COURSE GUIDE

CIT 208
INFORMATION SYSTEMS

Course Team Dr. A. S. Sodiya (Developer/Writer) - UAA
Prof. Afolabi Adebanjo (Programme Leader) - NOUN
A.A. Afolorunso (Coordinator) - NOUN

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
University Village
Plot 91, Cadastral Zone
Nnamdi Azikiwe Expressway
Jabi, Abuja

Lagos Office
14/16 Ahmadu Bello Way
Victoria Island, Lagos

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Printed 2013, 2017

ISBN: 978-058-392-0

CONTENTS **PAGE**

INTRODUCTION

CIT208 – Information Systems is a-three [3] credit unit course of fifteen units. It deals with the various forms of information technology used by people to accomplish specific organisational or individual objectives.

It also gives an insight into computer technology and data communications technology which are the specific technologies that collectively sum up into information technology as a whole. Since information comprises of data, this course takes you through the various ways through which data is created and manipulated to produce relevant information needed. It also deals with the various advances in computer hardware, software, and networking technologies which have spurred an evolution in the structure, design, and use of corporate information systems.

This course is divided into three modules. The first module deals with the basic introduction to the concept of Information Systems, SQL and Database Programming with JDBC.

The second module deals with Conceptual modeling, Schema design, Functional dependency, Regular expression and Relational algebra.

The third module deals with Web services, XML and database recovery. This Course Guide gives you a brief overview of the course content, course duration, and course materials.

WHAT YOU WILL LEARN IN THIS COURSE

The main purpose of this course is to provide the necessary tools for designing and managing Information Systems. It makes available the steps and tools that will enable you to make proper and accurate decision on database designs and operations whenever the need arises. Thus, we intend to achieve through the following.

COURSE AIMS

- Introduce the concepts associated with information systems development;
- Provide necessary tools for analyzing, designing, developing a database of any size;
- Provide you with the necessary foundation in database programming
- Introduction of web services and their architectural frameworks; and
- Provide you with the necessary foundation on the use of XML

COURSE OBJECTIVES

A number of objectives have been set out to ensure that the course achieves its aims. Apart from the course objectives, every unit of this course has set objectives. In the course of the study, you will need to confirm, at the end of each unit, if you have met the objectives set at the beginning of each unit. By the end of this course you should be able to:

- explain the term information system.
- identify the various types of information systems.
- write Structured Query Language statements.
- state the meaning, classification and properties of functional dependency and the description of Relational Algebra.
- use Structured Query Language (SQL) to retrieve data from and manipulate data in a database.
- use the JDBC to access databases.
- describe the basic concept and application of web services.
- identify web services framework.
- state the rules of XML documents.
- indentify programming interfaces that work with XML documents.
- define the concept of regular expression

WORKING THROUGH THIS COURSE

In order to have a thorough understanding of the course units, you will need to read and understand the contents, practise the steps by designing an information system of your own, and be committed to learning and implementing your knowledge.

This course is designed to cover approximately seventeen weeks, and it will require your devoted attention. You should do the exercises in the Tutor-Marked Assignments and submit to your tutors.

COURSE MATERIALS

These include:

1. Course Guide
2. Study Units
3. Recommended Texts
4. A file for your assignments and for records to monitor your progress.

STUDY UNITS

Module 1

Unit 1 Introduction to Information Systems
Unit 2 Introduction to Basic SQL
Unit 3 SQL Syntax I
Unit 4 SQL Syntax II
Unit 5 More SQL Statements
Unit 6 Database Programming and JBDC

Module 2

Unit 1 Conceptual Modelling and Schema Design
Unit 2 Functional Dependence
Unit 3 Regular Expression
Unit 4 Relational Algebra

Module 3

Unit 1 Web Services
Unit 2 Introduction to XML
Unit 3 XML and XML Queries
Unit 4 Database Recovery

Make use of the course materials, do the exercises to enhance your learning.

TEXTBOOKS AND REFERENCES

Dostal, J. (2007). School Information Systems (Skolni Informacni Systemy). In: Infotech-Modern Information and Communication Technology in Education. Olomouc, EU: Votobia, 2007. s. 540 – 546. ISBN 978-80-7220-301-7.

Lindsay, John (2000). *Information Systems – Fundamentals and Issues*. Kingston University, School of Information Systems.

Laudon, Kenneth C. and Laudon, Jane P. (1996). *Management Information Systems*: Organisation and Technology, (4th ed). Upper Saddle River, NJ: Prentice-Hall.

Oz, Effy (1998). *Management Information Systems*. Cambridge, MA: Course Technology.

James Hoffman, (1997). SQL Tutorials. *Teach yourself SQL in 21 days,* (2nd ed.). Macmillan computer publishing

SQL – A practical introduction by Akeel I. Din

Introduction to SQL 9i from Oracle University www.wiki_SQL.com

Database System Concepts, (5th ed).

Teach Yourself SQL in 21 Days, (2nd ed), Macmillan Computer Publishing

An Introduction to Database Systems, Eighth Edition, C. J. Date,

Addison Wesley, 2004, ISBN: 0-321-19784-4.

Functional Dependencies, Barbara L. Marcolin 1999
http://www.lightenna.com/book/export/s5/155

Ullman, J.D. and Widom, J. (2002). A *First Course in Database Systems*, (2nd ed.).Prentice Hall.

T.J. Teorey Database modelling and Design, (3rd ed.). University of Michigan.

Mastering Regular Expressions by Jeffrey E. F. Friedl.

Jan, Goyvaerts. Regular Expressions.

Ashmore, D. C. (2000). "*Best Practices for JDBC Programming."* Java Developers Journal, 5: no. 4: 4254.

Blaha, M. R., W. J. (1988). Premerlani and J. E. Rumbaugh. "*Relational Database Design Using an Object-Oriented Methodology*." Communications of the ACM, 31: no. 4: 414427.

Brunner, R. J. (2000). "*The Evolution of Connecting*." Java Developers Journal, 5: no. 10: 2426.

Brunner, R. J. (2000). "*After the Connection*." Java Developers Journal, 5: no. 11: 4246.

Callahan, T. (1998). "*So You Want a Stand-Alone Database for Java*." Java Developers Journal, 3: no. 12: 2836.

Codd, E. F. (1970). "*A Relational Model of Data for Large Shared Data Banks*." Communications of the ACM, June.

Codd, E. F. (1972). "*Further Normalization of the Data Base Relational Model*." Courant Computer Science Symposia, Vol. 6, Data Base Systems. Upper Saddle River, NJ: Prentice Hall.

Codd, E. F. (2000). "*Fatal Flaws in SQL*." Datamation, 34: no. 16 (1988): 4548.

Cooper, J. W. "*Making Databases Easier for Your Users*." Java Pro, 4: no. 10: 4754.

Date, C. J. (2003). *An Introduction to Database Systems*, 8/e. Reading, MA: Pearson Education.

Deitel, H. M.; Deitel, P. J. and D. R. Choffnes (2004). *Operating Systems*, (Third Edition). Upper Saddle River, NJ: Prentice Hall.

Duguay, C. (1999/2000). "*Electronic Mail Merge*." Java Pro, Winter, 2232.

Ergul, S. (2001). "*Transaction Processing with Java*." Java Report, January, 3036.

Jeffrey, Ullman. Relational Algebra.

Ramakrishnan and Gehrke, J. Database Management Systems (3rd ed.).

Isabelle, Bichindaritz. Database Systems Design.

Paul, Werstein. Relational Algebra.

A word definition from webopedia computer dictionary

Luis Felipe Cabrera (2005). *Web Services Atomic Transaction* (WS-Atomic Transaction)

Bright; *et al*. (1992). *Policy of Exchanging Data with Other Databases.*

Heimbigner and McLeod (1985).

Sheth and Larson (1990). heterogeneous database system C/C++: See Rick Parrish's article at www-106.ibm.com/developerworks/library/x-ctlbx.html (developerWorks, September 2001).

Java: See Doug Tidwell's article at www-106.ibm.com/developerworks/library/j-java-xml-toolkit/index.html (developerWorks, May 2000).

Perl: See Parand Tony Darugar's article at www-106.ibm.com/developerworks/library/x-perl-xml-toolkit/index.html (developerWorks, June 2001).

PHP: See Craig Knudsen's article at www-106.ibm.com/developerworks/library/x-php-xml-toolkit.html (developerWorks,June 2000).

ASSIGNMENT FILE

These are of two types: the Self Assessment Exercises and the Tutor-Marked Assignments. The self assessment exercises will enable you monitor your performance by yourself, while the Tutor-Marked Assignment is a supervised assignment. The assignments take a certain percentage of your total score in this course. The Tutor-Marked Assignments will be assessed by your tutor within a specified period. The examination at the end of this course will aim at determining the level of mastery of the subject matter. This course includes twelve Tutor-Marked Assignments and each must be done and submitted accordingly. Your best scores, however, will be recorded for you. Be sure to send these assignments to your tutor before the deadline to avoid loss of marks.

PRESENTATION SCHEDULE

The Presentation Schedule included in your course materials gives you the important dates for the completion of tutor-marked assignments and attending tutorials. Remember, you are required to submit all your assignments by the due date. You should guard against lagging behind in your work.

ASSESSMENT

There are two aspects to the assessment of the course. First, are the tutor-marked assignments; second, is a written examination.

In tackling the assignments, you are expected to apply the information and knowledge acquired during this course. The assignments must be submitted to your tutor for formal assessment in accordance with the deadlines stated in the Assignment File. The work you submit to your tutor for assessment will count for 30% of your total course mark.

At the end of the course, you will need to sit for a final three-hour examination. This will also count for 70% of your total course mark.

TUTOR-MARKED ASSIGNMENTS (TMAS)

There are twelve tutor-marked assignments in this course. You need to submit all the assignments. The total marks for the best four (4) assignments will be 30% of your total course mark.

Assignment questions for the units in this course are contained in the Assignment File. You should be able to complete your assignments from the information and materials contained in your set textbooks, reading and study units. However, you may wish to use other references to broaden your viewpoint and provide a deeper understanding of the subject.

When you have completed each assignment, send it together with the form to your tutor. Make sure that each assignment reaches your tutor on or before the deadline given. If, however, you cannot complete your work on time, contact your tutor before the assignment is done to discuss the possibility of an extension.

FINAL EXAMINATION AND GRADING

The final examination for the course will carry 70% percentage of the total marks available for this course. The examination will cover every aspect of the course, so you are advised to revise all your corrected assignments before the examination.

This course endows you with the status of a teacher and that of a learner. This means that you teach yourself and that you learn, as your learning capabilities would allow. It also means that you are in a better position to determine and to ascertain the what, the how, and the when of your language learning. No teacher imposes any method of learning on you.

The course units are similarly designed with the introduction following the table of contents, then a set of objectives and then the dialogue and so on.

The objectives guide you as you go through the units to ascertain your knowledge of the required terms and expressions.

COURSE MARKING SCHEME

This table shows how the actual course marking is broken down.

Table 1: **Course Marking Scheme**

Assessment	**Marks**
Assignments 1- 4	Four assignments, best three marks of the four count at 30% of course marks
Final Examination	70% of overall course marks
Total	100% of course marks

COURSE OVERVIEW

Unit	**Title of Work**	**Weeks Activity**	**Assessment (End of Unit)**
	Course Guide	Week 1	
	Module 1		
1	Introduction to Information Systems	Week 1	Assignment 1
2	Introduction to Basic SQL	Week 2	Assignment 2
3	SQL Syntax I	Week 3	Assignment 3
4	SQL Syntax II	Week 4 - 5	Assignment 4
5	More SQL Statements	Week 6	Assignment 5
6	Database Programming with JDBC	Week 7	Assignment 6
	Module 2		
1	Conceptual Modelling and Schema Design	Week 8	Assignment 7
2	Functional Dependency	Week 9	Assignment 8
3	Regular Expression	Week 10	Assignment 9
4	Relational Algebra	Week 11	Assignment 10
	Module 3		
1	Web Services	Week 12	Assignment 11
2	Introduction to XML	Week 13	Assignment 12
3	XML and XQueries	Week 14	Assignment 13
4	Database Recovery	Week 15	Assignment 14
	Revision	Week 16	
	Examination	Week 17	
	Total	**15 weeks**	

HOW TO GET THE MOST FROM THIS COURSE

In distance learning, the study units replace the university lecturer. This is one of the great advantages of distance learning; you can read and work through specially designed study materials at your own pace, and at a time and place that suit you best. Think of it as reading the lecture instead of listening to a lecturer. In the same way that a lecturer might set you some reading to do, the study units tell you when to read your set books or other material. Just as a lecturer might give you an in-class exercise, your study units provide exercises for you to do at appropriate points.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit and how a particular unit is integrated with the other units and the course as a whole. Next is a set of learning objectives. These objectives enable you know what you should be able to do by the time you have completed the unit. You should use these objectives to guide your study. When you have finished the units you must go back and check whether you have achieved the objectives. If you make a habit of doing this you will significantly improve your chances of passing the course.

Remember that your tutor's job is to assist you. When you need help, do not hesitate to call and ask your tutor to provide it.

1. Read this Course Guide thoroughly.
2. Organise a study schedule. Refer to the 'Course Overview' for more details. Note the time you are expected to spend on each unit and how the assignments relate to the units. Whatever method you chose to use, you should decide on it and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they lag behind in their course work.
4. Turn to Unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the 'Overview' at the beginning of each unit. You will almost always need both the study unit you are working on and one of your set of books on your desk at the same time.
6. Work through the unit. The content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit you will be instructed to read sections from your set books or other articles. Use the unit to guide your reading.

7. Review the objectives for each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.
8. When you are confident that you have achieved a unit's objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.
9. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor-marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have any questions or problems.
10. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this Course Guide).

FACILITATORS/TUTORS AND TUTORIALS

There are 15 hours of tutorials provided in support of this course. You will be notified of the dates, times and location of these tutorials, together with the name and phone number of your tutor, as soon as you are allocated 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 or submit your tutor-marked assignments 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 telephone, or e-mail if you need help. The following might be circumstances in which you would find help necessary. Contact your tutor if:

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

You should try your best to attend the tutorials. This is the only chance to have face to face contact 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 the maximum benefit from course tutorials, prepare a question list before attending them. You will learn a lot by participating in discussions actively.

SUMMARY

Information Systems introduce you to the concepts associated with Information systems development which is critical in understanding the various computer technology and data communications technology. The content of the course material was planned and written to ensure that you acquire the proper knowledge and skills for the appropriate situations. Real-life situations have been created to enable you identify with and create some of your own. The essence is to help you in acquiring the necessary knowledge and competence by equipping you with the necessary tools to accomplish this.

We hope that by the end of this course you would have acquired the required knowledge to view Information Systems in a new way.

I wish you success with the course and hope that you will find it both interesting and useful.

MAIN COURSE

CONTENTS **PAGE**

MODULE 1

UNIT 1 INTRODUCTION TO INFORMATION SYSTEMS

CONTENTS

1.0 INTRODUCTION

Having read through the course guide, you will have a general understanding of what this unit is about and how it fits into the course as a whole. This unit describes the general concept of Information Systems (IS), types and its application areas.

2.0 OBJECTIVES

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

- explain the term information system
- identify the various types of IS
- relate the history of IS
- describe IS department.

3.0 MAIN CONTENT

3.1 Information Systems

The term Information System (IS) refers to information technology that is used by people to accomplish a specified organisational or individual objective. The technology may be used in the gathering, processing, storing, and/or dissemination of information, and the users are trained in the use of that technology, as well as in the procedures to be followed in doing so. The specific technologies that collectively comprise information technology are computer technology and data communications technology.

Information system (IS) sometimes refers to a system of persons, data records and activities that process the data and information in an organisation, and it includes the organisation's manual and automated processes. Computer-based information systems are the field of study for information technology, elements of which are sometimes called an "information system" as well; a usage some consider to be incorrect.

Advances in computer hardware, software, and networking technologies have spurred an evolution in the structure, design, and use of corporate information systems.

3.2 Overview

The term "Information System" has different meanings:

- Generally, Information System is described by three objects: Structure, channels and networks.

Structure:

- Repositories, which hold data permanently or temporarily, such as buffers, RAM, hard disks, cache, etc.
- Interfaces, which exchange information with the non-digital world, such as keyboards, speakers, scanners, printers, etc.

Channels: which connect repositories, such as buses, cables, wireless links, etc. Network: is a set of logical or physical: an introduction to informatics in organisations.

The most common view of an information system is one of Input-Process-Output.

3.3 History of Information Systems

The study of information systems originated as a sub-discipline of computer science in an attempt to understand and rationalise the management of technology within organisations. It has matured into a major field of management that is increasingly being emphasised as an important area of research in management studies, and is taught in all major universities and business schools in the world. Börje Langefors introduced the concept of "Information Systems" at the third International Conference on Information Processing and Computer Science in New York in 1965.

Information technology is a very important malleable resource available to executives. Many companies have created a position of Chief Information Officer (CIO) that sits on the executive board with the Chief Executive Officer (CEO), Chief Financial Officer (CFO), Chief Operating Officer (COO) and Chief Technical Officer (CTO).The CTO may also serve as CIO, and vice versa. The Chief Information Security Officer (CISO), which focuses on information security within an organisation, normally reports to the CIO.

3.4 Types of Information Systems

From prior studies and experiences with information systems there are at least four classes of information systems:

3.4.1 Transaction Processing Systems

These record and track an organisation's transactions, such as sales transactions or inventory items, from the moment each is first created until it leaves the system. This helps managers at the day-to-day operational level keep track of daily transactions as well as make decisions on when to place orders, make shipments, and so on.

3.4.2 Management Information and reporting Systems (MIS)

These systems provide mid-level and senior managers with periodic, often summarized, reports that help them assess performance (e.g., a particular region's sales performance in a given time period) and make appropriate decisions based on that information. MIS is a subset of the overall internal controls of a business covering the application of people, documents, technologies, and procedures by management accountants to solving business problems such as costing a product, service or a business-wide strategy. Management Information Systems are distinct from regular information systems in that they are used to analyse other information systems applied in operational activities in the organisation.

Academically, the term is commonly used to refer to the group of information management methods tied to the automation or support of human decision making, e.g. Decision Support Systems, Expert systems, and Executive information systems.

3.4.3 Decision Support Systems

These systems are designed to help mid-level and senior managers make those difficult decisions about which not every relevant parameter is known. These decisions, referred to as *semi-structured decisions*, are characteristic of the types of decisions made at the higher levels of management. A decision on whether or not to introduce a particular (brand new) product into an organisation's product line is an example of a semi-structured decision. Another example is the decision on whether or not to open a branch in a foreign country. Some of the parameters that go into the making of these decisions are known. The value of a Decision Support System (DSS) is in its ability to permit "what-if" analyses (e.g., What if interest rates rose by 2 per cent? What if our main competitor lowered its price by 5 per cent? What if import tariffs are imposed/increased in the foreign country in which we do, or plan to do, business?). That is, a DSS helps the user (decision maker) to model and analyse different scenarios in order to arrive at a final, reasonable decision, based on the analysis. There are decision support systems that help groups (as opposed to individuals) to make consensus-based decisions. These are known as Group Decision Support Systems (*GDSS*).

A type of decision support system that is geared primarily toward high-level senior managers is the Executive Information System (EIS) or Executive Support System (ESS). While this has the capability to do very detail analyses, just like a regular DSS, it is designed primarily to help executives keep track of a few selected items that are critical to their day-to-day high-level decisions. Examples of such items include performance trends for selected product or customer groups, interest rate yields, and the market performance of major competitors.

3.4.4 Expert Systems

An expert system is built by modeling into the computer the thought processes and decision-making heuristics of a recognised expert in a particular field. Thus, this type of information system is *theoretically* capable of making decisions for a user, based on input received from the user. However, due to the complex and uncertain nature of most business decision environments, expert system technology has traditionally been used in these environments primarily like decision support systems "â€" that is, to help a human decision maker arrive at a

reasonable decision, rather than to actually *make* the decision for the user.

3.5 Information System Department

The IS department partly governs the information system development, use, application and influence on a business or corporation. An IS department typically provides:

- technologically implemented medium for recording, storing, and disseminating information
- techniques for drawing conclusions from such information.

Nowadays, IS department is also known as MIS, IT or simply Systems department.

4.0 CONCLUSION

In this unit, you have been introduced to the fundamental concepts of Information Systems. You have also learnt the different types of information systems and its areas of application. It is the basis for information technology systems.

5.0 SUMMARY

What you have learnt in this unit concerns:

- introduction to Information Systems which refers to a system of persons, data records and activities that process the data and information in an organisation.
- the study of information systems originated as a sub-discipline of computer science in an attempt to understand and rationalise the management of technology within organisations.
- areas of application or work which include:
  - Information Systems Strategy
  - Information Systems Management and
  - Information Systems Development.
- types of Information Systems: Management Information Systems (MIS) or Reporting Systems, Decision Support Systems, Transaction Information System (TIS) and Expert Systems.

SELF-ASSESSMENT EXERCISE

i. What do you understand by information system?
ii. A system built by modeling into the computer the thought processes and decision-making heuristics of a recognised expert in a particular field is called…….

6.0 TUTOR-MARKED ASSIGNMENT

i. List and explain the different types of Information Systems.
ii. Write a short note on the areas of application of IS.

7.0 REFERENCES/FURTHER READING

Dostal, J. (2007). *School Information Systems* (Skolni Informacni System). *In*: Infotech-Modern Information and Communication Technology in Education. Olomouc, EU: Votobia, 2007. s. 540 – 546. ISBN 978-80-7220-301-7.

Lindsay, John (2000). *Information Systems – Fundamentals and Issues*. Kingston University, School of Information Systems.

Laudon, Kenneth C. and Laudon, Jane P. (1996). Management Information Systems: *Organisation and Technology* (4th ed.). Upper Saddle River, NJ: Prentice-Hall.

Oz, Effy (1998). *Management Information Systems*. Cambridge, MA: Course Technology.

UNIT 2 INTRODUCTION TO BASIC SQL

CONTENTS

1.0 INTRODUCTION

An aspect of information Systems is data processing, with the prior knowledge of information systems and computer technology as a whole it will be easy to introduce the basic concept of Structured Query Language which is a useful tool in accessing and manipulating databases. You will be introduced to the basic statement of the SQL programmes that will enable you write simple database programmes.

2.0 OBJECTIVES

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

- explain the basics of the SQL programmes
- define the standard keywords of the SQL programmes
- work on any SQL platform/server such as mysql, Ms Access etc
- manage and access databases of any size easily.

3.0 MAIN CONTENT

3.1 Introduction to Database and Structured Query Language (SQL)

A **database** is an organised collection of data. There are many different strategies for organising data to facilitate easy access and manipulation. A database management system (DBMS) provides mechanisms for storing, organising, retrieving and modifying data for many users. Database management systems allow for the access and storage of data without concern for the internal representation of data.

Today's most popular database systems are **relational databases**. A language called **SQL** pronounced "sequel," or as its individual letters is the international standard language used almost universally with relational databases to perform queries (i.e., to request information that satisfies given criteria) and to manipulate data.

- SQL is an acronym for Structured Query Language, stands for Structured Query Language
- SQL is used to access and manipulate databases
- SQL is an ANSI (American National Standards Institute) standard

It is a database language that is used for querying and modifying relational databases. SQL is a programming language for querying and modifying data and managing databases. Using SQL, you can communicate with the database server. SQL has the following advantages:

- efficient
- easy to learn and use
- functionally complete(With SQL, you can define, retrieve, and manipulate data in the tables)

Although SQL is an ANSI (American National Standards Institute) standard, there are many different versions of the SQL language.

Note: Most of the SQL database programmes also have their own proprietary extensions in addition to the SQL standard!

3.2 History of SQL

SQL was developed by IBM Research in the mid 70s and standardised by the ANSI and later by the ISO. Most database management systems implement a majority of one of these standards and add their proprietary extensions. SQL allows the retrieval, insertion, updating, and deletion of data. A database management system also includes management and administrative functions. Most – if not all – implementations also include a command-line interface (SQL/CLI) that allows for the entry and execution of the language commands, as opposed to only providing an application programming interface (API) intended for access from a graphical user interface (GUI).

The first version of SQL was developed at IBM by Andrew Richardson, Donald C. Messerly and Raymond F. Boyce in the early 1970s. This version, initially called **SEQUEL**, was designed to manipulate and retrieve data stored in IBM's original relational database product; System R. IBM patented their version of SQL in 1985, while the SQL language was not formally standardised until 1986 by the American National Standards Institute (ANSI) as SQL-86. Subsequent versions of the SQL standard have been released by ANSI and as International Organisation for Standardisation (ISO) standards.

Originally designed as a declarative query and data manipulation language, variations of SQL have been created by SQL database management system (DBMS) vendors that add procedural constructs, flow-of-control statements, user-defined data types, and various other language extensions. With the release of the SQL: 1999 standard, many such extensions were formally adopted as part of the SQL language via the SQL Persistent Stored Modules (SQL/PSM) portion of the standard.
SQL was adopted as a standard by ANSI in 1986 and ISO in 1987. In the original SQL standard, ANSI declared that the official pronunciation for SQL is "es queue el". However, many English-speaking database professionals still use the nonstandard pronunciation /ˈsiːkwəl/ (like the word "sequel"). As mentioned above, SEQUEL was an earlier IBM database language, a predecessor to the SQL language. SQL is designed for a specific purpose: to query data contained in a relational database. SQL is a set-based, declarative query language, not an imperative language such as C or BASIC. However, there are extensions to Standard SQL which add procedural programming language functionality, such as control-of-flow constructs. An example is the Procedural Language of SQL (PL/SQL).

3.3 The Basic Categories of SQL Statements

SQL statements are basically divided into four; viz:

- Data Manipulation Language (DML)
- Data Definition Language (DDL)
- Data Control Language (DCL)
- Transaction Control

Data Manipulation Language (DDL)

- **DML** retrieves data from the database, enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. The basic Data Manipulation Language (DDL) includes the following:
  - select statement
  - insert statement
  - update statement
  - delete statement
  - merge statement
- **DDL** sets up, changes and removes data structures from tables. The basic Data Definition Language includes the following:
  - create statement
  - alter statement
  - drop statement
  - rename statement
  - truncate statement
  - comment statement
- **DCL** gives or removes access rights to both a database and the structures within it. The basic Data Control Languages are:
  - grant statement
  - revoke statement
- **Transaction Control** manages the changes made by the DML statements. Changes to the data can be grouped together into logical transactions. The basic Transaction Control Languages are:
  - commit
  - rollback
  - save point

Using the following simple rules and guidelines, you can construct valid statements that are both easy to read and easy to edit.

- SQL statements are not case sensitive, unless indicated
- SQL statements can be entered on one or many lines
- Keywords cannot be split across lines or abbreviated

- Clauses are usually placed on separate lines for readability
- Indents should be used to make code readable
- Keywords typically are entered in uppercase; all other words, such as table names and columns are entered in lowercase

3.4 Viewing the Structure of a Table

The structure of any database table can be viewed by using the describe clause of the SQL statement. The general syntax of the *describe* statement is given below:

DESCRIBE table:

For the purpose of this course two tables called Departments and Employees in the Oracle database will be used. Thus, we need to see the structure of this table so that we will be able to familiarise ourselves with the column used in the table. To do this, we write the query:

DESCRIBE departments:

Name	NULL?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

From the table above, we can infer that departments table has 4 columns and that 2 of these columns are not allowed to be null.

DESCRIBE employee:

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)

DEPARTMENT_ID		NUMBER(4)

From the table above, we can infer that employees table has 11 columns and that 5 of these columns are not allowed to be null.

3.5 Writing Basic SQL Select Statements

To extract data from the database, you need to use the SQL SELECT statement. You may need to restrict the columns that are displayed. Using a SELECT statement, you can do the following:

- **Projection:** You can use the projection capability to choose the columns in a table that you want to return by your query. You can choose as few or as many columns of the table as you require.
- **Selection:** You can use the selection capability in SQL to choose the rows in a table that you want to return by a query. You can use various criteria to restrict the rows that you use.
- **Joining:** You can use the join capability to bring together data that is stored in different tables by creating a link between them.

3.6 Summary of the Function of SQL

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views
- SQL can allow the construction codes manipulating database

3.7 Using SQL for Web Site

To build a web site that shows some data from a database, you will need the following:

- An RDBMS database programme (i.e. MS Access, SQL Server, MySQL)
- A server-side scripting language, like PHP or ASP
- SQL
- HTML / CSS

3.8 Relational Database Management System

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. Relational database will further be described in Module 2.

3.9 Introduction to SQL SYNTAX

3.9.1 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”

P_Id	LastName	FirstName	Address	City
1	Akinbode	Ola	10, Odeku Str.	Lagos
2	Okafor	Chris	23, Princewill Drive	Porthacourt
3	Amodu	Ali	20, Dauda lane	Kaduna

The table above contains three records (one for each person) and five columns (P_Id, LastName, FirstName, Address, and City).

3.9.2 Format of SQL Statements

Most of the actions you need to perform on a database are done with SQL statements.

The following SQL statement will select all the records in the “Persons” table:

```
SELECT * FROM Persons
```

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.

3.9.3 SQL, DML and DDL

SQL can be divided into two parts: The Data Manipulation Language (DML) and 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 defines indexes (keys), specifies links between tables, and imposes constraints between tables. The most important DDL statements in SQL are:

- **CREATE DATABASE** - creates a new database
- **ALTER DATABASE** - modifies a database
- **CREATE TABLE** - creates a new table
- **ALTER TABLE** - modifies a table
- **DROP TABLE** - deletes a table
- **CREATE INDEX** - creates an index (search key)
- **DROP INDEX** - deletes an index

4.0 CONCLUSION

In this unit you have been introduced to the fundamental concept of a typical database computing environment. You also learnt the specific requirement of an environment for the development of SQL statements. You were also introduced to various SQL statements necessary in writing simple SQL codes.

5.0 SUMMARY

What you have learned in this unit concerns.

- Structured Query Language (SQL) which is a standard language for accessing and manipulating databases.
- The different SQL statements
- What SQL statement can be used for
- Not all SQL statements accept semicolon at the end of it depending on the platform or server that is used to execute the statement.

SELF-ASSESSMENT EXERCISE

i. What does SQL mean?
ii. List and state the functions of the component of the DDL parts of SQL programme.

6.0 TUTOR-MARKED ASSIGNMENT

Write a short note on structured query language programme and explain the basic components of SQL.

7.0 REFERENCE/FURTHER READING

Hoffman, James (1997). SQL Tutorials.

UNIT 3 SQL SYNTAX I

CONTENTS

1.0 INTRODUCTION

This unit will introduce you to how to write basic SQL programmes such as creating tables, selecting a view from a table and familiarise you with basic SQL operators.

2.0 OBJECTIVES

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

- write simple SQL programs
- familiarise yourself with standard keywords of the SQL programs
- understand how to construct a good SQL statement
- manage and access databases using the create, select, where and the logical operators.

3.0 MAIN CONTENT

3.1 SQL Create Table Statement

The CREATE TABLE statement is used to create a table in a database.

3.1.1 SQL CREATE TABLE Syntax

```
CREATE TABLE table_name
(
column_name1 data_type,
column_name2 data_type,
column_name3 data_type,
....
)
```

The data type specifies what type of data the column can hold. For a complete reference of all the data types available in MS Access, MySQL, and SQL Server visit www.datatyperef.com

3.1.2 CREATE TABLE Example

Now we want to create a table called “Persons” that contains five columns: P_Id, LastName, FirstName, Address, and City.

We use the following CREATE TABLE statement:

```
CREATE TABLE Persons
(
P_Id int,
LastName varchar(255),
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

The P_Id column is of type int and will hold a number. The LastName, FirstName, Address, and City columns are of type varchar with a maximum length of 255 characters.

The empty “Persons” table will now look like this:

P_Id	LastName	FirstName	Address	City

The empty table can be filled with data with the INSERT INTO statement.

3.2 The SQL SELECT Statement

The SELECT statement is used to select data from a database.

The result is stored in a result table, called the result-set.

3.2.1 SQL SELECT Syntax

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.

3.2.2 An SQL SELECT Example

The “Persons” table:

P_Id	LastName	FirstName	Address	City
1	Akinbode	Ola	10, Odeku Str.	Lagos
2	Okafor	Chris	23, Princewill Drive	Porthacourt
3	Amodu	Ali	20, Dauda lane	Kaduna

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
Akinbode	Ola
Okafor	Chris
Amodu	Ali

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	Akinbode	Ola	10, Odeku Str.	Lagos
2	Okafor	Chris	23, Princewill Drive	Porthacourt
3	Amodu	Ali	20, Dauda lane	Kaduna

3.2.3 Navigation in a Result Set

Most database software systems allow navigation in the result-set with programming functions, like: Move-To-First-Record, Get-Record-Content, Move-To-Next-Record, etc.

3.3 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.

3.3.1 SQL SELECT DISTINCT Syntax

```
SELECT DISTINCT column_name(s)
FROM table_name
```

3.3.2 SELECT DISTINCT Example

The "PersonsOne" table:

P_Id	LastName	FirstName	Address	City
1	Akinbode	Ola	10, Odeku Str.	Lagos
2	Okafor	Chris	23, Princewill Drive	Lagos
3	Amodu	Ali	20, Dauda lane	Kaduna

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
Lagos
Kaduna

3.4 SQL WHERE Clause

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified criterion.

3.4.1 SQL WHERE Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator value
```

3.4.2 WHERE Clause Example

The “Persons” table:

P_Id	LastName	FirstName	Address	City
1	Akinbode	Ola	10, Odeku Str.	Lagos
2	Okafor	Chris	23, Princewill Drive	Porthacourt
3	Amodu	Ali	20, Dauda lane	Kaduna

Now we want to select only the persons living in the city “Sandnes” from the table above.

