using any text editor, such as notepad, as shown in the figure below. To comment a line, proceed it with two dashes. You can now run the script file by typing source test.sql from MySQL command prompt, as shown in the figure below.



```
create table Course(
courseId char(5),
subjectId char(4) not null,
courseNumber integer,
title varchar(50) not null,
numOfCredits integer,
primary key (courseId)
);
```

Figure 3.4 You can use Notepad to create a text file for SQL commands



```
mysql> drop table course;
Query OK, 0 rows affected (0.03 sec)

mysql> source C:\book\test.sql
Query OK, 0 rows affected (0.09 sec)

mysql>
```

Figure 3.5: You can run the SQL commands in a script file from MySQL

3.4. SQL STATEMENTS

Most of the actions you need to perform on a database are done with SQL statements[i]. Some database systems require a semicolon at the end of each SQL statement. Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server. We are using MS Access and SQL Server 2000 and we do not have to put a semicolon after each SQL statement, but some database programs force you to use it. SQL statements can be divided into two parts:

1. The Data Manipulation Language (DML)
2. The Data Definition Language (DDL).

The query and update commands form the DML part of SQL:

SELECT - extracts data from a database
UPDATE - updates data in a database
DELETE - deletes data from a database
INSERT INTO - inserts new data into a database

The DDL part of SQL permits database tables to be created or deleted. It also define indexes (keys), specify links between tables, and impose constraints between tables. The most important DDL statements in SQL are:

CREATE DATABASE	-	to create a new database **ALTER**
DATABASE	-	to modify an existing database
CREATE TABLE	-	creates a new table in a database
ALTER TABLE	-	modifies an existing table in a database
DROP TABLE	-	deletes a table
CREATE INDEX	-	creates an index (search key)
DROP INDEX	-	deletes an index

The table below contains a list of SQL commands and functions:

Table 3.1: SQL Commands and Functions

SQL Basic	**SQL Advanced**	**SQL Functions**
SQLSyntax SQLSelect SQLDistinct SQL Where SQL And&Or SQL OrderBy SQLInsert SQLUpdate SQL Delete	SQLTop SQL Like SQL Wildcards SQL In SQL Between SQL Alias **SQLJoins** SQL InnerJoin SQL LeftJoin SQL RightJoin SQL FullJoin SQLUnion SQL Select Into SQL CreateDB SQL CreateTable SQLConstraints SQL NotNull SQL Unique SQL Primary Key SQL ForeignKey SQLCheck SQLDefault SQL CreateIndex SQLDrop SQLAlter SQL Increment SQL Views	SQLavg() SQLcount() SQLfirst() SQLlast() SQLmax() SQLmin() SQLsum() SQL GroupBy SQLHaving SQLucase() SQLlcase() SQLmid() SQLlen() SQLround() SQLnow() SQL format() SQL isnull()

	SQLDates SQL Nulls	

3.4.1 CREATING AND DROPPING TABLES

Tables are the essential objects in a database. To creates a table, use the create table statement to specify a table name, attributes, and types, as in the following example

```
create table Course(

courseId char(5),

subjectId char(4) not null,

courseNumber integer,

title varchar(50) not null,

numOfCredits integer,

primary key (courseId)

);
```

This statement creates the course table with attributes courseId, subjectId, courseNumber, title and numOfCredits. Each attribute has a data type that specifies the type of data stored in the attribute. char(5) specifies that courseId consists of five characters. varchar(50) specifies that title is a variant-length string with a maximum of fifty characters. Integer specifies that courseNumber is an integer. The primary key is courseId.

The table Student and Enrollment can be created as follows:

```
create table Student (

ssn char(9)

firstName varchar(5),

mi char (1)

lastName varchar (25)

birthDate date,

street varchar (25),

phone char(11)
```

```
zipCode char (5),
deptId char(4),
primary key (ssn)
);
create table Enrollment (
ssn char(9), courseId
char(5)
dateRegistered date,
grade char (10,
primary key (ssn, courseId)
foreign key (ssn) references student,
foreign key (courseId) references Course
);
```

If a table is no longer needed, it can be dropped permanently using the drop table command. For example, the following statements drops the Course table:

```
drop table Course;
```

If a table to be dropped is referenced by other tables, you have to drop other tables first. For example, if you have created the tables Course, Student and Enrollment and want to drop Course, you have to first drop Enrollment, because Course is referenced by Enrollment.

3.4.2 THE SQL SELECT STATEMENT

The SELECT statement is used to select data from a database. Depending on the form it

takes, it searches through the table in the database and selects the data that matches the

criteria. The result is stored in a result table, called the result-set.

SQL SELECT Syntax

```
SELECT column_name(s)

FROM table_name

and

SELECT * FROM table_name
```

Note: SQL is not case sensitive. SELECT is the same as select.

An SQL SELECT Example

The "Persons" table:

Table 3.2: The "Persons" table

P_Id	LastName	FirstName	Address		City
1	Hansen	Ola	Timoteivn 10		Sandnes
2	Svendson	Tove	Borgvn 23		Sandnes
3	Pettersen	Kari	Storgt 20		Stavanger

Now we want to select the content of the columns named "LastName" and "FirstName"

from the table above. We use the following SELECT statement: SELECT LastName, FirstName FROM Persons.

The result-set will look like this:

LastName	FirstName
Hansen	Ola
Svendson	Tove
Pettersen	Kari

SELECT * Example

Now we want to select all the columns from the "Persons" table.

We use the following SELECT statement: SELECT * FROM Persons

Tip: The asterisk (*) is a quick way of selecting all columns!

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

3.4.3 The SQL SELECT DISTINCT Statement

In a table, some of the columns may contain duplicate values. This is not a problem,

however, sometimes you will want to list only the different (distinct) values in a table.

The DISTINCT keyword can be used to return only distinct (different) values.

SQL SELECT DISTINCT Syntax

```
SELECT DISTINCT column_name(s)

FROM table_name
```

SELECT DISTINCT Example

Now we want to select only the distinct values from the column named "City" from the

table above.

We use the following SELECT statement: SELECT DISTINCT City FROM Persons

The result-set will look like this:

City
Sandnes
Stavanger

The WHERE clause is used to filter records.

3.4.4 The WHERE Clause

The WHERE clause is used to extract only those records that fulfill a specified criterion.

SQL WHERE Syntax

```
SELECTcolumn_name(s)
FROMtable_name
WHERE column_name operator value
```

WHERE Clause Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to select only the persons living in the city "Sandnes" from the table above.

We use the following :

```
SELECT * FROM Persons
WHERE City='Sandnes'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

3.4.5 Quotes Around Text Fields

SQL uses single quotes around text values (most database systems will also accept double

quotes). Although, numeric values should not be enclosed in quotes.

For text values:
This is correct:

SELECT * FROM Persons WHERE FirstName='Tove'

This is wrong:

SELECT * FROM Persons WHERE FirstName=Tove

For numeric values:

This is correct:

SELECT * FROM Persons WHERE Year='1965'

This is wrong:

```
        SELECT * FROM Persons WHERE Year=1965
```

3.4.6 Operators Allowed in the WHERE Clause

With the WHERE clause, the following operators can be used:

Operator	Description
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	If you know the exact value you want to return for at least one of the columns

Note: In some versions of SQL the <> operator may be written as !=

The AND & OR operators are used to filter records based on more than one condition.

3.4.7. The AND & OR Operators

The AND operator displays a record if both the first condition and the second condition is true.

The OR operator displays a record if either the first condition or the second condition is true.

AND Operator Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to select only the persons with the first name equal to "Tove" AND the last name equal to "Svendson":

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE FirstName='Tove'

AND LastName='Svendson'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
2	Svendson	Tove	Borgvn 23	Sandnes

OR OPERATOR EXAMPLE

Now we want to select only the persons with the first name equal to "Tove" OR the first name equal to "Ola":

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE FirstName='Tove'

OR FirstName='Ola'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

COMBINING AND & OR

You can also combine AND and OR (use parenthesis to form complex expressions).

Now we want to select only the persons with the last name equal to "Svendson" AND the first name equal to "Tove" OR to "Ola":

We use the following SELECT statement:

```
SELECT * FROM Persons WHERE

LastName='Svendson'

AND (FirstName='Tove' OR FirstName='Ola')
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
2	Svendson	Tove	Borgvn 23	Sandnes

3.4.8. The ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set. The result is sorted against a specified column and is in ascending order by default. To sort the records in a descending order, we can use the DESC keyword.

```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s) ASC|DESC
```

THE ORDER BY Syntax

ORDER BY Example

```
SELECT * FROM Persons
```

Now we want to select all the persons from the “Persons” table above, however, we want to sort the persons by their last name.

We use the following SELECT statement:

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
4	Nilsen	Tom	Vingvn 23	Stavanger
3	Pettersen	Kari	Storgt 20	Stavanger
2	Svendson	Tove	Borgvn 23	Sandnes

ORDER BY DESC Example

Now we want to select all the persons from the table above, however, we want to sort the

persons descending by their last name.

```
SELECT * FROM Persons

ORDER BY LastName DESC
```

We use the following SELECT statement:

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Tom	Vingvn 23	Stavanger
1	Hansen	Ola	Timoteivn 10	Sandnes

3.4.9. The INSERT INTO Statement

The INSERT INTO statement is used to insert a new row in a table.

THE INSERT INTO Syntax

It is posssible to write the INSERT INTO statement in two forms.

```
INSERT INTO table_name

VALUES (value1, value2, value3,...)
```

The first form doesn't specify the column names where the data will be inserted, only their

values:

```
INSERT INTO table_name (column1, column2, column3,...)

VALUES (value1, value2, value3,...)
```

The second form specifies both the column names and the values to be inserted:

INSERT INTO Example

We have the following "Persons" table:

P_Id	LastName	FirstName	Address		City
1	Hansen	Ola	Timoteivn 10		Sandnes
2	Svendson	Tove	Borgvn 23		Sandnes
3	Pettersen	Kari	Storgt 20		Stavanger

```
INSERT INTO Persons

VALUES (4,'Nilsen', 'Johan', 'Bakken 2', 'Stavanger')
```

Now we want to insert a new row in the "Persons" table.

We use the following SQL statement:

The "Persons" table will now look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger

Insert Data Only in Specified Columns

It is also possible to only add data in specific columns.

```
INSERT INTO Persons (P_Id, LastName, FirstName)

VALUES (5, 'Tjessem', 'Jakob')
```

The following SQL statement will add a new row, but only add data in the "P_Id",

"LastName" and the "FirstName" columns:

The "Persons" table will now look like this:

P_Id	**LastName**	**FirstName**	**Address**	**City**
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger
5	Tjessem	Jakob		

3.4.10. The UPDATE Statement

The UPDATE statement is used to update existing records in a table.

```
UPDATE table_name

SET column1=value, column2=value2,...

WHERE some_column=some_value
```

THE UPDATE Syntax

Note: Notice the WHERE clause in the UPDATE syntax. The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated!

SQL UPDATE Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger
5	Tjessem	Jakob		

```
UPDATE Persons

SET Address='Nissestien 67', City='Sandnes'

WHERE LastName='Tjessem' AND FirstName='Jakob'
```

Now we want to update the person "Tjessem, Jakob" in the "Persons" table.

We use the following SQL statement:

The "Persons" table will now look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

4	Nilsen	Johan	Bakken 2	Stavanger
5	Tjessem	Jakob	Nissestien 67	Sandnes

UPDATE Warning

```
UPDATE Persons

SET Address='Nissestien 67', City='Sandnes'
```

Be careful when updating records. If we had omitted the WHERE clause in the example

above, like this:

The "Persons" table would have looked like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Nissestien 67	Sandnes
2	Svendson	Tove	Nissestien 67	Sandnes
3	Pettersen	Kari	Nissestien 67	Sandnes
4	Nilsen	Johan	Nissestien 67	Sandnes
5	Tjessem	Jakob	Nissestien 67	Sandnes

3.4.11. The DELETE Statement

The DELETE statement is used to delete rows in a table.

```
DELETE FROM table_name

WHERE some_column=some_value
```

THE DELETE Syntax

Note: Notice the WHERE clause in the DELETE syntax. The WHERE clause specifies which record or records that should be deleted. If you omit the WHERE clause, all records will be deleted!

SQL DELETE Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger
5	Tjessem	Jakob	Nissestien 67	Sandnes

```
DELETE FROM Persons

WHERE LastName='Tjessem' AND FirstName='Jakob
```

Now we want to delete the person "Tjessem, Jakob" in the "Persons" table.

We use the following SQL statement:

The "Persons" table will now look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger

Delete All Rows

```
DELETE FROM table_name

or

DELETE * FROM table_name
```

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

Note: Be very careful when deleting records. You cannot undo this statement!

3.4.12. The TOP Clause

The TOP clause is used to specify the number of records to return.

The TOP clause can be very useful on large tables with thousands of records. Returning a large number of records can impact on performance.

Note: Not all database systems support the TOP clause.

```
SELECT TOP number|percent column_name(s)

FROM table_name
```

THE Server Syntax

SQL SELECT TOP Equivalent in MySQL and Oracle

```
SELECT column_name(s)

FROM table_name

LIMIT number
```

MySQL Syntax

Example

```
SELECT *
FROM Persons
LIMIT 5
```

Oracle Syntax

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number
```

Example

```
SELECT *

FROM Persons

WHERE ROWNUM <=5
```

SQL TOP Example

The "Persons" table:

P_Id	**LastName**	**FirstName**	**Address**	**City**
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Tom	Vingvn 23	Stavanger

```
SELECT TOP 2 * FROM Persons
```

Now we want to select only the two first records in the table above.

We use the following SELECT statement:

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

SQL TOP PERCENT Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Tom	Vingvn 23	Stavanger

Now we want to select only 50% of the records in the table above.

```
SELECT TOP 50 PERCENT * FROM Persons
```

We use the following SELECT statement:

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

3.4.13. The LIKE Operator

The LIKE operator is used to search for a specified pattern in a column. It is used in a WHERE clause to search for the specified pattern in the column.

```
SELECT column_name(s)

FROM table_name

WHERE column_name LIKE pattern
```

The SQL LIKE Syntax

LIKE Operator Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to select the persons living in a city that starts with "s" from the table above.

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE City LIKE 's%'
```

The "%" sign can be used to define wildcards (missing letters in the pattern) both before and after the pattern.

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Next, we want to select the persons living in a city that ends with an "s" from the "Persons" table.

```
SELECT * FROM Persons

WHERE City LIKE '%s'
```

We use the following SELECT statement:

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

Next, we want to select the persons living in a city that contains the pattern "tav" from the

"Persons" table.

```
SELECT * FROM Persons

WHERE City LIKE '%tav%'
```

We use the following SELECT statement:

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
3	Pettersen	Kari	Storgt 20	Stavanger

It is also possible to select the persons living in a city that NOT contains the pattern "tav"

from the "Persons" table, by using the NOT keyword.

```
SELECT * FROM Persons

WHERE City NOT LIKE '%tav%'
```

We use the following SELECT statement:

The result-set will look like this:

P_Id	LastName	FirstName	Address	City

1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

3.4.14. SQL Wildcards

SQL wildcards can be used when searching for data in a database. They can substitute for one or more characters when searching for data in a database and they must be used with the LIKE operator.

With SQL, the following wildcards can be used:

Wildcard	**Description**
%	A substitute for zero or more characters
_	A substitute for exactly one character
[charlist]	Any single character in charlist
[^charlist] or[!charlist]	Any single character not in charlist

SQL Wildcard Examples

We have the following "Persons" table:

P_Id	**LastName**	**FirstName**	**Address**	**City**
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Using the '%' Wildcard

Now we want to select the persons living in a city that starts with "sa" from the "Persons" table.

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE City LIKE 'sa%'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

Next, we want to select the persons living in a city that contains the pattern "nes" from the "Persons" table.

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE City LIKE '%nes%'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes

Using the '_' Wildcard

Now we want to select the persons with a first name that starts with any character, followed by "la" from the "Persons" table.

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE FirstName LIKE '_la'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

Next, we want to select the persons with a last name that starts with "S", followed by any character, followed by "end", followed by any character, followed by "on" from the "Persons" table.

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE LastName LIKE 'S_end_on'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
2	Svendson	Tove	Borgvn 23	Sandnes

Using the [charlist] Wildcard

Now we want to select the persons with a last name that starts with "b" or "s" or "p" from the "Persons" table.

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE LastName LIKE '[bsp]%'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Next, we want to select the persons with a last name that do not start with "b" or "s" or "p" from the "Persons" table.

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE LastName LIKE '[!bsp]%'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

3.4.15. The IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

```
SELECT column_name(s)

FROM table_name

WHERE column_name IN (value1,value2,...)
```

THE IN Syntax

IN Operator Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to select the persons with a last name equal to "Hansen" or "Pettersen" from the table above.

We use the following SELECT statement:

```
SELECT * FROM Persons

WHERE LastName IN ('Hansen','Pettersen')
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

3.4.16. The BETWEEN Operator

The BETWEEN operator selects a range of data between two values. The values can be numbers, text, or dates. It is usually added to a WHERE clause to select data between the a range specified

```
SELECT column_name(s)
FROM table_name
WHERE column_name
BETWEEN value1 AND value2
```

The SQL BETWEEN Syntax

BETWEEN Operator Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to select the persons with a last name alphabetically between "Hansen" and

"Pettersen" from the table above.

```
SELECT * FROM Persons

WHERE LastName

BETWEEN 'Hansen' AND 'Pettersen'
```

We use the following SELECT statement:

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

Note: The BETWEEN operator is treated differently in different databases.

In some databases, persons with the LastName of "Hansen" or "Pettersen" will not be listed, because the BETWEEN operator only selects fields that are between and excluding the test values).

In other databases, persons with the LastName of "Hansen" or "Pettersen" will be listed, because the BETWEEN operator selects fields that are between and including the test values).

And in other databases, persons with the LastName of "Hansen" will be listed, but

"Pettersen" will not be listed (like the example above), because the BETWEEN operator

selects fields between the test values, including the first test value and excluding the last test value. Therefore, Check how your database treats the BETWEEN operator.

Example 2

```
SELECT * FROM Persons

WHERE LastName

NOT BETWEEN 'Hansen' AND 'Pettersen'
```

To display the persons outside the range in the previous example, use NOT BETWEEN:

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

3.4.17. SQL Alias

You can give a table or a column another name by using an alias. This can be a good thing to do if you have very long or complex table names or column names.
An alias name could be anything, but usually, it is short.

```
SELECT column_name(s)

FROM table_name

AS alias_name
```

SQL Alias Syntax for Tables

SQL Alias Syntax for Columns

```
SELECT column_name AS alias_name

FROM table_name
```

Alias Example

Assume we have a table called "Persons" and another table called "Product_Orders". We will give the table aliases of "p" an "po" respectively.

Now we want to list all the orders that "Ola Hansen" is responsible for.

```
SELECT po.OrderID, p.LastName, p.FirstName

FROM Persons AS p,

Product_Orders AS po

WHERE p.LastName='Hansen'

WHERE p.FirstName='Ola'
```

We use the following SELECT statement:

The same SELECT statement without aliases:

```
SELECT Product_Orders.OrderID, Persons.LastName, Persons.FirstName

FROM Persons,

Product_Orders

WHERE Persons.LastName='Hansen'

WHERE Persons.FirstName='Ola'
```

As you'll see from the two SELECT statements above; aliases can make queries easier to both write and to read.

3.4.18. SQL JOIN

The JOIN keyword is used in an SQL statement to query data from two or more tables, based on a relationship between certain columns in these tables. Tables in a database are often related to each other with keys. A primary key is a column (or a combination of columns) with a unique value for each row. Each primary key value must be unique within the table. The purpose is to bind data together, across tables, without repeating all of the data in every table.

Look at the "Persons" table:

P_Id	**LastName**	**FirstName**	**Address**	**City**
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Note that the "P_Id" column is the primary key in the "Persons" table. This means that **no** two rows can have the same P_Id. The P_Id distinguishes two persons even if they have the same name.

Next, we have the "Orders" table:

Table 3.4: The “Orders”

O_Id	**OrderNo**	**P_Id**
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	15

Note that the "O_Id" column is the primary key in the "Orders" table and that the "P_Id" column refers to the persons in the "Persons" table without using their names.

Notice that the relationship between the two tables above is the "P_Id" column.

Different SQL JOINs

Before we continue with examples, we will list the types of JOIN you can use, and the differences between them.

JOIN: Return rows when there is at least one match in both tables
LEFT JOIN: Return all rows from the left table, even if there are no matches in the right table
RIGHT JOIN: Return all rows from the right table, even if there are no matches in the left table
FULL JOIN: Return rows when there is a match in one of the tables

The FULL JOIN Keyword
The FULL JOIN keyword return rows when there is a match in one of the tables.

```
SELECT column_name(s)
FROM table_name1
FULL JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

SQL FULL JOIN Syntax

SQL FULL JOIN Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

The "Orders" table:

O_Id	OrderNo	P_Id
1	77895	3

2	44678	3
3	22456	1
4	24562	1
5	34764	15

Now we want to list all the persons and their orders, and all the orders with their persons.

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo

FROM Persons

FULL JOIN Orders

ON Persons.P_Id=Orders.P_Id

ORDER BY Persons.LastName
```

We use the following SELECT statement:

The result-set will look like this:

LastName	FirstName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562

Pettersen	Kari	77895
Pettersen	Kari	44678
Svendson	Tove	
		34764

The FULL JOIN keyword returns all the rows from the left table (Persons), and all the rows

from the right table (Orders). If there are rows in "Persons" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Persons", those rows will be listed as well.

3.4.19. The AVG() Function
The AVG() function returns the average value of a numeric column.

```
SELECT AVG(column_name) FROM table_name
```

SQL AVG() Syntax

SQL AVG() Example

We have the following "Orders" table:

O_Id	OrderDate	OrderPrice	Customer
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

Now we want to find the average value of the "OrderPrice" fields.

```
SELECT AVG(OrderPrice) AS OrderAverage FROM Orders
```

We use the following SQL statement:

The result-set will look like this:

OrderAverage
950

Now we want to find the customers that have an OrderPrice value higher than the average OrderPrice value.

```
SELECT Customer FROM Orders
WHERE OrderPrice>(SELECT AVG(OrderPrice) FROM Orders)
```

We use the following SQL statement:

The result-set will look like this:

Customer
Hansen
Nilsen
Jensen

3.4.20. The COUNT() Function

The COUNT() function returns the number of rows that matches a specified criteria.

The COUNT(column_name) Syntax

```
SELECT COUNT(column_name) FROM table_name
```

The COUNT(column_name) function returns the number of values (NULL values will not be counted) of the specified column:

SQL COUNT(*) Syntax

```
SELECT COUNT(*) FROM table_name
```

The COUNT(*) function returns the number of records in a table:

SQL COUNT(DISTINCT column_name) Syntax

```
SELECT COUNT(DISTINCT column_name) FROM table_name
```

The COUNT(DISTINCT column_name) function returns the number of distinct values of the

specified column:

Note: COUNT(DISTINCT) works with ORACLE and Microsoft SQL Server, but not with Microsoft Access.

SQL COUNT(column_name) Example

We have the following "Orders" table:

O_Id	OrderDate	OrderPrice	Customer
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen

3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

Now we want to count the number of orders from "Customer Nilsen".

```
SELECT COUNT(Customer) AS CustomerNilsen FROM Orders

WHERE Customer='Nilsen'
```

We use the following SQL statement:

The result of the SQL statement above will be 2, because the customer Nilsen has made 2 orders in total:

CustomerNilsen
2

SQL COUNT(*) Example

```
SELECT COUNT(*) AS NumberOfOrders FROM Orders
```

If we omit the WHERE clause, like this:

The result-set will look like this:

NumberOfOrders
6

which is the total number of rows in the table.
SQL COUNT(DISTINCT column_name) Example

Now we want to count the number of unique customers in the "Orders" table.

```
SELECT COUNT(DISTINCT Customer) AS NumberOfCustomers FROM Orders
```

We use the following SQL statement:

The result-set will look like this:

NumberOfCustomers
3

which is the number of unique customers (Hansen, Nilsen, and Jensen) in the "Orders" table.

3.4.21. The MAX() Function

The MAX() function returns the largest value of the selected column.

```
SELECT MAX(column_name) FROM table_name
```

The MAX() Syntax

SQL MAX() Example

We have the following "Orders" table:

O_Id	OrderDate	OrderPrice	Customer
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

Now we want to find the largest value of the "OrderPrice" column.

```
SELECT MAX(OrderPrice) AS LargestOrderPrice FROM Orders
```

We use the following SQL statement:

The result-set will look like this:

LargestOrderPrice
2000

3.4.22. The UCASE() Function

The UCASE() function converts the value of a field to uppercase.

```
SELECT UCASE(column_name) FROM table_name
```

The UCASE() Syntax

```
SELECT UPPER(column_name) FROM table_name
```

Syntax for SQL Server

SQL UCASE() Example

We have the following "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to select the content of the "LastName" and "FirstName" columns above, and

convert the "LastName" column to uppercase.

```
SELECT UCASE(LastName) as LastName,FirstName FROM Persons
```

We use the following SELECT statement:

The result-set will look like this:

LastName	**FirstName**
HANSEN	Ola
SVENDSON	Tove
PETTERSE N	Kari

3.4.23. The SUM() Function

The SUM() function returns the total sum of a numeric column.

```
SELECT SUM(column_name) FROM table_name
```

SQL SUM() Syntax

SQL SUM() Example
We have the following "Orders" table:

O_Id	OrderDate	OrderPrice	Customer
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

Now we want to find the sum of all "OrderPrice" fields".

```
SELECT SUM(OrderPrice) AS OrderTotal FROM Orders
```

We use the following SQL statement:

The result-set will look like this:

OrderTotal
5700

3.5. CONCLUSION

The structured Query language (SQL) is composed of about 30 commands that enable users to create database and table structures, perform various types of data manipulation, and query the database to extract useful information. Almost all RDBMS software supports SQL, many software vendors have developed extensions to the basic SQL command set.

3.6. SUMMARY

SQL - Structured Query Language is a standard language for accessing and manipulating databases. SQL lets you access and manipulate databases. The Structured Query Language can be used to execute queries against a database, retrieve data from a database, insert records in a database, update records in a database, delete records from a database, create new databases, create new tables in a database, create stored procedures in a database, create views in a database, and set permissions on tables, procedures, and views among other things. It could also be used when creating a web application i.e. a build web site that shows some data from a database.

3.7. TUTOR MARKED ASSIGNMENT

1. How do the relations (tables) in SQL differ from the relations defined formally in relational data model? Discuss the other differences in terminology. Why does SQL allow duplicate tuples in a table or in a query result?

2. List the data types that are allowed for SQL attributes.

3. What is a view in SQL, and how is it defined? Discuss the problems that may arise when one attempts to update a view. How are views typically implemented?

4. Describe the six clauses in the syntax of an SQL query, and show what type of constructs can be specified in each of the six clauses. Which of the six clauses are required and which are optional?

5. Choose some database application that you are familiar with

   (A) Design a relational database schema for your database application

   (b) Declare your relations, using the SQL DDL

   Specify a number of queries in SQL that are needed by your database application

   (d) Based on your expected use of the database, chosen some attributes that should have indexes specified on them.

   (e) Implement your database, if you have a DBMS that supports SQL.

3.8. FURTHER READINGS

Chamberlin D., and Boyce, R. [1984] "SEQUEL: A Structured English Query Language", in SIGMOD [1984].

Date, C [1983] "The Outer Join, "Proceedings of the Second International Conference on Database (ICOD -2), 1983.

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[1986]. "Database System Concepts", McGraw – hill, 1986.

www.w3schools.com

MODULE 1: DATABASE DESIGN AND IMPLEMENTATION

UNIT 4: DATABASE SYSTEM CATALOGUE

4.0.INTRODUCTION.

The catalogue is the place where among other things, all of the various schemas (external, conceptual, internal) and all of the corresponding mappings (external/conceptual, conceptual/internal) are kept. In other words, the catalogue contains detailed information (sometimes call descriptor information or metadata) regarding the various objects that are of interest to the system itself.

4.1.OBJECTIVES.

To understand the description of a database catalogue system

4.2.WHATISDATABASESYSTEMCATALOGUE ?

As explained earlier, every DBMS must provide a catalogue or dictionary function. The catalogue is the place where among other things, all of the various schemas (external, conceptual, internal) and all of the corresponding mappings (external/conceptual, conceptual/internal) are kept. In other words, the catalogue contains detailed information (sometimes call descriptor information or metadata) regarding the various objects that are of interest to the system itself. Examples of such objects are relvars, indexes, users integrity constraints, security constraints, and so on and so forth. Descriptor information is essential if the system is to do its job properly. For example, the optimizer uses catalogue information about indexes and other physical storage structures, as well as much other information, to help it decided how to implement user request. Likewise, the security subsystem uses catalogue information about seers and security contains to grant or deny such requests in the first place.

Now, one of the nice features of relational systems is that, in such a system, the catalogue itself consists of relvars (more precisely, system relvars, so called to distinguish them from ordinary user ones). As a result, users can interrogate the catalogue in exactly the same way they interrogate their own data. For example, the catalogue will typically include two system relvars called TABLE and COLUMN, the purpose of which is to describe the tables (i.e. relvars) in the database and the columns in those tables (We say “typically” because the catalogue is not the same in every system; the difference arise because the catalogue for a particular system necessarily contains a good deal of information that is specific to that system) for the departments and employees database.

The catalogue should normally be self-describing i.e. it should include entries describing the catalogue relvars themselves. Now suppose some user of the departments and employees database want to know exactly what columns relvar DEPT contains (obviously we are assuming that for some reason the user does not already have this information). Then the expression

[COLUMN WHERE TABNAME = DEPT] [COLMNAME]

does the job. Note, if we had wanted to keep the result of this query in some more permanent fashion, we could have assigned the value of the expression to some other relvar.

Here is another example: “which relvars includes a column called EMP#?

(COLUMN WHERE COLNAME = ‘EMP#) {TABNAME}

4.3.CONCLUSION.

The catalogue should normally be self-describing i.e. it should include entries describing the catalogue relvars themselves.

4.4.SUMMARY.

DBMS must provide a catalogue or dictionary function. The catalogue is the place where among other things, all of the various schemas (external, conceptual, internal) and all of the corresponding mappings (external/conceptual, conceptual/internal) are kept.

4.5.TUTOR-MARKEDASSIGNMENT(TMA).

What is a database catalogue system?

4.6.FURTHERREADINGS

Date, C. [2000] An Introduction Database System, 7th ed.; Addison-Wesley 2000.

Atzeni, P., and De Antonellis, V. [1993] Relational Database Theory Benjamin / Cummings, 1993.

Batini., C., Ceri, S., and Navathe, S. [1992] Database Design: An Entity – Relationship Approach, Benjamin/Cummings, 1992.

Bernstein, P. [1976] "Synthesizing Third Normal Form Relations from Functional Dependencies", TODS, 1:4, December 1976.

Codd, E [1970] "A Relational Model for Large Shared Data BANks" CACM, 136, June 1970.

MODULE 2: ADVANCED FEATURES OF DBMS

UNIT 1: QUERY PROCESSING & EVALUATION

1.0 INTRODUCTION

In the preceding units, we have considered how to structure the data in the database. These decisions are made at the time the database is designed. Although it is possible to change this structure, it is relatively costly to do so. Thus, when a query is presented to the system, it is necessary to find the best method of finding the answer using the existing database structure.

There are large numbers of possible strategies for processing a query, especially if the query is complex. Nevertheless, it is usually worthwhile for the system to spend a substantial amount of time on the selection of a strategy. Typically, strategy selection can be done using information available in main memory, with little or no disk accesses. The actual execution of the query will involve many accesses to disk. Since the transfer of data from disk is slow relative to the speed of main memory and the central processor of the computer system, it is advantageous to spend a considerable amount of processing to save disk accesses.

1.1. OBJECTIVES

To understand the basics of query optimization

Ability to estimate costs involved in query processing

1.2 QUERY INTERPRETATION

Given a query, there are generally a variety of methods for computing the answer. For example, we saw that in SQL a query could be expressed in several different ways. Each way of expressing the query “suggests” a strategy for finding the answer. However, we do not expect users to write their queries in a way that suggests the most efficient strategy. Thus, it becomes the responsibility of the system to transform the query as entered by the user into an equivalent query which can be computed more efficiently. This “optimizing” or more accurately, improving of the strategy for processing a query, is called query optimization. There is a close analogy between code optimization by a compiler and query optimization by a database system. We shall study the issues involved in efficient query processing both in high – level language and at the level of the physical access to the data.

Query optimization is an important issue in any database system since the difference in execution time between a good strategy and a bad one may be huge. In the network model and the hierarchical model, query optimization is left, for the most part, to the application programmer. Since the data manipulation language statements are embedded in a host programming language, it is not easy to transform a network or hierarchical query to an equivalent one unless one has knowledge about the entire application program.

Since a relational query can be expressed entirely in a relational query language without the use of host language, it is possible to optimize queries automatically. Since the most useful optimization techniques apply of the relational model, we shall emphasize the relational model in this module.

Before query processing can begin, the system must translate the query into a usable form. Language such as SQL are suitable for human use, but ill-suited to be the system's internal representation of a query. A more useful internal representation of query is one based on the relational algebra.

Thus, the first action the system must take on a query is to translate the query into its internal form. This translation process is similar to that done by the parser of a compiler. In the process of generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of relation in the database, etc. if the query was expressed in terms of a view, the parser replaces all reference to the view name with the relational algebra expression to compute a view.

The details of the parser are beyond the scope of this text.

Once the query has been translated to an internal relational algebra form, the optimization process begins. The first phase of optimization is done at the relational algebra level. An attempt is made to find an expression that is equivalent to the given expression but that is more efficient to execute. The next phase involves the selection of a detailed how the query will be executed. A choice of specific indices to use must be made. The order in which tuples are processed must be determined. The final choice of a strategy is based primarily on the number of disk accesses required.

1.3 EQUIVALENCE OF EXPRESSIONS

The relational algebra is a procedural language. Thus, each relational algebra expression represents a particular sequence of operations. We have already seen that there are several ways to express a given query in the relational algebra. The first step in selecting a query processing strategy is to find a relational algebra expression that is equivalent to the given query and is efficient to execute.

We use our bank example to illustrate optimization techniques. In particular, we shall use the relations customer *(customer-scheme),* deposit *(deposit-scheme)*, and branch *(branch-scheme).* As was the case earlier, we define our relation scheme as follows;

$$- = (-,, -)$$

$$- = (-, -, -,)$$

$$- = (-,, -)$$

1.3.1. Selection Operation.

Let us consider the relational algebra expression we wrote earlier for the query "find the assets and name of all banks who have depositors living in Port Chester"

$$\Pi_{-,}(_{--''''}$$

$$(\bowtie \quad \bowtie))$$

This expression constructs a large relation, $\bowtie \quad \bowtie$. However, we are interested in only a few tuples of this relation [those pertaining to residents of Port Chester], and in only two of the eight attributes of this relation. The large intermediate result:

$$\bowtie \quad \bowtie$$

is probably too large to be kept in main memory and thus must be stored on disk. This means that in addition to the disk accesses required to read the relations *customer*, *deposit*, and *branch*, the system will need to access disk to read and write intermediate results. Clearly, we could process the query more efficiently if there were a way to reduce the size of the intermediate result.

Since we are concerned only about tuples for which *customer-city* = Port Chester", we need not consider those tuples of the customer relation that do not have *customer-city* = "Port Chester". By reducing the number of tuples of the customer relation that we need to access, we reduce the size of the intermediate result. Our query is now represented by the relational algebra expression:

$$\Pi_{-,}((_{-=''''}()))$$

$$\bowtie \quad \bowtie)$$

The above example suggests the following rule for transforming relational algebra queries:

Perform selection operations as early as possible

In our example, we recognized that the selection operator pertained only to the customer relation, so we performed the selection on customers directly. Suppose that we modify our original query to restrict attention to customers with a balance over ₦1000. the new relational algebra query is

$$\Pi_{-,}(_{-=\text{Port Chester} \wedge >1000}$$

$$(\bowtie \quad \bowtie))$$

We cannot apply the selection

$$- \quad = \quad \text{“”} \wedge \quad > \quad 1000$$

Directly to the customer relation, since the predicate involves attributes of customer and deposit. However, the branch relation does not involve either customer-city or balance. If we decide to process the join as:

$$(\bowtie \quad \bowtie)$$

Then we can rewrite our query as:

$$\Pi_{-,}((_{-=\text{Port Chester} \wedge >1000}(\bowtie))$$

$$\bowtie)$$

Let us examine the sub query:

$$_{-=\text{“”} \quad >1000}( )$$

We can split the selection predicate into two, forming the expression:

$$_{-=\text{“”}}(_{>1000}(\bowtie))$$

Both of the above expressions select tuples with *customer-city* = “Port Chester” and *balance* > 1000. However, the latter form of the expression provides a new opportunity to apply the "perform selections early" rule. We now rewrite our query as:

$$(_{-=\text{“”}}()) \bowtie (_{>1000}())$$

We now add a second transformation rule:

Replace expressions of the form:

$$_{1 \wedge 2}()$$

by

$_1(_2(0))$

where $_1$ and $_2$ are predicates and

is a relational algebra expression.

An easy way to remember this transformation is by nothing the following equivalences among relational algebra expression:

$_1(_2(0)) = {}_1(_2(0)) = {}_{1\wedge 2}(0)$

1.3.2. Natural Join Operations.

By modifying queries so that selections are done early, we reduce the size of temporary results. Another way to reduce the size of temporary results is to choose an optimal ordering of the *join* operations. We mentioned in earlier that *natural join* is associative. Thus, for all relations

$_1, _2, _3$

$(_1 \bowtie {}_2) \bowtie {}_3 = {}_1 \bowtie (_2 \bowtie {}_3)$

Although these expressions are equivalent, the costs of computing them may differ. Consider again the expression:

$\Pi_{-,}((_{-=\cdots}(0))$

$\bowtie \quad \bowtie)$

We could choose to compute $\bowtie$ first and then join the result with:

$_{-=\cdots}(0)$

However, $\bowtie$ is likely to be a large relation since it contains one tuple for every account. However,

$_{-=\cdots}(0)$

is probably a small relation. To see this, note that since the bank has a large number of widely distributed branches, it is likely that only a small fraction of the bank's customers live in Port Chester. If we compute:

$(_{-=\cdots}(0)) \bowtie$

First, we obtain one tuple for each account held by a resident of Port Chester. Thus, the temporary relation we must store is smaller than if we computer deposit x borrow first.,

There are other options to consider for evaluating our query. We do not care about the order in which attributes appear in a join, since it is easy to change the order before displaying the result. Thus, for all relations $_1$ $_2$.

$$_1 \bowtie _2 = _2 \bowtie _1$$

That is, natural join is commutative.

Using this fact, we can consider rewriting our relational algebra expression as

$$\Pi_{-,}(((_{-=} ()) \bowtie) \bowtie)$$

That is, we could join $_{-=}()$ with *branch* as the first join operation performed. Note, however, that there are no attributes in common between *Branch-scheme* and *Customer-scheme*, so the join is really just a Cartesian product. If there are *c* customers in Port Chester and *b* branches, this Cartesian product generates

tuples, one for every possible pair of customers and branches (without regard for whether or not the customer has an account at the branch). Thus, it appears that this Cartesian product will produce a large temporary relation. As a result, we would reject this strategy. However, if the user had entered the above expression, we could use the associativity and commutativity of natural join to transform this expression to the more efficient expression we used earlier.

1.3.3. Projection Operations.

We now consider another technique for reducing the size of temporary result. The projection operation, like selection operation, reduces the size of relations. Thus, whenever we need to generate a temporary relation, it is advantageous to apply any projections that are possible. This suggests a companion to the perform selections early rule we stated earlier:

Perform projections early

Consider the following form of our example query:

$$\Pi_{-,}(((_{-=\text{Port Chester}}()) \bowtie) \bowtie)$$

When we compute the sub-expression:

$$\left(\left(\_ = ()\right) \bowtie \right)$$

We obtain a relation whose scheme is

$(- , - , - , - ,)$

We can eliminate several attributes from the scheme. The only attributes we must retain are those that:

Appear in the result of the query or
Are needed to process subsequent operations

By eliminating unneeded attributes, we reduce the number of columns of the intermediate result. Thus, the size of the intermediate result is reduced. In or example, the only attribute we need is branch-name. therefore, we modify the expression to:

$$\Pi_{-,}\left(\left(\Pi_{-}\left(\left(\_ \blacksquare ()\right) \bowtie \right)\right) \bowtie \right)$$

1.4. ESTIMATION OF QUERY-PROCESSING COSTS

The strategy we choose for a query depends upon the size of each relation and the distribution of value within columns. In the example we have used, the fraction of customers who live in Port Chester has a major impact on the usefulness of our techniques. In order to be able to choose a strategy based on reliable information, database systems may store statistics for each relation r. these statistics include:

1. the number of tuples in the relation

2. the size of a record (tuple) of relation

in bytes (for fixed-length records)

3. $(,)$, the number of distinct values that appear in the relation r for attribute

The first two statistics allow us to estimate accurately the size of a Cartesian product. The Cartesian product $\times$ contains tuples. Each tuple of $\times$ occupies bytes.

The third statistic is used to estimate how many tuples satisfy a selection predicate of the form:

$< - > = < >$

However, in order to perform such an estimation, we need to know how often each value appears in a column. If we assume that each value appears with equal probability, then $\blacksquare()$ is estimated to have $/(,)$ tuples. However, it may not always be realistic to assume that each value appears with equal probability. The *branch-name* attribute in the *deposit* relation is an example of such a case. There is one tuple in the *deposit* relation for each amount. It is reasonable to expect that the large branches have more accounts than smaller branches. Therefore certain *branch-name* values appear with greater probability than others.

Despite the fact that our uniform distribution assumption is not always true, it a good approximation of reality in many cases. Therefore, many query processors make such an assumption when choosing a strategy. For simplicity, we shall assume a uniform distribution for the remainder of this unit.

Estimation of the size of a natural join is somewhat more complicated than estimation of the size of a selection or a Cartesian product. Let $_1(_1)$ and $_2(_2)$ be relations. If $_1 \cap _2 =$ then $_1 \bowtie _2$ is the same as $_1 \times _2$, and we an use our estimation technique for Cartesian products. If $_1 \cap _2$ is a key for $_1$, then we know that a tuple of $_2$ will join with exactly one tuple from $_1$. Therefore, the number of tuples in $_1 \bowtie _2$ is no greater than the number of tuples in $_2$.

The most difficult case to consider is when $_1 \cap _2$ is a key for neither $_1$ nor $_2$. In this case, we use the third statistic and assume, as before, that each value appears with equal probability. Consider a tuple

of $_1$ and assume $_1 \cap _2 = \{\}$, we estimate that there are $_2/(,_2)$ tuples in $_2$ with an

value of $[]$. So tuple

produces:

$$\frac{_2}{(,_2)}$$

tuples in $_1 \bowtie _2$. considering all of the tuples in $_1$, we estimate that there are:

$$\frac{_{12}}{(,_2)}$$

Tuples in $_1 \bowtie _2$. Observe that if we reverse the roles of $_1$ and $_2$ in the above estimate, we obtain an estimate of $_1 {}_2 / (,_1)$ tuples in $_1 \bowtie _2$. These two estimates differ if $(,_1) \neq (,_2)$. If this situation occurs, there are likely to be some dangling tuples do not participate in the join. Thus, the lower of the two estimates is probably the better one.

The above estimate of join size may be too high if the $(_1)$ value in $_1$ have few values in common with the $(,_2)$ value in $_2$. However, it is unlikely that our estimate will be very far off in practice since dangling tuples are likely to be only a small fraction of the tuples in a real-world relation. If dangling tuples appear frequently, then a correction factor could be applied to our estimates.

If we wish to maintain accurate statistics, then every time a relation is modified, it is necessary also to update the statistics. This is a substantial amount of overhead. Therefore, most systems do not update the statistics on every modification. Instead, statistics are updated during periods of light load on the system. As a result, the statistics used for choosing a query processing strategy may not be accurate. However, if the interval between the update of the statistics is not too long, the statistics will be sufficiently accurate to provide a good estimation of the size of the results of expressions.

Statistical information about relations is particularly useful when several indices are available to assist in the processing of a query, as we shall see in unit 1.5

1.5. ESTIMATION OF COSTS OF ACCESS USING INDICES

The cost estimates we have considered for relational algebra expressions did not consider the effects of *indices* and *has functions* on the cost of evaluating an expression. The presence of these structures, however, has a significant influence on the choice of a query–processing strategy.

> *Indices* and *has functions* allow fast access to records containing a specific value on the index key
> *Indices* (though not most *has functions*) allow the records of a file to be read in sorted order. If an index allows the records of a file to be read in an order that corresponds to the physical order of records, we call that index a *clustering index*. Clustering indices allow us to take advantage of the physical clustering of records into blocks.

The detailed strategy for processing a query is called an *access plan* for the query. A plan includes not only the relational operations to be performed but also the indices to be used and the order in which tuples are to be accessed and the order in which operations are to be performed.

Of course, the use of indices imposes the overhead of access to those blocks containing the index. We need to take these blocks accesses into account when we estimate the cost of a strategy that involves the use of indices.

In this unit, we consider queries involving only one relation. We use the selection predicate to guide us in the choice of the best index to use in processing the query.

As an example of the estimation of the cost of a query using indices assume that we are processing the query:

select *account – number*

from *deposit*

where *branch-name* = "Perryridge' **and** *customer-name* = "Williams"

Assume that we have the following statistical information about the deposit relation:

20 tuples of deposit fit in one block
V(deposit, branch – name) = 50
V(deposit, customer-name) = 200
V(deposit, balance) = 5000
the deposit relation has 10,000 tuples.

Let us assume that the following indices exists on deposit:

A clustering, $^{+}$-tree index for branch-name
A non-clustering, $^{+}$-tree index for customer-name

As before, we shall make the simplifying assumption that value are distributed uniformly.

Since *V(deposit, branch-name)* = 50, we expect that 10000/50 = 200 tuples of the *deposit* relation pertain to the Perryridge branch. If we use the index on branch-name, we will need to read these 200 tuples and check each one for satisfaction of the **where** clause. Since the index is a clustering index, 200/20 = 10 block reads are required to read the *deposit* tuples. In addition, several index blocks must be read. Assume the $^{+}$- tree index stores 20 pointers per node. This means that the $^{+}$+ - tree index must have between 3 and 5 leaf nodes. With this number of leaf nodes, the entire tress has depth of 2, so at most 2 index blocks must be read. Thus the above strategy requires 12 total block reads.

We conclude that it is preferable to use the index for *branch-name*.

If we use the index for *customer-name*, we estimate the number of block accesses as follows. Since *V(deposit, customer-name)* = 200, we expect that 10000/200 = 50 tuples of the deposit relation pertain to Williams. However, since the index for *customer-name* is non-clustering, we anticipate that one block read will be required for each tuple. Thus, 50 block reads are required, just to read the deposit tuples. Let us assume that 20 pointers fit into one node of $^{+}$- tree index for *customer-name*. Since there are 200 customer names, the tree has between 11 and 20 leaf nodes. So, as was the case for other $^{+}$- tree index, the index for customer-name has a depth of 2 and2 block accesses are required to read the necessary index blocks. Therefore, this strategy requires a total of 52 block reads. We conclude that it is preferable to use the index for *branch-name*.

Observe that if both indices were non-clustering, we would prefer to use the index for *customer-name* since we expect only 50 tuples with *customer-name* = Williams" versus 200 tuple with branch- name = "Perryridge" without the clustering property, our first strategy would have required 200 block accesses to read the data plus 2 index block accesses for a total of 202 block reads. However, because of the clustering property of the branch-name index, it is actually less expensive in this example to use the branch-name index.

We did not consider using the balance attribute and the predicate balance >1000 as a starting point for a query processing strategy for two reasons:

there is no index for balance

the selection predicate on balance involves a “greater than” comparison. In general, equality predicates are more selective than “greater than” predicates. Since we have an equality predicate available to us (indeed, we have two), we prefer to start by using such a predicate since it is likely to select fewer tuples.

Estimation of the cost of access using indices allows us to estimate the complete cost, in terms of block accesses, of a plan. For a given relational algebra expression, it may be possible to formulate several plans. The access plan selection phase of a query optimizer chooses the best plan for a given expression.

We have seen that different plans may have significant differences in cost. It is possible that a relational algebra expression for which a good plan exists may be preferable to an apparently more efficient algebra expression for which only inferior plans exist. Thus, it is often worthwhile for a large number of strategies to be evaluated down to the access plan level before a final choice of query–processing strategy is made.

1.6. JOIN STRATEGIES

Earlier, we estimated the size of the result of a relational algebra expression involving a natural join. In this unit, we apply our techniques for estimating the cost of processing a query to the problem of estimating the cost of processing a join. We shall see that several factors influence the selection of an optimal strategy:

The physical order of tuples in a relation

The presence of indices and the type of index (clustering or nonclustering)

The cost of computing a temporary index for the sole purpose of processing one query

Let us begin by considering the expression

$$\bowtie$$

and assume that we have no indices whatsoever. Let:

$$= 10{,}000$$
$$= 200$$

1.6.1. Simple Iteration

If we are not willing to create an index, we must examine every possible pair of tuples $_1$ in *deposit* and $_2$ in *customer*. Thus, we examine $10000 * 200 = 2000000$ pairs of tuples.

If we execute this query cleverly, we can reduce the number of block accesses significantly. Suppose that we use the procedure of figure 1.1 for computing the join. We read each tuple of *deposit* once. This may require as many as 10,000 block accesses. However, if the tuples of deposit are stored together physically, fewer accesses are required. If we assume that 20 tuples of deposit fit in one block, then reading deposit requires 10000/20 = 500 block accesses.

