

Database Management Systems

V. Vidhya • G. Jeyaram • K.R. Ishwarya



Alpha Science

Database Management Systems

Database Management Systems

V. Vidhya
G. Jeyaram
K. R. Ishwarya



Alpha Science International Ltd.
Oxford, U.K.

Database Management System

424 pages. | 226 Figs.

**V. Vidhya
G. Jeyaram
K. R. Ishwarya**

Department of Computer Science & Engineering
Annai Vailankanni College of Engineering
Kanyakumari

Copyright © 2016

ALPHA SCIENCE INTERNATIONAL LTD.

7200 The Quorum, Oxford Business Park North
Garsington Road, Oxford OX4 2JZ, U.K.

www.alphasci.com

ISBN 978-1-78332-213-8

E-ISBN 978-1-78332-318-0

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of the publisher.

Dedicated to my mother

Preface

Database management systems have become fundamental tools for managing information. Database Management systems is a collection of databases and DBMS utilities to help develop application satisfying user requirements under the supervision of database administrators. In the information era the growth of information has become very fast so managing of information is essential and integral part of computer science curricula. This book covers the fundamentals of modern database management systems. This book is for anyone who is interested to gain some familiarity in database management systems. The main intention is to present the material in a clear and simple style with worked out examples. It offers a balanced view of concepts, languages and architectures, with concrete reference to current technology.

Chapter I provides a general introduction to database management systems. The chapter compares the traditional file processing system with database system. The overall system architecture is explained in detail.

Chapter II explains in detail the various data models available. The merits and demerits of all database model are dealt in detail. Various integrity constraints: primary key and referential constraints; null values are also briefly discussed. The ER diagram included in this chapter helps the readers to draw ER diagrams for any system. Extended ER concepts like generalization, specialization and aggregation are explained with suitable examples.

Chapter III deals with relational algebra. The basic operation, additional and extended relational algebra operations are explained in detail. Relational calculus deals with tuple relational calculus and domain relational calculus.

Chapter IV and Chapter V is devoted for SQL. The units are organized in easily understandable form. The chapters include examples for all operations with screen shots. Different databases like client server database and distributed databases are explained in detailed. Types of SQL like embedded and dynamic SQL are explained in detail.

Chapter VI deals with relational database design. Functional dependency is explained in detail. The normal forms and different types of normal forms are explained with suitable examples.

Chapter VII provides a detailed description about the transaction concepts. Transaction states and ACID properties of transaction are explained in detail. The concept of serializability and its types are explained in detail.

Chapter VIII deals with concurrency control. The unit describes the problem with concurrent transactions and the method to solve the problem caused. Locking protocols, types of locking protocols and deadlocks are dealt in detail.

Chapter IX deals with data recovery techniques. The chapter explains the techniques available to recover the transactions from various types of failures.

Chapter X deals with record storage. It explains the different types of storage available. The unit explains the way to store and retrieve the data. The concept of RAID is explained in detail.

Chapter XI is physical database design. File organization – variable and fixed size, organization of records in files, are explained in detail.

Chapter XII & Chapter XIII deals with indexing, hashing and query processing. How to optimize query and how to process the query is explained in detail.

Chapter XIV is devoted to PL/SQL. Any reader can understand the programming concept of PL/SQL. The unit has been organized from basic to advanced concepts with simple examples to help readers to understand the concept easily.

Chapter XV is Data mining and Data Warehousing. The chapter explains the architecture of Data mining and Data Warehousing, KDD process, Data mining functionalities, classification of data mining are explained. Data preprocessing is dealt in detail.

**V. Vidhya
G. Jeyaram
K. R. Ishwarya**

Contents

1. Introduction	1.1—1.14
1.1 Introduction to DBMS	1.1
1.1.1 Structure of DBMS	1.1
1.1.2 Applications of Database	1.2
1.1.3 Classification of Database Management System	1.2
1.2 File Processing System	1.3
1.2.1 Drawbacks of Conventional File Processing System	1.3
1.3 Advantages of Database	1.5
1.4 Disadvantages of Database	1.7
1.5 Views of Data	1.8
1.5.1 Data Abstraction	1.8
1.5.2 Data Independence	1.9
1.5.3 Instances and Schemas	1.9
1.6 Database System Structure	1.9
1.6.1 Database Users and Administrators	1.11
1.6.1.1 Database Users	1.11
1.6.1.2 Database Administrators	1.12
1.6.2 Application Architectures	1.13
1.6.2.1 Two-tier Architecture	1.13
1.6.2.2 Three-tier Architecture	1.13
Review Questions	1.13
2. Data Models	2.1—2.26
2.1 Data Models	2.1
2.2 Hierarchical Model	2.1
2.3 Network Model	2.2
2.4 Object-Oriented Model	2.3
2.5 Object Relational Model	2.3
2.6 Relational Model	2.4
2.6.1 Characteristics of Relational Model	2.4

2.6.2	E. F. Codd's Laws for a Fully Functional Relational Database Management System	2.4
2.6.3	Principle Components of Relational Model	2.5
2.7	Comparison Between the Various Database Models	2.7
2.8	Entity Relationship Model	2.8
2.8.1	Basic Concepts	2.8
2.8.1.1	Entity	2.8
2.8.1.2	Attributes	2.8
2.8.1.3	Relationships and Relationships Sets	2.9
2.8.2	Constraints	2.10
2.8.2.1	Mapping Cardinalities	2.10
2.8.2.2	Participation Constraint	2.12
2.8.3	Keys	2.12
2.8.4	Entity-Relationship Diagram	2.12
2.8.5	Dependency	2.14
2.8.6	Sample E-R Diagram	2.15
2.8.7	E-R Diagram	2.16
2.8.8	Extended E-R Features	2.19
2.8.8.1	Specialization	2.20
2.8.8.2	Generalization	2.20
2.8.8.3	Aggregation	2.21
2.8.9	Reduction of an E-R Schema to Tables	2.22
2.8.9.1	Tabular Representation of Strong Entity Sets	2.22
2.8.9.2	Tabular Representation of Relationship Sets	2.23
2.8.9.3	Tabular Representation of Weak Entity Sets	2.23
2.8.9.4	Redundancy of Tables	2.24
2.8.9.5	Tabular Representation of Generalization	2.24
2.8.9.6	Tabular Representation of Aggregation	2.25
	Review Questions	2.25
3.	The Relational Algebra	3.1—3.21
3.1	Basic operations	3.1
3.2	Additional Operations	3.8
3.3	Extended Relational-Algebra Operations	3.13
3.4	Modification of the Database	3.16
3.5	Relational Calculus	3.17
3.5.1	Tuple Relational Calculus	3.17
3.5.2	Domain Relational Calculus	3.19
	Review Questions	3.20
4.	SQL-Fundamentals	4.1—4.57
4.1	Introduction	4.1
4.2	Advantages of SQL	4.1

4.3	Parts of SQL	4.2
4.4	Domain Types in SQL	4.2
4.5	Terminology	4.3
4.6	Data Definition Language (DDL)	4.3
4.6.1	DDL Commands	4.4
4.7	Data Manipulation Language (DML)	4.7
4.8	Basic Structure of SQL Expression	4.13
4.8.1	General Form of SQL Query	4.13
4.8.2	Selecting all Columns from a Relation	4.13
4.8.3	Selecting Specific Columns	4.14
4.8.4	Usage of DISTINCT Keyword in SELECT Statement	4.14
4.8.5	Usage of ALL Keyword in SELECT Statement	4.15
4.9	Column Alias Name	4.16
4.10	String Operation	4.17
4.11	Concatenation Operation	4.18
4.12	Ordering the Display of Tuples	4.19
4.13	Set Operations	4.20
4.13.1	Union	4.21
4.13.2	Union All	4.22
4.13.3	Intersect	4.22
4.13.4	Minus or Except	4.23
4.14	Where Clause	4.24
4.14.1	Operators in the WHERE Clause	4.25
4.15	Operators	4.27
4.15.1	Arithmetic Operators	4.27
4.15.2	SQL Comparison Operators	4.28
4.15.3	SQL Logical Operators	4.28
4.16	Aggregate Function	4.29
4.16.1	AVG	4.29
4.16.2	SUM	4.29
4.16.3	MAX	4.30
4.16.4	MIN	4.30
4.16.5	COUNT	4.31
4.16.6	COUNT*	4.31
4.17	Group by Clause	4.32
4.18	Having Clause	4.33
4.19	Nested Subqueries	4.33
4.19.1	Subqueries with the SELECT Statement	4.34
4.19.2	Subqueries with the INSERT Statement	4.34
4.19.3	Subqueries with the UPDATE Statement	4.35
4.19.4	Subqueries with the DELETE Statement	4.36

4.20	Null values	4.37
4.21	Database Objects	4.37
4.21.1	Tables	4.37
4.21.2	Views	4.38
4.21.2.1	Advantages of View	4.39
4.21.2.2	Creating Views	4.39
4.21.2.3	The WITH CHECK OPTION	4.40
4.21.2.4	Updating a View	4.40
4.21.2.5	Inserting Rows into a View	4.41
4.21.2.6	Deleting Rows into a View	4.41
4.21.2.7	Dropping Views	4.42
4.21.3	Sequences	4.42
4.21.3.1	Creating a Sequence	4.42
4.21.3.2	Dropping a Sequence	4.43
4.21.3.3	Using a Sequence	4.43
4.21.4	Triggers	4.44
4.21.4.1	Benefits of Triggers	4.44
4.21.4.2	Creating Triggers	4.44
4.21.4.3	Types of Triggers	4.45
4.21.4.4	Trigger Execution Hierarchy	4.45
4.21.4.5	Triggering a Trigger	4.46
4.21.4.6	Disabling Triggers	4.47
4.21.4.7	Enabling Trigger	4.47
4.21.4.8	Dropping triggers	4.47
4.21.5	Indexes	4.47
4.21.5.1	The CREATE INDEX Command	4.48
4.21.5.2	The DROP INDEX Command	4.48
4.21.5.3	When should indexes be avoided?	4.49
4.22	DCL (Data Control Language)	4.49
4.22.1	GRANT Command	4.49
4.22.2	REVOKE Command	4.50
4.22.3	Privileges and Roles	4.50
4.22.4	Creating Roles	4.51
4.23	TCL (Transaction Control Language)	4.52
4.23.1	The COMMIT Command	4.52
4.23.2	The ROLLBACK Command	4.53
4.23.3	The SAVEPOINT Command	4.54
4.23.4	The RELEASE SAVEPOINT Command	4.55
4.23.5	The SET TRANSACTION Command	4.55
	Review Questions	4.55

5. Joins, Constraints and Advanced SQL	5.1–5.33
5.1 Joins	5.1
5.1.1 Joins Types	5.2
5.1.1.1 Inner Join	5.3
5.1.1.2 Left Join	5.3
5.1.1.3 Right Join	5.4
5.1.1.4 Full Join	5.5
5.1.1.5 Self Join	5.6
5.1.1.6 Cartesian Join	5.6
5.2 Constraints	5.7
5.2.1 Types of Constraints	5.8
5.2.1.1 DOMAIN Integrity Constraints	5.8
5.2.1.2 Entity Integrity Constraints	5.10
5.2.1.3 Referential Integrity Constraints	5.12
5.2.1.4 Index Constraint	5.13
5.2.1.5 Default Constraint	5.14
5.2.1.6 Assertion	5.15
5.3 Security	5.15
5.3.1 Authorization	5.16
5.3.2 Roles	5.17
5.3.3 Authorization and Views	5.17
5.3.4 Granting of Privileges	5.18
5.3.4.1 Authorization Grant Graph	5.18
5.3.5 Audit Trails	5.19
5.3.6 Limitations of SQL Authorization	5.19
5.3.7 Encryption	5.19
5.3.8 Authentication (Challenge Response System)	5.20
5.3.9 Digital Certificates	5.20
5.4 Embedded SQL	5.21
5.4.1 Host Variables	5.21
5.4.2 Indicator Variables	5.23
5.4.3 SQL Communications Area (SQLCA)	5.23
5.4.4 Cursors	5.24
5.4.5 Transaction Control	5.24
5.5 Dynamic SQL	5.25
5.6 Distributed Databases	5.26
5.6.1 Features of Distributed Databases	5.27
5.6.2 Types of Distributed Databases	5.27
5.6.3 Components of a Distributed Databases	5.28
5.6.4 Distributed Data Storage	5.28

5.6.4.1	Data Replication	5.29
5.6.4.2	Data Fragmentation	5.29
5.6.4.3	Transparency	5.29
5.6.5	Advantages of Distributed System	5.30
5.6.6	Disadvantages of Distributed System	5.30
5.7	Client/Server Databases	5.30
5.7.1	Two-tier Architecture	5.31
5.7.2	Three-tier Architecture	5.31
5.7.3	Merits of Client/Server Computing	5.32
	Review Questions	5.33