We use the following SELECT statement:

```
SELECT * FROM Persons
WHERE City='Kaduna'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Amodu	Ali	20, Dauda lane	Kaduna

3.4.3 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='Chris'
This is wrong:
SELECT * FROM Persons WHERE FirstName=Chris
```

For numeric values:

```
This is correct:
SELECT * FROM Persons WHERE Year=1965
This is wrong:
SELECT * FROM Persons WHERE Year='1965'
```

3.4.4 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 !=

3.5 SQL AND & OR Operators

The AND & OR operators are used to filter records based on more than one condition.

The AND operator displays a record if both the first condition and the second condition is true while the OR operator displays a record if either the first condition or the second condition is true.

3.5.1 AND Operator Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Akinbode	Ola	10, Odeku Str.	Lagos
2	Okafor	Chris	23, Princewill Drive	Porthacourt
3	Amodu	Ali	20, Dauda lane	Kaduna

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='Ola'
AND LastName='Akinbode'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Akinbode	Ola	10, Odeku Str.	Lagos

3.5.2 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='Chris'
OR FirstName='Ali'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
2	Okafor	Chris	23, Princewill Drive	Porthacourt
3	Amodu	Ali	20, Dauda lane	Kaduna

3.5.3 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='Akinbode' OR LastName='Okafor'
```

The result-set will look like this:

P_Id	LastName	FirstName	Address	City
1	Akinbode	Ola	10, Odeku Str.	Lagos
2	Okafor	Chris	23, Princewill Drive	Porthacourt

SELF-ASSESSMENT EXERCISE

i. Create a table called student which will contain the following: student_id, studentname, dept, level and grade?
ii. Write the syntax to insert into the table created in exercise 2.1

4.0 CONCLUSION

In this unit you have been introduced to the fundamental Queries of a typical database computing environment e. g. SQL. You also learnt the specific operators that work for SQL statements. You were also introduced to various SQL statements necessary in writing simple SQL codes.

5.0 SUMMARY

The summaries of what you have learnt are:

- structured Query Language (SQL) which is a standard language for accessing and manipulating databases.
- the syntax of different SQL statement
- what SQL statement can be used for
- not all SQL statements accept semicolon at the end of it depending on the platform or server that is used to execute the statement.

SELF-ASSESSMENT EXERCISE

i. Create Table Student
(Student_id Varchar (25),

Student name Varchar (70),
Dept varchar2 (255),
Level char (12),
Grade number (3))

ii. INSERT INTO Student (Student_id, Student name, Dept, Level, Grade)
VALUES (value1, value2, value3, value4, value5)

6.0 TUTOR-MARKED ASSIGNMENT

Create a database performing all the discussed activities in this unit. i. e. create a table, insert into that table etc.

7.0 REFERENCES/FURTHER READING

Hoffman, James (1997). SQL Tutorials.

Teach Yourself SQL in 21 Days, (2nd ed.). Macmillan Computer Publishing SQL – A Practical Introduction by Akeel I. Din.

UNIT 4 SQL SYNTAX II

CONTENTS

1.0 INTRODUCTION

This unit introduces you to how to write basic SQL programmes such as creating tables, selecting a view from a table and familiarise you with basic SQL operators.

2.0 OBJECTIVES

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

- write simple SQL programmes
- familiarise yourself with standard keywords of the SQL programmes
- state how to construct a good SQL statement
- manage and access databases using the create, select, where and the logical operators.

3.0 MAIN CONTENT

3.1 The ORDER BY Keyword

3.1.2 ORDER BY Syntax

The ORDER BY keyword is used to sort the result-set. The ORDER BY keyword is used to sort the result-set by a specified column. The ORDER BY keyword sorts the records in ascending order by default. If you want to sort the records in a descending order, you can use the DESC keyword. The order by clause comes last in a select statement.

The syntax of the order by clause given below:

- SELECT *expr*
- FROM *table*
- [WHERE *condition(s)*]
- [ORDER BY{*column, expr}* [ASC|DESC]
- In the syntax,
- ORDER BY specifies the order in which the retrieved rows are displayed
- ASC specifies rows in ascending order (this is the default value)
- DESC order the rows in descending order

With the ORDER BY clause:

- numeric values are displayed with the lowest value first e.g 1-999
- date values are displayed with the earliest value first e.g 01-JAN-92 before 01-JAN-95
- character values are displayed in alphabetical order
- null values are displayed last for ascending sequences and first for descending sequences.

3.1.3 ORDER BY Example

Sorting in descending order

SELECT last_name, job_id, department_id, hire_date
FROM employees

ORDER BY last_name:

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
Akinbode	SA_REP	80	21-APR-00
Amodu	SA_REP	20	21-APR-00
Buba	SA_REP	80	24-MAR-00
Ngozi	ST_CLERK	50	08-MAR-00
Okafor	SA_REP	80	23-FEB-00
Sowale	ST_CLERK	50	06-FEB-00

We also sort using multiple columns.
For example,

SELECT last_name, department_id, salary FROM employees

ORDER BY department_id, salary DESC;

3.2 SQL INSERT INTO Statement

3.2.1 SQL INSERT INTO Syntax

The **INSERT INTO** statement is used to insert new records into a new row in a table.

The syntax of the insert statement is:

INSERT INTO *table [{column, [,column…..])]*
VALUES (*value [, value….]);*
In the syntax,
table is the name of the table
column is the name of the column
v*alue* is the corresponding value for the column.

3.2.2 SQL INSERT INTO Example

INSERTING NEW ROWS

E.G

INSERT INTO departments (department_id, department_name, manager_id, location_id)

VALUES (170, ‘Public Relations’,100,1700);

INSERTING ROWS WITH NULL VALUES

Implicit method example

INSERT INTO departments (department_id, department_name)
Values (30,'Purchasing'):

Explicit Method example
INSERT INTO departments
Values (100, 'Finance', NULL, NULL);
INSERTING SPECIAL VALUES

Example
INSERT INTO employees (employee_id, first_name, last_name, email, phone_number, hire_date, job_id,salary, commission_pct, manager_id, department_id)

Values (7, 'Adeola', 'Chalse', 'ade_char', '2348039990985', SYSDATE, 'AC_ACCOUNT', 6900, NULL, 205, 100);

3.3 SQL UPDATE Statement

3.3.1 SQL UPDATE Statement Syntax

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
```

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!

3.3.2 SQL UPDATE Examples

Updating rows in a table
UPDATE employees
SET department_id=70
WHERE employee_id = 113;

3.4 DQL DELETE Statement

3.4.1 SQL DELETE Syntax

The DELETE statement is used to delete records and rows in a table.

DELETE FROM table_name
WHERE some_column=some_value

3.4.2 SQL DELETE Examples

DELETE *
FROM employees;
DELETE FROM employees
Where department_id =60;

3.4.3 Delete All Rows

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:

```
DELETE FROM table_name
or
DELETE * FROM table_name
```

Note: Be very careful when deleting records. You cannot undo this statement!

3.5 Joining Tables

The select statement can be used to join two tables together. It can be used to extract part of Table A and part of Table B to form Table C. For example, assuming *student* and *studentclass* are two different tables. Let us look at this instruction:

- Select student.SID, student.name, studentclass.classname
- From sudent, studentclass
- Where student.SID = studentclass.SID

This statement shows that SID, name are columns or fields from student table and classname and SID are also columns from studentclass table. The fields in the new table to form by this instruction are:
SID name classname

SELF-ASSESSMENT EXERCISE

i. Write the SQL statement to delete two rows from student, Name and grade = 56?
ii. What is the syntax to arrange the element of table in Ascending and Descending order?

4.0 CONCLUSION

In this unit you have been introduced to the fundamental Queries of a typical database computing environment e. g. SQL. You also learnt the specific operators that work for SQL statements. You were also introduced to various SQL statement syntax necessary in writing simple SQL codes.

5.0 SUMMARY

What you have learnt in this unit concerns:

- Structured Query Language (SQL) which is a standard language for accessing and manipulating databases
- the syntax of different SQL statements
- what SQL statement can be used for

ANSWER TO SELF-ASSESSMENT EXERCISE

i. DELETE FROM student
   a. WHERE Name='Tjessem' AND grade= 56
ii. SELECT * FROM Tablename
   ORDER BY LastName DESC

6.0 TUTOR-MARKED ASSIGNMENT

Create a database performing all the discussed activities in this unit. i. e. create a table, insert into that table etc.

7.0 REFERENCES/FURTHER READING

Hoffman, James (1997). SQL Tutorials.

Teach Yourself SQL in 21 Days, (2nd ed). Macmillian Computer Publishing SQL – A Practical Introduction by Akeel I. Din.

UNIT 5 MORE SQL STATEMENTS

CONTENTS

1.0 INTRODUCTION

Since SQL is a database language that is used for querying and modifying relational databases, this unit introduces more SQL instructions manipulating database. It presents other statements apart from those described in Module 2.

2.0 OBJECTIVES

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

- write SQL statements
- use the ORDER BY statement
- use the INSERT INTO statement
- update a group of data
- delete rows from a table

3.0 MAIN CONTENT

3.1 Arithmetic Operations

Create expressions with number and date data by using arithmetic operators. You may need to modify the way in which data is displayed, perform calculations, or look at what-if scenarios. These are all possible using arithmetic expressions. An arithmetic expression can contain column names, constant numeric values and arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

3.1.1 Using Arithmetic Operators

Let us first extend Table persons to Table employees by adding salary column to it and adding three more records. For subsequent examples in this unit, the table is also assumed to have more than 5 columns.

P_Id	LastName	FirstName	Address	City	Salary
1	Akinbode	Ola	10, Odeku Str.	Lagos	4800
2	Okafor	Chris	23,Princewill Drive	Porthacourt	17000
3	Amodu	Ali	20, Dauda lane	Kaduna	12000
4	Buba	Ibrahim	12, Dongorayaro Str.	Kastina	9000
5	Ngozi	Ebe	10, Felix Str.	Imo	7700
6	Sowale	Ayo	63, Atoba road	Abeokuta	24000

The example below describes a scenario in which arithmetic operators can be used.

SELECT last_name, salary, salary+300
FROM employees;

This gives

LastName	Salary	Salary+300
Akinbode	4800	5100
Okafor	17000	17300
Amodu	12000	12300
Buba	9000	9300
Ngozi	7700	8000
Sowale	24000	24300

3.1.2 Operator Precedence

If an arithmetic expression contains more than one operator, multiplication and division are evaluated first. If operators within an expression are of the same priority, then evaluation is done from left to right. Parenthesis can be used to force the expression within parentheses to be evaluated first.

Multiplication and division take priority over addition and subtraction
A query that shows how operator precedence works is shown below:

SELECT last_name, salary, 12*salary+100
FROM employees

LastName	Salary	12*Salary+100
Akinbode	4800	57700
Okafor	17000	204100
Amodu	12000	144100
Buba	9000	108100
Ngozi	7700	92500
Sowale	24000	288100

A query that uses brackets to override the operator precedence is shown below:

SELECT last_name, salary, 12*(salary+100)
FROM employees

LastName	Salary	12*(Salary+100)
Akinbode	4800	58800
Okafor	17000	205200
Amodu	12000	145200
Buba	9000	109200
Ngozi	7700	93600
Sowale	24000	289200

3.1.3 Defining a Null Value

A null is a value that is unavailable, unassigned, unknown, or inapplicable. If a row lacks the data value for a particular column, that value is said to be null, or to contain a null. Columns of any data type can contain nulls. However, some constraints, NOT NULL and PRIMARY KEY, prevent nulls from being used in the column.

A query that shows the null values is shown below:

SELECT last_name, job_id, salary, commission_pct
FROM employees:

LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
Akinbode	ST_MAN	4800	
Okafor	ST_CLERK	17000	
Amodu	ST_CLERK	12000	
Buba	ST_CLERK	9000	
Ngozi	ST_CLERK	7700	
Sowale	ST_CLERK	24000	

In the COMMISSION_PCT column in the EMPLOYEES table, notice that only a sales manager (form the job_id column) can earn a commission.

Null Values in Arithmetic Expressions
Arithmetic expressions containing a null value evaluate to null.
E.g.

SELECT last_name, 12*salary*commission_pct
FROM employees:

LAST_NAME	12*SALARY*COMMISSION_PCT
Akinbode	
Okafor	
Amodu	
Buba	
Ngozi	
Sowale	

4.0 CONCLUSION

In this unit, you have learnt how to write basic SQL statements, using operators in SQL, how to use the SQL ORDER statement, to arrange a group of data. Also the SQL INSERT statement was explained, including how to update and delete rows in a table.

5.0 SUMMARY

What you have learned in this unit concerns:

- writing basic SQL statements
- ordering a group of data using the ORDER BY statement
- updating, inserting and deleting rows in a table using SQL statements

6.0 TUTOR-MARKED ASSIGNMENT

7.0 REFERENCES/FURTHER READING

Introduction to SQL 9i from Oracle University. www.wiki_SQL.com
Database System Concepts, (5th ed).

SQL Tutorials by James Hoffman, (1997). *Teach yourself SQL in 21 days*, (2nd ed). Macmillan Computer Publishing.

SQL – A Practical Introduction by Akeel I. Din (2004). An Introduction to Database Systems, (8th ed.). C. J. Addison Wesley. ISBN: 0-321-19784-4.

UNIT 6 DATABASE PROGRAMMING AND JBDC

CONTENTS

1.0 INTRODUCTION

In this unit, you will be introduced to the concept of database programming, which involves knowing what a database is, how to construct a simple database. The unit then teaches you how to query the database using SQL learnt in previous units then writing programmes in form of procedures which carry out specific assignments on the database.

2.0 OBJECTIVES

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

- explain the term database and Database Management Systems (DBMS) in use
- construct entity relationship diagrams
- identify the basic database objects
- programme a database using JDBC
- use JDBC in real world applications.

3.0 MAIN CONTENT

3.1 Introduction to Database

3.1.1 Definition of Database

A **database** is a structured collection of records or data that is stored in a computer system. A **database** is an organised body of related information that is organised so that it can be easily accessed, managed and updated. In one view, databases can be easily classified according to types of content bibliographic, full-text, numeric and images. The structure is achieved by organising the data according to a database model. The model in most common use today is the relational model. Other models such as the hierarchical model and the network model use a more explicit representation of relationships.

3.1.2 Classification of Database

In computing, databases are sometimes classified according to their organisational approach and these can be described as follows:

- **Relational database**: The most prevalent approach is the relational database. A tabular database in which data is defined so that it can be reorganised and accessed in number of different ways.
- **Distributed database:** A distributed database is one that can be dispersed or replicated among points in a network.
- **Object-oriented programming database:** An object-oriented programming database is one that is congruent with the data defined in object classes and subclasses. Computer databases typically contain aggregations of data records or files, such as sales transaction, product catalogs and inventories, and customer profiles. Typically, a database manager provides the capabilities of controlling read/write access specifying report generation, and analysing usage. Databases and database managers are prevalent in large mainframe systems, but are also present in smaller distributed workstations and mid-range systems such as the AS/400 and on personal computers.
- **Structural Query Language (SQL):** This is a language for making interactive queries from and updating a database such as IBM's DB2, Microsoft Access and database products from oracle, Sybase and computer associates.

3.1.3 Database Management System

A computer database relies upon software to organise the storage of data. This software is known as a database management system (DBMS). Database management systems are categorised according to the database model that they support. The model tends to determine the query languages that are available to access the database. A great deal of the internal engineering of a DBMS, however, is independent of the data model, and is concerned with managing factors such as performance, concurrency, integrity, and recovery from hardware failures. In these areas, there are large differences between products.

3.1.4 Relational Database Model

The relational database model is a logical representation of data that allows relationship among data to be considered without concern for the physical structure of the data. A relational database uses relations or two-dimension tables to store information. A table is the basic storage structure of a Relational Database Model. A table holds all the data necessary about something in the real world, such as employees, invoices, or customers. For example, one might want to store information about employees in a company. In relational database, one creates several tables to store different pieces of information about a company's employees, such as an employee table, a department table and a salary table.

A relational database:

- can be accessed and modified by executing structured SQL statements.
- contains a collection of tables with no physical pointers
- uses a set of operators

3.2 Database Object and Constraints

3.2.1 Database Objects

The commonly used database objects are:

- **Table:** This is the basic unit of storage; composed of rows
- **View:** This logically represents subsets of data from one or more tables
- **Sequence:** Generates numeric values
- **Index:** This improves the performance of some queries
- **Synonym:** Gives alternative names to objects

A **table** is the basic storage structure of a Relational Database Model. With **views,** one can present and hide data from tables. Many applications require the use of unique numbers as primary key values. One can either build code into the application to handle this requirement or use a **sequence** to generate unique numbers. If the performance of some queries is to be improved, one should consider creating an **index**. Indexes can also be used to enforce uniqueness on a column or a collection of columns. Alternative names can be given to database objects by using **synonyms.**

3.2.3 Constraints

Constraints enforce rules at the table level whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed. Constraints prevent the deletion of a table if there are dependencies. The following constraint types are valid:

- NOT NULL: This specifies that the column cannot contain a null value
- UNIQUE: This specifies that a column or combination of columns whose values must be unique for all rows in the table.
- PRIMARY KEY: This uniquely identifies each row of the table
- FOREIGN KEY: This establishes and enforces a foreign key relationship between the column and a column of the referenced table.
- CHECK: This specifies a condition that must be true.

3.3 Database Programming

It is very essential to pay special attention to database programming because databases are the heart and soul of many of the recent enterprise applications. For a better performing database a Database Administrator (DBA) and a specialist database programmer is needed. If database specialists are not used during a programme development cycle, database often ends up becoming the performance bottleneck.

An application that does not collate its data in a database is at stake. Programming languages reflect this trend. That is why most languages provide a robust and flexible library for database access.

Databases can be managed in a programming context by using a database engine called Relational Database Management Systems. This engine allows an interconnection between programming codes and the database that keeps the data that is reserved to be used by the codes. Database connections within the context of programming are of two forms:

- Using a built-in (or an internal) Database Management System: Since most new high level programming languages come with a built-in DBMS, programmers most times prefer designing the database and connecting to a database in the programming environment. This reduces the stress the programmer would have passed through if done otherwise. Some examples of databases that allow internal database connections are; Java, C#, Visual Basic, etc.
- Using an external Database Management System: External RDBMS are not resident in the programming environment, thus an external connection is needed in order to connect to them from a programming environment.

In this section, one of the commonly used Database Management Systems will be considered. It is called the Java Database Connectivity (**JDBC).**

3.3.1 Database Programming in Java Using JDBC

Java provides the Java Database Connectivity (JDBC) API to access different databases. JDBC, which is short for Java Database Connectivity, can be defined as "An API for the Java programming language that defines how a client may access a database." In other words, JDBC specifies the ways and methods that are needed to access a database, more specifically a relational database such as Oracle, MySQL, and others. The main aspect of JDBC is that it is a specification and not a product. So different vendors (here RDBMS vendors) provide their own implementation for the specification.

For example, the JDBC implementation for Oracle database is provided by Oracle itself and the same is the case for all others. The implementations provided by the vendors are known as JDBC Drivers. The most important point to be kept in mind is that JDBC Drivers are installed at client-side and not at server-side.

JDBC has been with the Java Standard Edition (JSE) from version 1.1. The latest version is 4 and is being shipped with JSE 6. Regardless of the version, JDBC supports four types of implementations or drivers. They are:

1. Type I or JDBC-ODBC Bridge
2. Type II or Partly Java Partly Native
3. Type III or Network Protocol Driver
4. Type IV or Pure Java Driver

The types are defined by how the driver provides communication between the application and the database server. Here are the details.

A Type I Driver is also called a JDBC-ODBC Bridge. The reason is that in this case JDBC internally makes calls to the ODBC. It is the ODBC that communicates with the database server. The job of JDBC is to provide the queries to the ODBC in a form understandable by ODBC and to deliver the result provided by ODBC to the application in a form that is understandable by the application. This Driver works mainly with the Windows platform. This is the only type of Driver that is shipped with a Java installation.

A Type II Driver uses the native API of the target database server to communicate with the server. Hence it is known as a Native Protocol Driver as well as a Partly Java Partly Native Driver. This Driver doesn't contain pure Java code as it uses the client-side API provided by the target database server. To call the client-side API of the database, it uses JNI. However, since it does not have the overhead of calling ODBC, a Type II Driver is faster than a Type I. Also, by using a Type II Driver, one can access functionalities that are specific to the database server which is being used.

A Type III Driver is also known as Network Protocol Driver. Type III Drivers target the middleware. The middleware then communicates with the database server. In essence, Type III Drivers are like Type I with the exception that Type III Drivers are completely written in Java and use the network protocol of the middleware instead of ODBC API. Type III Drivers are more secure since middleware is in the picture. In a nutshell, in Type III Drivers the conversion logic is at the middleware level and not at the client-side.

The **Type IV Driver** is known as a Pure Java Driver. It is comparable to Type II as it directly interacts with the database server. Unlike Type II, Type IV does not use native API calls. Instead the API has been written in Java by the vendor. Apart from being a pure Java implementation of database client API, a Type IV Driver delegates the processing to the database server. That means at client side no processing related to the database or SQL translation occurs. The only job the client does is connecting to the server, passing the queries and input to the database server and getting the result back through the same connection. Due to the delegation of all the processes to the server, the Type IV Driver is also known as the Thin Client Driver.

The choice of which driver to use depends on the type of application that is being developed. For example, if the application is web-based, the best option is Type IV as it releases the application server from being a

part of database transactions. In this case, the application server would only have to provide services to look up the name of the connection pool and maintain the pool. All other data-related operations would be delegated to the database server. Next we will discuss the steps involved in using JDBC.

3.3.2 Accessing the Database Using JDBC Step by Step

The best part of using JDBC for database programming is that if one has the required type of driver, regardless of the database server, the steps to connect and query the database remain more or less the same. The steps to access database for a typical relational database server include:

- loading the driver
- creating a connection
- instantiating a statement object
- retrieving a result set object
- accessing the data from the result set object.

All of these steps are the same for any database, be it Oracle or MySQL. The only change comes in the query to be passed in step four. Here are the details.

Loading the Driver

The driver, regardless of type, can be loaded in one of two ways: using the Class loader, or explicitly creating the instance. The difference between them, apart from how the driver is instantiated, is whether the Driver has to be registered explicitly or not.

Using the Class Loader

A class can be loaded at runtime using the forName() method of the Class class. The method accepts a String having the name of the class to be loaded. Also, once the class is loaded, calling the newInstance() method of Class will create a new object of the loaded class. When the forName() method is used, the Driver need not be registered explicitly. For example, to instantiate a driver of Type I using the forName method, the statement would be:

- **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();**
- **Explicitly creating the instance**
- The second way to load a driver is to instantiate it explicitly using the new operator. This is similar to that of creating a new instance of any class. However, when the driver is being explicitly

instantiated, one will have to register the driver with the runtime environment using the register() method of the DriverManager class. For example to load Type I Driver, the statements would be:

- **Driver driver=new sun.jdbc.odbc.JdbcOdbcDriver();**
- **DriverManager.register(driver);**
- Or the statements can be merged as:
- **DriverManager.register(new sun.jdbc.odbc.JdbcOdbcDriver());**

Once the driver is loaded and registered, the next step is to get a connection.

Creating a Connection

Once the driver has been loaded and registered, the next step is creating a connection with the database server. The connection is created when one creates an instance of Connection. To get an instance of Connection, the getConnection() method of the DriverManager class has to be called. In reality, Connection is an interface and when getConnection() is called, the DriverManager provides an instance of the proper implementing class to a reference variable of Connection. There are three forms of the getConnection() method which are:

- **getConnection(String url)** - the URL contains all the necessary information including the URL of the database server, user name and password.
- **getConnection(String url, Properties info)** - the URL contains only the URL to the server. The user name and password are passed as part of the Properties instance.
- **GetConnection(String url, String user, String password)** - as with the previous form, this form also contains the URL to the server in the URL parameter. The user name and password are passed as separate parameters.

The URL is of the form jdbc:<subprotocol>:subname where subprotocol refers to the protocol used by the database server and subname refers to the database to which the connection has to be made. For example, an Oracle subname refers to the tablespace within the database server. So, in order to create a connection using the ODBC data source name or DSN, the statement would be:

- Connection connection = DriverManager.getConnection ("jdbc:odbc:test",
- "test1", "test123");

- where odbc is the subprotocol and test is the DSN which points to the database to connect to. The next step is to create a statement object.

Instantiating a statement object

A statement represents a query to be executed at the database server against a database. In other words, a statement object is responsible for executing a SQL query as well as retrieving the result of the executed query. JDBC provides three types of statements based on the type of query to be executed. They are: Statement, PreparedStatement, and CallableStatement. They are based on the type of query to be executed.

Statement is the simplest type that represents a simple query. Its object can be instantiated using any of the following forms of the createStatement() method of the Connection interface:

- createStatement() -
- Returns a Statement object with default concurrency conditions.
- createStatement(int resultSetType, int resultSetConcurrency)-

Returns a Statement object with concurrency condition and type of ResultSet according to the values passed as values. The most commonly used resulSetTypes include ResultSet.TYPE_FORWARD_ONLY (indicating that the data can be read only in forward direction and once read cannot be moved back to a previous data) and ResultSet.TYPE_SCROLL_SENSITIVE (indicating that the data can be read in both forward and backward directions, and that the changes done by any other operation are visible instantly). The commonly used values for resultSetConcurrency are CONCUR_READ_ONLY (indicating that ResultSet may not be updated) and CONCUR_UPDATABLE (indicating that object may be updated).

For example, to create a Statement object that would provide a ResultSet object which is scrollable and updatable, the statement would be

Statement statement = connection.createStatement(

ResultSet.TYPE_SCROLL_SENSITIVE,

ResultSet.CONCUR_UPDATABLE);

PreparedStatement conserves resources. Whenever a query is sent to the database server, it goes through four steps: parsing the query, compiling the query, linking and executing the query. When a statement object is used to execute a query all four steps are repeated again and

again. This can create resource-hogging. The alternative to it is a PreparedStatement object. If a PreparedStatement object is used, the first three steps are performed only once at the start and in successive calls; the values are then passed to the linked query and it is executed. To create an object of PreparedStatement, any of the following forms of prepareStatement can be used:

- prepareStatement(String query)-

This form accepts a parameterised SQL query as a parameter and returns an object of PreparedStatement. "Select * from user where user_id=?" is an example of a parameterized query.

- prepareStatement(String query, int resultSetType, int resultSetConcurrency)-

This form is similar to the first form with the added options of specifying whether ResultSets are scrollable and updatable or not. The values for the two parameters are the same as those described in the Statement section.

For example, to create an instance of PreparedStatement which provides an updatable and scrollable ResultSet, the statements would be:

- String query= "Select * from user where user_id=?";
- PreparedStatement pStatement = connection.prepareStatement(
- query,
- ResultSet.TYPE_SCROLL_SENSITIVE,
- ResultSet.CONCUR_UPDATABLE);

To call procedures and functions within a database, one can use **CallableStatement**. However, if the underlying database does not support procedures and functions, then the CallableStatement object will not work. For example, versions of MySQL database prior to 5.0 did not support functions and procedures. To create an object of CallableStatement, use any of the following forms of the prepareCall() method of Connection:

- prepareCall(String query)-
  This returns a CallableStatement object that can be used to execute a procedure or function, which is passed as the query. The query is of the form "{sum(?,?)}" where sum is the function/procedure to be called.
- prepareCall(String sql, int resultSetType, int resultSetConcurrency)-

To get a ResultSet which is both updatable and scrollable, this form can be used. The resultSetType and resultSetConcurrency are same as that used with prepareStatement().

For example, to call a procedure whose name is sum, the statement would be:

- CallableStatement cStatement = connection.prepareCall(
- "{sum(?,?)}");

The next step is to retrieve the ResultSet.

Retrieving the ResultSet object

The rows retrieved by the execution of a SQL query are given back by JDBC in the form of a ResultSet object. A ResultSet contains all the rows retrieved by a query. To retrieve a ResultSet object, one can call the executeQuery() method of the Statement object. If the Statement object is of the type PreparedStatement, then executeQuery() without any argument needs to be called. If it is of the type Statement, then a SQL query will have to be passed to the method. For example, to retrieve a ResultSet from a Statement for the query "Select * from user", the code would be

- ResultSet result = statement.executeQuery("Select * from user");

The next step is to get data from the ResultSet

Accessing the data from the Resultset object

The specialty of ResultSet is that it can be iterated over as a collection and for each iteration, the data can be accessed as it is an array using an index. A ResultSet object can be iterated over using its next() method. During each iteration, one row is retrieved from the number of rows returned by the execution of the SQL query. The columns within the row can be accessed using different forms of the get() method of the ResultSet object. The forms depend upon the data-type of the column to be accessed such as getString() if the column type is varchar, getInt() if the column type is int, and so on. Mostly getString() is used to retrieve data from the columns. The argument that needs to be passed to the method is either a string containing the column name or the integer value representing the index. The index starts from 1 and not 0.

For example, if the user table has a column named "name," then the statements to retrieve the values for the "name" column would be

- while(result.next){
- System.out.println(result.getString("name));
- }

3.3.3 Using JDBC in the Real World

It is now time to learn how to develop a practical application. An application that implements what has been explained will be considered in this section:

- **GenericDAO** - Connects to the database and provides a Statement object. It is generic in the sense that it accepts a driver name and URL as an argument of constructor.
- **DataOp** - Implements database operations.
- **DAOTest** - Tests the DAO and DataOp classes.

So, here is the GenericDAO class. It accepts the driver class and URL to connect to as constructor arguments along with the user name and password.

```
package jdbctest;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
public class GenericDAO
{
Connection connection;
Statement statement;
public GenericDAO()
{
connection=null;
statement=null;
}
public GenericDAO(String driverClass,String connectionURL,String
user,String password)
{
try
{
Class.forName(driverClass).newInstance();
connection=DriverManager.getConnection(connectionURL,user,passwo
rd);
statement=connection.createStatement();
}
catch (InstantiationException e)
{
```

```
e.printStackTrace();
}
catch (SQLException e)
{
e.printStackTrace();
}
catch (IllegalAccessException e)
{
e.printStackTrace();
}
catch (ClassNotFoundException e)
{
e.printStackTrace();
}
}
public void setStatement(Statement statement)
{
this.statement = statement;
}
public Statement getStatement()
{
return statement;
}
}
```

Next is the DataOp class. It has one method that operates on the user table. This class is not generic.

```
package jdbctest;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
public class DataOp
{
Statement statement;
public DataOp(Statement statement)
{
this.statement=statement;
}
public List getUserList(String user)
{
List list=new ArrayList();
try
{
ResultSet result=statement.executeQuery("Select * from user where
user_id="'+user+'"");
```

```
 while(result.next())
{
list.add(result.getString(1));
}
}
catch (SQLException e)
{
e.printStackTrace();
list=null;
}
return list;
}
}
```

Last is the class that tests the GenericDAO and DataOp classes. Here we are passing the driver name corresponding to Type IV of MySQL JDBC driver and the corresponding URL.

```
package jdbctest;

public class DAOTest
{
public static void main(String args[])
{
//create instance of DAO class. Here we are using MySQL Type IV Driver
DAO dao=new
DAO("com.mysql.jdbc.Driver","jdbc:mysql://localhost/test","r
oot","root123");
//Creating instance of DataOp class
DataOp dataOp=new DataOp();
//calling the getUserList method for user whose id is 23
System.out.println(dataOp.getUserList("23"));

}
}
```

That completes a basic application.

4.0 CONCLUSION

In this unit you have been introduced to the fundamental concepts of Database and Database Management Systems. You have also learnt the different types of SQL statements, constraints and database programming.

5.0 SUMMARY

What you have learnt in this unit concerns:

- introduction to Information Systems which refers to a system of persons, data records and activities that process the data and information in an organisation.
- the study of information systems originated as a sub-discipline of computer science in an attempt to understand and rationalise the management of technology within organisations.
- areas of application or work which includes:
  - Information Systems Strategy
  - Information Systems Management and
  - Information Systems Development.
- types of Information Systems which Management Information Systems (MIS) or Reporting Systems, Decision Support Systems, Transaction Information System (TIS) and Expert Systems.

6.0 TUTOR-MARKED ASSIGNMENT

i. List and explain the various SQL statements.
ii. Write a short note on Database Management System and JBDC.

7.0 REFERENCE/FURTHER READING

SQL – A Practical Introduction by Akeel I. Din (2004). *An Introduction to Database Systems,* (8th ed.). C. J. Addison Wesley. ISBN: 0-321-19784-4.