```
for each tuple
in deposit do
    begin
        for each tuple c in customer do
            begin
                test pair (d, c) to see if a tuple should be added to the result
```

We read each tuples of customer once for each tuple of deposit. This suggests that we read each tuples of customer 10,000 times. Since $= 200$, we could make as many as 2,000,000 accesses to read customer tuples. As was the case for *deposit*, we can reduce the required number of accesses significantly if we store the customer tuples together physically. If we assume that 20 *customer* tuples fit in one block, then only 10 accesses are required to read the entire customer relation. Thus, only 10 accesses per tuple of *deposit* rather than 200 are required. This implies that only 100,000 block accesses are needed to process the query.

1.6.2. Block–Oriented Iteration

A major savings in block accesses results if we process the relations on a per – block basis rather than a per-tuple basis. Again, assuming that deposit tuples are stored together physically and that customer tuples are stored together physically, we can use the procedure of figure 1.2 to compute

$$\bowtie .$$

This procedure performs the join by considering an entire blocks of *deposit* tuples at once. We still must read the entire *deposit* relation at a cost of 500 accesses. However, instead of reading the *customer* relation once for each tuple of deposit, we read the customer relation once for each block of deposit. Since there are 500 blocks of deposit tuples and 10 blocks of customer tuples, reading customer once for every block of deposit tuples requires 10 x 500 = 5000 block accesses.

Thus, the total cost in terms of block accesses is 5500 accesses (5000 accesses to customer blocks plus 500 accesses to deposit blocks). Clearly, this is a significant improvement over the number of accesses that were necessary for our initial strategy.

Our choice of deposit for the outer loop and customer for enter loop was arbitrary. If we had used *customer* as the relation for the outer loop, the cost of our final strategy would have been slightly lower (5010 block accesses). A major advantage to the use of the smaller relation (*customer*) in the inner loop is that it may be possible to store the entire relation in main memory temporarily. This speeds query processing significantly since it is necessary to read the inner loop relation only once. If customers is indeed small enough to fit in main memory, out strategy requires only 500 blocks to read deposit plus 10 blocks to read customer for a total of 510 block accesses.

```
for each block

of

do

begin

    for each block

of customer do

    begin

        for each tuple b in

      do
```

1.6.3. Merge-Join

In those cases in which neither relation fits in main memory inform be store and possible to process the join efficiently if both relations happens merge-join in sorted order on the join

```
pd: = address of first tuple of deposit;
pc: = address of first tuple of customer;
while (pc ≠ null) do
    begin
        := tuple to which pc points;
        := {}:
        set pc to point to next tuple of customer;
        done: = false;
        while (not done) do
            begin;
                := tuple to which pc points;
                if [ - ] = [ - ]
                    then begin
                        := ∪ {}:
                        set pc to point to next tuple of customer;
                    end
                    else done := true;
            end
            :=    ;
            set pd to point to next tuple of deposit;
            while ([ - ] < [ - ] do
```

attributes. Suppose that both with each deposit are sorted by *customer-name*. We can then perform irrespective operation. To compute a merge–join, we associate one point through the relation. These pointers point initially to the first tuple of the join relations. As the algorithm proceeds, the pointers move the other relation. A group of tuples of one relation with the same value with the attribute is read. Then the corresponding tuples (if any) allow us relations relation are read. Since the relations are in sorted order, tuples us to same value on the join attributes are in consecutive order. Thus allow us to read each tuple only once. In the case in which the example of relations are stored together physically, compute the join by reading each block exactly once.

For our example which the $\bowtie$ there is a total of 510 block accesses. This is as good as the earlier join method we presented for the special case in which the entire customer relations fit in main memory. The algorithm of figure 1.3 does not require the entire relation to fit in main memory large. Rather, it suffices to keep all tuple with the same value for the join attributes in main memory. This is usually feasible even if both relations.

A disadvantage of the merge–join method is the requirement relations be sorted physically. However, it may be worthwhile to sort the relations in order to allow a merge-join to be performed.

1.6.4. Use of an Index

Frequently, the join attributes form a search key for an index relations being joined. In such a case, we may consider a join strategy that uses such an index. The simple strategy of figure 1.1 is more of index exists on *customer* for *customer-name*. Given a tuple

in deposit, it is

no longer necessary to read the entire customer relation. Instead, the index is used to look up tuples in customer for which the *customer-name* value is *d[customer-name].*

Without use of an index, and without special assumptions about the physical storage of relations, it was shown that as many as 2 million accesses might be required. Using the index, but without making any assumptions about physical storage, the join can be computed with significantly fewer block accesses. We still need 10,000 accesses to read deposit. However, for each tuple of deposit only an index lookup is required. If we assume (as before) that $= 200$, and that 20 pointers fit in one block, then this lookup requires at most 2 index block accesses plus a block access to read the customer tuple itself. We access 3 blocks per tuple of deposit instead of 200. Adding this to the 10,000 accesses to read deposit, we find that the total cost of this strategy is 40,000 accesses.

Although a cost of 40,000 accesses appears high, we must remember that we achieved more efficient strategies only when we assumed that tuples were stored physically together. If this assumption does not hold for the relations being joined, then the strategy we just presented is highly desirable. Indeed the savings (160,000 accesses saved) is enough to justify creation of the index. Even if we create the index for the sole purpose of processing this one query and erase the index afterwards, we may perform fewer accesses than if we use the strategy of figure 1.1.

1.6.5 Three – Way Join

Let us now consider a join involving three relations

$\bowtie \quad \bowtie$

Assume that and are as above and that = 50. Not only do we have a choice of strategy for join processing, but also we have a choice of which join to compute first. There are many possible strategies to consider. We shall analyze several of them below and leave others to the exercises.

Strategy 1. Let us first compute the join $(\bowtie)$ using one of the strategies we presented above. Since customer-name is a key for customer, we know that the result of this join has at most 10,000 tuples (the number of tuples in deposit). If we build an index on branch for branch – name, we can compute:

$\bowtie (\bowtie)$

by considering each tuple

of (deposit x customer) and looking up the tuple in branch with a branch – name value of t(branch-name). since branch-name is a key for branch, we know that we must examine only one branch tuple for each of the 10,000 tuples in $(\bowtie)$. The exact number of block accesses required by this strategy dependes on the way we compute (deposit x customer) and on the way in which branch is stored physically. Several exercises examine the costs of various possibilities.

Strategy 2. Compute the join without constructing any indices at all. This requires checking 50 * 10000 * 200 possibilities, a total of 100,000,000

Strategy 3. Instead of performing two joins, we perform the pair of joins at once. The technique is first to build two indices:

On branch for branch-name

On customer for customer –name

Next we consider each tuple t in deposit. For each t, we look up the corresponding tuples in customer and the corresponding tuples in branch. Thus, we examine each tuple of deposit exactly once.

Strategy 3 represents a form of strategy we have not considered before. It does not correspond directly to relational algebra operation. Instead, it combines two operations into one special-purposes operation. Using strategy 3, it is often possible to perform a join of three relations more efficiently than it is using two joins of two relations. The relative costs depends on the way in which the relations are stored, the distribution of values within columns, and the presence of indices. The exercises provide an opportunity to compute these costs in several examples.

1.7.STRUCTUREOFQUERYOPTIMIZER

We have seen only some of the many query processing strategies used in database systems. Most systems implement only a few strategies and, as a result, the number of strategies to be considered by the query optimizer is limited. Other systems consider a large number of strategies. For each strategy a cost estimate is computed.

In order to simplify the strategy selection task, a query may be split into several sub-queries. This not only simplifies strategy selection but also allows the query optimizer to recognize cases where a particular sub-query appear several times in the same query. By performing such sub-queries only once, time is saved both in the query optimizing phase and in the execution of the query itself. Recognition of common sub-queries is analogous to the recognition of common sub-expressions in many optimizing compilers for programming languages.

Clearly, examination of the query for common subs queries and the estimation of the cost of a large number of strategies impose a substantial overhead on query processing. However, the added cost of query optimization is usually more than offset by the savings at query execution time. Therefore, most commercial systems include relatively sophisticated optimizers.

1.8. CONCLUSION

There are a large number of possible strategies for processing a query, especially if the query is complex. Strategy selection can be done using information available in main memory, with little or no disk accesses. The actual execution of the query will involve many accesses to disk. Since the transfer of data from disk is slow relative to the speed of main memory and the central processor of the computer system, it is advantageous to spend a considerable amount of processing to save disk accesses.

Given a query, there are generally varieties of methods for computing the answer. It is the responsibility of the system to transform the query as entered by the user into an equivalent query, which can be computed more efficiently. This “optimizing” or, more accurately, improving of the strategy for processing a query is called *query optimization*.

The first action the system must take on a query is to translate the query into its internal form, which (for relational database systems) is usually based on the relational algebra. In the process of generating the internal form of the query, the parser checks the syntax of the user’s query, verifies that the relations names appearing in the query are names of relation in the database, etc. if the query was expressed in terms of a view, the parser replaces all references to the view name with the relational algebra expression to compute the view.

1.9. SUMMARY

Each relational algebra expression represents a particular sequence of operations. The first step in selecting a query – processing strategy is to find a relational algebra expression that is equivalent to the given expression and is efficient to execute. There are a number of different rules for transforming relational algebra queries, including to:

perform selection operations as early possible
perform projections early

The strategy we choose for a query depends upon the size of each relation and the distribution of values within columns. In order to be able to choose a strategy based on reliable information, database systems may store statistics for each relation r. these statistics include:

the number of tuples in the relation r
the size of a record (tuple) of relation r in bytes (for fixed-length records)
the number of distinct values that appear in the relation r for a particular attribute.

The first two statistics allow us to estimate accurately the size of a Cartesian product. The third statistics allows us to estimate how many tuples satisfy a simple selection predicate.

Statistical information about relations is particularly useful when several indices are available to assist in the processing a query. The presence of these structures has a significant influences on the choice of a query – processing strategy.

Queries involving a natural join may be processed in several ways, depending on the availability of indices and the form of physical storage used for the relations. If tuples of a relation are stored together physically, a block – oriented join strategy may be advantageous. If the relations sorted, a merge – join may be desirable. It may be more efficient to sort a relation prior to join computation (so as to allow use of the merge – join strategy). It may also be advantageous to compute a temporary index for the sole purpose of allowing a more efferent join strategy to be used.

1.10. TUTOR-MARKED ASSIGNMENT (TMA).

1. At what point during query processing does optimization occur?

2. Why is it not desirable to force users to make an explicit choice of a query processing strategy? Are there cases in which it is desirable for users to be aware of the costs of competing query-processing strategies?

3. Show that the following equivalences hold, and explain how they can be applied to improve the efficiency of certain queries:

a. $({}_1 \cup {}_2) = ({}_1) \cup ({}_2)$

b. $({}_1 - {}_2) = ({}_1) - {}_2 = ({}_1) - ({}_2)$

c. $({}_1\ {}_2) \cup {}_3 = {}_1 \cup ({}_2 \cup {}_3)$

d. $_1 \cup _2 = _2 \cup _1$

4. Consider the relations $_1(,,)$, $_2(,,)$ and $_3(,)$, with primary keys *A*, *C*, and *E* respectively. Assume that $_1$ has 1000 tuples, $_2$ has 1500 tuples and $_3$ has 750 tuples. Estimate the size of $_1\ _2\ _3$ and give an efficient strategy for computing the join.

1.11. FURTURE READINGS

Boyce, R., Chamberlin, D., King, W., and Hammer, M. [1975] "Specifying Queries as Relational Expressions", CACM, 18 : 11, November 1975.

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[1986]. "Database System Concepts", McGraw – hill, 1986.

www.w3schools.com

Zobel, J., Moffat, A., and Sacks–Davis, R., [1992]. "An Efficient Indexing Technique for Full – Text Database Systems," in VLDB [1992.

MODULE 2: ADVANCED FEATURES OF DBMS

UNIT 2: TRANSACTION MANAGEMENT AND RECOVERY

2.0. INTRODUCTION

Transaction processing systems are systems with large databases and hundreds of concurrent users. It provides and "all-or-noting" proposition stating that each work –unit performed in database must either complete in its entirety or have no effect whatsoever. Users of database systems are usually consider consistency and integrity of data as highly important. A simple transaction is usually issued to the database system in a language like SQL wrapped in a transaction.

2.1. OBJECTIVES

At the end of this unit, the student should be able to discuss:

- database transaction and its properties
- how database transactions are managed
- concurrency control and the role it plays in maintaining the database's integrity
- locking methods and how they work
- how database recovery management is used to maintain database integrity

2.2. WHAT IS A TRANSACTION?

A transaction is a logical unit of database processing, which can include one or more database operations, such as insertion, deletion, modification, or retrieval operations. In a database environment, transactions have two main purposes:

1. To provide reliable units of work that allow correct recovery from all failures and keep a database consistent even in vases of system failure, when execution stops (completely or partially) and many operations upon the database remain uncompleted, with unclear status.
2. To provide isolation between programs accessing a database concurrently. Without this, the programs outcomes are typically erroneous.

Transaction processing systems are systems with large databases and hundreds of concurrent users. It provides and "all-or-noting" proposition stating that each work –unit performed in database must either complete in its entirety or have no effect whatsoever. Further, the system must isolate each transaction from the other, results must conform to existing constraints in the database and transactions that complete successfully must be written to durable storage.

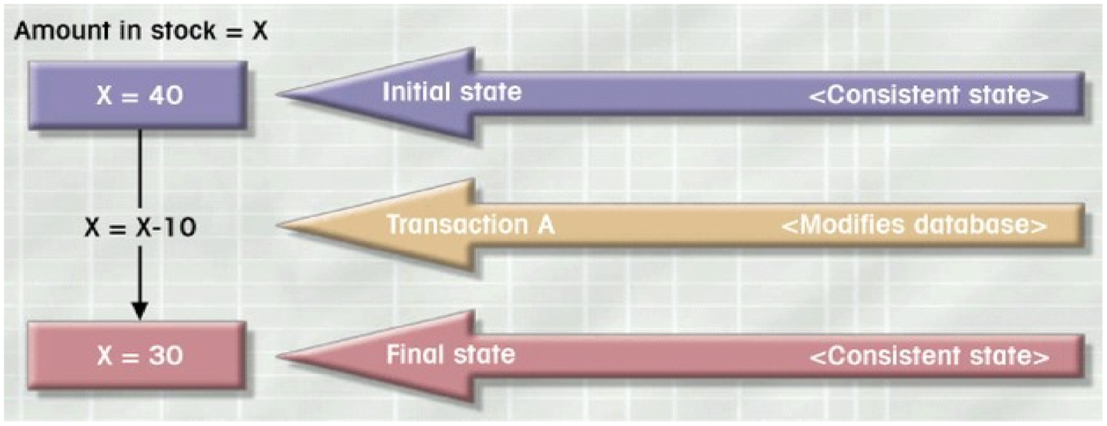


Figure 2.1: A consistent database state is one in which all data integrity constraints are satisfied.

2.2.2. Transaction Properties

The set of properties that guarantee that database transactions are processed reliably is given the acronym ACID (Atomicity, Consistency, Isolation and Durability)

- **Atomicity** - this refers to the ability of the DBMS to guarantee that either all of the transactions are performed or non of them are. For example, the transfer of funds from one account to another can be completed or it can fail for a multitude of reasons, but atomicity guarantees that one account would not be debited if the other is not credited. Atomicity states that database modifications must follow an “all-or-nothing” rule. Each transaction is said to be atomic if when one part of the transaction fails, the entire transaction fails. It is critical that database management system maintain the atomic nature of transitions in spite of any DBMS operating system or hardware failure.

- **Consistency** – this ensures that the database remains consistent before the start of the transaction and after the transaction is over (whether successful or not) Consistency states that only valid data will be written to the database. If for some reason, a transaction violates the database’s consistency rule, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules. On the other hand, if the transaction successfully executes, it will take the database from one state that is consistent with the rules to another state that is also consistent with the rules.

- **Isolation** – this refers to the requirement that other operations cannot access or see data in as immediate state during a transaction. This constraint is requires to maintain the performance as well as the consistency between transactions in a DBMS. Thus, each transaction is unaware of another transaction executing concurrently in the system.

- **Durability** – this refers to the guarantee that once the user has been notified of success, the transaction will persist and not be undone. This means it will survive system failure, and that the database system has checked the integrity of the constraints and would not need to abort the transaction. Many databases implement durability by writing all transactions into a transaction log.
  Durability does not imply a permanent state of the database. Another transaction may overwrite any changes made by the current transaction without hindering durability.

2.2.2. Transaction Management with SQL

Users of database systems are usually consider consistency and integrity of data as highly important. A simple transaction is usually issued to the database system in a language like SQL wrapped in a transaction using the pattern similar to that below:

1. Begin the transaction.
2. Execute several data manipulations and queries.
3. If no error occurs commit the transaction and end it.
4. If errors occur, then rollback the transaction and end it.

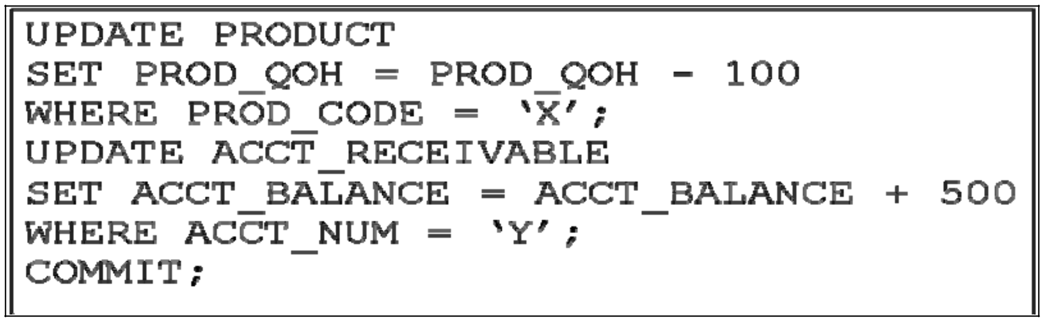
If no errors occur during the execution of the transaction then the system commits the transaction. A transaction commit operation applies all data manipulation within the scope of the transaction and persists the results to the database. If an error occurs during the transaction, or if the user specifies a rollback operation, the data manipulations within the transactions are not persisted into the database. In no case can a partial transaction be committed to the database since that would leave the database in an inconsistent state.

A **START TRANSACTION** statement in SQL or any other statement that will modify data stars with a transaction within a relational database management system. The result of any work done after this point will remain invincible to other database-users until the system process a COMMIT statement. A **ROLLBACK** statement can also occur, which will undo any work performed since the *START TRANSACTION* command. Both *COMMIT* and *ROLLBACK* will end the transaction: another *START TRANSACTION* will need to be issued to start another one.

An example of a transaction is given below:

- Transaction support
  - COMMIT
  - ROLLBACK
- User initiated transaction sequence must continue until:

  - COMMIT statement is reached
  - ROLLBACK statement is reached
  - End of a program reached
  - Program reaches abnormal termination (leads to ROLLBACK)
- Register credit sale of 100 units of product X to customer Y for ₦500