6. Relational Database Design

6.1–6.24

6.1	Introduction	6.1
6.2	Anomalies in Databases	6.1
6.3	Redundant Information	6.2
6.4	Functional Dependency	6.2
6.4.1	Notation of Functional Dependency	6.2
6.4.2	Compound Determinants	6.3
6.4.3	Types of Functional Dependency	6.3
6.4.4	Uses of Functional Dependency	6.4
6.5	Closure of a Set of Functional Dependencies	6.4
6.5.1	Armstrong's Axioms	6.4
6.5.2	Additional Rules	6.4
6.5.3	Closure of Attribute Sets	6.5
6.6	Canonical Cover	6.6
6.6.1	Extraneous Attributes	6.7
6.6.2	Computing a Canonical Cover	6.8
6.7	Normalization	6.8
6.7.1	Purpose of Normalization	6.9
6.7.2	Normalization Forms	6.9
6.7.3	First Normal Form (1NF)	6.9
6.7.4	Second Normal Form (2NF)	6.11
6.7.5	Third Normal Form (3NF)	6.14
6.7.5.1	Summary of Normal Forms	6.15
6.8	Loss Less Decomposition	6.15
6.9	Dependency Preservation	6.16
6.10	Boyce Codd Normal Form (BCNF)	6.16
6.10.1	Testing Decomposition of BCNF	6.19
6.11	Multi Valued Dependencies & Fourth Normal Form (4NF)	6.19
6.11.1	Multi Valued Dependencies (MVD)	6.19
6.11.2	Fourth Normal Form	6.20

6.12	Join Dependencies & Fifth Normal Form (5NF)	6.22
6.12.1	Join Dependencies	6.22
6.12.2	Fifth Normal Form (5NF)	6.22
	Review Questions	6.24
7.	Transaction Processing	7.1—7.12
7.1	Introduction	7.1
7.1.1	Transaction Concepts	7.1
7.2	Reasons for Transaction Failures	7.4
7.3	Transaction and System Concepts	7.4
7.3.1	Transaction States and Additional Operations	7.5
7.3.2	State Transition of a State Transition Diagram	7.5
7.4	The System Log	7.6
7.5	Acid Properties or Transaction Properties	7.7
7.6	Serializability	7.8
7.6.1	Schedules and Serializability	7.9
7.6.2	Conflict Serializability	7.9
7.6.3	View Serializability	7.11
7.6.4	Testing for Serializability	7.12
	Review Questions	7.12
8.	Concurrency Control	8.1—8.19
8.1	Introduction	8.1
8.2	Three Concurrency Problems	8.1
8.2.1	The Lost Update Problem	8.1
8.2.2	The Uncommitted Dependency Problem	8.2
8.2.3	The Inconsistent Analysis Problem	8.3
8.2.4	Primary Operations Leading to Concurrency Problems	8.4
8.3	Locking	8.5
8.3.1	Locking Protocol or Data Access Protocol	8.5
8.4	Strict Two Phase Locking Protocol	8.6
8.4.1	Last Update Problem	8.6
8.4.2	The Uncommitted Dependency Problem	8.7
8.4.3	The Inconsistent Analysis Problem	8.7
8.5	Deadlock	8.8
8.5.1	Deadlock Detection	8.9
8.5.2	Deadlock Avoidance	8.10
	8.5.2.1 Other Technique that can be used to Prevent the Deadlock Situation	8.11
8.5.3	Deadlock Recovery	8.12
	8.5.3.1 Choice of Deadlock Victim	8.12
	8.5.3.2 Detection Versus Prevention	8.13

8.6	Isolation Level	8.13
8.6.1	Phantoms	8.14
8.7	Intent Locking	8.14
8.8	Dropping Acid	8.15
8.8.1	Immediate Check for Constraint	8.16
8.9	SQL Facilities	8.18
	Review Questions	8.19

9. Database Recovery Techniques

9.1—9.14

9.1	Introduction	9.1
9.2	Different Types of Database Failures	9.1
9.2.1	Recovery Facilities	9.2
9.2.2	Main Recovery Techniques	9.3
9.3	Transaction Recovery	9.3
9.4	System Recovery	9.4
9.4.1	System Failure and Recovery	9.5
9.5	Media Recovery	9.6
9.6	Crash Recovery	9.6
9.7	Recovery Manager	9.6
9.8	Aries Algorithm	9.7
9.8.1	Terminologies Used in Aries	9.7
9.8.2	Elements of Aries	9.9
9.9	Two Phase Commit	9.11
9.9.1	Working	9.11
9.10	Savepoints	9.12
9.11	SQL Facilities for Recovery	9.12
	Review Questions	9.14

10. Record Storage

10.1—10.17

10.1	Introduction	10.1
10.2	Physical Storage Media Overview	10.3
10.2.1	Storage Device Hierarchy	10.4
10.3	Magnetic Disk	10.5
10.3.1	Physical Characteristics	10.5
10.3.2	Disk Controller	10.6
10.3.3	Disk Performance Measurement	10.6
10.3.4	Optimization of Disk-Block Access	10.7
10.4	Raid	10.8
10.4.1	Data Striping	10.9
10.4.2	Redundancy	10.9
10.4.3	Raid Levels	10.10
10.4.4	Choice of Raid Level	10.15

10.4.5	Hardware Issues	10.15
10.5	Tertiary Storage	10.16
10.5.1	Optical Disks	10.16
10.5.2	Magnetic Tapes	10.17
	Review Questions	10.17
11.	Physical Database Design	11.1—11.13
11.1	Introduction	11.1
11.2	Physical Design Steps	11.1
11.3	Operations in Files	11.2
11.4	File Organization	11.3
11.4.1	Fixed-Length Records	11.4
11.4.2	Variable-Length Records	11.7
11.4.2.1	Byte-String Representation	11.7
11.4.2.2	Fixed-Length Representation	11.8
11.5	Organization of Records in Files	11.10
11.5.1	Sequential File Organization	11.10
11.5.2	Clustering File Organization	11.12
	Review Questions	11.13
12.	Indexing and Hashing	12.1—12.16
12.1	Introduction	12.1
12.2	Ordered Indices	12.2
12.2.1	Primary Index	12.2
12.2.2	Secondary Indices	12.3
12.2.3	Multilevel Indices	12.4
12.3	B ⁺ Tree	12.5
12.3.1	Structure of B ⁺ Tree	12.5
12.3.2	Queries on B ⁺ Trees	12.6
12.4	Hashing Techniques	12.7
12.4.1	Hash Functions	12.7
12.4.2	Handling of Bucket Overflows	12.7
12.4.3	Hash Indices	12.9
12.4.4	Dynamic Hashing	12.10
12.4.5	Extendable Hashing	12.10
12.4.6	Queries and Updates	12.11
12.4.7	Comparison with Other Schemes	12.16
	Review Questions	12.16
13.	Query Processing	13.1—13.18
13.1	Introduction	13.1
13.2	Steps in Query Processing	13.1

13.3	Measures of Query Cost	13.3
13.4	Selection Operation	13.4
13.4.1	Basic Algorithms	13.4
13.4.2	Selections Using Indices	13.4
13.4.3	Selections Involving Comparisons	13.5
13.4.4	Implementation of Complex Selections	13.5
13.5	Sorting	13.6
13.5.1	External Sort Merge Algorithm	13.7
13.6	Join Operation	13.8
13.6.1	Nested-Loop Join	13.9
13.6.2	Block Nested-Loop Join	13.9
13.6.3	Indexed Nested-Loop Join	13.10
13.6.4	Merge Join or Sort-merge Join	13.11
13.6.5	Hash Join	13.12
13.7	Database Tuning	13.14
13.7.1	Types of Database Tuning	13.15
13.8	Evaluation of Expressions	13.16
13.8.1	Materialization	13.16
13.8.2	Pipelining	13.17
13.8.3	Implementation of Pipelining	13.17
	Review Questions	13.18

14. PL/SQL**14.1–14.71**

14.1	Introduction	14.1
14.1.1	Structure if PL/SQL	14.2
14.2	PL/SQL Language Elements	14.3
14.3	PL/SQL Scalar Data Types	14.5
14.4	Variable Declaration in PL/SQL	14.9
14.4.1	Initializing Variables in PL/SQL	14.10
14.4.2	Variable Scope in PL/SQL	14.11
14.4.3	Assigning SQL Query Results to PL/SQL Variables	14.12
14.4.4	Declaring a Constant	14.15
14.5	PL/SQL Operator	14.17
14.5.1	Arithmetic Operators	14.17
14.5.2	Relational Operators	14.17
14.5.3	Comparison Operators	14.18
14.5.4	Logical Operators	14.18
14.5.5	PL/SQL Operator Precedence	14.18
14.6	PL/SQL Control Structure	14.19
14.6.1	Conditional Control	14.19
14.6.2	Iterative Control	14.27

14.6.2.1	Labeling a PL/SQL Loop	14.35
14.6.2.2	The Loop Control Statements	14.36
14.7	PL/SQL Strings	14.41
14.7.1	Declaring String Variables	14.42
14.7.2	PL/SQL String Functions and Operators	14.43
14.8	PL/SQL Arrays	14.47
14.8.1	Creating a VARRAY Type	14.47
14.9	PL/SQL Subprogram	14.49
14.9.1	Parts of a PL/SQL Subprogram	14.50
14.9.1.1	PL/SQL Procedure	14.50
14.9.1.2	Creating a Procedure	14.51
14.9.1.3	Deleting a Standalone Procedure	14.53
14.9.1.4	Parameter Modes in PL/SQL Subprograms	14.53
14.9.2	PL/SQL Functions	14.54
14.9.2.1	PL/SQL Recursive Functions	14.58
14.9.2.2	Parameters	14.59
14.10	PL/SQL Cursors	14.60
14.10.1	Implicit Cursors	14.60
14.10.2	Explicit Cursors	14.62
14.11	PL/SQL Packages	14.64
14.11.1	Creating a Package	14.65
14.11.2	Referencing Package Subprograms	14.66
14.11.3	Removing a Package	14.66
14.11.4	Parts of Package	14.66
14.12	PL/SQL Exception	14.68
14.12.1	Raising Exceptions	14.69
14.12.2	User-defined Exceptions	14.70
14.12.3	Pre-defined Exceptions	14.70
	Review Questions	14.71

15. Data Mining & Data Warehousing**15.1—15.33**

15.1	Introduction	15.1
15.1.1	What is Data Mining?	15.1
15.1.2	Essential Step in the Process of Knowledge Discovery in Databases	15.1
15.1.3	Architecture of a Typical Data Mining System	15.2
15.2	Data in Data Mining	15.3
15.2.1	Advanced Database Systems and Advanced Database Applications	15.4
15.3	Data Mining Functionalities	15.5
15.4	Classification of Data Mining Systems	15.7
15.5	Primitives for Specifying a Data Mining Task	15.8
15.6	Integration of a Data Mining System with a Data Warehouse	15.8

15.7	Issues in Data Mining	15.10
15.8	Data Preprocessing	15.11
15.8.1	Data Cleaning	15.12
15.8.1.1	Missing Values	15.13
15.8.1.2	Noisy Data	15.13
15.8.1.3	Data Cleaning as a Process	15.14
15.8.1.4	Disadvantages in data cleaning process	15.15
15.8.2	Data Integration	15.15
15.8.2.1	Handling Redundant Data in Data Integration	15.16
15.8.3	Data Transformation	15.16
15.8.3.1	Normalization	15.16
15.8.4	Data Reduction Techniques	15.17
15.8.4.1	Data Cube Aggregation	15.17
15.8.4.2	Attribute Sub Selection	15.18
15.8.4.3	Dimensionality Reduction	15.19
15.8.4.4	Numerosity Reduction	15.21
15.8.5	Data Discretization and Concept Hierarchies	15.23
15.8.5.1	Concept Hierarchy Generation for Category Data	15.23
15.9	Data Warehouse Introduction	15.24
15.10	Terminology	15.25
15.11	Benefits of Data Warehousing	15.25
15.12	Data Warehouse Characteristics	15.26
15.13	Data Warehouse Architecture and its Seven Components	15.26
15.14	Classification of Data Warehouse Design	15.29
15.14.1	Logical Design	15.29
15.14.2	Physical Design	15.31
	Review Questions	15.33

Index**I.1—I.8**

Chapter 1

Introduction

1.1 INTRODUCTION TO DBMS

In modern society Databases and database systems play a vital role. Most of our day to day activities involve the use of databases. Bank application, ticket reservations for any sort of applications, computerized library etc can be cited as instances for use of databases.

Data: Known facts that can be recorded that have implicit meaning.

Information: When data is processed, organized, structured or presented in a given context it becomes information.

Database: Collection of data.

DBMS: DBMS is a collection of interrelated data and a set of program to access those data. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient and efficient*.

The DBMS manages incoming data, organizes it, and provides ways for the data to be modified or extracted by users or other programs.

Database System: Database and DBMS collectively known as database system. Database system is a computerized record keeping system. It is a repository or container for a collection of computerized data files. Users of the system can perform or request the system to perform a variety of operations such as adding new files to the database, inserting data into existing files, retrieving or deleting data from existing files, modifying data in existing file or removing existing files from the database.

1.1.1 Structure of DBMS

An overview of the structure of database management system is shown in Fig. 1.1. A DBMS is a software package, which translates data from its logical representation to its physical representation and vice versa.

The DBMS uses an application specific database description to define this translation. The database description is generated by a database designer from his or her conceptual view of the database, which is called the Conceptual Schema. The translation from the conceptual schema to

the database description is performed using a Data Definition Language (DDL) or a graphical or textual design interface.