MODULE 2

UNIT 1 CONCEPTUAL MODELLING AND SCHEMA DESIGN

CONTENTS

1.0 INTRODUCTION

A database conceptual model is a high-level view of database structure. Its purposes are to:

- aid understanding of the database structure by all who want to use it
- help decide where to make changes to existing systems
- provide a firm basis from which to initiate application development projects.

2.0 OBJECTIVES

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

- the meaning of conceptual modelling
- schema designs:
  - database schema designs
  - considerations for schema designs
  - schema building blocks
- Database relationships.

3.0 MAIN CONTENT

3.1 Conceptual Models

A conceptual model represents 'concepts' (entities) and relationships between them. Conceptual modelling is a well known technique of data modelling, together with logical modelling and physical modelling. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage. The aim of conceptual model is to express the meaning of terms and concepts used by domain experts to discuss the problem, and to find the correct relationships between different concepts. This is also called semantic model. The conceptual model attempts to clarify the meaning of various usually ambiguous terms, and ensures that problems with different interpretations of the terms and concepts cannot occur. Such differing interpretations could easily cause the software projects that are based on the interpretation of the concepts to fail. Once the domain concepts have been modelled, the model becomes a stable basis for subsequent development of applications in the domain. The concepts of the conceptual model can be used as basis of object-oriented design and implemented in programme code, in particular as classes in object-oriented languages. The realisation of conceptual models of many domains can be combined to a coherent platform.

Conceptual model is a term that has been used for a long time in database design. It has long been the practice in IT to describe a large system in terms of a set of interacting modules. If you can describe what each module does and describe how they interact with each other, you have a high level description of the system. Furthermore, if you describe each module in terms of sub-modules each interacting with each area, you have now a more detailed description of the system. Thus, arises the concept of having being able to zoom into parts of the system in more and more detail and being able to zoom out to see a wider and wider part of the system.

In a conceptual model, activities (the conceptual level modules) rarely interact using programmatic interfaces. Instead the two most common forms of interaction are:

- sharing data, one activity provides data, others use it.
- through the action of external entities. For instance in an airline system a crucial factor in the interaction between the check-in and departure gate activities is the passengers moving from the check-in desk to the departure gate.

What is distinctive about our approach to conceptual modelling is the use of the box bag model which is designed to show how activities interact in terms of sharing data or passing data, and to show how activities and data depend on, or are related to, external activities, things or people.

The conceptual model is often described with a class diagram in which classes represent concepts, associations represent relationships between concepts and role types of an association represent role types taken by instances of the modelled concepts in various situations. In ER (Entity Relationship) notation, the conceptual model is described with an ER(Entity Relationship) Diagram in which entities represent concepts.

3.1.1 Mapping EER to Relational Data Model

External and conceptual schemas are designed in the EER data model**.** Since there is no commercially available DBMS based on EER data model, and since most modern DBMS are based on relational data model, conceptual schema has to be mapped into the relational data model**.** Conversion is done using a mapping algorithm**.** The mapping algorithm, we are going to consider, contains seven steps

Steps of the Mapping Algorithm

1. Map regular entity types
2. Map weak entity types
3. Map relationship types with:
   - 1 : 1 cardinality ratio
   - 1 : N cardinality ratio
   - N : M cardinality ratio
4. Map super class / subclass relationships
5. Map categories
6. Map multivalued attributes
7. Define referential integrity constraints

Step 1 Map Regular Entity Types

- Map each EER schema regular entity type *E* into a relation schema with:
  - The same name (as the entity type *E)*,
  - The set of attributes containing all simple, single valued entity type *E* attributes (including simple, single valued, but excluding multivalued components of the composite attributes),
  - The same set of keys as the entity type *E*

- Result of applying step 1 onto Fig a:
  - *Department* (*DeptId*, *DeptName*),
  - *Student (StudId, StudName*)

Step 2 Map Weak Entity Types

- For each weak entity type *W*, having owners *E1*,…, *Ek* in the EER schema, create a relation schema with:
  - The same name (as the entity type *W*),
  - The set of attributes containing all simple, single valued entity type *W* attributes (including simple, single valued, but excluding multivalued components of the composite attributes), and containing the union of the primary keys of all owners *E1*,…, *Ek*,
  - The primary key, composed of the union of the primary keys of all owners *E1*,…, *Ek*, and the partial key of the entity type *W*

- Result of applying step 2 onto Fig a:
  - *Course* (*DeptId*, *CourNo*, *CourName*)

Step 3.1 Mapping Relationship Types (1:1)

- Map binary relationship type with 1 : 1 cardinality ratio between entity types *S* and *T*, according to the rules that depend on participation constrains
- Cardinality ratio 1:1 is a seldom but complicated case.

Step 3.2 Mapping Relationship Types 1: M

- Consider a binary relationship type having entity type *S* with the primary key *KeyS* on the 1 side, and entity type *T* with primary key *KeyT* on the M side
- Map the binary relationship type by inserting the primary key *KeyS* of the entity type *S* on the 1 side as the foreign key, together

with all simple, single valued relationship type *R* attributes, into relation schema *T* representing the entity type on the M side

- If the participation constraint of the entity type *T* on the M side is partial, put *Null* (*T, KeyS*) = *Yes*
- If the participation constraint of the entity type *T* on the M side is total, put *Null* (*T, KeyS*) = *Not*

Step 3.3 Mapping Relationship Types M:N

- Map each EER schema relationship type *R* with the cardinality ratio M : N into one relation schema with:
  - The same name (as relationship type *R*),
  - The set of attributes containing all simple, single valued relationship type *R* attributes (including simple, single valued, but excluding multivalued components of the composite attributes), and including the primary keys of connected entity types
  - The primary key composed of connected entity type primary keys

- Result of applying step 5 onto Fig a:
  - *Exam* (*StudId, DeptId, CourNo, Grade*)

Step 3.4 Relationship Types of the Order > 2

- Map each EER schema relationship type *R* of the order greater than two into one relation schema with:
  - The same name (as relationship type *R*),
  - The set of attributes containing all simple, single valued relationship type *R* attributes (including simple, single valued, but excluding multivalued components of the composite attributes), and including the primary keys of all connected entity types
  - The primary key that is the proper or improper subset of the union of the connected entity type primary keys.

Step 4 Mapping IS-A Hierarchies

- Mapping of a superclass / subclass (IS-A hierarchy) relationship can be done in three ways:

1. Each subclass and the superclass is mapped to one, separate relation schema
   - The superclass is mapped as a regular entity type
   - The subclass is mapped as being a Weak entity type with partial key being an empty set,

2. Each subclass is mapped into one relation schema, containing union of the superclass and this subclass attribute sets (superclass is contained in each subclass, and there is no superclass relation schema)
3. All subclasses together with the superclass are mapped into one relation schema.

Step 5 Mapping Categories

- A category is the subclass of the union of two or more superclasses
- A category is mapped to one relation schema with:
  - The same name as the category type,
  - All the single valued attributes of the category (including the attribute *CategoryType*), and
  - An artificial attribute, so called surrogate key
  - The relationship between category relation schema and superclass relation schemas is accomplished by inserting the surrogate key in each superclass relation schema.

Step 6 Mapping Multivalued Attributes

- For each multivalued attribute *V* in an entity or relationship type *T*, represented by relation schema *T* (containing all type *T* single valued attributes), create a new relation schema with:
  - the name *V*,
  - the set of attributes containing all simple, single valued attributes in *V*, and the primary key *K* of *T*, and
  - the primary key as the union of *K* and some attributes from *V*

- Result of applying step 9 on Fig a:

  - *Lecturer* (*DeptId, LectNo, LectName, HireDate*).

Step 7 Defining Referential Integrities

- Define a *referential integrity constraint* for each primary key / foreign key pair
- No referential integrity defined should represent a logical consequence of another primary key / foreign key pair constraint

The Set of Relation Schemas

S = {

- *Department* ({*DeptId, DeptName* }, {*DeptId* }),
- *Student* ({*StudId, StudName* }, {*StudId* }),
- *Course* ({*DeptId, CourNo, CourName* }, {*DeptId +CourNo* })
- *Exam* ({*DeptId, CourNo, StudId, Grade* }, {*DeptId +CourNo +,StudId* })
- *Lecturer* ({*DeptId, LectNo, LectName, HireDate* },{*DeptId + LectNo* })

}

Referential Integrity Constraints

- The set of referential integrity constraints of the example conceptual schema :
  - *IC* = {*Lecturer* [*DeptId*] Í *Department* [*DeptId*],
  - *Course* [*DeptId*] Í *Department* [*DeptId*],
  - *Exam* [*StudId*] Í *Student* [*StudId*],
  - *Exam*[(*DeptId, CourNo*)] Í *Course* [(*DeptId, CourNo*)] }

Note that *IC* doesn't contain referential integrity

- *Exam*[*DeptId*] Í *Department* [*DeptId*]
- (because it is a logical consequence of
- *Exam*(*DeptId, CourNo*)] Í *Course* [(*DeptId, CourNo*)] } and *Course* [*DeptId*] Í *Department* [*DeptId*])

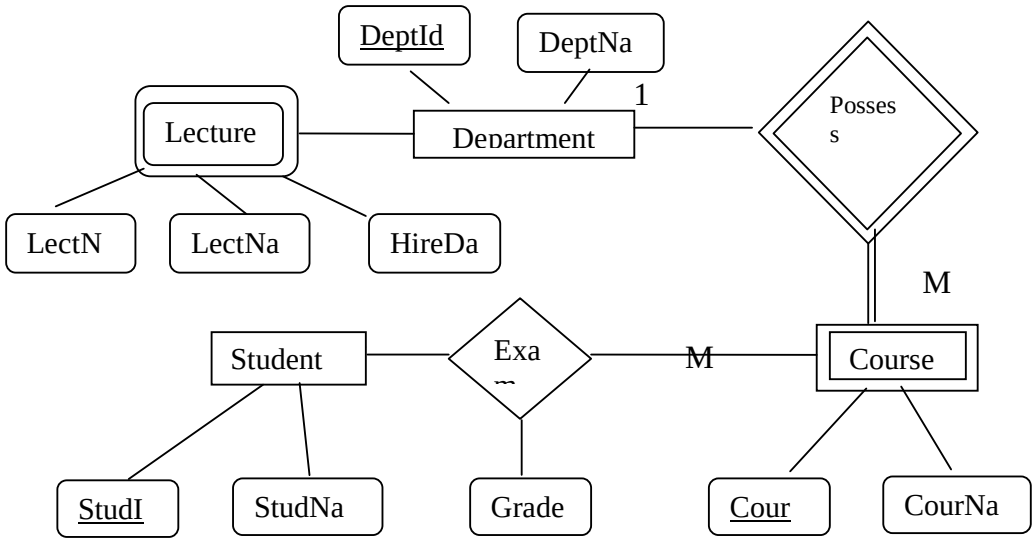


Fig. 1: Example of Conceptual Schema

3.2 Schema Design

3.2.1 Database Schema Design

The goal is to design a schema that models the problem domain as precisely as possible, such that both current known and future unknown use cases may employ the database. A possible approach is to go through the requirements and through the texts of all the use cases and put the nouns in a dictionary. Nouns that do something, i.e., are related to a verb, are candidates for database tables. Nouns that have no verb are possible attributes (table columns).

For example, consider the use case text: "The item is given an identification number." We have two nouns: item and identification number. "Item"—has the verb "is given" and is a candidate for a database table. "Identification number"—no related verb, it will probably be a column in the "Item" table.

Database constraints can be defined over a schema. The constraints are usually on the contents of database tables. When the user tries to change the contents of a table, the relevant constraints are checked first. If there is a constraint violation—the database engine raises an exception and does not allow the update. The list of some common constraints follows:

- **PRIMARY KEY** a column, where all values are unique. Example is the table Person (id, firstName, lastName, dateOfBirth, address). The primary key "id" is underlined. A table cannot have two entries with equal primary keys. Primary key may be a column combination, for example: Chair (model, color, weight, price). A certain model may exist in different colors. A better solution, however, is Chair (id, model, color, weight, price).
- **FOREIGN KEY** is a column (or several columns), whose values are limited to to the values of certain column(s) of another table, called the *referenced* table. For example: consider a table Pair(pairId, personId1, personId2). The values in "personId1" and "personId2" columns denote the Id's of two persons and we want these values to originate from the "Person" table. We do not want to have a person Id in the "Pair" table, that does not appear in the "Person" table.
- Foreign key column(s) must reference column(s),that have a PRIMARY KEY or UNIQUE constraint (see below).
- **UNIQUE** column(s) are columns whose values are unique. The difference from the primary key is that a table can have several UNIQUE constraints but at most one PRIMARY KEY constraint. Consider the table Supplier(id, name, address). The "id" column

is a primary key, but we can also define a UNIQUE constraint over (name, address). By that we emphasise the fact, that there are no two suppliers that have the same name and address.

- **CHECK** constraint is a user-defined boolean expression, that must be true at all times. For example, in a table Book(bookid, . . . , price) we may define a CHECK (*price* > 0).

Normalisation: The columns of the tables may be either key or non-key columns. In a normalised schema a non-key column appears only once, in one table in the database. This is a special case of the software engineering rule, saying that a piece of data or code shall appear only once in the entire program. The motivation for this recommendation is that when this rule is followed, an update has only to be done in one place. If the rule is not followed, an update may have to be done in a number of places. It is not the amount of work of updating in a number of different places that bothers, but the possibility that one forgets to update in one of the several required places.

3.2.2 Consideration for Schema Design

The schema design for a database affects its usability and performance in many ways, so it is important to make the initial investment in time and research to design a database that meets the needs of its users. This section is not intended to provide a detailed guide to database design, but only to present some ideas to consider in designing a database.

A well-designed schema takes into account the following considerations:

- **What are the processes of the business?**

Identify the main processes of the business; for example, taking orders for the product, filling out insurance claims, or tracking promotions. These processes are different for every business, but they must be clearly identified and defined in order to create a useful database. The people who know the processes are the people who work in the business, and interviews are essential to determine these processes.

- **What do the users want to accomplish with the database?**

The database should reflect the business, both in what it measures and tracks and in the terminology used to describe the facts and dimensions of the business. Interviews with managers and users will reveal what they want to know, how they measure the business, what criteria they use to make decisions, and what words they use to describe these things.

This information helps determine the contents of the fact and dimension tables.

- **Where will the data come from?**

The data to populate the tables in the database must be complete enough to be useful and must be valid, consistent data. An analysis of the proposed input data and its sources will reveal whether the available data can support the proposed schema.

- **What are the dimensions of the business and their attributes that will be reflected by the dimension tables?**

Independent dimensions should be represented by separate tables. If dimensions are not independent, they can be combined in a single table. Attributes are usually textual and discrete values; for example, product descriptions or geographic locations. They are used to form query constraints and to determine report breaks. The interviews and data analysis will provide guidance in setting up these tables.

- **Are the dimensions going to change over time?**

If a dimension changes frequently, it probably should be measured as a fact, not stored as a dimension.

- **What facts should be measured?**

Facts are usually numerical and continuous values; for example, revenue or inventory. Facts that are additive can be summed to produce valid measures in reports. For example, sales for each month are additive and can be summed to produce year-to-date totals. Month-end inventory balances, however, are not additive in the sense that a yearly total of month-end inventory balances is of dubious value, but a monthly average might be meaningful.

- **Is a family of fact tables needed?**

Facts that are measured with different dimensions or use different timing should be stored in separate tables. For example, a single database can be used for orders, shipments, and manufacturing. Although the facts measured in each area of the business are different, they share some but not all of the same dimensions.

- **What is the granularity of the facts?**

Granularity refers to the level of detail of the information stored in each row of the fact table. Each row should hold the same type of data. For example, each row could contain daily sales by store by product or daily line items by store.

Differing data granularities can be handled by using multiple fact tables (daily, monthly, and yearly tables) or by modifying a single table so that a *granularity flag* (a column to indicate whether the data is a daily, monthly, or yearly amount) can be stored along with the data. Also consider the amounts of data, space, and performance requirements in deciding how to handle different granularities.

- **How will changes be handled, and how important is historical information?**

If change occurs infrequently and if historical information is not very important, dimension tables can be modified to reflect only the new reality without any loss of useful data. However, if previous history is important, dimension tables can be modified to reflect both the old and new conditions. If a dimension changes frequently, perhaps it should be considered time-dependent and include a time-based attribute; for example, month, quarter, or year.

3.2.3 Schema Building Blocks

The following figures illustrate some common schema examples. Tables named Fact or Fact*x* represent fact (referencing) tables. The other tables represent dimension (referenced) tables. The following figures apply to both single-star and multistar schemas.

A schema can consist of a single dimension table.

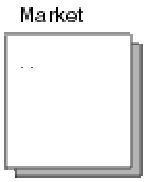


Fig. 2: Single Dimension Table

A schema can be a star schema with one fact table and one dimension table.

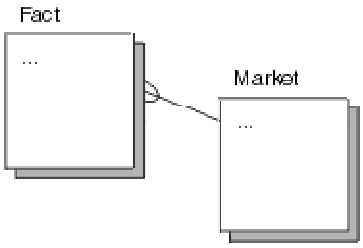


Fig. 3: Star Schema with One Dimension Table

A schema can be a star schema with one fact table and several dimension tables.

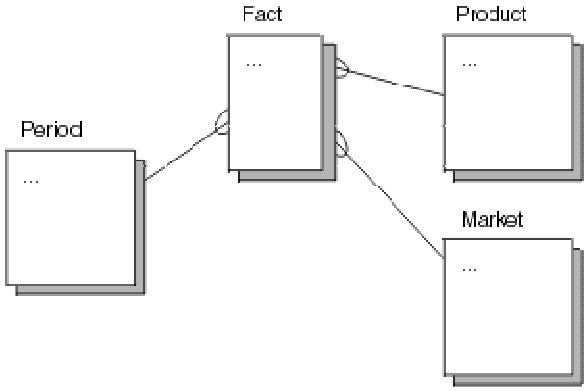


Fig. 4: Star Schema with Several Dimension Tables

A schema can be a multiple star schema, with a family of fact tables that share some, but not necessarily all, dimension tables.

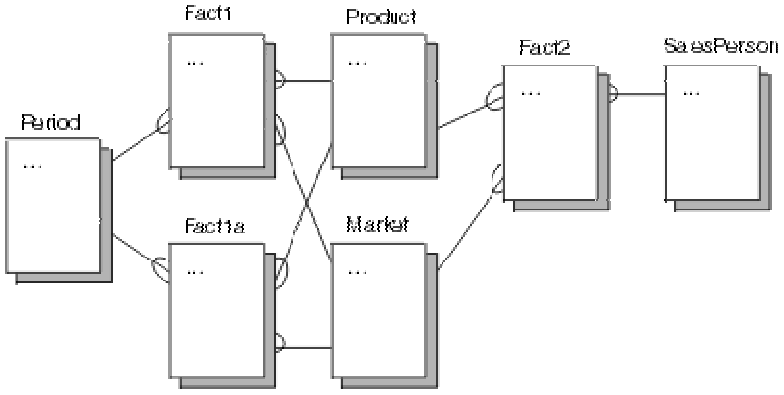


Fig. 5: Star Schema with Several Fact Tables

A schema can be an extended star schema with dimension tables that reference other dimension tables (outboard tables).

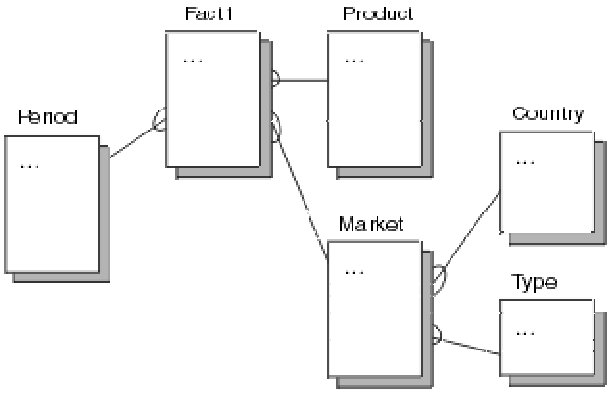


Fig. 6: Star Schema with Outboard Tables

A schema can be a star schema with a fact table that contains multiple foreign keys that reference single dimension tables.

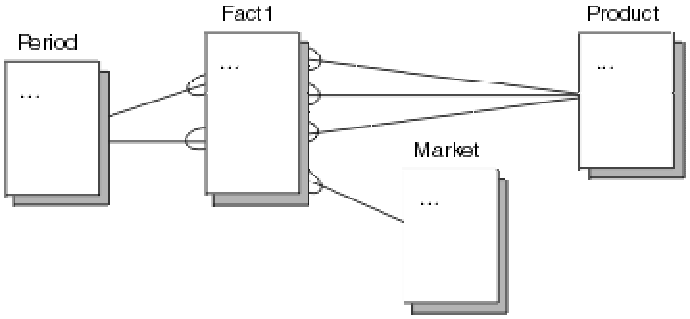


Fig. 7: Star Schema with Multiple Foreign Keys

Example: Salad dressing database

This example illustrates how the schema design affects both usability and usefulness of the database.

This database tracks the sales of salad dressing products in supermarkets at weekly intervals over a four-year period and is typical consumer-goods marketing database. The salad dressing product category contains 14,000 items at the Universal Product Code (UPC) level. Data is summarised for each of 120 geographic areas (markets) in the U.S. and for each of 208 weekly time periods spanning four years.

The salad dressing database has one fact table, sales, and three dimension tables: Product, Week, and Market, as illustrated in the following figure.

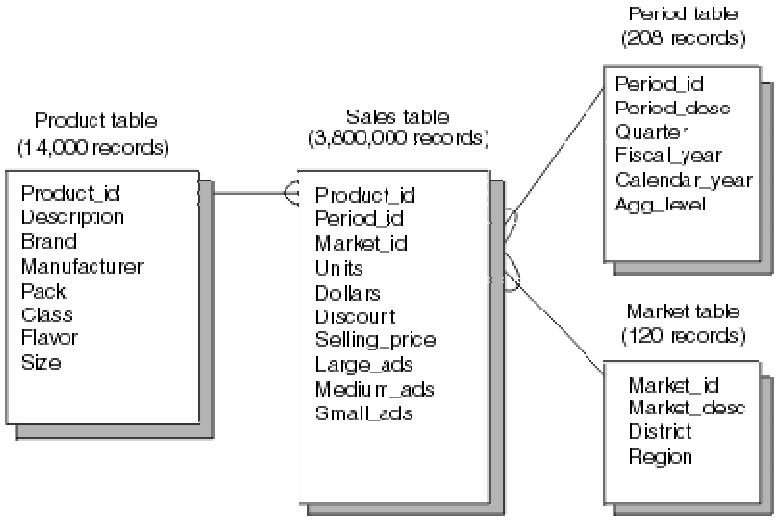


Fig. 8: Salad Dressing Database Example

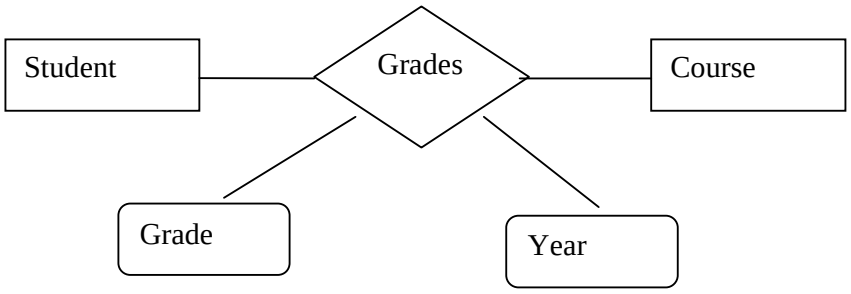
Each record in the Sales fact table contains a field for each of the three dimensions: Product, Period, and Market. The columns in the Sales table containing these fields are the foreign keys whose concatenated values give each row in the Sales table a unique identifier. Sales also contain seven additional fields that contain values for measures of interest to market analysts.

Each dimension table describes a business dimension and contains one primary key and some attribute columns for that dimension.

3.3 Database Relationships

3.3.1 Relationship and Relationship Types

A relationship is an association between two or more entities. A relationship can be represented by combining representations of associated entities and properties of their association. A relationship type is an abstract representation of an association between two or more entity sets, and a representation of a set of corresponding relationships. Graphically, a relationship is represented by a diamond-shaped box connecting associated entity types.



Recursive Relationships

A relationship may be defined between entities of the same entity set. Associations between entities of the same set are called ***recursive***. Every entity in a relationship has a ***role***

3.3.2 Enhanced ER Data Model

Enhanced ER data model brings a number of new concepts:

1. ***Superclass / subclass*** relationship, called ***IS-A*** hierarchy, as well, together with specialisation / generalisation procedures, and
2. ***Category*** (a subset of the union of two different super class entity sets)
3. ***Aggregate*** (as a representation of complex objects)

Super class / Subclass Relationship

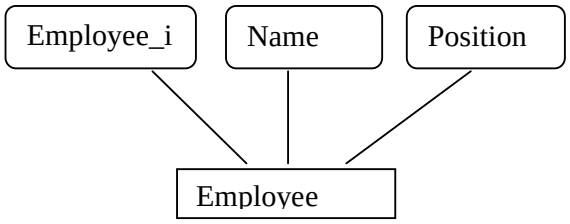
An entity class can posses groups of such entities that contain some special properties, not imminent to the other ones e.g.

- Person class contains many such groups like: students, employees, children, retirees etc
- Student class may be divided into full time and part time students, or undergraduate, graduate, master, and doctorate students.

A subclass contains only specific attributes and may participate in relationship types by its own. A subclass is a specialisation of its super class. The super class is generalisation of all its subclasses. Specialisation and generalisation are inverses of each other. An instance of the subclass is also an instance of the super class. These two instances represent the same real entity. A subclass instance inherits all the super class attribute values and all relationship participations from its image in the super class.

Super class / subclass relationship is introduced to:

- Enhance the ***semantic representation power*** of the ER data model,
- To avoid ***null values*** of the type "not applicable" in the extension of the super class, and
- To avoid impression that super class instances participate in meaningless ***subclass specific relationships***



4.0 CONCLUSION

In this unit you have been given an insight into high-level modelling of database structures and the processes involved in designing a schema that models an arbitrary problem domain as precisely as possible, you were also introduced to the various database relationship types and modelling using enhanced ER Data.

5.0 SUMMARY

- A conceptual model represents 'concepts' (entities) and relationships between them.
- A relationship is an association between two or more entities. A relationship can be represented by combining representations of associated entities and properties of their association.
- Each relationship type can be mapped as a separate relation schema, but it is considered to be a good practice to map it as:
  - A separate relation schema in the case of M: N (and 1:1) cardinality ratios, and
  - By primary key propagation in the other cases
- The relation schema that gets the propagated primary key as the foreign key, simultaneously represents an entity type and a relationship type
- There are three possible ways to map a IS-A hierarchy:
  - Each class as a separate relation schema, with subclass relation schemas inheriting super class primary key,
  - Only subclasses as separate relation schemas that inherit all the super class attributes, and
  - The super class and all the subclasses map into one relation schema
- Each set of mutivalued attributes is mapped into separate relation schema
- EER data model is introduced to provide more semantic power to UoD modelling
- EER introduces a number of new modelling constructs and a diagrammatic technique
- Set of similar UoD entities is represented by an entity type
- Set of associations between two or more entities is represented by a relationship type

- Classification is represented by super class / subclass relationship
- Category is a subset of the union of two (or more) super classes

6.0 TUTOR-MARKED ASSIGNMENT

i. Using the mapping algorithm, create a relational data model using your own example.
ii. Design a database schema using the relationship types, taking notes of the important considerations in database designs.

7.0 REFERENCES/FURTHER READING

Borgida, A. and Brachman, R.J. Conceptual Modelling with Description Logics.

Fowler, M. (1997). *Analysis Patterns, Reusable Object Models.* Addison-Wesley Longman. ISBN 0-201-89542-0.

Mogin, P. (2008). *Database Design*. Victoria University of Wellington.

Mogin, P. (2008). *Entity – Relationship Data Model*. Victoria University of Wellington.

UNIT 2 FUNCTIONAL DEPENDENCY

CONTENTS

1.0 INTRODUCTION

This unit will describe the concepts of functional dependency as well as the concepts of relational algebra.

2.0 OBJECTIVES

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

- state the meaning, classification and properties of functional dependency
- define the closure of a set of functional dependency as well as the algorithm to determine the closure
- outline and identify the basic types of keys after performing closure of asset of functional dependency
- explain database normalisation, determine if a relation is in 1NF, 2NF, 3NF or BCNF and how to decompose into 3NF or BCNF
- describe the concept of relational algebra.

3.0 MAIN CONTENT

3.1 Functional Dependence

A functional dependency is a constraint between two sets of attributes in a relation from a database. Given a relation R, a set of attributes X in R is said to functionally determine another attribute Y, also in R (written X→Y) if and only if each X value is associated with precisely one Y value. Customarily we call X the determinant set and Y the dependent attribute. Thus, given a tuple and the values of the attributes in X, one can determine the corresponding value of the Y attribute. For the purposes of simplicity, given that X and Y are the sets of attribute R, X→Y denotes that X functionally determines each of the members of Y- in this case Y is known as the dependent set. Thus, a candidate key is a minimal set of attributes that functionally determine all of the attributes in a relation.

A functional dependency FD: X→Y is called trivial if Y is a subset of X. The determination of functional dependencies is an important part of designing databases in the relational model and in database normalisation and denormalisation. The functional dependency along with the attribute domains are selected so as to generate constraints as much data inappropriate to the user domain from the system as possible.

For example, suppose one is designing a system to track vehicles and the capacity of their engines. Each vehicle has a unique vehicle identification number (VIN). One could write VIN→ Enginecapacity because it would be inappropriate for a vehicle's engine to have more than one capacity. However, Enginecapacity→ VIN is incorrect because there could be many vehicles with the same engine capacity.

3.2 Classification of Functional Dependency

Functional dependency can be classified as follows:

3.2.1 Fully Functional Dependency

It indicates that if A and B are attributes of a table, B is fully functionally dependent on A if B is functionally dependent on A but not on A but on any proper subset of A. E.g. StaffID→ BranchID

3.2.2 Partial Functional Dependency

It indicates that if A and B attributes of a table, B is partially dependent on A if there is some attribute that can be removed from A and yet the dependency still holds. Say for example; consider the following functional dependency that exists in the table staff: StaffID, Name BranchID.

BranchID is functionally dependent on a subset of A (StaffID, Name) namely StaffID.

3.2.3 Transitive Functional Dependency

A condition where A, B and C are attributes of a table such that if A is functionally dependent on B and B is functionally dependent on C then C is transitively dependent on A via B. Say for example, consider the following functional dependency that exists in the staff table and Branch table:

- StaffID→ Name, Sex, Position, Sal, BranchID, Br_Address
- BranchID→ Br_Address

So, StaffID attribute functionally determines Br_Address via BranchID attribute.

3.3 Properties of Functional Dependency

Given that X, Y and Z are sets of attributes in a relation R, one can derive several properties of functional dependency. Among the most important are Armstrong's axioms which are used in database normalisation:

1. Subset property(Axiom of reflexivity): If Y is a subset of X, then X→ Y
2. Augmentation(Axiom of augmentation): If X→ Y, then XZ→ YZ
3. Transitivity(Axiom of transitivity): If X→ Y and Y→ Z then X→ Z

From these rules, we can derive these secondary rules:

1. Union: If X→ Y and X→ Z, then X→ YZ
2. Decomposition: If X→ YZ then X→ Y and X→ Z
3. Pseudo-transitivity: If X→ Y and YZ→ W then XZ→ W

3.4 Closure of a Set of Functional Dependency

The closure of a set of dependencies is the set of all possible dependencies that can be derived from it. To get the closure of a set of functional dependencies, one needs to consider all functional dependencies that hold. Given a set F of functional dependencies, we can prove that certain other ones also hold. We say these ones are also logically implied by F. We denote the closure of F by F+.
Suppose we have a scheme R= {A, B, C} and FD: A→ C
B→ D

To compute F+, we can use some rules of inference called Armstrong's Axioms as stated in the properties of functional dependency above. These rules are sound because they do not generate any incorrect functional dependencies. They are also complete as they generate all of F+. To make life easier we can also use the secondary rules derivable from Armstrong's Axioms.

3.4.1 Algorithm to Determine Closure

1. Let C→ CA
2. Let the next dependency be A→B. If A is in CA and B is not, then C→CA +B
3. Continue step 2 until no new attributes can be added to CA.

The result of this algorithm is CA that is equal to C+. The above algorithm may also be used to remove redundant dependencies. For example, to check if C→ A is redundant, we find closure of C without using C→A as redundant.

For example, if we wish to determine the closure of a relation using the following functional dependencies: A→ B, C
C→ D
A, D→ F

A+ is all attributes that can be derived from A. Applying the above algorithm:

- we first initialize A+ =A
- because A→ B, C add BC to A+

- because C is in A+ and C→ D add D to A+
- because A and D are in A+, add F to A+

Therefore A+ = A, B, C, D, F

3.5 Keys

There are basically five kinds of keys which are sets of attributes of a relation functionally depending on one or more attributes of the relation:

3.5.1 Super Key

A super key is defined in the relational model as a set of attributes of a relation for which it holds that in all instances of the relation there are no two distinct tuples that have the same values for the attributes in this set. Equivalently, a super key can also be defined as those sets of attributes of a relation upon which all attributes of the relation are functionally dependent.