```
UPDATE PRODUCT
SET PROD_QOH = PROD_QOH - 100
WHERE PROD_CODE = 'X';
UPDATE ACCT_RECEIVABLE
SET ACCT_BALANCE = ACCT_BALANCE + 500
WHERE ACCT_NUM = 'Y';
COMMIT;
```

Figure 2.2: A query with the *commit* statement

- Transaction begins when first SQL statement is encountered, and ends at COMMIT or end

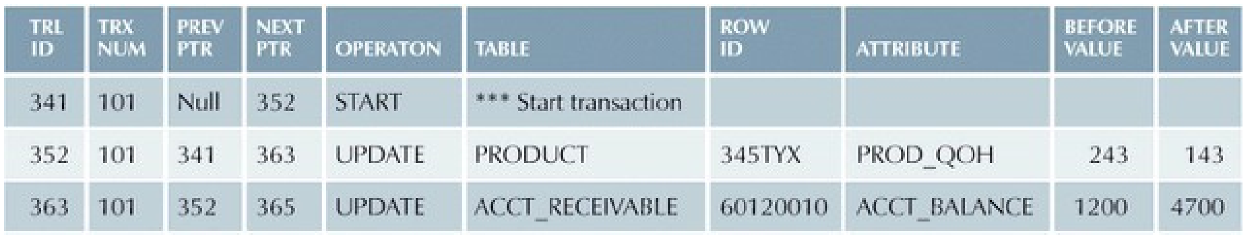
2.2.3. The Transaction Log

This is a history of all the actions executed by a database management system to guarantee ACID properties over crashes or hardware failures. Physically, a log is a file of updates done to the database and stored in stable storage. If after a start, the database is found in an inconsistent state or is not shut down properly, the database management system reviews the database logs for uncommitted transactions and rolls back the changes made by these transactions. Additionally, all transactions already committed but whose changes were not yet materialized in the database are re-applied. Both are done to ensure atomicity and durability of transactions.

A transaction log is made up of:

Log Sequence Number: A unique id for a log record. With LSNs, logs can be recovered in constant time. Most logs' LSNs are assigned in monotonically increasing order, which is useful in recovery algorithms.
PrevLSN: A link to the last log record. This implies database logs are constructed in linked list form.
Transaction ID number: A reference number to the database transaction generating the log record.
Type: Describes the type of transaction log.
Information about the actual changes that triggered the log record to be written.

Example



TRL ID	TRX NUM	PREV PTR	NEXT PTR	OPERATON	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	*** Start transaction				
352	101	341	363	UPDATE	PRODUCT	345TYX	PROD_QOH	243	143
363	101	352	365	UPDATE	ACCT_RECEIVABLE	60120010	ACCT_BALANCE	1200	4700

- TRL_ID: transaction log record ID (Log Sequence Number)
- TRX_NUM: transaction number (Transaction ID)
- PREV_PTR: pointer to previous transaction record (PrevLSN)
- NEXT_PTR: pointer to next transaction record

2.2.4. Types of transaction log records

All log records include the general log attributes described above and also other attributes depending on their type (which is recorded in the Type attribute above).

Update Log Record notes an update (change) to the database. It includes this extra information

PageID: A reference to the page ID of the modified page.
Length and Offset: Length byte and offset of the page are usually included.
Before and After images: includes the value of the bytes of the page change. Some databases may have logs which include one or both images.

Compensation Log Record notes the rollback of a particular change to the database. Each corresponds with exactly on other Update Log Record (although the corresponding log record is not typically stored in the Compensation Log Record). It includes this extra information:

undoNextLSN: This field contains the LSN of the next log record that is to be undone for the transaction that wrote the last Update Log.

Commit Record notes a decision to commit a transaction.

Abort Record notes a decision to abort and hence rollback a transaction.

Checkpoint Record notes that a checkpoint has been made. These are used to speed up recovery. They record information that eliminates the need to read a long way into the log's log. This varies according to the checkpoint algorithm. If all dirty pages are flushed while creating the check point (as in PostgreSQL), it might contain:

redoLSN: This is a reference tot the first log record that corresponds to a dirty page i.e. the first update that wasn't flushed at the checkpoint time. This is where redo must begin recovery.
undoLSN: This is a reference to the oldest log record of the oldest in-progress transaction. This is the oldest log record needed to undo all in-progress transactions.

Completion Record notes that all work has been done for this particular transaction. (It has been fully committed or aborted)

2.3.CONCURRENCYCONTROL

Concurrency control is the process of managing/controlling simultaneous operations on the database. It is required because actions from different users or applications taking place upon a database must not interfere. It establishes order of concurrent transactions.

Interleaving operations can lead to the database being in an inconsistent state. Three potential problems which should be addressed by successful concurrency control are as follows:

- Lost Updates
- Uncommitted data
- Inconsistent Retrievals**.**

The Scheduler is a module that is responsible for implementing a particular strategy for concurrency control. It:

- Establishes order for concurrent transaction execution.
- Interleaves execution of database operations to ensure serializability
- Bases actions on concurrency control algorithms
  - Locking
  - Time stamping
- Ensures efficient use of computer's CPU

2.3.1. Concurrency Control with Locking Methods

A transaction will use a lock to deny data access to other transactions and so prevent incorrect updates. Locks can be Read (shared) or Write (exclusive) locks. Write Locks on a data item prevents other transaction from reading that data item whereas Read Locks simply stop other transactions from editing (writing to) the data item.

Locks are used in the following ways:

1. Any transaction that needs to access a data items must first lock the item, requesting a shared lock for read only access or exclusive lock for read and write access.

2. If the item is not locked by another transaction, the lock will be granted.

3. If the item is currently locked, the DBMS determines whether the request is compatible with the existing lock. If a shared lock is requested on an item that already has shared lock on it, request will be granted. Otherwise, transaction must wait until the existing lock is released.

4. Transaction continues to hold a lock until it is explicitly released either during execution or when it terminates (aborts or commit). It is only when the exclusive lock has been released that the effects of the write operation will be made visible to another transaction.

2.3.2. Types of Locks

The two main lock types are:

- Binary locks
- Shared/Exclusive locks

Binary locks

Binary can be in one of two states i.e. the locked states (denoted by 1) and the unlocked state (denoted by 0) . Locked objects are unavailable to other objects and this is managed by the DBMS. Unlocked objects however, are open to any order transaction. Its each transaction that locks its objects and the same unlocks it when complete. It is also possible to change DBMS default with LOCK TABLE and other SQL commands.

Shared/Exclusive locks

Shared locks exist when concurrent transactions grants READ access. This produces no conflict for read-only transactions and it is issued when transactions want to read and an exclusive lock is not held on the item.

Exclusive locks exist when access is reserved for locking transactions. This is used when potential conflict exists and is exclusively granted if object does not have a lock yet. It is used when transactions have to update unlocked data.

2.3.3. Two-Phase locking to Ensure Serializability

A transaction follows the “two-phase” lock if all lock requests come before the first unlock operation within the transaction. This means there are two main phases within the transaction:

Growing phase (all locks are acquired)

Shrinking phase (locks are released – no new lock can be acquired)

2.3.4. Deadlocks

Deadlocks occur when two or more transactions are waiting for locks held by each other to be released. The only way to break a deadlock is to abort one of the transactions so that a lock is released so the other transaction can proceed. The DBMS can manage this process of aborting a transaction when necessary. The aborted transaction is typically restarted so that it is able to execute and commit without the user being aware of any problems occurring. A timeout could also be used. The transaction that requests a lock waits for at most a specified period. Deadlocks can be prevented when the DBMS looks ahead to determine if the transaction would cause a deadlock and thus never allow the dead lock to occur.

2.4. CONCURRENCY CONTROL WITH TIME STAMPING METHODS

Whenever a transaction starts, it is given a timestamp. This is so we can tell which order that the transactions are supposed to be applied in. So given two transactions that affect the same object, the transaction that has the earlier timestamp is meant to be applied before the other one. However, if the wrong transaction is actually presented first, it is aborted and must be restarted.

Every object in the database has a **read timestamp**, which is updated whenever the object's data is read, and a **write timestamp**, which is updated whenever the object's data is changed.

If a transaction wants to read an object, but the transaction started *before* the object's **write timestamp** it means that something changed the object's data after the transaction started. In this case, the transaction is cancelled and must be restarted.

If a transaction wants to write to an object, but the transaction started *before* the object's **read timestamp** it means that something has had a look at the object, and we assume it took a copy of the object's data. So we can't write to the object as that would make any copied data invalid, so the transaction is aborted and must be restarted.

Timestamp Resolution

This is the minimum time elapsed between two adjacent timestamps. If the resolution of the timestamp is too large (coarse), the possibility of two or more timestamps being equal is increased and thus enabling some transactions to commit out of correct order. For example, assuming that we have a system that can create one hundred unique timestamps per second, and given two events that occur 2 milliseconds apart, they will probably be given the same timestamp even though they actually occurred at different times.

Timestamp Locking

Even though this technique is a non-locking one, in as much as the Object is not locked from concurrent access for the duration of a transaction, the act of recording each timestamp against the Object requires an extremely short duration lock on the Object or its proxy.

2.5. CONCURRENCY CONTROL WITH OPTIMISTIC METHODS

Optimistic Concurrency Control (OCC) is a concurrency control method that assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back.

However, if conflicts happen often, the cost of repeatedly restarting transactions hurts performance significantly; other concurrency control methods have better performance under these conditions.

2.5.1 Optimistic Concurrency Control Phases

More specifically, OCC transactions involve these phases:

- **Begin**: Record a timestamp marking the transaction's beginning.
- **Modify**: Read and write database values.
- **Validate**: Check whether other transactions have modified data that this transaction has modified. Always check transactions that completed after this transaction's start time. Optionally, check transactions that are still active at validation time.
- **Commit/Rollback**: If there is no conflict, make all changes part of the official state of the database. If there is a conflict, resolve it, typically by aborting the transaction, although other resolution schemes are possible.

OCC is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions wait for other transactions' locks to clear, leading to higher throughput than other concurrency control methods.

2.6.DATABASERECOVERYMANAGEMENT

Database Recovery deals with restoring the database to a correct state in the event of a failure. We have various storage that holds data and they include main memory, magnetic disk, magnetic tape and optical disks. Failure that could make the DBMS require recovery includes:

1. System crashes (Software & Hardware)
2. media failures
3. application software error
4. natural physical disaster
5. unintentional distraction
6. sabotage
7. Programming Exemption

Levels of backup

a. Full database backup
b. Differential backup
c. Transaction log backup

Transaction recovery comprises of the following:

- Write-ahead protocol
- Redundant transaction logs
- Database buffers
- Database checkpoints

There are two basic features for transaction recovery they are

1. **Deferred-write and Deferred-update**
   - Changes are written to the transaction log
   - Database updated after transaction reaches commit point
2. **Write-through**
   - Immediately updated during execution
   - Before the transaction reaches its commit point
   - The transaction log is also updated
   - If transaction fails, database uses log information to ROLLBACK

2.7. CONCLUSION

Transaction processing systems are systems with large databases and hundreds of concurrent users. It provides and "all-or-noting" proposition stating that each work –unit performed in database must either complete in its entirety or have no effect whatsoever. Further, the system must isolate each transaction from the other, results must conform to existing constraints in the database and transactions that complete successfully must be written to durable storage.

2.8. SUMMARY

A transaction is a logical unit of database processing, which can include one or more database operations, such as insertion, deletion, modification, or retrieval operations. A transaction will use a lock to deny data access to other transactions and so prevent incorrect updates. Database recovery is necessary to deal with restoring the database to a correct state in the event of a failure.

2.9. TUTOR-MARKED ASSIGNMENT (TMA)

1. What is a transaction?

2. Discuss the properties of database transactions.

3. Discuss how simultaneous transactions are controlled.

2.10. FURTHER READINGS

Eswaran, K., Gray, J., Lorie, R., and Traiger, I. [1976] “The Notions of Consistency and Predicate Locks in a Data Base System”, CACM, 19 : 11 November 1976.

Gray. J. [1981] “The Transaction Concept: Virtues and Limitations”, in VLDB [1981].

Gray., J., and Reuter, A [1983] Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[1986]. “Database System Concepts”, 4th ed., McGraw – hill, 1986.

www.w3schools.com

Zobel, J., Moffat, A., and Sacks–Davis, R., [1992]. “An Efficient Indexing Technique for Full – Text Database Systems,” in VLDB [1992.

MODULE 2: ADVANCED FEATURES OF DBMS

UNIT 3: DATABASE SECURITY & AUTHORIZATION

3.0. INTRODUCTION

The data stored in the database needs to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. In this unit, we examine the ways in which data may become inconsistent or be misused. We then present mechanisms to guard against their occurrence.

3.1 OBJECTIVES

To understand the various security and authorization concepts and forms

To understand the various constraints in ensuring database consistency

3.2. SECURITY AND INTEGRITY VIOLATIONS

Misuse of the database can be categorized as being either intentional (malicious) or accidental. Accidental loss of data consistency may result from:

Crashes during transaction processing

Anomalies due to concurrent access to the database

Anomalies due to the distribution of data over several computers

A logical error that violates the assumption that transactions preserve the database consistency constraints.

It is easier to protect accidental loss of data consistency than to protect against malicious access to the database. Among the forms of malicious access are the following:

Unauthorized reading of data (theft of information)

Unauthorized modification of data

Unauthorized destruction of data

Absolute protection of the database from malicious abuse is not possible, but the cost of the perpetrator can be made sufficiently high to deter most if not all attempts to access the database without proper authority. The term database security usually refers to security from malicious access, while integrity refer to the avoidance of accidental loss of consistency. In practice, the dividing line between security and integrity is not always clear. We shall use the term security to refer to both security and integrity in cases where the distinction between these concepts is not essential.

In order to protect the database, security measures must be taken at several levels:

Physical. The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders.
Human. Authorization of users must be done carefully to reduce the chance of authorized user giving access to an intruder in exchange for a bribe or other favours.
Operating system. No matter how secure the database system is, weakness in operating system security may serve as a means of unauthorized access to the database. Since almost all database systems allow remote access through terminals or networks, software-level security within the operation system is as important as physical security.
Database system. Some authorized database system users may be authorized to access only a limited portion of the database. Other users may be allowed to issue queries, but may be forbidden to modify the data. It is the responsibility of the database system to ensure that these restrictions are not violated.

It is worthwhile in many applications to devote a considerable effort to the preservation of the integrity and security of the database. Large databases containing payroll or other financial data are inviting targets to thieves. Database that contain data pertaining to corporate operations may be of interest to unscrupulous competitions. Furthermore, loss of such data, whether via accident or fraud, can seriously impair the ability of the corporation to function.

In the remainder of this unit, we shall address security at the database system level. Despite the importance of physical and human level security these subjects are far beyond the scope of this text. Security within the operating system is implemented at several levels ranging from passwords for access to the system to the isolation of concurrent processes running within the system. The file system also provides some degree of protection. We shall present our discussion of security in terms of the relational data model, although the concepts of this unit are equally applicable to all data models.

3.3.AUTHORIZATION&VIEWS

The concept of views is a means of providing a user with a "personalized" model of the database. A view can hide data that a user does not need to see. The ability of views to hide data serves both to simplify usage of the system and to enhance security. System usage is simplified since the user is allowed to restrict attention to the data of interest. Security is provided if there is a mechanism to restrict the user to his or her personal view or views. Relational database systems typically provide security at two levels:

Relation. A user may be permitted or denied direct access to a relation
View. A user may be permitted or denied access to data appearing in a view.

Although a user may be denied direct access to a relation, the user may be able to access part of that relation through a view. Thus, a combination of relational – level security and view – level security can be used to limit a user's access to precisely the data that user needs.

In our bank example, consider a clerk who needs to know the names of the customers of each branch. This clerk is not authorized to see information regarding specific loans and accounts that the customer may have. Thus, the clerk must be denied direct access to the borrow and deposit relations. In order for the clerk to have access to the information needed, we grant the user access to the view all – customer which we define as:

create view *all – customer* **as**

(**select** *branch – name, customer – name*

from *deposit*)

union

(**select** *branch-name, customer-name*

from *borrow*)

A user may have several forms of authorization on part of the database. Among these are the following:

Read authorization, which allows reading, but not modification of data

Insert authorization, which allows insertion of new data, but not the modification of existing data

Update authorization, which allows modification, but not deletion, of data

Delete authorization, which allows deletion of data.

In addition to the above forms of authorization for access to data, a user may be granted authorization to modify the database scheme:

Index authorization, which allow creation and deletion of indices

Resources authorization, which allow the creation new relations
Alteration authorization, which allow the addition or deletion of attributes in a relation
Drop authorization, which allows the deletion of relations

The drop and delete authorization differ in that delete authorization allows deletion of tuples only. If a user deletes all tuples of a relation, the relation still exists, but it is empty. If a relation is dropped, it no longer exists.

The ultimate form of authority is that given to the database administrator. The database administrator may authorize new users, restructure the database, etc. this form of authorization is analogous to that provided to a "super user" or operator for an operating system.

A user who has been granted some form of authority may be allowed to pass this authority on to other users. However, we need to be careful about how authorization may be passed among users in order to ensure that we can revoke authorization at some future time.

Let us consider, as an example, the granting of update authorization on the *deposit* relation of the bank database. Assume that, initially, the database administrator (DBA) grants update authorization on deposit to users $_1$,$_2$, and $_3$. Users $_1$,$_2$, and $_3$ may in turn pass this authorization on to other users. We represent the passages of authorization from one user to another by an *authorization graph*. The nodes of this graph are the users. An edge $(,)$ is included in the graph if user grants update authorization on *deposit* to . A sample graph appears in figure 3.1. observe that user $_5$ is granted authorization by both $_1$ and $_2$.

Suppose that the database administrator decided to revoke the authorization of user $_1$. Since $_4$ has authorization granted from $_1$, that authorization should be revoked as well. However, $_5$ was granted authorization by both $_1$ and $_2$. Since the database administrator did not revoke update authorization on *deposit* from $_2$, $_5$ retains update authorization on *deposit*. If $_2$ eventually revokes authorization from $_5$, then $_5$ loses the authorization.

A pair of devious users might attempt to defeat the above rules for revocation of authorization by granting authorization to each other as shown in figure 3.2a. if the database administrator revokes authorization from $_2$, $_2$ retains authorization through $_3$, as shown in figure 3.2b.

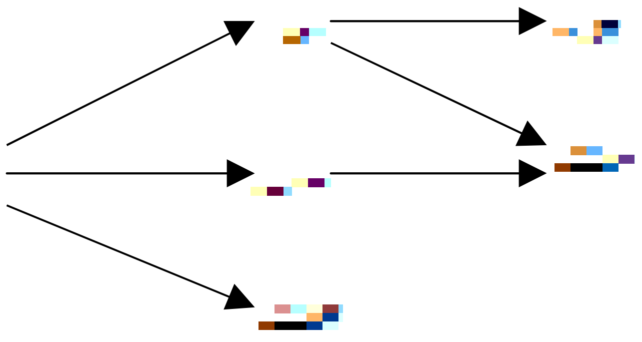


Figure 3.1 An authorization – grant tree

If authorization is revoked subsequently from 3, 3 retains authorization through U2 as shown in figure 3.2c

To avoid problems like that above, we require that all edges in an authorization graph be part of some path originating with the database administration. Under this rule, the authorization graph of figure 3.2b would still be the result of revocation of authorization from 2. However, when the database administrator subsequently revokes authorization from 3, the edges from 3 to 2 and from 2 to 3 are no longer part of a path starting with the database administration. Therefore, those edges are deleted and the resulting authorization graph is as shown in figure 3.3.

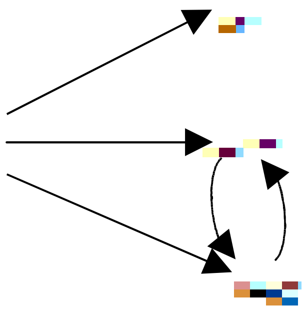
Alternative schemes for managing authorization have been used in several database systems. Several of these are discussed in the bibliographic notes and in the exercises.

As an example of how authorization is granted and revoked in an actual database system, we present examples from SQL. To grant authorization, one uses the **grant** statement :

grant <privilege list> **on** <relation name or view name> **to** <user list>

The *privilege* list allows the granting of several privileges in one command. Members of this list may be the privileges **read, insert, drop, delete, index, alteration**, and **resource**, all of which we have discussed earlier. Update authorization may be given on all attribute of the relation or only some. **Update** authorization is included in a **grant** statement, the list of attributes on which update authorization is to be granted is listed in parentheses: The

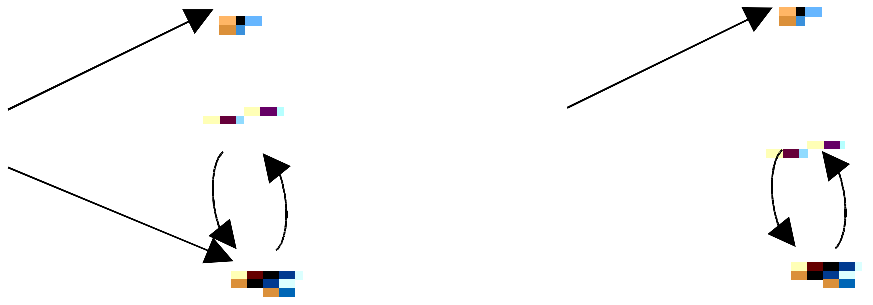
following grant statement grants three users 1, 2, and 3 update authorization on the



(a)

balance attribute of the *deposit* relation:

grant update (*balance*) **on** deposit **to** [illegible]



(b) (c)

Figure 3.2 Attempt to defeat authorization revocation.

To revoke authorization, the **revoke** statement is used. It takes a form almost identical to that of **grant**:

revoke <privilege list>

on <relation name or view name> **from** <user list>

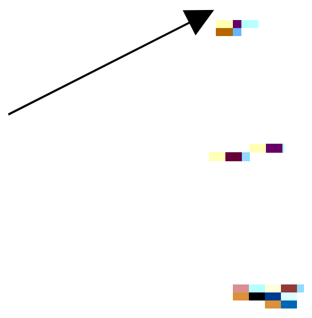
thus, to revoke the privilege we granted above, we write:

revoke update (balance) **on** deposit **from** [illegible]

3.4.INTEGRITYCONSTRAINTS

The forms of authorization discussed earlier are a means by which the database system can be protected against malicious or unauthorized access. Integrity constraints, however, serve a different purpose. They provide a means of ensuring that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damaged to the database.

Figure 3.3 Authorization graph.



Above, we discussed the notion of a transaction. We required that all transactions preserve the constraints, and using that assumption, showed how to preserve consistency despite system crashes. We saw also how ensure that no anomalies result from concurrent access to the database. In practice, of course, programs have bugs and interactive users of the database system make mistakes. The integrity control component of the database system is intended to detect as many of these errors as possible.

We have already seen a form of integrity constraint for relational database in Module 1. Data dependencies (functional, multivalued, and join) are statements about the enterprise we are modelling. We wish to restrict the database to contain only relations that satisfy these dependencies. In the network model and the E-R model, we saw integrity constraints in the form of:

Key declarations, the stipulation that certain attributes form a candidate key for a given entity set constrains the set of legal insertions

Form of a relationship, many-to–many, one–to–many, one–to–one. A one-to–one or one–to-many relationship restricts the set of legal relationships among entities of a collection of entity sets.

Another example of an integrity constraint is set retention in the network model.

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, we usually limit ourselves to integrity constraints that can be tested with minimal overhead. This is the purpose behind dependency – preserving decompositions of relation schemes. Recall that in a dependency – preserving decomposition, it is possible to test for satisfaction of the data dependencies without the need to compute any joins. Domain – key normal (DKNF; see module 1) is an ideal design from the point of view of efficient testing of integrity constraints, since the only forms of constraint that need be tested are key constraints and domain constraints.

If the key and domain constraints are satisfied, and the database scheme is in DKNF, then all integrity constraints on the database are satisfied.

Key constraints are one of the most easily tested forms of consistency constraint, especially if an index is maintained on that candidate key. During the process of inserting a record into the database a lookup must be performed using the index and any duplicate key values that may exist are found. Since not all index search keys are candidate keys for the relation (Indices may be for secondary keys), we need to declare an index to be either

Unique. Only one record may exist for a key value
Non-unique. Multiple records are allowed to have the same key value
Another form of constraint that is easy to test is domain constraints. Testing domain constraints is analogous to runtime – type checking in a programming language. A form of constraint closely related to domain constraints involves the admissibility of null values. We may forbid null values for certain attributes but allow them for others.

Relatively few systems allow the expression of constraints that are more complex than key declarations or domain constraints. The original proposal for the SQL language included a general – purpose construct called the assert statement for the expression of integrity constraints.

An assertion pertaining to a single relation takes the form:

assert < assertion-name> **on** <relation-name><predicate>

For example, if we wish to define an integrity constraint that no account balance is negative we write:

assert balance-constraint **on** deposit

balance $\geq$ 0

In its most general form, the **assert** statement takes the form:

assert <assertion-name> : <predicate>

Let us consider a more complicated constraint involving more than one relation scheme. Suppose that we do not allow a customer to open an account unless the customer appears in the customer relation. We write the following assertion:

assert *address – constraint:*

(**select** *customer – name*

from *customer)*

contains

(**select** *customer-name*

from *deposit*)

When an assertion is made, the system test it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause an assertion to be violated. This testing may introduce a significant amount of overhead if complex assertions have been made.

Because of the high overhead of assertion test, an alternative scheme called *triggering* is sometimes used for integrity preservation. A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database.

To design a trigger mechanism, we must:

specify the conditions under which the trigger is to be executed
specify the actions to be taken when the trigger executes.

Suppose that instead of allowing negative account balances, the bank deals with overdrafts by setting the account balance to zero and creating a loan in the amount of the overdraft. This loan is given a loan number equal to the account number of the overdrawn account. For the above example, the condition for executing the trigger is an update to the *deposit* relation that results in a negative *balance* value. Let t denote the tuple with a negative *balance* value. The actions to be taken are as follows:

Insert a new tuple *s* in the *borrow* relation with:

s[branch-name] = t[branch-name]

s[loan-number]= t[account-number]

s[amount] = - t [balance]

s[customer-name] = t[customer-name]

(Note that since ▯ is negative, we negate ▯ to get the loan amount, a positive number]

Set *t [balance]* to 0.

3.5. ENCRYPTION

The various provisions a database system may make for authorization may not be sufficient protection or highly sensitive data. In such cases, data may be *encrypted*. It is not possible for encrypted data to be read unless the reader knows how to decipher (*decrypt*) the encrypted data.

There are a vast number of techniques for the encryption of data. Simple techniques for encryption may not provide adequate security since it may be easy for an authorized user to break the code. As an example of a bad encryption technique, consider substitution of each character with the next character in the alphabet. Thus:

Perryridge

becomes:

Qfsszshehf

If an unauthorized user sees only “Qfsszsjehf,” there is probably insufficient information to break the code. However, if the intruder sees a large number of *encrypted* branch names, the intruder could use statistical data regarding the relative frequency of characters (for example, “e” is more common than “x”) to guess what substitution is being made.

Good encryption techniques have the following properties:

it is relatively simple for authorized users to encrypt and decrypt data.
The encryption scheme depends not on the secrecy of the algorithm but on a parameter of the algorithm called the *encryption key*.
It is extremely difficult for an intruder to determine the encryption key.

The *Data Encryption Stan*dard is an approach which does both a substitution of characters and a rearrangement of their order based on an encryption key. In order for this scheme to work, the authorized users must be provided with the encryption key via a secure mechanism. This is a major weakness since the scheme is no more secure than the secureness of the mechanism by which the encryption key is transmitted.

There is an alternative scheme that avoids some of the problems with the Data Encryption Standard. This scheme, called *public–key encryption*, is based on two keys, a *public key* and *private key*. Each user has his or her own public key and private key . All public keys are published. Each user’s private key is known only to the one user to whom the key belongs. If user $_1$ wants to store encrypted data, $_1$ encrypts it using his or her public key $_1$. decryption requires the private key $_1$.

Because the encryption key for each user is public, it is possible to exchange information securely using this scheme. if user $_1$ wants to share data with $_2$, U1 encrypts the data using $_2$, the public key of $_2$. Since only user $_2$ knows how to decrypt the data, secure information transfer is accomplished.

For public key encryption to work, there must be a scheme for encryption that can be made public without making it easy to figure out the scheme for decryption. Such a scheme does exist. It is based on the following:

There is an efficient algorithm for testing whether or not a number is prime
No efficient algorithm is known for finding the prime factors of a number.

Data is treated as a collection of integers for purposes of this scheme. A public key is created by computing the product of two large prime numbers $_1$ and $_2$. The private key consists of the $(_1, _2)$, and the decryption algorithm cannot be used successfully if only the product $_{12}$ is known. Since all that is published in the product $_{12}$, an unauthorized user would need to be able to factor $_{12}$ in order to steal data. By choosing $_1$ and $_2$ to be sufficiently large (over 100 digits). We can make the cost of factoring $_{12}$ prohibitively high (on the order of years of computation time even on the fastest computers). The details of public key encryption and the mathematical justification of its properties are referenced in the bibliographic notes.

3.6. STATISTICAL DATABASES

Suppose that our bank grants an outsider access to its database under the condition that only statistical studies(averages, medians, etc.) are made on the data and that information about individuals is not divulged. In this unit, we examine the difficulty of ensuring the privacy of individuals while allowing use of data for statistical purposes.

One weakness in a statistical database is unusual cases. For example, there may be a city in which only one bank customer lives. Suppose that one asks for the total bank account balances for all customers living in Smalltown. If only one customer happens to live in Smalltown, the system has divulged information about an individual. Of course, a security breach has occurred only if the user knows that only one customer lives in Smalltown. However, that information is easily determined by the statistical query, “Find the number of customers living in Smalltown”.

A simple way to deal with potential security breaches like that described above is for the system to reject any query that involves fewer than some predetermined number of individuals. Suppose this predetermined number is n. A malicious user who has an account with our bank can find an individual’s balance in two queries. Suppose he wants to find how much money Rollo has on deposit. He chooses *n* customers and finds:

x, the total balance for himself and the n customers.

y, the total balances for Rollo and the n customers.

Rollo’s total balance is:

$$- \quad + \quad '$$

The critical flow that was exploited in the above example is that the two queries referred to many of the same data items. The number of data items the queries $_1$ and $_2$ have in common is called the intersection of $_1$ and $_2$.

Thus, in addition to requiring that a query reference data pertaining to at least *n* individuals, we may require that no two queries have an intersection larger than *m*. By adjusting *n* and *m*, we can increase the difficulty of a user determining data about an individual, but we cannot eliminate it entirely.

These two restrictions do not preclude the possibility of some extremely clever query that divulges data. However, if all queries are restricted to computing sums, counts, or averages, and if a malicious user knows only the data value for himself, it can be shown that it will take at least $1 + (- 2)/$ queries for the malicious user to determine data about an individual. The proof of this is beyond the scope of this text and is referenced in the bibliographic notes. this act is only partially reassuring. We can limit a user to less than

$1 + (- 2)/$ queries, but a conspiracy of two malicious users can result in data being divulged.

Another approach to security is data pollution. This involves the random falsification of data provided in response to a query. This falsification must be done in such a way that the statistical significance of the response is not destroyed. A similar technique involves random modification of the query itself. For both of these techniques, the goals involve a trade – off between accuracy and security.

Regardless of the approach taken to security of statistical data, it is possible for a malicious user to determine individual data values. However, good techniques can make the expense in terms of cost and time sufficiently high to be a deterrent.

3.7. CONCLUSION

The various provisions a database system may make for authorization may not sufficient protection for highly sensitive data. In such cases, data may be *encrypted*. It is not possible for encrypted data to be read unless the reader knows how to decipher (*decrypt*) the encrypted data.

It is difficult to ensure the privacy of individuals while allowing use of data for statistical purposes. A simple way to deal with potential security breaches is for the system to reject any query that involves fewer than some predetermined number of individuals. Another approach to security is *data pollution*. This involves the random falsification of data provided in response to a query. A similar technique involves random modification of the query itself. For both of these techniques, the goals involve a trade-off between accuracy and security . Regardless of the approach taken to security of statistical data, it is possible for a malicious user to determine individual data values. However, good techniques can make the expense in terms of cost and time sufficiently high to be a deterrent.

3.8. SUMMARY

The data stored in the database needs to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. It is easier to protect against accidental loss of data consistency than to protect against malicious access to the database. Absolute protection of the database from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most if not all attempts to access the database without proper authority.

The concept of *views* provides a means for a user to design a "personalized" model of the database. A view can hide data that a user does not need to see. Security is provided if there is a mechanism to restrict the user to his or her personal view or views. A combination of relational-level security and view-level security can be used to limit a user's access to precisely the data that user needs.

A user may have several forms of authorization on parts of the database. Authorization is a means by which the database system can be protected against malicious or unauthorized access. A user who has been granted some form of authority may be allowed to pass this authority on to other uses. However, we need to be careful about how authorization may be passed among users in order to ensure that we can revoke authorization at some future time.

Integrity constraints provide a means of ensuring that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database. In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, we usually limited ourselves to integrity constraints that can be tested with minimal overhead.

Because of the high overhead of assertion test, an alternative scheme called *triggering* is sometimes used for integrity preservation. A trigger is a statement that is executed automatically by the system as a side effect of modification to the database.

3.10. TUTOR-MARKED ASSIGNMENT (TMA)

1. Mention the causes of data loss from database.
2. How can accidental loss of data consistency be prevented?
3. Discuss the techniques of ensuring database security.

3.11. FURTHER READINGS

Date, C. [1995] An Introduction Database System, 7th ed; Addison-Wesly 2000.

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[1986]. "Database System Concepts", McGraw – hill, 1986.

www.w3schools.com

Zobel, J., Moffat, A., and Sacks–Davis, R., [1992]. "An Efficient Indexing Technique for Full – Text Database Systems," in VLDB, 1992.

MODULE 3: DISTRIBUTED DATABASES

UNIT 1: ENHANCED DATABASE MODEL

1.0. INTRODUCTION

A single database can be divided into several fragments. The fragments can be stored on different computers within a network. Processing, too, can be dispersed among several different network sites or nodes. The multisite database forms the core of the distributed database systems.

1.1. OBJECTIVES

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

differentiate between centralized and distributed databases
design a distributed database
database atomicity

discuss distributed query processing

discuss the concurrency control schemes

1.2 DISTRIBUTED DATABASES

In a distributed database system, the database is stored on several computers. The computers in a distributed system communicate with each other through various communication media, such as high–speed buses or telephone lines. They do not share main memory, nor do they share a clock.

The processors in a distributed system may vary in size and function. They may include small microcomputers, work stations, minicomputers, and large general–purpose computer systems.

These processors are referred to by a number of different names such as *sites*, *nodes*, *computers*, and so on, depending on the context in which they mentioned, we mainly use the term *site*, in order to emphasize the physical distribution of these systems.

A distributed database system consists of a collection of sites, each of which may participate in the execution of transactions which access data of one site, or several sites. The main difference between centralized and distributed database systems is that, in the former, the data resides in one single location, while in the latter, the data resides in several locations. As we shall see, this distribution of data is the cause of many difficulties that will be addressed in this unit.

1.3 STRUCTURE OF DISTRIBUTED DATABASE

A distributed database system consists of a collection of sites, each of which maintains a local database system. Each site is able to process *local transactions*, those transactions that access data only in that single site. In addition, a site may participate in the execution of *global transactions*, those transactions that access data in several sites. The execution of global transactions requires communication among the sites.

The sites in the system can be connected physically in a variety of ways. The various topologies are represented as graphs whose nodes correspond to sites. An edge from node *A* to node *B* corresponds to a direct connection between the two sites. Some of the most common configurations are depicted in figure 1.1. the major differences among these configurations involve:

Installation cost. The cost of physical linking the sites in the system.

Communication cost. The cost in time and money to send a message from *site A* to *site B*.

Reliability. The frequency with which a link or site fails.

Availability. The degree to which data can be accessed despite the failure of some links or sites.

As we shall see, these differences play an important role in choosing the appropriate mechanism for handling the distribution of data.

The sites of a distributed database system may be distributed physically either over a large geographical area (such as the United States), or over a small geographical area (such as a single building or a number of adjacent buildings). The former type of network is referred to as a *long–haul* network, while the latter is referred to as a *local–area* network.

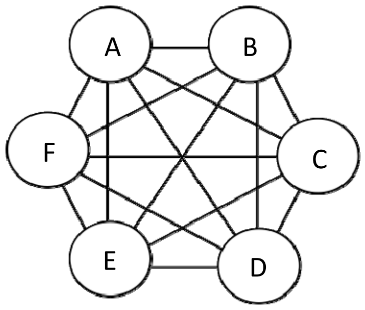
Since the sites in long–haul networks are distributed physically over a large geographical area, the communication links are likely to be relatively slow and less reliable as compared with local–area networks. Typical long-haul links are telephone lines, microwaves links, and satellite channels. In contrast, since all the sites in local-area networks are close to each other, the communication links are of higher speed and lower error rate than their counterparts in long-haul networks. The most common links are twisted pair, based band coaxial, broadband coaxial, and fiber optics.

Let us illustrate these concepts by considering a banking system consisting of four branches located in four different cities. Each branch has its own computer with a database consisting of all the accounts maintained at that branch. Each such installation is thus a site. There also exists one single site which maintains information about all the branches of the bank. Suppose that the database systems at the various sites are based on the relational model. Thus, each branch maintains (among others) the relation *deposit (Deposit–scheme)* where

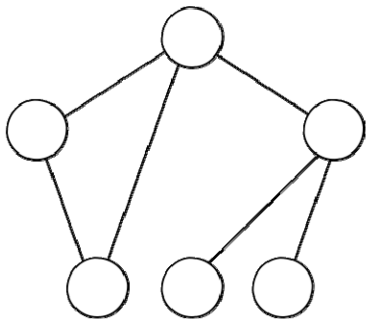
$$– = (–, –, –,)$$

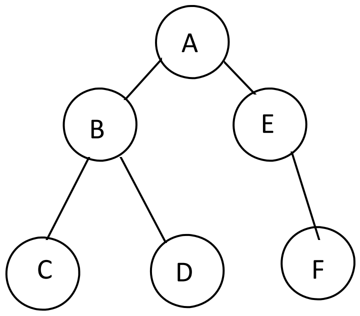
The site containing information about the four branches maintains the relation *branch (Branch–scheme)*, where

$$– = (–,, –)$$

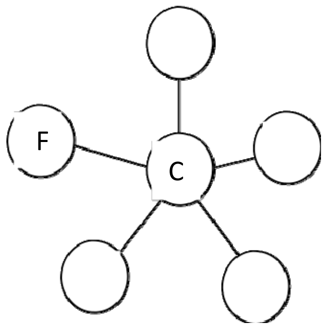


fully connected network

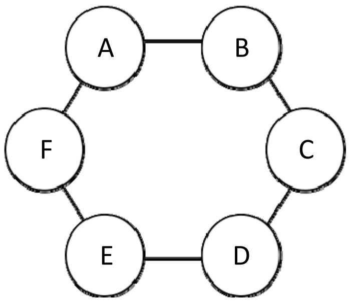

partially connected network



Tree structured



star network



Ring network

Figure 1.1 Network topology

There are other relations maintained at the various sites which are ignored for the purpose of our example. A local transaction is a transaction that accesses accounts in the *one single site*, at which the transaction was initiated. A global transaction, on the other hand, is one which either accesses accounts in a site different from the one at which the transaction was initiated, or accesses accounts in several different sites. To illustrate the difference between these two types of transactions, consider the transaction to add N50 to account number 177 located at the Valleyview branch. If the transaction was initiated at the Valleyview branch, then it is considered local; otherwise it is considered global. A transaction to transfer N50 from account 177 to account 305, which is located at the hillside branch, is a global transaction since accounts in two different sites are accessed as a result of its execution.

What makes the above configuration a distributed database system are the facts that:

The various sites are aware of each other.

Each site provides an environment for executing both local and global transactions.

1.4 TRADE–OFFS IN DISTRIBUTING THE DATABASE

There are several reasons for building distributed database systems, including sharing of data, reliability and availability, and speedup of query processing. However, along with these advantages come several disadvantages, including software development cost, greater potential for bugs, and increased processing overhead. In this unit, we shall elaborate briefly on each of these.

1.4.1 Advantages of Data Distribution

The primary advantage of distributed database systems is the ability to share and access data in a reliable and efficient manner.

Data Sharing and Distributed Control

If a number of different sites are connected to each other, then a user at one site may be able to access data that is available at another site. For example, in the distributed banking system described in unit 1.3, it is possible for a user in one branch to access data in another branch. Without this capability, a user wishing to transfer funds from one branch to another would have a resort to some external mechanism for such a transfer. This external mechanism would, in effect, be a single centralized database.

The primary advantage to accomplishing data sharing by means of data distribution is that each site is able to retain a degree of control over data stored locally. In a centralized system, the database administrator of the central site controls the database. In a distributed system, there is a global database administrator responsible for the entire system. A part of these responsibilities is delegated to the local database administrator for each site. Depending upon the design of the distributed database system, each local administrator may have a different degree of autonomy. This is referred to as *local autonomy*. The possibility of local autonomy is often a major advantage of distributed databases.

Reliability and Availability

If one site fails in a distributed system, the remaining sites may be able to continue operating. In particular, if data are replicated in several sites, a transaction needing a particular data item may find it in several sites. Thus, the failure of a site does not necessarily imply the shutdown of the system.

The failure of one site must be detected by the system, and appropriate action may be needed to recover from the failure. The system must no longer use the services of the failed site. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it smoothly back into the system.

Although recovery from failure is more complex in distributed systems than in centralized systems, the ability of most of the system to continue to operate despite the failure of one site results in increased availability. Availability is crucial for database systems used for real–time applications. Loss of access to data by, for example, an airline may result in the loss of potential ticket buyers to competitors.

Speedup of Query Processing

If a query involves data at several sites, it may be possible to split the query into subqueries that can be executed in parallel by several sites. Such parallel computation allows for faster processing of a user's query. In those cases in which data is replicated, queries may be directed by the system to the least heavily loaded sites.

1.4.2 Disadvantages of Data Distribution

The primary disadvantage of distributed database systems is the added complexity required to ensure proper coordination among the sites. This increased complexity takes the form of:

Software development cost. It is more difficult to implement a distributed database system and thus, more costly.

Greater potential for bugs. Since the sites that comprise the distributed system operate in parallel,, it is harder to ensure the correctness of algorithms. The potential exists for extremely subtle bugs. The art of constructing distributed algorithms remains an active and important area of research.

Increased processing overhead. The exchange of messages and the additional computation required to achieve inter-site coordination is a form of overhead that does not arise in centralized systems.

In choosing the design for a database system, the designer must balance the advantages against the disadvantages of distribution of data. We shall see that there are several approaches of distributed database design ranging from fully distributed designs to designs which include a large degree of centralization.

1.5 DESIGN OF DISTRIBUTED DATABASES

The principles of database design that we discussed earlier apply to distributed databases as well. In this unit, we focus on those design issues that are specific to distributed databases.

Consider a relation *r* that is to be stored in the database. There are several issues involved in storing this relation in the distributed database including:

Replication. The system maintains several identical replicas (copies) of the relation. Each replica is stored in a different site, resulting in data replication. The alternative to replication is to store only one copy of relation *r*.

Fragmentation. The relation is partitioned into several fragments. Each fragment is stored in a different site.

Replication and fragmentation. This is a combination of the above two notions. The relation is partitioned into several fragments. The system maintains several identical replicas of each such fragment.

In the following subunits, we elaborate on each of these.

1.5.1 Data Replication

If relation *r* is replicated, a copy of relation *r* is stored in two or more sites. In the most extreme case, we have *full replication*, in which a copy is stored in every site in the system.

Availability. If one of the sites containing relation r fails, then the relation *r* may be found in another site. Thus, the system may continue to process queries involving *r* despite the failure of one site.

Increased parallelism. In the case where the majority of access to the relation *r* results in only reading of the relation, then several sites can process queries involving *r* in parallel. The more replicas of *r* there are, the greater the chance that the needed data is found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites.

Increased overhead on update. The system must ensure that all replicas of a relation *r* are consistent since otherwise erroneous computations may result. This implies that whenever *r* is updated, this update must be propagated to all sites containing replicas, resulting in increased overhead. For example, a in a banking system, where account information is replicated in various sites, it is necessary that transactions assure that the balance in a particular account agrees in all sites.

In general, replication enhances the performance of read operations and increases the availability of data to read transactions. However, update transaction incur greater overhead. The problem of controlling concurrent updates by several transactions to replicated data is more complex than the centralized approach to concurrency control. We may simplify the management of replicas of relation *r* by choosing one of them as the *primary copy of r*. For example, in banking system, an account may be associated with the site in which the account has been opened. Similarly, in an airline reservation system, a flight may be associated with the site at which the flight originates. We shall examine the options for distributed concurrency control in unit 1.9.

1.5.2 Data Fragmentation

If the relation *r* if fragmented, *r* is divided into a number of *fragments* [illegible]. These fragments contain sufficient information to reconstruct the original relation *r*. As we shall see, this reconstruction can take place through the application of either the union operation or a special type of join operation on the various fragments. There are two different schemes for fragmenting a relation: *horizontal* fragmentation and *vertical* fragmentation. Horizontal fragmentation splits the relation by assigning each tuple of *r* to one or more fragments. Vertical fragmentation splits the relation by decomposing the scheme *R* of relation *r* in a special way that we shall discuss. These two schemes can be applied successively to the same relation, resulting in a number of different fragments. Note that some information may appear in several fragments.

Below we discuss the various ways for fragmenting a relation. We shall illustrate these by fragmenting the relation deposit, with scheme:

[illegible]

The relations *deposit (deposit–scheme)* is shown in Table 1.1

Table 1.1 sample *deposit* relation

Branch-name	*Account – number*	*Customer- name*	*Balance*
Hillside	305	Lowman	500
Hillside	226	Camp	336
Valleyiew	117	Camp	205
Valleyview	402	Kahn	10000

Hillside	155	Kahn	62
Valleyview	408	Kahn	1123
Valleyview	639	Green	750

Horizontal Fragmentation

The relation *r* is partitioned into a number of subsets, 1, 2, ..., . Each subset consists of a number of tuples of relation *r*. Each tuple of relation r must belong to one of the fragments, so that the original relation can be reconstructed, if needed.

A fragment may be defined as a selection on the global relation *r*. that is, a predicate

used to construct fragment as follows:

$$= ()$$

The reconstruction of the relation *r* can be obtained by taking the union of all fragments, that is,

$$= \bigcup_{=1}$$

To illustrate this, suppose that the relation *r* is the *deposit* relation of Table 1.1. This relation can be divided into *n* different fragments, each of which consists of tuples of accounts belonging to a particular branch. If the

Table 1.2 Horizontal fragmentation of relation deposit

Branch – name	*Account–number*	*Customer–name*	*Balance*
Hillside	305	Lowman	500
Hillside	226	Camp	336
Hillside	155	Kahn	62

(a)

Branch – name	*Account–number*	*Customer- name*	*Balance*
Valleyview	177	Camp	205
Valleyview	402	Kahn	10000
Valleyview	408	Kahn	1123

Valleyview	639	Green	750

(b)

banking system has only two branches, Hillside and Valleyview, then there are two different fragments:

$$_1 = . = ^{\prime\prime\prime}()$$

$$_2 = . = ^{\prime\prime\prime}()$$

These two fragments are shown in Table 1.2. Fragment $_1$ is stored in the Hillside site. Fragment $_2$ is stored in the Valleyview site.

In our example, the fragments are disjoint. By changing the selection predicates used to construct the fragments, we may have a particular tuple of *r* appear in more than one of the . This is a form of data replication about which we shall say more at the end of this unit.

Vertical Fragmentation

In its most simple form, vertical fragmentation is the same as decomposition. Vertical fragmentation of () involves the definition of several subsets $1, 2, \dots,$ of *R* such that $\bigcup_{=1} =$. Each fragment of *r* is defined by:

$$= \Pi()$$

relation *r* can be reconstructed from the fragments by taking the natural join:

$$= {}_1 \bowtie {}_2 \bowtie {}_3 \bowtie \dots \bowtie$$

Table 1.3 the deposit relation of Table 1.1 with tuple-ids.

Branch – name	*Account-number*	*Customer-name*	*Balance*	*Tuple- id*
Hillside	305	Lowman	500	1
Hillside	226	Camp	336	2
Valleyview	177	Camp	205	3
Valleyview	402	Kahn	10000	4
Hillside	155	Kahn	62	5
Valleyview	408	Kahn	1123	6

Valleyview	639	Greeen	750	7

More generally, vertical fragmentation is accomplished by adding a special attribute called a *tuple–id* to the scheme *R*. A tuple-id is a physical or local address for a tuple. Since each tuple in *r* must have a unique address, the *tuple-id* attribute is a key for the augmented scheme.

In Table 1.3, we show the relation *deposit'*, the *deposit* relation of Table 1.1 with tuple-ids added. Table 1.4 shows a vertical decomposition of the scheme $- \cup -$ into:

$$- - 3 = (-, -, -)$$

$$- - 4 = (-,, -)$$

The two relations shown in Table 1.4 result from computing:

$$_3 = \Pi_{- -3} \circ$$

$$_4 = \Pi_{- -4} \circ$$

Table 1.4 Vertical fragmentation of relation *deposit*.

Branch- name	*Customer – name*	*Tuple – id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

(a)

Account – number	*Balance*	*Tuple – id*
305	500	1
226	336	2
177	205	3

402	10000	4
155	62	5
408	1123	6
639	750	7

(b)

To reconstruct the original *deposit* relation from the fragments, we compute

$$\Pi_{-}({}_3 \bowtie {}_4)$$

Note that the expression

$${}_3 \bowtie {}_4$$

is a special from of natural join. The join attribute is *tuple–id*. Since the *tuple–id* value represents an address, it is possible to pair a tuple of $_3$ with the corresponding tuple of $_4$ by using the address given by the *tuple–id* value. This address allows direct retrieval of the tuple without the need for an index. Thus, this natural join may be computed much more efficiently than typical natural joins.

Although the *tuple–id* attribute is important in the implementation of vertical partitioning, it is important that this attribute not be visible to users. If users are given access to tuple–ids, it becomes impossible for the system to change tuple addresses. Furthermore, the accessibility of internal addresses violates the notion of data independence, one of the main virtues of the relational model.

Mixed Fragmentation

The relation *r* is divided into a number of fragment relations [illegible]. Each fragment is obtained as the result of applying either horizontal fragmentation or vertical fragmentation scheme on relation *r*, or a fragment of *r* which was obtained previously.

To illustrate this, suppose that the relation *r* is the deposit relation of Table 1.1. This relation is divided initially into the fragments $_3$ and $_4$ as defined above. We can now further divide fragment $_3$ using the horizontal fragmentation scheme into the following two fragments:

$${}_3 = [illegible]({}_3)$$

$${}_3 = [illegible]({}_3)$$

Thus relation *r* is divided into three fragments $_3$, $_3$, and $_4$. Each of these may reside in a different site.

1.5.3 Data Replication and Fragmentation

The technique described above for data replication and data fragmentation can be applied successively to the same relation. That is, a fragment can be replicated; replicas can be fragmented; etc. For example, consider a distributed system consisting of sites $1, 2, \ldots, 10$. We can fragment deposit into $_3$, $_3$, and $_4$, and, for example, store a copy of $_3$ in site S_1, S_3, and S_7, a copy of $_3$ at sites S_7 and S_{10}, and a copy of $_4$ at sites S_2, S_8, and S_9.

1.6 TRANSPARENCY AND AUTONOMY

In the previous unit, we saw that a relation *r* may be stored in a variety of ways in a distributed database system. It is essential that the system minimize the degree to which a user needs to be aware of how a relation is stored. As we shall see, a system can hide the details of the distribution of data in the network. We call this *network transparency*.

Network transparency is related, in some sense, to the issue of local autonomy. Network transparency is the degree to which system users may remain unaware of the details of the design of the distributed system. Local autonomy is the degree to which a designer or administrator of one site may be independent of the remainder of the distributed system.

We shall consider the issues of transparency and autonomy from the points of view of:

Naming of data items.

Replication of data items.

Fragmentation of data items.

Location of fragments and replicas.

1.6.1 Naming and Local Autonomy

Every data item in the database must have a unique name. this property is easy to ensure in a non distributed database. However, in a distributed database, the various sites must ensure that two sites do not use the same name for distinct data items.

One solution to this problem is to require all names to be registered in a central *name–server*. This approach, however, suffers from several disadvantages:

The name server may become a bottleneck.

If the name server crashes, it may not be possible for any site in the distributed system to continue to run.

There is little local autonomy since naming is controlled centrally.

An alternative approach that results in increased local autonomy is to require that each site prefix its own site identifier to any name it generates. This ensures that no two sites generate the same name (since each site has a unique identifier). Furthermore, no central control is required.

The above solution to the naming problem achieves local autonomy, but fails to achieve network transparency since site identifiers are attached to names. Thus, the *deposit* relation might be referred to as *site17.deposit* rather than simply *deposit*. We shall soon see how to overcome this problem.

Each replica of a data item and each fragment of a data item must have a unique name. it is important that the system be able to determine those replicas that are replicas of the same data item and those fragments that are fragments of the same data item. We adopt the convention of post fixing $“.1”, “.2”, \ldots, “.”$ to fragments of a data item and $“.1”, “.2”, \ldots, “.”$ to replicas. Thus

$$17..3.2$$

refers to replica 2 of fragment 3 of *deposit*, and this item was generated by site 17.

1.6.2 Replication and Fragmentation Transparency

It is undesirable to expect users to refer to a specific replica of a data item. Instead, the system should determine which replica to reference on a read request, and update all replicas on a modification request.

When a data item is requested, the specific replica need not be named. Instead, a catalog table is used by the system to determine all replicas for the data item.

Similarly, a user should not be required to know how a data item is fragmented. As we observed earlier, vertical fragments may contain *tuple–ids*, which represent addresses of tuples. Horizontal fragments may involve complicated selection predicates. Therefore, a distributed database system should allow request to be stated in terms of the unfragmented data items. This presents no major difficulty, since it is always possible to reconstruct the original data item from its fragments. However, it may be inefficient to reconstruct data from fragments. Returning to our horizontal fragmentation of deposit, consider the query:

$$- = [illegible]$$

This query could be answered using only the $_1$ fragment. However, fragmentation transparency requires that the user not be aware of the existence of fragments $_1$ and $_2$. If we reconstruct deposit prior to processing the above query, we obtain the expression:

$$- = \text{[illegible]}(_1 \cup _2)$$

The optimization of this expression is left to the query optimizer (see unit 1.7)

1.6.3 Location Transparency

If we have replication and fragmentation transparency provided by the system a large part of the design of the distributed database is hidden from the user. However, the site–identifier component of names forces the user to be aware of the fact that the system is distributed.

Location transparency is achieved by creating a set of alternative names or *aliases* for each user. A user may thus refer to data items by simple names that are translated by the system to complete names.

By using aliases, the user can be unaware of the physical location of a data item. Furthermore, the user is unaffected if the database administrator should decide to move a data item from one site to another.

1.6.4 Complete Naming Scheme

We have seen that a name provided by the user is translated in several steps before it refers to a specific fragment at a specific site. Figure 1.2 shows the complete translation scheme. To illustrate the operation of the scheme, consider a user located in the Hillside branch (site S_1). This users use the alias *local–deposit* for the local fragment $._1$ of the deposit relation. When this user references *local–deposit*, the query processing subsystem look up *local–deposit* in the alias table and replaces it with $1..1$ it is possible that $1..1$ is replicated. If so, the replica table must be consulted in order to choose a replica. This replica could itself be fragmented, requiring examination of the fragmentation table. In most cases, only one or two tables must be consulted. However, the name translation scheme of Figure 1.2 is sufficiently general to deal with any combination of successive replication and fragmentation of relations.

1.6.5 Transparency and Updates to Replicated Data

Providing transparency for users that update the database is somewhat more difficult than providing transparency for readers. The main problem is ensuring that all replicas of a data item are updated and that all affected fragments are updated. In its full generality, the update problem for