1.1.2 Applications of Database

- **Banking:** for all sorts of bank transactions like withdrawal, deposit etc.
- **Airlines:** for reserving tickets and to prepare schedules
- **Universities:** for student information, registration, grades
- **Sales:** customers, products, purchases
- **Online retailers:** order tracking, customized recommendations
- **Manufacturing:** production, inventory, orders, supply chain
- **Human resources:** employee records, salaries, tax deductions

Databases touch all aspects of our lives.

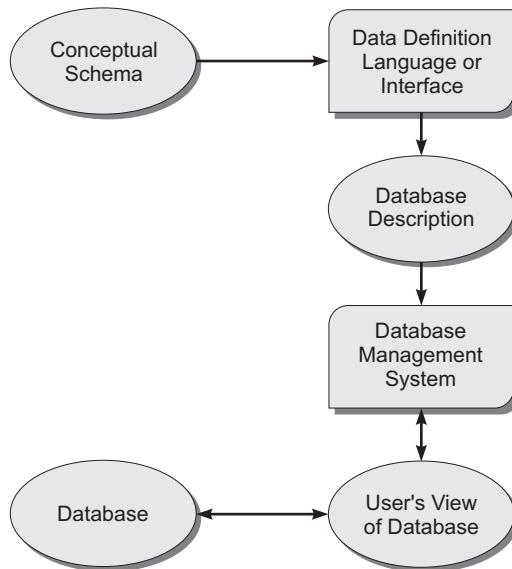


Fig. 1.1. Structure of database management system.

1.1.3 Classification of Database Management System

The database management system can be broadly classified into:

1. Passive Database Management System
2. Active Database Management System

1. **Passive Database Management System:** Passive Database Management Systems are program-driven. In passive database management system the users query the current state of database and retrieve the information currently available in the database. Traditional DBMS is passive. Applications send requests for operations to be performed by the DBMS and wait for the DBMS to confirm and return any possible answers.

- 2. Active Database Management System:** Active Database Management Systems are data-driven or event-driven systems. In active database management system, the users specify to the DBMS the information they need. If the information is currently available, the DBMS actively monitors the arrival of the desired information and provides it to the relevant users. The scope of a query in a passive DBMS is limited to the past and present data, whereas the scope of a query in an active DBMS additionally includes future data.

1.2 FILE PROCESSING SYSTEM

A file processing system is a collection of files and programs that access or modify these files. New files and programs are added by the programmers when there is a need for,

- Storing new information and
- New ways to access information

File processing system is supported by a conventional operating system. The system stores permanent records in various files. The system needs different application programs to extract records from and add records to the appropriate files. Each program defines and manages its own data.

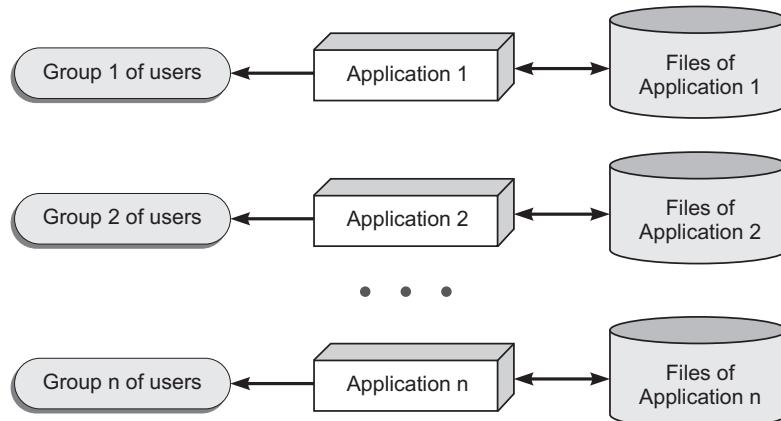


Fig. 1.2. File based system.

1.2.1 Drawbacks of Conventional File Processing System

The conventional file processing system has a number of drawbacks.

- (i) **Data Redundancy and Inconsistency:** Since the files and application programs are created by different programmers over a long period of time, the files have different formats and the programs may be written in several programming language. The same piece of information may be duplicated in several files. For instance, in bank application, the address and phone number of particular customer may appear in a file that consists of personal information and in saving account records file too. This redundancy leads to data consistency, that is, the various copies of the same data may no longer agree.

For example, a changed customer address may be reflected in personal information file, but not in saving account records file.

This redundancy leads to wastage of storage space, high access cost and data inconsistency.

- (ii) **Difficulty in Accessing Data:** Conventional file processing environments do not allow needed data to be retrieved in a convenient and efficient manner.

For example, if a bank officer needs to find out the names of all customers who live within a particular area, he has two choices:

1. Get the list of customers and extract the needed information manually.
2. Ask the data processing department to have a system programmer write the necessary application program. Both alternatives are unsatisfactory.

- (iii) **Data Isolation:** Data is scattered in various files, and as the files may be in different formats, it is difficult to write new application programs to retrieve appropriate data.

- (iv) **Concurrent Access Anomalies:** In order to improve the overall performance of the system and obtain a faster response time many systems allow multiple users to update the data simultaneously. In such environment, interaction of concurrent updates may result in inconsistent data.

Consider bank account A, with ₹ 500. If two customers withdraw funds (say ₹ 50 and ₹ 100 respectively) from account A at the same time, the result of the concurrent executions ₹ 400, rather than ₹ 350. In order to guard against this possibility, some form of supervision must be maintained in the system.

- (v) **Security Problems:** Not every user of the database system should be able to access all the data.

For example, in a banking system, pay roll personnel need to be only part of the database that has information about the various bank employees. They do not need access to information about customer accounts. Since application programs are added to the system in an ad-hoc manner, it is difficult to enforce such security constraints.

- (vi) **Integrity Problems**

- The data values stored in the database must satisfy certain types of consistency constraints.

For example, the balance of a bank account may never fall below a prescribed amount (say ₹ 100). These constraints are enforced in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

- (vii) **Atomicity Problems**

- Like all devices a computer system may also be subjected to failure. If a failure occurs the data existed prior to failure can be restored to the consistent state.
- Consider a program to transfer ₹ 50 from account A to account B. If a system failure occurs during the execution of the program, it is possible that the ₹ 50 is removed from account A but is not credited to account B, resulting in an inconsistent database state.
- The funds transfer must be atomic, it must happen entirely or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

1.3 ADVANTAGES OF DATABASE

Database is a way to consolidate and control the operational data centrally. The DBMS has a number of advantages as compared to traditional computer file processing approach. The DBA must keep in mind these benefits or capabilities while designing databases, coordinating and monitoring the DBMS. The major advantages of having a centralized database are as follows:

(i) **Controlling Data Redundancy**

- In traditional computer file processing, each application program has its own files. In this case, the duplicated copies of the same data are created at many places. In DBMS, all the data of an organization is integrated into a single database. The data is recorded at only one place in the database and it is not duplicated.
- When they are converted into database, the data is integrated into a single database so that multiple copies of the same data are reduced to single copy.
- In DBMS, the data redundancy can be controlled or reduced but is not removed completely. Sometimes, it is necessary to create duplicate copies of the same data items in order to relate tables with each other.
- By controlling the data redundancy, you can save storage space. Similarly, it is useful for retrieving data from database using queries.

(ii) **Data Consistency**

- By controlling the data redundancy, the data consistency is obtained. If a data item appears only once, any update to its value has to be performed only once and the updated value is immediately available to all users.

(iii) **Data Sharing**

- In DBMS, data can be shared by authorized users of the organization. The DBA manages the data and gives rights to users to access the data. Many users can be authorized to access the same set of information simultaneously.
- The remote users can also share same data. Similarly, the data of same database can be shared between different application programs.

(iv) **Data Integration**

- Data in database is stored in tables. A single database contains multiple tables and relationships can be created between tables. This makes easy to retrieve and update data.

(v) **Integrity Constraints**

- Integrity constraints or consistency rules can be applied to database so that the correct data can be entered into database.
- The constraints may be applied to data item within a single record or they may be applied to relationships between records.

(vi) **Data Security**

- Data security is the protection of the database from unauthorized users. Only the authorized persons are allowed to access the database. Some of the users may be allowed to access only a part of database i.e., the data that is related to them or related to their department.

- Mostly, the DBA or head of a department can access all the data in the database. Some users may be permitted only to retrieve data, whereas others are allowed to retrieve as well as to update data. The database access is controlled by the DBA. He creates the accounts of users and gives rights to access the database. Typically, users or group of users are given usernames protected by passwords.
- The user enters his/her account number or username and password to access the data from database.

For example, if you have an account of e-mail in the “gmail.com”, you have to give your correct username and password to access your account of e-mail. Similarly, when you insert your ATM card into the Auto Teller Machine (ATM) in a bank, the machine reads your ID number printed on the card and then asks you to enter your pin code to access your account.

(vii) Data Atomicity

- A transaction in commercial databases is referred to as atomic unit of work. For example, when you purchase something from a point of sale terminal, a number of tasks are performed such as;
 - Company stock is updated.
 - Amount is added in company’s account.
 - Sales person’s commission increases etc.
- All these tasks collectively are called an atomic unit of work or transaction. These tasks must be completed in all; otherwise partially completed tasks are rolled back. Thus through DBMS, it is ensured that only consistent data exists within the database.

(viii) Development of Application

- The cost and time for developing new applications is reduced. The DBMS provides tools that can be used to develop application programs.
- For example,** some wizards are available to generate Forms and Reports. Stored procedures facility reduces the size of application programs.

(ix) Creating Forms

- Form is very important object of DBMS. You can create Forms very easily and quickly in DBMS, once a Form is created, it can be used many times and it can be modified very easily. The created Forms are also saved along with database and behave like a software component.
- A Form provides very easy way (user-friendly interface) to enter data into database, edit data, and display data from database.
- The non-technical users can also perform various operations on databases through Forms without going into the technical details of a database.

(x) Report Writers

- Most of the DBMSs provide the report writer tools used to create reports. The users can create reports very easily and quickly.
- Once a report is created, it can be used many times and it can be modified very easily.

- The created reports are also saved along with database and behave like a software component.

(xi) Control over Concurrency

- In a computer file-based system, if two users are allowed to access data simultaneously, it is possible that they will interfere with each other.

For example, if both users attempt to perform update operation on the same record, then one may overwrite the values recorded by the other. Most DBMSs have sub-systems to control the concurrency so that transactions are always recorded with accuracy.

(xii) Backup and Recovery Procedures

- In a computer file-based system, the user creates the backup of data regularly to protect the valuable data from damaging due to failures to the computer system or application program. It is a time consuming method, if volume of data is large.
- Most of the DBMSs provide the ‘backup and recovery’ sub-systems that automatically create the backup of data and restore data if required.

For example, if the computer system fails in the middle or at the end of an update operation of the program, the recovery sub-system is responsible for making sure that the database is restored to the state it was before the program started executing.

(xiii) Data Independence

- The separation of data structure of database from the application program that is used to access data from database is called data independence.
- In DBMS, database and application programs are separated from each other. The DBMS sits in between them. You can easily change the structure of database without modifying the application program.

For example, you can modify the size or data type of a data items. On the other hand, in computer file-based system, the structure of data items is built into the individual application programs. Thus the data is dependent on the data file and vice versa.

(xiv) Advanced Capabilities

- DBMS also provides advance capabilities for online access and reporting of data through Internet.
- Today, most of the database systems are online.
- The database technology is used in conjunction with Internet technology to access data on the web servers.

1.4 DISADVANTAGES OF DATABASE

Although there are many advantages the DBMS may also have some minor disadvantages. They are:

(i) Cost of Hardware and Software

- A processor with high speed of data processing and memory of large size is required to run the DBMS software.
- It means that you have to upgrade the hardware used for file-based system. Similarly, DBMS software is also very costly.

(ii) Cost of Data Conversion

- When a computer file-based system is replaced with a database system, the data stored into data file must be converted to database files. It is difficult and time consuming method to convert data of data files into database.
- It is necessary to hire a DBA or database designer and system designer along with application programmers.

(iii) Cost of Staff Training

- Most DBMSs are often complex systems so the training for users to use the DBMS is required.
- Training is required at all levels, including programming, application development, and database administration.
- The organization has to pay a lot of amount on the training of staff to run the DBMS.

(iv) Appointing Technical Staff

- The trained technical persons such as database administrator and application programmers are required to handle the DBMS.

(v) Database Failures

- In most of the organizations, all data is integrated into a single database.
- If database is corrupted due to power failure or it is corrupted on the storage media, our valuable data may be lost or the whole system stops.

1.5 VIEWS OF DATA

A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained.

1.5.1 Data Abstraction

The main objective of DBMS is to store and retrieve information efficiently; all the users should be able to access required data. The designers use complex data structure to represent the data, so that data can be efficiently stored and retrieved, but it is not necessary for the users to know physical database storage details because all database users are not computer trained. The developers hide the complexity from users through several levels of abstraction. The data abstraction helps the users easily interact with the database system. There are three levels of data abstraction:

- Physical level:** It is the lowest level of abstraction that describes how the data are actually stored. The physical level describes complex low-level data structures in details.
- Logical level:** It is the next higher level of abstraction that describes what data are stored in the database and what relationships exist among those data.
- View level:** It is the highest level of abstraction that describes only part of the entire database.