3.5.2 Primary Key

A primary key can be said to be a super key that also serves as part of the determinants in the functional dependencies.

3.5.3 Candidate Key

A candidate key is a subset of attributes that is unique for every tuple. Therefore, a valid candidate key determines all other attributes in the tuple. Simply said, a candidate key is the minimal super key. A candidate key can also be a primary key.

3.5.4 Secondary Key

A secondary key is a key that is part of the candidate key and not part of the primary key.

3.5.5 Alternative Key

An alternative key is a key that is part of the super keys and not part of the candidate keys.

3.5.6 Keys Example

Consider the relation R (A, B, C, D) having its functional dependencies to be B→ C
B→ D

Find

- all the super key(s)
- primary key(s)
- candidate key(s)
- secondary key(s)
- alternative key(s)

Solution:

To find the keys, it is advisable to first compute the closure of the relation. The number of different attributes that can be derived from the relation can known by using the formula; 2^n – I. which implies that the numbers of the different attributes = I5 and so, the attribute are; A, B,C, D, AB,AC, AD, BC, BD, CD,ABC,ABD, ACD, BCD and ABCD.

The next thing to do now is to determine the closure for each attribute:

- A+ =A
- B+ =BC, BCD B→ C, B→D
- C+ =C
- D+ =D
- AB+ =ABC, ABCD AB→ C, AB→D
- AC+ =AC
- AD+ =AD
- BC+ =BCD BC→ D
- BD+ = BCD BD→ C
- CD+ =CD
- ABC+ = ABCD ABC→ D
- ABD+ = ABCD ABD→C
- ACD+ =ACD
- BCD+ =BCD
- ABCD+ =ABCD

From here, it is much easier for us to find all the keys.

- Since super keys are those sets of attributes of a relation upon which all attributes of the relation are functionally dependent. Therefore, the super keys are AB,ABC,ABD
- Primary key = none
- Candidate key=min(super key)= min(AB,ABC,ABD) =AB
- Secondary key = candidate key – primary key =AB
- Alternative key = super key- candidate key=ABCD- AB =CD

3.6 Database Normalisation

Database normalisation is a stepwise formal process that allows us to decompose database tables in such a way that both data redundancy and update anomalies are minimised. It makes use of functional dependencies that exist in a relation and the primary key/candidate keys in analysing the relation.

Three norm forms were initially proposed called First norm form (INF), Second norm form (2NF) and Third norm form (3NF). Subsequently R.Boyce and E.F. Codd introduced a stronger definition of 3NF called Boyce Codd norm form (BCNF).With the exception of INF, all these norm forms are based on functional dependencies among the attributes of a table. Higher norm forms that go beyond BCNF were introduced later such as 4NF and 5NF. However, these later norm forms deal with situations that are very rare.

- **INF*:*** A relation is said to be in INF if every functional dependency contains only atomic values i.e. intersection of each functional dependency should contain one and only one value. We often assume that it always holds**.**
- **2NF*:*** A relation is said to be in 2NF if it is in INF and there are no partial dependencies i.e. every non-primary key attribute of the relation is fully dependent on the primary key.
- **3NF*:*** A relation is said to be in 3NF if every FD: X→ A that holds over the relation, X is a super key or A is part of some keys in the relation.
- **BCNF*:*** A relation is in BCNF if for every FD: X→ A that holds over the relation, X is a super key and functional dependency must be trivial.
- **4NF:** 4NF is stronger than BCNF as it prevents relation from containing non-trivial multivalued functional dependencies (MVD) and hence data redundancy. The normalisation of BCNF tables to 4NF involves the removal of MVDs from the table by placing attributes in a new relation along with the copy of the determinant(s).
- **5NF:** 5NF is also called Project Join Norm Form (PJRF) and it specifies that a 5NF table has no join dependency.

3.6.1 Example

Using the relation R above, determine whether the relation is in INF, 2NF, 3NF or in BCNF.

Solution:

- Looking at the functional dependencies that were been given, it is very hard to deduce what B, C or D represents and more so, we often assume that INF holds over any given relation. Therefore, the relation is INF.
- Combining the two functional dependencies, we can see that the result is not fully dependent. Therefore, the relation is not in 2NF.
- Since a relation that is in 3NF must have its determinant to be a super key or the attribute that was been determined is part of some keys in the super key. So, the relation is in 3NF because C and D form a part of the super keys which are the alternative keys.
- The relation violates BCNF because the determinant is not a super key and more so, the new functional dependencies that are just been derived are non-trivial.

3.7 Decomposition

As we have seen, a relation in BCNF is free of redundancy and a relation schema in 3NF comes close. If a relation schema is not in one of these normal forms, the FDs that cause a violation can give us insight into the problems. The main technique for addressing such redundancy-related problems is decomposing a relation schema into relation schemas with fewer attributes.

A decomposition of a relation schema R consists of replacing the relation schema by two or more relation schema that each contains a subset of the attribute of R and together include all the attributes in R. Intuitively, we want to store the information in any given instance of R by sorting projections of the intense.

We begin with the relation example from above. This relation has attributes ABCD and two FDs: B→C and B→ D .Assuming B is not a key and D is not part of any key. The second FD causes a violation of 3NF.

Our decision to decompose ABCD into ABC and BD, rather than say AB and BC was just a good guess. It was guided by the observation that the dependency B→ D caused the violation of 3NF; the most natural way to deal with this violation is to remove the attribute D from the schema. To compensate for removing D from the main schema, we can add a relation BD because each B value is associated with at most one D value according to the FD: B→ D

A very important question must be asked at this point. If we replace a legal instance r of relation schema ABCD with its projections on AC (r1) and BD (r2), can we recover r from r1 and r2? The decision to decompose ABCD into ABC and BD is equivalent to saying that we will store instances r1 and r2 instead of r. However, it is the instance r that captures the intended entities or relationships. If we cannot compute r from r1 and r2, our attempt to deal with redundancy has effectively thrown out the baby with the bathwater.

3.7.1 Lossless-Join Decomposition

Let R be a relation schema and let F be a set of FDs over R. A decomposition of R into two schemas with attribute sets X and Y is said to be a lossless-join decomposition with respect to F if for every instance r of R that satisfies the dependencies in F, Π x(r) ∞ Π γ(r) = R or let R be a relation and F be a set of FDs that hold over R. The decomposition of R into relations with attribute sets R1 and R2 is lossless if and only if F+ contains either the FD R1 $\cap$ R2 $\rightarrow$ R1 or the FD R1 $\cap$ R2
$\rightarrow$ R2.

In other words, the attributes common to R1 and R2 must contain a key for either R1 or R2. If a relation is decomposed into two relations, this test is a necessary and sufficient condition for the decomposition to be lossless-join. If a relation is decomposed into more than two relations, an efficient algorithm is available to test whether or not the decomposition is lossless, but we will not discuss it.

Consider the relation again. It has attributes ABCD and the FD B$\rightarrow$ D cause a violation of 3NF. We dealt with this violation by decomposing the relation into ABC and BD. Since R is common to both decomposed relations and B$\rightarrow$ D holds, this decomposition is lossless-join.

3.7.2 Decomposition into BCNF

We now present an algorithm for decomposing a relation schema R into a collection of BCNF relation schemas:

- Step 1. Suppose that R is not in BCNF. Let X be a subset of R, A be a single attribute in R, and X$\rightarrow$ A be an FD that causes a violation of BCNF. Decompose R into R-A and XA.
- Step 2. If either R-A or XA is not in BCNF, decompose them further by a recursive application of this algorithm.

R-A denotes the set of attributes other than A in R, and XA denotes the union of attributes in X and A. Since X$\rightarrow$ A violates BCNF, it is not a

trivial dependency. Further, A is a single attribute. Therefore, A is not in X; i.e. X∩A is empty. Thus, each decomposition carried out in step (1) is lossless-join.

The set of dependencies associated with R-A and XA is the projection of F onto their attributes. If one of the new relations is not in BCNF, we decompose further in step (2). Since decomposition results in relations with strictly fewer attributes, this process will terminate, leaving us with a collection of relations schemas that are all in BCNF. Further, joining instances of the relations obtained through this algorithm will yield precisely the corresponding instance of the original relation.

Consider the contracts relation with attributes CSJDPQV and key C. We are given FDs JP→ C and SD→ P. By using the dependency SD→ P to guide the decomposition, we get the two schemas SDP and CSJDQV. SDP is in BCNF. Suppose that we also have the constraint that each project deals with a single supplier J→ S. This means that the schema CSJDQV is not in BCNF. So we decompose it further into JS and CJDQV. C→ JDQV holds over CJDQV; the only other FDs that hold are those obtained from this FD by augmentation and therefore all FDs contain a key in the left side. Thus, each of the schemas SDP, JS and CJDQV is in BCNF and this collection of schemas also represents a lossless-join decomposition of CSJDQV.

3.7.3 Decomposition into 3NF

Clearly the approach that we outlined for lossless-join decomposition into BCNF will also give us lossless-join decomposition into 3NF but this approach does not ensure dependency preservation.

A simple modification, however, yields a decomposition into 3NF relations that is lossless-join. Before we describe this modification, we need to introduce the concept of a minimal cover for a set of FDs.

3.8 Minimal Cover for a Set of FDs

A minimal cover for a set F of FDs is a set G of FDs such that:

- every dependency in G is of the form X→ A, where A is a single attribute.
- the closure F+ is equal to the closure of G+.
- if we obtain a set H of dependencies from G by adding one or more dependencies, or by deleting attributes from a dependency in G, then F+ is not equal to H+.

Intuitively, a minimal cover for a set F of FDs is an equivalent set of dependencies that is minimal in two respects:

- every dependency is as small as possible i.e. each attribute on the left side is necessary and the right side is a single attribute.
- every dependency in it is required in order for the closure to be equal to F+.

As an example, let F be the set of dependencies: A→ B, ABCD→ E, EF→ G, EF→ H and ACDF→ EG.

First, let us rewrite ACDF→ EG so that every side is a single attribute: ACDF→ E and ACDF→ G.

Next consider ACDF→ G. This dependency is implied by the following FDs: A→ B, ABCD→ E and EF→ G.

Therefore, we can delete it. Similarly, we can delete ACDF→ E. Next consider ABCD→ E. Since A→ B holds, we can replace it with ACD→ E. Thus, a minimal cover for F is the set: A→ B, ACD→ E, EF→ G and EF→ H.

The preceding example suggests a general algorithm for obtaining a minimal cover of a set of FDs:

- put the FDs in a standard form: obtain a collection G of equivalent FDs with a single attribute on the right side using decomposition axiom.
- minimise the left side of each FD: for each FD in G, check each attribute in the left side to see if it can be deleted while preserving equivalence to F+.
- delete redundant FDs: Check each remaining FD in G to see if it can be deleted while preserving equivalence to F+.

Note that the order in which we consider FDs while applying these steps could produce different minimal covers, there could be several minimal covers for a given set of FDs.

SELF-ASSESSMENT EXERCISE

i. Given SUPPLIERS(S, A, I, P) and
   $F = \{S \rightarrow A, S + I \rightarrow P\}$.
   Show that S + I is a key.
ii. Prove the following: If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

4.0 CONCLUSION

In this unit, you have been introduced to the fundamental concepts of a functional dependency, the classification of functional dependency, properties of functional dependency, closure keys, database normalisation, decomposition and minimal cover for a set of functional dependencies.

5.0 SUMMARY

What you have learnt in this unit concerns:

- a functional dependency which comes in various classifications namely; fully, partial and transitive functional dependency.
- identification of keys which include primary, candidate, secondary, alternative and super keys.
- data normalisation which consists of 1NF, 2Nf, 3NF, 4NF, 5NF and the BCNF
- decomposition.

6.0 TUTOR-MARKED ASSIGNMENT

Consider the relation (A, B, C) having functional dependencies: B $\rightarrow$C
B$\rightarrow$ D

Find:

i. Super keys
ii. Primary keys
iii. Candidate keys
iv. Alternative keys
v. Secondary keys

7.0 REFERENCES/FURTHER READING

Functional Dependencies, Barbara L. Marcolin (1999). http://www.lightenna.com/book/export/s5/155

J.D. Ullman and J. Widom. (2002). *A First Course in Database Systems* (2nd ed.). Prentice Hall.

Teorey, T.J. *Database Modelling and Design* (3rd ed). University of Michigan.

REGULAR EXPRESSIONS

CONTENTS

1.0 INTRODUCTION

In this unit, you will be introduced to the craft powerful time-saving regular expressions, starting with the basic concepts, so that you can follow through this unit even if you know nothing at all about regular expressions. Regular expression engines are discussed briefly.

2.0 OBJECTIVES

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

- write the regular expression equivalence of a character or string
- convert any regular expression to its character or string equivalence.

3.0 MAIN CONTENT

3.1 General Introduction

Basically, a regular expression is a pattern describing a certain amount of text. The **regex** (*regular expressions*) are very useful for programmers. Using this regular expression, you can describe every string that presents to it is inside certain regularity. Regular expressions can be used to search for patterns and, once found, to modify the patterns in complex ways. They can also be used to launch programmatic actions that depend on patterns.

Their name comes from the mathematical theory on which they are based. In writings, it is often abbreviated to regex or regexp. In this unit, regex is used because it is easy to pronounce the plural "regexes".

3.2 Regular Expressions

The easiest and most convenient way to define "Regular Expression" is to say: "They are search patterns to match characters in strings." All regex are case sensitive unless told not to be so.

Think about having a web page with a form with the following fields:

- **Name**
- **Surname**
- **E-mail**
- **Phone number**

Once you have filled in the format and sent the data to the script, it is very important to check if they are correct.

You need to define the specific area:

- **Name**: It is made by one word only and by alphabetical letter.
- **Surname**: It is made by one or more words that can be made only by alphabetical letter
- **Email**: It is made by 3 parts: the first one is made by alphanumeric issues, underscore (_) and period (.), there is the second one made by alphanumeric issues and dash, followed by a period, which is always followed by 2 to 4 alphabetical letters. This one is compulsory.
- **Phone number**: made by 2 parts. It is divided by a dash.

All the following fields owe a specific regularity and there is a specific expression that defines them. These are the expressions:

- **name**: [a-zA-Z]*
- **surname**: [a-zA-Z']+
- **email**: [a-zA-Z0-9_\.]+@[a-zA-Z0-9-]+\.[a-zA-Z]{0,4}
- **phone number**: [0-9]+\-[0-9]+

3.3 Elements/Metacharacters of Regular Expressions

Regexes use characters with a special meaning: metacharacters. To find them literally they must be escaped. This is done with a preceding backslash: * + ? . ( ) [ ] { } \ / | ^ $

3.3.1 Classes

The operator sign **[]** is made of two square brackets. Several constant characters can be inserted into this metacharacter. Through this metacharacter it is possible to characterise a single occurrence of one of the present characters to its inside, if it is inserted like normal characters or using constants. For example, the class **[a]** represents the single occurrence of the **a** character and allows to us verify if it is inside a string and/or executing some operation on it. Otherwise, the class **[abcd]** represents the single occurrence in one of the four characters present inside it and permit to verify if they are present in the strings and execute operations on them.

3.3.2 Range Operator

The sign **-** is an operator that permits us to identify a range, for example:

- **a-z** for all the lower case letters
- **A-Z** for all the upper case letters
- **0-9** for all the numbers.
- **[a-fA-F0-9]** individualizes all the **figures** and the letters from **a** to **f** (lower case and upper case) all the characters that are inside an hexadecimal figure.

3.3.3 Class Repetition Operators

The first one we are going to analyze is the star *****. It is the one that can verify how many times a class is repeated inside a string and to select the entire consecutive occurrence. For example, the following regular expression **[a-z]*** selects in a string all the consecutive occurrence of

alphabetical letters. This operator considers an empty set as positive solution.

Very similar to the star is the plus **+** operator that works in the same way, but it verifies if a class it is repeated inside a string one or more times. This operator considers an empty set as a negative solution.

Another operator is made by 2 **{}** braces, in their inside it can be a number **{3}** or a numerical range **{12,58}**. The first one individualises all the repetitions of 3 characters that verify the class. The second one individualises from 12 to 58 repetitions of characters that verify the class. For example **[0-9]{3,4}\-[0-9]{7}** individualises all the telephone numbers in an area code made by 3 or 4 figures and a suffix of 7 figures.

3.3.4 Backslash Operator

The backslash \ operator is used before a character if it is an operator and it does not consider it as character, if we put it before a letter it is a constant. The dash is used to indicate a range and therefore if we want to use as a character we have to write it down this way: **\-**

Now you can understand the regex that we used to verify the email:

- **[a-zA-z0-9_\.]+@[a-zA-Z0-9-]+\.[a-zA-Z]{0,4}**

And the one for the telephone number:

- **[0-9]+\[0-9]+**

3.3.5 Repetition Operator's Specific Characteristics

One of the characteristics of the repetition operators is selecting everything that is related to the expressions. This characteristic could be counterproductive sometimes. If we want to eliminate from an html page all the tags, we can use the following regular expression:

- **<.+>**

If this operation does not satisfy our demand we need to use one of the following methods:

- **<.+?>**
- **<[^<>]+>**

The first one makes the repetition operator less strong and it makes it stop in the first part of the closing character.

The second individuates inside a strings or series of characters that start with < followed by any characters different from < and > followed by an >.

3.3.6 Class Denying

Let us focus on a different problem. Let us suppose having a story and we need to individuate all the sentences present inside it. If inside the story the period is used only at the end of the sentences, we have to deny a class in order to individuate a sentence in an easier way.

- **[^\.]+**

The ^ sign if it is put immediately after the first bracket of a class, it denies the class.

3.3.7 The Period

The period is a constant, and if it is inserted in a regex it is equivalent to a class that has all the characters but the "new line". This is just an example to better understand the function of the period

- **c.s.**

The former regex individuates all the 4 characters sequences that start with c followed by any characters and then followed by a and s. It creates different combinations such as:

- **case**
- **cosa**
- **cose**
- **c%s9**
- **cÂ£sl**

3.3.8 Alternancy Operator

This operator is in form of a pipe | which has the same function of the OR. For example, the regex **george|stuart** individuates inside a string the word george or the word stuart.

3.3.9 Anchors

Another problem can be faced if you need to modify one or more elements inside a CSV (comma-separated value) database, a textual database in which fields are separated by commas and which records are

divided by a new line. The following database is an example that represents the daily gain of an expense made by three friends.

- 12â,¬, 50â,¬, 70â,¬
- 30â,¬, 46â,¬, 68â,¬
- 15â,¬, 52â,¬, 73â,¬
- 16â,¬, 30â,¬, 85â,¬

If one day one of the friends was banned from expense, his data would not be useful anymore and could be necessary to remove them. If there were thousands data the regex would be the fastest solution. If the data of the banned friend is the ones in the third column, the fastest solution to remove them would be to eliminate the entire occurrence in the following regex:

- **,[0-9]*â,¬$**

The **$** character does not identify any characters, but a position, the end of a line. Therefore the former regex finds all the consecutive characters series that start with a comma followed by some numbers, followed by the â,¬, followed by the ending of a line.

3.3.10 Groups

We can consider a characters series as a single group, we can operate on it using some of the operators that build the regex. We could find out inside a text a code we do not know its length, which is composed by 5 numbers followed by a letter, followed by 5 numbers followed by a letter etc etc…until it terminates with a new line. There is only a solution to find this code; we need to use a group. In this example the group is made by a class which has numbers only repeated five times, followed by an only letters class. This group has to be repeated at least once and must end with a new line. It could be written down as:

- **([0-9]{5}[a-zA-Z])+$**

The regex creates this effect:

- My secret code is **12345T45345R12343F34567j**
- Phil’s secret code is **34526g54638j92725K63723H72829D12345I**
- 12345T45345R12343F34567j is not phil’s code.

3.3.11 Question Mark

In the groups the question mark can be used to avoid the match memorisation. We have already seen that question mark could be used to restrict the repetitions. Now we will see that there exist many different functions for this simple character.

The first function makes a group optional, as you can see in the following example:

- **michael (owen)?**

In the former regex the group **(owen)** is made optional and therefore it will be possible to select both the simple occurrence of the word **michael** and the occurrence of the word couple **michael owen**.

The second function is being an **anchor**. The question mark can also be used in the groups as a keeper, to individuate it as a position inside the text. Example:

- **michael(?=owen)**

The former regex selects the word **michael** in a text only if it is followed by the group **(owen)** that will not be selected.

You can also use the question mark to individuate the absence of a position. For example the following function selects the word **michael** only if it is not followed by the group **(owen)**:

- **michael(?!owen)**

3.4 Regular Expressions Engines

A regular expression “engine” is a piece of software that can process regular expressions, trying to match the pattern to the given string. Usually, the engine is part of a larger application and you do not access the engine directly. Rather, the application will invoke it for you when needed; making sure the right regular expression is applied to the right file or data.

As usual in the software world, different regular expression engines are not fully compatible with each other. It is not possible to describe every kind of engine and regular expression syntax (or “flavor”). The Perl 5, regex flavor is the most popular one, and deservedly so. Many more recent regex engines are very similar, but not identical, to the one of Perl 5. Examples are the open source PCRE engine (used in many tools and

languages like PHP), the .NET regular expression library, and the regular expression package included with version 1.4 and later of the Java JDK. There are certain important differences in regex flavor.

4.0 CONCLUSION

In this unit, you learnt about regular expressions which are search patterns to match characters in a string, their metacharacters and elements, we also considered regular expressions search engines. You can also transform characters or strings to their regular expression equivalence.

5.0 SUMMARY

What did we learn in this chapter?

- Regular expressions search for any character. "er" looks for exactly these letters in that order. All regex are case-sensitive unless told not to be so.
- Regexes use characters with a special meaning: metacharacters. To find them literally they must be escaped. This is done with a preceding backslash: * + ?. ( ) [ ] { } \ / | ^ $
- A dot "." is used to a single unknown character. It is a metacharacter.
- There are metacharacters which symbolize groups of characters like \d for digits ([0-9]) \D for non-digits ([^0-9]) \w for alphanumeric characters ([a-zA-Z0-9_]) \W for non-alphanumeric characters ([^a-zA-Z0-9_])
- It is possible to define your own set of character-classes by using square brackets e.g. "[A-Z]". A ^ as first character in the square bracket negates the class.

6.0 TUTOR-MARKED ASSIGNMENT

i. Transform the following regular expressions into their character or string equivalence:
   a. <[a-zA-Z0-9_\.]*\@[a-zA-Z0-9-]*\.[a-zA-Z]*>
   b. ([a-zA-Z][0-9]|[0-9][a-zA-Z])
   c. ^[\!\@\#\$\%\^\&*\(\)_\+\.\{\}\[\]\\\|\:\;\’\,\.\?\/a-zA-Z0-9]*[a-zA-Z0-9]+[\!\@\#\$\%\^\&*\(\)_\+\.\{\}\[\]\\\|\:\;\’\,\.\?\/a-zA-Z0-9]*$
   d. (!@#$%^&*()_+.{}[]\|:;’”,.?/)

ii. Transform the following strings and statements to their regular expression equivalence:

a. "I have got 7 telephone numbers, but this is my cell-phone: 0004578907″.
b. "I live in the 5^{th} house on 85^{th} street".

7.0 REFERENCES/FURTHER READING

Jeffrey, E. F. Fried. *Mastering Regular Expressions*.

Jan, Goyvaerts. *Regular Expressions*.

UNIT 4 RELATIONAL ALGEBRA

CONTENTS

1.0 INTRODUCTION

In this unit, you will be introduced to Query languages in particular Relational Algebra which is a procedural query language. Basically, a query language allows users of a database to request information from the database. A complete query language has facilities for inserting and deleting tuples from relations as well as for modifying existing tuples. It consists of set operations which are either unary or binary, meaning that either one or two relations are operands to the set operations.

2.0 OBJECTIVES

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

- explain what a query language does
- differentiate between procedural and non procedural languages
- have a good knowledge of fundamental operations in Relational Algebra
- use basic operators in Relational Algebra to form queries
- construct Relational Algebra expressions to perform operations.

3.0 MAIN CONTENT

3.1 Introduction to Query Language

A query language is a language in which a database user requests information from the database. Most query languages are on a higher-level than standard programming languages like C and Java. Query languages fall into a category of languages known as 4GL.

Query languages can be broadly categorized into two groups: procedural languages and nonprocedural languages.

- A procedural query language requires the user to specify a sequence of operations on the db to compute the desired result. (User specifies how and what.) e.g. Relational Algebra
- A nonprocedural query language requires the user to describe the desired result without needing to specify the sequence of operations required to obtain the result. (User specifies only what.) E.g. Relational tuple calculus, Relational domain calculus.

3.2 Relational Algebra

The *relational algebra* is a procedural query language. It consists of set operations which are either unary or binary, meaning that either one or two relations are operands to the set operations.

- Each of the set operations produces a relation as its output.
- Relational Algebra is an important tool in query optimisation whereby a SQL query is transformed into relational algebra equivalent, optimised and executed.
- There are five fundamental operations in the relational algebra and several additional operations which are defined in terms of the five fundamental operations.
- There is also a *rename* operation which is sometimes referred to as a fundamental operation; we will save this one for a little while.
- The five fundamental operations are: *select, project, union, set difference,* and *Cartesian product.* We will examine each operation individually before combining operations into more powerful expressions.

3.3 Operations in Relational Algebra

The five fundamental operations are: select, project, union, set difference, and Cartesian product.

- There are several additional (redundant) operations that have been defined in the relational algebra. The most common of these include: intersection, natural join, division, semi-join, and outer join.
- We will examine each operation individually before combining operations into more powerful expressions.

3.3.1 Selection Operator

Type: unary
Symbol: Greek letter sigma, σ
General form: σ(predicate)(relation instance)
Schema of result relation: same as operand relation
Size of result relation (tuples): ≤ |operand relation |

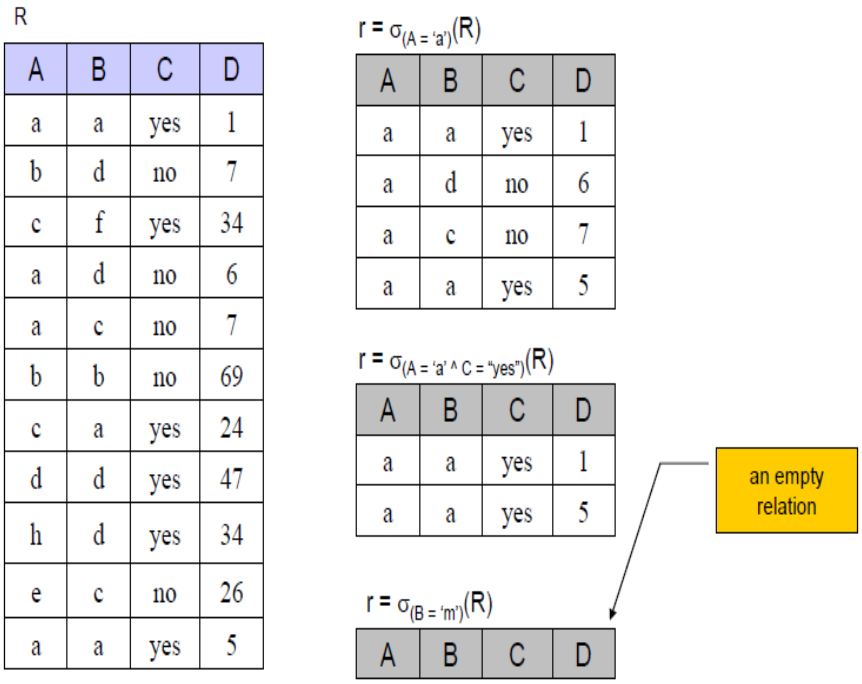
Examples:

- σ(major = “CS”)(students)
- σ(major = “CS” and hair-color = “brown”)(students)
- σ(hours-attempted > hours-earned)(students)

The select operation selects tuples from a relation instance which satisfy a specified predicate.

- In general, a predicate, may contain any of the logical comparative operators, which are =, ≠, <, ≤, >, ≥. Furthermore, several predicates may be combined using the connectives and (∧), or (∨), and not (¬).
- The select operation may be thought of as providing a horizontal cross section of the operand relation.

Selector operator examples



3.3.2 The Projection Operator

Type: unary
Symbol: Greek letter pi, π
General form: π(attribute-list)(relation instance)
Schema of result relation: specified by <attribute-list>
Size of result relation (tuples): ≤| operand relation|

Examples:

- π(student-id, name, major)(students)
- π(name, advisor)(students)
- π(name, gpa, hours-attempted)(students)

- The project operation can be viewed as producing a vertical cross-section of the operand relation.
- If the operation produces duplicate tuples, these are typically removed from the result relation in keeping with its set-like characteristics.

Projector operator examples

R

A	B	C	D
a	a	yes	1
b	d	no	7
c	f	yes	34
a	d	no	6
a	c	no	7
b	b	no	69
c	a	yes	24
d	d	yes	47
h	d	yes	34
e	c	no	26
a	a	yes	5

$r = \pi_{(A, C)}(R)$

A	C
a	yes
b	no
c	yes
a	no
d	yes
h	yes
e	no

$r = \pi_{(A, D)}(R)$

A	D
a	1
b	7
c	34
a	6
a	7
b	69
c	24
d	47
h	34
e	26
a	5

$r = \pi_{(C)}(R)$

C
yes
no

3.3.3 The Union Operator

Type: binary
Symbol: union symbol, $\cup$
General form: r $\cup$ s, where *r* and *s* are union compatible
Schema of result relation: schema of operand relations
Size of result relation (tuples): $\leq$ max {|r|+|s|}

Examples:

- r $\cup$ s $\quad \pi_{(a, b)}(r) \cup \pi_{(a, b)}(s)$

The union operation provides a means for extracting information. This resides in two operand relations which must be *union compatible*. Union compatibility requires that two conditions hold:

1. relations *r(R)* and *s(S)* in the expression *r* $\cup$ *s* must be of the same degree (*arity)*. That is, they must have the same number of attributes.
2. the domains of the *i*th attribute of *r(R)* and the *i*th attributes of *s(S)* must be the same, for all *i*.

Union operator examples

R

A	B	D
a	a	1
b	d	7
c	f	34
a	d	6
a	c	7

T

A	B
a	a
b	d
c	f
a	d
a	c

r = R ∪ T

not valid – R and T are not union compatible

r = R ∪ S

E	F	G
a	a	1
b	d	7
c	f	34
a	d	6
a	c	7
a	m	4
b	c	22
a	d	16

S

X	Y	Z
a	m	4
b	c	22
a	d	16
a	c	7

X

A	B
a	a
b	d
a	c

r = T ∪ X

A	B
a	a
b	d
c	f
a	d
a	c

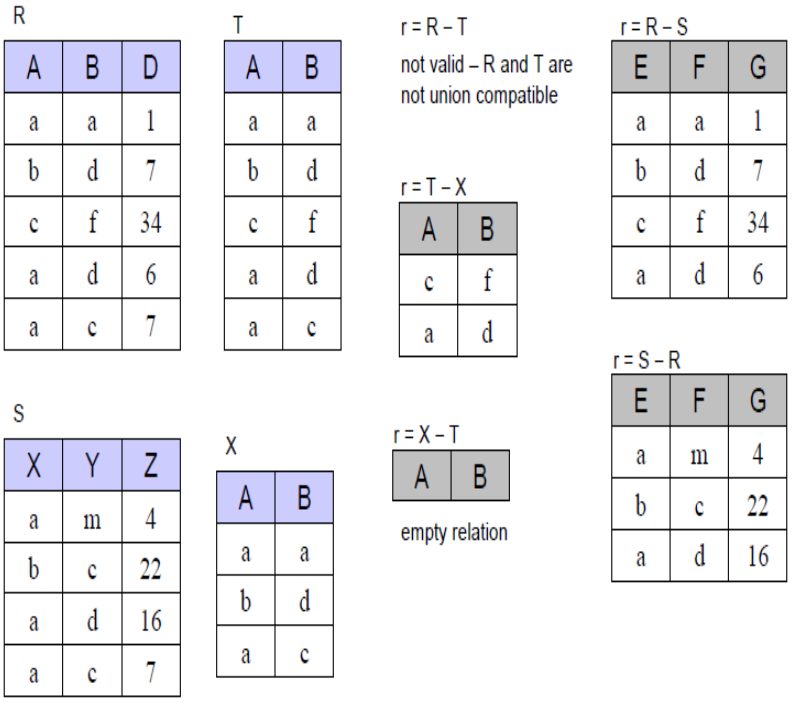
3.3.4 The Set Difference Operator

Type: binary
Symbol: −
General form: r − s, where *r* and *s* are union compatible
Schema of result relation: schema of operand relation
Size of result relation (tuples): ≤ |relation r|

Examples*:* r − s

The set difference operation allows for the extraction of information contained in one relation that is not contained in a second relation. As with the union operation, the set difference operation requires that the two operand relations be union compatible.