```
        If name appears in the alias table
            then expression := map (name)
        **else** *expression* : = *name*;


        function *map* (*n*)
        **if** *n* appears in the replica table
            **then** n := name of replica of *n*;
        **if** *n* appears in the fragment table
        **then begin**
            *result* := expression to construct fragment;
            **for each** *n′* **in** *result* **do begin**
            replace *n′* in *result* with *map* (*n′*);
        **end**
    **return** *result*;
```

Figure 1.2 Name translation algorithm.

replicated and fragmented data is related to the view update problem that we discussed earlier.

Consider our example of the deposit relation and the insertion of the tuple:

(Valleyview, 733, Jones, 600)

If deposit is fragmented horizontally, there is a predicate associated with the fragment. We apply to the tuple (Valleyview, 733, Jones, 600) to test if that tuple must be inserted in the fragment. Using our example of deposit being fragmented into

$_1$ = [illegible]

$_2$ [illegible]

the tuple would be inserted into $_2$.

Now consider a vertical fragmentation of deposit into $_3$ and$_4$. The tuple (Valleyview, 733, Jones, 600) must be split into two fragments: one to be inserted into $_3$ and one to be inserted into $_4$.

If the deposit relation is replicated, the tuple (Valleyview, 733, Jones, 600) must be inserted in all replicas. This presents a problem if there is concurrent access to the deposit relation, since it is possible that one replica is updated earlier than another. We consider this problem in unit 1.9.

1.7 DISTRIBUTED QUERY PROCESSING

For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses. In a distributed system, we must take into account several other issues, including:

The cost of data transmission over the network

The potential gain in performance from having several sites process parts of the query in parallel.

The relative cost of data transfer over the network and data transfer to and from disk varies widely depending on the type of network and speed of the disks. Thus, in general, we cannot focus solely on disk cost or on network costs. Rather, we must find a good trade–off between the two. We shall emphasize the relational model although most of our techniques are applicable to the other models.

1.7.1 Replication and Fragmentation

Let us consider an extremely simple query, "Find all the tuples in the *deposit* relation". Although the query is simple, and indeed trivial, processing of this query is not trivial since the *deposit* relation may be fragmented, replicated, or both, as we saw in unit 1.4. If the deposit relation is replicated, we have a choice of replica to make. If no replicas are fragmented, we choose the replica for which the transmission cost is lowest. However, if a replica is fragmented, the choice is not as easy to make since several joins or unions need to be computed to reconstruct the *deposit* relation. In this case, the number of strategies for our simple example may be large. Indeed, choosing a strategy may be a complex a task as an arbitrary query.

Fragmentation transparency implies that a user may write a query such

$$\_ = [illegible]()$$

Since deposit is defined as

$$_1 \cup _2$$

the expression that results from the name translation scheme is

$$\_ = [illegible](_1 \cup _2)$$

Using query optimization techniques, we can simplify the above expression automatically.

1.7.2 Simple Join Processing

A major aspect of the selection of a query processing strategy is choosing a join strategy. Consider the relation algebra expression:

$$\bowtie \quad \bowtie$$

Assume that the three relations are neither replicated nor fragmented and that customer is stored at site, *deposit* at, and branch at . Let denote the site at which the query was issued. The system needs to produce the result at site. Among the possible strategies for processing this query are the following:

Ship copies of all three relations to site ${}_1$, then choose a strategy for processing the entire query locally at site S_1

Ship a copy of the customer relation to site and compute $\bowtie$ at . Ship $\bowtie$ from to where $(\bowtie)\bowtie$ is computed. The result of this computation is shipped to

- Strategies similar to the one above may be devised with the roles of , exchanged.

It is not the case that one strategy is always the best one. Among the factors that must be considered are the amount of data being shipped, the cost of transmitting a block of data between a pair of sites, and the relative speed of processing at each site. Consider the first two strategies listed above. If we ship all three relations to and indices exist on these relations, we may need to recreate these indices at. This entails extra processing overhead and extra disk accesses. However, the second strategy has the disadvantage that a potentially large relation $(\bowtie)$ must be shipped from to. This relation repeats the address data for a custosmer oncefor each account the customer has. Thus, the second strategy may result in extra network transmission as compared with the first strategy.

1.7.3 Join Strategies that Exploit Parallelism

Let us consider a join of our relations:

$${}_1 \bowtie {}_2 \bowtie {}_3 \bowtie {}_4$$

where relation is stored at site . Assume that the result must be presented at site. There are of course, many strategies to be considered. One attractive strategy is to compute two joins in parallel. For example, ${}_1$ can be shipped to ${}_2$ and ${}_1 \bowtie {}_2$ computed at ${}_2$. At the same time, ${}_3$ can be shipped to ${}_4$ and ${}_3 \bowtie {}_4$ computed at ${}_4$. We are then left with the task of computing:

$$({}_1 \bowtie {}_2) \bowtie ({}_3 \bowtie {}_4)$$

where $({}_1 \bowtie {}_2)$ is stored at site ${}_2$ and $({}_3 \bowtie {}_4)$ is stored at site ${}_4$. Although we need to compute three natural joins to answer the query ${}_1 \bowtie {}_2 \bowtie {}_3 \bowtie {}_4$, the use of parallelism allows the query to be computed by the distributed system in the time it would have taken a single processor to compute two joins.

To achieve even greater parallelism in the above example, site $_2$can ship tuples of $(_1 \bowtie {}_2)$ to as they are produced rather than waiting for the entire join to be computed. Similarly, $_4$can ship tuples of $(_3 \bowtie {}_4)$ to . Once tuples of $(_1 \bowtie {}_2)$ and $(_3 \bowtie {}_4)$ arrive at , site can begin the computation of $(_1 \bowtie {}_2) \bowtie (_3 \bowtie {}_4)$ in parallel with the computation of $(_1 \bowtie {}_2)$ at $_2$ and the computation of $(_3 \bowtie {}_4)$ at $_4$.

1.7.4 Semijoin Strategies

In situations in which network transmission costs are high, the *semijoin* operator, denoted by$\ltimes$, can be used for efficient processing of queries involving joins. Let *r* and *s* be relations on schemes *R* and *S*, respectively. The *semijoin* of r and s, denoted $\ltimes$, is

$$\Pi_R(\ \bowtie\ )$$

Note that, in general, $\ltimes \neq \ltimes$. The expression $\ltimes$ selects those tuples of *r* that participate in $\ltimes$. That is, it selects those tuples in *r* such that there is a tuple in *s* such that:

$$[(\ \cap\ ]\ =\ [\ \cap\ ]$$

We use this fact in a distributed system to avoid the shipment over the network of tuples that are not needed to compute a given join.

Consider the expression $1 \bowtie 2$, where r1 and r2 are stored at sites $_1$ and$_2$, respectively. Let the schemes of $_1$ $_2$ be R1 and R2/ The following is a semijoin strategy for computing $_1$ $_2$ and producing the result at $_1$:

1. Compute *temp*1 = $\Pi_{1 \cap 2}(_1)$ at$_1$.
2 Ship *temp*1 from $_1$ to$_2$.
3. Compute *temp*2 = $_2 \ltimes \Pi_1 \cap {}_2(_1)$at $_2$
4. Ship *temp*2 from $_2$ to $_1$
5. Computer $_1 \bowtie 2$ at $_1$.

For the above strategy to be correct, $_1 \bowtie {}_2$ must equal:

$$_1 \bowtie (_2 \ltimes \Pi_{1 \cap 2}(_1))$$

To see this, note that the definition of *semijoin*, implies that $(_2 \ltimes \Pi_{1 \cap 2}(_1)$ is:

$$\Pi_2(_2 \bowtie \Pi_{1 \cap 2}(_1))$$

This is equivalent to:

$$\Pi_2(_2 \bowtie _1)$$

Or:

$$_2 \ltimes _1$$

Thus,

$$_1 \bowtie (_2 \ltimes \Pi_{1 \cap 2}(_1)) = {}_1 \bowtie (\Pi_2(_2 \bowtie 1)) = {}_1 \bowtie _2$$

The strategy thus generates the correct result. Let us now examine why this strategy may be advantageous in a distributed system. Suppose that relatively few tuples of $_1$ and $_2$ participate in the join. In such a case, $_2 \ltimes _1$ has many fewer tuples than$_2$. Thus, it is significantly less costly to ship $_2 \ltimes _1$ from $_2$ to $_1$ than to ship all of $_2$ from $_2$ to$_1$. The above semijoin strategy does require the shipment of $\Pi_{1 \cap 2}(_1)$ from $_1$ to$_2$, but this cost may be deminated by the savings from the use of the semijoin.

It may appear that *semijoin* strategies are useful only if there are a large number of dangling tuples. Such situations arise primarily in cases where $_1$ or $_2$ are not relations in the database but rather the result of a relational algebra expression.

A substantial body of theory has been developed regarding the use of *semijoins* for query optimization.

1.8 RECOVERY IN DISTRIBUTED SYSTEMS

A transaction must be executed atomically. That is, the instructions associated with it are either all executed to completion, or none are performed. In addition, in the case of concurrent execution, the effect of executing a transaction must be the same as if the transaction executed alone in the system.

1.8.1 System Structure

When dealing with a distributed database system, however, it becomes much more complicated to ensure the atomicity property of a transaction. This is due to the fact that several sites may be participating in the execution of the transaction. The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

It is the function of the transaction manager of a distributed database system to ensure that the execution of the various transactions in the distributed system preserves atomicity. Each site has its own local transaction manager. The various transaction managers cooperate to manage global transactions. To understand how such a manager can be implemented, let us define as abstract model of a transaction system. Each site of the system contains two subsystems:

Transaction manager, whose function is to manage the execution of those transactions (for *substransactions*) that access data stored in that site. Note that each such transaction may be either a local transaction (that is, a transaction that only executes at that site), or part of a global transaction (that is, a transaction that executes at several sites).

Transaction coordinator, whose function is to coordinate the execution of the various transactions (both local and global) initiated at that site.

The overall system architecture is depicted in Figure 1.3

The structure of a transaction manager is similar in many respects to the structure used in the centralized system case. Each transaction manager is responsible for:

Maintaining a log for recovery purposes.

Participating in an appropriate concurrency control scheme to coordinate the parallel execution of the transactions executing at that site.

As we shall see, both the recovery and concurrency schemes need to be modified in order to accommodate the distribution of transactions.

The transaction coordinator subsystem is not needed in the centralized environment, since a transaction accessed data only at one single site. A transaction coordinator, as its name implies, is responsible for coordinating the execution of all the transactions initiated at that site. For each such transaction, the coordinator is responsible for:

Starting the execution of the transaction.

Breaking the transaction into a number of *subtransactions*, and distributing these *subtransactions* to the appropriate sites for execution.

Coordinating the termination of the transaction which may result in the transaction being committed at all sites or aborted at all sites.

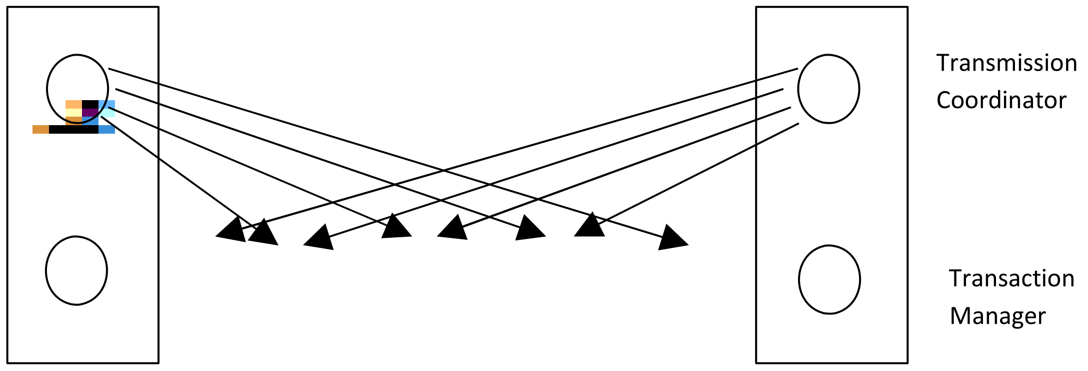


Figure 1.3 System Architecture.

1.8.2 Robustness

A distributed system may suffer from the same types of failure that centralized system does (for example, memory failure, disk crass). There are, however, additional failures that need to be dealt with in a distributed environment, including

The failure to a site
The failure of a link
Loss of messages
Network partitions

In order for the system to be robust, it must therefore detect any of these failures, reconfigure the system so that computation may continue, and recover when a processor or a link is repaired. It is generally not possible to differentiate between link failure, site failure, message loss, and network partition. We can usually detect that a failure has occurred, but we may not be able to identify what kind of failure it is. For example, suppose that site s1 is not able to communicate with S2. if could be that S2 has failed. However, another possibility is that the link between S1 and S2 has failed.

Suppose that site S1 has discovered that a failure has occurred. It must then initiate a procedure that will allow the system to reconfigure and continue with its normal mode of operation.

If replicated data is stored at the failed site, the catalog should be updated so that queries do not reference the copy at the failed site.

If transactions were active at the failed site at the time of the failure, these transactions should be aborted. It is desirable to abort such transactions promptly since they may hold locks on data at sites that are still active.

If the failed site is a central server for some subsystem, an "election" must be held on determine the new server. Examples of central servers include a name server, a concurrency coordinator, or a global deadlock detector.

Since it is, in general, not possible to distinguish between network link failures and site failures, any reconfiguration scheme must be designed to work correctly in case of a partitioning of the network. In particular, following situations must be avoided:

Two or more servers are elected in distinct partitions

More than one partition updates a replicated data item.

Reintegration of a repaired site or link into the system also requires some care. When a failed site recovers, it must initiate a procedure to update its system tables to reflect changes made while it was down. If the site had replicas of any data items, it must obtain the current values of these data items and ensure that it receives all future updates. This is more complicated than it seems to be a first glance, since there may be updates to the data items processed during the time the site is recovering.

An easy solution is temporarily to halt the entire system while the failed site rejoins the system. In most applications, this solution is unacceptably disruptive. A preferable solution is to represent the recovery tasks as a series of transactions. The concurrent control subsystem and transaction management subsystem may then be relied upon for proper reintegration of the site. If a failed link recovers, it is possible that two or more partitions are rejoined. Since a partitioning of the network limits the allowable operations by some or all sites, it is desirable to inform sites promptly of the recovery of the link. This can be done by a message broadcast to all sites. In networks in which broadcast is not feasible, alternative schemes may be used.

1.8.3 Commit Protocols

In order to assure atomicity, all the sites in which a transaction *T* executed must agree on the final outcome of the execution. *T* either commits at all sites, or aborts at all sites. In order to ensure this property, the transaction coordinator of *T* must execute a *commit protocol*.

There are a number of different commit protocols that can be used. We present the most widely used commit protocol, called *two–phase* commit. Let T be a transaction initiated at site. , and let the transaction coordinator at be . When *T* completes its execution, that is, when all the sites at which *T* has executed inform that *T* has completed, then starts the two–phase commit protocol.

> **Phase 1**. adds the record <*prepare*. *T*> to the long and forces it onto stable storage. Once this is done, it sends a *prepare–to commit* message to all sites at which *T* executed. Upon receiving such a message, the transaction manager at that site determines whether it is willing to commit its portion of *T*, or not. If the answer is no, it adds a record <*no T*> to the log and then it responds by sending an *abort T* message to . If the answer is yes, it ads a record <*ready T*> to the log and forces all the log records corresponding to *T* onto stable storage. Once this is aaccomplished, it replies with a *ready T* message to .
>
> **Phase 2**. When receives responses to the *prepare T* message from all the sites, or a prespecified interval of time has elapsed since the *prepare T* message was sent out, can determine whether the transaction *T* can be committed or aborted. Transaction *T* can be committed if received a *ready T* message from all the participating sites. Otherwise, transaction *T* must be aborted. Depending on the verdict, either a record <*commit T*> or a record <*abort T*> is added to the log and forced onto stable storage. At this point, the fate of the transaction has been sealed. Following this, the coordinator sends either a *commit T* or an *abort T* message to all participating sites. When a site receives that message, it records it in the log and sends a message *acknowledge T* to the coordinator. When the coordinator receives the *acknowledge T* message from all the sites, it adds the record <*complete T*> to the log.

Since any site at which *T* executed could fail at any time, the protocol is designed so that the transaction *T* will be aborted in all cases prior to the time that all sites involved in the execution of *T* have recorded changes made by *T* in stable storage. The *ready T* message is, in effect, a promise by a site to follow the coordinator's order to commit *T* or abort *T*. The only means by which a site can make such a promise is if the needed information is stored in stable storage. Otherwise, if the site crashes after sending *ready T*, it may be unable to make good on its promise.

Since unanimity is required to commit a transaction, the fate of *T* is sealed as soon as at least one site responds *abort T*. since the coordinator site is one of the sites at which *T* executed, the coordinator can decide unilaterally to abort *T*. The final verdict regarding *T* is determined at the time the coordinator writes that verdict (commit or abort) to the log and forces it to stable storage. The final *acknowledge T* messages are not a necessary part of two–phase commit.

We now examine in detail how two–phase commit responds to various types of failures:

Failure of participating site. When a participating site recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst of execution when the failure occurred. Let T be one such transaction. We consider each of the possible case below.

The log contains a <*commit T*> record. In this case, the site executes ***redo*** (*T*)

The log contains an <*abort T*> record. In this case, the site executes ***undo*** (T)

The log contains a <*ready T*> record. In this case, the site must consult to determine the fate of *T*. if is up, it notifies as to whether *T* committed or aborted. In the former case, it executes *redo* (*T*), while in the latter case, it executes *undo* (*T*). If is down, must try to find the fate of *T* from other sites. It does so by sending a *query–status T* message to all the sites in the system. Upon receiving such a message, a site must consult its log to determine whether *T* has executed there, and if so, whether *T* committed or aborted. It then notifies 0about this outcome. If no site has the appropriate information (that is, whether T committed or aborted), must periodically resend the *query–status* message until a site recovers that contains the needed information. Until the outcome of *T* is determined, must remain in the *ready T* state. This may require that locks held by *T* at not be released. Note that the site at which resides always has the needed information.

The log contains no control records concerning *T*. In this case, knows that has aborted *T* since has not responded to the *prepare T* message from and thus by our algorithm must abort T. Hence, must execute *undo* (*T*).

Failure of the Coordinator. When the coordinator fails, a decision must be made as to whether to commit or abort each transaction that was being coordinated by the failed coordinator.

Transaction *T* can be committed only if there exists an active site that contains a <*commit T*> record in its log.

Transaction *T* can be aborted only if there exists an active site that contains either a *no T*> or an <*abort T*> record in the log.

If neither of the above can be established, then the fate of transaction *T* cannot be determined and this must be postponed until the coordinator recovers.

Failure of a link. When a link fails, all the messages that are in the process of being routed through the link do not arrive at their destination intact. From the viewpoint of the sites connected throughout that link, it appears that the other sites have failed. Thus our previous schemes apply here as well.

Network partition. When a network partitions, two possibilities exist.

The coordinator and all of its participants remain in one partition. In this case, this failure has no effect on the commit protocol. The coordinator and its participants belong to several partitions. In this case, messages between the participant and the coordinator lost, reducing the case to a link failure discussed above.

1.9 CONCURRENCY CONTROL

In this unit, we show how some of these schemes can be modified so that they can be used in a distributed environment.

1.9.1 Locking Protocols

Below, we present several possible schemes, the first of which deals with the case where no data replication is allowed. The other schemes are applicable to the more general case where data can be replicated in several sites. We shall assume the existence of the *shared* and *exclusive* lock modes.

Nonreplicated Scheme

If no data is replicated in the system, then the locking schemes can be applied as follows. Each site maintains a local lock manager whose function is to administer the lock and unlock requests for those data items that are stored in that are stored in that site. When a transaction wishes to lock data item X at site it simply sends a message to the lock manager at site requesting a lock (in a particular lock mode). If data item X is locked in an incompatible mode, then the request is delayed until it can be granted. Once it has been determined that the lock request can be granted, the lock manager sends a message back to the initiator of the request indicating that the lock request has been granted.

The scheme has the advantage of simple implementation. It requires two message transfers for handling lock request, and one message transfer for handling unlock requests. However, deadlock handling is more complex. Since the lock and unlock request are no longer made at one single site, the various deadlock handling algorithms must be modified, as will be discussed in unit 1.10.

Single–Coordinator Approach

The system maintains one *single* lock manager that resides in one single chosen site, say . All lock and unlock requests are made at site . When a transaction needs to lock a data item, it sends a lock request to . The lock manager determines whether the lock can be granted immediately. If so, it sends a message to that effect to the site at which the lock request was initiated. Otherwise, the request is delayed until it can be granted, at which time; a message is sent to the site at which the lock request was initiated. The transaction can read the data item from any one of the sites at which a replica of that data item resides. In the case of a write, all the sites where a replica of that data item resides must be involved in the writing.

The scheme has the following advantages:

Simple implementation. This scheme requires two messages for handling lock request, and on message for handling unlock requests.

Simple deadlock handling. Since all lock and unlock requests are made at one site, the deadlock handling algorithms discussed in unit 1.11 can be applied directly to this environment.

The disadvantages of this scheme include the follows:

Bottleneck. The site becomes a bottleneck since all requests must be processed there.

Vulnerability. If the site fails, the concurrency controller is lost. Either processing must stop, or a recovery scheme such as those of unit 1.10 must be used.

A compromise between the advantages and disadvantages noted above can be achieved via a *multiple–coordinator approach* in which the lock manager function is distributed over several sites.

Each lock manager administers the *lock* and *unlock* requests for a subset of the data items. Each *lock manager* resides in a different site. This reduces the degree to which the coordinator is a bottleneck, but it complicates deadlock handling, since the lock and unlock requests are not made at one single site.

Majority Protocol

The majority protocol is a modification of the non-replicated data scheme that we presented earlier. The system maintains a lock manager at each site. Each manager manages the locks for all the data items or replicas of data items stored at that site. When a transaction wishes to lock a data item X, which is replicated in n different sites, it must send a lock request to more than half of the n sites in which X is stored. Each lock manager determines whether the lock can be granted immediately (as far as it is concerned). As before, the response to the request is delayed until it can be granted. The transaction does not operate on X until it has successfully obtained a lock on a majority of the replicas of X.

This scheme has the advantage of dealing with replicated data in a decentralized manner. This avoids the drawbacks of central control. However, it suffers from the following disadvantages.

Implementation. This scheme is more complicated to implement than the previous schemes. It requires $2(/2 + 1)$ messages for handling lock requests and $(/2 + 1)$ messages for handling unlock requests.

Deadlock handling. As will be discussed in unit 1.10. In addition, it is possible for a deadlock to occur even if only one data item is being locked. To illustrate this, consider a system with four sites and full replication. Suppose that transactions $_1$ and$_2$ wish to lock data item Q in exclusive mode. Transaction $_1$ may succeed in lock Q at sites$_1$ and $_3$ while transaction $_2$ may succeed in locking Q at sites $_2$ and$_4$. Each then must wait to acquire the third lock, and hence a deadlock has occurred.

Biased Protocol

This protocol is based on a model similar to that of the majority protocol. The difference is that requests for shared locks are given more favourable treatment than requests for exclusive locks. The system maintains a lock manager at each site. Each manager manages the locks for all the data items stored at that site. We differentiate between the way shared and exclusive locks are handled.

Shares locks. When a transaction needs to lock data item X, it simply requests a lock on X from the lock manager at one site containing a replica of X.

Exclusive locks. When a transaction needs to lock data item X, it requests a lock on X from the lock manager at all sites containing a replica of X.

As before, the response to the request is delayed until it can be granted. The scheme has the advantage of imposing less overhead on read operations than does the majority protocol. This is especially significant in common cases in which the frequency of read is much greater than the frequency of write. However, the additional overhead on writers is a disadvantage. Furthermore, the biased protocol shares the majority protocol's disadvantages of complexity in handling deadlock.

Primary Copy

In the case of data replication, we may choose one of the replicas as the primary copy. Thus, for each data item X, the primary copy of X must reside in precisely one site, which we call the *primary site* of X.

When a transaction needs to lock a data item X, it requests a lock at the primary site of X. As before, the response to the request is delayed until it can be granted.

Thus, we are able to handle concurrency control for replicated data in a manner similar to that in which we handled unreplicated data. This allows for a simple implementation. However, if the primary site of X fails, X is inaccessible even though other sites containing a replica may be accessible.

1.9.2 Time stamping

The principal idea behind is that each transaction is given a unique timestamp that is used in deciding the serialization order. Our first task, then, in generalizing the centralized scheme to a distributed scheme is to develop a scheme for generating unique timestamps. Once this has been accomplished, our previous protocols can be directly applied to the nonreplicated environment.

Generating Unique Timestamps

There are two primary methods for generating unique timestamps, one centralized and one distributed. In the centralized scheme, a single site is chosen for distributing the timestamps. The site can use a logical counter or its own local clock for this purpose.

In the distributed scheme, each site generates a unique local timestamp using either a logical counter or the local clock. The global unique timestamp is obtained by concatenating the unique local timestamp with the site identifier, which must be unique (Figure 1.4). The order of concatenation is important. We use the site identifier in the least significant position in order to ensure that the global timestamps generated in one site are not always greater than those generated in another site. Compare this technique for generating unique timestamps with the one we saw earlier for generating unique name.

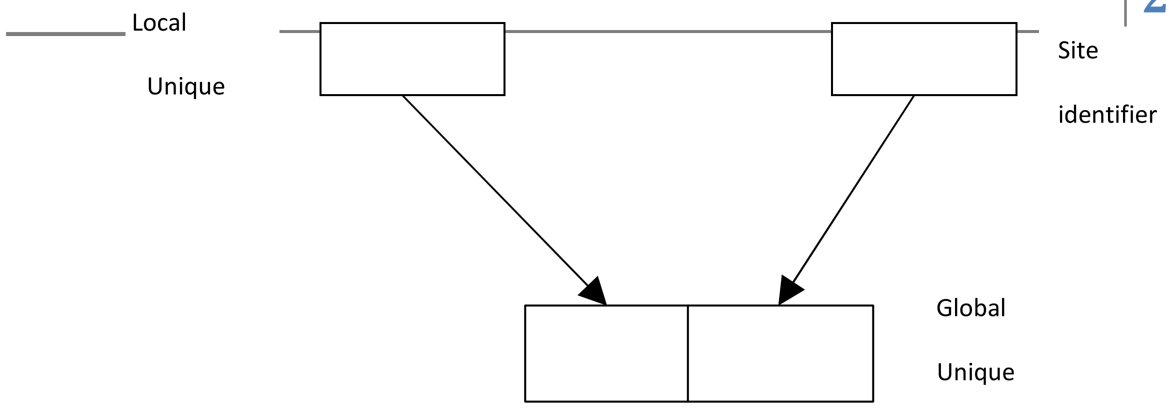


Figure 1.4 Generating unique timestamps

We may still have a problem if one site generates local timestamps at a faster rate than other sites. In such a case, the fast site's logical counter will be larger than that of other sites. Therefore, all timestamps generated by the fast site will be larger than those generated by other sites. What is needed is a mechanism to ensure that local timestamps are generated fairly across the system. To accomplish this, we define within each site a *logical clock* ( ) which generates the unique local timestamp. The logical clock can be implemented as a counter that is incremented after a new local timestamp is generated. To ensure that the various logical clocks are synchronized, we require that a site advance its logical clock whenever a transaction with timestamp < , > visits that site and x is greater than the current value of. In this case, site advances its logical clock to the value + 1.

If the system clock is used to generate timestamps, then timestamps are assigned fairly provided that no site has a system clock that runs fast or slow. Since clocks may not be perfectly accurate, it is necessary that a technique similar to that used for logical clocks be used to ensure that no clock gets very far ahead or behind another clock.

Concurrency Control Schemes

As in the centralized case, cascading rollbacks may result if no mechanism is used to prevent a transaction from reading a data item value which is not yet committed. To eliminate cascading rollbacks we can combine the basic timestamp scheme with the two–phase commit protocol of unit 1.7 to obtain a protocol that ensures Serializability with no cascading rollbacks. We leave the development of such an algorithm as an exercise for the reader.

The basic timestamp scheme described above suffers from the undesirable property that conflicts between transactions are resolved through rollbacks rather than waits. To alleviate this problem, we can buffer the various read and write operations (that is, *delay* them) until a time when we are assured that these operations can take place without causing aborts. A **read** (*x*) operation by must be delayed if there exists a transaction that will perform a **write** (*x*) operation but has not yet done so, and() < (). There are various methods for ensuring this property. One such method, called *conservative timestamp ordering scheme*, requires each site to maintain a **read** and **write** queue consisting of all the **read** and **write** requests, respectively, that are to be executed at the site and which must be delayed in order to preserve the above property. We shall not present the scheme here. Rather, we shall leave the development of the algorithm as an exercise for the reader.

1.10 DEADLOCK HANDLING

Deadlock prevention may result in some unnecessary waiting and rollback. Furthermore, some of the deadlock prevention techniques may require more sites to be involved in the execution of a transaction than would otherwise be the case.

If we allow deadlocks to occur and rely on deadlock detection, the main problem in a distributed system is deciding how to maintain the wait–for graph. We describe several common techniques to deal with this issue. These schemes require that each site keep a *local* wait–for graph. The nodes of the graph correspond to all the transactions (local as well as nonlocal) that are currently either holding or requesting any of the items local to that site. For example, in Figure 1.5 we have a system consisting of two sites, each maintaining its local wait–

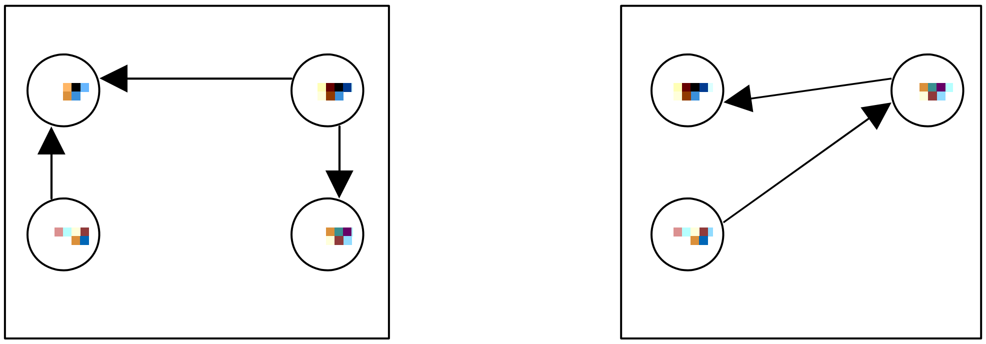


Figure 1.5 Local wait-for graphs

for graph. Note that transactions $_2$ and $_3$ appear in both graphs, indicating that the transactions have requested items at both sites.

These local wait-for graphs are constructed in the usual manner for local transactions and data items. When a transaction on site $_1$ needs a resources held by transaction in site $_2$, a request message is sent by to site $_2$. The edge is then inserted in the local wait-for graph of site $_1$.

Clearly, if any local wait-for graph has a cycle, deadlock has occurred. On the other hands, the fact that there are no cycles in any of the local wait-for graphs does not mean that there are no deadlocks. To illustrate this problem, consider the local wait-for graphs of Figure 1.5. Each wait-for graph is acyclic; nevertheless, a deadlock exists in the system. A deadlock exists because the union of the local wait-for graphs contains a cycle. This graph is shown in Figure 1.6

There are a number of different methods for organizing the wait-for graph in a distributed system. Several common schemes are described below.

1.10.1. Centralized Approach

In the centralized approach, a global wait-for graph (union of all the local graphs) is constructed and maintained in a *single* site, the deadlock detection coordinator. Since there is communication delay in the system, we must distinguish between two types of wait-for graphs. The *real* graph describes the real but unknown state of the system at any instance in time, as would be seen by an omniscient observer. The *constructed* graph is an approximation generated by the controller during the execution of its algorithm. Obviously, the constructed graph must be generated in such a way that whenever the detection algorithm is invoked, the reported results are correct in a sense that, if a deadlock exists it is reported promptly, and if it reports a deadlock, then the system is indeed in a deadlock state.

The global wait-for graph may be constructed:

Whenever a new edge is inserted or removed in one of the local wait-for graphs.

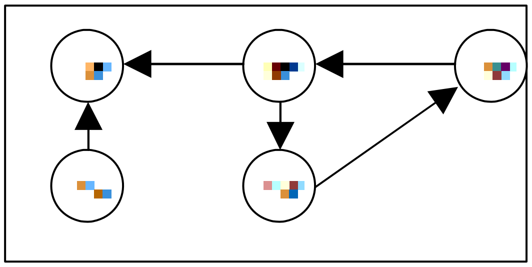


Figure 1.6 Global wait-for graph for Figure 1.5

Periodically when a number of changes have occurred in a local wait-for graph.

Whenever the coordinator needs to invoke the cycle detection algorithm.

When the deadlock detection algorithm is invoked, the coordinator searches its global graph. If a cycle is found, a victim is selected to be rolled back. The coordinator must notify all the sites that a particular transaction has been selected as victim. The sites, in turn, roll back the victim transaction.

We note that in this scheme unnecessary rollbacks may result, as a result of two situations:

False cycles may exist in the global wait-for graph. To illustrate this point, consider a snapshot of the system represented by the local wait-for graphs of Figure 1.7. Suppose that $_2$ releases the resource it is holding in site$_1$, resulting in the deletion of the edge $_1$ → $_2$ in $_1$ .Transaction$_2$ then request a resource held by $_3$ at site$_2$, resulting in the addition of the edge $_2$ → $_3$ in $_2$. if the *insert* message $_2$ → $_3$ from $_2$ arrives before the *remove* $_1$ → $_2$ message from $_1$, the coordinator may discover the false cycle $_1$ → $_2$ → $_3$ after the *insert* (but before the *remove*). Deadlock recovery may be initiated, although no deadlock has occurred.

Unnecessary rollbacks may also result when a *deadlock* has indeed occurred and a

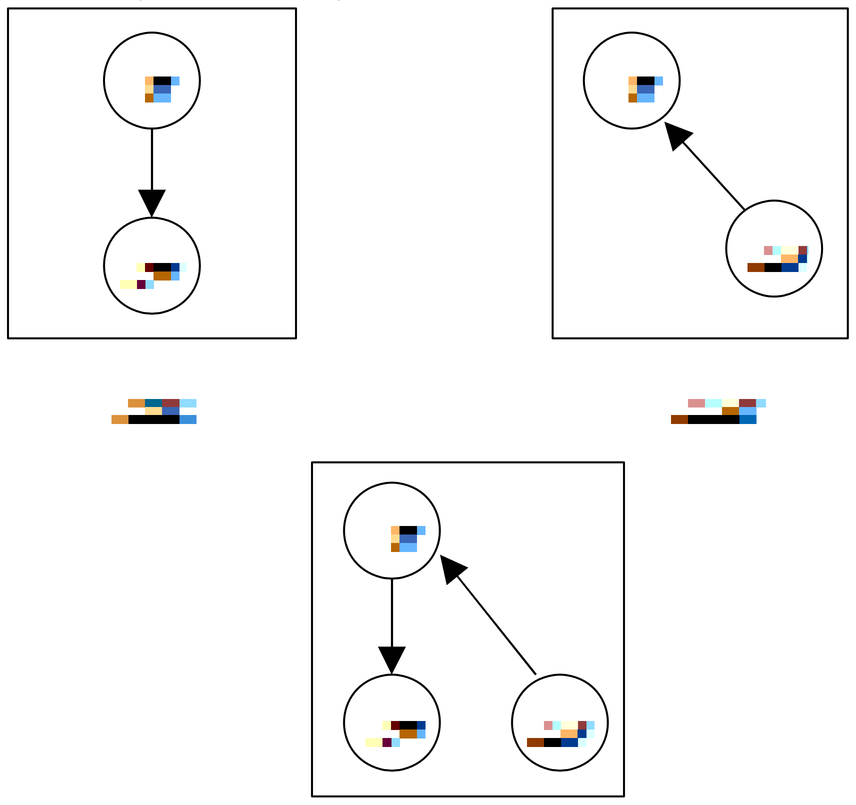


Figure 1.7 False cycles in the global wait-for graph.

victim has been picked, while at the same time of the transactions was aborted for reasons unrelated to the deadlock. For example, suppose that site $_1$ in Figure 1.5 decides to abort$_2$. At the same time, the coordinator has discovered a cycle and picked $_3$as a victim. Both $_2$ and $_3$ are now rolled back, although only $_2$ needed to be rolled back.

We note that the same problem exist in solutions employing the other option.

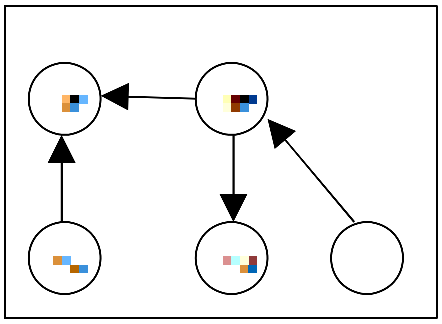
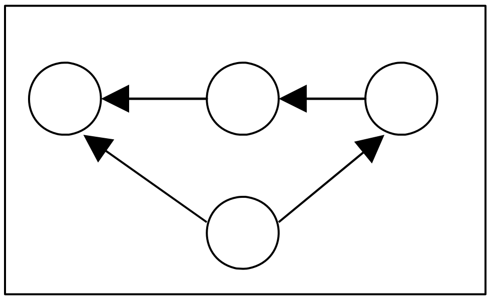
1.10.2 Fully Distributed Approach

In the *fully distributed* deadlock detection algorithm, all controllers share the responsibility for detecting deadlock equally. In this scheme, every site constructs a wait–for graph which represents a part of the total graph, depending on the dynamic behaviour of the system. The idea is that if a deadlock exists, a cycle will appear in (at least) one of the partial graphs. Below we present one such algorithm which involves construction of partial graphs in every site.

Each site maintains its own local wait-for graph. A local wait–for graph differs from the one described above in that we add one additional node to the graph. An arc $\rightarrow$ exists in the graph if is waiting for a data item in another site being held by any transaction. Similarly, an arc

→ exists in the graph if there exists a transaction at another site which is waiting to acquire a resource currently being held by in this local site.

To illustrate this, consider the two local wait-for graph of Figure 1.5. The addition of the node

S Site Site

Figure 1.8 Local wait-for graphs

in both graphs results is the local wait-for graphs shown in Figure 1.8.

If a local wait-for graph contains a cycle which does not involve node, then the system is in a deadlock state. If, however, there exists a cycle involving, then this implies that there is a possibility of a deadlock. In order to ascertain this, a distributed deadlock detection algorithm must be invoked. Suppose that site contains a cycle in its local wait for graph involving node . This cycle must be of the form:

$$\rightarrow \; \rightarrow_{2} \rightarrow \cdots \rightarrow \; \rightarrow$$

which indicates that transaction in is waiting to acquire a data item in some other site, say . Upon discovering this cycle, site sends to site a deadlock detection message containing information about that cycle.

When a site receives this deadlock detection message, it updates its local wait-for graph with the new information it has obtained. When this is done, it searches the newly constructed wait-for graph for a cycle not involving. If one exists, a deadlsock is found and an appropriate recovery scheme is invoked. If a cycle involving is discovered, then transmits a deadlock detection message to the appropriate site, say . Site, in return, repeats the above procedure. Thus, after a finite number of rounds, either a deaelock is discovered, or the deadlock detection computation halts.

To illustrate this, consider the local wait–for graphs of Figure 1.8. Suppose that site $_1$ discover the cycle:

$$\rightarrow {}_{2} \rightarrow {}_{3} \rightarrow$$

Since $_3$ is waiting to acquire a data item in site $_2$, a deadlock detection message describing that cycle is transmitted from site $_1$ to site $_2$. When site $_2$ receives this message, it updates its local wait-for graplh obtaining the wait-for graph of Figure 1.9. This graph contains the cycle:

$$_2 \rightarrow {}_3 \rightarrow {}_4 \rightarrow {}_2$$

That does not include node . Therefore, the system is sin a deadlock state and an approspriate recovery scheme must be invoked.

Note that the outcome would be the same if site $_2$ discovered the cycle first in its local wait-for graph and it sent the deadlock detection message to site $_1$. In the worst case, both sites discover the cycle at about the same time, and two deadlock detection messages will be sent, one by $_1$ to $_2$ and another by $_2$ to $_1$. This result is unnecessary message transfter and overhead in updating the two local wait-for graphs and searching for cycles in both graphs.

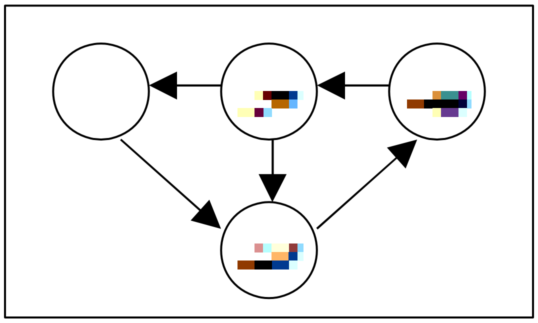
To reduce message traffic, we assign to each transaction a unique identifier, which we denote by $(\ )$. When site discovers that its local wait-for graph contains a cycle involving node of the form

$$\rightarrow {}_1 \rightarrow {}_2 \rightarrow \cdots \rightarrow \ \rightarrow$$

It will sent a deadlock detection message to another site only if

$$(\ ) < (\ )$$

Otherwise, site continues with its normal execution leaving the burden of initiating the deadlock detection algorithm to some other site.



Site

Figure 1.9 Local wait-for graph

To illustrate this, consider again the wait-for graph maintained at sites $_1$ and $_2$ of Figure 1.8. Suppose that

$$(_1) < (_2) < (_3) < (_4)$$

Let both sites discover these local cycles at about the same time. The cycle is site $_1$ is of the form

$$\rightarrow {}_2 \rightarrow {}_3 \rightarrow$$

Since $(_3) > (_2)$, site $_1$ does not send a deadlock detection message to site $_2$.

The cycle in site $_2$ is of the form

$$\rightarrow {}_3 \rightarrow {}_4 \rightarrow {}_2 \rightarrow$$

Since $(_2) < (_3)$, site $_2$ does send a deadlock detection message to site $_1$, which upon receiving the message dupdates its local wait-for graph. It then searches for a cycle in the graph and discovers that the system is in a deadlock state.

1.11 COORDINATOR SELECTION

Some of the algorithms we have presented above require the use of a coordinator. If the coordinator fails due to the failure of the site at which it resides, the system can continue execution only by restarting a new coordinator on some other site: this can be accomplished by maintaining a backup to the coordinator that is ready to assume responsibility if the coordinator fails. Another approach is to choose the new coordinator after the coordinator has failed. The algorithms that determine where a new copy of the coordinator should be restarted are called election algorithms.

1.11.1 . Backup Coordinators

A *backup coordinator* is a site which, in addition other task, maintains enough information locally to allow it to assume the role of coordinator with minimal disruption of the distributed system. All messages directed to the coordinator are received by both the coordinator and its backup. The backup coordinator executes the same algorithms and maintains the same internal state information (such as, for a concurrency coordinator, the lock table) bas the actual coordinator and its backup is that the backup does not take any action that affects other sites. Such actions are left to the actual coordinator.

In the event that the backup coordinator detects the failure of the actual coordinator, it assumes the role of coordinator. Since the backup has all of the information available to it that the failed coordinator had, processing can proceed without interruption.

The prime advantage to the backup approach is the ability to continuer processing immediately. If a backup were not ready to assume the coordinator's responsibility, it would be necessary for a newly appointed coordinator to seek information from all sites in the system so that it could execute the coordination tasks. Frequently, the only source of some of the requisite information is the failed coordinator. In this case, it may be necessary to abort several (or all) active transaction s and restart them under the aegis of the new coordinator. Thus, the backup coordinator approach avoids a substantial amount of delay while the distributed system recovers from a coordinator failure.

The disadvantage to the backup coordinator approach is the overhead of duplicate execution of the coordinator's tasks. Furthermore, a coordinator and its backup need to communicate regularly to ensure that their activities are synchronized.

Thus, the backup coordinator approach involves overhead during normal processing in order to allow fast recovery from a coordinator failure. In the next unit, we consider a lower-overhead recovery scheme that requires somewhat more effort in order to recover from a failure.

1.11.2 Election Algorithms

Election algorithms require that a unique identification number be associated with each active site in the system. For easy of notation, we shall assume that the identification number of site is *.* Also, to simplify our discussion, we assume that the coordinator resides always at the site with the largest identification number. The goal of an election algorithm is to choose a site for the new coordinator. Hence, when a coordinator fails, the algorithm must elect that active site with the largest identification number. This number must be sent to each active site in the system. Additionally, the algorithm must provide a mechanism by which a site recovering from a crash may identify the current coordinator.

There are a number of different election algorithms. These usually differ in terms of the network configuration. In this unit, we present one of these algorithms, the *bully* algorithm,

Suppose that site sends a request that is into answered by the coordinator within a prespecified time interval *T*. In this situation, it is assumed that the coordinator has failed, and tries to elect itself as the site for the new coordinator.

Site sends an elections message to every site with a higher identification number. Site then waits for a time interval *T* for an answer from any one of these sites.

If no response is received within time *T*, it is assumed that sites with numbers greater than

have failed, and elects itself as the site for the new coordinator and sends a message to inform all active sites with identification numbers less than *i* that is the site at which the new coordinator resides.

However, if an answer is received, begins a time interval *T*, waiting to receive a message informing it that a site with a higher identification number has been elected. (Some other site is electing itself coordinator, and should report the results within time *T*). If no message is sent within , then the site with a higher number is assumed to have failed, and site restarts the algorithm.

After a failed site recovers, it immediately begins execution of the same algorithm. If there are no active sites with higher numbers, the recovered site forces all sites with lower numbers to let it become the coordinator site, even if there is a currently active coordinator with a lower number. It is for this reason that the algorithm is termed the *bully* algorithm.

1.12. CONCLUSION

There are several reasons for building distributed database systems, including sharing of data, reliability and availability, and speedup of query processing. However, along with these advantages come several disadvantages, including software development cost, greater potential for bugs, and increased processing overhead. The primary disadvantages of distributed database systems is the added complexity required to ensure proper coordination among the sites.

1.13. SUMMARY

A distributed database system consists of a collection of sites, each of which maintains a local database system. Each site is able to process *local transactions*, those transactions that access data only in that single site. In addition, a site may participate in the execution of *global transactions*, those transactions that access data in several sites. The execution of global transactions requires communication among the sites.

There are several issues involved in storing a relation in the distributed database, including replication and fragmentation. It is essential that the system minimize the degree to which a user needs to be aware of how a relation is stored.

A distributed system may suffer from the same types of failure that a centralized system does. There are, however, additional failures that need to be dealt with in a distributed environment, including the failure of a site, the failure of a link, loss of messages, and network partition. Each of these needs to be considered when designing a distributed recovery scheme. In order for the system to be robust, it must therefore *detect* any of these failures, *reconfigure* the system so that computation may continue, and recover when a processor or a link is repaired.

In order toe assure atomicity, all the sites in which a transaction *T* executed must agree on the final outcome of the execution. *T* either commits at all sites or aborts at all sites. In order to ensure this property, the transaction coordinator of *T* must execute a *commit protocol*. There are a number of different commit protocols that can be used; the one most widely used is *the two–phase commit* protocol.

The various concurrency control schemes which can be used in a centralized system can be modified so that they can be used in a distributed environment. In the case of locking protocols, the only change that needs to be incorporated is in the way we implement the lock manager.

There are a variety of different ways of doing so. One or more central coordinators may be used. If instead a distributed approach is taken, replicated data must be treated specially. There are several protocols for doing this including the majority, biased, and primary – copy protocols. In the case of timestamping and validation schemes, the only needed change is in developing a mechanism for generating unique *global* timestamps. This can be done by either concatenating a local timestamp with the site identification, or by advancing local clocks whenever a message arrives with a larger timestamp.

The primary method for dealing with deadlocks in a distributed environment is deadlock detection. The main problem is in deciding how to maintain the wait-for graph. There are a number of different methods for organizing the wait-for graph including a centralized approach, a hierarchical approach, and a fully distributed approach.

Some of the distributed algorithms require the use of a coordinator. If the coordinator fails due to the failure of the site at which it resides, the system can continue execution only by restarting a new copy of the coordinator on some other site. This can be accomplished by maintaining a backup to the coordinator that is ready to assume responsibility if the coordinator fails. Another approach is to choose the new coordinator after the coordinator has failed. The algorithms that determine where a new copy of the coordinator should be restarted are called *election* algorithms.

1.14 TMA

1. Discuss the relative advantages of centralized and distributed database.

2. How might a distributed database designed for a local-area network differ from one designed for a long-haul network?

3. Consider the relations

employee (*name*, *address*, *salary*, *plant-number*)

machine (*machine-number*, *type*, *plant-number*)

Assume the employee relation is fragmented horizontally, by *plant-number* and each fragment is stored locally at its corresponding plant site. Assume the *machine* relation is stored in its entirety at the Armonk site. Describe a good strategy for processing each of the following queries

a. Find all employees at the plant containing machine number 1130

b. Find all employee at plants containing machines whose type is “milling machine”

b. find all machines at the Almaden plant

c. Find $\bowtie$.

4. Does $\ltimes$ necessarily equals $\ltimes$? Under what conditions does $\ltimes = \ltimes$ hold?

1.15. FURTHER READINGS

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[1986]. "Database System Concepts", McGraw – Hill, 1986.

Zobel, J., Moffat, A., and Sacks–Davis, R., [1992]. "An Efficient Indexing Technique for Full – Text Database Systems," in VLDB [1992.

www.w3schools.com

MODULE 3: DISTRIBUTED DATABASES

UNIT 2: OBJECT ORIENTED DATABASE

2.0. INTRODUCTION

An object database (also object-oriented database) is a database model in which information is represented in the form of objects as used in object-oriented programming. Object databases are a niche field within the broader DBMS market dominated by Relational database management systems (RDBMS). Object databases have been considered since the early 1980s and 1990s but they have made little impact on mainstream commercial data processing, though there is some usage in specialized areas.

An object-oriented database management system (OODBMS) sometimes shortened to ODBMS for object database management system), is a database management system (DBMS) that supports the modeling and creation of data as objects. This includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects. There is currently no widely agreed-upon standard for what constitutes an OODBMS, and OODBMS products are considered to be still in their infancy. In the meantime, the object-relational database management system (ORDBMS), the idea that object-oriented database concepts can be superimposed on relational databases, is more commonly encountered in available products. An object-oriented database interface standard is being developed by an industry group, the Object Data Management Group (ODMG). The Object Management Group (OMG) has already standardized an object-oriented data brokering interface between systems in a network.

2.1. OBJECTIVES

- To develop a framework for an OODM.
- To understand the basics of persistent programming languages.
- To understand the main strategies for developing an OODBMS.
- To design an object-oriented database.

2.2. FEATURES OF AN OBJECT-ORIENTATED DBMS.

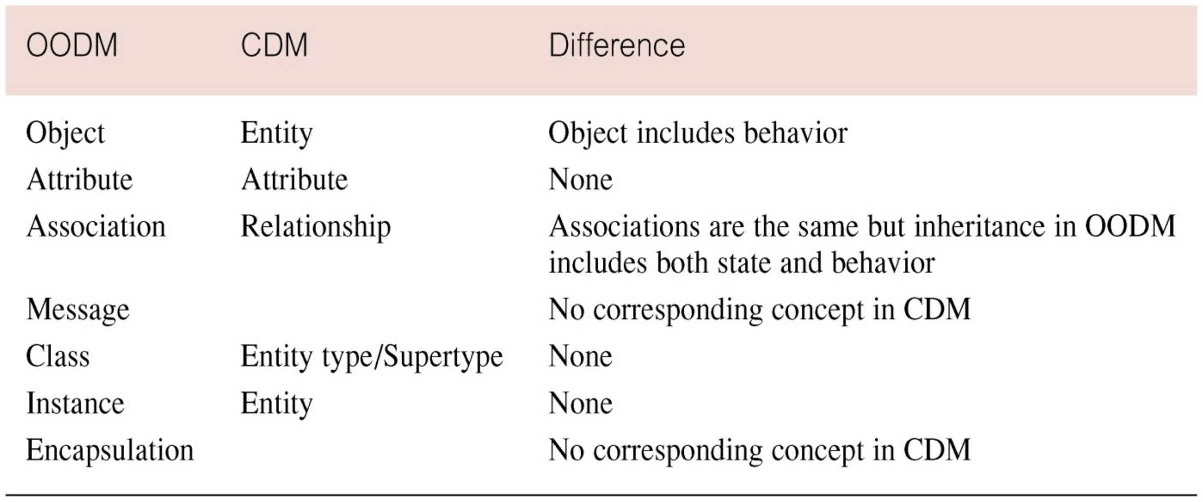
An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an ad hoc query facility. The second one translates into eight features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness.

Mostly, object databases offer some kind of query language, allowing objects to be found by a more declarative programming approach. It is in the area of object query languages, and the integration of the query and navigational interfaces, that the biggest differences between products are found. An attempt at standardization was made by the ODMG with the Object Query Language, OQL.

Access to data can be faster because joins are often not needed (as in a tabular implementation of a relational database). This is because an object can be retrieved directly without a search, by following pointers. (It could, however, be argued that "joining" is a higher-level abstraction of pointer following.) Another area of variation between products is in the way that the schema of a database is defined. A general characteristic, however, is that the programming language and the database schema use the same type definitions. Multimedia applications are facilitated because the class methods associated with the data are responsible for its correct interpretation.

Many object databases, for example VOSS, offer support for versioning. An object can be viewed as the set of all its versions. Also, object versions can be treated as objects in their own right. Some object databases also provide systematic support for triggers and constraints which are the basis of active databases. The efficiency of such a database is also greatly improved in areas which demand massive amounts of data about one item. For example, a banking institution could get the user's account information and provide them efficiently with extensive information such as transactions, account information entries etc. The Big O Notation for such a database paradigm drops from O(n) to O(1), greatly increasing efficiency in these specific cases.

2.3. OBJECT ORIENTED DATABASE DESIGN



OODM	CDM	Difference
Object	Entity	Object includes behavior
Attribute	Attribute	None
Association	Relationship	Associations are the same but inheritance in OODM includes both state and behavior
Message		No corresponding concept in CDM
Class	Entity type/Supertype	None
Instance	Entity	None
Encapsulation		No corresponding concept in CDM

OODBMS; Advantages & Disadvantages

Advantages

1. **Composite Objects and Relationships:** Objects in an OODBMS can store an arbitrary number of atomic types as well as other objects. It is thus possible to have a large class which holds many medium sized classes which themselves hold many smaller classes, ad infinitum. In a relational database this has to be done either by having one huge table with lots of null fields or via a number of smaller, normalized tables which are linked via foreign keys. Having lots of smaller tables is still a problem since a join has to be performed every time one wants to query data based on the "Has-a" relationship between the entities. Also an object is a better model of the real world entity than the relational tuples with regards to complex objects. The fact that an OODBMS is better suited to handling complex, interrelated data than an RDBMS means that an OODBMS can outperform an RDBMS by ten to a thousand times depending on the complexity of the data being handled.

2. **Class Hierarchy:** Data in the real world usually has hierarchical characteristics. The ever popular Employee example used in most RDBMS texts is easier to describe in an OODBMS than in an RDBMS. An Employee can be a Manager or not, this is usually done in an RDBMS by having a type identifier field or creating another table which uses foreign keys to indicate the relationship between Managers and Employees. In an OODBMS, the Employee class is simply a parent class of the Manager class.

3. **Circumventing the Need for a Query Language:** A query language is not necessary for accessing data from an OODBMS unlike an RDBMS since interaction with the database is done by transparently accessing objects. It is still possible to use queries in an OODBMS however.

4. **No Impedance Mismatch:** In a typical application that uses an object oriented programming language and an RDBMS, a significant amount of time is usually spent mapping tables to objects and back. There are also various problems that can occur when the atomic types in the database do not map cleanly to the atomic types in the programming language and vice versa. This "impedance mismatch" is completely avoided when using an OODBMS.

5. **No Primary Keys:** The user of an RDBMS has to worry about uniquely identifying tuples by their values and making sure that no two tuples have the same primary key values to avoid error conditions. In an OODBMS, the unique identification of objects is done behind the scenes via OIDs and is completely invisible to the user. Thus there is no limitation on the values that can be stored in an object.

6. **One Data Model:** A data model typically should model entities and their relationships, constraints and operations that change the states of the data in the system. With an RDBMS it is not possible to model the dynamic operations or rules that change the state of the data in the system because this is beyond the scope of the database. Thus applications that use RDBMS systems usually have an Entity Relationship diagram to model the static parts of the system and a separate model for the operations and behaviors of entities in the application. With an OODBMS there is no disconnecting between the database model and the application model because the entities are just other objects in the system. An entire application can thus be comprehensively modeled in one UML diagram.

Disadvantages

1. **Schema Changes:** In an RDBMS modifying the database schema either by creating, updating or deleting tables is typically independent of the actual application. In an OODBMS based application modifying the schema by creating, updating or modifying a persistent class typically means that changes have to be made to the other classes in the application that interact with instances of that class. This typically means that all schema changes in an OODBMS will involve a system wide recompile. Also updating all the instance objects within the database can take an extended period of time depending on the size of the database.

2. **Language Dependence:** An OODBMS is typically tied to a specific language via a specific API. This means that data in an OODBMS is typically only accessible from a specific language using a specific API, which is typically not the case with an RDBMS.

3. **Lack of Ad-Hoc Queries:** In an RDBMS, the relational nature of the data allows one to construct ad-hoc queries where new tables are created from joining existing tables then querying them. Since it is currently not possible to duplicate the semantics of joining two tables by "joining" two classes then there is a loss of flexibility with an OODBMS. Thus the queries that can be performed on the data in an OODBMS is highly dependent on the design of the system.

2.4. HOW OO CONCEPTS HAVE INFLUENCED THE RELATIONAL MODEL.

There are concepts in the relational database model that are similar to those in the object database model. A relation or table in a relational database can be considered to be analogous to a class in an object database. A tuple is similar to an instance of a class but is

different in that it has attributes but no behaviors. A column in a tuple is similar to a class

attribute except that a column can hold only primitive data types while a class attribute can hold data of any type. Finally classes have methods which are computationally complete (meaning that general purpose control and computational structures are provided while relational databases typically do not have computationally complete programming capabilities although some stored procedure languages come close.

Relationships

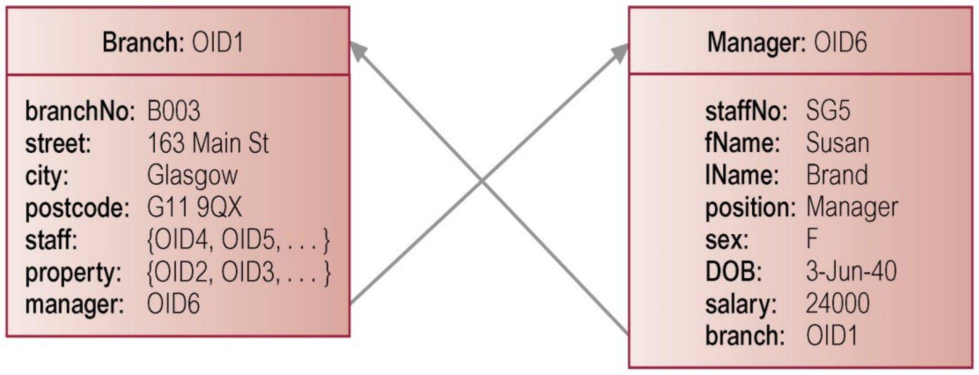
In OODMS, relationships represented using reference attributes which are typically

implemented using Object Identifiers (OIDs). Sample tables are given below. Consider how to represent the following binary relationships according to their cardinality:

1:1- one-to-one

1:* - one-to-many

:. – many-to-many



- One-to-one relationship: First, we add a reference attribute to A and to maintain referential integrity, we reference attribute to B also as shown below.

Figure 2.1: A one-to-one relationship

- One-to-many: First add reference attribute to B and attribute containing set of references to A.

Figure 2.2: A one-to-many relationship

- Many-to-many relationship

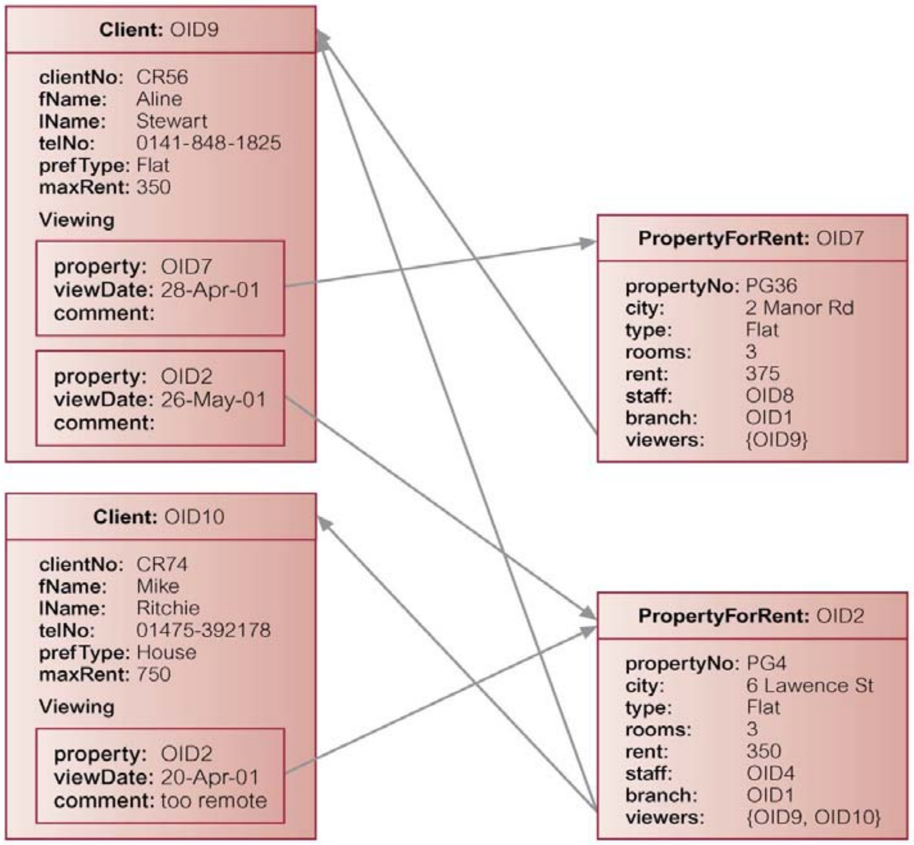


Figure 2.3: A many-to-many relationship

Referential Integrity

This is a property of data which, when satisfied, requires every value of one attribute (column) of a relation (table) to exist as a value of another attribute in a different (or the same) relation (table).

Less formally, and in relational databases: For referential integrity to hold, any field in a table that is declared a foreign key can contain only values from a parent table's primary key or a candidate key. For instance, deleting a record that contains a value referred to by a foreign key in another table would break referential integrity. Some relational database management systems (RDBMS) can enforce referential integrity, normally either by deleting the foreign key rows as well to maintain integrity, or by returning an error and not performing the delete.

If a record in the Managers table is deleted, all corresponding records in the Employees table must be deleted using a cascading delete.

Several techniques can be used to handle referential integrity in an object oriented database model.

Do not allow user to explicitly delete objects. The system is responsible for "garbage collection".

Allow user to delete objects when they are no longer required. The system may detect invalid references automatically and set reference to NULL or disallow the deletion.

2.5. CONCLUSION

There is currently no widely agreed-upon standard for what constitutes an OODBMS, and OODBMS products are considered to be still in their infancy. In the meantime, the object-relational database management system (ORDBMS), the idea that object-oriented database concepts can be superimposed on relational databases, is more commonly encountered in available products.

2.6. SUMMARY

Object databases are a niche field within the broader DBMS market dominated by Relational database management systems (RDBMS). An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages.

2.7 TUTOR-MARKEDASSIGNMENT(TMA).

1. How would you define object orientation? What are some of its benefits?
2. What is the difference between an object and a class in the object-oriented data model (OODM)?
3. What are the advantages and disadvantages of OODBMS?

2.8. FURTHER READINGS

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[1986]. “Database System Concepts”, McGraw – Hill, 1986.

www.w3schools.com

MODULE 3: DISTRIBUTED DATABASES

UNIT 3: DATABASE AND XML

3.0. INTRODUCTION

There are four central problems in data management capture, storage retrieval and exchange. The focus for most of this book has been on storage (i.e. data modeling) and retrieval (i.e. SQL). Now it is time to consider capture and exchange. Capture has always been an important issue, and the guiding principle is to capture once in the cheapest possible manner. Likewise, data exchange has long been an issue, but the internet has elevated the import this issue. Electronic data interchange (EDI), the traditional standard of data exchange for large organizations, is giving way to XML, which is likely to become the data exchange standard for all organizations, irrespective of size.

3.1. OBJECTIVES

To understand XML
To understand the relationships among DTDs, Schemas, and XML
To understand the principles of Web site management with XML

3.2. DEFINE THE PURPOSE OF XML

What You Should Already Know

Before you continue you should have a basic understanding of the following:

HTML
JavaScript
HTML was designed to display data. XML was designed to transport and store data.

What is XML?

XML stands for eXtensible Markup Language
XML is a markup language much like HTML
XML was designed to carry data, not to display data
XML tags are not predefined. You must define your own tags
XML is designed to be self-descriptive

The Difference Between XML and HTML

XML is not a replacement for HTML. XML and HTML were designed with different goals:

XML was designed to transport and store data, with focus on what data is.
HTML was designed to display data, with focus on how data looks.
HTML is about displaying information, while XML is about carrying information.

XML Does not DO Anything

Maybe it is a little hard to understand, but XML does not DO anything. XML was created to structure, store, and transport information.

The following example is a note to Tove from Jani, stored as XML:

```
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The note above is quite self descriptive. It has sender and receiver information, it also has a heading and a message body. But still, this XML document does not DO anything. It is just pure information wrapped in tags. Someone must write a piece of software to send, receive or display it.

XML is Just Plain Text

XML is nothing special. It is just plain text. Software that can handle plain text can also handle XML. However, XML-aware applications can handle the XML tags specially. The functional meaning of the tags depends on the nature of the application.

With XML You Invent Your Own Tags

The tags in the example above (like <to> and <from>) are not defined in any XML standard. These tags are "invented" by the author of the XML document. That is because the XML language has no predefined tags. The tags used in HTML (and the structure of HTML) are predefined. HTML documents can only use tags defined in the HTML standard (like <p>, <h1>, etc.). XML allows the author to define his own tags and his own document structure. XML however is not a replacement for HTML. It is just a complement to HTML.

In most web applications, XML is used to transport data, while HTML is used to format and display the data. The best way to describe XML is that **XML is a software- and hardware-independent tool for carrying information.**

XML is Everywhere

XML is now as important for the Web as HTML was to the foundation of the Web. XML is everywhere. It is the most common tool for data transmissions between all sorts of applications, and is becoming more and more popular in the area of storing and describing information. XML is used in many aspects of web development, often to simplify data storage and sharing.

XML Separates Data from HTML

If you need to display dynamic data in your HTML document, it will take a lot of work to edit the HTML each time the data changes. With XML, data can be stored in separate XML files. This way you can concentrate on using HTML for layout and display, and be sure that changes in the underlying data will not require any changes to the HTML. With a few lines of JavaScript, you can read an external XML file and update the data content of your HTML.

XML Simplifies Data Sharing

In the real world, computer systems and databases contain data in incompatible formats.XML data is stored in plain text format. This provides a software- and hardware-independent way of storing data. This makes it much easier to create data that different applications can share.

XML Simplifies Data Transport

With XML, data can easily be exchanged between incompatible systems. One of the most time-consuming challenges for developers is to exchange data between incompatible systems over the Internet. Exchanging data as XML greatly reduces this complexity, since the data can be read by different incompatible applications.

XML Simplifies Platform Changes

Upgrading to new systems (hardware or software platforms), is always very time consuming. Large amounts of data must be converted and incompatible data is often lost. XML data is stored in text format. This makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

XML Makes Your Data More Available

Since XML is independent of hardware, software and application, XML can make your data more available and useful. Different applications can access your data, not only in HTML pages, but also from XML data sources. With XML, your data can be available to all kinds of "reading machines" (Handheld computers, voice machines, news feeds, etc), and make it more available for blind people, or people with other disabilities.

XML is Used to Create New Internet Languages

A lot of new Internet languages are created with XML. Here are some examples:

XHTML the latest version of HTML
WSDL for describing available web services
WAP and WML as markup languages for handheld devices
RSS languages for news feeds
RDF and OWL for describing resources and ontology
SMIL for describing multimedia for the web

The future might give us word processors, spreadsheet applications and databases that can read each other's data in a pure text format, without any conversion utilities in between. We can only pray that all the software vendors will agree.

3.3. XML TREE

XML documents form a tree structure that starts at "the root" and branches to "the leaves".

An Example XML Document

XML documents use a self-describing and simple syntax:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

The first line is the XML declaration. It defines the XML version (1.0) and the encoding used (ISO-8859-1 = Latin-1/West European character set).

The next line describes the **root element** of the document (like saying: "this document is a note"):

```
<note>
```

The next 4 lines describe 4 **child elements** of the root (to, from, heading, and body):

```
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
```

And finally the last line defines the end of the root element:

```
</note>
```

You can assume, from this example, that the XML document contains a note to *Tove* from *Jani*.

XML Documents Form a Tree Structure

XML documents must contain a **root element**. This element is "the parent" of all other elements.

The elements in an XML document form a document tree. The tree starts at the root and branches to the lowest level of the tree. All elements can have sub elements (child elements):

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

The terms parent, child, and sibling are used to describe the relationships between elements. Parent elements have children. Children on the same level are called siblings (brothers or sisters).

All elements can have text content and attributes (just like in HTML).

Example:

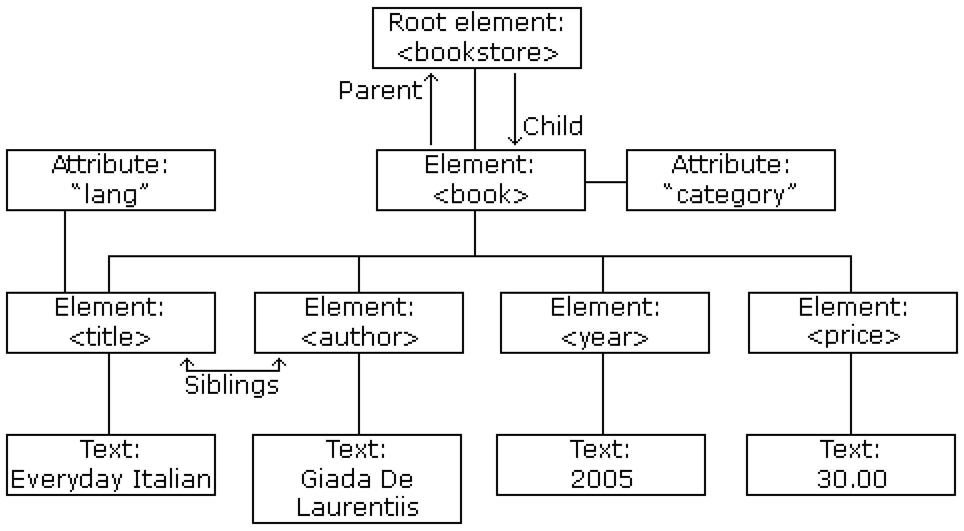


Figure 3.1: A Book representation

The image above represents one book in the XML below:

```
<bookstore>
 <book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
 </book>
 <book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
 </book>
 <book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
 </book>
</bookstore>
```

The root element in the example is <bookstore>. All <book> elements in the document are contained within <bookstore>. The <book> element has 4 children: <title>,< author>, <year>, <price>.

3.4. XML SYNTAX RULES

The syntax rules of XML are very simple and logical. The rules are easy to learn, and easy to use.

All XML Elements Must Have a Closing Tag

In HTML, you will often see elements that don't have a closing tag:

```
<p>This is a paragraph
<p>This is another paragraph
```

In XML, it is illegal to omit the closing tag. All elements **must** have a closing tag:

```
<p>This is a paragraph</p>
<p>This is another paragraph</p>
```

Note: You might have noticed from the previous example that the XML declaration did not have a closing tag. This is not an error. The declaration is not a part of the XML document itself, and it has no closing tag.

XML Tags are Case Sensitive

XML elements are defined using XML tags. XML tags are case sensitive. With XML, the tag <Letter> is different from the tag <letter>. Opening and closing tags must be written with the same case:

```
<Message>This is incorrect</message>
<message>This is correct</message>
```

Note: "Opening and closing tags" are often referred to as "Start and end tags". Use whatever you prefer. It is exactly the same thing.

XML Elements Must be Properly Nested

In HTML, you might see improperly nested elements:

```
<b><i>This text is bold and italic</b></i>
```

In XML, all elements **must** be properly nested within each other:

```
<b><i>This text is bold and italic</i></b>
```

In the example above, "Properly nested" simply means that since the <i> element is opened inside the <b> element, it must be closed inside the <b> element.

XML Documents Must Have a Root Element

XML documents must contain one element that is the **parent** of all other elements. This element is called the **root** element.

```
<root>
 <child>
```

```
  <subchild>.....</subchild>
 </child>
</root>
```

XML Attribute Values Must be Quoted

XML elements can have attributes in name/value pairs just like in HTML. In XML the attribute value must always be quoted. Study the two XML documents below. The first one is incorrect, the second is correct:

```
<note date=12/11/2007>
 <to>Tove</to>
 <from>Jani</from>
</note>
```

```
<note date="12/11/2007">
 <to>Tove</to>
 <from>Jani</from>
</note>
```

The error in the first document is that the date attribute in the note element is not quoted.

Entity References

Some characters have a special meaning in XML. If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element. This will generate an XML error:

```
<message>if salary < 1000 then</message>
```

To avoid this error, replace the "<" character with an **entity reference**:

```
<message>if salary &lt; 1000 then</message>
```

There are 5 predefined entity references in XML:

&lt;	<	less than
&gt;	>	greater than
&amp;	&	ampersand
&apos;	'	apostrophe
&quot;	"	quotation mark

Note: Only the characters "<" and "&" are strictly illegal in XML. The greater than character is legal, but it is a good habit to replace it.

Comments in XML

The syntax for writing comments in XML is similar to that of HTML.

```
<!-- This is a comment -->
```

White-space is Preserved in XML

HTML truncates multiple white-space characters to one single white-space:

HTML:	Hello my name is Tove
Output:	Hello my name is Tove.

With XML, the white-space in a document is not truncated.

XML Stores New Line as LF

In Windows applications, a new line is normally stored as a pair of characters: carriage return (CR) and line feed (LF). The character pair bears some resemblance to the typewriter actions of setting a new line. In Unix applications, a new line is normally stored as a LF character. Macintosh applications use only a CR character to store a new line.

3.5. XML ELEMENTS

An XML document contains XML Elements.

What is an XML Element?