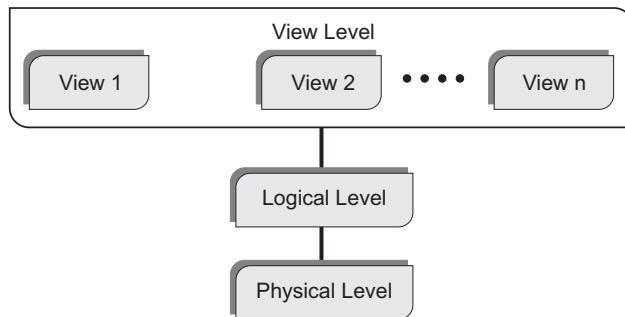


Fig. 1.3. The three levels of data abstraction.

1.5.2 Data Independence

The ability to modify a scheme definition in one level without affecting a scheme definition in the next higher level is called data independence.

There are two levels of data independence:

1. **Physical data independence** is the ability to modify the physical scheme without causing application programs to be rewritten. Modifications at the physical level are occasionally necessary in order to improve performance.
2. **Logical data independence** is the ability to modify the conceptual scheme without causing application programs to be rewritten. Modifications at the conceptual level are necessary whenever the logical structure of the database is altered.

Logical data independence is more difficult to achieve than physical data independence since application programs are heavily dependent on the logical structure of the data they access.

1.5.3 Instances and Schemas

Database change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an instance of the database. The overall design of the database is called the database schema.

Types of database schemas

- (i) **Physical schema:** It describes the database design at the physical level.
- (ii) **Logical schema:** It describes the database design at the logical level.
- (iii) **Subschema:** A database may also have several sub schemas at the view level called as subschemas that describe different views of the database.

1.6 DATABASE SYSTEM STRUCTURE

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into

- Storage Manager
- Query Processor

Database System Structure

(i) Storage Manager

- A storage manager is a program module that provides the interface between the low level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible for the interaction with the file manager.
- The storage manager translates the various DML statements into low-level file system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

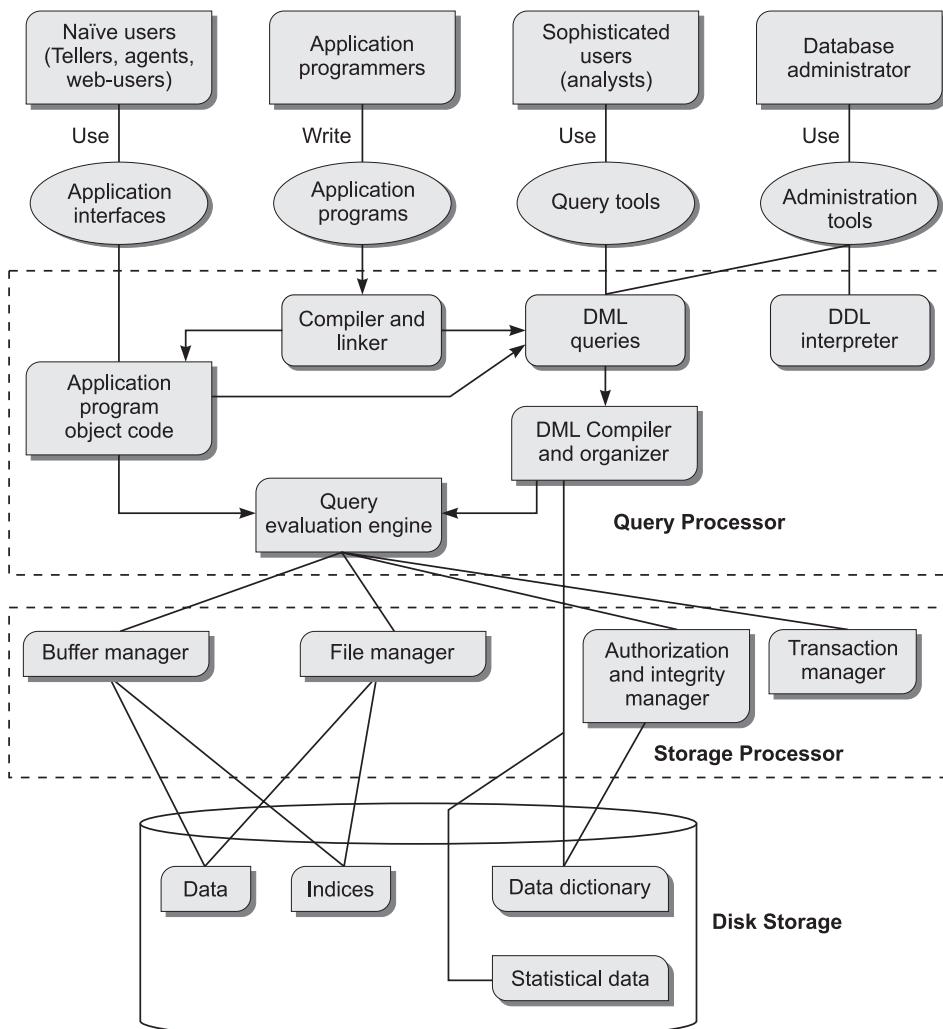


Fig. 1.4. Database system architecture.

Components of the storage manager are:

- **Authorization and integrity manager:** It tests for satisfaction of various integrity constraints and checks the authority of users accessing the data.
- **Transaction manager:** It ensures that the database remains in a consistent state despite system failures, and concurrent executions proceed without conflicting.
- **File manager:** It manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager:** It is responsible for fetching data from disk storage into main memory and deciding what data to cache in main memory.

The storage manager implements several data structures as part of physical system implementation.

- **Data files:** Which store the database itself
- **Data dictionary:** It contains metadata that is data about data. The schema of a table is an example of metadata. A database system consults the data dictionary before reading and modifying actual data.
- **Indices:** Which provide fast access to data items that hold particular values.

(ii) The Query Processor

The query processor is an important part of the database system. It helps the database system to simplify and facilitate access to data.

The query processor components include:

- **DDL interpreter,** which interprets DDL statements and records the definitions in the data dictionary.
- **DML compiler,** which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

A query can be translated into any number of evaluation plans that give the same result. The DML compiler also performs query optimization, that is, it picks up the lowest cost evaluation plan from among the alternatives.

Query evaluation engine, which executes low-level instructions generated by the DML compiler.

1.6.1 Database Users and Administrators

People who work with a database can be categorized as:

- Database users
- Database administrators

1.6.1.1 Database Users

1. Naïve Users

- Naïve users who interact with the system by invoking one of the application programs that have been written previously.
- Naïve users use a form interface, where the user can fill in appropriate fields of the form. Naïve users may also simply read reports generated from the database.

2. Application Programmers

- Application programmers are computer professionals who write application programs.
- Rapid application development (RAD) tools are tools that enable an application programmer to construct forms and reports without writing a program.
- Special types of programming languages that combine control structures with data manipulation language. These languages are sometimes called as fourth-generation languages.

3. Sophisticated Users

- Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a query processor that the storage manager understands.
- Online analytical processing (OLAP) tools simplify analyst's tasks.
- Another tool for analysts is data mining tools, which help them to find certain kinds of patterns in data.

4. Specialized Users

- Specialized users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.
- The applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types

1.6.1.2 Database Administrators

A person who has such central control over the system is called a database administrator (DBA).

The functions of a DBA

- **Schema definition:** The DBA creates the original database schema by executing a set of data definition statements in the DDL.
- **Storage structure and access-method definition.**
- **Schema and physical-organization modification:** The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization.
- **Granting of authorization for data access:** By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
- **Routine maintenance:** Examples of the database administrator's routine maintenance activities are:
 1. Periodically backing up the database
 2. Ensuring that enough free disk space
 3. Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

1.6.2 Application Architectures

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between **client** machines, on which remote database users work, and server machines, on which the database system runs.

Database applications are usually partitioned into two or three parts, and according to the partition they are named as:

1. Two-tier architecture
2. Three-tier architecture

1.6.2.1 Two-tier Architecture

In this architecture, the application is partitioned into a component that resides at the client machine, which invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

1.6.2.2 Three-tier Architecture

In this architecture, the client machine acts as front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface. The application server in turn communicates with a database system to access data. Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.

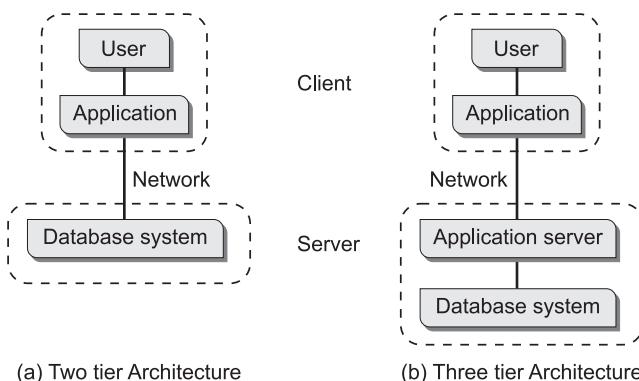


Fig. 1.5. Application architecture.

REVIEW QUESTIONS

1. Define

(a) Data	(b) Information	(c) Database
(d) DBMS	(e) Database System	
2. List out the applications of database.

3. Explain the drawbacks of conventional file processing system over DBMS.
4. Explain the merits and demerits of DBMS.
5. Define data abstraction. How is data abstraction implemented in database system?
6. With a neat sketch explain the different components of Database management system and its functionality.
7. Explain application architecture in details.
8. List out the roles of database users and administrators.



Chapter 2

Data Models

2.1 DATA MODELS

- Underlying structure of the database is called as data models.
- It is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

Different types of data models are:

- Hierarchical model
- Network model
- Object oriented model
- Object relational model
- Relational model
- Entity relationship model

2.2 HIERARCHICAL MODEL

The hierarchical data model organizes data in a tree structure. In this model, each entity has only one parent but can have several children. Only one entity at the top of the hierarchy is called as Root. The structure is based on the rule that one parent can have many children but children are allowed only one parent. Linkages are only possible vertically but not horizontally or diagonally, i.e. there is no relation between different trees at the same level unless they share the same parent.

Advantages

- High speed of access to large datasets.
- **Data security:** Hierarchical model was the first database model that offered the data security that is provided and enforced by the DBMS.
- **Efficiency:** The hierarchical database model is very efficient when the database contains a large number of transactions using data whose relationships are fixed.
- The model allows easy addition and deletion of new information. Data at the top of the Hierarchy is very fast to access. It is very easy to work with the model because it works

well with linear type data storage such as tapes. The model relates very well to natural hierarchies such as assembly plants and employee organization in corporations. It relates well to anything that works through one to many relationships.

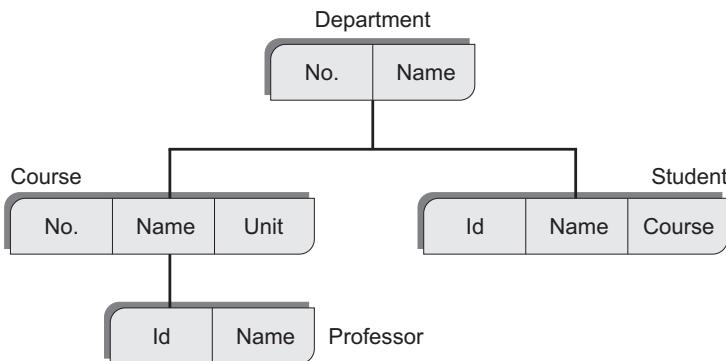


Fig. 2.1. Hierarchical model.

Disadvantages

- Implementation complexity
- Database management problems
- Lack of structural independence.
- This model cannot be used for more sophisticated relationships. It requires data to be repetitively stored in many different entities. The database can be very slow when searching for information on the lower entities.
- We no longer use linear data storage mediums such as tapes so that advantage is null.
- Searching for data requires the DBMS to run through the entire model from top to bottom until the required information is found, making queries very slow. It can only model one to many relationships; many to many relationships are not supported.

2.3 NETWORK MODEL

The Network Data Model is also known as the “CODASYL Data Model” or sometimes as “DBTG Data Model.” The model is based on directed graph theory. The network model replaces the hierarchical tree with a graph thus allowing more general connections among the nodes. The main difference of the network model from the hierarchical model is its ability to handle many-to-many ($n : n$) relationship or in other words, it allows a record to have more than one parent.

Advantages

- Conceptual simplicity
- Capability to handle more relationship types
- Data independence

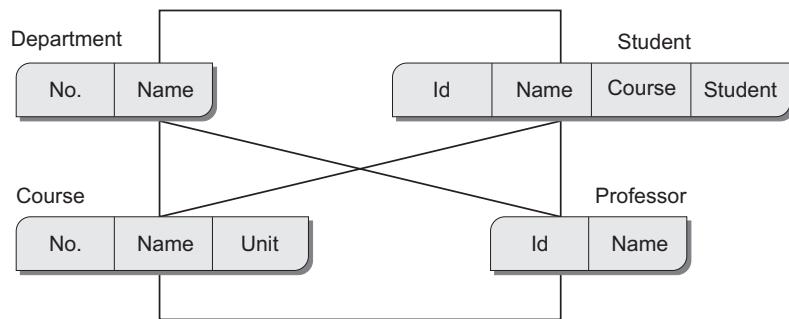


Fig. 2.2. Network model.