Set difference operator examples



3.3.5 The Cartesian product Operator

Type: binary
Symbol: ×
General form: r × s (no restrictions on *r* and *s*)
Schema of result relation: schema *r* × schema *s* with renaming
Size of result relation (tuples) : >|relation *r* |and >|relation *s*|

Examples: r × s

The Cartesian product operation allows for the combining of any two relations into a single relation. Recall that a relation is by definition a subset of a Cartesian product of a set of domains, so this gives you some idea of the behavior of the Cartesian product operation.

Cartesian Product Operator examples

T

A	B
a	a
b	d

X

A	B
a	a
b	d
a	c
c	a

r = T × X

T.A	T.B	X.A	X.B
a	a	a	a
a	a	b	d
a	a	a	c
a	a	c	a
b	d	a	a
b	d	b	d
b	d	a	c
b	d	c	a

R

A	B	C	D
a	a	1	yes
b	d	7	yes
c	f	34	no

S

X	Y	Z
a	m	4
b	c	22
a	d	16
a	c	7

r = R × S

A	B	C	D	X	Y	Z
a	a	1	yes	a	m	4
a	a	1	yes	b	c	22
a	a	1	yes	a	d	16
a	a	1	yes	a	c	7
b	d	7	yes	a	m	4
b	d	7	yes	b	c	22
b	d	7	yes	a	d	16
b	d	7	yes	a	c	7
c	f	34	no	a	m	4
c	f	34	no	b	c	22
c	f	34	no	a	d	16
c	f	34	no	a	c	7

3.4 Additional Operations in Relational Algebra

Relational database and set theoretic operations are not sufficient to perform some common database queries

- To enhance the power of relational algebra, there are some new operations introduced:

- Aggregate functions (SUM, AVERAGE, MAX, MIN, COUNT),
- Grouping, and
- Outer join

- ***Aggregate functions*** are defined on numeric attributes and applied on all relation tuples
- Relation is partitioned by means of ***grouping attribute*** values, and the defined aggregate functions are computed for the tuples of each group
- Outer join extends the join operation to cope with null values

AGGREGATE FUNCTIONS AND GROUPING

Notation: <grouping attributes> F <(function, attribute) list>(r (N))
where:

- <grouping attributes> is a list of attributes from R,
- F (pronounced as "script F") is the symbol used to denote
- aggregate operation, and
- <(function, attribute)_list> is a list of pairs (aggregate function from {SUM, AVERAGE, COUNT, MIN, MAX }, attribute from R)
- The resulting relation has columns for grouping attributes, and one column with the name of the form FUNCTION_ATTRIBUTE for each (function, attribute) pair

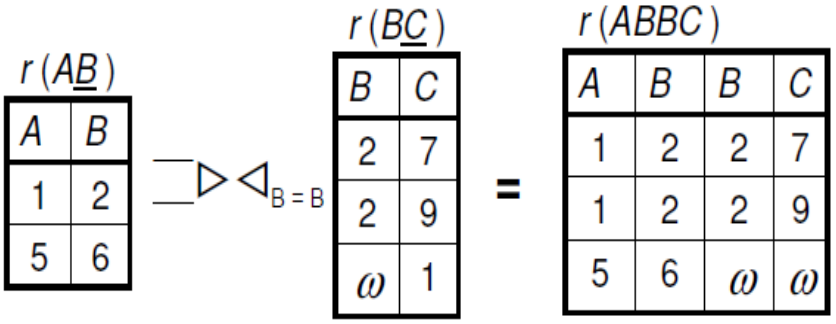
OUTER JOIN

Introduced to insert those tuples that don't match, or contain null values for join attributes into join relation

- Notation:

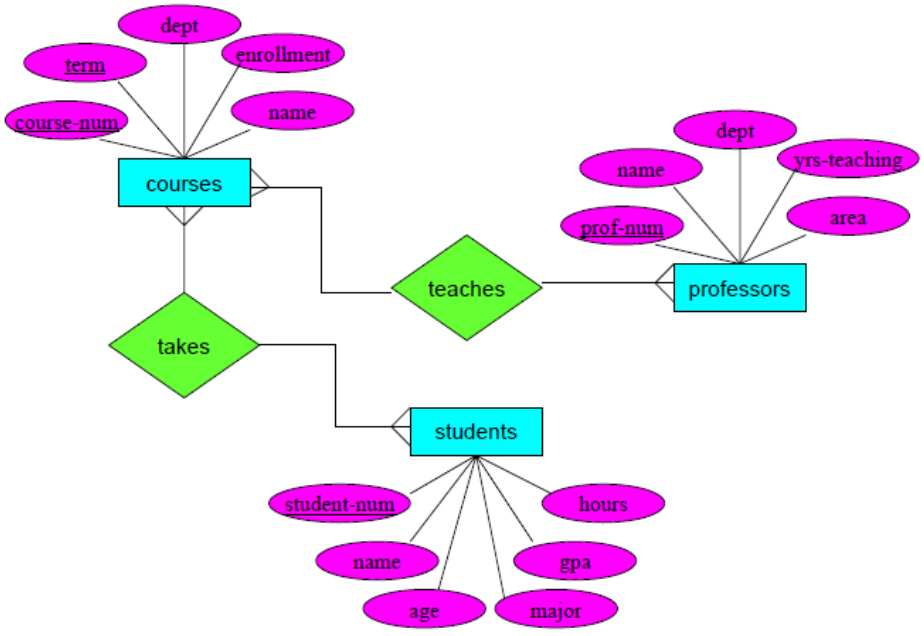
⟕ LEFT, ⟖ RIGHT, and ⟗ FULL outer join

Example



3.5 Relational Algebra Expressions

While each of the five fundamental relational algebra operators can be used individually to form a query, their expressive power is tremendously enhanced when they are combined together to form query expressions. Before we introduce the redundant operations in relational algebra we will look at forming more complicated combinations of the five fundamental operations. This will also make you appreciate the redundant operations all the more. To form meaningful queries, we need to be able to pose them against a database. For all of the examples that follow, we will use the following database:



Using the techniques for converting an ERD into a set of relational schemas we have the following resulting schemas:

- S = STUDENTS(s#, name, age, major, gpa, hours_completed)
- C = COURSES(c#, term, name, dept, enrollment)
- P = PROFESSORS(p#, name, dept, yrs_teaching, area)
- TA = TAKES(s#, c#, term, grade)
- TE = TEACH(p#, c#, term)

When you first begin to write queries in a new query language, it is sometimes helpful to actually visualise the data that might be in one of the operand (argument) relations upon which you are operating. To this end, the last two pages of this set of notes provide an instance of each of the relations above so that you can perform this visualisation. However, this is something that you will need to move away from as you get more advanced in your query composition, because you do not want to influence the design of your query by visualising a relation instance that may not contain all possible tuples that your query will encounter.

Example Query 1:

Find the names of all the students who are Computer Science majors.

Approach:

- First select all of the students who are CS majors.
- r = σ(major = “Computer Science”)(S)
- Next project only the name attribute from the previous result.
- result = π(name)(r)
- Complete Query Expression:
- result = π(name)(σ(major = “Computer Science”)(S))

Example Query 2:

Find the student-num (s#) and name of all the students who have completed more than 90 hours.

Approach:

- First select all of the students who have completed more than 90 hours.
- r = σ(hours_completed > 90)(S)
- Next project the student-num and name attributes from the previous result.
- result = π(s#, name)(r)

- Complete Query Expression:
- result = π(s#, name)(σ(hours_completed > 90)(S))

Example Query 3:

Find the names of all those students who are less than 20 years old who have completed more than 80 hours.

Approach:

- First select all of the students who have completed more than 80 hours and are less than 20 years old.
- r = σ((hours_completed > 80) AND (age < 20))(S)
- Next project the name attribute from the previous result.
- result = π(name)(r)
- Complete Query Expression:
- result = π(name)(σ((hours_completed > 80) AND (age < 20))(S))

Example Query 4:

Find the names of all the courses that are offered by either Computer Science or Physics.

Approach:

- First select all of the courses that are offered by either CS or Physics.
- r = σ((dept = Computer Science) or (dept = Physics))(C))
- Next project the name attribute from the previous result.
- result = π(name)(r)
- Complete Query Expression:
- result = π(name)(σ((dept = Computer Science) or (dept = Physics))(C))

Example Query 5:

Find the names of all the students who took a course in the Fall 2006 term that was taught by a professor who had more than 20 years of teaching experience.

Approach:

- First put the professor information together with the course information together with the teachers information together with the takes information.

- Next, select only related students, professors and courses from previous result.
- Finally, select only the students name from the previous result.
- Complete Query Expression:
- result = π(S.name)(σ((TA.term = Fall 2006) AND (P.yrs_teaching > 20) AND (S.s# = TA.s#) AND (P.p# = TE.p#)AND (TA.c# = TE.c#) AND (TA.term = TE.term))(S × P × TA × TE))

Example Query 6:

Find the names of all the professors who are either in the Computer Science department or have more than 20 years of teaching experience.

Complete Query Expression:

- result = [π(name)(σ(dept = Computer Science)(P))] ∪ [π(name)(σ(yrs_teaching > 20)(P))]
- or:
- result = π(name)(σ((dept = Computer Science) OR (yrs_teaching > 20))(P))

4.0 CONCLUSION

In this unit, you have been introduced to the fundamental concepts of query languages and relational algebra. You have also learnt the different types of operations in relational algebra, how to use the operators, constructing relational algebra expressions and using these expressions to query a database.

SELF-ASSESSMENT EXERCISE

Using the sample database given under 3.5, write query expressions to:

i. Find the name of the professor who taught a course in the Fall 2006 term
ii. Find the student numbers for those students who were enrolled only in the spring 2007 term.

5.0 SUMMARY

What you have learnt in this unit concerns:

- What a Query Language does
- Procedural and non – procedural language
- Fundamental operations in Relational Algebra

- How to construct relational Algebra expressions using operators

6.0 TUTOR-MARKED ASSIGNMENT

Consider the following relational schemas which represent Sailors and Reserves, where sid is Sailor's identity, bid is Boat identity and sname is Sailor's name.

- R1(sid, bid, day)
- S1(sid,sname,rating,age)
- S2(sid,sname,rating,age)

Using relational Algebra, find:

1. The names of sailors who reserved all boats
2. Find sailors who reserved a red or a green boat

7.0 REFERENCES/FURTHER READING

Hoffman, James (1997). SQL Tutorials.

Teach Yourself SQL in 21 Days, (2nd ed). Macmillan Computer Publishing SQL – A Practical Introduction by Akeel I. Din.

MODULE 3

UNIT 1 WEB SERVICES

CONTENTS

1.0 INTRODUCTION

Web services increasingly tie together a number of participants forming large distributed applications. The resulting activities may have complex structure and relationships.

The current set of web service specifications defines protocols for web service interoperability.

2.0 OBJECTIVES

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

- describe the basic concept and application of web services.
- identify web services framework
- describe web services architecture.

3.0 MAIN CONTENT

3.1 What is Web Services?

The term web services describes a standardized way of integrating web-based applications using the XML, SOAP, WSDL and UDDI open standards over an Internet protocol backbone. XML is used to tag the data, SOAP is used to transfer the data, WSDL is used for describing the services available and UDDI is used for listing what services are available. Used primarily as a means for businesses to communicate with each other and with clients, web services allow organisations to communicate data without intimate knowledge of each other's IT systems behind the firewall.

3.1.1 Web Service Security

Web Service Security defines how to use XML Encryption and XML Signature in SOAP to secure message exchanges, as an alternative or extension to using HTTPS to secure the channel.

3.1.2 Web Service Reliability

Web Service Reliability describes an OASIS standard protocol for reliable messaging between two web services.

3.1.3 Web Services Transaction

Web Services Transaction is a way of handling transactions. It is also defined as protocols that govern the outcome of atomic transactions.

3.2 Application of Web Services

Web services are a set of tools that can be used in a number of ways. The three most common ways of use are Remote procedure calls (RPC), SOA and REST.

3.2.1 Remote Procedure Calls (RPC)

Remote procedure calls web services present a distributed function (or method) call interface that is familiar to many developers. Typically, the basic unit of RPC Web services is the WSDL operation.

The first web services tools were focused on RPC, and as a result this style is widely deployed and supported. However, it is sometimes criticised for not being loosely coupled, because it was often implemented by mapping services directly to language-specific functions or method calls.

3.2.2 Service-Oriented Architecture

Web services can also be used to implement architecture according to Service-oriented architecture (SOA) concepts, where the basic unit of communication is a message, rather than an operation. This is often referred to as "message-oriented" services.

SOA web services are supported by most major software vendors and industry analysts. Unlike RPC Web services, loose coupling is more likely, because the focus is on the "contract" that WSDL provides, rather than the underlying implementation details.

3.2.3 Representational State Transfer

Finally, **RESTful Web Services** attempt to emulate HTTP and similar protocols by constraining the interface to a set of well-known, standard operations (e.g., GET, PUT, DELETE). Here, the focus is on interacting with stateful resources, rather than messages or operations. Restful web services can use WSDL to describe SOAP messaging over HTTP, which defines the operations, or can be implemented as an abstraction purely on top of SOAP (e.g., WS-Transfer).

3.3 Web Services Framework

A list of web service frameworks is given below:

Name	Platform	Messaging Model (Destination)	Specifications	Protocols
ActionWeb Service	Ruby (on Rails)	Client/Server	?	SOAP, XML-RPC, WSDL
Apache Axis	Java/C++	Client/Server	WS-ReliableMessaging, WS-Coordination, WS-Security, WS-AtomicTransaction, WS-Addressing	SOAP, WSDL
Apache Axis2	Java/C	Client/Server/ Asyn Support	WS-ReliableMessaging,WS-Security, WS-AtomicTransaction, WS-Addressing ,MTOM ,WS-Policy, WS-MetadataExchange	SOAP,MTOM , WSDL 2.0, WSDL
Apache CXF	Java	Client/Server/ Asyn Support	WS-Addressing, WS-Policy, WS-ReliableMessaging, WS-Security, MTOM	SOAP1.1, SOAP1.2,MTOM, WSDL 2.0, WSDL
AlchemySOAP	C++	Client/Server	WS-Addressing	SOAP
csoap	C	Client/Server	?	SOAP
Halcyon	Ruby	Client/Server	N/A	JSON
Hessian	Java, Ruby, Python, Erlang, PHP, others	Client/Server	Hessian 1.0.1	Hessian
JSON-RPC-Java	Java	Server	???	JSON-RPC
JSON-RPC-Lua	Lua	Server	???	JSON-RPC
Java Web Services Development Pack / GlassFish	Java	Client/Server	WS-Addressing, WS-Security, ???	SOAP, WSDL, ???
NuSOAP	PHP	Client/Server	Object Oriented, Creates Users Help document,	SOAP, WSDL

SOAP Lite	Perl	Client/Server	???	SOAP, WSDL, ???
Web Services Interoperability Technology	Java	Client/Server	WS-Addressing, WS-ReliableMessaging, WS-Coordination, WS-AtomicTransaction, WS-Security, WS-Security Policy, WS-Trust, WS-SecureConversation, WS-Policy, WS-MetadataExchange	SOAP, WSDL, MTOM
Web Services Invocation Framework	Java	Client	???	SOAP, WSDL
Windows Communication Foundation	.Net	Client/Server ?	WS-Addressing, WS-ReliableMessaging, WS-Security	SOAP, WSDL
XFire ***became Apache CXF***	Java	Client/Server	WS-Addressing, WS-Security	SOAP, WSDL
XML Interface for Network Services	Java	Server ?	??	SOAP, XML-RPC
gSOAP	C/C++	Client/Server	WS-Addressing, WS-Discovery, WS-Enumeration, WS-Security	SOAP, XML-RPC, WSDL
Zolera SOAP Infrastructure (ZSI)	Python	Client/Server	???	SOAP, WSDL
WSO2 Web Services Framework for C(WSO2 WSF/C)	C (build on Axis2/c)	Client/Server, Publish/Subscribe	WS-Addressing, WS-Policy, WS-Security, WS-SecurityPolicy, WS-ReliableMessaging, WS-Eventing	SOAP, WSDL, TLS
WSO2 WSF/PHP	PHP	Client/Server	WS-Addressing, WS-Policy, WS-Security, WS-SecurityPolicy, WS-ReliableMessaging, WS-SecureConversation, MTOM	SOAP, WSDL, WSDL 2.0
WSO2	Ruby on	Client/Server	WS-Addressing, WS-	SOAP, WSDL

WSF/Ruby	Rails		Security, WS-SecurityPolicy, WS-ReliableMessaging, MTOM	

3.4 Web Services Architecture

3.4.1 Purpose of Web Services Architecture

This Web Service Architecture (WSA) is intended to provide a common definition of a web service, and define its place within a larger web services framework to guide the community. The WSA provides a conceptual model and a context for understanding web services and the relationships between the components of this model.

The architecture does not attempt to specify how web services are implemented, and imposes no restriction on how web services might be combined. The WSA describes both the minimal characteristics that are common to all web services, and a number of characteristics that are needed by many, but not all, web services.

The web services architecture is interoperability architecture: it identifies those global elements of the global web services network that are required in order to ensure interoperability between web services.

3.4.2 Agents and Services

A web service is an abstract notion that must be implemented by a concrete agent. (See Figure 1-1) The agent is the concrete piece of software or hardware that sends and receives messages, while the service is the resource characterised by the abstract set of functionality that is provided. To illustrate this distinction, you might implement a particular web service using one agent one day (perhaps written in one programming language), and a different agent the next day (perhaps written in a different programming language) with the same functionality. Although the agent may have changed, the web service remains the same.

3.4.3 Requesters and Providers

The purpose of a web service is to provide some functionality on behalf of its owner -- a person or organisation, such as a business or an individual. The provider entity is the person or organisation that

provides an appropriate agent to implement a particular service. (See Figure 1-1: Basic Architectural Roles.)

A requester entity is a person or organisation that wishes to make use of a provider entity's web service. It will use a requester agent to exchange messages with the provider entity's provider agent.

(In most cases, the requester agent is the one to initiate this message exchange, though not always. Nonetheless, for consistency we still use the term "requester agent" for the agent that interacts with the provider agent, even in cases when the provider agent actually initiates the exchange.).

3.4.4 Service Description

The mechanics of the message exchange are documented in a web service description (WSD) (Fig.1). The WSD is a machine-processable specification of the web service's interface, written in WSDL. It defines the message formats, datatypes, transport protocols, and transport serialisation formats that should be used between the requester agent and the provider agent. It also specifies one or more network locations at which a provider agent can be invoked, and may provide some information about the message exchange pattern that is expected. In essence, the service description represents an agreement governing the mechanics of interacting with that service. (Again this is a slight simplification that will be explained further in **3.3 Using Web Services**.)

3.4.5 Semantics

The semantics of a web service is the shared expectation about the behavior of the service, in particular in response to messages that are sent to it. In effect, this is the "contract" between the requester entity and the provider entity regarding the purpose and consequences of the interaction. Although this contract represents the overall agreement between the requester entity and the provider entity on how and why their respective agents will interact, it is not necessarily written or explicitly negotiated. It may be explicit or implicit, oral or written, machine-processable or human-oriented, and it may be a legal agreement or an informal (non-legal) agreement.

While the service description represents a contract governing the mechanics of interacting with a particular service, the semantics represents a contract governing the meaning and purpose of that interaction. The dividing line between these two is not necessarily rigid. As more semantically-rich languages are used to describe the mechanics

of the interaction, more of the essential information may migrate from the informal semantics to the service description. As this migration occurs, more of the work required to achieve successful interaction can be automated.

3.4.6 Overview of Engaging a Web Service

There are many ways that a requester entity might engage and use a web service. In general, the following broad steps are required, as illustrated in Figure 1: (1) the requester and provider entities become known to each other (or at least one becomes known to the other); (2) the requester and provider entities somehow agree on the service description and semantics that will govern the interaction between the requester and provider agents; (3) the service description and semantics are realised by the requester and provider agents; and (4) the requester and provider agents exchange messages, thus performing some task on behalf of the requester and provider entities. (I.e., the exchange of messages with the provider agent represents the concrete manifestation of interacting with the provider entity's web service). These steps are explained in more detail in 3.4 Web Service Discovery. Some of these steps may be automated, others may be performed manually.

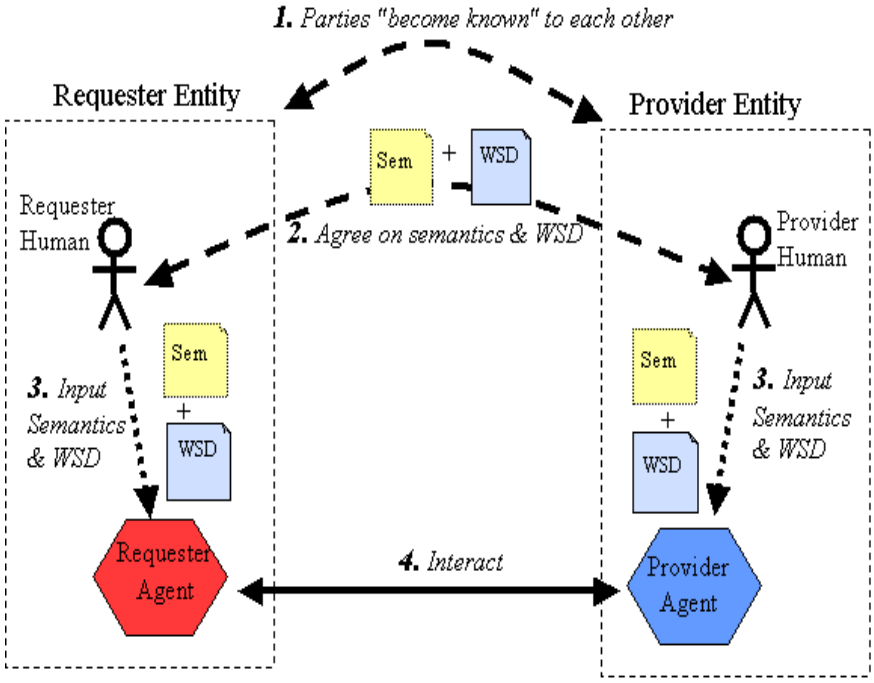


Fig. 1: The General Process of Engaging a Web Service

3.5 Concepts and Relationship

3.5.1 Introduction

The formal core of the architecture is this enumeration of the concepts and relationships that are central to web services' interoperability.

3.5.2 How to Read this Section

The architecture is described in terms of a few simple elements: concepts, relationships and models. Concepts are often noun-like in that they identify things or properties that we expect to see in realisations of the architecture, similarly relationships are normally linguistically verbs.

As with any large-scale effort, it is often necessary to structure the architecture itself. We do this with the larger-scale meta-concept of model. A model is a coherent portion of the architecture that focuses on a particular theme or aspect of the architecture.

3.5.3 Concepts

A concept is expected to have some correspondence with any realisations of the architecture. For example, the message concept identifies a class of object (not to be confused with Objects and Classes as are found in Object-Oriented Programming languages) that we expect to be able to identify in any web services context. The precise form of a message may be different in different realisations, but the message concept tells us what to look for in a given concrete system rather than prescribing its precise form.

Not all concepts will have a realisation in terms of data objects or structures occurring in computers or communications devices; for example the person or organisation refers to people and human organisations. Other concepts are more abstract still; for example, message reliability denotes a property of the message transport service — a property that cannot be touched but nonetheless is important to web services.

Each concept is presented in a regular, stylised way consisting of a short definition, an enumeration of the relationships with other concepts, and a slightly longer explanatory description. For example, the concept of agent includes as relating concepts the fact that an agent is a computational resource, has an identifier and an owner. The description part of the agent explains in more detail why agents are important to the architecture.

3.5.4 Relationships

Relationships denote associations between concepts. Grammatically, relationships are verbs; or more accurately, predicates. A statement of a relationship typically takes the form: concept predicate concept. For example, in agent, we state that:

- **An agent is**

A computational resource

This statement makes an assertion, in this case, about the nature of agents. Many such statements are descriptive, others are definitive:

- **A message has**

A message sender

Such a statement makes an assertion about valid instances of the architecture: we expect to be able to identify the message sender in any realisation of the architecture. Conversely, any system for which we cannot identify the sender of a message is not conformant to the architecture. Even if a service is used anonymously, the sender has an identifier but it is not possible to associate this identifier with an actual person or organisation.

3.5.5 Concept Maps

Many of the concepts in the architecture are illustrated with *concept maps*. A concept map is an informal, graphical way to illustrate key concepts and relationships. For example the diagram below shows three concepts which are related in various ways. Each box represents a concept, and each arrow (or labeled arc) represents a relationship.

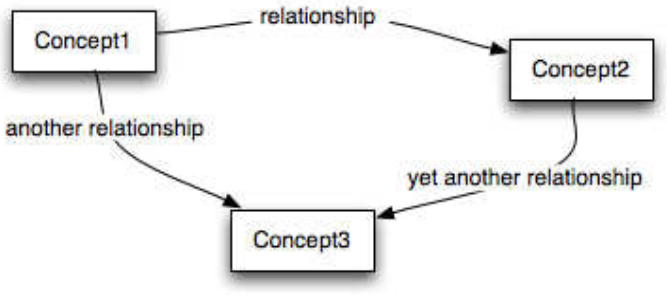


Fig. 2: Concept Map

The merit of a concept map is that it allows rapid navigation of the key concepts and illustrates how they relate to each other. It should be stressed, however, that these diagrams are primarily navigational aids; the written text is the definitive source.

3.5.6 Model

A model is a coherent subset of the architecture that typically revolves around a particular aspect of the overall architecture. Although different models share concepts, it is usually from different points of view; the major role of a model is to explain and encapsulate a significant theme within the overall web services architecture.

For example, the Message-Oriented Model focuses and explains web services strictly from a message-passing perspective. In particular, it does not attempt to relate messages to services provided. The Service - Oriented Model, however, lays on top of, and extends the Message - Oriented Model in order to explain the fundamental concepts involved in service-in effect to explain the purpose of the messages in the Message-Oriented Model.

Each model is described separately below, in terms of the concepts and relationships inherent to the model. The ordering of the concepts in each model section is alphabetical; this should not be understood to imply any relative importance. For a more focused viewpoint the reader is directed to the Stakeholder's perspectives section which examines the architecture from the perspective of key stakeholders of the architecture.

The reason for choosing an alphabetical ordering is that there is a large amount of cross-referencing between the concepts. As a result, it is very difficult, if not misleading, to choose a non-alphabetic ordering that reflects some sense of priority between the concepts. Furthermore, the optimal ordering depends very much on the point of view of the reader. Hence, we devote the Stakeholders perspectives section to a number of prioritized readings of the architecture.

3.5.7 Conformance

Unlike language specifications, or protocol specifications, conformance to architecture is necessarily a somewhat imprecise art. However, the presence of a concept in this enumeration is a strong hint that, in any realisation of the architecture, there should be a corresponding feature in the implementation. Furthermore, if a relationship is identified here, then there should be corresponding relationships in any realised architecture. The consequence of non-conformance is likely to be reduced interoperability: The absence of such a concrete feature may not

prevent interoperability, but it is likely to make such interoperability more difficult.

A primary function of the Architecture's enumeration in terms of models, concepts and relationships is to give guidance about conformance to the architecture. For example, the architecture notes that a message has a message sender; any realisation of this architecture that does not permit a message to be associated with its sender is not in conformance with the architecture. For example, SMTP could be used to transmit messages. However, since SMTP (at present) allows forgery of the sender's identity, SMTP by itself is not sufficient to discharge this responsibility.

3.5.8 The Architectural Models

This architecture has four models, illustrated in Fig. 3. Each model in the figure is labeled with what may be viewed as the key concept of that model.

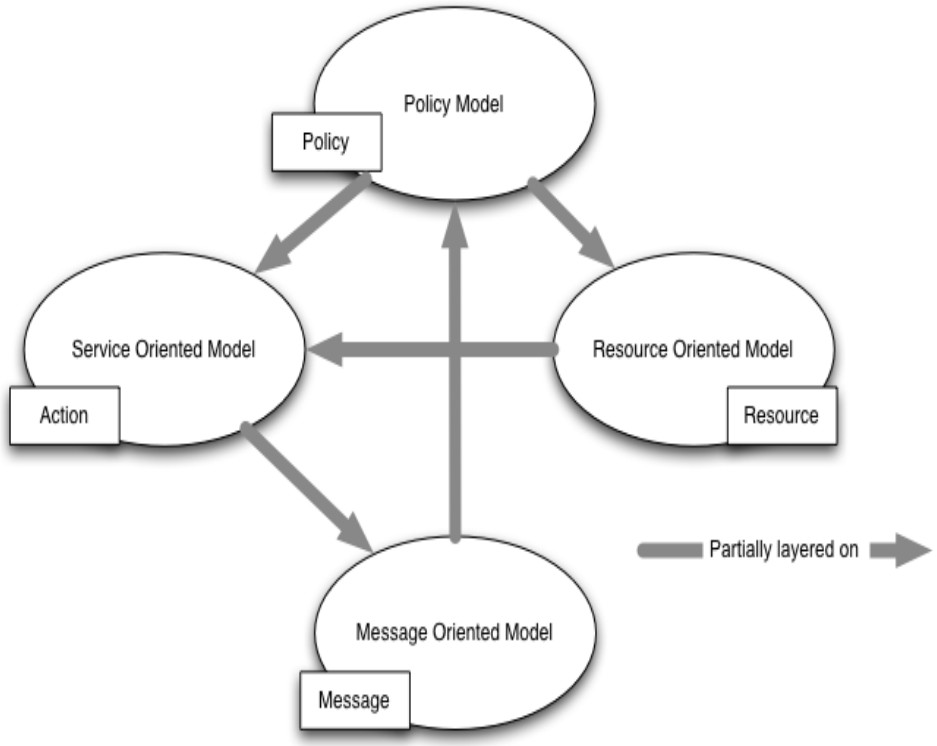


Fig. 3: Meta Model of the Architecture

The four models are:

- The Message-Oriented Model focuses on messages, message structure, and message transport and so on — with neither particular reference as to the reasons for the messages, nor to their significance.

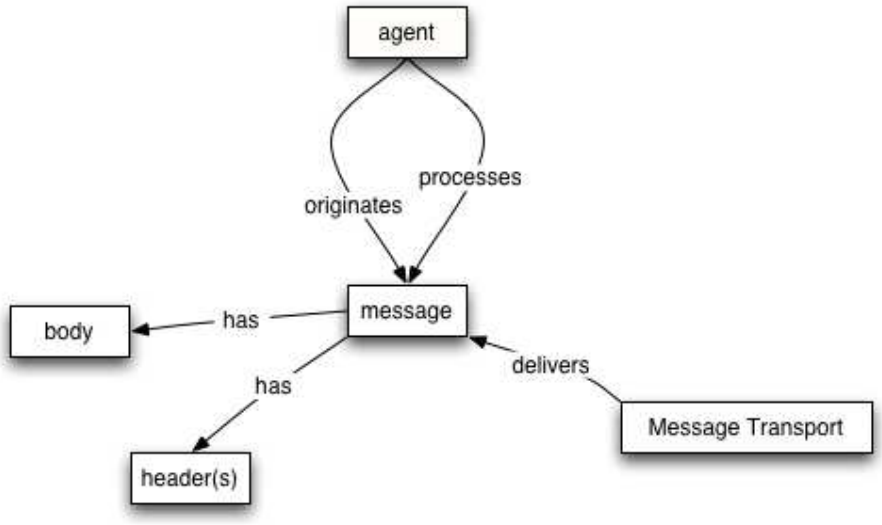


Fig. 4: Simplified Message- Oriented Model

The essence of the message model revolves around a few key concepts illustrated above: the agent that sends and receives messages, the structure of the message in terms of message headers and bodies and the mechanisms used to deliver messages. Of course, there are additional details to consider: the role of policies and how they govern the message level model. The abridged diagram shows the key concepts; the detailed diagram expands on this to include many more concepts and relationships.

- The Service-Oriented Model focuses on aspects of service, action and so on. While clearly, in any distributed system, services cannot be adequately realised without some means of messaging, the converse is not the case: messages do not need to relate to services.

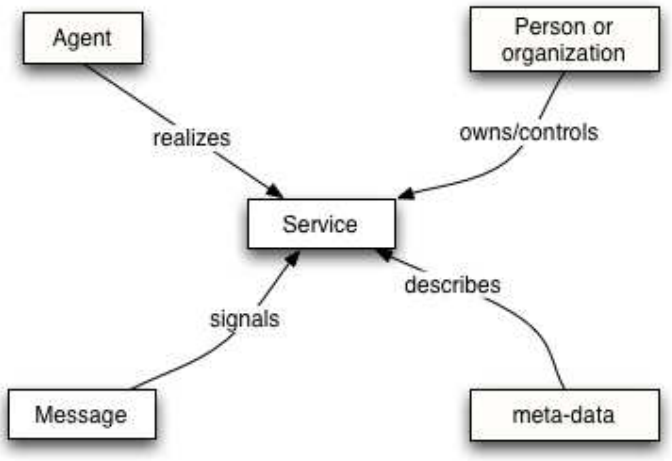


Fig. 5: Simplified Service-Oriented Model

The Service-Oriented Model is the most complex of all the models in the architecture. However, it too revolves around a few key ideas. A service is realised by an agent and used by another agent. Services are mediated by means of the messages exchanged between requester agents and provider agents.