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

An element can contain other elements, simple text or a mixture of both. Elements can also have attributes.

```
<bookstore>
  <book category="CHILDREN">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

In the example above, <bookstore> and <book> have **element contents**, because they contain other elements. <author> has **text content** because it contains text.

In the example above only <book> has an **attribute** (category="CHILDREN").

3.5.1. XML Naming Rules

XML elements must follow these naming rules:

Names can contain letters, numbers, and other characters
Names cannot start with a number or punctuation character
Names cannot start with the letters xml (or XML, or Xml, etc)
Names cannot contain spaces

Any name can be used, no words are reserved.

Best Naming Practices

Make names descriptive. Names with an underscore separator are nice: <first_name>, <last_name>. Names should be short and simple, like this: <book_title> not like this: <the_title_of_the_book>.

Avoid "-" characters. If you name something "first-name," some software may think you want to subtract name from first.

Avoid "." characters. If you name something "first.name," some software may think that "name" is a property of the object "first."
Avoid ":" characters. Colons are reserved to be used for something called namespaces (more later).
XML documents often have a corresponding database. A good practice is to use the naming rules of your database for the elements in the XML documents.
Non-English letters like éòá are perfectly legal in XML, but watch out for problems if your software vendor doesn't support them.

XML Elements are Extensible

XML elements can be extended to carry more information. Look at the following XML example:

```
<note>
<to>Tove</to>
<from>Jani</from>
<body>Don't forget me this weekend!</body>
</note>
```

Let's imagine that we created an application that extracted the <to>, <from>, and <body> elements from the XML document to produce this output:

```
MESSAGE

To: Tove
From: Jani

Don't forget me this weekend!
```

Imagine that the author of the XML document added some extra information to it:

```
<note>
<date>2008-01-10</date>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

Should the application break or crash? No. The application should still be able to find the <to>, <from>, and <body> elements in the XML document and produce the same output. One of the beauties of XML, is that it can often be extended without breaking applications.

3.6. XML ATTRIBUTES

XML elements can have attributes in the start tag, just like HTML. Attributes provide additional information about elements.

XML Attributes

From HTML you will remember this: <img src="computer.gif">. The "src" attribute provides additional information about the <img> element. In HTML (and in XML) attributes provide additional information about elements:

```
<img src="computer.gif">
<a href="demo.asp">
```

Attributes often provide information that is not a part of the data. In the example below, the file type is irrelevant to the data, but important to the software that wants to manipulate the element:

```
<file type="gif">computer.gif</file>
```

XML Attributes Must be Quoted

Attribute values must always be enclosed in quotes, but either single or double quotes can be used. For a person's sex, the person tag can be written like this:

```
<person sex="female">
```

or like this:

```
<person sex='female'>
```

If the attribute value itself contains double quotes you can use single quotes, like in this example:

```
<gangster name='George "Shotgun" Ziegler'>
```

or you can use character entities:

```
<gangster name="George &quot;Shotgun&quot; Ziegler">
```

XML Elements vs. Attributes

Take a look at these examples:

```
<person sex="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person>
  <sex>female</sex>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

In the first example sex is an attribute. In the last, sex is an element. Both examples provide the same information. There are no rules about when to use attributes and when to use elements. Attributes are handy in HTML. In XML my advice is to avoid them. Use elements instead.

The following three XML documents contain exactly the same information: A date attribute is used in the first example:

```
<note date="10/01/2008">
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!
</body>
</note>
```

A date element is used in the second example:

```
<note>
  <date>10/01/2008</date>
  <to>Tove</to>
  <from>Jani</from>
```

```
 <heading>Reminder</heading>
 <body>Don't forget me this weekend!</body>
</note>
```

An expanded date element is used in the third:

```
<note>
 <date>
  <day>10</day>
  <month>01</month>
  <year>2008</year>
 </date>
 <to>Tove</to>
 <from>Jani</from>
 <heading>Reminder</heading>
 <body>Don't forget me this weekend!</body>
</note>
```

Avoid XML Attributes?

Some of the problems with using attributes are:

attributes cannot contain multiple values (elements can)
attributes cannot contain tree structures (elements can)
attributes are not easily expandable (for future changes)

Attributes are difficult to read and maintain. Use elements for data. Use attributes for information that is not relevant to the data.

Don't end up like this:

```
<note day="10" month="01" year="2008"
to="Tove" from="Jani" heading="Reminder"
body="Don't forget me this weekend!">
</note>
```

XML Attributes for Metadata

Sometimes ID references are assigned to elements. These IDs can be used to identify XML elements in much the same way as the ID attribute in HTML. This example demonstrates this:

```
<messages>
```

```
 <note id="501">
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
 </note>
 <note id="502">
  <to>Jani</to>
  <from>Tove</from>
  <heading>Re: Reminder</heading>
  <body>I will not</body>
 </note>
</messages>
```

The ID above is just an identifier, to identify the different notes. It is not a part of the note itself.

Hence, metadata (data about data) should be stored as attributes, and that data itself should be stored as elements.

XML Validation

XML with correct syntax is "Well Formed" XML.

XML validated against a DTD is "Valid" XML.

Well Formed XML Documents

A "Well Formed" XML document has correct XML syntax.

The syntax rules:

XML documents must have a root element
XML elements must have a closing tag
XML tags are case sensitive
XML elements must be properly nested
XML attribute values must be quoted

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>Tove</to>
<from>Jani</from>
```

```
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

Valid XML Documents

A "Valid" XML document is a "Well Formed" XML document, which also conforms to the rules of a Document Type Definition (DTD):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The DOCTYPE declaration in the example above, is a reference to an external DTD file. The content of the file is shown in the paragraph below.

XML DTD

The purpose of a DTD is to define the structure of an XML document. It defines the structure with a list of legal elements:

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

XML Schema

W3C supports an XML-based alternative to DTD, called XML Schema:

```
<xs:element name="note">

<xs:complexType>
```

```
  <xs:sequence>
   <xs:element name="to" type="xs:string"/>
   <xs:element name="from" type="xs:string"/>
   <xs:element name="heading" type="xs:string"/>
   <xs:element name="body" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

</xs:element>
```

A General XML Validator

Note: Only Internet Explorer will actually check your XML against the DTD. Firefox, Mozilla, Netscape, and Opera will not.

Viewing XML Files

Raw XML files can be viewed in all major browsers.

Don't expect XML files to be displayed as HTML pages.

Viewing XML Files

```
<?xml version="1.0" encoding="ISO-8859-1"?>

 - <note>

     <to>Tove</to>

     <from>Jani</from>

     <heading>Reminder</heading>
```

```
    <body>Don't forget me this weekend!</body>

  </note>
```

The XML document will be displayed with color-coded root and child elements. A plus (+) or minus sign (-) to the left of the elements can be clicked to expand or collapse the element structure. To view the raw XML source (without the + and - signs), select "View Page Source" or "View Source" from the browser menu.

Note: In Chrome, Opera, and Safari, only the element text will be displayed. To view the raw XML, you must right click the page and select "View Source"

Viewing an Invalid XML File

If an erroneous XML file is opened, the browser will report the error. Look at this XML file: note_error.xml

Why Does XML Display Like This?

XML documents do not carry information about how to display the data. Since XML tags are "invented" by the author of the XML document, browsers do not know if a tag like <table> describes an HTML table or a dining table. Without any information about how to display the data, most browsers will just display the XML document as it is.

Below is a fraction of the XML file. The second line links the XML file to the CSS file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/css" href="cd_catalog.css"?>
<CATALOG>
 <CD>
  <TITLE>Empire Burlesque</TITLE>
  <ARTIST>Bob Dylan</ARTIST>
  <COUNTRY>USA</COUNTRY>
  <COMPANY>Columbia</COMPANY>
  <PRICE>10.90</PRICE>
  <YEAR>1985</YEAR>
 </CD>
 <CD>
  <TITLE>Hide your heart</TITLE>
  <ARTIST>Bonnie Tyler</ARTIST>
  <COUNTRY>UK</COUNTRY>
  <COMPANY>CBS Records</COMPANY>
  <PRICE>9.90</PRICE>
  <YEAR>1988</YEAR>
  </CD>
.
.
.
</CATALOG>
```

Formatting XML with CSS is not the most common method.

W3C recommend using XSLT instead.

Displaying XML with XSLT

XSLT is the recommended style sheet language of XML. XSLT (eXtensible Stylesheet Language Transformations) is far more sophisticated than CSS. XSLT can be used to transform XML into HTML, before it is displayed by a browser.

3.7. CONCLUSION

Processing an XML document requires a software program called an **XML Parser** or **XML Processor**. Most XML parsers are available at no charge and for a variety of programming languages (e.g., Java, Perl, C++).

3.8. SUMMARY

XML is actually a language used to create other markup languages to describe data in a structured manner. It is a widely supported **open technology** (i.e., non-proprietary technology) for data exchange and storage.

3.9. TUTOR MARKED ASSIGNMENT

1. Differentiate between XML and Html.
2. State which of the following statements are true and which are false. If false, explain why.
   (a) XML is a technology for creating markup languages.

   (b) XML is not case sensitive.

   (c) Forward and backward slashes (/ and \) delimit XML markup text.

   (d) All XML start document may contain only one root element.

(e) All XML start tags must have corresponding end tags.

(f) A DTD/Schema defines the presentation style of an XML document.

3. Write a scheme that provides tags for a person’s first name, last name, weight, and shoe size. Weight and shoe size tags should have attributes to designate measuring systems.

3.10. FURTHER READINGS/ RESOURCES

Bradley, Neil The XML Companion. Harlow, UK: Addison-Wesley, 2000.

Goldfarb, Charles F. and Prescod, Paul The XML Handbook. Upper Saddle River, NJ: Prentice Hall PTR, 2000.

Graham, Ian S. and Quin, Liam XML Specification Guide. New York, NY: John Wiley & Sons, Inc., 1999.

.

Hoque, Reaz XML for Real Programmers. San Diego, CA: Morgan Kaufmann, 2000.

Hunter, David Beginning XML. Acock's Green, Birmingham, UK: Wrox Press Ltd, 2000.

Kay, Michael XSLT: Programmer's Reference. Acock's Green, Birmingham, UK: Wrox Press Ltd, 2000.

Sturm, Jake Developing XML Solutions. Redmond, WA: Microsoft Press, 2000.

Young, Michael J. XML: Step by Step. Redmond, WA: Microsoft Press, 2000.

http://www.alphaWorks.ibm.com/ — IBM alphaWorks http://www.w3.org/XML/ — Extensible Markup Language (XML) http://msdn.microsoft.com/xml/default.asp — MSDN Online: XML Developer Center
~~http://msdn.microsoft.com/downlo~~ads/webtechnology/xml/msxml.asp — Microsoft XML Parser Preview Release

http://www.xmlsoftware.com/ — XMLSOFTWARE: The XML Software Site

http://www.oasis-open.org/cover/sgml-xml —The SGML/XML Web Page - Home Page

MODULE 3: DISTRIBUTED DATABASES

4.0. INTRODUCTION

A data warehouse is the central repository for the decision support systems (DSS). It is not a static database; instead it is a dynamic decision support framework, that is, almost by definition, always a work in progress.

4.1. OBJECTIVES

To understand data warehousing as an active decision support framework

To understand the architectures of data ware houses

To understand data ware housing schemes

4.2. WHAT IS A DATA WAREHOUSE?

A data warehouse is a relational database that is designed for query and analysis rather than for transaction processing. It usually contains historical data derived from transaction data, but it can include data from other sources. It separates analysis workload from transaction workload and enables an organization to consolidate data from several sources. In addition to a relational database, a data warehouse environment includes an extraction, transportation, transformation, and loading (ETL) solution, an online analytical processing (OLAP) engine, client analysis tools, and other applications that manage the process of gathering data and delivering it to business users.

A common way of introducing data warehousing is to refer to the characteristics of a data warehouse as set forth by William Inmon:

Subject Oriented

Integrated

Non-volatile

Time Variant

Subject Oriented

Data warehouses are designed to help you analyze data. For example, to learn more about your company's sales data, you can build a warehouse that concentrates on sales. Using this warehouse, you can answer questions like "Who was our best customer for this item last year?" This ability to define a data warehouse by subject matter, sales in this case, makes the data warehouse subject oriented.

Integrated

Integration is closely related to subject orientation. Data warehouses must put data from disparate sources into a consistent format. They must resolve such problems as naming conflicts and inconsistencies among units of measure. When they achieve this, they are said to be integrated.

Non-volatile

Non-volatile means that, once entered into the warehouse, data should not change. This is logical because the purpose of a warehouse is to enable you to analyze what has occurred.

Time Variant

In order to discover trends in business, analysts need large amounts of data. This is very much in contrast to online transaction processing (OLTP) systems, where performance requirements demand that historical data be moved to an archive. A data warehouse's focus on change over time is what is meant by the term time variant.

Contrasting OLTP and Data Warehousing Environments

Figure 4.1 illustrates key differences between an OLTP system and a data warehouse.

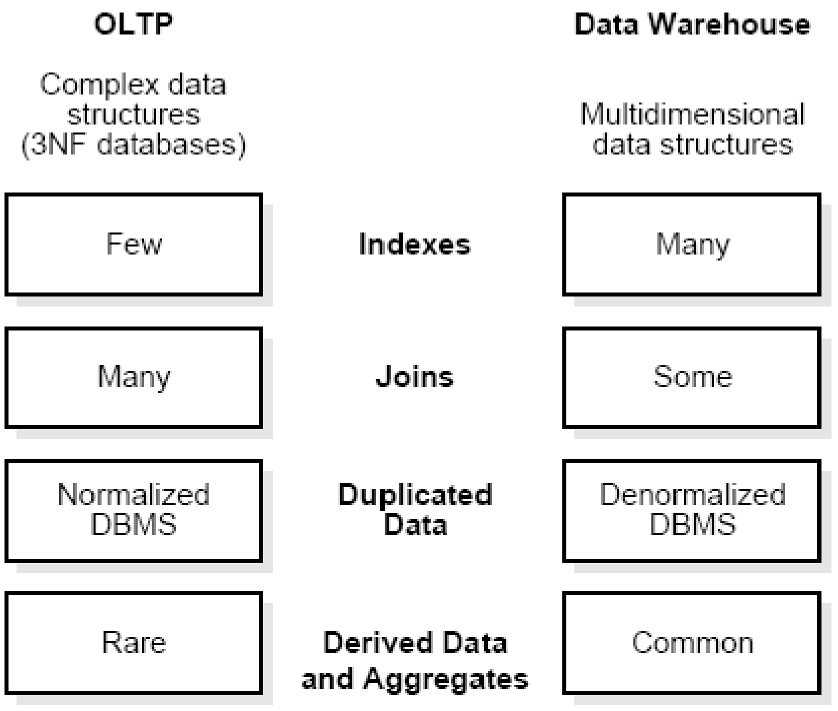


Figure 4.1: OLTP and Data Warehouse

One major difference between the types of system is that data warehouses are not usually in third normal form (3NF), a type of data normalization common in OLTP environments.

4.3. DATA WAREHOUSE ARCHITECTURES

Data warehouses and their architectures vary depending upon the specifics of an organization's situation. Three common architectures are:

Data Warehouse Architecture (Basic)

Data Warehouse Architecture (with a Staging Area)

Data Warehouse Architecture (with a Staging Area and Data Marts)

4.3.1. Data Warehouse Architecture (Basic)

In Figure 4.2, the metadata and raw data of a traditional OLTP system is present, as is an additional type of data, summary data. Summaries are very valuable in data warehouses because they pre-compute long operations in advance. For example, a typical data warehouse query is to retrieve something like August sales. A summary in Oracle is called a materialized view.

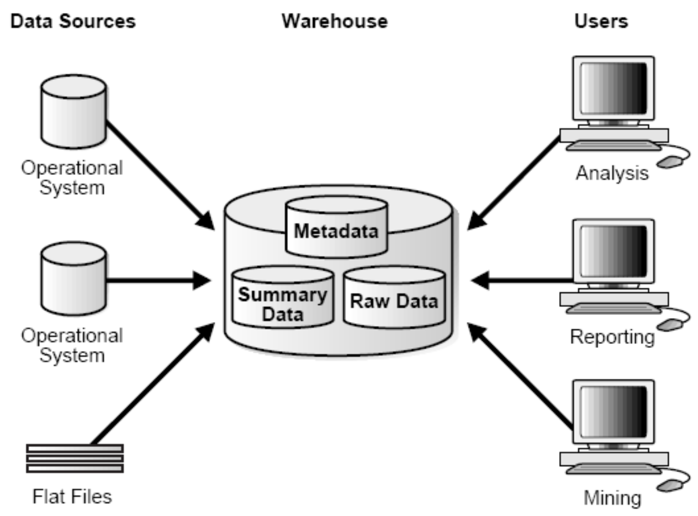


Figure 4.2: Architecture of a Data Warehouse.

4.3.2. Data Warehouse Architecture (with a Staging Area)

In Figure 4.2, you need to clean and process your operational data before putting it into the warehouse. You can do this programmatically, although most data warehouses use a staging area instead. A staging area simplifies building summaries and general warehouse management. Figure 4.3 illustrates this typical architecture.

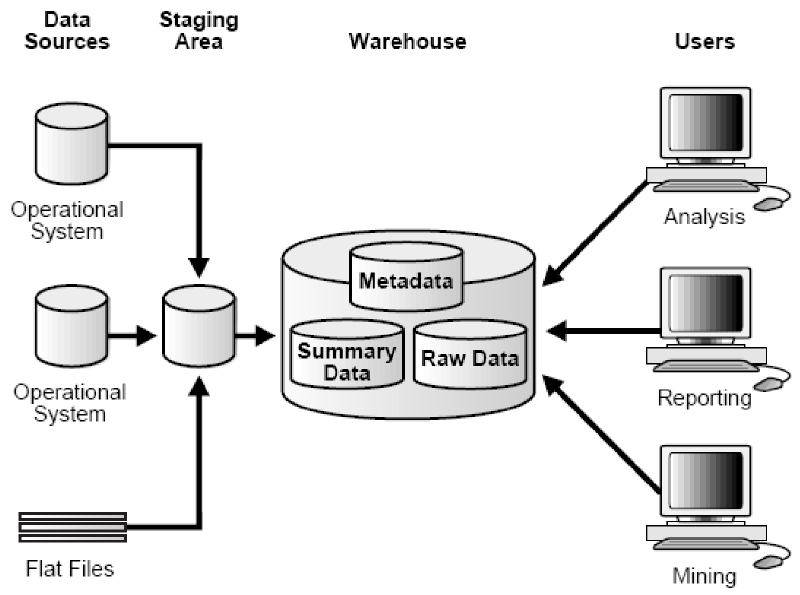


Figure 4.3: Architecture of a Data Warehouse with a Staging Area

4.3.3. Data Warehouse Architecture (with a Staging Area and Data Marts)

Although the architecture in Figure 4.3 is quite common, you may want to customize your warehouse's architecture for different groups within your organization. You can do this by adding data marts, which are systems designed for a particular line of business. Figure 4.4 illustrates an example where purchasing, sales, and inventories are separated. In this example, a financial analyst might want to analyze historical data for purchases and sales.

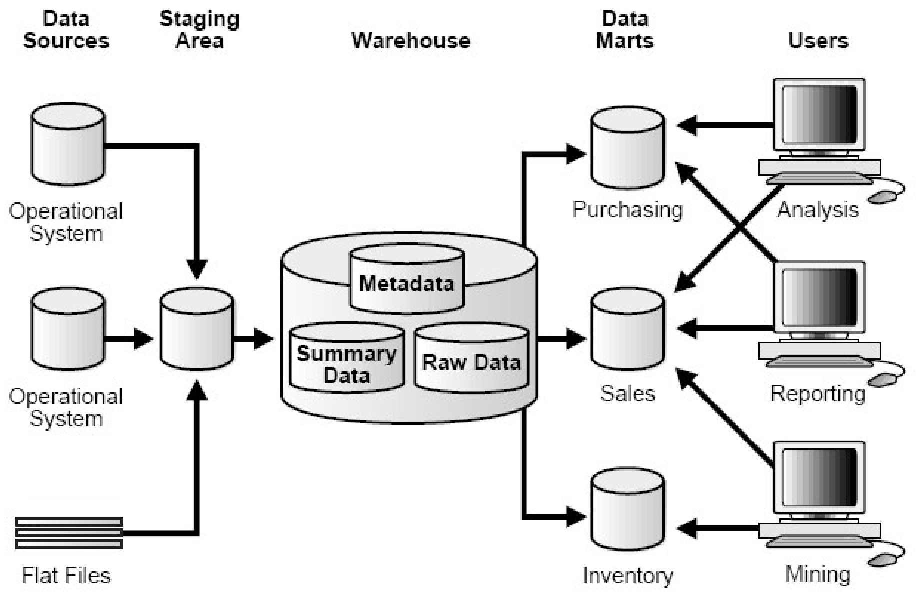


Figure 4.4: Architecture of a Data Warehouse with a Staging Area and Data Marta

4.4. LOGICAL VERSUS PHYSICAL DESIGN IN DATA WAREHOUSES

Your organization has decided to build a data warehouse. You have defined the business requirements and agreed upon the scope of your application, and created a conceptual design. Now you need to translate your requirements into a system deliverable. To do so, you create the logical and physical design for the data warehouse. You then define:

1. The specific data content
2. Relationships within and between groups of data
3. The system environment supporting your data warehouse
4. The data transformations required
5. The frequency with which data is refreshed

The logical design is more conceptual and abstract than the physical design. In the logical design, you look at the logical relationships among the objects. In the physical design, you look at the most effective way of storing and retrieving the objects as well as handling them from a transportation and backup/recovery perspective.

Orient your design toward the needs of the end users. End users typically want to perform analysis and look at aggregated data, rather than at individual transactions. However, end users might not know what they need until they see it. In addition, a well-planned design allows for growth and changes as the needs of users change and evolve.

By beginning with the logical design, you focus on the information requirements and save the implementation details for later.

Creating a Logical Design

A logical design is conceptual and abstract. You do not deal with the physical implementation details yet. You deal only with defining the types of information that you need.

One technique you can use to model your organization's logical information requirements is entity-relationship modelling. Entity-relationship modelling involves identifying the things of importance (entities), the properties of these things (attributes), and how they are related to one another (relationships).

The process of logical design involves arranging data into a series of logical relationships called entities and attributes. An entity represents a chunk of information. In relational databases, an entity often maps to a table. An attribute is a component of an entity that helps define the uniqueness of the entity. In relational databases, an attribute maps to a column.

To be sure that your data is consistent, you need to use unique identifiers. A unique identifier is something you add to tables so that you can differentiate between the same items when it appears in different places. In a physical design, this is usually a primary key.

While entity-relationship diagramming has traditionally been associated with highly normalized models such as OLTP applications, the technique is still useful for data warehouse design in the form of dimensional modelling. In dimensional modelling, instead of seeking to discover atomic units of information (such as entities and attributes) and all of the relationships between them, you identify which information belongs to a central fact table and which information belongs to its associated dimension tables. You identify business subjects or fields of data, define relationships between business subjects, and name the attributes for each subject.

Your logical design should result in (1) a set of entities and attributes corresponding to fact tables and dimension tables and (2) a model of operational data from your source into subject-oriented information in your target data warehouse schema.

You can create the logical design using a pen and paper, or you can use a design tool such as Oracle Warehouse Builder (specifically designed to support modelling the ETL process) or Oracle Designer (a general purpose modelling.

4.5. DATA WAREHOUSING SCHEMAS

A schema is a collection of database objects, including tables, views, indexes, and synonyms. You can arrange schema objects in the schema models designed for data warehousing in a variety of ways. Most data warehouses use a dimensional model. The model of your source data and the requirements of your users help you design the data warehouse schema. You can sometimes get the source model from your company's enterprise data model and reverse-engineer the logical data model for the data warehouse from this. The physical implementation of the logical data warehouse model may require some changes to adapt it to your system parameters—size of machine, number of users, storage capacity, type of network, and software.

Star Schemas

The star schema is the simplest data warehouse schema. It is called a star schema because the diagram resembles a star, with points radiating from a centre. The centre of the star consists of one or more fact tables and the points of the star are the dimension tables, as shown in Figure 4.5.

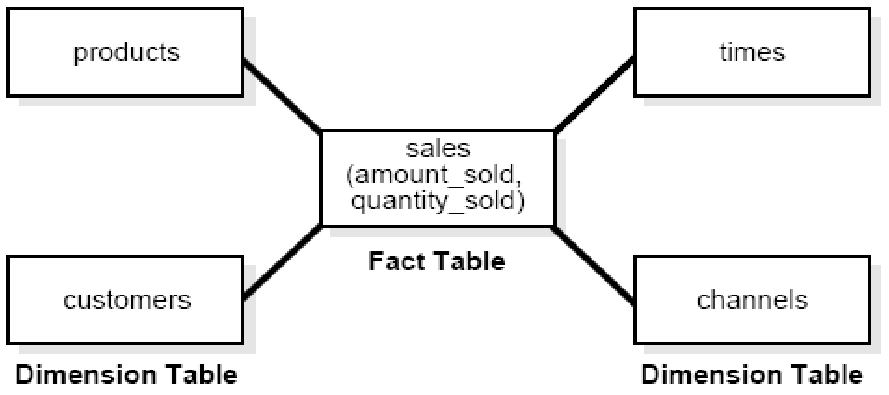


Figure 4.5: Star Schema

The most natural way to model a data warehouse is as a star schema, only one join establishes the relationship between the fact table and any one of the dimension tables.

A star schema optimizes performance by keeping queries simple and providing fast response time. All the information about each level is stored in one row.

Other Schemas

Some schemas in data warehousing environments use third normal form rather than star schemas. Another schema that is sometimes useful is the snowflake schema, which is a star schema with normalized dimensions in a tree structure.

4.6. DATA WAREHOUSING OBJECTS

Fact tables and dimension tables are the two types of objects commonly used in dimensional data warehouse schemas.

Fact tables are the large tables in your warehouse schema that store business measurements. Fact tables typically contain facts and foreign keys to the dimension tables. Fact tables represent data, usually numeric and additive, that can be analyzed and examined. Examples include sales, cost, and profit.

Dimension tables, also known as lookup or reference tables, contain the relatively static data in the warehouse. Dimension tables store the information you normally use to contain queries. Dimension tables are usually textual and descriptive and you can use them as the row headers of the result set. Examples are customers or products.

4.6.1. Fact Tables

A fact table typically has two types of columns: those that contain numeric facts (often called measurements), and those that are foreign keys to dimension tables. A fact table contains either detail-level facts or facts that have been aggregated. Fact tables that contain aggregated facts are often called summary tables. A fact table usually contains facts with the same level of aggregation. Though most facts are additive, they can also be semi-additive or non-additive. Additive facts can be aggregated by simple arithmetical addition. A common example of this is sales. Non-additive facts cannot be added at all. An example of this is averages. Semi-additive facts can be aggregated along some of the dimensions and not along others. An example of this is inventory levels, where you cannot tell what a level means simply by looking at it.

Creating a New Fact Table

You must define a fact table for each star schema. From a modelling standpoint, the primary key of the fact table is usually a composite key that is made up of all of its foreign keys.

4.6.2. Dimension Tables

A dimension is a structure, often composed of one or more hierarchies, that categorizes data. Dimensional attributes help to describe the dimensional value.

They are normally descriptive, textual values. Several distinct dimensions, combined with facts, enable you to answer business questions. Commonly used dimensions are customers, products, and time.

Dimension data is typically collected at the lowest level of detail and then aggregated into higher level totals that are more useful for analysis. These natural rollups or aggregations within a dimension table are called hierarchies.

Hierarchies

Hierarchies are logical structures that use ordered levels as a means of organizing data. A hierarchy can be used to define data aggregation. For example, in a time dimension, a hierarchy might aggregate data from the month level to the quarter level to the year level. A hierarchy can also be used to define a navigational drill path and to establish a family structure.

Within a hierarchy, each level is logically connected to the levels above and below it. Data values at lower levels aggregate into the data values at higher levels. A dimension can be composed of more than one hierarchy. For example, in the product dimension, there might be two hierarchies—one for product categories and one for product suppliers.

Dimension hierarchies also group levels from general to granular. Query tools use hierarchies to enable you to drill down into your data to view different levels of granularity. This is one of the key benefits of a data warehouse.

When designing hierarchies, you must consider the relationships in business structures. For example, a divisional multilevel sales organization.

Hierarchies impose a family structure on dimension values. For a particular level value, a value at the next higher level is its parent, and values at the next lower level are its children. These familial relationships enable analysts to access data quickly.

Levels A level represents a position in a hierarchy. For example, a time dimension might have a hierarchy that represents data at the month, quarter, and year levels. Levels range from general to specific, with the root level as the highest or most general level. The levels in a dimension are organized into one or more hierarchies.

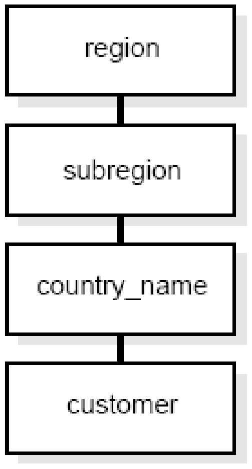
Level Relationships Level relationships specify top-to-bottom ordering of levels from most general (the root) to most specific information. They define the parent-child relationship between the levels in a hierarchy.

Hierarchies are also essential components in enabling more complex rewrites. For example, the database can aggregate an existing sales revenue on a quarterly base to a yearly aggregation when the dimensional dependencies between quarter and year are known.

Typical Dimension Hierarchy

Figure 4.6 illustrates a dimension hierarchy based on customers.

Figure 4.6: Typical Levels in a Dimension Hierarchy



Unique Identifiers

Unique identifiers are specified for one distinct record in a dimension table.

Artificial unique identifiers are often used to avoid the potential problem of unique identifiers changing. Unique identifiers are represented with the # character. For example, #customer_id.

Relationships

Relationships guarantee business integrity. An example is that if a business sells something, there is obviously a customer and a product. Designing a relationship between the sales information in the fact table and the dimension tables products and customers enforces the business rules in databases.

Example of Data Warehousing Objects and Their Relationships

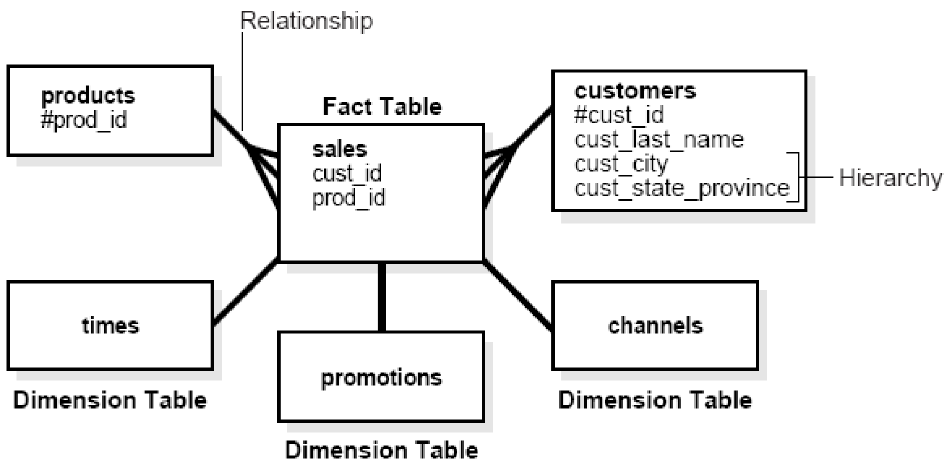


Figure 4.7: Typical Data Warehousing Objects

Figure 4.7 illustrates a common example of a sales fact table and dimension tables customers, products, promotions, times, and channels.

Moving from Logical to Physical Design

Logical design is what you draw with a pen and paper or design with Oracle

Warehouse Builder or Designer before building your warehouse. Physical design is the creation of the database with SQL statements.

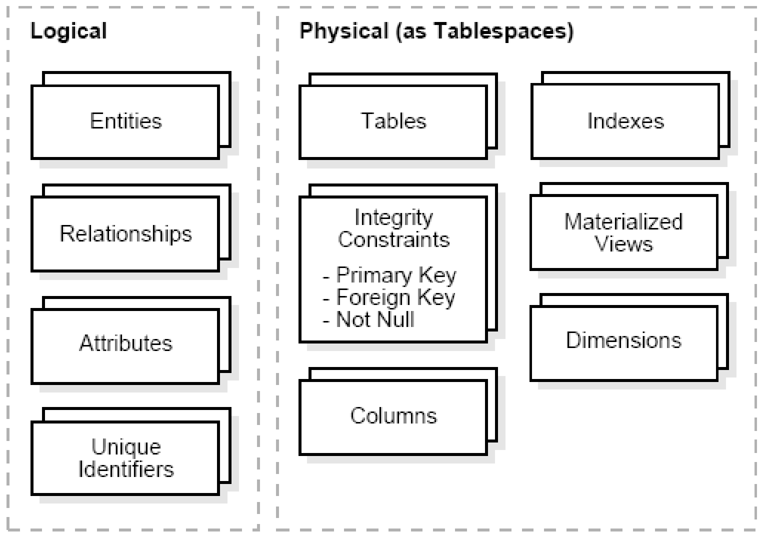


Figure 4.8: Logical Design with Physical Design

During the physical design process, you convert the data gathered during the logical design phase into a description of the physical database structure. Physical design decisions are mainly driven by query performance and database maintenance aspects. For example, choosing a partitioning strategy that meets common query requirements enables Oracle to take advantage of partition pruning, a way of narrowing a search before performing it.

Physical Design

During the logical design phase, you defined a model for your data warehouse consisting of entities, attributes, and relationships. The entities are linked together using relationships. Attributes are used to describe the entities. The unique identifier (UID) distinguishes between one instance of an entity and another.

Figure 4.8 offers you a graphical way of looking at the different ways of thinking about logical and physical designs.

During the physical design process, you translate the expected schemas into actual database structures. At this time, you have to map:

Entities to tables

Relationships to foreign key constraints

Attributes to columns

Primary unique identifiers to primary key constraints

Unique identifiers to unique key constraints

Physical Design Structures

Once you have converted your logical design to a physical one, you will need to create some or all of the following structures:

Tablespaces

Tables and Partitioned Tables

Views

Integrity Constraints

Dimensions

Some of these structures require disk space. Others exist only in the data dictionary.

Additionally, the following structures may be created for performance improvement:

Indexes and Partitioned Indexes

Materialized Views

Tablespaces

A tablespace consists of one or more datafiles, which are physical structures within the operating system you are using. A datafile is associated with only one tablespace. From a design perspective, tablespaces are containers for physical design structures.

Tablespaces need to be separated by differences. For example, tables should be separated from their indexes and small tables should be separated from large tables.

Tablespaces should also represent logical business units if possible. Because a tablespace is the coarsest granularity for backup and recovery or the transportable tablespaces mechanism, the logical business design affects availability and maintenance operations.

Tables and Partitioned Tables

Tables are the basic unit of data storage. They are the container for the expected amount of raw data in your data warehouse.

Using partitioned tables instead of non-partitioned ones addresses the key problem of supporting very large data volumes by allowing you to decompose them into smaller and more manageable pieces. The main design criterion for partitioning is manageability, though you will also see performance benefits in most cases because of partition pruning or intelligent parallel processing. For example, you might choose a partitioning strategy based on a sales transaction date and a monthly granularity. If you have four years' worth of data, you can delete a month's data as it becomes older than four years with a single, quick DDL statement and load new data while only affecting 1/48th of the complete table. Business questions regarding the last quarter will only affect three months, which is equivalent to three partitions, or 3/48ths of the total volume.

Partitioning large tables improves performance because each partitioned piece is more manageable. Typically, you partition based on transaction dates in a data warehouse. For example, each month, one month's worth of data can be assigned its own partition.

Data Segment Compression

You can save disk space by compressing heap-organized tables. A typical type of heap-organized table you should consider for data segment compression is partitioned tables.

To reduce disk use and memory use (specifically, the buffer cache), you can store tables and partitioned tables in a compressed format inside the database. This often leads to a better scale up for read-only operations. Data segment compression can also speed up query execution. There is, however, a cost in CPU overhead.

Data segment compression should be used with highly redundant data, such as tables with many foreign keys. You should avoid compressing tables with much update or other DML activity. Although compressed tables or partitions are updatable, there is some overhead in updating these tables, and high update activity may work against compression by causing some space to be wasted.

Views

A view is a tailored presentation of the data contained in one or more tables or other views. A view takes the output of a query and treats it as a table. Views do not require any space in the database.

Integrity Constraints

Integrity constraints are used to enforce business rules associated with your database and to prevent having invalid information in the tables. Integrity constraints in data warehousing differ from constraints in OLTP environments. In OLTP environments, they primarily prevent the insertion of invalid data into a record, which is not a big problem in data warehousing environments because accuracy has already been guaranteed. In data warehousing environments, constraints are only used for query rewrite. NOT NULL constraints are particularly common in data warehouses. Under some specific circumstances, constraints need space in the database. These constraints are in the form of the underlying unique index.

Indexes and Partitioned Indexes

Indexes are optional structures associated with tables or clusters. In addition to the classical B-tree indexes, bitmap indexes are very common in data warehousing environments. Bitmap indexes are optimized index structures for set-oriented operations. Additionally, they are necessary for some optimized data access methods such as star transformations.

Indexes are just like tables in that you can partition them, although the partitioning strategy is not dependent upon the table structure. Partitioning indexes makes it easier to manage the warehouse during refresh and improves query performance.

Materialized Views

Materialized views are query results that have been stored in advance so long-running calculations are not necessary when you actually execute your SQL statements. From a physical design point of view, materialized views resemble tables or partitioned tables and behave like indexes.

Dimensions

A dimension is a schema object that defines hierarchical relationships between columns or column sets. A hierarchical relationship is a functional dependency from one level of a hierarchy to the next one. A dimension is a container of logical relationships and does not require any space in the database. A typical dimension is city, state (or province), region, and country.

4.7. SUMMARY

The data warehouse is an integrated subject-oriented, time-variant, non-volatile database that provides support for decision making. The data warehouse is usually a read-only database optimized for dart analysis and query processing. Business data analysts access the data warehouse via front-end tools and end user application software to extract the data in usable forms.

4.8. CONCLUSION

Although it is easy to focus on the data warehouse database as the DSS central data repository, it is noteworthy that the decision support infrastructure includes hardware, software, people, and procedures, as well as data.

4.9. TUTOR-MARKED ASSIGNMENT (TMA).

1. What is data warehouse?

2. What are the main characteristics of data warehouse?

3. Your data warehousing project group is acquiring data about prototyping a data warehouse before its implementation. The project group members are especially concerned about the need to acquire some data warehousing skills between implementing the enterprise-wide data warehouse. What would you recommend? Explain your recommendations.

4.10.FURTHERREADINGS

Lane, P., Schupmann, V., [2002] *Oracle9i Data Warehousing Guide, Release 2 (9.2),* Oracle Corporation, 2002.

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[1986]. "Database System Concepts", McGraw – Hill, 1986.

www.w3schools.com

MODULE 3: DISTRIBUTED DATABASES

UNIT 5: INTRODUCTION TO DATA MINING

5.0. INTRODUCTION

Data mining is the search for relationships and global patterns that exist in large databases but are bidden in the vast amounts of data. In data mining, an analyst combines knowledge of the data with advanced machine learning technologies to discover nuggets of knowledge hidden in the data. Data mining software can find meaningful relationships that might take years to find with conventional techniques. The software is designed to sift through large collections of data and by using statistical and artificial intelligence techniques, identify hidden relationships. The mined data typically include electronic point of sales records, inventory, customer transactions, and customer records with matching demographics, usually obtained from an external source. Data mining does not require the presence of a data warehouse. An organization can mine data from its operational files or independent databases. However, data mining independent files will not uncover relationships that exist between data in different files. Data mining will usually be easier and more effective when the organization accumulates as much data as possible in a single data store, such as a data warehouse. Recent advances in processing speeds and lower storage costs have made large-scale mining of corporate data a reality.