Disadvantages

- Detailed structural knowledge is required
- Lack of structural independence

2.4 OBJECT-ORIENTED MODEL

Object DBMSs add database functionality to object programming languages. Object DBMSs extend the semantics of the C++, Smalltalk and Java object programming languages to provide full-featured database programming capability, while retaining native language compatibility. A major benefit of this approach is the unification of the application and database development into a seamless data model and language environment. As a result, applications require less code, use more natural data modeling, and code bases are easier to maintain. Object developers can write complete database applications with a modest amount of additional effort.

The object-oriented model is based on a collection of objects. An object contains values stored in instance variables within the object. An object also contains bodies of code that operate in the object, these bodies of code are called methods. Objects that contain the same types of values and the same methods are grouped together into classes.

Advantages

- Applications require less code
- Applications use more natural data model
- Code is easier to maintain
- It provides higher performance management of objects and complex interrelationships between objects
- Object-oriented features improve productivity
- Data access is easy.

2.5 OBJECT RELATIONAL MODEL

Object relational database management systems (ORDBMSs) add new object storage capabilities to the relational systems at the core of modern information systems. These new facilities integrate management of traditional fielded data, complex objects such as time-series and geospatial data

and diverse binary media such as audio, video, images, and applets. By encapsulating methods with data structures, an ORDBMS server can execute complex analytical and data manipulation operations to search and transform multimedia and other complex objects.

A system that includes both object infrastructure and set relational extenders. Object-relational systems combine the advantages of modern object-oriented programming languages with relational database features such as multiple views of data and a high-level, non-procedural query language. Some of the object-relational systems available in the market are IBM DB2 universal server, oracle corporation's oracle 8, Microsoft Corporations SQL server 7 and so on.

2.6 RELATIONAL MODEL

The relational model was introduced by Dr. E. F. Codd in 1970. The relational model represents data in the form of two dimensional tables. The organization of data into relational tables is known as the logical view of the database. Software such as Oracle, Microsoft SQL Server, Sybase, are based on the relational model.

2.6.1 Characteristics of Relational Model

- The relational model eliminated all parent child relationships and instead represented all data in the database as simple row/column tables of data values.
- A relation is similar to a table with rows/columns of data values.
- Each table is an independent entity and there is no physical relationship between tables.
- Most data management systems based on the relational model have a built-in support for query languages like ANSI SQL or QBE. These queries are simple English constructs that allow adhoc data manipulation from a table.
- Relational model of data management is based on set theory.
- The user interface used with relational models is non-procedural because only what needs to be done is specified and not how it has to be done.

2.6.2 E. F. Codd's Laws for a Fully Functional Relational Database Management System

CODD's 12 rules define an ideal relational database which is used as a guideline for designing relational database systems. Though no commercial database system completely conforms to all 12 rules, they do interpret the relational approach. The CODD's 12 rules are as follows:

Rule 0: Foundation rule: The system must qualify as relational both as a database and as a management system.

Rule 1: The information rule: All information in the database must be represented in one and only one way (that is, as values in a table).

Rule 2: The guaranteed access rule: All data should be logically accessible through a combination of table name, primary key value and column name.

Rule 3: Systematic treatment of null values: A DBMS must support Null Values to represent missing information and inapplicable information in a systematic manner independent of data types.

Rule 4: Active online catalog based on the relational model: The database must support online relational catalog that is accessible to authorized users through their regular query language.

Rule 5: The comprehensive data sublanguage rule: The database must support at least one language that defines linear syntax functionality, supports data definition and manipulation operations, data integrity and database transaction control.

Rule 6: The view updating rule: Representation of data can be done using different logical combinations called Views. All the views that are theoretically updatable must also be updatable by the system.

Rule 7: High-level insert, update, and delete: The system must support set at the time of insert, update and delete operators.

Rule 8: Physical data independence: Changes made in physical level must not have any impact and require a change to be made in the application program.

Rule 9: Logical data independence: Changes made in logical level must not impact and require a change to be made in the application program.

Rule 10: Integrity independence: Integrity constraints must be defined and separated from the application programs. Changing Constraints must be allowed without affecting the applications.

Rule 11: Distribution independence: The user should be unaware about the database location i.e. whether or not the database is distributed in multiple locations.

Rule 12: The non subversion rule: If a system provides a low level language, then there should be no way to subvert or bypass the integrity rules of high-level language. Of all the rules, rule 3 is the most controversial. This is due to a debate about three-valued or ternary logic. Codd's rules and SQL use ternary logic, where null is used to represent missing data and comparing anything to null results in an unknown truth state. However, when both Booleans or operands are false, the operation is false; therefore, not all data that is missing is unknown, hence the controversy.

2.6.3 Principle Components of Relational Model

The relational model consists of following three basic components:

1. Data structure
2. Data integrity
3. Data manipulation

1. The Relational Data Structure:

The relational data structure is shown in Fig 2.3. It is based on the employee relation.

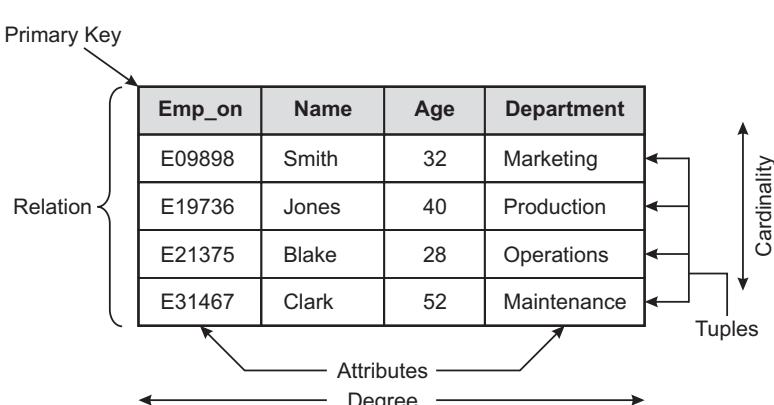


Fig. 2.3. The relational data structure.

Structural terminology summary

<i>Formal relational term</i>	<i>Informal Equivalence</i>
Relation	Table
Tuple	Row or record
Cardinality	Number of rows
Attribute	Column or field
Degree	Number of columns
Primary key	Unique identifier
Domain	Pool of legal values

- 2. Relational Integrity:** Integrity constraints means when changes made to the database by authorized users that should not result in a loss of data consistency.

There are two main types of integrity constraints.

- Domain constraints
- Referential integrity

Domain Constraints

- Domain constraints specify the set of values that can be associated with an attribute.
- Domain constraints are tested easily by the system whenever a new data item is entered into the database.
- Domain constraints also prohibit use of null values for particular fields.

Referential Integrity

- A value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This is called referential integrity.

Referential Integrity in the E-R Model: Referential integrity constraints arise frequently. If we derive our relational database scheme by constructing tables from E-R diagrams then every relation arising from a relationship set has referential integrity constraints.

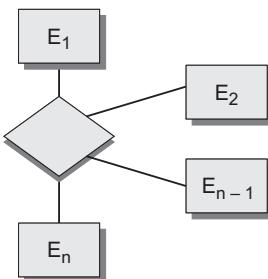


Fig. 2.4. Referential integrity in the E-R model.

As shown in Fig. 2.4, an n-array relationship set R, relating entity sets E₁, E₂, ..., E_n. Let K_i denote the primary key of E_i. The attributes of the relation scheme for relationship set

R include $K_1 \cup K_2 \cup \dots \cup K_n$. Each K_i in the scheme for R is a foreign key that leads to a referential integrity constraints.

Referential Integrity in SQL: Using SQL primary key, candidate key, and foreign key are defined as part of the create table statement as given below

Example: Create table deposit (br-name char (15), acc-no char(10), cust-name cher (20) not null, balance integer, primary key (acc-no, cust-name), foreign key (branch-name), Foreign key(cust-name) references customer);

Null: “Null represents a value for an attribute that is currently unknown or is not applicable for this tuple.”

Other integrity constraints are:

- Entity integrity
- Enterprise constraints.

Entity Integrity: “In a base relation, no attribute of a primary key can be null”.

A primary key is used to identify tuples uniquely. This means that no subset of the primary key is sufficient to provide unique identification of tuples. Therefore, primary key should not be null.

Enterprise Constraints: These are additional rules specified by the uses or database administrators of a database.

3. **Data Manipulation:** The manipulated part of the relational algebra model consists of a set of operators known collectively as the relational algebra together with relational calculus.

Advantages of Relational Model

- Structural independence
- Conceptual simplicity
- Design, implementation, maintenance and usage ease
- Good for adhoc requests
- It is simpler to navigate
- Greater flexibility.

Disadvantages of Relational Model

- Significant hardware and software overheads
- Not as good for transaction process modeling as hierarchical and network models
- May have slower processing than hierarchical and network models.

2.7 COMPARISON BETWEEN THE VARIOUS DATABASE MODELS —

Model	Data element organization	Relationship representation	Identity	Access language
Hierarchical	Files, records	Tree	Record based	Procedural
Network	Files, records	Graph	Record based	Procedural
Relational	Tables	Foreign key concept	Value based	Non-Procedural
Object-oriented	Objects	Logical containment	Record based	Procedural
Object- Relational	Objects	Relational extenders	Value based	Non-Procedural

2.8 ENTITY RELATIONSHIP MODEL

The E-R data model considers the real world consisting of a set of basic objects, called entities, and relationships among these objects.

2.8.1 Basic Concepts

The E-R data model employs three basic notions:

- Entity
- Attributes
- Relationship sets

2.8.1.1 Entity

An *entity* is ‘thing’ or ‘object in the real world that is distinguishable from all other objects.

For example, each person is an entity.

An entity has a set of properties, and the values for some set of properties may uniquely identify an entity.

For example, a customer with customer-id property with value C101 uniquely identifies that person.

An entity may be concrete, such as person or a book, or it may be abstract, such as a loan, or a holiday.

Entity sets: An *entity set* is a set of entities of the same type that share the same properties, or attributes.

2.8.1.2 Attributes

- The properties that describe an entity are called attributes.
- An entity is represented by a set of attributes.
- The attributes of customer entity set are customer_id , customer_name and city.

Attributes are classified as:

1. Simple
2. Composite
3. Single-valued
4. Multi-valued
5. Derived
6. Stored

1. Simple attribute: Cannot be subdivided

Example: Age, sex, GPA.

2. Composite attribute: Can be subdivided

Example: Address: street, city state zip.

3. Single-valued attribute: Has only a single value

Example: Social security number.

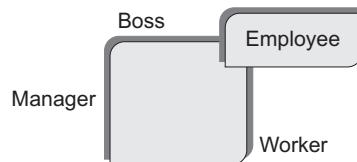
4. **Multi-valued attribute:** Can have many values
Example: Person may have several college degrees.
5. **Derived attribute:** Can be calculated from other information
Example: Age can be derived from D.O.B.
6. **Stored attributes:** The attributes stored in a database are called stored attributes.

2.8.1.3 Relationships and Relationships Sets

- **Relationship** is an association among several entities.
- **Relationship set** is a set of relationships of the same type.

Types of Relationships

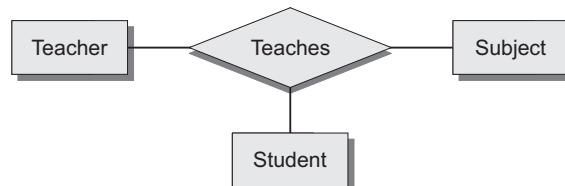
- (i) **Unary relationship:** A unary relationship exists when an association is maintained within a single entity.



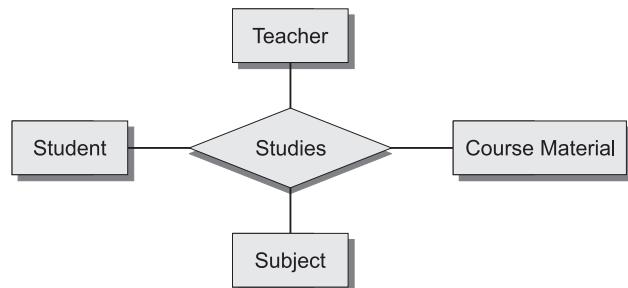
- (ii) **Binary relationship:** A binary relationship exists when two entities are associated.



- (iii) **Ternary relationship:** A ternary relationship exists when there are three entities associated.



- (iv) **Quaternary relationship:** A quaternary relationship exists when there are four entities associated.



Entity role: The function that an entity plays in a relationship is called that entity's role. A role is one end of an association.



Here, Entity role is Employee.

2.8.2 Constraints

An E-R enterprise schema may define certain constraints to which the contents of a database system must conform.