A very important aspect of services is their relationship to the real world: services are mostly deployed to offer functionality in the real world. We model this by elaborating on the concept of a service owner — which, whether it is a person or an organisation, has a real world responsibility for the service.

Finally, the Service-Oriented Model makes use of meta-data, which, as described in **3.1 Service- Oriented Architecture**, is a key property of Service-Oriented Architecture. This meta-data is used to document many aspects of services: from the details of the interface and transport binding to the semantics of the service and what policy restrictions there may be on the service. Providing rich descriptions is the key to successful deployment and use of services across the Internet.

- The Resource-Oriented Model focuses on resources that exist and have owners.

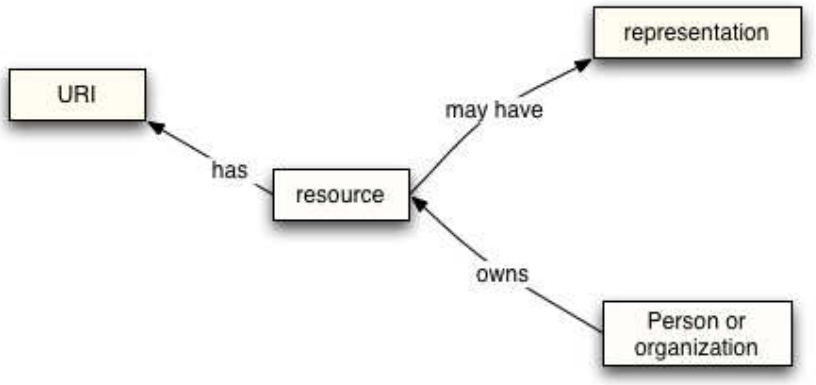


Fig. 6: Simplified Resource- Oriented Model

The resource model is adopted from the Web Architecture concept of resource. We expand on this to incorporate the relationships between resources and owners.

- The Policy Model focuses on constraints on the behaviour of agents and services. We generalize this to resources since policies can apply equally to documents (such as descriptions of services) as well as active computational resources.
-

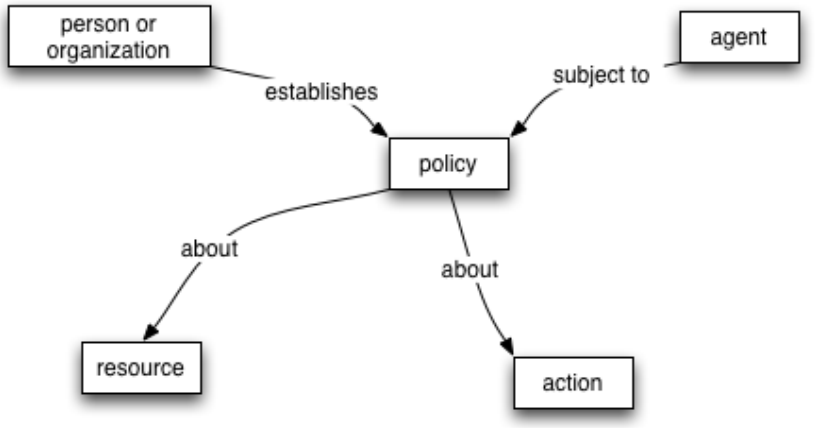


Fig. 7: Simplified Policy Model

Policies are about resources. They are applied to agents that may attempt to access those resources, and are put in place, or established, by people who have responsibility for the resource.

Policies may be enacted to represent security concerns, quality of service concerns, management concerns and application concerns.

3.5.9 Message-Oriented Model

The Message-Oriented Model focuses on those aspects of the architecture that relate to messages and their processing (Fig 8). Specifically, in this model, we are not concerned with any semantic significance of the content of a message or its relationship to other messages. However, the MOM does focus on the structure of messages, on the relationship between message senders and receivers and how messages are transmitted.

.

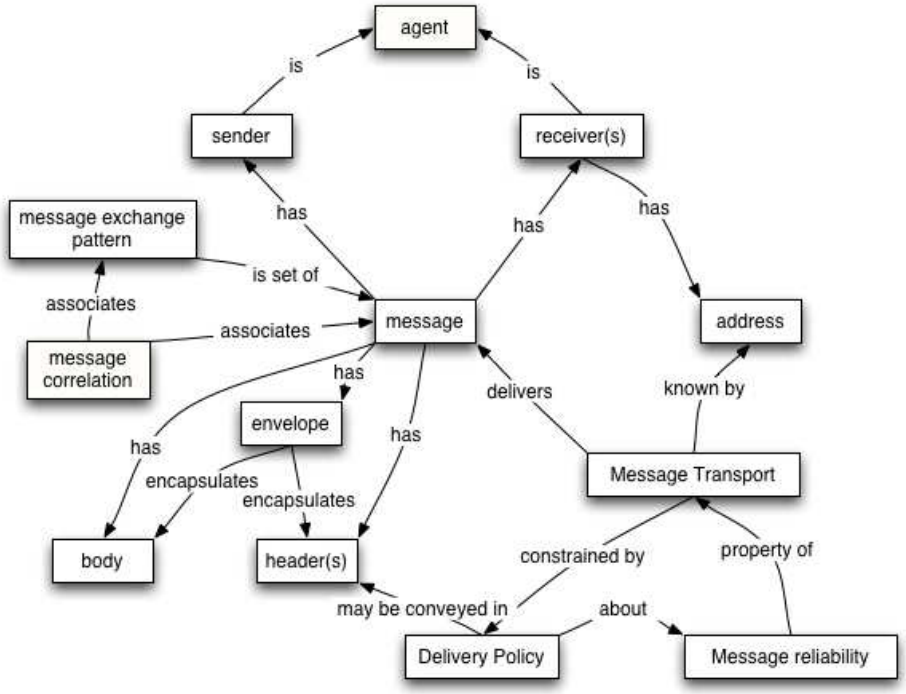


Fig. 8: Message-Oriented Model

SELF-ASSESSMENT EXERCISE

1. Explain the various web services applications
2. Briefly describe the various architectural models of web services.

4.0 CONCLUSION

In this unit you have been introduced to the fundamental concepts of web services, web services framework and the various phases of web service architecture.

5.0 SUMMARY

What you have learned in this unit concerns:

- introduction to web service, web Service Security, web Service Reliability, web Services Transaction and Applications of web Services
- web services frame work
- purpose of Web services Architecture, Agents and Services, Requesters and Providers, Service Description, Semantics, Overview of Engaging a web Service
- concepts and Relationships, Concept Maps, Models, Conformance, the Architectural Models: policy models, service oriented models, resource oriented models, message oriented models.

6.0 TUTOR-MARKED ASSIGNMENT

List and explain the various phases of web service architecture.

7.0 REFERENCES/FURTHER READING

A Word Definition from Webopedia Computer Dictionary.

Louis, Felipe Cabrera (2005). *Web Services Atomic Transaction* (WS-Atomic Transaction).

UNIT 2 INTRODUCTION TO XML

CONTENTS

1.0 INTRODUCTION

In this unit, you will be introduced to Extensible Markup Language (XML), origin, terminologies and goals. You will also learn about XML document rules, document contents, programming interfaces, standards and several case studies.

2.0 OBJECTIVES

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

- explain why XML was created
- list the rules of XML documents
- define what an XML document can and cannot contain
- explain the programming interfaces that work with XML documents
- explain what the main XML standards are and how they work together
- describe how companies use XML in the real world.

3.0 MAIN CONTENT

3.1 General Introduction

Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behaviour of computer programmes which process them. XML is an application profile or restricted form of SGML, the Standard

Generalised Markup Language [ISO 8879]. By construction, XML documents are conforming SGML documents.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

A software module called an **XML processor** is used to read XML documents and provide access to their content and structure. It is assumed that an XML processor is doing its work on behalf of another module, called the **application**. This specification describes the required behaviour of an XML processor in terms of how it must read XML data and the information it must provide to the application.

3.2 Origin and Goals

XML was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996. It was chaired by Jon Bosak of Sun Microsystems with the active participation of an XML Special Interest Group (previously known as the SGML Working Group) also organised by the W3C. The membership of the XML Working Group is given in an appendix. Dan Connolly served as the Working Group's contact with the W3C.

The design goals for XML are:

- XML shall be straightforwardly usable over the Internet.
- XML shall support a wide variety of applications.
- XML shall be compatible with SGML.
- It shall be easy to write programmes which process XML documents.
- The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
- XML documents should be human-legible and reasonably clear.
- The XML design should be prepared quickly.
- The design of XML shall be formal and concise.
- XML documents shall be easy to create.
- Terseness in XML markup is of minimal importance.

This specification, together with associated standards (Unicode [Unicode] and ISO/IEC 10646 [ISO/IEC 10646] for characters, Internet BCP 47 [IETF BCP 47] and the Language Subtag Registry [IANA-LANGCODES] for language identification tags), provides all the information necessary to understand XML Version 1.0 and construct computer programmes to process it.

This version of the XML specification may be distributed freely, as long as all text and legal notices remain intact.

3.3 Terminology

The terminology used to describe XML documents is defined in the body of this specification. The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when EMPHASIZED, are to be interpreted as described in [IETF RFC 2119]. In addition, the terms defined in the following list are used in building those definitions and in describing the actions of an XML processor:

- **Error**

A violation of the rules of this specification; results are undefined. Unless otherwise specified, failure to observe a prescription of this specification indicated by one of the keywords MUST, REQUIRED, MUST NOT, SHALL and SHALL NOT is an error. Conforming software MAY detect and report an error and MAY recover from it.

- **Fatal Error**

An error which a conforming XML processor MUST detect and report to the application. After encountering a fatal error, the processor MAY continue processing the data to search for further errors and MAY report such errors to the application. In order to support correction of errors, the processor MAY make unprocessed data from the document (with intermingled character data and markup) available to the application. Once a fatal error is detected, however, the processor MUST NOT continue normal processing (i.e., it MUST NOT continue to pass character data and information about the document's logical structure to the application in the normal way).

- **At User Option**

Conforming software MAY or MUST (depending on the modal verb in the sentence) behave as described; if it does, it MUST provide users a means to enable or disable the behaviour described.

- **Validity Constraint**

A rule which applies to all valid XML documents. Violations of validity constraints are errors; they MUST, at user option, be reported by validating XML processors.

- **Well-Formedness Constraint**

A rule which applies to all well-formed XML documents. Violations of well-formedness constraints are fatal errors.

- **Match**

(Of strings or names:) Two strings or names being compared are identical. Characters with multiple possible representations in ISO/IEC 10646 (e.g. characters with both precomposed and base+diacritic forms) match only if they have the same representation in both strings. No case folding is performed. (Of strings and rules in the grammar:) A string matches a grammatical production if it belongs to the language generated by that production. (Of content and content models:) An element matches its declaration when it conforms in the fashion described in the constraint.

- **For Compatibility**

Marks a sentence describing a feature of XML included solely to ensure that XML remains compatible with SGML.

- **For Interoperability**

Marks a sentence describing a non-binding recommendation included to increase the chances that XML documents can be processed by the existing installed base of SGML processors which predate the WebSGML Adaptations Annex to ISO 8879.

3.4 Why Do We Need XML?

HTML is the most successful markup language of all time. You can view the simplest HTML tags on virtually any device, from palmtops to mainframes, and you can even convert HTML markup into voice and other formats with the right tools. Given the success of HTML, why did the W3C create XML? To answer that question, take a look at this document:

- <p>Mrs. Mary Wobo
-

- 14 Ken Street
-
- Enugu</p>

The trouble with HTML is that it was designed with humans in mind. Even without viewing the above HTML document in a browser, you and I can figure out that it is someone's postal address. (Specifically, it is a postal address for someone in Nigeria addresses; you could probably guess what this represents.).

As humans, you and I have the intelligence to understand the meaning and intent of most documents. A machine, unfortunately, cannot do that. While the tags in this document tell a browser how to display this information, the tags do not tell the browser **what the information is**. You and I know it is an address, but a machine does not.

3.5 Rendering HTML

To render HTML, the browser merely follows the instructions in the HTML document. The paragraph tag tells the browser to start rendering on a new line, typically with a blank line beforehand, while the two break tags tell the browser to advance to the next line without a blank line in between. While the browser formats the document beautifully, the machine still does not know this is an address.

3.6 Processing HTML

To wrap up this discussion of the sample HTML document, consider the task of extracting the postal code from this address. Here is an (intentionally brittle) algorithm for finding the postal code in HTML markup:

- If you find a paragraph with two
 tags, the postal code is the second word after the first comma in the second break tag.
  Although this algorithm works with this example, there is any number of perfectly valid addresses worldwide for which this simply would not work. Even if you could write an algorithm that found the postal code for any address written in HTML, there is any number of paragraphs with two break tags that do not contain addresses at all. Writing an algorithm that looks at any HTML paragraph and finds any postal codes inside it would be extremely difficult, if not impossible.

3.6.1 A Sample XML Document

Now let us look at a sample XML document. With XML, you can assign some meaning to the tags in the document. More importantly, it is easy for a machine to process the information as well. You can extract the postal code from this document by simply locating the content surrounded by the <postal-code> and </postal-code> tags, technically known as the <postal-code> element.

- <address>
- <name>
- <title>Mrs.</title><first-name>
- Mary
- </first-name>
- <last-name>
- McGoon
- </last-name>
- </name>
- <street>
- 1401 Main Street
- </street>
- <city>Anytown</city>
- <state>NC</state>
- <postal-code>
- 34829
- </postal-code>
- </address>

3.7 Tags, Elements and Attributes

There are three common terms used to describe parts of an XML document: *tags*, *elements*, and *attributes*. Here is a sample document that illustrates the terms:

- <address>
- <name>
- <title>Mrs.</title>
- <first-name>
- Mary
- </first-name>
- <last-name>
- McGoon
- </last-name>
- </name>

- <street>
- 1401 Main Street
- </street>
- <city state="NC">Anytown</city>
- <postal-code>
- 34829
- </postal-code>
- </address>

- A tag is the text between the left angle bracket (<) and the right angle bracket (>).There are starting tags (such as <name>) and ending tags (such as </name>)
- An element is the starting tag, the ending tag, and everything in between. In the sample above, the <name> element contains three child elements: <title>, <first-name>, and <last-name>.
- An attribute is a name-value pair inside the starting tag of an element. In this example, state is an attribute of the <city> element; in earlier examples,<state> was an element (see A sample XML document).

3.8 How XML is Changing the Web

Now that you have seen how developers can use XML to create documents with self-describing data, let us look at how people are using those documents to improve the web. Here are a few key areas:

- **XML simplifies data interchange.** Because different organisations (or even different parts of the same organisation) rarely standardise on a single set of tools, it can take a significant amount of work for applications to communicate. Using XML, each group creates a single utility that transforms their internal data formats into XML and vice versa. Best of all, there is a good chance that their software vendors already provide tools to transform their database records (or LDAP directories, or purchase orders, and so forth) to and from XML.
- **XML enables smart code.** Because XML documents can be structured to identify every important piece of information (as well as the relationships between the pieces), it is possible to write code that can process those XML documents without human intervention. The fact that software vendors have spent massive amounts of time and money building XML development tools means writing that code is a relatively simple process.
- **XML enables smart searches.** Although search engines have improved steadily over the years, it is still quite common to get erroneous results from a search. If you are searching HTML

pages for someone named "Chip," you might also find pages on chocolate chips, computer chips, wood chips, and lots of other useless matches. Searching XML documents for <first-name> elements that contained the text Chip would give you a much better set of results.

3.9 XML Document Rules

3.9.1 Overview

If you have looked at HTML documents, you are familiar with the basic concepts of using tags to mark up the text of a document. This section discusses the differences between HTML documents and XML documents. It goes over the basic rules of XML documents, and discusses the terminology used to describe them.

One important point about XML documents: **The XML specification requires a parser to reject any XML document that does not follow the basic rules.** Most HTML parsers will accept sloppy markup, making a guess as to what the writer of the document intended. To avoid the loosely structured mess found in the average HTML document, the creators of XML decided to enforce document structure from the beginning. (By the way, if you are not familiar with the term, a *parser* is a piece of code that attempts to read a document and interpret its contents.).

Invalid, valid, and well-formed documents

There are three kinds of XML documents:

- **Invalid documents** do not follow the syntax rules defined by the XML specification. If a developer has defined rules for what the document can contain in a DTD or schema, and the document does not follow those rules, that document is invalid as well. Valid documents follow both the XML syntax rules and the rules defined in their DTD or schema.
- **Well-formed documents** follow the XML syntax rules but do not have a DTD or schema.

3.9.2 The Root Element

An XML document must be contained in a single element. That single element is called the **root element**, and it contains all the text and other elements in the document. In the following example, the XML document is contained in a single element, the <greeting> element.

Notice that the document has a comment that is outside the root element; that's perfectly legal.

- <?xml version="1.0"?>
- <!-- A well-formed document -->
- <greeting>
- Hello, World!
- </greeting>

Here is a document that does not contain a single root element:

- <?xml version="1.0"?>
- <!-- An invalid document -->
- <greeting>
- Hello, World!
- </greeting>
- <greeting>
- Hola, el Mundo!
- </greeting>

An XML parser is required to reject this document, regardless of the information it might contain.

3.9.3 Elements Cannot Overlap

XML elements cannot overlap. Here is some markup that is not legal:

- <!-- NOT legal XML markup -->
-
- I really love XML.
-
-

If you begin a element inside a element, you have to end it there as well. If you want the text XML to appear in italics, you need to add a second element to correct the markup:

- <!-- legal XML markup -->
-
- I really
- love
- XML.
-

An XML parser will accept only this markup; the HTML parsers in most web browsers will accept both.

3.9.4 End Tags are Required

You can not leave out any end tags. In the first example below, the markup is not legal because there are no end paragraph ( ) tags. While this is acceptable in HTML (and, in some cases, SGML), an XML parser will reject it.

- <!-- NOT legal XML markup -->
- Yada yada yada...
- Yada yada yada...
- ...

If an element contains no markup at all it is called an **empty element**; the HTML break (
) and image (<img>) elements are two examples. In empty elements in XML documents, you can put the closing slash in the start tag. The two break elements and the two image elements below mean the same thing to an XML parser:

- <!-- Two equivalent break elements -->
-
</br>
-
- <!-- Two equivalent image elements -->
- <img src="../img/c.gif"></img>
-

3.9.5 Elements are Case-Sensitive

XML elements are case-sensitive. In HTML, <h1> and <H1> are the same; in XML, they are not. If you try to end an <h1> element with a </H1> tag, you will get an error. In the example below, the heading at the top is illegal, while the one at the bottom is fine.

- <!-- NOT legal XML markup -->
- <h1>Elements are
- case sensitive</H1>
- <!-- legal XML markup -->
- <h1>Elements are
- case sensitive</h1>

3.9.6 Attributes Must Have Quoted Values

There are two rules for attributes in XML documents:

- Attributes must have values
- Those values must be enclosed within quotation marks

Compare the two examples below. The markup at the top is legal in HTML, but not in XML. To do the equivalent in XML, you have to give the attribute a value, and you have to enclose it in quotes.

<!-- NOT legal XML markup -->
<ol compact>
<!-- legal XML markup -->
<ol compact="yes">

You can use either single or double quotes, just as long as you are consistent.

If the value of the attribute contains a single or double quote, you can use the other kind of quote to surround the value (as in name="Doug’s car"), or use the entities " for a double quote and ' for a single quote. An *entity* is a symbol, such as ", that the XML parser replaces with other text, such as ".

3.9.7 XML Declarations

Most XML documents start with an *XML declaration* that provides basic information about the document to the parser. An XML declaration is recommended, but not required. If there is one, it must be the first thing in the document.

The declaration can contain up to three name-value pairs (many people call them attributes, although technically they are not). The version is the version of XML used; currently this value must be 1.0. The encoding is the character set used in this document. The ISO-8859-1 character set referenced in this declaration includes all of the characters used by most Western European languages. If no encoding is specified, the XML parser assumes that the characters are in the UTF-8 set, a Unicode standard that supports virtually every character and ideograph from the world's languages.

<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>

Finally, standalone, which can be either yes or no, defines whether this document can be processed without reading any other files. For

example, if the XML document does not reference any other files, you would specify standalone="yes". If the XML document references other files that describe what the document can contain (more about those files in a minute), you could specify standalone="no". Because standalone="no" is the default, you rarely see standalone in XML declarations.

3.9.8 Other Things in XML Documents

There are a few other things you might find in an XML document:

Comments: Comments can appear anywhere in the document; they can even appear before or after the root element. A comment begins with <!-- and ends with -->. A comment cannot contain a double hyphen (--) except at the end; with that exception, a comment can contain anything. Most importantly, any markup inside a comment is ignored; if you want to remove a large section of an XML document, simply wrap that section in a comment. (To restore the commented-out section, simply remove the comment tags.) Here is some markup that contains a comment:

- <!—Here's a PI for Cocoon: -->
- <?cocoon-process type="sql"?>

Processing instructions: A processing instruction is markup intended for a particular piece of code. In the example above, there is a processing instruction (sometimes called a PI) for Cocoon, an XML processing framework from the Apache Software Foundation. When Cocoon is processing an XML document, it looks for processing instructions that begin with cocoon-process, then processes the XML document accordingly. In this example, the type="sql" attribute tells Cocoon that the XML document contains a SQL statement.

<!-- Here's an entity: -->
<!ENTITY dw "developerWorks">

Entities: The example above defines an *entity* for the document. Anywhere the XML processor finds the string &dw;, it replaces the entity with the string developerWorks. The XML spec also defines five entities you can use in place of various special characters. The entities are:

- < for the less-than sign
- > for the greater-than sign
- " for a double-quote
- ' for a single quote (or apostrophe)
- & for an ampersand.

3.9.9 Namespaces

XML's power comes from its flexibility, the fact that you and I and millions of other people can define our own tags to describe our data. Remember the sample XML document for a person's name and address? That document includes the <title>element for a person's courtesy title, a perfectly reasonable choice for an element name. If you run an online bookstore, you might create a <title> element for the title of a book. If you run an online mortgage company, you might create a <title> element for the title to a piece of property. All of those are reasonable choices, but all of them create elements with the same name. How do you tell if a given <title>element refers to a person, a book, or a piece of property? With *namespaces*.

To use a namespace, you define a *namespace prefix* and map it to a particular string. Here is how you might define namespace prefixes for our three <title> elements:

- <?xml version="1.0"?>
- <customer_summary
- xmlns:addr="http://www.xyz.com/addresses/"
- xmlns:books="http://www.zyx.com/books/"
- xmlns:mortgage="http://www.yyz.com/title/">
- ... <addr:name><title>Mrs.</title> ... </addr:name> ...
- ... <books:title>Lord of the Rings</books:title> ...
- ... <mortgage:title>NC2948-388-1983</mortgage:title> ...

In this example, the three namespace prefixes are addr, books, and mortgage.

Notice that defining a namespace for a particular element means that all of its child elements belong to the same namespace. The first <title> element belongs to the addr namespace because its parent element, <addr:Name>, does. One final point: **The string in a namespace definition is just a string.** Yes, these strings look like URLs, but they are not. You could define xmlns:addr="mike" and that would work just as well. The only thing that is important about the namespace string is that it is unique; that is why most namespace definitions look like URLs. The XML parser does not go to http://www.zyx.com/books/ to search for a DTD or schema; it simply uses that text as a string. It is confusing, but that is how namespaces work.

3.10 Defining Documents Content

3.10.1 Overview

So far, in this unit you have learned about the basic rules of XML documents; that is all well and good, but you need to define the elements you are going to use to represent data. You will learn two ways of doing that in this section.

One method is to use a **Document Type Definition,** or **DTD.** A DTD defines the elements that can appear in an XML document, the order in which they can appear, how they can be nested inside each other, and other basic details of XML document structure. DTDs are part of the original XML specification and are very similar to SGML DTDs.

The other method is to use an **XML Schema.** A schema can define all of the document structures that you can put in a DTD, and it can also define data types and more complicated rules than a DTD can. The W3C developed the XML Schema specification a couple of years after the original XML spec.

3.10.2 Document Type Definitions

A DTD allows you to specify the basic structure of an XML document. The next couple of panels look at fragments of DTDs. First of all, here is a DTD that defines the basic structure of the address document example in the section, What is XML? :

- <!-- address.dtd -->
- <!ELEMENT address (name, street, city, state, postal-code)>
- <!ELEMENT name (title? first-name, last-name)>
- <!ELEMENT title (#PCDATA)>
- <!ELEMENT first-name (#PCDATA)>
- <!ELEMENT last-name (#PCDATA)>
- <!ELEMENT street (#PCDATA)>
- <!ELEMENT city (#PCDATA)>
- <!ELEMENT state (#PCDATA)>
- <!ELEMENT postal-code (#PCDATA)>

This DTD defines all of the elements used in the sample document. It defines three basic things:

- An <address> element contains a <name>, a <street>, a <city>, a <state>, and a <postal-code>. All of those elements must appear, and they must appear in that order.

- A <name> element contains an optional <title> element (the question mark means the title is optional), followed by a <first-name> and a <last-name> element.
- All of the other elements contain text. (#PCDATA stands for parsed character data; you cannot include another element in these elements.) Although the DTD is pretty simple, it makes it clear what combinations of elements are legal. An address document that has a <postal-code> element before the <state> element is not legal, and neither is one that has no <last-name> element.

Also, **notice that DTD syntax is different from ordinary XML syntax.** (XML Schema documents, by contrast, are themselves XML, which has some interesting consequences.) Despite the different syntax for DTDs, you can still put an ordinary comment in the DTD itself.

3.10.3 Symbols in DTDs

There are a few symbols used in DTDs to indicate how often (or whether) something may appear in an XML document. Here are some examples, along with their meanings:

- <!ELEMENT address (name, city, state)>
  The <address> element must contain a <name>, a <city>, and a <state> element, in that order. All of the elements are required. **The comma indicates a list of items.**
- <!ELEMENT name (title?, first-name, last-name)>
  This means that the <name> element contains an optional <title> element, followed by a mandatory <first-name> and a <last-name> element. **The question mark indicates that an item is optional; it can appear once or not at all.**
- <!ELEMENT addressbook (address+)>
  An <addressbook> element contains one or more <address> elements. You can have as many <address> elements as you need, but there has to be at least one.**The plus sign indicates that an item must appear at least once, but can appear any number of times.**
- <!ELEMENT private-addresses (address*)>
  A <private-addresses> element contains zero or more <address> elements. **The asterisk indicates that an item can appear any number of times, including zero.**
- <!ELEMENT name (title?, first-name, (middle-initial |middle-name)?, last-name)>
  A <name> element contains an optional <title> element, followed by a <first-name> element, possibly followed by either a <middle-initial> or a <middle-name> element, followed by a

<last-name> element. In other words, both <middle-initial> and <middle-name> are optional, and you can have only one of the two. **Vertical bars indicate a list of choices; you can choose only one item from the list.** Also notice that this example uses parentheses to group certain elements, and it uses a question mark against the group.

- <!ELEMENT name ((title?, first-name, last-name) | (surname, mothers-name, given-name))>
  The <name> element can contain one of two sequences: An optional <title>, followed by a <first-name> and a <last-name>; or a <surname>, a <mothers-name>, and a <given-name>.

3.10.4 A Word about Flexibility

Before going on, a quick note about designing XML document types for flexibility. Consider the sample name and address document type; I clearly wrote it with U.S. postal addresses in mind. If you want a DTD or schema that defines rules for other types of addresses, you would have to add a lot more complexity to it. Requiring a <state> element might make sense in Australia, but it would not in the UK. A Canadian address might be handled by the sample DTD, but adding a <province> element is a better idea. Finally, be aware that in many parts of the world, concepts like title, first name, and last name do not make sense. The bottom line: If you are going to define the structure of an XML document, you should put as much forethought into your DTD or schema as you would if you were designing a database schema or a data structure in an application. The more future requirements you can foresee, the easier and cheaper it will be for you to implement them later.

310.5 Defining Attributes

This introductory unit does not go into great detail about how DTDs work, but there is one more basic topic to cover here: defining attributes. You can define attributes for the elements that will appear in your XML document. Using a DTD, you can also:

- define which attributes are required
- define default values for attributes
- list all of the valid values for a given attribute

Suppose that you want to change the DTD to make state an attribute of the <city> element. Here is how to do that:

- <!ELEMENT city (#PCDATA)>
- <!ATTLIST city state CDATA #REQUIRED>

This defines the <city> element as before, but the revised example also uses an ATTLIST declaration to list the attributes of the element. The name city inside the attribute list tells the parser that these attributes are defined for the <city> element. The name state is the name of the attribute, and the keywords CDATA and #REQUIRED tell the parser that the state attribute contains text and is required (if it's optional, CDATA #IMPLIED will do the trick).

To define multiple attributes for an element, write the ATTLIST like this:

- <!ELEMENT city (#PCDATA)>
- <!ATTLIST city state CDATA #REQUIRED
- postal-code CDATA #REQUIRED>

This example defines both state and postal-code as attributes of the <city> element.

Finally, DTDs allow you to define default values for attributes and enumerate all of the valid values for an attribute:

- <!ELEMENT city (#PCDATA)>
- <!ATTLIST city state CDATA (AZ|CA|NV|OR|UT|WA) "CA">

The example here indicates that it only supports addresses from the states of Arizona (AZ), California (CA), Nevada (NV), Oregon (OR), Utah (UT), and Washington (WA), and that the default state is California. Thus, you can do a very limited form of data validation. While this is a useful function, it is a small subset of what you can do with XML schemas.

3.10.6 XML Schemas

With XML schemas, you have more power to define what valid XML documents look like. They have several advantages over DTDs:

- **XML schemas use XML syntax.** In other words, an XML schema is an XML document. That means you can process a schema just like any other document. For example, you can write an XSLT style sheet that converts an XML schema into a Web form complete with automatically-generated JavaScript code that validates the data as you enter it.
- **XML schemas support datatypes.** While DTDs do support datatypes, it is clear those datatypes were developed from a publishing perspective. XML schemas support all of the original datatypes from DTDs (things like IDs and ID references).They

also support integers, floating point numbers, dates, times, strings, URLs, and other datatypes useful for data processing and validation.

- **XML schemas are extensible.** In addition to the datatypes defined in the XML schema specification, you can also create your own, and you can derive new datatypes based on other datatypes.
- **XML schemas have more expressive power.** For example, with XML schemas you can define that the value of any <state> attribute cannot be longer than 2 characters, or that the value of any <postal-code> element must match the regular expression [0-9]{5}(-[0-9]{4})?. You cannot do either of those things with DTDs.

3.10.7 A Sample XML Schema

Here is an XML schema that matches the original name and address DTD. It adds two constraints: The value of the <state> element must be exactly two characters long and the value of the <postal-code> element must match the regular expression [0-9]{5}(-[0-9]{4})?. Although the schema is much longer than the DTD, it expresses more clearly what a valid document looks like. Here's the schema:

- <?xml version="1.0" encoding="UTF-8"?>
- <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
- <xsd:element name="address">
- <xsd:complexType>
- <xsd:sequence>
- <xsd:element ref="name"/>
- <xsd:element ref="street"/>
- <xsd:element ref="city"/>
- <xsd:element ref="state"/>
- <xsd:element ref="postal-code"/>
- </xsd:sequence>
- </xsd:complexType>
- </xsd:element>
- <xsd:element name="name">
- <xsd:complexType>
- <xsd:sequence>
- <xsd:element ref="title" minOccurs="0"/>
- <xsd:element ref="first-Name"/>
- <xsd:element ref="last-Name"/>
- </xsd:sequence>
- </xsd:complexType>

- </xsd:element>
- <xsd:element name="title" type="xsd:string"/>
- <xsd:element name="first-Name" type="xsd:string"/>
- <xsd:element name="last-Name" type="xsd:string"/>
- <xsd:element name="street" type="xsd:string"/>
- <xsd:element name="city" type="xsd:string"/>
- <xsd:element name="state">
- <xsd:simpleType>
- <xsd:restriction base="xsd:string">
- <xsd:length value="2"/>
- </xsd:restriction>
- </xsd:simpleType>
- </xsd:element>
- <xsd:element name="postal-code">
- <xsd:simpleType>
- <xsd:restriction base="xsd:string">
- <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
- </xsd:restriction>
- </xsd:simpleType>
- </xsd:element>
- </xsd:schema>

3.10.8 Defining Elements in Schemas

The XML schema in A sample XML schema defined a number of XML elements with the <xsd:element> element. The first two elements defined, <address> and <name>, are composed of other elements. The <xsd:sequence> element defines the sequence of elements that are contained in each. Here's an example:

- <xsd:element name="address">
- <xsd:complexType>
- <xsd:sequence>
- <xsd:element ref="name"/>
- <xsd:element ref="street"/>
- <xsd:element ref="city"/>
- <xsd:element ref="state"/>
- <xsd:element ref="postal-code"/>
- </xsd:sequence>
- </xsd:complexType>
- </xsd:element>
- As in the DTD version, the XML schema example defines that an <address> contains a <name>, a <street>, a <city>, a <state>, and a <postal-code> element, in that order. Notice that the schema

actually defines a new datatype with the <xsd:complexType> element.