Database marketing, a common application of data mining, is also one of the best examples of the effective use of the technology. Database marketers use data mining to develop, test, implement, measure, and modify tailored marketing programs. The intention is to use data to maintain a lifelong relationship with a customer. The database marketer wants to anticipate and fulfill the customer’s needs as they emerge. For example, recognizing that a customer buys a new car every three or four years and with each purchase gets an increasingly more luxurious car, the car dealer contacts the customer during the third year of the life of the current car with a special offer on its latest luxury model.

5.1. OBJECTIVES

To understand data mining as a tool for data analysis

To explore the applications and technologies of data mining

5.2. DATA MINING USES

There are many applications of data mining. For example:

Predicting the probability of default for consumer loan applications. Data mining can help lenders to substantially reduce loan losses by improving their ability to predict bad loans.

Reducing fabrication flaws in VLSI chips. Data mining systems can sift through vast quantities of data collected during the semiconductor fabrication process to identify conditions that are causing yield problems.

Predicting audience share for television programs. A market share prediction system allows television programming executives to arrange sow schedules to maximize market share and increase advertising revenues.

Predicting the probability that a cancer patient will respond to radiation therapy. By more accurately predicting the effectiveness of expensive medical procedures, health-care costs can be reduced without affecting quality of care.

Predicting the probability that an offshore oil well is going to produce oil. An offshore oil well may cost N30million. Data mining technology can increase the probability that this investment will be profitable.

Identifying quasars from trillion of bytes of satellite data. This was one of the earliest applications of data mining system as the technology was first applied in the scientific community.

Fingerhut. One of the U.S's oldest and largest catalog publishers, with sales of nearly N1.8 billion in 2000, claims to have collected data on 30 million households. However, it has not found a complete data analysis package to meet its needs. Because different business problems require different data analysis programs, the company has been creating a "toolbox" of these applications. The most recent tool in the box is Quad-stone, a data-mining solution which is being used to determine which customers should be contacted, how often, and what method works the best, catalogs and mailers, phone centers, or the internet. The software provides workers, not just the IAT managers, the capability to run analyses of model customer behaviour.

In addition. Fingerhut uses software from SAS and SPSS Inc, depending on which tool is more applicable to the data. Regression techniques are used in the SAS modules to derive formulas for customer habits. Sequences of events, such as the series of steps taken during a purchase, are analyzed using SPSS. Another tool, called knowledge-seeker from Angoss, helps analyze customers in categories, such as urban female shoppers. Around 100 data models are used-by Fingerhut to analyze customer data and behaviour on a regular basis. In addition, at least 100 marketing campaigns are undertaken each year.

Source: Tillet L. S. 2001. Data tools for Fingerhut. Internet week. Mar 5 (851):19

Figure 3.1 Creating a toolbox

5.3. DATA MINING FUNCTIONS

Based on the functions they perform, five types of data mining functions exist:

ASSOCIATIONS

An association function identifies affinities existing among the collection of items in a given set of records. These affinities or relationships can be expressed by rules such as, 72 percent of all the records that contain items *A*, *B*, and *C* also contain items *D* and *E*. Knowing that 85 percent of customers who buy a certain brand of wine also buy a certain type of pasta can help supermarkets improve use of shelf space and promotional offers. Discovering that fathers on the way home from work on Friday often grab a six-pack of beer after buying some diapers, enabled a supermarket to improve sales by placing beer specials next to diapers.

SEQUENTIAL PATTERNS

Sequential pattern mining functions identify frequently occurring sequences from given records. For example, these functions can be used to detect the set of customers associated with certain frequent buying patterns. Data mining might discover, for example, that 32 percent of female customers within six months of ordering a red jacket also buy a gray skirt. A retailer with knowledge of this sequential pattern can then offer the red-jacket buyer a coupon or other enticement to attract the prospective gray-skirt buyer.

CLASSFYING

Classifying divides predefined classes (e.g. types of customers) into mutually exclusive groups such that the members of each group are as close as possible to one another and different groups are as far as possible from one another, where distance is measured with respect to specific predefined variables. The classification of groups is done before data analysis. Thus, based on sales, customers may be first categorized as infrequent, occasional and frequent. A classifier could be used to identify those attributes, from a given set, that discriminate among the three types of customers. For example, a classifier might identify frequent customers as those with incomes above N50,000 and having two or more children. Classification functions have been used extensively in applications such as credit risk analysis, portfolio selection, health risk analysis, and image and speech recognition. Thus, when a new customer is recruited, the firm can use the classifying function to determine the customer's sales potential and accordingly tailor its market to that person.

CLUSTERING

Whereas classifying starts with predefined categories, clustering starts with just the data and discovers the hidden categories. These categories are derived from the data. Clustering divides a dataset into mutually exclusive groups such that the members of each group are as close as possible to one another and different groups are as far as possible from one another, where distance is measured with respect to all available. The goal of clustering is to identify categories. Clustering could be used, for instance, to identify natural groupings of customers by processing all the available data on them. Examples of applications that can use clustering functions are market segmentation, discovering affinity groups, and defect analysis.

PREDICTION

Prediction calculates the future value of a variable. For example, it might be used to predict the revenue value of a new customer based on that person's demographic variables.

These various data mining techniques can be used together. For example, a sequence pattern analysis could identify potential customers (e.g. red jacket leads to gray skirt), and then classifying could be used to distinguish between those prospects who are converted to customers and those who are (i.e. did not follow the sequential pattern of buying a gray-skirt). This additional analysis should enable the retailer to further its marketing strategy to increase the conversion rate of red jacket customers to gray skirt purchasers.

5.4. DATA MINING TECHNOLOGIES

Data miners use technologies that are based on statistical analysis and data visualization:

DECISION TREES

Tree-shaped structures can be used to represent decisions and rules for the classification of a dataset. As well as being easy to understand, tree-based models are suited to selecting important variables and are best when many of the predictors are irrelevant.

GENETIC ALGORITHMS

Genetic algorithms are optimization techniques that use processes such as genetic combination, mutation, and natural selection in a design based on the concepts of evolution. Possible solutions for a problem compete with each other. In an evolutionary struggle of the survival of the fittest, the best solution survives the battle. Genetic algorithms are suited for optimization problems with many candidate variables (e.g. candidates for a long).

K-NEAREST NEIGHBOR METHOD

The nearest neighbor method is used for clustering and classification. In the case of clustering, the method first plots each record in n-dimensional space, where attributes are used in the analysis. Then, it adjusts the weights for each dimension to cluster together data points with similar goal features. For instance, if the goal is to identify customers who frequently switch phone companies, the k-nearest method would adjust weights for relevant variables (such as monthly phone bill and percentage of non-U.S. calls) to cluster switching customers in the same neighborhood. Customers who did not switch would be clustered some distance apart.

NEURAL NETWORKS

A neural network, mimicking the neurophysiology of the human brain, can learn from examples to find patterns in data and classify data. While neural networks can be used for classification, they must first be trained to recognize patterns in a sample dataset. Once trained, a neural network can make predictions from new data. Neutral networks are suited to combining information from many predictor variables and work well when many of the predictors are partially redundant. One shortcoming of a neural network is that it can be viewed as a black box with no explanation of the results provided. Often managers are reluctant to apply models they do not understand, and this can limit the applicability of neural networks.

Figure 5.2: The SAS data mining method

DATA VISUALIZATION

Data visualization can make it possible for the analyst to gain a deeper, intuitive understanding of data. Because they present data in a visual format, visualization tools take advantage of our capability to rapidly discern visual pattern. Data mining can enable the analyst to focus attention on important pattern and trends and explore these in dept

A senior statistician at SAS Institute, a large supplier of statistical analysis software, advocates a five-step process to data mining, which he calls SEMMA.

Sample:	extract a portion of the dataset for data mining. This set should be large enough to contain significant data and small enough for rapid data mining.
Explore:	search for unanticipated trends and relationships to gain insights to the data and ideas for further exploration.
Modify:	create, select, and transform variables with the intention of building a model.
Model:	specify a relationship of variables that reliably predicts a desired goal.
Assess:	evaluate the practical value of the findings and the model resulting from the data mining effort.

Source: SAS Institute, 1996. Data mining reveals the diamonds in your database.

Figure 3.2 The SAS mining method

using visualization techniques. Data mining and data visualization work especially well together.

5.5. SUMMARY

Organizations are increasingly realizing that data are a key resource. It is necessary for the daily operations of a business and its future success. Recent developments in hardware and software have given organizations the capability to accumulate and process vast collections of data using organizational intelligence technologies. Data warehouse software supports the creation and management of huge data stores. The two approaches to exploiting data are verification and discovery. DSS, EIS and OLAP are mainly data verification methods. Data mining, a data discovery approach, uses statistical analysis techniques to discover bidden relationships. The relational model was not designed for OLAP, and the MDDB is the appropriate data store to support OLAP, MDDB design is based on recognizing variable and identifier dimensions.

5.6. CONCLUSION

Data management is a rapidly evolving discipline. Where once the spotlight was clearly on TPSs and the relational model, there are now multiple centers of attention. In an information economy, the knowledge to be gleaned from data collected by routine transactions can be an important source of competitive advantage. The more an organization can learn about its customers by studying their behaviour, the more likely it can provide superior products and services to retain existing customers and lure prospective buyers. As a result, data managers now have the dual responsibility of administering database that keep the organization in business today and tomorrow. They must now master the organizational intelligence technologies described in this unit.

5.7. TMA

1. What is data mining?
2. How does data mining differ from traditional decision support systems (DSS) tools?

5.8. FURTHER READINGS

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[2000]. "Database System Concepts", 4th ed., McGraw – hill, 2000.

Zobel, J., Moffat, A., and Sacks–Davis, R., [1992]. "An Efficient Indexing Technique for Full – Text Database Systems," in VLDB [1992.

www.w3schools.com

MODULE 4 : EMERGING TRENDS & EXAMPLES OF DATABASE ARCHITECTURE

UNIT 1: EMERGING DATABASE MODELS

1.0. INTRODUCTION

Applications in domains such as Multimedia, Geographical Information Systems, and digital libraries demand a completely different set of requirements in terms of the underlying database models. The conventional relational database model is no longer appropriate for these types of data. Furthermore the volume of data is typically significantly larger than in classical database systems. Finally, indexing, retrieving and analyzing these data types require specialized functionality not available in conventional database systems. The unit will cover the some requirements of these emerging databases such as multimedia databases, spatial databases, temporal databases and biological/genome databases, their underlying technologies, data models and languages.

1.1. OBJECTIVES

To understand the various database models and their limitations
To understand temporal management concepts, design, and challenges
To understand the design & Implementation of temporal databases

1.2. LIMITATIONS OF CONVENTIONAL DATABASES

Conventional databases typically consist of many tables, each of which is composed of a number of columns. The definition of those tables and columns determine the storage capabilities of the database, whereas the relations between the columns define the kinds of facts that can be stored in such a database. Those columns and relations determine the database structure that defines the expression capabilities of the database. Similar rules apply for the structure of data exchange files and thus for the information that is exchanged in electronic data files. This conventional database technology has some major constraints:

When data was not covered during the database design and thus is not included in the data model, then such data cannot be stored in the database nor exchanged via such a data file structure. Different databases have different data structures, which causes that data in one database cannot be integrated with data from other databases nor exchanged between databases without dedicated data conversion. A database modification or extension requires redesign of the database structure, modification of software and data conversion, which makes it a relatively complicated and costly exercise.

Another characteristic of conventional databases is that there are hardy international standards available or used for the content of the databases, being the data that is entered by its users. This typically means that local conventions are applied to limit the diversity of data that may be entered in those databases. As local conventions usually differ from other local conventions this has as disadvantage that data that are entered in one database cannot be compared or integrated with data in other databases, even if those database structures are the same and even if the application domain of the databases is the same. For example, within a company there may be various implementations of the same system in various sites for the storage of data about equipment, whereas for example the performance data about the same type of equipment still cannot be compared with the performance data in another location, because the equipment types have different names and the properties are also different.

Characteristics of a Gellish Database

A Gellish database does not have the semantic limitations that conventional databases have, because of the flexible and open Gellish language and because of its standard universal data structure (grammar), which is simple, computer and human interpretable. A Gellish database consists of one or more database tables, each of which has the same table structure (column definitions). The fact that those Gellish Database tables are standardized and universally applicable makes a Gellish database application independent. A standardized Gellish database table is universally applicable because it enables the application of the following two fundamental principles:

Explicit classification of individual things or explicit specialization of classes, with an unlimited number of classes in a dictionary. The Gellish database table enables to store any kind of object; because any individual object can be introduced by specification of an explicit classification relation between the object and a class, whereas classes (kinds of objects or concepts) can be selected from the very large number of classes that are already defined in the Gellish English Dictionary and if the proper class is not available it can be added by specification of a subtype-supertype relation with a direct supertype of the new class. This is fundamentally different from conventional databases that predefine the object types (classes) about which information can be stored by defining a limited number of entity types and attribute types in a fixed data model.

Explicit classification of relations (facts), by an extensible unlimited number of standardized relation types. The Gellish database table enables to store any kind of fact about any kind of object, because any fact is expressed by a relation, whereas those relations are explicitly classified by relation types that can be selected from the standardized relation types that are defined in the Gellish Dictionary or by relation types that are added to the dictionary as proprietary extensions. This is fundamentally different from conventional databases.

1.3. WHAT IS A MULTIMEDIA DATABASE?

A multimedia database is a database that hosts one or more primary media file types such as .txt (documents), .jpg (images), .swf (videos), .mp3 (audio), etc. And loosely fall into three main categories:

Static media (time-independent, i.e. images and handwriting)
Dynamic media (time-dependent, i.e. video and sound bytes)
Dimensional media (i.e. 3D games or computer-aided drafting programs- CAD) All primary media files are stored in binary strings of zeros and ones, and are encoded according to file type. The term "data" is typically referenced from the computer point of view, whereas the term "multimedia" is referenced from the user point of view.

Types of Multimedia Databases

There are numerous different types of multimedia databases, including:

The Authentication Multimedia Database (also known as a Verification Multimedia Database, i.e. retina scanning), is a 1:1 data comparison

The Identification Multimedia Database is a data comparison of one-to-many (i.e. passwords and personal identification numbers

A newly-emerging type of multimedia database, is the Biometrics Multimedia Database; which specializes in automatic human verification based on the algorithms of their behavioral or physiological profile. This method of identification is superior to traditional multimedia database methods requiring the typical input of personal identification numbers and passwords. Due to the fact that the person being identified does not need to be physically present, where the identification check is taking place. This removes the need for the person being scanned to remember a PIN or password. Fingerprint identification technology is also based on this type of multimedia database.

Difficulties Involved with Multimedia Databases

The difficulty of making these different types of multimedia databases readily accessible to humans is:

The tremendous amount of bandwidth they consume;

Creating Globally-accepted data-handling platforms, such as *Joomla*, and the special considerations that these new multimedia database structures require.

Creating a Globally-accepted operating system, including applicable storage and resource management programs need to accommodate the vast Global multimedia information hunger.

Multimedia databases need to take into accommodate various human interfaces to handle 3D-interactive objects, in an logically-perceived manner (i.e. SecondLife.com).

Accommodating the vast resources required to utilize artificial intelligence to its fullest potential- including computer sight and sound analysis methods.

The historic relational databases (i.e. the Binary Large Objects - BLOBs- developed for SQL databases to store multimedia data) do not conveniently support content-based searches for multimedia content. This is due to the relational database not being able to recognize the internal structure of a Binary Large Object and therefore internal multimedia data components cannot be retrieved.

Basically, a relational database is an "everything or nothing" structure- with files retrieved and stored as a whole, which makes a relational database completely inefficient for making multimedia data easily accessible to humans. In order to effectively accommodate multimedia data, a database management system, such as an Object Oriented Database (OODB) or Object Relational Database Management System (ORDBMS). Examples of Object Relational Database Management Systems include Odaptor (HP), UniSQL, ODB-II, and Illustra. The flip-side of the coin, is that unlike non-multimedia data stored in relational databases, multimedia data cannot be easily indexed, retrieved or classified, except by way of social bookmarking and ranking-rating, by actual humans. This is made possible by metadata retrieval methods, commonly referred to as tags, and tagging. This is why you can search for dogs, as an example, and a picture comes up based on your text search term. This is also referred to a schematic mode. Whereas doing a search with a picture of a dog to locate other dog pictures is referred to as paradigmatic mode.

However, metadata retrieval, search, and identify methods severely lack in being able to properly define uniform space and texture descriptions, such as the spatial relationships between 3D objects, etc.

The Content-Based Retrieval multimedia database search method (CBR), however, is specifically based on these types of searches. In other words, if you were to search an image or sub-image; you would then be shown other images or sub-images that related in some way to your the particular search, by way of color ratio or pattern, etc.

1.4 TEMPORAL DATABASES: MODELING TEMPORAL DATA; TEMPORAL SQL

1.4.1 Introduction

A wide range of database applications manage time-varying data. In contrast, existing database technology provides little support for managing such data. The research area of temporal databases aims to change this state of affairs by characterizing the semantics of temporal data and providing expressive and efficient ways to model, store, and query temporal data. This unit offers a brief introduction to temporal database research. It concisely introduces fundamental temporal database concepts, surveys state-of-the-art solutions to challenging aspects of temporal data management, and also offers a look into the future of temporal database research.

Temporal database management is a vibrant field of research, with an active community of several hundred researchers who have produced some 2000 papers over the last two decades. Most of these papers are listed in a series of seven cumulative bibliographies (the newest one provides pointers to its predecessors). The field has produced a comprehensive glossary of terminology , an edited volume which captures state of the art circa 1993, and three workshop proceedings. The near complete SQL3 standard includes a Part 7, SQL/Temporal. The topic of temporal databases is now covered in textbooks and in encyclopedia.

1.4.2 Temporal Data Semantics

Before we proceed to consider temporal data models and query languages, we examine, in data model-independent terms, the association of times and facts, which is at the core of temporal data management.

Initially, a brief description of terminology is in order. A database models and records information about a part of reality, termed either the *modeled reality* or the *mini-world*. Aspects of the mini-world are represented in the database by a variety of structures that we will simply term *database entities*. We will employ the term “fact” for any (logical) statement that can meaningfully be assigned a truth value, i.e., that is either true or false. In general, times are associated with database entities.

The facts recorded by the database entities are of fundamental interest. Several different temporal aspects may be associated with these. Most importantly, the *valid time* of a fact is the collected times—possibly spanning the past, present, and future—when the fact is true in the mini-world. Valid time thus captures the time-varying states of the mini-world. All facts have a valid time by definition. However, the valid time of a fact may not necessarily be recorded in the database, for any of a number of reasons. For example, the valid time may not be known, or recording it may not be relevant for the applications supported by the database. If a database models different possible worlds, the database facts may have several valid times, one for each such world.

Next, the *transaction time* of a database fact is the time when the fact is current in the database. Unlike valid time, transaction time may be associated with any database entity, not only with facts. For example, transaction time may be associated with objects and values that are not facts because they cannot be true or false in isolation. To be more concrete, the value “63” may be stored in a database, but does not denote a logical statement. It is meaningful to associate transaction time with “63,” but not valid time. Thus, all database entities have a transaction-time aspect. This aspect may or may not, at the database designer’s discretion, be captured in the database. The transaction-time aspect of a database entity has a duration: from insertion to deletion, with multiple insertions and deletions being possible for the same entity. As a consequence of the semantics of transaction time, capturing this aspect of database entities renders deletions purely logical. Deleting an entity does not physically remove the entity from the database; rather, the entity remains in the database, but ceases to be part of the database’s current state. Transaction time captures the time-varying states of the database, and applications that demand accountability or "traceability" rely on databases that record transaction time.

Observe that the transaction time of a database fact, say “*F*,” is the valid time of the related fact, “*F is current in the database*.” This would indicate that supporting transaction time as a separate aspect is redundant. However, both valid and transaction time are aspects of the content of all databases, and recording both of these is essential in a wide range of applications. In addition, transaction time, due to its special semantics, is particularly well-behaved and may be supplied automatically by the DBMS. Specifically, the transaction times of facts stored in the database are bounded by the time the database was created at one end of the time line and by the current time at the other end. This provides the rationale for the focus of most temporal database research on providing improved support for valid time and transaction time as separate aspects.

In addition, some other times have been considered, e.g., decision time . But the desirability of building decision time support into temporal database technologies is unclear, because the number and meaning of “the decision time(s)” of a fact varies from application to application and because decision times, unlike transaction time, generally do not exhibit specialized properties. The valid and transaction time values of database entities are drawn from some appropriate time domain. There is no single answer to how to perceive time in reality and how to represent time in a database, and different time domains may be distinguished with respect to several orthogonal characteristics. First, the time domain may or may not stretch infinitely into the past and future. Second, time may be perceived as discrete, dense, or continuous. Some feel that time is really continuous; others contend that time is discrete and that continuity is just a convenient abstraction that makes it easier to reason mathematically about discrete phenomena. In databases, a finite and discrete time domain is typically assumed, e.g., in the SQL standards. Third, a variety of different structures have been imposed on time. Most often, time is assumed to be totally ordered, but various partial orders have also been suggested, as having cyclic time.

An aspect of time that has been intriguing philosophers for centuries and that is difficult to describe fully is the concept of the current time, which we term *now* . This concept is unique to time; indeed, there really does not exist any other notion quite like it. Among its properties, the current time is ever-increasing, all activity is trapped at the current time, and the current time separates the past from the future. The spatial equivalent, *here*, simply fails to enjoy the properties of *now*. As Merrick Furst puts it, “The biggest difference between time and space is that you can’t reuse time.” The uniqueness of *now* is one of the reasons why techniques from other research areas are not readily, or not at all, applicable to temporal data; *now* offers new data management challenges, which are particular to temporal databases.

Much research has been conducted on the semantics and representation of time, from quite theoretical topics such as temporal logic and infinite periodic time sequences to more applied questions such as how to represent time values in minimal space. Substantial research has been conducted that concerns the use of different time granularities and calendars in general, as well as the issues surrounding the support for indeterminate time values. Also, there is a significant body of research on time data types, e.g., time points, time intervals (or “periods”), and temporal elements (sets of intervals) .

Designing Temporal Databases

The design of appropriate database schemas is critical to the effective use of database se technology and the construction of effective information systems that exploit this technology. Database schemas capturing time-referenced data are often particularly complex and thus difficult to design.

The first of the two traditional contexts of database design is the data model of the DBMS to be used for managing the data. This data model, generally a variant of the relational model, is assumed to conform to the ANSI/X3/SPARC three-level architecture. In this context, database design must thus be considered at each of the view, logical, and physical (or, “internal”) levels. In the second context, a database is modeled using a high-level, conceptual design model, typically the Entity-Relationship model. This model is independent of the particular implementation data model that is eventually to be used for managing the database, and it is designed specifically with data modeling as its purpose, rather than implementation or data manipulation, making it more attractive for the DBMS to be used.

1.4.3 Temporal Data Models and Query Languages

Temporal data management can be very difficult using conventional (non-temporal) data models and query languages. Accommodating the time-varying nature of the enterprise is largely left to the developers of database applications, leading to ineffective and inefficient ad-hoc solutions that must be reinvented each time a new application is developed. The result is that data management is currently an excessively involved and error-prone activity. unit 1.4.3.1 considers temporal data models, and unit 1.4.3.2 then covers query languages that are based on these data models. The subsequent step of providing support for temporal data modeling and database design is covered in unit 1.4.3.4.

We proceed to consider in turn logical and conceptual design of temporal databases.

1.4.3.1 Logical Design

A central goal of conventional relational database design is to produce a database schema consisting of a set of *relation schemas*. In normalization theory, normal forms constitute attempts at characterizing "good" relation schemas, and a wide variety of normal forms has been proposed, the most prominent being third normal form and Boyce-Codd normal form. An extensive theory has been developed to provide a solid formal footing for relational database design, and most database textbooks expose their readers to the core of this theory.

In temporal databases, there is an even greater need for database design guidelines. However, the conventional normalization concepts are not applicable to temporal relational data models because these models employ relational structures different from conventional relations. New temporal normal forms and underlying concepts that may serve as guidelines during temporal database design are needed. In response to this need, a range of temporal normalization concepts have been proposed, including temporal dependencies, keys, and normal forms. The relation in Figure 1.0 also rules out any of the dependencies when we apply regular dependencies directly. Considering that the different representations of the *CheckedOut* relation model the same miniworld and are capable of recording the same information, it may reasonably be assumed that these different representations would satisfy the same dependencies. At *any point in time*, a customer may have checked out several tapes. In contrast, a tape can only be checked out by a single customer at a single point in time. With this view, *TapeNumtemporally* determines *CustomerID*, but the reverse does not hold.

If we consider the information contents of a temporal relation, independent of its actual format, to be the set of conventional snapshot relations it logically comprises, we achieve a means of applying the conventional relational normalization theory that leads to a temporal theory, which naturally generalizes conventional dependencies and may be applied to dependencies other than functional. Specifically, a temporal relation satisfies a temporal dependency if all its snapshots satisfy the corresponding conventional dependency.

With this notion of temporal dependency based on snapshots, a temporal normalization theory may be built that parallels conventional normalization theory and that is independent of any particular representation of a temporal relation. However, the resulting theory, while temporal in that it applies to temporal databases, is actually atemporal, in that it applies to each snapshot of a temporal relation in isolation. This theory therefore fails to account for "temporal" aspects of data. Temporal data models generally define timeslice operators, which may be used to determine the snapshots contained in a temporal relation. Accepting a temporal relation as their argument and a time point as their parameter, these operators return the snapshot of the relation corresponding to the specified time point. For example, a timeslice operator for temporal relations like the one in Figure 1.0 may take a point (,) in bitemporal space as its parameter. It returns the tuples of the argument relation that contain this time point, but omitting the timestamp attribute. It is also relevant to consider dependencies and associated normal forms that effectively hold *between* time points. One approach to achieve this is to build the notion of time granularity into the normalization concepts. As a result, it not only is possible to consider snapshots computed at non-decomposable time points, but it is also possible to consider snapshots computed at coarser granularities. In our example relations, we have used day as the finest granularity; with the generalized theory, weeks and months may also be considered.

Another approach to taking the temporal aspects of data into account during database design is to introduce new concepts that capture the temporal aspects of data and may form the basis for new database design guidelines .

Perhaps most prominently, *time patterns* may be used for capturing when the values of an attribute for an entity change in the modeled reality and in the database. For example, the set of tapes checked out by a customer may be expected to change substantially more frequently than the customer's address, meaning that the addresses of customers and their checked out video tapes should be stored in separate relations. (In this example, the temporal counterpart of Boyce-Codd normal form may reasonably be assumed to also imply such a decomposition, but this does not apply generally.)

Next, the concept of *lifespan*, that captures when an attribute of an entity has values, also has implications for database design. Specifically, if the *lifespan* of two attributes differ, null values of the unattractive "do not exist" variety result unless the attributes are stored in separate relations. Assuming that the temporal data model used timestamps tuples, attributes should also be stored separately when different temporal aspects need to be captured for them or when the temporal aspects are captured with differing precisions (resulting in different timestamp granularities).

1.4.3.2 Conceptual Design

By far, most research on the conceptual design of temporal databases has been in the context of the Entity-Relationship (ER) model. This model, in its varying forms, is enjoying a remarkable, and increasing, popularity in industry. Building on the example introduced in unit 1.5.3.1, Figure 1.0 illustrates a conventional ER diagram for video rentals. The research on temporal ER modeling is well motivated. It is widely known that the temporal aspects of the mini-world are very important in a broad range of applications, but are also difficult to capture using the ER model. Put simply, diagrams that would be intuitive and easy to comprehend without the temporal aspects become obscure and cluttered when an attempt is made to capture the temporal aspects.

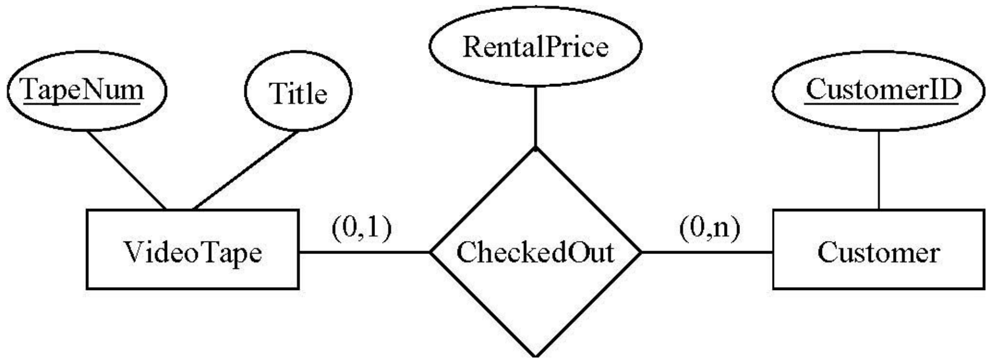


Figure 1.0: Non-temporal Conventional ER Diagram for Video Rentals

The diagram in Figure 1.0 is non-temporal, modeling the mini-world at a single point in time. Attempting to capture the temporal aspects that are essential for this application complicates matters. It is necessary to capture the time when a customer has checked out a video tape. And since it is possible for the same customer to have checked out the same tape at different times, the *CustomerID* and *TapeNum* attributes do not identify a single instance of *CheckedOut*. Instead, it is necessary to make *CheckedOut* a ternary relationship type, *start* introducing a new, somewhat artificial, entity type that captures the times of rentals. Including *start* and *end* time attributes on this entity type, the *CustomerID*, *TapeNum*, and *StartTime* attributes identify instances of *CheckedOut*. But simply requiring that these three attributes be a key of a relation representing rentals does not ensure the integrity of this relation—it remains possible for the same tape to be checked out more than once at the same point in time. As another issue, rental prices may vary over time, e.g., due to promotions and films getting old. Finally, including transaction time leads to further complications. As a result, some industrial users simply choose to ignore all temporal aspects in their ER diagrams and supplement the diagrams with textual phrases to indicate that a temporal dimension to data exists, e.g., “full temporal support.” The result is that the mapping of ER diagrams to relations must be performed by hand; and the ER diagrams do not document fully the temporally extended relational database schemas used by the application programmers. The research community’s response to this predicament has been to develop temporally enhanced ER models. Indeed, about a dozen such models have been reported in the research literature . These models represent attempts at modeling the temporal aspects of information more naturally and elegantly. The proposed extensions are based on quite different approaches.

One approach is to give all existing ER model constructs temporal semantics, basically following the “applies to all snapshots” approach used for making conventional normalization concepts “(a)temporal” in the previous unit. In its extreme form, this approach does not result in any new syntactical constructs— all the original constructs have simply become temporal. The simplicity of this wholesale approach is attractive. However, this approach rules out databases with non-temporal parts; and legacy diagrams are no longer valid, i.e., while their syntax remains valid, their semantics have changed, and they therefore no longer describe the existing relational databases.

Another approach is to devise new notational shorthand that replace some of the patterns that occur frequently in ER diagrams when temporal aspects are being modeled. One example is the pattern that occurs when modeling a time-varying attribute in the ER model (e.g., the *RentalPrice* in our example). With this approach, it is possible to retain the existing ER-model constructs with their old semantics. This type of model may be more difficult to understand, but it does not invalidate legacy diagrams, and it is also possible to design non-temporal databases as well as databases where some parts are non-temporal while others are temporal.

In brief, the ideal temporal ER model is easy to understand in terms of the ER model; does not invalidate legacy diagrams and database applications; and does not restrict the database to be temporal, but rather permits the designer to mix temporal and non-temporal parts. The existing models typically assume that their schemas are mapped to schemas in the relational model that serves as the implementation data model. The mapping algorithms are constructed to add appropriate time-valued attributes to the relation schemas. None of the models have one of the many time-extended relational models as their implementation model. These models have data definition and query language capabilities that better support the management of temporal data and would thus constitute natural candidate implementation platforms. Also, mappings to emerging models (e.g., SQL3) are missing. It remains a challenge to design mappings that maximally exploit these and other candidate implementation platforms.

1.5TEMPORALDBMSIMPLEMENTATION

There has been a vast amount of work in storage structures and access methods for temporal data, and a dozen-odd temporal DBMS prototypes have been reported . Two basic approaches may be discerned. Traditionally, an *integrated* approach has been assumed, in which the internal modules of a DBMS are modified or extended to support time-varying data. More recently, a *layered* approach has also received attention . Here, a software layer interposed between the user-applications and a conventional DBMS effectively serves as an advanced application that converts temporal query language statements into conventional statements that are subsequently executed by the underlying DBMS, which is itself not altered. While the former approach ensures maximum efficiency, the latter approach is more realistic in the short and medium term. Consistent with the vast majority of papers on temporal DBMS implementation, this unit assumes an integrated approach utilizing *timestamping* of tuples with time intervals, unless explicitly stated otherwise.

1.5.1. Query Processing

A query formulated in some high-level, user-oriented query language is typically translated into an equivalent query, formulated in a DBMS-internal, algebraic query language. The DBMS then optimizes this algebraic expression by transforming it into an equivalent expression that is expected to be more efficient to process, the result being better query processing performance.

Optimization of temporal queries offers new challenges over optimization of conventional queries. At the core of the matter, temporal database queries are often large and complex. A recent book offers a wide range of examples of typical temporal database queries that are generally very complex . Because of this added complexity, it is not only more important, but also more challenging, to optimize temporal database queries.

Specifically, the predicates used in temporal queries make these queries difficult to optimize. In non-temporal database applications, predicates are often equality predicates. As a reflection of this, much research in query processing has concentrated on equality predicates, and existing DBMSs are optimized for equality predicates (which occur in, e.g., equi-joins and natural joins). In contrast, temporal queries typically involve numerous inequality predicates. The perhaps most prominent source of such predicates is the test of overlap among two intervals. Inherent in temporal joins, this test occurs frequently in temporal queries and results in two equality predicates. Specifically, two intervals

and

overlap if the begin value of

is less than or equal to the end value of

and the begin value of

is less than or equal to the end value of

. Conventional DBMSs typically resort to nested-loop implementations of joins involving such inequality predicates, with their associated inefficiency. Other challenges posed by the complexity of temporal queries concern the coalescing of data and the interactions among coalescing, duplicate removal, and ordering .

There are new and unexploited opportunities for query optimization when time is present. The current time advances continuously; and for transaction time, the time value used most recently in updates is the largest value used so far. This implies that a natural clustering or sort order will manifest itself. If relations are partitioned so that current and logically deleted tuples are stored separately, the relation with current tuples will be clustered on their transaction-time start values, while the tuples in the relation with logically deleted tuples will be clustered on their end times. Characteristics such as these can be exploited during query optimization and evaluation.

As another example of an optimization opportunity, the integrity constraint that the begin value of an interval is less than or equal to its end value holds for all intervals in the database. Next, for many relations, the intervals associated with a key value are contiguous in time, with one interval starting exactly when the previous interval ended. Semantic query optimization can exploit these integrity constraints, as well as additional ones that can be inferred.

1.5.2. Implementing Algebraic Operators

As explained earlier, a user-specified query is translated into an internal, algebraic form, which is then optimized using equivalence-preserving transformations. The DBMS has available a library of algorithms that implement the operations that occur in the resulting algebraic formulation of the query. As the next step, algorithms are chosen from the library for each operation, upon which the query is ready for execution. Good performance is dependent on the availability of good implementations of the operations.

Focus has been on a number of temporal algebraic operators, including selection, joins, aggregates, and duplicate elimination. Conventional approaches to computing these operators typically have poor performance, and new opportunities exist for efficiently implementing these operators. The selection operator is examined in the next unit, as its implementation often involves a temporal index.

A wide variety of binary joins have been considered, including *time-join* and *time-equijoin* (TE-join), *event-join* and *TE-outerjoin*, *contain-join*, *containsemijoin* and *intersect-join*,and *temporal natural join* . The various algorithms proposed for these joins have generally been extensions to nested loop or merge joins that exploit sort orders or local workspace, as well as partitioning-based joins, but incremental techniques have also been proposed.

More generally, these latter techniques are particularly attractive for implementing operators on relations capturing transaction time, because these relations retain complete records of their past states. Incremental techniques cache results of previous computations and at later times reuse these results, together with records of the updates to the underlying relations that have occurred since the results were cached, to efficiently compute the up-to-date results. With support for transaction time, the records of updates are already contained in the relations .

Next, time-varying aggregates are especially challenging. While there has been much work on the topic in the data warehousing context, only a few papers have considered the more general problem. Finally, *coalescing* is an important operation in temporal databases. Coalescing merges value-equivalent tuples with adjacent intervals (and possibly also value-equivalent tuples with intervals that overlap). This operation may be implemented by first sorting the argument relation on the explicit attribute values as well as the valid time. In a subsequent scan, the merging is then accomplished.

1.5.3. Indexing Temporal Data

A variety of conventional indexes have long been used to reduce the need to scan an entire relation to access a subset of its tuples, to support the conventional selection algebraic operator and temporal joins. Similarly, a number of temporal indexing strategies are available . Many of the indexes are based on B$^+$-trees, which index on values of a single key; most of the remainder are based on R-trees, which index on ranges (intervals) of multiple keys. The worst-case performance for most proposals has been evaluated in terms of total space required, updates per change, and several important types of queries. Most of this work is in the context of the selection operator. As also mentioned, indexes may be used to efficiently implement temporal joins and also coalescing and aggregates—this is an area of active investigation.

1.6. OUTLOOK

Although many important insights and results have been reported, many research challenges still remain in temporal database management, some of which are considered here. The lack of consideration of some of these challenges has reduced the potential of earlier results. In many cases, core concepts have been established, but it remains to be shown how they may be combined and applied, to simplify and automate the management of time-referenced data in practice.

There is a need for increased *legacy-awareness* in a number of areas within temporal databases. Research is needed that takes into account the reality that most databases are in fact legacy temporal databases and that the applications running on them are in fact legacy temporal database applications. In contrast, most research so far has assumed that applications will be designed using a new temporal data model, implemented using novel temporal query languages, and run on as yet nonexistent temporal DBMSs. In the short to medium term, this is an unrealistic assumption. Indeed, perhaps in part because of this and despite the obvious need in the marketplace, as yet no prominent commercial temporal relational DBMS exists.

The recent growth in database architectures, including the various types of middleware, prompts a need for increased *architecture-awareness*. Studies are needed that provide the concepts, approaches, and techniques necessary for third-party developers to efficiently and effectively implement temporal database technology while maximally exploiting available architectural infrastructure, as well as the functionality already offered by existing DBMSs. The resulting temporal DBMS architectures will provide a highly relevant alternative to the standard integrated architecture that is generally assumed. As a next step, research is needed on how to exploit existing and novel performance-improving advances, such as temporal algebraic operator implementations and indices, in these architectures. Finally, approaches for transitioning legacy applications will become increasingly sought after as temporal technology moves from research to practice.

The results on the *conceptual design* of temporal databases have potential for finding application in practice, but additional research is needed. When database designers actually understand the core temporal database concepts, perhaps most prominently valid and transaction time, they are able to design better databases using existing models and tools. A central challenge is to provide complete conceptual models, with associated design tools, that cover all aspects of designing a temporal database; empirical evaluation of these by real users is needed to provide essential insights. Reengineering of legacy databases is also a very relevant challenge in this context.

Concerning *performance*, more empirical studies are needed to compare temporal algebraic operator implementations, and to possibly suggest even more efficient implementations. Indexing techniques is an important aspect. While preliminary performance studies have been carried out for all or most of the proposed temporal indexes in isolation, there has been little effort to empirically compare them. More work is also needed on exploiting temporal indexes in algebraic operations other than selection. Finally, there has been little work in refining and validating cost models of temporal operators, or of developing and maintaining database statistics. For example, the cardinality (number of specific values) of an attribute is less useful than the average cardinality at a point in time. Another useful statistic is the number of *long-lived tuples*, the presence of which is the bane of some index structures and temporal algebraic operators.

A number of research areas that are either separate within temporal databases, overlap with this area, or take temporal databases as their point of departure also pose important challenges. Although not mentioned exhaustively in the coverage below, these areas are slated to offer ample challenges to those researchers concerned with effective and efficient implementation of advanced database functionality.