Two types of constraints are:

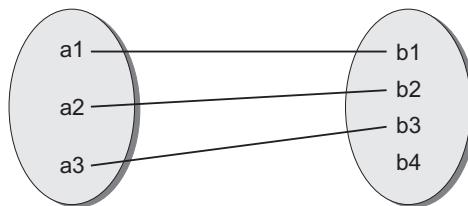
- Mapping cardinalities
- Participation constraints

2.8.2.1 Mapping Cardinalities

- Mapping cardinalities express the number of entities to which another entity can be associated via a relationship set.
- Cardinality in E-R diagram that is represented by two ways:
 - (i) Directed line (→)
 - (ii) Undirected line (—)

For a binary relationship set R between entity sets A and B, the mapping cardinalities must be one of the following:

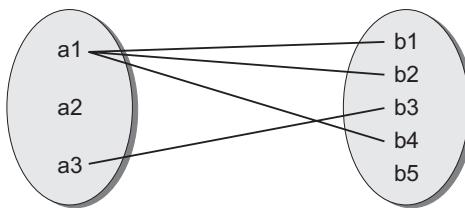
- (i) **One to one:** An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.



Example: A customer with single account at a branch is shown by one-to-one relationship as given below.



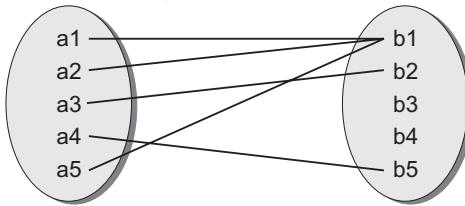
- (ii) **One-to-many:** An entity in A is associated with any number of entities (zero or more) in B. An entity in B, however, can be associated with at most one entity in A.



Example: A customer having two accounts at a given branch is shown by one-to-many relationship as given below.



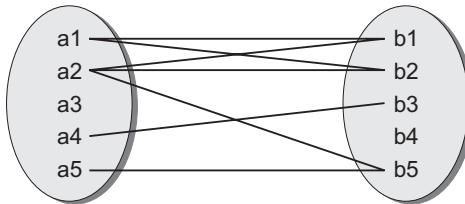
- (iii) **Many-to-one:** An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A.



Example: Many employees works for a company. This relationship is shown by many-to-one as given below.



- (iv) **Many-to-many:** An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.



Example: Employee works on number of projects and project is handled by number of employees. Therefore, the relationship between employee and project is many-to-many as shown below.



2.8.2.2 Participation Constraint

- Participation can be divided into two types.
 1. Total
 2. Partial
- If every entity in E participates in at least one relationship in R, the participation is called Total Participation
- If only some entities in E participate in relationships in R, then the participation is called Partial Participation.

2.8.3 Keys

- A key allows us to identify a set of attributes and thus distinguishes entities from each other.
- Keys also help uniquely identify relationships, and thus distinguish relationships from each other.

Key Type	Definition
Superkey	Any attribute or combination of attributes that uniquely identifies a row in the table. <i>Example:</i> Roll_No attribute of the entity set ‘student’ distinguishes one student entity from another.
Candidate Key	Minimal Superkey is called candidate key. A superkey that does not contain a subset of attributes that is itself a superkey. <i>Example:</i> Student_name and Student_street, are sufficient to uniquely identify one particular student.
Primary Key	The candidate key selected to uniquely identify all rows. Cannot contain null values <i>Example:</i> Roll_No is a primary key of ‘student’ entity set.
Foreign Key	An attribute (or combination of attributes) in one table that must either match the primary key of another table or be null <i>Example:</i> Consider in the staff relation the branch_no attribute exists to match staff to the branch office they work in. In the staff relation, branch_no is foreign key.
Secondary Key	An attribute or combination of attributes is used to make data retrieval more efficient.

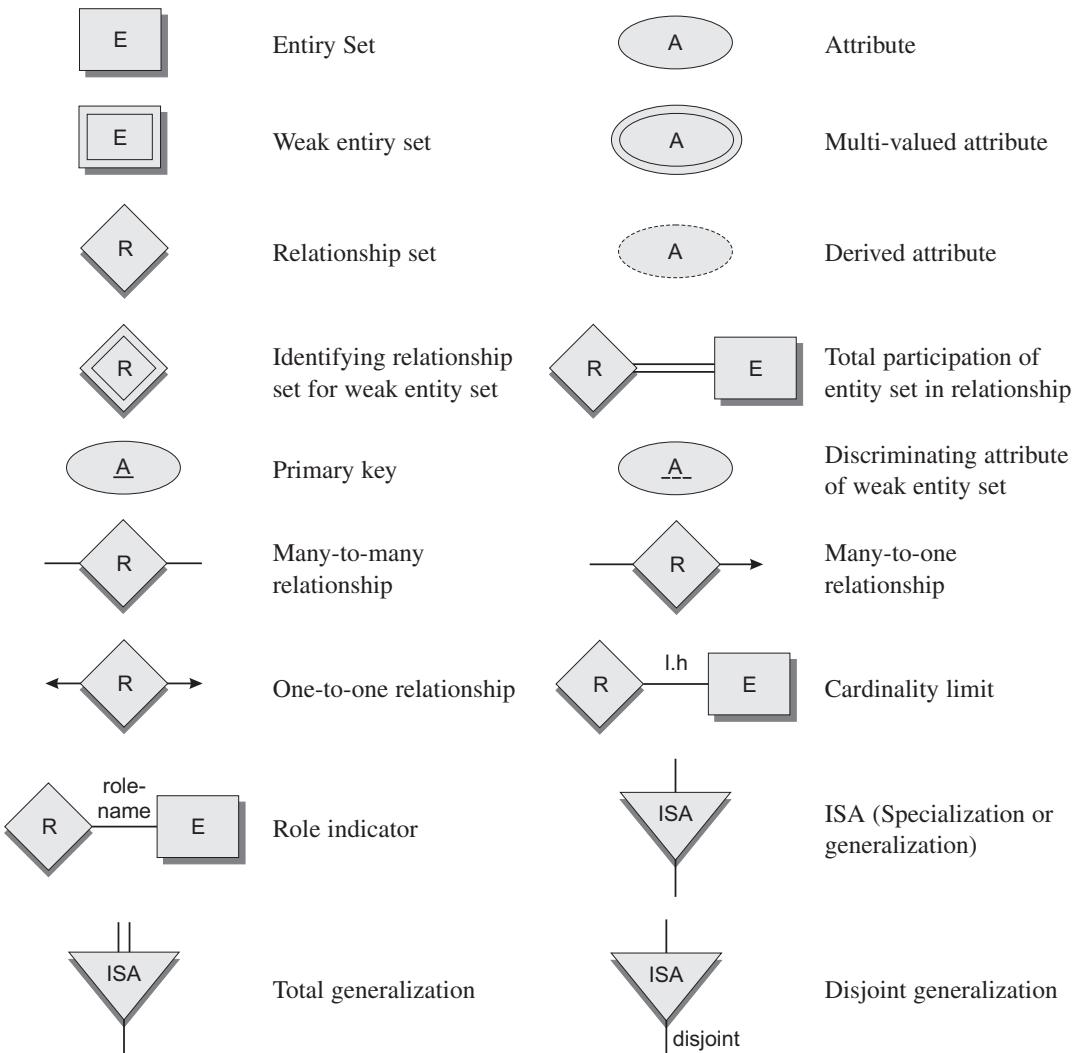
2.8.4 Entity-Relationship Diagram

- E-R diagram** can express the overall logical structure of a database graphically.
- E-R diagram** consists of the following major components:

Component name	Symbol	Description
Rectangles		Represent entity sets
Ellipses		Represent attributes
Diamonds		Represent relationship sets

Lines		Link attributes to entity sets and entity sets to relationship sets
Double ellipses		Represent multi-valued attributes
Dashed ellipses		Represent derived attributes
Double lines		Represent total participation of an entity in a relationship set
Double rectangles		Represent weak entity sets

Summary of ER diagram notation



2.8.5 Dependency

Existence Dependencies

- If the existence of entity x depends on the existence of entity y, then x is said to be existence dependent on y.
- If y is deleted, so is x.
- Entity y is said to be a dominant entity, and x is said to be subordinate entity.

Example: Figure 2.5 shows the dominant entity set ‘loan’ which is also called **Strong entity set**, and subordinate entity set ‘payment’ which is also called **weak entity set**, connected by relationship ‘loan-payment’ .

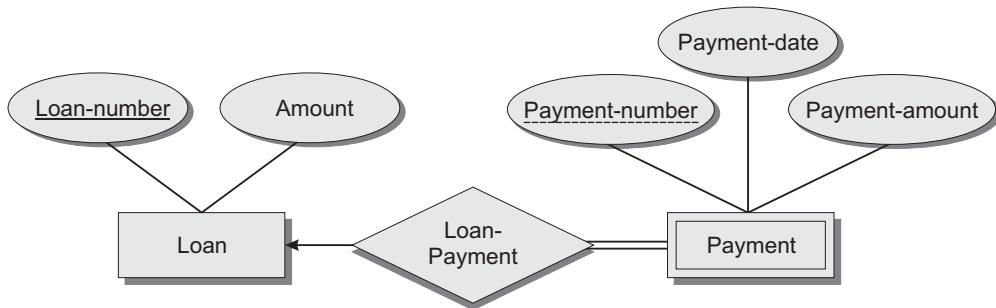
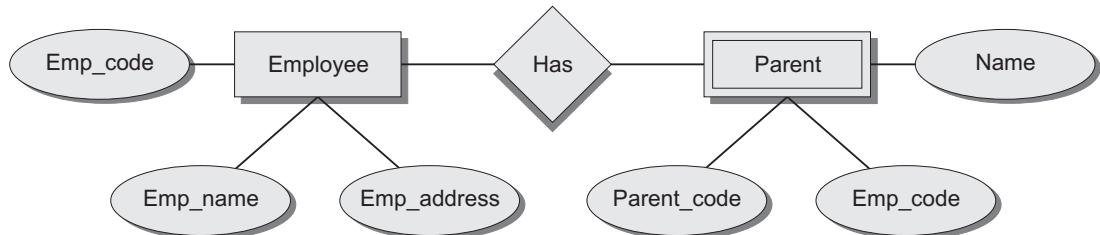


Fig. 2.5. Dependency.

Definition: Strong and Weak Entity Sets

- An entity that is existence dependent on some other entity is called a weak entity type.
- An entity set on which weak entity set depends is called strong entity set.

Example: Figure below shows weak entity set ‘Parent’ which depends on strong entity set ‘Employee’.



Representation of Role

- The function that an entity plays in a relationship is called its role.

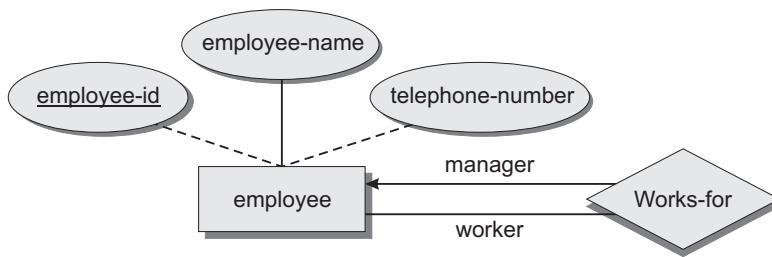


Fig. 2.6. E-R Diagram with role indicators.

Strong Entity Set	Weak Entity Set
<ol style="list-style-type: none"> 1. It has its own primary key. 2. It is represented by a rectangle. 3. It contains the primary key represented by an underline. 4. The members of strong entity set is called as dominant entity set. 5. The primary key is one of its attributes which uniquely identifies its member. 6. The relationship between two strong entity set is represented by a diamond symbol. 7. The line connecting strong entity set with the relationship is single. 8. The total participation in the relationship may or may not exist. 	<ol style="list-style-type: none"> 1. It does not have sufficient attributes to form a primary key on its own 2. It is represented by double rectangle. 3. It contains a Partial key or discriminator represented by a dashed line. 4. The member of weak entity set is called as subordinate entity set. 5. The primary key of weak entity set is a combination of partial key and primary key of the strong entity set. 6. The relationship between one strong and one weak entity set is represented by a double diamond sign. It is known as identifying relationship. 7. The line connecting weak entity set with the identifying relationship is double. 8. Total participation in the identifying relationship always exists.

2.8.6 Sample E-R Diagram

1. Total participation of an entity set in a relationship set

- In Fig: 2.6., Double line from loan to borrower indicates that each loan must have at least one associated customer

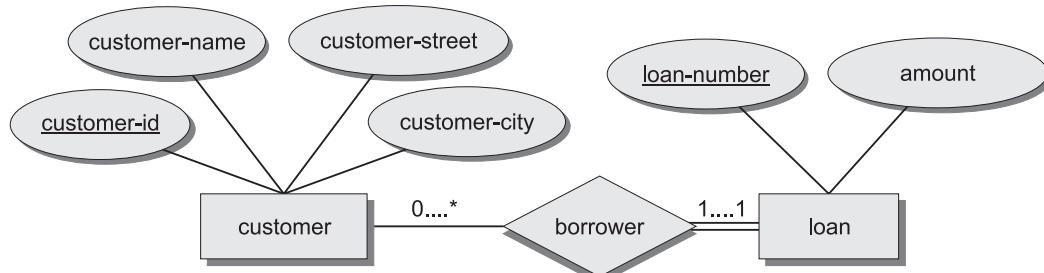


Fig. 2.7. Total participation of an entity set in a relationship set.