Most of the elements contain text; defining them is simple. You merely declare the new element, and give it a datatype of xsd:string:

- <xsd:element name="title" type="xsd:string"/>
- <xsd:element name="first-Name" type="xsd:string"/>
- <xsd:element name="last-Name" type="xsd:string"/>
- <xsd:element name="street" type="xsd:string"/>
- <xsd:element name="city" type="xsd:string"/>

3.10.9 Defining Element Content in Schemas

The sample schema defines constraints for the content of two elements: The content of a <state> element must be two characters long, and the content of a <postal-code> element must match the regular expression [0-9]{5}(-[0-9]{4})?. Here's how to do that:

- <xsd:element name="state">
- <xsd:simpleType>
- <xsd:restriction base="xsd:string">
- <xsd:length value="2"/>
- </xsd:restriction>
- </xsd:simpleType>
- </xsd:element>
- <xsd:element name="postal-code">
- <xsd:simpleType>
- <xsd:restriction base="xsd:string">
- <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
- </xsd:restriction>
- </xsd:simpleType>
- </xsd:element>

For the <state> and <postal-code> elements, the schema defines new data types with restrictions. The first case uses the <xsd:length> element, and the second uses the <xsd:pattern> element to define a regular expression that this element must match.

This summary only scratches the surface of what XML schemas can do; there are entire books written on the subject. For the purpose of this introduction, suffice to say that XML schemas are a very powerful and flexible way to describe what a valid XML document looks like.

3.11 XML Programming Interfaces

3.11.1 Overview

This section takes a look at a variety of programming interfaces for XML. These interfaces give developers a consistent interface for working with XML documents. There are many APIs available; this section looks at four of the most popular and generally useful ones: the Document Object Model (DOM), the Simple API for XML (SAX), JDOM, and the Java API for XML Parsing (JAXP).

3.11.2 The Document Object Model

The Document Object Model, commonly called the DOM, defines a set of interfaces to the parsed version of an XML document. The parser reads in the entire document and builds an in-memory tree, so your code can then use the DOM interfaces to manipulate the tree. You can move through the tree to see what the original document contained, you can delete sections of the tree; you can rearrange the tree, add new branches, and so on. The DOM was created by the W3C, and is an Official Recommendation of the consortium.

3.11.3 DOM Issues

The DOM provides a rich set of functions that you can use to interpret and manipulate an XML document, but those functions come at a price. As the original DOM for XML documents was being developed, a number of people on the XML-DEV mailing list voiced concerns about it:

- The DOM builds an in-memory tree of an entire document. If the document is very large, this requires a significant amount of memory.
- The DOM creates objects that represent everything in the original document, including elements, text, attributes, and whitespace. If you only care about a small portion of the original document, it is extremely wasteful to create all those objects that will never be used.
- A DOM parser has to read the entire document before your code gets control. For very large documents, this could cause a significant delay. These are merely issues raised by the design of the Document Object Model; despite these concerns, **the DOM API is a very useful way to parse XML documents.**

3.11.4 The Simple API for XML

To get around the DOM issues, the XML-DEV participants (led by David Megginson) created the SAX interface. SAX has several characteristics that address the concerns about the DOM:

- A SAX parser sends events to your code. The parser tells you when it finds the start of an element, the end of an element, text, the start or end of the document, and so on. You decide which events are important to you, and you decide what kind of data structures you want to create to hold the data from those events. If you do not explicitly save the data from an event, it is discarded.
- A SAX parser does not create any objects at all; it simply delivers events to your application. If you want to create objects based on those events, it is up to you.
- A SAX parser starts delivering events to you as soon as the parse begins. Your code will get an event when the parser finds the start of the document, when it finds the start of an element, when it finds text, and so on. Your application starts generating results right away; you do not have to wait until the entire document has been parsed.

Even better, if you are only looking for certain things in the document, your code can throw an exception once it is found what it is looking for. The exception stops the SAX parser, and your code can do whatever it needs to do with the data it has found.

Having said all of these things, both SAX and DOM have their place. The remainder of this section discusses why you might want to use one interface or the other.

3.11.5 SAX Issues

To be fair, SAX parsers also have issues that can cause concern:

- SAX events are stateless. When the SAX parser finds text in an XML document, it sends an event to your code. That event simply gives you the text that was found; it does not tell you what element contains that text. If you want to know that, you have to write the state management code yourself.
- SAX events are not permanent. If your application needs a data structure that models the XML document, you have to write that code yourself. If you need to access data from a SAX event, and you did not store that data in your code, you have to parse the document again.

- SAX is not controlled by a centrally managed organisation. Although **this has not caused a problem to date**, some developers would feel more comfortable if SAX were controlled by an organisation such as the W3C.

3.11.6 JDOM

Frustrated by the difficulty in doing certain tasks with the DOM and SAX models, Jason Hunter and Brett McLaughlin created the JDOM package. JDOM is a Java technology-based, open source project that attempts to follow the 80/20 rule: Deliver what 80% of users need with 20% of the functions in DOM and SAX. JDOM works with SAX and DOM parsers, so it is implemented as a relatively small set of Java classes.

The main feature of JDOM is that it greatly reduces the amount of code you have to write. Although this introductory unit does not discuss programming topics in depth, JDOM applications are typically one-third as long as DOM applications, and about half as long as SAX applications. (DOM purists, of course, suggest that learning and using the DOM is good discipline that will pay off in the long run.) JDOM does not do everything, but for most of the parsing you want to do, it is probably just the thing.

3.11.7 The Java API for XML Parsing

Although DOM, SAX, and JDOM provide standard interfaces for most common tasks, there are still several things they do not address. For example, the process of creating a DOMParser object in a Java programme differs from one DOM parser to the next. To fix this problem, Sun has released JAXP, the Java API for XML Parsing. This API provides common interfaces for processing XML documents using DOM, SAX, and XSLT. JAXP provides interfaces such as the DocumentBuilderFactory and the DocumentBuilder that provide a standard interface to different parsers. There are also methods that allow you to control whether the underlying parser is namespace-aware and whether it uses a DTD or schema to validate the XML document.

Determining the right interface

To determine which programming interface is right for you, you need to understand the design points of all of the interfaces, and you need to understand what your application needs to do with the XML documents you are going to process. Consider these questions to help you find the right approach.

- **Will your application be written in Java?** JAXP works with DOM, SAX, and JDOM; if you are writing your code in Java, you should use JAXP to isolate your code from the implementation details of various parsers.
- **How will your application be deployed?** If your application is going to be deployed as a Java applet, and you want to minimise the amount of downloaded code, keep in mind that SAX parsers are smaller than DOM parsers. Also be aware that using JDOM requires a small amount of code in addition to the SAX or DOM parser.
- **Once you parse the XML document, will you need to access that data many times?** If you need to go back to the parsed version of the XML file, DOM is probably the right choice. When a SAX event is fired, it is up to you (the developer) to save it somehow if you need it later. If you need to access an event you did not save.

3.11.8 Which Interface is Right for You?

To determine which programming interface is right for you, you need to understand the design points of all of the interfaces, and you need to understand what your application needs to do with the XML documents you are going to process. Consider these questions to help you find the right approach.

- **Will your application be written in Java?** JAXP works with DOM, SAX, and JDOM; if you are writing your code in Java, you should use JAXP to isolate your code from the implementation details of various parsers.
- **How will your application be deployed?** If your application is going to be deployed as a Java applet, and you want to minimise the amount of downloaded code, keep in mind that SAX parsers are smaller than DOM parsers. Also be aware that using JDOM requires a small amount of code in addition to the SAX or DOM parser.
- **Once you parse the XML document, will you need to access that data many times?** If you need to go back to the parsed version of the XML file, DOM is probably the right choice. When a SAX event is fired, it is up to you (the developer) to save it somehow if you need it later. If you need to access an event you did not save, frustrated by the difficulty in doing certain tasks with the DOM and SAX models, Jason Hunter and Brett McLaughlin created the JDOM package. JDOM is a Java technology-based, open source project that attempts to follow the 80/20 rule: Deliver what 80% of users need with 20% of the functions in DOM and SAX. JDOM works with SAX and DOM

parsers, so it is implemented as a relatively small set of Java classes.

The main feature of JDOM is that it greatly reduces the amount of code you have to write. Although this introductory unit does not discuss programming topics in depth, JDOM applications are typically one-third as long as DOM applications, and about half as long as SAX applications. (DOM purists, of course, suggest that learning and using the DOM is good discipline that will pay off in the long run.) JDOM does not do everything, but for most of the parsing you want to do, it is probably just the thing.

3.12 XML Standards

3.12.1 Overview

A variety of standards exist in the XML universe. In addition to the base XML standard, other standards define schemas, style sheets, links, web services, security, and other important items. This section covers the most popular standards for XML, and points you to references to find other standards.

3.12.2 The XML Specification

This spec, located at *w3.org/TR/REC-xml,* defines the basic rules for XML documents. All of the XML document rules discussed earlier in this unit is defined here. In addition to the basic XML standard, the Namespaces spec is another important part of XML. You can find the namespaces standard at the W3C as well:*w3.org/TR/REC-xml-names/.*

3.12.3 XML Schema

The XML Schema language is defined in three parts:

- **A primer**, located at *w3.org/TR/xmlschema-0* , that gives an introduction to XML schema documents and what they are designed to do;
- A standard for **document structures**, located at *w3.org/TR/xmlschema-1* , that illustrates how to define the structure of XML documents;
- A standard for **data types**, located at *w3.org/TR/xmlschema-2* , that defines some common data types and rules for creating new ones. This unit discussed schemas briefly in Defining document content; if you want the complete details on all the things you can do with XML schemas, the primer is the best place to start.

3.12.4 XSL, XSLT, and XPath

The Extensible Stylesheet Language, XSL, defines a set of elements (called formatting objects) that describes how data should be formatted. For clarity, this standard is often referred to as XSL-FO to distinguish it from XSLT. Although it is primarily designed for generating high-quality printable documents, you can also use formatting objects to generate audio files from XML. The XSL-FO standard is at *w3.org/TR/xsl/*.

The Extensible Stylesheet Language for Transformations, XSLT, is an XML vocabulary that describes how to convert an XML document into something else. The standard is at *w3.org/TR/xslt* (no closing slash).

XPath, the XML Path Language, is a syntax that describes locations in XML documents. You use XPath in XSLT style sheets to describe which portion of an XML document you want to transform. XPath is used in other XML standards as well, which is why it is a separate standard from XSLT. XPath is defined at *w3.org/TR/xpath* (no closing slash).

3.12.5 DOM

The Document Object Model defines how an XML document is converted to an in-memory tree structure. The DOM is defined in a number of specifications at the W3C:

- **The Core DOM** defines the DOM itself, the tree structure, and the kinds of nodes and exceptions your code will find as it moves through the tree. The complete spec is at *w3.org/TR/DOM-Level-2-Core/*.
- **Events** define the events that can happen to the tree, and how those events are processed. This specification is an attempt to reconcile the differences in the object models supported by Netscape and Internet Explorer since Version 4 of those browsers. This spec is at *w3.org/TR/DOM-Level-2-Events/* .
- **Style** defines how XSLT style sheets and CSS style sheets can be accessed by a programme. This spec is at *w3.org/TR/DOM-Level-2-Style/*
- **Traversals and Ranges** define interfaces that allow programmes to traverse the tree or define a range of nodes in the tree. You can find the complete spec at *w3.org/TR/DOM-Level-2-Traversal-Range/*.
- **Views** define an AbstractView interface for the document itself. See *w3.org/TR/DOM-Level-2-Views/* for more information.

3.12.6 SAX, JDOM, and JAXP

The Simple API for XML defines the events and interfaces used to interact with a SAX-compliant XML parser. You can find the complete SAX specification at *www.saxproject.org*.

The JDOM project was created by Jason Hunter and Brett McLaughlin and lives at *jdom.org/*. At the JDOM site, you can find code, sample programmes, and other tools to help you get started. (For *developerWorks* articles on JDOM, see Resources on page 32).

One significant point about SAX and JDOM is that both of them came from the XMLdeveloper community, not a standards body. Their wide acceptance is a tribute to the active participation of XML developers worldwide.

You can find out everything there is to know about JAXP at *java.sun.com/xml/jaxp/*.

3.12.7 Linking and Referencing

There are two standards for linking and referencing in the XML world: XLink and XPointer:

- **XLink,** the XML Linking Language, defines a variety of ways to link different resources together. You can do normal point-to-point links (as with the HTML <a> element) or extended links, which can include multipoint links, links through third parties, and rules that define what it means to follow a given link. The XLink standard is at *w3.org/TR/xlink/*.
- **XPointer,** the XML Pointer Language, uses XPath as a way to reference other resources. It also includes some extensions to XPath. You can find the spec at *www.w3.org/TR/xptr/*.

3.12.8 Security

There are two significant standards that address the security of XML documents. One is the **XML Digital Signature** standard (*w3.org/TR/xmldsig-core/)*, which defines an XML document structure for digital signatures. You can create an XML digital signature for any kind of data, whether it is an XML document, an HTML file, plain text, binary data, and so on. You can use the digital signature to verify that a particular file was not modified after it was signed. If the data you are signing is an XML document, you can embed the XML document in the signature file itself, which makes processing the data and the signature very simple.

The other standard addresses encrypting XML documents. While it is great that XML documents can be written so that a human can read and understand them, this could mean trouble if a document fell into the wrong hands. The **XML Encryption** standard (*w3.org/TR/xmlenc-core/*) defines how parts of an XML document can be encrypted. Using these standards together, you can use XML documents with confidence. I can digitally sign an important XML document, generating a signature that includes the XML document itself. I can then encrypt the document (using my private key and your public key) and send it to you. When you receive it, you can decrypt the document with your private key and my public key; that lets you know that I'm the one who sent the document. (If need be, you can also prove that I sent the document.) Once you have decrypted the document, you can use the digital signature to make sure the document has not been modified in any way.

3.13 Web Services

Web services are an important new kind of application. A web service is a piece of code that can be discovered, described, and accessed using XML. There is a great deal of activity in this space, but the three main XML standards for web services are:

- **SOAP:** Originally the Simple Object Access Protocol, SOAP defines an XML document format that describes how to invoke a method of a remote piece of code.
- My application creates an XML document that describes the method I want to invoke, passing it any necessary parameters, and then it sends that XML document across a network to that piece of code. The code receives the XML document, interprets it, invokes the method I requested, then sends back an XML document that describes the results. Version 1.1 of the SOAP spec is at *w3.org/TR/SOAP/*. Visit *w3.org/TR/* to see all of the W3C's SOAP-related activities.
- **WSDL:** The Web Services Description Language is an XML vocabulary that describes a web service. It is possible to write a piece of code that takes a WSDL document and invokes a web service it is never seen before. The information in the WSDL file defines the name of the web service, the names of its methods, the arguments to those methods, and other details. You can find the latest WSDL spec at *w3.org/TR/wsdl* (no closing slash).
- **UDDI:** The Universal Description, Discovery, and Integration protocol defines a SOAP interface to a registry of web services. If you have a piece of code that you would like to deploy as a web service, the UDDI spec defines how to add the description of your service to the registry. If you are looking for a piece of code that provides a certain function, the UDDI spec defines how to

query the registry to find what you want. The source of all things UDDI is *uddi.org*.

3.14 Other Standards

A number of other XML standards exist. In addition to widely-applicable standards like Scalable Vector Graphics (*www.w3.org/TR/SVG/*) and SMIL, the Synchronised Multimedia Integration Language (*www.w3.org/TR/smil20/*), there are many industry-specific standards. For example, the HR-XML Consortium has defined a number of XML standards for Human Resources; you can find those standards at *hr-xml.org*.

Finally, for a good source of XML standards, visit the XML Repository at *xml.org/xml/registry.jsp*. This site features hundreds of standards for a wide variety of industries.

3.15 Case Studies

a. Real-world examples

At this point, I hope you are convinced that XML has tremendous potential to revolutionise the way eBusiness works. While potential is great, what really counts is actual results in the marketplace. This section describes three case studies in which organisations have used XML to streamline their business processes and improve their results.

All of the case studies discussed here come from IBM's jStart program. The jStart team exists to help customers use new technologies to solve problems. When a customer agrees to a jStart engagement, the customer receives IBM consulting and development services at a discount, with the understanding that the resulting project will be used as a case study. If you would like to see more case studies, including case studies involving web services and other new technologies, visit the jStart Web page at *ibm.com/software/jstart*.

Be aware that the jStart team is no longer doing engagements for XML projects; the team's current focus is Web services engagements. Web services use XML in a specialized way, typically through the SOAP, WSDL, and UDDI standards mentioned earlier in Web services.

b. A messaging-based system

The bank's distributed applications are built on a messaging infrastructure, using IBM's MQSeries to deliver messages to the OS/390 system. The message content is based on a specification called the

Common Interface Message (CIM), a First Union proprietary standard. Both the front-end and back-end components of the application are dependent on the message format. Using XML as the data format isolates both sides of the application from future changes and additions to the messaging protocol.

c. Using XML tools to automate data flows

In developing this XML-based application, the First Union and IBM team created a service that converts the CIM into an XML document. Another part of the application converts the XML request into the appropriate format for the back-end processing systems. Finally, a third service converts COBOL copy books into DTDs. Once the copy book has been converted into a DTD, First Union can use the DTD and the XML4J parser to validate the XML document automatically; the bank can then be sure that the XML document matches the COBOL data structure that OS/390 expects.

Using Java technology and XML has been very successful for First Union. According to Bill Barnett, Manager of the Distributed Object Integration Team at First Union, "The combination of Java and XML really delivered for us. Without a platform-independent environment like Java and the message protocol independence we received from the use of XML, we would not have the confidence that our distributed infrastructure could evolve to meet the demand from our ever-growing customer base."

4.0 CONCLUSION

At this point, I hope you are convinced that XML is the best way to move and manipulate structured data. If you are not using XML already, how do you get started? Here are some suggestions:

- **Decide what data you want to convert to XML.** Typically this is data that needs to be moved from one system to another, or data that has to be transformed into a variety of formats.
- **See if there are any existing XML standards.** If you are looking at very common data, such as purchase orders, medical records, or stock quotes, chances are good that someone out there has already defined XML standards for that data.
- **See if your existing tools support XML.** If you are using a recent version of a database package, a spreadsheet, or some other data management tool, it is likely that your existing tools (or upgrades to them) can use XML as an input or output format.
- **Learn how to build XML-based applications.** You need to understand how your data is stored now, how it needs to be

transformed, and how to integrate your XML development efforts with your existing applications. Benoît Marchal's *Working XML* column is a great place to start; you can find a current listing of all his columns at *http://www-106.ibm.com/developerworks/xml/library/x-wxxmcol/* .

- **Join the appropriate standards groups.** Consider joining groups like the World-Wide Web Consortium (W3C), as well as industry-specific groups like HR-XML.org. Being a member of these groups will help you keep track of what is happening in the industry, and it gives you the chance to shape the future of XML standards.
- **Avoid proprietary shenanigans.** It is important that you use only standards-based technology in your development efforts; resist the lures of vendors who offer so-called improvements to you. One of XML's advantages is that you have complete control of your data. Once it is held hostage by a proprietary data format, you have given up a tremendous amount of control.
- **Contact the jStart team.** If you think your enterprise could work with the jStart engagement model, contact the team to see what your possibilities are.
- **Stay tuned to *developerWorks*.** Our XML zone has thousands of pages of content that deal with various XML topics, including DTD and schema development, XML programming, and creating XSLT style sheets.

5.0 SUMMARY

In this unit you have been introduced to the fundamental concepts of XML, XML document rules, document content, XML programming interfaces, XML standards and the various Case Studies of XML Data Framework.

6.0 TUTOR-MARKED ASSIGNMENT

i. Briefly explain the concept of XML and elaborate on why XML is required for modern day application development.

ii. Define the following terminologies
   a. error fatal error
   b. at user option validity constraint
   c. well-formedness constraint match
   d. for compatibility for interoperability

iii. Design an XML Document for a personnel address book

iv. Design an XML Document File for New User Email Registration

7.0 REFERENCES/FURTHER READING

Here are some resources to get you started:

The dW XML zone is your one-stop shop for XML resources. See *www-106.ibm.com/developerworks/xml* for everything you always wanted to know about XML. **XML tools:** *developerWorks* has "Fill your XML toolbox" articles that describe XML programming tools for a variety of languages:

C/C++: See Rick Parrish's article at *www-106.ibm.com/developerworks/library/x-ctlbx.html* (*developerWorks*, September 2001).

Java: See Doug Tidwell's article at *www-106.ibm.com/developerworks/library/j-java-xml-toolkit/index.html* (*developerWorks*, May 2000).

Perl: See Parand Tony Darugar's article at *www-106.ibm.com/developerworks/library/x-perl-xml-toolkit/index.html* (*developerWorks*, June 2001).

PHP: See Craig Knudsen's article at *www-106.ibm.com/developerworks/library/x-php-xml-toolkit.html* (*developerWorks*,June 2000).

In addition to these articles, see David Mertz's review of Python XML tools in his article "Charming Python: Revisiting XML tools for Python" at *www-106.ibm.com/developerworks/library/l-pxml.html* .

XML units: Dozens of units on XML topics are available on *developerWorks*; see *http://www-106.ibm.com/developerworks/views/xml/units.jsp* for the latest list.

IBM's jStart team: The jStart team works at very low cost to help customers build solutions using new technology (XML Web services, for example). In return, those customers agree to let IBM publicize their projects as a case study.

UNIT 3 XML AND XML QUERIES

CONTENTS

1.0 INTRODUCTION

In this unit, you will be introduced to the understanding of Extensible Markup Language (XML) and basic XML Queries. You will also learn about central notions of XML.

2.0 OBJECTIVES

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

- define XML
- write and use XML Queries to solve real life problems.

3.0 MAIN CONTENT

3.1 General Introduction

Most web applications have connections to databases and use XML to transfer data from the database to the web application and vice versa. Every major database vendor has proprietary extensions for using XML with relational databases, but they take completely different approaches, and there is no interoperability between them. Many developers need to be able to write applications that work for databases from multiple vendors. XQuery and SQL/XML are two standards that use declarative, portable queries to return XML by querying data. In both standards, the XML can have any desired structure, and the queries can be arbitrarily complex. XQuery is XML-centric, while SQL/XML is SQL-centric.

SQL/XML is an extension of SQL that is part of ANSI/ISO SQL 2003. It lets SQL queries create XML structures with a few powerful XML publishing functions.

For a SQL programmer, SQL/XML is easy to learn because it involves only a few small additions to the existing SQL language. Since SQL is a mature language, there are a lot of tools and infrastructure for SQL. For instance, SQL/XML uses JDBC to return results, and there is currently no equivalent standard API for XQuery. SQL also has functionality not yet found in XQuery, such as updates or stored procedures.

Note

SQL/XML is completely different from Microsoft's SQLXML, a proprietary technology used in SQL Server. The similarity in names has caused a great deal of confusion in the industry.

XQuery is a completely new query language that uses XML as the basis for its data model and type system. It is being developed in the XML Query Working Group [XQWG], which is a part of the World Wide Web Consortium. In this paper, we characterise XQuery as a "Native XML Programming Language". XQuery is based on XML in the same way that SQL is based on the relational model or object-oriented languages are based on the object-oriented model - XML is central to its type system, in which elements and attributes are just as fundamental as integers and strings. Although XQuery per se has no concept of relational data, several products and many projects provide ways to query relational data using an XML view of the database, and the need to make this possible has influenced the design of XQuery throughout its development. XQuery allows you to work in the XML world no

matter what type of data you are working with - relational, XML or object data.

XQuery is ideal for native XML programming. When used with XML views of relational data, it is also ideal for queries data that must represent results as XML, to query XML stored inside or outside the database, or to span relational and XML sources.

For queries based only on relational data, SQL/XML and XQuery have substantially similar functionality. However, the way in which a given task is done is quite different, since SQL/XML operates on the borderline between SQL and XML, and XQuery lives in a purely XML world. Even when the data is all relational, the two languages appeal to very different audiences - SQL/XML is very much an extension of SQL, designed for SQL programmers, and XQuery takes a purely XML view of the world. For queries that span relational and XML sources, XQuery has important advantages.

This talk uses a series of concrete queries written in each language to show the advantages of each. It explains why we need both languages, discussing the ways in which the languages differ and in which they overlap. It also explores the role of SQL/XML mappings as a way of creating XML views for XQuery.

3.2 XML and Relational – Opposites Attract

XML and relational databases are tightly wed in most web applications, but a look at the two models shows that it is an unlikely marriage - though a necessary one. The relational model is based on two dimensional tables which have neither hierarchy nor significant order. XML is based on trees in which order is significant. In the relational model, neither hierarchy nor sequence may be used to model information; in XML, hierarchy and sequence are the main ways to represent information. Although this is one of the more fundamental differences between the two models, it is by no means the only one.

In many environments, the same information is represented in relational databases when it is stored or queried, but in XML when it is exchanged or displayed on web pages. These representations are often completely different due to the differences in the models.

On web pages, XML is useful because the structure of XML closely matches the structure used to display the same information in HTML. If you look at web pages, they often use a distinctly hierarchical structure to present data for users - after all, users do not want to look at a bunch of tables and do joins in their head.

But most of the data for these web pages comes from relational databases, and needs to be converted to appropriate XML hierarchies.

For web messages, the format of a web message is often specified by a standards organisation or a trade partner, and these formats are generally hierarchical. Again, the data for a web message generally comes from relational data, and the consumer of a web message often needs to put data into a relational database.

For instance, suppose a consulting company needs to represent a set of projects and the companies for whom the projects are being done. In a relational database, this might be represented by the following tables:

```
        Projects

ProjId      Name                 CustId
----------- -------------------- -----------
1           Medusa               1
2           Pegasus              4
8           Typhon               4
10          Sphinx               5
```

```
         Customers

CustId      Name                  City
----------- --------------------- --------------
1           Woodworks             Baltimore
2           Software Solutions    Boston
3           Food Supplies         New York
4           Hardware Shop         Washington
5           Books Inc.            New Orleans
```

In SQL, if we want to see the projects associated with each customer, we would do the following query:

```
select *
from Customers c, Projects p
where c.CustId = p.CustId
order by c.CustId, p.ProjId
```

Here is the output of the above query:

CustId	CustName	City	ProjId	ProjName
1	Woodworks	Baltimore	1	Medusa
4	Hardware Shop	Washington	2	Pegasus
4	Hardware Shop	Washington	8	Typhon
5	Books Inc.	New Orleans	10	Sphinx

Suppose we want to translate this information into XML for use on a web page, in a document, or in a web message. Like most XML applications, we will leverage the hierarchy of XML to express relationships, listing the projects for each customer within the element that represents the customer:

```
<?xml version="1.0" encoding="UTF-8"?>
<customers>
    <customer id="1">
        <name>Woodworks</name>
        <city>Baltimore</city>
            <projects>
                <project id="1"><name>Medusa</name></project>
            </projects>
    </customer>
    <customer id="4">
       <name>Hardware Shop</name>
       <city>Washington</city>
            <projects>
                <project id="2"><name>Pegasus</name></project>
                <project id="8"><name>Typhon</name></project>
            </projects>
    </customer>
    <!-- !!! SNIP !!! -->
</customers>
```

Note that in the original SQL tables, each customer is represented only once. This is also true of the XML. The SQL result set, however, contains multiple rows for a given customer if that customer is associated with more than one project, and these rows contain duplicate information. Translating this result set into the desired XML is tedious for the programmer. And just as a single relational database may be used with an infinite number of queries, it may also be used to create an infinite number of XML documents with different structures. Today, many programmers spend a great deal of time doing this kind of translation.

3.3 XML and Relational: Four Approaches

XML applications that use relational data can choose from four approaches, each with distinct advantages and disadvantages. The first three of these are compared in some detail, with code samples, in [SQL/XML-JDBC].

The programmer can use JDBC or ODBC together with SAX or DOM and perhaps XSLT to transform the results of SQL queries to XML. For instance, the programme might first query for customers, then perform an additional query to find the projects associated with each customer. This is inefficient because of the number of queries required.

Another approach would be to use SQL to create a table that lists customers and their projects, and pick through the rows to determine when a row represents a new customer. This requires more code, but is more efficient. Both of these approaches require significant amounts of tedious code, but they are often used when database independence is important.

The programmer can use the XML extensions provided by the major database vendors. These are based on several different approaches. Some of these are simpler to use or maintainable than others, but they all make the task easier. However, since these extensions are all proprietary, they are not an option when a database-independent solution is needed.

The programmer can use SQL/XML, which is part of SQL 2003. For a SQL programmer, this approach requires little new learning - a small set of XML publishing functions have been added to SQL to allow queries to create any desired XML structure. This approach will be explored with examples in the next section. SQL/XML is being supported by Oracle and IBM, but not by Microsoft. Database-independent implementations of SQL/XML are also available, and can be used with any major relational database. SQL/XML can be used with traditional database APIs such as JDBC.

The programmer can use XQuery, a native XML query language. Since XQuery is a new language, it requires more learning for SQL programmers, but it is likely to be more natural for XML programmers. Unlike SQL/XML, XQuery is optimal for processing XML, and it is also particularly good for applications that must process XML together with relational data, with full support for XML. Most of the major database vendors intend to support XQuery. The first standardized API for XQuery, XQuery for Java (JSR 225), is now being developed under

Java Community Process, and is expected to be available shortly after the XQuery Recommendation is released.

3.4 SQL/XML

SQL/XML refers to the XML extensions of SQL. These are developed by INCITS H2.3, with participation from Oracle, IBM, Microsoft (which does not plan to implement SQL/XML), Sybase, and DataDirect Technologies. In SQL 2003, these extensions include:

- XML Publishing Functions
- The XML Datatype
- Mapping Rules

The XML Publishing Functions are the part that are directly used in a SQL query. The XML Datatype governs the result of a query, and the Mapping Rules determine how SQL data or metadata is represented as XML.

3.5 XML Publishing Functions

The XML Publishing Functions allow SQL to create any desired XML structure. They are part of SQL 2003, and can be used in normal SQL expressions. Here are the XML publishing functions of SQL 2003:

xmlelement()	Creates an XML element, allowing the name to be specified.
xmlattributes()	Creates XML attributes from columns, using the name of each column as the name of the corresponding attribute.
xmlroot()	Creates the root node of an XML document.
xmlcomment()	Creates an XML comment.
xmlpi()	Creates an XML processing instruction.
xmlparse()	Parses a string as XML and returns the resulting XML structure.
xmlforest()	Creates XML elements from columns, using the name of each column as the name of the corresponding element.
xmlconcat()	Combines a list of individual XML values to create a single value containing an XML forest.
xmlagg()	Combines a collection of rows, each containing a single XML value, to create a single value containing an XML forest.

Let us compare a traditional SQL query with one that uses an XML publishing function. Here is a traditional SQL query that shows customers and their associated projects:

```
select c.CustId, c.Name as CustName
from customers c
```

Here is an excerpt of the result:

```
CustId  CustName
------- -------------------
1       Woodworks
4       Hardware Shop
6       Photo Shop
8       Computer Supplies
```

Now let is wrap the result in XML elements using xmlelement(), one of the publishing functions:

```
select xmlelement(name "Customer",
      xmlelement(name "CustId", c.CustId),
      xmlelement(name "CustName", c.Name)
      xmlelement(name "City", c.City))
from Customers  c
```

Each row in the result contains one Customer element. A Customer element looks like this:

```
<Customer>
  <CustId>1</CustId>
  <CustName>Woodworks</CustName>
  <City>Baltimore</City>
</Customer>
```

xmlforest() is an XML publishing function that creates elements from a list of columns, using the name of the column as the name of the element. Using xmlforest() simplifies many queries significantly. For instance, the following query is equivalent to the previous one:

```
select xmlelement(name "Customer",
      xmlforest(c.CustId, c.Name as CustName, c.City))
from Customers  c
```

Now suppose we want to show customers and the projects associated with them. This is easily done with the following SQL query:

```
select *
from Customers c, Projects p
where c.CustId = p.CustId
order by c.CustId, p.ProjId
```

However, the result of this query is that shown in the CustomerProject table in the previous section, with one row for each Customer/Project pair. If a customer is associated with more than one project, there will be a row for that customer for each project. Here is a SQL/XML query that creates the XML equivalent to that table:

```
select xmlelement(name "CustomerProj",
   xmlforest(c.CustId, c.Name as CustName, p.ProjId, p.Name as ProjName))
from Customers c, Projects p
where p.CustId=c.CustId
order by c.CustId
```

Here are the results of this query:

```
<CustomerProj>
  <CustId>1</CustId>
  <CustName>Woodworks</CustName>
  <ProjId>1</ProjId>
  <ProjName>Medusa</ProjName>
</CustomerProj>
<CustomerProj>
  <CustId>4</CustId>
  <CustName>Hardware Shop</CustName>
  <ProjId>2</ProjId>
  <ProjName>Pegasus</ProjName>
</CustomerProj>
<CustomerProj>
  <CustId>4</CustId>
  <CustName>Hardware Shop</CustName>
  <ProjId>8</ProjId>
  <ProjName>Typhon</ProjName>
</CustomerProj>
```

This is a straightforward XML translation of the SQL result set shown in the previous section, but for most XML applications it is not what we would want. Instead, we want to represent each customer once, with a list of that customer's projects, as shown in the XML output in the previous section. In SQL/XML, this can be done by using a sub-query. Here is a subquery that retrieves the projects associated with each customer. In this subquery we use xmlattributes(), an XML publishing function that creates attributes within an element. The names of the attributes are taken from the names of the columns.

```
(select xmlelement(name project,
       xmlattributes(p.ProjId as id),
       xmlforest(p.Name as name))
from Projects p
where p.CustId=c.CustId)
```

Here is the output of the above sub-query when c.CustId is 4:

```
<project id='2'>
  <name>Pegasus</name>
</project>
<project id='8'>
  <name>Typhon</name>
</project>
```

This output contains two rows, with one element in each row. Subqueries in SQL/XML are allowed to return only one row; therefore, to return more than one row of values in a SQL/XML subquery, they must be combined to form a single value. xmlagg() is an XML publishing function that produces a forest of elements by collecting the XML values that are returned from multiple rows and concatenating the values to make one value. Here is a query that uses the above subquery to create the XML output from the previous section:

```
select
  xmlelement(name customer,
    xmlattributes(c.CustId as id),
    xmlforest(c.Name as name, c.City as city),
    xmlelement(name projects,
    (select xmlagg(xmlelement(name project,
       xmlattributes(p.ProjId as id),
       xmlforest(p.Name as name)))
      from Projects p
     where p.CustId=c.CustId))) as "customer-projects"
from Customers c
```

The above query illustrates a very common pattern used to create XML hierarchies using SQL/XML.