These may be extended to take into account prior history and temporal trends. For example, an absolute temperature reading in a nuclear power plant may be acceptable if it is part of a decreasing trend, but may signal a problem if it represents an increase. Some initial work has been reported in this area, but as yet there has been little integration

For example, many "*moving objects*" such as people, animals, cars, aircraft, and ships will be equipped with wireless devices (e.g., GPS) that track their positions and make these available for storage in databases. The continued advances in wireless communications hardware and software constitute powerful drivers for these types of applications. While we are already witnessing the appearance of traffic-related systems (e.g., for tracking taxi's or fleet management), a very substantial growth can be expected in the number, sizes, diffusion, and diversity of these applications.

Next, *multimedia* presentations and *virtual reality* scenarios are in fact special breeds of spatiotemporal databases. And, again, there are powerful enablers of these kind of spatiotemporal database applications. We are witnessing continued advances in data storage, processor, network, and user interaction technologies. In short, the integration of temporal databases with spatial databases offers exciting new challenges and promises to become an important research area in the future. To mention just one example challenge, no index seems to accommodate well the past, current, and anticipated future positions of moving objects such as those mentioned above.

The area of *temporal data mining* is a relatively new one, where exploration has only recently started in earnest. While extracting static associations from a mass of data is an important goal, more effort needs to be focused on associations that capture time-varying behavior, such as "when stock A goes up, stock B goes up within two weeks." The fairly recent focus among vendors, users, and researchers alike on *data warehousing* has brought new prominence to temporal databases. W. H. Inmon who is known as the founder of data warehousing cites time variance as one of four salient characteristics of a data warehouse , and there is general consensus that a data warehouse is likely to contain several years of time-referenced data. Being temporal, data warehouses are thus prime candidates to benefit from the advances in temporal databases. But cross-fertilization between temporal databases and data warehousing is largely absent. In fact, some of the original impetus for a separate data model and query language for data warehouses arose from a perceived lack of temporal support in the relational model and SQL. Few attempts have been made to exploit the advances in temporal databases in the context of data warehousing, although exceptions do exist. The special architecture of a data warehouse and the emphasis on supporting advanced query functionality, e.g., application-specific time-series analysis, bring novel challenges to temporal database researchers. Reconciling the differences between general relational database schemas and specialized star schemas would help enable users and developers to achieve an integrated view of an enterprise. Another active area of commercial products is that of *time-series* abstract data types (i.e., Informix's datablades, Oracle's cartridges). These data type extensions are highly useful for specialized applications, particularly in the financial sphere, but do not address the general problem of easy expression of the temporal constraints, queries, and modification. Rather, a more comprehensive approach along the lines of the extensions being considered for SQL/Temporal appears to be attractive.

1.8. CONCLUSION

Adopting a longer term and more abstract perspective, it is likely that new database management technologies and application areas will continue to emerge that provide 'temporal' challenges. Due to the ubiquity of time and its importance to most database management applications, and because built-in temporal support generally offers many benefits and is challenging to provide, research in the temporal aspects of new database management technologies will continue to flourish for existing as well as new application areas.

1.8. SUMMARY

This unit has briefly introduced you to temporal data management, emphasizing central concepts, surveying important results, and describing the challenges faced. This unit briefly summarizes the current state-of-the-art. A great amount of research has been conducted on temporal data models and query languages, which has shown itself to be an extraordinarily complex challenge with subtle issues. The (snapshot-based) semantics of standard temporal relational schemas are well understood, as are the implications for database design. The *Bitemporal Conceptual Data Model* is gaining acceptance as a desirable model in which to consider data semantics and as a good foundation for a temporal query language.

Many languages have been proposed for querying temporal databases, half of which have a formal basis. The numerous types of temporal queries are fairly well understood. The TSQL2 query language has consolidated many years of research results into a single, comprehensive language. New languages employ so-called *statement modifiers*, which offer a wholesale approach to giving temporal semantics to query language statements. Constructs from TSQL2, enhanced with *statement modifiers*, are being incorporated into the part of SQL3 called SQL/Temporal . The semantics of the time domain, including its structure, dimensionality, and indeterminacy, are quite well understood, and representational issues of timestamps have recently been resolved. Operations on timestamps are now well understood, and efficient implementations exist.

Temporal joins, aggregates, and coalescing are well understood, and efficient implementations exist. More than a dozen temporal index structures have been proposed, supporting valid time, transaction time, or both. A handful of prototype temporal DBMS implementations have been developed.

1.9. TUTOR-MARKED ASSIGNMENT (TMA)

1. What is a multimedia database?
2. Enumerate the limitations of conventional database.
3. Mention other database models you know and discuss their limitations.

1.10. FURTHER READINGS

Date, C. [1995] An Introduction Database System, 7th ed; Addison-Wesly 2000.

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[1986]. "Database System Concepts, McGraw–Hill, 1986.

V. J. Tsotras and X. S. Wang. Temporal Databases. *Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons, 1999.

X. S. Wang, C. Bettini, A. Brodsky, and S. Jajodia. Logical Design for Temporal Databases with Multiple Granularities. *ACM Transactions on Database Systems*, 22(2):115–171, June 1997.

Zobel, J., Moffat, A., and Sacks–Davis, R., [1992]. "An Efficient Indexing Technique for Full – Text Database Systems," in VLDB [1992.

MODULE 4 : EMERGING TRENDS & EXAMPLES OF DATABASE ARCHITECTURE

UNIT 2: THE MAJOR APPLICATION DOMAINS

2.0. INTRODUCTION

So far, we have discussed a variety of issues related to the modeling, design, and function of databases as well as to the internal representation issues related to databases management systems. Applications in domains such as Multimedia, Geographical Information Systems, and digital libraries demand a completely different set of requirements in terms of the underlying database models. The conventional relational database model is no longer appropriate for these types of data. Furthermore the volume of data is typically significantly larger than in classical database systems. Finally, indexing, retrieving and analyzing these data types require specialized functionality not available in conventional database systems. The module will cover the requirements of these emerging database technologies and the major application domains, such as multimedia databases, spatial databases, temporal databases and biological/genome databases, their underlying technologies, data models and languages.

2.1. OBJECTIVES

To gain insight into emerging database technologies and their application domains.

To appreciate the role of Genetics in information technology.

2.2. DATABASE ON THE WORLD WIDE WEB

In Web technology, a basic client-servers in publicly accessible shared files encoded using Hyper Text Markup Language (HTML). A number of tolls enable users to create Web pages formatted with HTML tags, freely mixed with multimedia content from graphics to audio and even to video. A page has many intersperse hyperlinks-literally a link that enables user to “browse” or move from one page to another across the internet. This ability has given a tremendous power to end users in searching and navigating related information-often across different containments. Information on the web is organized according to a Uniform Resource Locator (URL)-something similar to an address that provides the complete pathname of a file. The pathname consists of a string of machine and directory names separated slashes and ends in a filename.

2.3.GISAPPLICATION

It is possible to divide GIS s into categories: (1) cartographic applications (2) digital terrain modeling applications, and (3) geographic applications

In cartographic and terrain modeling applications, variations in spatial attributes are captured- for example, soil characteristics, crop density, and air quality. In geographic objects application, objects of interest are identified from a physical domain – for example, power plants, electoral districts, property parcels, product distribution districts, and city landmarks. These power consumption, voting patterns, property sales volumes, product sales volume and traffic density. The first two categories of GIS applications require a field-based representation, whereas the third category requires and object-based one. The cartographic approach involves special functions that can include the overlapping of layers of maps to combine attribute data that will allow, for example, the measuring or distances in three-dimensional space and the reclassification of data on the map. Digital terrain modeling requires a digital representation of parts of earth's surface using land elevations at sample points that are connected to yield a surfaces model such as a three-dimensional net (connected observed points as well as visualization. In object-based geographic applications, additional spatial functions are needed to deal with data related to roads, physical pipelines, communication cables, power lines, and such. For example, for a given region, comparable maps can be used for comparison at various points of time to show changes in certain.

2.3.1. SPECIFIC GIS DATA OPERATION

GIS Applications are conducted through the use of special operators such as the following:

Interpolation: The process derives elevation data for points at which no samples have been taken. It includes computation at single points, computation for a rectangular grid or along a contour, and so forth. Most interpolation methods are based on triangulation that uses the TIN method for interpolating elevation s inside the triangle based on those of its vertices.

Interpretation Digital terrain modeling involves the interpretation of operations on terrain data such as editing, smoothing, reducing details, and enhancing. Additional operations involve patching or zipping the borders of triangles (in TIN data), and merging, which implies combining overlapping models and resolving conflicts among attribute data. Conversions among grid models, contour models, and TIN data are involved in the interpretation of the terrain.

Proximity analysis: Several classes of proximity analysis include computations of "zones of interest" around objects, such as the determination of a buffer around a car on a highway. Shortest path algorithms using 2D or 3D information is an important class of proximity analysis.

Raster image processing: This process can be divided into two categories map algebra, which is used to integrate geographic features on different map layers to produce new maps algebraically, and digital image analysis, which deals with analysis of a digital image for features such as edge detection and object detection. Detecting roads in stalling image of city is an example of the latter.

Analysis of networks: Networks occur in GIS as many contexts that must be analyzed and may be subjected to segmentations overlays, and so on. Network overlay refers to a type of spatial join database for example, accident locations to yield, in this case a profile of high-accident roadways.

Other Database Functionality: The functionality of a GIS database is also subject to other considerations.

Extensibility: GISs are required to be extensible to accommodate a variety of constantly evolving applications and corresponding data types. If a standard DBMS is used, it must allow a core set of data types with a provision for defining additional types and methods for those types.
Data quality control: A sin many other applications, quality of source data is of paramount importance for providing accurate result to queries. This problems is particularly significant in the GIS content because of the variety of data, sources and measurement techniques involved and the absolute accuracy expected by application users.

Visualization: A crucial function in GISs related to visualization the graphical distributes to go with it. Major visualization techniques includes contouring through the uses of isoclines, spatial units of lines or arcs of equal attributes values: hill shading an illumination method used for qualitative relief depiction using perspective projection methods from computer graphics. These techniques impose cartographic data and other three-dimensional objects on terrain data providing animated scene renderings such as those in flight simulations and animated movies.

Such requirements clearly illustrate that standard RDBMSs or ODBMSs do not meet the special needs of GIS. It is therefore necessary to design systems that support the vector and raster representations and the spatial functionality as well as the required RDBMS functionality in the INFO part of the system, is briefly discussed in the subsection that follows. More systems are likely to be designed in the future to work with relational or object databases that will contain some of the spatial and most of the conceptual information.

2.3.2 An Example of A GIS: Arc-Info

ARC/INFO-A popular GIS LAUCHED IN 1981 BY environmental system research institute (ESRI)- uses they are node model to store spatial data. A geographic layer - called **coverage** in ARC / INFO-consists of three primitive. nodes(points), arcs (similar to lines), topological information. An arc has a start node and an end node (and it therefore has direction too). In addition, the polygons to the left and the right of the shape of arc are also stored along with each arc. The database managed by the INFO RDBMS thus consists of three required tables: (1) node attribute table (NAT), (2)are attribute table(ATT), and(3) polygon attribute table(PAT).Additional information can be stored in separate tables and joined with any of these three tables.

The NAT contains an internal ID for the node, a user-specified ID, the coordinates of the node, and any other information associated with that node, and any other information associated with that node (e.g. names of the intersection). The AAT contains an internal ID for the arc, a user-pacified ID, the internal ID of the start and end notes, the internal ID of polygons to the left and the tight, a series of coordinates of shape points (if any), the length of the arc, any other data associated with arc (e.g., the name of the road the arc represent). The PAT contains an internal ID for the polygon, a user-specified ID, the area of the polygon, the perimeter of the polygon, and any other associated data (e.g. name of the country the polygon represent).

Typical spatial queries are related to adjacency, containment, and connectivity. The arc node model has enough information to satisfy all three types of queries, but the RDBMS is not ideally suited for this type of querying. A simple example will highlight the number of times a relational database has to be queried to extract adjacency information. Assume that we are trying to determine whether two polygons, A and B, are adjacent to each other. We would have to exhaustively look at the ATT to determine whether there is an edge that has *A* on one side and *B* on the other. The search cannot be limited to the edges of either polygon as we do not explicitly store all the acts that make a polygon in the PAT. Storing all the arcs sin the PAT would redundant because all the information is already there in the AAT.

ESRI has released Arc Storm (Arc Store Management) which allows multiple users to use the same GIS, handles distributed database, and integrates with other commercial RDBMS s like ORCLE, INFORMIX, and SYBASE. While it offers many performance and function advantage over ARC/INFOR, it is essentially an RDBMS embedded within a GIS.

2.4. GENOME DATA MANAGEMENT

2.4.1 BIOLOGICAL SCIENCES AND GENETICS

The biologic sciences encompass an enormous variety of information. Environment scence gives us a view of how species live and interacting world filled with natural phenomena. Biology and ecology study particular species. Anatomy focuses on the overall structure of an organism, documenting the physical aspects of individual bodies. Traditional medicine and physiology break the organism into systems and tissues and strive to collect information on the workings of these systems and the organism as a whole. Histology and cell biology delve into the tissue and cellular levels and provide knowledge about the inner structure and function of the cell. This wealth of information that has only recently become a major application of database technology.

Genetic has emerged as an ideal field for the application of information technology. In a broad sense, it can be thought of as the construction of models based on information about genes-which can be defined as basic units of heredity-and populations and the seeking out of relationships in that information. The study of genetics can be divided into three branches: (1)

Mendelian genetics, (2) molecular genetics, and (3) population genetics. Mendelian genetics is the study of the transmission of traits between generations. Molecular generics are the study of the chemical structure and function of genes at the molecular genetics is the study of how genetic information varies across population of organism.

Molecular genetics provides a more detailed look at genetic information by allowing researchers to examine the composition, structure and function of genes. The origins of molecular genetics can be traced to two important discoveries. The first occurred in 1869 when Friedrich Melcher discovered unclean and its primary component, deoxyribonucleic acid (DNA). Subsequent Research DNA and a related compound, ribonucleic acid (RNA), were found to be composed of nucleotides (a sugar a phosphate, and a base, which combined to form nucleic acid) linked into long polymers via the sugar and phosphate. The second discovery was the demonstration in 1944 by Oswald Avery that DNA was is compose chains of nucleic arranged information. Genes were thus shown to be composed of chins of nucleic acids arranged linearly on chromosomes and to serve three primary functions: (1) replicating genetic information between generations, (2) providing blueprints for the creation of polypeptides, and (3) accumulating changes thereby allowing evolution to occur.

2.4.2 CHARACTERISTICS OF BIOLOGICAL DATA

Biological data exhibits many special characteristics that make management of biological information a particularly challenging problem. We will thus begin by summarizing the characteristics related to biological information and focusing on a multidisciplinary field called **bioinformatics** that has emerged, with graduate degree programs now in plaoce in several universities. Bioinformatics address information management of genetic information with special emphasis on DNA sequence analysis. It needs to be broadened into a wider scope to harness all types of biological information-its modeling, storage, retrieval, and management.

Characteristics 1: Biological data is highly complex when compared with most other domains on applications. Definitions of such data must thus be able to represent a complex substructure of data as well as relationships and to ensure that no information is lost during biological data modeling. The structure of biological data often provides an additional context for interpretation of the information. Biological information systems must be able to represent any level of complexity in any data sheme relationship or shema substructure-not just hierarchical, binary , or table data. As an example MITOMAP is a database documenting the human mitochondral genome. This single genome is a small, circular encoding messenger RNA, ribosomal RNA, and transfer RNA; 1,000 known population variants: over 60 known disease association; and a limited set of knowlegde on the complex molecular interactions of the biochemical energy producing pathway of oxidant phosphorylation. As might be expected, its management has encoutered a large number of problems; we have been unable to use the traditional RDBMS traditional RDBMS or ODBMS **approaches** to capture all aspects of the data.

Characteristics 2: The amount and range of variability in data is high. Hence, biological system must be flexible in handling data types and values. With such a wide range of possible data values**,** placing constraints on data types must be limited since this many exclude unexpected values-e.g. outlier values-that are particularly common in the biological domain. Exclusion of such values results in a loss of information. In addition, frequent expectations of biological data structures may require a choice of data types to available for given piece of data.

Characteristics 3: Schemas in biological databases change at a rapid pace. Hence, for improved information flow between generations or releases of database, shema evolution and data object migration must be supported. The ability to extend schema, a frequent occurrence in the biological setting, is unsupported in most relational and object database systems. Presently systems such as Genbank release the entire database with new schemas once or twice a year rather than incrementally change the system as new schemas once or twice a year rather than incrementally changing the system as changes become neccesary. Such an evolutionary database would provide a timely and orderly mechanism of following change to individual data entities in biological database over time. This sort tracking is important for biological researchers to be able to access and reproduce previous results.

Characteristics 4: Representations of the same data by different biologists will likely be different (even when using the same system). Hence, mechanisms for "aligning" different biological schemas or different versions of schemas should be supported. Given the complexity of biological data, there are a multitude of modeling any given entity, with the results often reflecting the particular focus of the scientist. While two individuals may produce different data models will likely have numerous points in common. In such situations, it would be useful to biological investigators to be able to run queries across these common pints. By linking data elements in a network of schema, this could be accomplished.

Characteristics 5: Most users of biological data do not require write access to the database; read-only access is adequate. Write access is limited to privileged users called curators. For example, the database created as part of the MITOMAP project has no average more than 15,000 users per month on the Internet. There are fewer than twenty noncurator generated submission to MITOMAP every month. In other words, the number of users requiring write access is small. Users generate a wide variety of read-access pattern into the database, but these patterns are not the same as those seen in traditonal relational database. User requested ad hoc searches demand indexing of often unexpected combinations of data instance classes.

Characteristics 6: Most biologists are into likely to have any knowlegde of the internal structure of the database or about schema design. Biological database interfaces should sideplay information to users in a manner that is applicableto the problem they are trying to address and that reflects the underlying data structure. Biological users usually know which data they require, but they have no technical knowledge of the data structure or how a DBMS represents the data. They rely on technical user to provide them with view into the database. Relational schemas fail to provide cues or any intuitive information to the user regarding the meaning of their schema. Web interface in particular often provide present search interfaces, which may limit access into the database. However, if these interfaceses are generated directly from database structures, they are likely tp produce a wider posible range of access, although they may not guarrantee usablity.

Chracteristic 7: The context of data meaning for its use in biological applications. Hence, context must be maintained and conveyed to the user when appropriate. In addition, it should be posible to integrate as many context as possible to maximize theintepretation of a biological data value. Isolated values are of less use in bological systems. For example, the sequence of a DNA strand is not particularly useful without additional information describing its organisation, funtion, and such a single neuclotide on a DNA strands, for example seen in context with no disease causing strands could be seen as a causative element for sickle cell anaemia.

Characteristic 8: Defining and representing complex queries is extremly important to the edge of the data structure (see Characteristics 6), average users cannot construct a complex query across data sets on their own. Thus, in order to be truly useful, systems must provide some tools for building these queries. As mentioned previously, many systems provide predefined query templates.

Characteristic 9: User of biological information often require access to”old” values of the data-particularly when verifying previously reported results. Hence, changes to the values of data in the database must be surpported through a system archives. Access to the both the most recent version of a data value and its previous version are important in the biological domain. Investigators consistently want the most up-to-data, but they must also be able to reconstruct previous work and revaluate prior and current information. Consequently, values that are about to be updated in a biological database cannot simply be thrown away.

All of these characteristics clearly point to the fact that today’s DBMS do not fully cater to the requirements of complex biological data. A new direction in database management system is necessary.

2.4.3 THE HUMAN GENOME PROJECT AND EXISTING BIOLOGICAL DATABASES

The term genome is defined as the total genetic information that can be obtained about an entity. The **human genome,** for example generally refers to the complete set of genes required to create a human being-estimated to be between 100,000 and 300,000 genes spread over 23 pairs of chromosomes, with an estimated 3-4 billion nucleotides. The goal of the Human Genome Project (HGP) is to obtain the complete sequence-the ordering of the base-of those nucleotides. At present only 8,000 genes have been identified and and less than 10 percent of the human genome has been sequenced. However, the entire sequence is expected to the completed by the year 2002. In isolation, the human DNA sequence is not particularly useful. The sequence can however be combined with other data and useful and used as a powerful tool to help address questions in genetics, biochemistry medicine,anthropology, and agriculture. In the existing genome databases, the focus has been on’’curating” (or collecting with some initial scrunity and quality check) and classifying information about genome, numerous organism such as E.coli, Dropsophila, and C.elegan have been investigated. We will briefly discuss some of the existing database systems that are supporting or have grown out of the Human Genome Project.

Benbank. The preminent DNA requence database in the world today is Genback, maintained by the National Center for Biotechnology Information (NCBI) of the National Library of Medicine (NLM). It was established in 1978 as a central repository for DNA sequence data. Since then it has expanded somewhat in scope to include expressed sequence tag data, protein sequence data, three-dimensional protein structure, taxonomy and links to the literature (MEDLINE). Its latest release contains over 602,000,000 nucleotide base of more than 920,000 sequences form over 16,000 species with roughly ten new organism being added each day. The database size has double approximately every eighteen months for five years.

While it is a complex, comprehensive database, the scope of its coverage is focused on human sequences and links to the literature. Other limited data sources (e.g. three-dimensional structure and OMM, disscused PDB databases and redesigned the structure of the Genbank system to accommodate these new data sets. The system is maintained as a combination of flat files, relational databases, and files containing **Abstract Syntax Notation One (ASN.1)-**a syntax for defining data structure developed for the telecommunications industry. Each Genback entry is assigned a unique identifier by the UCBI. Updates are assigned a new identifier, with the identifier of the original entity remaining unchanged for archival purposes. Older refrence to an entity thus do not inadvertently indicate a new and possible inapproapriate values. The most current concepts also receive a second set of unique identifiers (UIDs), which marks the most up-to-date form of a concepts while allowing older version to be accessed via there original identifier.

The average user of the database is not able to access the structure of the data directly for querying or other functions, although complete snapshots of the database are available for export in a number of formats, including ASN.1 the query mechanism provide a via the enter application (or its world wide web version), which allows keywords, sequences, and Genbank UDID searching through a static interface.

The Genome Database (GDB). Created in 1989, the genome database (GDB) is a catalog human genome mapping data, a process that assocciates a piece of information with a particular location location on the human genome. The degree of precision of this location on the mapdepends upon the source of the data, but it is usually not at the level of Individual necleotide bases GDB data includes data describing primarly map information (distance and confidence limits), and Polymerase China Reaction (PCR) probe data (experimental conditions,PCR primer, and reagents used). More recently efforts have been made to add data on mutation linked to genetic lock, cell lines used in experiments, DNA probe libraries, and some limited polymorphism and population data.

The GDB systems is built around SYBASE, a commercial relational DBMS, and its data are modeled using standard Entity-Relationship techniques. The implementers of GDB have noted difficulties in using this model to capture more than simple map and probe data. In order to improve data integrity and to simplify the programming for the application writer, GDB distributes a Database Access Toolkits. However, most users use a web interfaces to search the ten interlinked data managers. Each manager keeps tracks of the link (relationships) for one of the ten tables within the GDB system. As with Genbank, users are given only a very high-level veiw of the data at the time of searching and thus cannot easily make use of any knowlegde gleaned from the structure of the GDB tables. Search methods are most useful when users are simply looking for an index into map or probe data.Expoloratory ad hoc searching of the database is not encouraged by present interfaces. Integration of the database structures of GDB and OMM was never fully established.

2.5. DIGITAL LIBRARIES

Digital libraries are an important and active research area. Conceptually, a digital library is an analog of a traditional library-a large collection of information soucres in various media-coupled with the advantages of traditional technologies. However, digital libraries differ from their traditional counter-parts in significant ways: storage is digital, remote access is quick and easy, and materials are copied from a master version. Furthermore, keeping extra copies on hand is easy and is not hampered by budget and storage restrictions, which are major problems in traditional libraries. Thus, digital technologies overcome many of the physical and economic limitations of traditional libraries.

The introduction to the April 1995 communications of the ACM special issues on digital libraries describes them grandly as the "opportunity…to fulfill the age-old dread of every human being: gaining ready access to humanity's store of information. Unlike the related data in a database, a digital library encompasses as multitude of sources, mainly unrelated. Logically, databases can be components of digital libraries.

The Digital Library Initiative (DLI), jointly fuced by SNF, DARPA, and NASA, has been a major accelerator of the development of digital libraries. This initiative provided significant funding to six major projects at six universities in its first phase covering a broad spectrum of enabling technologies (as will be discussed below). The initiative's web page (sedli.grainger.uiuc.edu/national.htm) define its focus as "dramatically advance the means to collect, store, and organize information in digital forms, and make it available for searching, retrieval, and processing via communication networks-all in user-friendly ways.

The manitude of these data collections as well as their diversity and multiple formats provides challenges on a new scale. The future progression of the development of digital libraries is likely to move from the present technology of retrieval via the internet, though Net searches of ondexed information in reprositiories, to a time of information correlation and analysis by intellegent networks. Techniques for collecting information, storing it, and organizing it to support informational requirements learned in decades of design and implementation of database will provide the baseline for development of approaches appropriate for digital libraries. Search, retrieval, and processing of many formss of digital information.

2.6. CONCLUSION

Applications in domains such as Multimedia, Geographical Information Systems, and digital libraries demand a completely different set of requirements in terms of the underlying database models which conventional relational database can no longer handle. The conventional relational database model is no longer appropriate for these types of data. Furthermore the volume of data is typically significantly larger than in classical database systems. Finally, indexing, retrieving and analyzing these data types require specialized functionality not available in conventional database systems. Hence, a new direction, such as described above, in DBMS is necessary.

2.7. SUMMARY

The requirements of these emerging database technologies and the major application domains, such as multimedia databases, spatial databases, temporal databases and biological/genome databases, their underlying technologies, data models and languages have been covered in this unit.

2.8. FURTHER READINGS

G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering. In *Proceedings of the 16th International Conference on Data Engineering*, San Diego, California, February/March 2000, to appear.

J. Bair, M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Notions of Upward Compatibility of Temporal Query Languages. *Wirtschaftsinformatik*, 39(1):25–34, February 1997.

J. Clifford and A. Tuzhilin (eds.). *Recent Advances in Temporal Databases: Proceedings of the International Workshop on Temporal Databases*. Workshops in Computing Series. Springer-Verlag 1995.

J. Wijsen. Temporal FDs on Complex Objects. *ACM Transactions on Database Systems*, 24(1): 127–176, March 1999.

K. Torp, C. S. Jensen, and R. T. Snodgrass. Stratum Approaches to Temporal DBMS Implementation. In *Proceedings of the 1998 International Database Engineering and Applications Symposium*, pp. 4–13, Cardiff, Wales, UK, July 1998.

M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag 1999.

O. Etzion, S. Jajodia, and S. Sripada (eds.). *Temporal Databases: Research and Practice*. LNCS 1399, Springer-Verlag 1998.

R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers 2000.

Rob, P., and Coronel, C. [2000] *Database Systems, Design, Implementation, and Management, 4th ed., Course Technology, 2000.*

Silberwschatz., A., and Korth, H.,[2000]. "Database System Concepts", 4th ed., McGraw – hill, 2000.

Zobel, J., Moffat, A., and Sacks–Davis, R., [1992]. "An Efficient Indexing Technique for Full – Text Database Systems," in VLDB [1992.

www.w3schools.com

MODULE 4: EMERGING TRENDS AND EXAMPLE OF DBMS

UNIT 3: POSTGRESQL

3.0 INTRODUCTION

PostgreSQL is the most advanced open source database server. In this section, you will learn about database operation on the open source software, alongside the history of PostgreSQL.

Three basic office productivity applications exist: word processors, spreadsheets, and databases. *Word processors* produce text documents critical to any business. *Spreadsheets* are used for financial calculations and analysis. *Databases* are used primarily for data storage and retrieval. You can use a word processor or spreadsheet to store small amounts of data. However, with large volumes of data or data that must be retrieved and updated frequently, databases are the best choice. Databases allow orderly data storage, rapid data retrieval, and complex data analysis.

3.1 OBJECTIVE

To understand the meaning and functionalities of the PostgreSQL

3.2 What is POSTGRESQL?

POSTGRESQL is an enhancement of the Postgres management system, a next-generation DBMS research prototype. While POSTGRESQL retains the powerful data model and rich data types of Postgres, it replaces the POSTQUEL query language with an extended subset of SQL. POSTGRESQL is free and the complete source is available.

POSTGRESQL development is performed by a team of Internet developers who all subscribe to the POSTGRESQL development mailing list. This team is responsible for all development of POSTGRESQL.

The authors of POSTGRESQL 1.01 were Andrew Yu and Jolly Chen. Many others

have contributed to the porting, testing, debugging, and enhancement of the code.

The original Postgres code, from which POSTGRESQL is derived, was the effort of many graduate students, undergraduate students, and staff programmers working under the direction of Professor Michael Stonebraker at the University of California, Berkeley.
The original name of the software at Berkeley was Postgres. When SQL functionality was added in 1995, its name was changed to Postgres95. The name was changed at the end of 1996 to POSTGRESQL.

It is pronounced Post-Gres-Q-L.

3.3 POSTGRESQL and other DBMS

There are several ways of measuring software: features, performance, reliability, support, and price.

3.3.1 Functions

Functions allow blocks of code to be executed by the server. Although these blocks can be written in SQL, the lack of basic programming operations which existed prior to version 8.4, such as branching and looping, has driven the adoption of other languages inside of functions. Some of the languages can even execute inside of triggers. Functions in PostgreSQL can be written in the following languages:

A built-in language called PL/pgSQL resembles Oracle's procedural language PL/SQL.
Scripting languages are supported through PL/Lua, PL/LOLCODE, PL/Perl, plPHP, PL/Python, PL/Ruby, PL/sh, PL/Tcl and PL/Scheme.
Compiled languages C, C++, or Java (via PL/Java).
The statistical language R through PL/R.

PostgreSQL supports row-returning functions, where the output of the function is a set of values which can be treated much like a table within queries. Custom aggregates and window functions can also be defined.

Functions can be defined to execute with the privileges of either the caller or the user who defined the function. Functions are sometimes referred to as stored procedures, although there is a slight technical distinction between the two.

3.3.2 Indexes

PostgreSQL includes built-in support for B+-tree, hash, GiST and GiN indexes. In addition, user-defined index methods can be created, although this is quite an involved process. Indexes in PostgreSQL also support the following features:

PostgreSQL is capable of scanning indexes backwards when needed; a separate index is never needed to support ORDER BY field DESC.
Expression indexes can be created with an index of the result of an expression or function, instead of simply the value of a column.
Partial indexes, which only index part of a table, can be created by adding a WHERE clause to the end of the CREATE INDEX statement. This allows a smaller index to be created.
The planner is capable of using multiple indexes together to satisfy complex queries, using temporary in-memory bitmap index operations.

3.3.3 Triggers

Triggers are events triggered by the action of SQL DML statements. For example, an INSERT statement might activate a trigger that checked if the values of the statement were valid. Most triggers are only activated by either INSERT or UPDATE statements.

Triggers are fully supported and can be attached to tables but not to views. Views can have rules, though. Multiple triggers are fired in alphabetical order. In addition to calling functions written in the native PL/PgSQL, triggers can also invoke functions written in other languages like PL/Perl.

3.3.4 MVCC

PostgreSQL manages concurrency through a system known as Multi-Version Concurrency Control (MVCC), which gives each user a "snapshot" of the database, allowing changes to be made without being visible to other users until a transaction is committed. This largely eliminates the need for read locks, and ensures the database maintains the ACID principles in an efficient manner.

3.3.5 Rules

Rules allow the "query tree" of an incoming query to be rewritten. One common usage is to implement updatable views.

3.3.6 Data types

A wide variety of native data types are supported, including:

Variable length arrays (including text and composite types) up to 1GB in total storage size.
Arbitrary precision numerics
Geometric primitives
IPv4 and IPv6 addresses
CIDR blocks and MAC addresses
XML supporting Xpath queries (as of 8.3)
UUID (as of 8.3)

In addition, users can create their own data types which can usually be made fully indexable via PostgreSQL's GiST infrastructure. Examples of these are the geographic information system (GIS) data types from the PostGIS project for PostgreSQL.

3.3.7 User-defined objects

New types of almost all objects inside the database can be created, including:

Casts
Conversions
Data types
Domains
Functions, including aggregate functions
Indexes
Operators (existing ones can be overloaded)
Procedural languages

3,3,8 Inheritance

Tables can be set to inherit their characteristics from a "parent" table. Data in child tables will appear to exist in the parent tables, unless data is selected from the parent table using the ONLY keyword, i.e. select * from ONLY PARENT_TABLE. Adding a column in the parent table will cause that column to appear in the child table.

Inheritance can be used to implement table partitioning, using either triggers or rules to direct inserts to the parent table into the proper child tables.

This feature is not fully supported yet—in particular, table constraints are not currently inheritable. As of the 8.4 release, all check constraints and not-null constraints on a parent table are automatically inherited by its children. Other types of constraints (unique, primary key, and foreign key constraints) are not inherited.

Inheritance provides a way to map the features of generalization hierarchies depicted in Entity Relationship Diagrams (ERD) directly into the PostgreSQL database.

3.3.9 Performance

POSTGRESQL runs in two modes. Normal fsync mode flushes every completed transaction to disk, guaranteeing that if the OS crashes or loses power in the next few seconds, all your data is safely stored on disk. In this mode, we are slower than most commercial databases, partly because few of them do such conservative flushing to disk in their default modes. In no-fsync mode, we are usually faster than commercial databases, though in this mode, an OS crash could cause data

corruption. We are working to provide an intermediate mode that suffers less

performance overhead than full fsync mode, and will allow data integrity within 30 seconds of an OS crash.

In comparison to MySQL or leaner database systems, we are slower on inserts/updates because we have transaction overhead. Of course, MySQL doesn't have any of the features mentioned in the Features section above. We are built for flexibility and features, though we continue to improve performance through profiling and source code analysis. There is an interesting Web page comparing

POSTGRESQL to MySQL at http://openacs.org/why-not-mysql.html.

We handle each user connection by creating a Unix process. Backend processes share data buffers and locking information. With multiple CPU's, multiple backends can easily run on different CPU's.

3.3.10 Reliability

We realize that a DBMS must be reliable, or it is worthless. We strive to release well-tested, stable code that has a minimum of bugs. Each release has at least one month of beta testing, and our release history shows that we can provide stable, solid releases that are ready for production use. We believe we compare favorably to

other database software in this area.

3.3.11 Support

Our mailing list provides a large group of developers and users to help resolve any problems encountered. While we can not guarantee a fix, commercial DBMS's don't always supply a fix either. Direct access to developers, the user community, manuals, and the source code often make POSTGRESQL support superior to other DBMS's. There is commercial per-incident support available for those who need it. (See support FAQ item.)

3.3.12 Price

We are free for all use, both commercial and non-commercial. You can add our code to your product with no limitations, except those outlined in our BSD-style license stated above.

3.3.13 psql

The primary front-end for PostgreSQL is the psql command-line program, which can be used to enter SQL queries directly, or execute them from a file. In addition, psql provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks; for example tab completion of object names and SQL syntax.

3.3.14 pgAdmin

pgAdmin is a free and open source graphical front-end administration tool for PostgreSQL, which is supported on most popular computer platforms. The program is available in more than a dozen languages, and is released under the Artistic License. The first prototype, named pgManager, was written for PostgreSQL 6.3.2 from 1998. The stable release (named pgAdmin II) was first released on 16 January 2002. The current version is pgAdmin III, which is released under the Artistic License.

3.3.15 phpPgAdmin

phpPgAdmin is a web-based administration tool for PostgreSQL written in PHP and based on the popular phpMyAdmin interface originally written for MySQL administration.

3.3.16 Proprietary

A number of companies offer proprietary tools for PostgreSQL. They often consist of a universal core that is adapted for various specific database products. These tools mostly share the administration features with the open source tools but offer improvements in data modelling, im/exporting or reporting. Among them are Navicat, SQL Maestro as well as pure data modeling tools like DeZign for Databases or ModelRight.

3.3.17 Benchmarks

Many informal performance studies of PostgreSQL have been done but the first industry-standard and peer-validated benchmark was completed in June 2007 using the Sun Java System Application Server (proprietary version of GlassFish) 9.0 Platform Edition, UltraSPARC T1 based Sun Fire server and Postgres 8.2. This result of 778.14 SPECjAppServer2004 JOPS@Standard compares favourably with the 874 JOPS@Standard with Oracle 10 on an Itanium based HP-UX system.

In August 2007, Sun submitted an improved benchmark score of 813.73 SPECjAppServer2004 JOPS@Standard. With the system under test at a reduced price, the price/performance improved from $US 84.98/JOPS to $US 0.57/JOPS.

3.3.18 Prominent users

Yahoo! for web user behavioural analysis, storing 2 petabytes and claimed to be the largest data warehouse using a heavily modified version of PostgreSQL with an entirely different column-based storage engine and different query processing layer. While for performance, storage, and query purposes the database bears little resemblance to PostgreSQL, The front-end maintains compatibility so that Yahoo can use many off-the-shelf tools already written to interact with PostgreSQL.
OpenStreetMap, a collaborative project to create a free editable map of the world.
Afilias, domain registries for .org, .info and others.
Sony Online multiplayer online games.
BASF, shopping platform for their agribusiness portal.
hi5.com social networking portal.
Skype VoIP application, central business databases.
Sun xVM, Sun's virtualization and datacenter automation suite
Evergreen, an open source integrated library system providing an Online Public Access Catalogue and cataloguing, management, and other functions for hundreds of libraries in the United States, Canada, and elsewhere

NextBus, provider of arrival-time prediction and GPS tracking systems for public transportation
Snooth, a wine database, featuring millions of reviews and hundreds of thousands of wines.
The Weather Channel Cable television network.

3.4 OPEN SOURCE SOFTWARE

POSTGRESQL is *open source software.* The term ``open source software'' often confuses people. With commercial software, a company hires programmers, develops a product, and sells it to users. With Internet communication, however, new possibilities exist. Open source software has no company. Instead, capable programmers with interest and some free time get together via the Internet and exchange ideas. Someone writes a program and puts it in a place everyone can access. Other programmers join and make changes. When the program is sufficiently functional, the developers advertise the program's availability to other Internet users. Users find bugs and missing features and report them back to the developers, who, in turn, enhance the program.

It sounds like an unworkable cycle, but in fact it has several advantages:

A company structure is not required, so there are no overhead and no economic restrictions.
Program development is not limited to a hired programming staff, but taps the capabilities and experience of a large pool of Internet programmers.
User feedback is facilitated, allowing program testing by a large number of users in a short period of time.
Program enhancements can be rapidly distributed to users.

3.5 CONCLUSION

Postgresql has been seen to be a very resourceful application for the management of databases.

3.6 SUMMARY

This unit has explored the long history of POSTGRESQL, starting with its roots in university research. POSTGRESQL would not have achieved its success without the Internet. The ability to communicate with people around the world has allowed a community of unpaid developers to enhance and support software that rivals commercial database offerings. By allowing everyone to see the source code and contribute to its ongoing development, POSTGRESQL continues to improve every day.

3.7 TUTOR MARKED ASSIGNMENT (TMA)

What is the primary frontend for PostgreSQL?

Explain the term open source software

3.8 FURTHER READING

Douglas, Korry (August 5, 2005). PostgreSQL (Second ed.). Sams. pp. 664. ISBN0672327562.

http://www.informit.com/store/product.aspx?isbn=0672327562.

Gilmore, W. Jason; Treat, Robert (February 27, 2006). BeginningPHPandPostgreSQL8:From NovicetoProfessional. Apress. pp. 896. ISBN1590595475. http://www.apress.com/book/view/1590595475.

Matthew, Neil; Stones, Richard (April 6, 2005). BeginningDatabaseswithPostgreSQL (Second ed.). Apress. pp. 664. ISBN1590594789. http://www.apress.com/book/view/9781590594780.

Worsley, John C.; Drake, Joshua D. (January 2002). Practical PostgreSQL. O'Reilly Media. pp. 636. ISBN 1565928466. http://oreilly.com/catalog/9781565928466/.

i