2. Cardinality limits on relationship sets

- In Fig. 2.8, edge between *loan* and *borrower* has a cardinality constraint of 1...1, meaning that each loan must have exactly one associated customer.
- The limit 0...* on the edge from *customer* to *borrower* indicates that a customer can have zero or more loans.

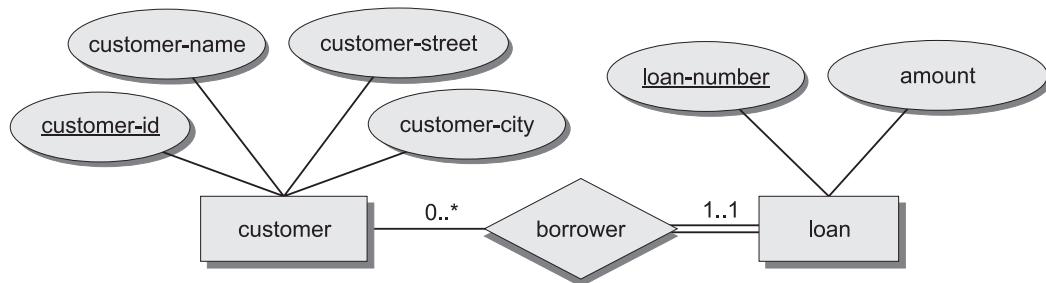


Fig: 2.8. Cardinality limits on relationship sets.

3. E-R diagram with an attribute attached to a relationship set

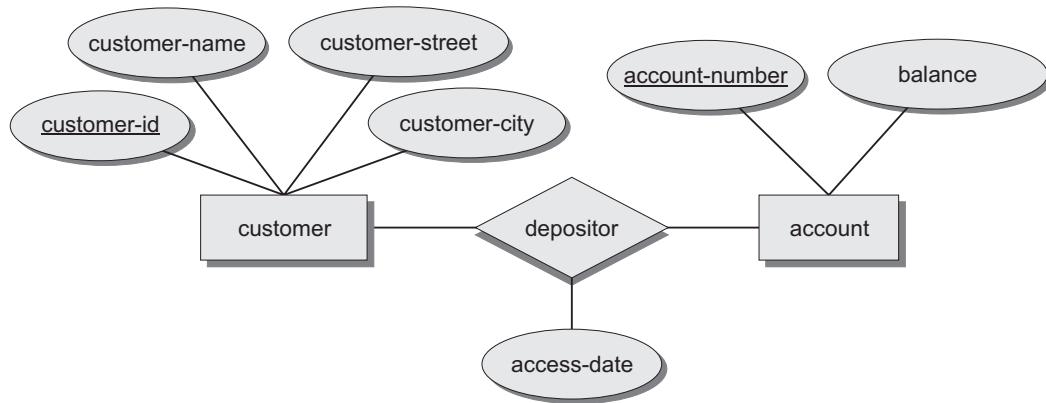


Fig. 2.9 E-R diagram with an attribute attached to a relationship set.

2.8.7 E-R Diagram

Let us design an Entity Relationship (ER) model for a college database.

- A college contains many departments.
- Each department can offer any number of courses.
- Many instructors can work in a department.
- An instructor can work only in one department.
- For each department there is a head.

- An instructor can be head of only one department.
- Each instructor can take any number of courses.
- A course can be taken by only one instructor.
- A student can enroll for any number of courses.
- Each course can have any number of students.

Step 1: Identify the Entities

What are the entities here?

From the statements given, the entities are;

1. Department
2. Course
3. Instructor
4. Student

Step 2 : Identify the relationships

1. One department offers many courses. But one particular course can be offered by only one department. Hence the cardinality between department and course is One to Many (1:N)
2. One department has multiple instructors. But instructor belongs to only one department. Hence the cardinality between department and instructor is One to Many (1:N)
3. One department has only one head and one head can be the head of only one department. Hence the cardinality is one to one. (1:1)
4. One course can be enrolled by many students and one student can enroll for many courses. Hence the cardinality between course and student is Many to Many (M:N)
5. One course is taught by only one instructor. But one instructor teaches many courses. Hence the cardinality between course and instructor is Many to One (N :1)

Step 3: Identify the key attributes

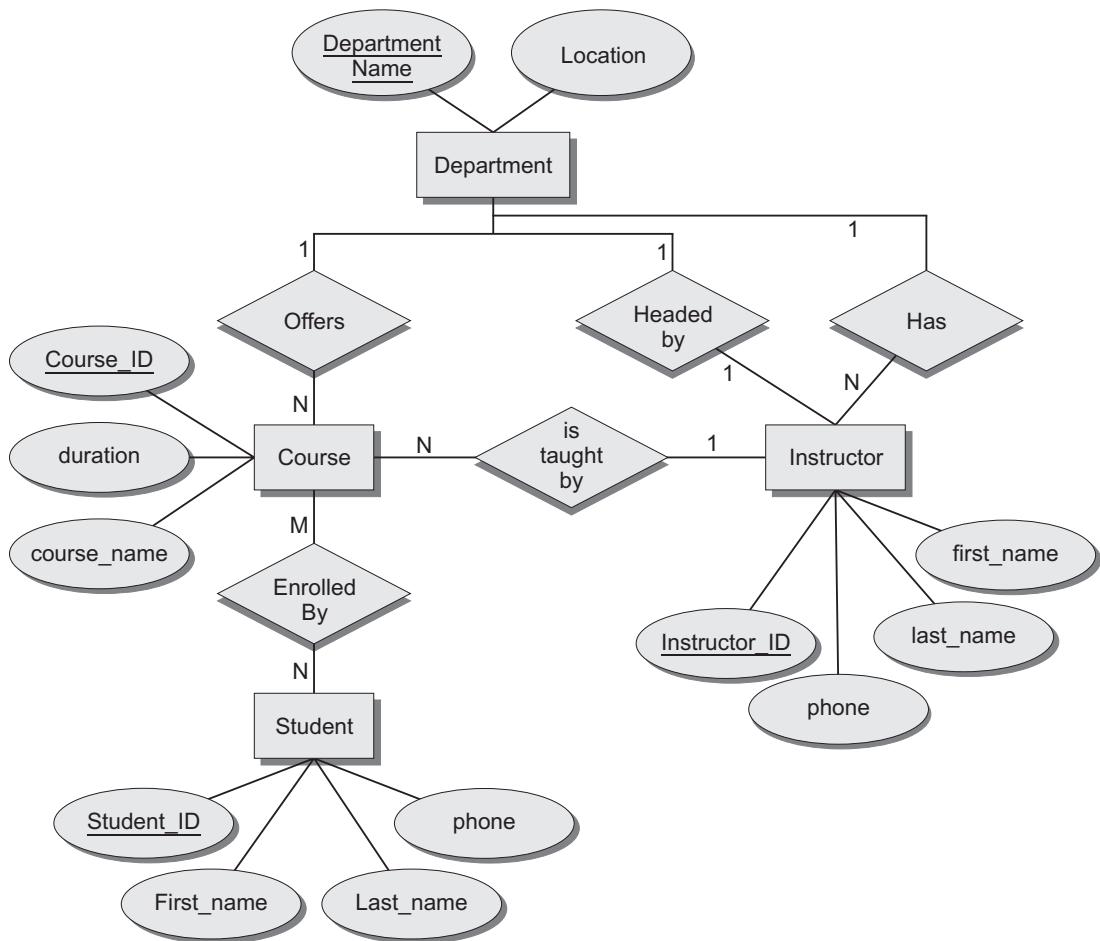
- “Departmen_Name” can identify a department uniquely. Hence Department_Name is the key attribute for the Entity “Department”.
- Course_ID is the key attribute for “Course” Entity.
- Student_ID is the key attribute for “Student” Entity.
- Instructor_ID is the key attribute for “Instructor” Entity.

Step 4: Identify other relevant attributes

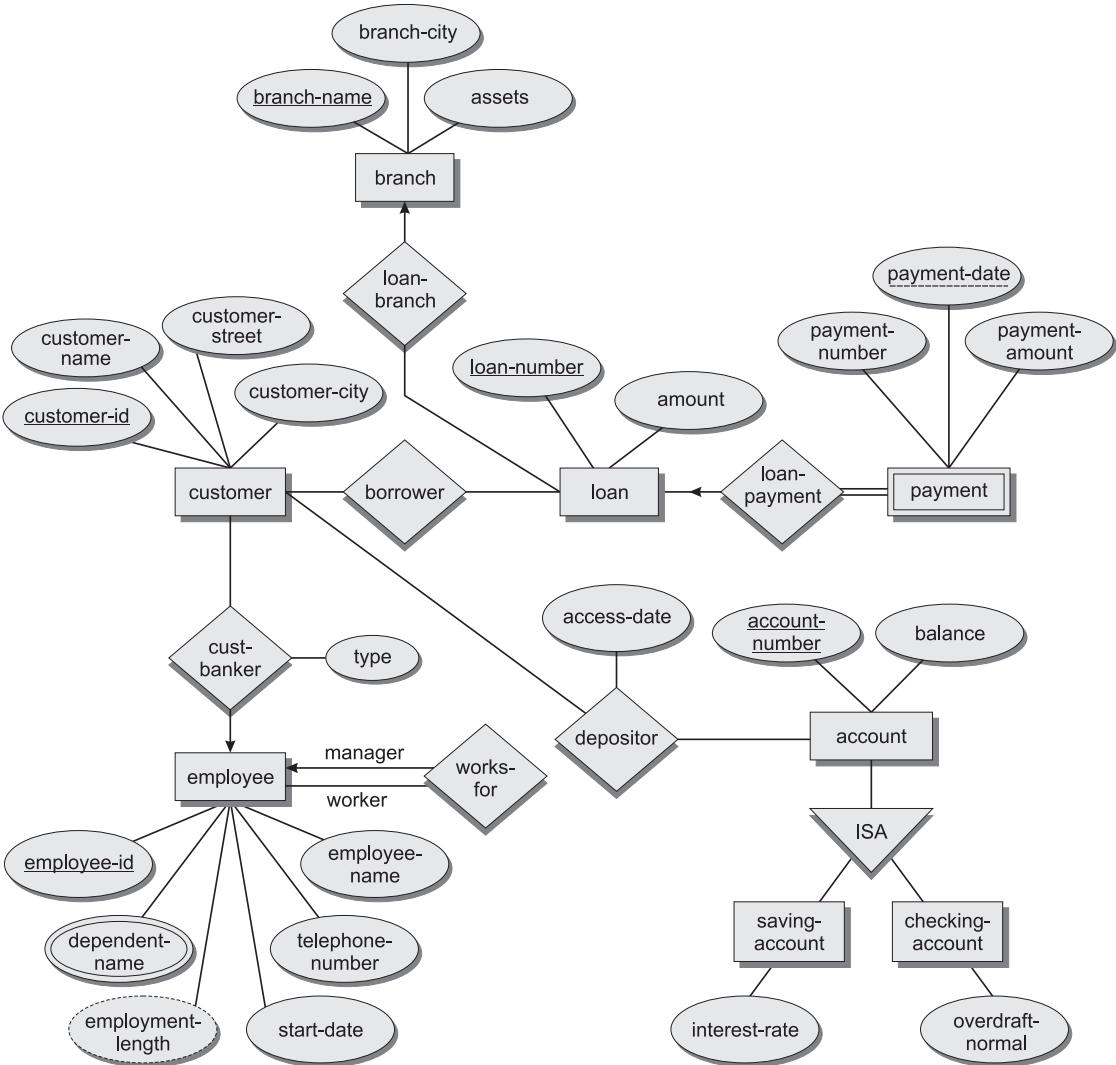
- For the department entity, other attributes are location
- For course entity, other attributes are course_name, duration
- For instructor entity, other attributes are first_name, last_name, phone
- For student entity, first_name, last_name, phone

Step 5: Draw complete E-R diagram

By connecting all these details, we can now draw E-R diagram as given below.



Another example for E-R diagram – Bank Enterprises



2.8.8 Extended E-R Features

- ER model that is supported with the additional semantic concepts is called the extended entity relationship model or EER model.
- EER model deals with
 1. Specialization
 2. Generalization
 3. Aggregation

2.8.8.1 Specialization

- The process of designating sub groupings within an entity set is called **Specialization**.
- **Specialization** is a top-down process.
- Consider an entity set *person*. A person may be further classified as one of the following:
 - customer
 - employee
- Specialization is depicted by a *triangle* component labeled **ISA**, as Fig. 2.10 shows.
- The label ISA stands for “is a” for example, that a customer “is a” person.
- The ISA relationship may also be referred to as a **super class-subclass** relationship.

2.8.8.2 Generalization

- Generalization is a simple inversion of specialization.
- Generalization is the process of defining a more general entity type from a set of more specialized entity types.
- **Generalization** is a bottom-up approach.
- Generalization results in the identification of a generalized super class from the original subclasses.