3.6 The XML Datatype

The XML Datatype is a datatype in the same way that integer, date, or CLOB are datatypes in SQL. Since SQL/XML allows a query to create XML instances, there must be a datatype that corresponds to these instances. It is anticipated that the XML Datatype will be supported in JDBC 4.0. It is too early to say exactly how it will be used in that specification, but it is likely that it will retrieve XML values much like other values, and that XMLvalues can be retrieved as text, DOM, or SAX events. This is the approach currently taken by DataDirect Connect for SQL/XML. To illustrate this, let us use a SQL/XML query to create

a table with two columns, an integer containing the CustId and an XML column containing the XML output from the previous query. Here is the query:

```
select c.CustId,
  xmlelement(name customer,
    xmlattributes(c.CustId as id),
    xmlforest(c.Name as name, c.City as city),
    xmlelement(name projects,
    (select xmlagg(xmlelement(name project,
       xmlattributes(p.ProjId as id),
       xmlforest(p.Name as name)))
      from Projects p
     where p.CustId=c.CustId))) as "customer-projects"
from Customers c
```

Suppose the above query is in a string called sqlxmlString. Then the following Java code can be used to execute the query and retrieve values.

```
Statement statement=con.createStatement();

ResultSet rs=statement.executeQuery(sqlxmlString);

while(rs.next())
  {
   int id=rs.getInt(1);
   com.ddtek.jdbc.jxtr.XMLType xmlC=
     (com.ddtek.jdbc.jxtr.XMLType)rs.getObject(2);
   org.w3c.dom.Document doc=xmlC.getDOM();
   doSomethingUseful(id, doc);
  }
```

The XML Type also plays a second important role - relational databases now routinely store XML in individual column, and the XML Type provides a standard type for such columns, which is useful both in SQL and in JDBC.

6.1 SQL/XML Mapping Rules

The XML publishing functions use SQL values to create XML values, and these XML values have W3C XML Schema types. When we discussed the XML publishing functions, we did not address specifically how the XML representation is determined. The mapping rules of SQL/XML describe in excruciating detail how SQL values can be mapped to and from XML values, and how SQL metadata can be mapped to and from W3C XML Schemas.

To give a flavor for the level of detail in which this is specified, here are the equivalent headings from the SQL/XML specification's table of contents:

- Mapping SQL character sets to Unicode.
- Mapping SQL <identifier>s to XML Names.
- Mapping SQL data types (as used in SQL-schemas to define SQL-schema objects such as columns) to XML Schema data types.
- Mapping values of SQL data types to values of XML Schema data types.
- Mapping an SQL table to an XML document and an XML Schema document.
- Mapping an SQL schema to an XML document and an XML Schema document.
- Mapping an SQL catalog to an XML document and an XML Schema document.
- Mapping Unicode to SQL character sets.
- Mapping XML Names to SQL <identifier>s.

These mappings can be parameterized in several ways, including the target namespace for the result, whether to handle nulls using xsi:nil or absence, and whether to map a table to a single element or a forest of elements. Here is an XML representation of the Customers table shown earlier, using a single element for each table and no target namespace:

```
<myschema>

 <Customers>
  <row>
    <CustId>1</CustId>
    <Name>Woodworks</Name>
    <Address>Baltimore</Address>
  </row>
  <row>
    <CustId>2</CustId>
    <Name>Software Solutions</Name>
    <Address>Boston</Address>
  </row>
  <!-- !!! SNIP !!! -->
 </Customers>
<myschema>
```

Here is an XML representation of the same table using a forest of elements to represent each table:

```
<myschema>

 <Customers>
    <CustId>1</CustId>
    <Name>Woodworks</Name>
    <Address>Baltimore</Address>
 </Customers>
 <Customers>
    <CustId>2</CustId>
    <Name>Software Solutions</Name>
    <Address>Boston</Address>
 </Customers>
  ...
<myschema>
```

These mappings are also defined on the metadata level. For instance, SQL/XML defines how the datatypes of SQL are represented in the equivalent XML Schema. Each SQL type is derived from an equivalent built-in W3C XML Schema type. Where needed, facets are used to represent constraints added to those of the base type:

```
    <xsd:simpleType name="INTEGER">
         <xsd:restriction base="xsd:int" />
    </xsd:simpleType>

    <xsd:simpleType name="CHAR_50">
        <xsd:restriction base="xsd:string">
            <xsd:length value="50"/>
        </xsd:restriction>
    </xsd:simpleType>
```

```
 <Customers>
  <row>
    <CustId>1</CustId>
    <Name>Woodworks</Name>
    <Address xsi:nil="true" />
  </row>
  <!-- !!! SNIP !!! -->
```

As mentioned above, there are two ways to represent null values. Suppose the City column may have null values. Here is a row in the Customer's table that represents a null value using the first strategy, a nilled element:

```
 <Customers>
  <row>
    <CustId>1</CustId>
```

```
    <Name>Woodworks</Name>
  </row>
  <!-- !!! SNIP !!! -->
```

6.2 XQuery and Native XML Programming

The XQuery language was designed for querying or processing XML. Just as a traditional SQL query takes a set of tables as input and returns an XML table as its result, XQuery takes sequences of XML nodes as input and evaluates to a sequence of XML nodes. However, from the very beginning, XQuery was designed to allow XML views of non-XML data, as well as serialised forms of non-XML data. The reason for this is simple: XML is used to represent almost any conceivable kind of information, and it is easiest to integrate information if it is given a common view.

If everything looks like a nail, all you need is a hammer. Conventional Internet applications often store and query data using SQL, process data using Java or C#, and exchange data as XML. Using XQuery, it is possible to store, query, process, and exchange data as XML. This eliminates some of the mismatches that cause complications when working with XML in other environments.

6.3 Native XML Programming

XQuery is a language designed for integrating data from multiple sources, including XML sources like documents or web messages and databases. It does this by leveraging the ability of XML to model virtually any kind of data.

To query anything with XQuery, it must be presented as though it were XML, either by serializing it as XML or by creating an XML view of the data through some form of middleware. For relational data, most systems use the SQL/XML mappings for the XML view, since they are quite suitable and have been specified in detail.

XML is the basis of XQuery's type system and data model. The fundamental types of XQuery include the kinds of nodes found in XML documents: document nodes, elements, attributes, processing instructions, comments, and text nodes. XQuery also supports the built-in datatypes of W3C XML Schema for representing integers, strings, dates, and other datatypes - these built-in datatypes are predefined in XQuery, and are available with or without a schema.

Most modern programming languages provide some form of complex user-defined types, such as structures or objects. In XQuery, the only complex types are XML documents, elements, attributes, and W3C XML Schema complex types. There is no need to write a schema to create and manipulate complex XML structures in XQuery.

However, if a query needs to ensure consistent use of the types in a schema, a schema may be imported into a query. This has an effect analogous to importing structure or class definitions in an object-oriented language.

Programmes tend to revolve around data, and the complex datatypes used in a language have a profound effect on the way that a language is used. As a result, languages are sometimes identified by the way they represent complex data; for instance, there are object-oriented languages and relational query languages. In this sense, XQuery can be considered Native XML Programming Language. XSLT and XPath are also Native XML Programming Languages.

Most other languages used to process XML, including Java, C#, Perl, and Python are not. SQL/XML is fundamentally an extension to a relational query language, providing a bridge to XML.

The concept of a Native XML Programming Language is new, and many XML programmers are used to thinking of XML in terms of the constructs used in the languages with which they process XML. On XML-related mailing lists it is reasonably common to see beginners assert that XML is fundamentally relational or object-oriented, and even sophisticated XML programmers have been known to assert that XML is just text. In fact, the phrase “XML is Unicode with pointy brackets” has come to identify a vocal part of the XML community.

3.9.1 XML is not Objects!

An XML document can be represented using objects, and this is precisely the approach taken by DOM and JDOM. An XML parser can be used to create an appropriate object representation of an XML document without involving the programmer. However, the fundamental types of XML are not fundamental in object-oriented languages, so casting and conversion is frequently required. Similarly, the basic notions of hierarchy and containment are not directly supported in the object-oriented model, so explicit navigation is often required. This causes significant work for the programmer.

Adam Bosworth pointed this out with the following example [Bosworth]. Suppose a programmer wants to compute price/earnings ratios from an XML feed. An individual stock might be represented as follows:

```
<stock>
  <name>Cindy's Snowshoes</name>
  <ticker>NASDAQ:RAKD</ticker>
  <price>20.00</price>
  <revenues>2.00</revenues>
  <expenses>1.00</expenses>
</stock>
```

To compute the price/earnings ratio, we use the formula "pe = price / (revenues - expenses)". To do this with the DOM, we also need to parse the XML, navigate to the places where this information is found, and convert the text of the document to the appropriate datatype. Here is the DOM code Adam provides for this:

```
Tree t = ParseXML("stock.xml");
PERatio = number(t.getmember("/stock/price"))
     /((number(t.getmember("/stock/revenues") -
      number(t.getmember("/stock/expenses"))
```

This solution would have been much messier if Adam had not used the path expressions of XPath, a simple Native XML language. In XQuery, path expressions are part of the language, and numeric conversions are automatically done for untyped data. If the data is validated against a schema, the types assigned by the schema are used. This makes it possible to solve the same problem much more simply:

```
let $stock := document('stock.xml')/stock
return $stock/price div ($stock/revenue - $stock/expenses)
```

For XML-centric applications, an object-oriented representation of an XML document imposes unneeded overhead that complicates programmes.

3.9.2 XML is not just Text!

To many intelligent and articulate XML programmers, "XML is just Unicode with pointy brackets" is almost a statement of faith. Predictably, these people also complain that it is difficult to process XML without a parser. For instance, Joe Gregorio [Gregorio1] notes that in XML this document:

```
<item xmlns:dc="http://purl.org/dc/elements/1.1/">
  <title>MetaData</title>
  <dc:date>2003-01-12T00:18:05-05:00</bc:date>

  <link>http://bitworking.org/news/8</link>

  <description>Upon waking, the dinosaur...</description>
</item>
```

must be treated as identical to this document:

```
<root:item xmlns:bc="http://purl.org/dc/elements/1.1/" xmlns:root="" >
  <root:title>MetaData</root:title>

  <bc:date>2003-01-12T00:18:05-05:00</bc:date>

  <root:link>http://bitworking.org/news/8</root:link>
  <description>Upon waking, the dinosaur...</description>
</root:item>
```

To many of us, this is merely an indication that XML must first be parsed and converted to an appropriate data model before it can easily be processed. In fairness to Joe, he initially assumed this as well, but then changed his mind:

- More XML experience is gained by yours truly and on many occasions I have found myself pining for the ability to do regular expression processing of XML. If only the pathologies of the above examples did not exist then I could use a combination of XPath and regular expressions to perform XML manipulations that would be easier for me to implement, understand and maintain.
- Today I reached the breaking point. The problem is not with regular expressions, the problem is with XML. The pathologies in XML that preclude the use of regular expressions are just that, pathologies, and ones that need to be excised.

As a result, he suggests that XML be subset as follows:

- all namespace declarations must be done in the root element.
- never a declaration for the "" namespace i.e. if an element sits the "" namespace then the element name will never have a namespace qualifier.
- no CDATA sections.
- no DTDs.

The above restrictions would make it easier for a programmer to work with XML without using an XML parser, but it is unlikely that the XML community will replace XML with something along these lines - especially since there are important usage scenarios for features like DTDs, schemas, and the ability to build compound documents without knowing, at the root level, all of the namespaces that may be used in a document. More to the point, Joe's original reason for trying to solve these problems with XPath and regular expressions was that the standard APIs do not make it easy to solve many simple problems.

Looking at his article as a whole and other articles he has written, we believe that many of these difficulties are caused by the same kind of semantic mismatches that a Native XML Programming Language is designed to solve.

In this paper, we assume that XML will remain as is, and that for general processing, the best approach is to use an XML parser to build a data model instance from the XML documents, and query the data model instance. Not everybody believes this is the best approach. Tim Bray, one of the editors of the original XML specification, objects to the Native XML Programming solution because he objects to the notion of an XML data model: [Bray]

The notion that there is an "XML data model" is silly and unsupported by real-world evidence. The definition of XML is syntactic: the "Infoset" is an afterthought and in any case is far indeed from being a data model specification that a programmer could work with. Empirical evidence:

- I can point to a handful of different popular XML-in-Java APIs each of which has its own data model and each of which works. So why would you think that there is a data model there to build a language around?
- Tim first says that there is no data model for XML, then argues that there are several. The differences among these data models, while annoying, are not great, and could have been avoided if XML had had a full-fledged data model. The differences between the DOM data model and the XPath data model are well known in the XML world.

XQuery, XPath, and XSLT now use one common data model, which can represent both XML and the XML Schema datatypes. Although it would have been convenient if XML had defined a data model, there is no requirement that the data model used by a Native XML Programming Language be the same as any particular data model used in a Java API. As long as the data model supports the structure of XML directly,

without losing or adding information in violation of the XML spec, it can be used as the basis for processing.

Tim also suggests that XML is "syntactic", as though this implies that there is no data model. This implies that syntax and structure are opposites, which is rather surprising, since the purpose of syntax is to describe the structure of a language. In the XML Recommendation, the structure that corresponds to a data model is called the logical structure:

Each XML document has both a logical and a physical structure. [. . .] Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup.

Like most modern computer languages, XML uses a BNF to describe the syntactic representation of these structures. For instance, here is a production from the XML Recommendation:

```
[39]  element  :=  EmptyElemTag| STag content ETag
```

The XML Recommendation is largely a description of these logical structures and the relationships among them. For instance, consider the following text:

- Example: The **element** structure of an **XML document** may, for validation purposes, be constrained using element type and attribute-list declarations. An element type declaration constrains the element's **content**.

- Element, XML document, and content all refer to logical structures that are represented in the BNF. These logical structures, taken together with the relationships among them as described in the XML Recommendation, come very close to being a data model, but the data model was not fully described.
  The whole point of parsing is to create structures from a sequence of characters, using a grammar to determine which structures to create. When a parser is used to interpret the characters of a programme in Java, it creates an Abstract Syntax Tree. When it is used to interpret the characters of XML, it creates a data model instance. We use parsers because:

- The parsed structure is more convenient for further processing,
- The parsed structure distinguishes information from noise, eliminating differences in the character representation that are not significant in the relevant model, and
- The parsed structure can fill in information not explicitly represented in the serialised form.

However, an XML parser is not enough. A parser creates a convenient representation of XML. We need a Native XML Programming Language to provide convenient processing of this XML.

3.9.3 What Should a Native XML Programming Language Do?

A Native XML Programming Language must provide the fundamental operations needed for XML. Some of these operations are required because of the structure of XML itself.

A Native XML Programming Language should be able to easily find anything in an XML structure. XQuery, like XSLT, uses XPath for this purpose. Every XPath expression is also an XQuery expression. For instance, if the variable $cust is bound to a Customers element that contains the rows of a relational table, represented using the SQL/XML mappings, then the following path expression finds all the CustIds from that table:

```
$cust/row/CustId
```

A Native XML Programming Language should be able to easily create any XML structure. XQuery uses the syntax of XML for this purpose. For instance, the following XQuery expression creates a Customer element:

```
<Customer>
  <CustId>17</CustId>
  <Name>Ferd Berfle</Name>
</Customer>
```

When XQuery uses the syntax of XML, a curly brace escapes to the syntax of XQuery, allowing dynamic expressions to be inserted. Here is an example that creates a customer with a new unique identifier:

```
<Customer>
  <CustId>{ max( $cust/row/CustId) + 1 }</CustId>
  <Name>Ferd Berfle</Name>
</Customer>
```

A Native XML Programming Language should be able to easily combine and restructure information from XML sources, operating at the logical level without requiring the programmer to think about the internal representation of the XML. For instance, if we are operating on the SQL/XML views of the customers' database, the following XQuery

combines customers and projects to show the name of a customer and all projects associated with that customer:

```
for $c in $cust/row
let $p := $proj/row[CustId = $c/CustId]
return
  <customer>
     <custName>{ string($c/name) }</custName>
     <projName{ string($p/name) }</projName>
  </customer>
```

A Native XML Programming Language should be able to easily use XML data in expressions. For instance, arithmetic operations should be able to work directly with XML content, observing the data types of typed data and converting appropriately when they encounter untyped data. It should be able to leverage schemas that have been imported into a query, but work well on XML structures for which no schema has been imported.

In short, a Native XML Programming Language should be able to work with XML the way XML users think of it, easily performing the kinds of tasks that XML users need to have done. XQuery attempts to do just that, based on the usage scenarios we gathered in the ### XML Query Use Cases.

6.3 XQuery and SQL/XML Views

Some people seem to believe that the purpose of XQuery is largely the same as that of SQL/XML - to allow XML structures to be created from relational data. Although XQuery is useful for this task, it has relatively few advantages over SQL/XML when this is all that is required. The reason for this is simple: SQL is a language designed for handling SQL data sources, and it does that very well. Adding XML publishing functions to SQL is a simple way to let it create XML. However, it is interesting to note that the SQL/XML views of relational tables have a very constrained structure, and XQuery performed on such views is generally quite similar to the equivalent SQL/XML.

For instance, let us write an XQuery equivalent to the last SQL/XML query we used. This query will operate on a SQL/XML view of the relational tables. The Projects table is represented as follows:

```
<Projects>
  <row>
    <ProjId>2</ProjId>
    <Name>Pegasus</Name>
    <CustId>4</CustId>
  </row>
  <row>
    <ProjId>8</ProjId>
    <Name>Typhon</Name>
    <CustId>4</CustId>
  </row>
  <!-- !!! SNIP !!! -->
```

The Customers table is represented as follows:

```
<Customers>
  <row>
    <CustId>4</CustId>
    <Name>Hardware Heaven</Name>
    <Address>Washington</Address>
  </row>
  <!-- !!! SNIP !!! -->
```

We want to rename these elements and create a representation that shows customers together with their projects. The output should look like this:

```
<customer id = "4">
    <name>Hardware Heaven</name>
    <projects>
     <project id = "2" name = "Pegasus"/>
     <project id = "8" name = "Typhon"/>
    </projects>
  </customer>
```

Here is an XQuery that creates the desired output:

```
for $c in $cust/row
return
  <customer id="{$c/CustId}">
    <name>{ string($c/Name) }</name>
    <projects>
     {
      for $p in $proj/row
      where $p/CustId = $c/CustId
      return
        <project id="{$p/ProjId}" name="{$p/Name}"/>
     }
    </projects>
  </customer>
```

Let us compare this XQuery to the SQL/XML query from a prior section:

SQL/XML	XQuery

select
xmlelement(name customer,
xmlattributes(c.CustId as id),
xmlforest(c.Name as name,
c.City as city),
xmlelement(name projects,
(select xmlagg(xmlelement
(name project,
xmlattributes(p.ProjId as id),
xmlforest(p.Name as name)))
from Projects p
where p.CustId=c.CustId)))
as "customer-projects"
from Customers c
	for $c in $cust/row
	return
	<customer id="{$c/CustId}">
	<name>{ string($c/Name)}</name>
	<projects>
	{
	for $p in $proj/row
	where $p/CustId = $c/CustId
	return
	<project id="{$p/ProjId}"
	name="{$p/Name}"/>
	}
	</projects>
	</customer>

Table 1.

In this example, as in most such examples, it is hard to argue that either solution is particularly superior to the other. Either SQL/XML or XQuery handle such tasks quite well. The real strength of XQuery is in the ability to easily process XML, whether or not relational data is being processed, including the XML that is frequently stored in columns of relational databases and the XML of web messages. Since XQuery also works well on SQL/XML views of relational data, it is particularly useful when both XML data and relational data must be used in processing.

This is explored in the next section.

3.11 Spanning Sources: XQuery, Web Messages and Database

XQuery, when combined with a SQL/XML view of a relational database, is extremely good for processing XML together with relational data. This is a very common requirement in many environments, including web message processing environments. To illustrate this, we will use example 1 from the SOAP Primer.

The task is as follows: an incoming message requests a flight to Los Angeles departing from New York as follows:

```
<!-- Example 1 from SOAP Primer -->
<env:Body>
   <p:itinerary xmlns:p="http://travel.org/reservation/travel">

   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2001-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
</p:itinerary>
```

According to the SOAP Primer, the proper response is to point out that there are three airports that depart from New York, so that the user can be prompted to pick one. Here is the desired output:

```
<env:Body>
  <p:itinerary xmlns:p="http://travel.org/reservation/travel">
   <p:airportChoices>JFK LGA EWR</p:airportChoices>
  </p:itinerary>
 </env:Body>
```

Reading between the lines, we assume that there is a database somewhere that lists the airports for each city. The SQL/XML view of the airports table might look like this:

```
<AIRPORTS>
  <row>
    <CITY>Raleigh / Durham</CITY>
    <AIRPORT>RDU</AIRPORT>
   </row>
   <row>
     <CITY>New York</CITY>
     <AIRPORT>JFK</AIRPORT>
   </row>
   <row>
     <CITY>New York</CITY>
     <AIRPORT>LGA</AIRPORT>
   </row>
```

We will assume that when there is only one airport for a city, the output should simply list that city, and that an error should be raised if there is no airport for a given city. The following XQuery handles all three of these cases:

```
for $city in doc("incoming.xml")//p:departing
let $airports := sql:table("airports")/AIRPORTS/row[CITY = $city]
return
    if (count($airports) = 0)
      then <error> No airports found for {$city}!</error>
    else if (count($airports) = 1)
      then <airport>{ string($airports/AIRPORT) }</airport>
    else if (count($airports) > 1)
      then
       <airportChoices>
        {
          for $c in $airports/AIRPORT
          return (string-value( $c ), " ")
        }
       </airportChoices>
    else ()
```

Note that this code operates at a level very close to the application domain, rather than navigating XML documents and converting from XML to appropriate types in the host language. XML data sources and relational data sources are treated in the same way - to the query, they both look like XML documents.

3.12 XQuery for Java (JSR 225)

SQL programmers are used to using APIs such as ODBC or JDBC to set up the environment, execute queries, and do processing in the business domain using the data returned by a query. Similar APIs are expected to emerge for XQuery. The first standard API for this purpose is now being developed under Java Community Process. It is known as XQuery for Java (XQJ), or JSR 225.

Significantly, the requirements of JSR 225 ensure that both XML documents and XML views of databases will be supported, and the results of a query can be processed using JAXP and StAX.

3.13 SQL/XML and XQuery: Do we Need Both?

Although SQL/XML and XQuery are both XML query standards, they are based on quite different models, and fit best in different architectures. SQL/XML fits cleanly into the relational model as a reasonably small extension to traditional SQL. This means that it works well in traditional SQL environments, providing full access to the existing SQL language, including features like updates and full-text queries that are not going to be part of XQuery

One of the other advantages of using SQL as a basis is that database manufacturers have many years of experience in optimizing SQL queries, which means that many of the optimization issues are well

known. Also, it has existing APIs, including ODBC and JDBC. In short, SQL/XML provides the functionality needed for creating XML from relational data while still fitting cleanly into the existing SQL environment. SQL/XML implementations will be available from Oracle and IBM, but not Microsoft, and a cross-database implementation is available from DataDirect Technologies. Oracle's implementation also provides functionality for querying and processing XML as well as SQL, and there is some interest in adding extensions along these lines to SQL/XML. Some members of the SQL/XML task force would also like to see parts of XQuery added to SQL/XML. XQuery fits more cleanly into the XML environment, providing Native XML Programming for both XML sources and non-XML sources accessed via an XML view. It is well designed for combining data from multiple sources, and is very efficient for a variety of XML programming tasks. However, XQuery is a brand new language – in fact, at the time of writing, XQuery 1.0 is merely a Working Draft, not likely to emerge until the second half of 2004. There is a great deal of enthusiasm surrounding XQuery, most major database vendors have announced support for it, and there is a great deal of research on optimizing XQuery. However, XQuery is a much younger language, the industry has little experience optimizing it, and it lacks some features, including updates and fulltext, which are very important for some kinds of tasks. Also, the API for XQuery, XQuery for Java (JSR 225) is just now being developed.

4.0 CONCLUSION

In this unit we are confident that you have learned both SQL/XML and XQuery which will play an important role in XML queries, and that XQuery will become very important for general purpose XML processing.

5.0 SUMMARY

Both languages will continue to evolve, trying to fill in the functionality found in the other. On the whole, we feel that SQL/XML is best for SQL programmers who think of their task in terms of SQL, but need to create results in XML. SQL/XML is used much like a report writing language, except that the reports are XML documents.

XQuery is best for XML programmers who are working only with XML, or need to work with XML and relational data together. In the short term, implementers and users of XQuery should be aware that it is both new and revolutionary - it shows great promise, but we have less industry experience with XQuery than with SQL/XML.

We are confident that both SQL/XML and XQuery will play an important role in XML queries, and that XQuery will become very important for general purpose XML processing. Native XML Programming is a revolution waiting to happen, and XQuery will be key to this revolution.

6.0 TUTOR-MARKED ASSIGNMENT

i. With your acquired knowledge, design an XML Query for the computation of student personal information for your department
ii. Design an XML programme for an Electronic Voting System.

7.0 REFERENCES/FURTHER READING

[XQueryExperts] XQuery from the Experts: A Guide to the W3C XML Query Language, by Howard Katz, Don Chamberlin, Denise Draper, Mary Fernandez, Michael Kay, Jonathan Robie, Michael Rys, Jerome Simeon, Jim Tivy, Philip Wadler. Addison-Wesley Pub Co; (1st ed.). (September 12, 2003) ISBN: 0321180607.

Jeffrey, Ullman .Relational Algebra.

Ramakrishnan and Gehrke, J. *Database Management Systems* (3rd ed.).

Isabelle, Bichindaritz. *Database Systems Design*.

Paul, Werstein. *Relational Algebra*.

UNIT 4 DATABASE RECOVERY

CONTENTS

1.0 INTRODUCTION

Database recovery involves the process of making a copy of database in case of an equipment failure or disaster, then recovering or retrieving the copied database if needed.

A database provides multiple autonomous centralised and homogenous views of data. The data in a database are structured according to a schema specified in a data definition language (DDL), and are manipulated using operations specified in a data manipulation language (DML).

2.0 OBJECTIVES

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

- describe the basic concept of database recovery and heterogeneity and web services definition for database
- describe the classification and techniques of database.
- describe the classification criteria for heterogeneous database
- describe the database recovery log and technique
- describe database sharing in a heterogeneous database system.

3.0 MAIN CONTENT

3.1 Data Integrity and Reliability

The integrity of a database comprises the accuracy, correctness, validity and consistency of data. Although, database system can provide little protection against data error which are in the real world before the data are even loaded into the system, some protection can be built into database to ensure that error within the system are minimised.

3.2 Database Recovery

Computer systems can fail, hardware can break down, programmes have bugs. Human procedures contain errors and people make mistakes. All these failure occur in database application. It is then important to recover database without any damage as soon as possible. This can be achieved by going back to a known point and reprocess the workload from there. The simplest form of this type is to make a copy periodically of the database and keep a record of all transaction that have been processed. Database recovery can be done in two ways.

Reprocessing: **since processing cannot be resumed at a precise point, the next best alternative is to go to recovery via rollback/roll forward.**

This is to make a copy of the database (database save) periodically and to keep a log of the changes made by the transaction against the database since the save.

Rollforward: The database is restored using the sort data, and all valid transactions since the save are reapplied.

Rollback: we undo changes made by erroneous or partially processed transaction by undoing the changes they have made in the database. Then the valid transactions that were processed at that time of the failure are restarted. Both of these required that a log of transaction be kept.

3.3 Database Recovery Log

A database recovery log keeps a record of all changes made to a database, including the addition of new tables or updates to existing ones. This log is made up of a number of log extents, each contained in a separate file called a log file. The database recovery log can be used to ensure that a failure (for example, a system power outage or application error) does not leave the database in an inconsistent state. In case of a failure, the changes already made but not committed are rolled back and

all committed transactions, which may not have been physically written to disk, are redone. These actions ensure the integrity of the database.

3.3.1 Definition of Data Recovery

Restoring data from disks, tapes, CDs and digital photo memory cards that have been damaged by accidents, disasters, power surges and malfunctioning electronics. Laptop hard disks are especially vulnerable if users are constantly on the move.

The best data recovery technique is to have data already backed up on another storage device either on the same computer, a network server or the internet. Data recovery becomes a simple copy procedure after the failed peripheral is replaced. At worst, applications may have to be re-installed if only user data was backed up, but unless the applications are vintage programmes that are no longer available, the data are far more valuable than the software.

3.3.2 Recovery from Damaged Media

If there is no backup and data must be recovered. There are some organisations that specialise in retrieving data from damaged computers. They may be able move the drive to a working computer, or they may have to open the drive and replace parts such as read/write heads, actuator arms and chips. Sometimes, the platters are removed and placed into another drive.

3.4 Classification Criteria for Heterogeneous Database

Data definition and manipulation languages are based on a data model that defines the semantics of the constructs and operations provided by these languages. Managing data in multiple pre-existing databases entails dealing with their data distribution, system (e.g. DBMS) heterogeneity, and semantic (e.g. schema) heterogeneity. Approaches to managing heterogeneous databases including linking heterogeneous databases via the World Wide Web (www), organising them into database federations or multidatabase systems, and constructing data warehouses. Common to these approaches is allowing component databases to preserve their autonomy, that is, their local definitions, applications, and policy of exchanging data with other databases (Bright *et al*. 1992).

Heterogeneous database systems have been traditionally classified by the type of schemas, extent of data sharing, and data access facilities they support. Schema supported by a heterogeneous database system includes (Sheth and Larson, 1990):

- local view expressed representing the schemas of component databases expressed in DDL of local databases and,
- global schema expressed in a common DDL, providing a unified view of all component databases.

Thus, every database in a heterogeneous database system can provide a subset of its schema as its export schema interface to other databases; each database in turn can have import schemas describing the export schemas of other databases in their local DDL (Heimbigner and McLeod, 1985). The global schema of a heterogeneous database system can range from a loose export schema to a fully integrated schema. Similarly, local views of the system can range from a loose collection of import schemas to an integration of the local schema with all import schemas. For example, a database federation can have a global (federation) schema that provides users with a uniform view of the federation and thus insulate them from the component databases, or local views that provide users with multiple views of the federation.

A data warehouse represents the materialisation of a global schema. That is, the warehouse database defined by the global schema is loaded with data from the component databases. Unlike database federations and data warehouses, multidatabase systems are collections of loosely coupled databases without global schemas.

3.4.1 Database Sharing in a Heterogeneous Database System

Database sharing in a heterogeneous database system can be at the level of:

- linking specific data items in the component databases; or
- generic (Schema-driven) correlations across component databases.

Individual data item links (e.g. hypertext links) between databases do not require or comply with schema correlations across databases. For schema correlations, data links need to be consistent with the constraints entailed by these correlations, such as inter-database referential integrity constraints.

SELF-ASSESSMENT EXERCISE

i. Explain the basic concepts of database recovery and heterogeneity
ii. Briefly describe the various classifications of database.

4.0 CONCLUSION

In this unit, you have been introduced to the fundamental concepts of database, its classification, database recovery and database heterogeneity.

5.0 SUMMARY

What you have learnt in this unit concerns:

- introduction to database recovery and heterogeneity and web services definition for database
- classification and techniques of database.
- classification criteria for heterogeneous database.
- database recovery log and technique.
- database sharing in a heterogeneous database system.

6.0 TUTOR-MARKED ASSIGNMENT

What is database recovery log and briefly describe the classification criteria for heterogeneous database.

7.0 REFERENCES/FURTHER READING

Bright; *et al*. (1992). *Policy of Exchanging Data with Other Databases*.

Heimbigner and McLeod (1985).

Sheth and Larson (1990). *Heterogeneous Database System*.