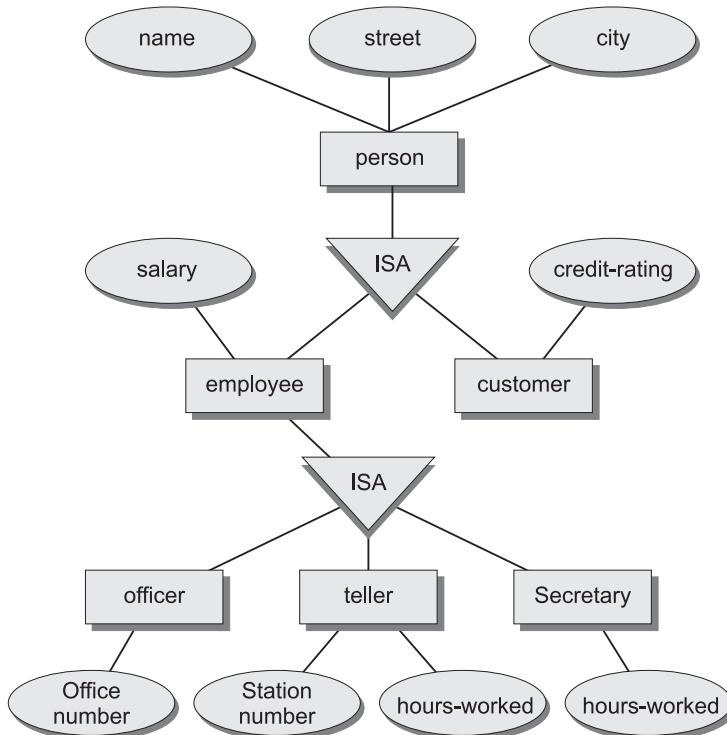


Fig. 2.10. Specialization and Generalization.

- *Person* is the higher-level entity set.
- *Customer* and *employee* are lower-level entity sets.
- The *person* entity set is the super class of the *customer* and *employee* subclasses.

2.8.8.2.1 Constraints on Generalizations

1. One type of constraint determines which entities can be members of a lower-level entity set. Such membership may be one of the following:
 - **Condition-defined.** In condition-defined lower-level entity sets, membership is evaluated on the basis of whether or not an entity satisfies an explicit condition.
 - **User-defined.** User defined constraints are defined by user.
2. A second type of constraint relates to whether or not entities may belong to more than one lower-level entity set within a single generalization. The lower-level entity sets may be one of the following:
 - **Disjoint.** A *disjointness constraint* requires that an entity belong to no more than one lower-level entity set.
 - **Overlapping.** Same entity may belong to more than one lower-level entity set within a single generalization.
3. A final constraint, the **completeness constraint** specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets .This constraint may be one of the following:
 - **Total generalization or specialization.** Each higher-level entity must belong to a lower-level entity set.
 - **Partial generalization or specialization.** Some higher-level entities may not belong to any lower-level entity set.

2.8.8.2.2 Attribute Inheritance

- A property of the higher- and lower-level entities created by specialization and generalization is **attribute inheritance**.
- The attributes of the higher-level entity sets are said to be **inherited** by the lower-level entity sets.

2.8.8.3 Aggregation

- Aggregation is a process when relation between two entities is treated as a single entity. Here the relation between Center and Course, is acting as an entity in relation with visitor.

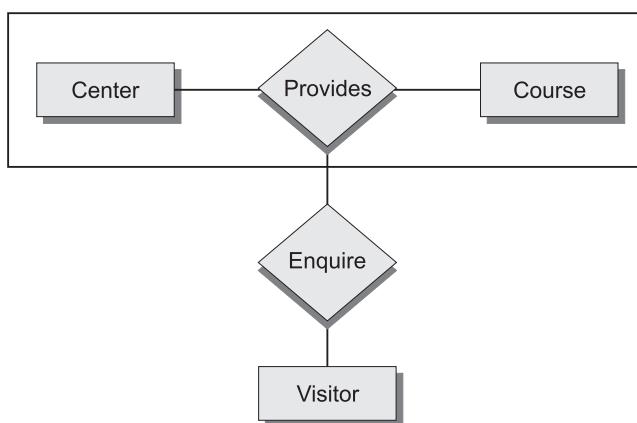


Fig. 2.11. E-R diagram with aggregation.

2.8.9 Reduction of an E-R Schema to Tables

- We can represent an E-R database by a collection of tables.
- For each entity set and for each relationship set in the database, there is a unique table.
- Each table has multiple columns, each of which has a unique name.
- Both the E-R model and the relational-database model are abstract, logical representations of real-world enterprises.
- Two models employ similar design principles; we can convert an E-R design into a relational design.

2.8.9.1 Tabular Representation of Strong Entity Sets

- Let E be a strong entity set with attributes a_1, a_2, \dots, a_n . We represent this entity by a table called E with n distinct columns, each of which corresponds to one of the attributes of E .
- Each row in this table corresponds to one entity of the entity set E .
- As an illustration, consider the entity set $loan$ of the E-R diagram in Figure This entity set has two attributes: $loan-number$ and $amount$.

The loan table

<i>Loan_number</i>	<i>Amount</i>
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

2.8.9.2 Tabular Representation of Relationship Sets

- Consider the relationship set *borrower* in the E-R diagram of Figure. This relationship set involves the following two entity sets:
 - customer*, with the primary key *customer-id*
 - loan*, with the primary key *loan-number*
- Since the relationship set has no attributes, the *borrower* table has two columns, labeled *customer-id* and *loan-number*, as shown in Figure

The *borrower* table

<i>Customer_id</i>	<i>Loan_number</i>
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

2.8.9.3 Tabular Representation of Weak Entity Sets

- Consider the entity set *payment*. This entity set has three attributes: *payment-number*, *payment-date*, and *payment-amount*.
- The primary key of the *loan* entity set, on which *payment* depends, is *loan-number*.
- Thus, we represent *payment* by a table with four columns labeled *loan-number*, *payment-number*, *payment-date*, and *payment-amount*, as in Figure

The *payment* table

<i>Loan_number</i>	<i>Payment_number</i>	<i>Payment_date</i>	<i>Payment_amount</i>
L-11	53	7 June 2001	125
L-14	69	28 May 2001	500
L-15	22	23 May 2001	300
L-16	58	18 June 2001	135
L-17	5	10 May 2001	50
L-17	6	7 June 2001	50
L-17	7	17 June 2001	100
L-23	11	17 May 2001	75
L-93	103	3 June 2001	900
L-93	104	13 June 2001	200

2.8.9.4 Redundancy of Tables

- In general, the table for the relationship set linking a weak entity set to its corresponding strong entity set is redundant and does not need to be present in a tabular representation of an E-R diagram.

Composite Attributes

- We handle composite attributes by creating a separate attribute for each of the component attributes; we do not create a separate column for the composite attribute itself.

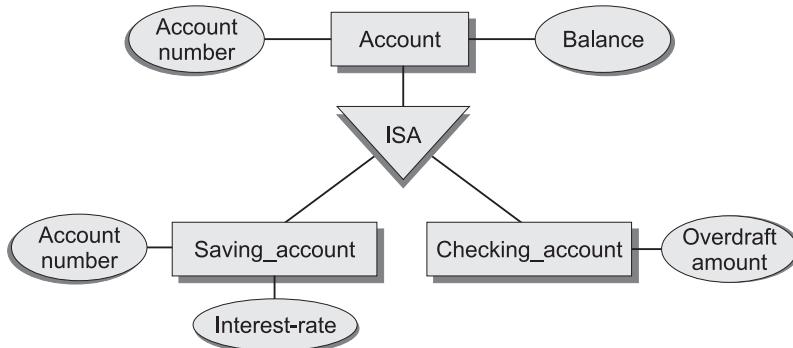
Multi-valued Attributes

- Consider an entity set *employee* with multi-valued attribute *dependent-name*. For this multi-valued attribute, create a separate table as *dependent-info* with attributes *dependent-name* and *employee-id*, which is the primary key of *employee* entity set.

2.8.9.5 Tabular Representation of Generalization

There are two different methods for transforming an E-R diagram that includes generalization to a tabular form.

Consider the generalization shown in fig. where account is a higher-level entity, while lower-level entity sets—that is, *savings-account* and *checking-account*.



- Create a table for the higher-level entity set.
 - For each lower-level entity set, create a separate table
 - Thus, for the E-R diagram of Figure we have three tables:
 - account*, with attributes *account-number* and *balance*
 - savings-account*, with attributes *account-number* and *interest-rate*
 - checking-account*, with attributes *account-number* and *overdraft-amount*
- An alternative representation is possible,
 - If the generalization is disjoint and complete

- Here, do not create a table for the higher-level entity set.
- Then, for the E-R diagram of Figure, we have two tables.
 - *savings-account*, with attributes *account-number*, *balance*, and *interest-rate*
 - *checking-account*, with attributes *account-number*, *balance*, and *overdraftamount*

2.8.9.6 Tabular Representation of Aggregation

- Transforming an E-R diagram containing aggregation to a tabular form is straightforward.
- The table for the relationship set *manages* between the aggregation of *works-on* and the entity set *manager* includes a column for each attribute in the primary keys of the entity set *manager* and the relationship set *works-on*.

REVIEW QUESTIONS

1. Define data model. List out different data models and explain the same.
2. State and explain the E. F. Codd's Laws for a Fully Functional Relational Database Management System.
3. Compare the different data models.
4. Explain the basic components of relational model.
5. Define ER model and describe the notations used in ER model.
6. Explain the different types of attributes with suitable examples.
7. Define keys. List out the different types of keys and explain the same.
8. Distinguish between weak and strong entity.
9. Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.
10. Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.
11. A university registrar's office maintains data about the following entities:
 - (a) courses, including number, title, credits, syllabus, and prerequisites;
 - (b) course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom;
 - (c) students, including student-id, name, and program; and
 - (d) instructors, including identification number, name, department, and title. Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled. Construct an E-R diagram for the registrar's office. Document all assumptions that you make about the mapping constraints.

12. Consider a database used to record the marks that students get in different exams of different course offerings.
 - (a) Construct an E-R diagram that models exams as entities, and uses a ternary relationship, for the above database.
 - (b) Construct an alternative E-R diagram that uses only a binary relationship between *students* and *course-offerings*. Make sure that only one relationship exists between a particular student and course-offering pair.
13. Consider a university database for the scheduling of classrooms for final exams. This database could be modeled as the single entity set *exam*, with attributes *course-name*, *section-number*, *room-number*, and *time*. Alternatively, one or more additional entity sets could be defined, along with relationship sets to replace some of the attributes of the *exam* entity set, as
 - (a) *course* with attributes *name*, *department*, and *c-number*
 - (b) *section* with attributes *s-number* and *enrollment*, and dependent as a weak entity set on *course*
 - (c) *room* with attributes *r-number*, *capacity*, and *building*



Chapter 3

The Relational Algebra

The relational algebra is a *procedural* query language.

It consists of a set of operations that take one or two relations as input and produce a new relation as their result.

Formal Definition

A basic expression in the relational algebra consists of either one of the following:

- A relation in the database
- A constant relation

Let E_1 and E_2 be relational-algebra expressions; the following are all relational-algebra expressions:

$$E_1 \cup E_2$$

$$E_1 - E_2$$

$$E_1 \times E_2$$

$$\sigma_p(E_1), P \text{ is a predicate on attributes in } E_1$$

$$\Pi_s(E_1), S \text{ is a list consisting of some of the attributes in } E_1$$

$$\rho_x(E_1), x \text{ is the new name for the result of } E_1$$

Operations can be divided into

Basic Operations or Fundamental Operations: Select, Project, Union, rename, set difference and Cartesian product.

Additional Operations: operations that can be expressed in terms of basic operations are called as additional operation. Set intersection, Natural join Division and Assignment.

Extended Operations: Generalized projection, Aggregate operations and Outer join.

3.1 BASIC OPERATIONS

- The select, project, and rename operations are called ***unary operations***, because they operate on one relation.

- The other three operations operate on pairs of relations and are called as ***binary operations***.
 - Select
 - Project
 - Union
 - Rename
 - Set difference
 - Cartesian product.

1. Select operation (σ)

The select operation selects tuples that satisfy a given predicate.

Syntax:

$$\sigma_{\text{<select condition>}}(R)$$

- symbol σ is used to denote the select operator
- <Selection condition> is an expression specified on the attributes of the relation R.

Example: Consider following Book relation.

Book_Id	Title	Author	Publisher	Year	Price
B001	DBMS	Korth	Narosa	2000	250
B002	Compiler	Ulman		2004	350
B003	OOMB	Rumbaugh		2003	450
B004	PPL	Sabista		2000	500

Following are the some examples of the select operation.

Example 1: Display books published in the 2000.

Query 1: $\sigma_{\text{year}=2000}(\text{Book})$

The output of query 1 is shown below.

Book_Id	Title	Author	Publisher	Year	Price
B001	DBMS	Korth	Narosa	2000	250
B004	PPL	Sabista		2000	500

Example 2: Display all books having price greater than 300.

Query 2: $\sigma_{\text{price}>300}(\text{Book})$

The output of query 2 is shown below.

Book_Id	Title	Author	Publisher	Year	Price
B002	Compiler	Ulman		2004	350
B003	OOMB	Rumbaugh		2003	450
B004	PPL	Sabista		2000	500

Example 3: Select the tuples for all books whose publishing year is 2000 or price is greater than 300.

Query 3: $\sigma_{(year=2000) \text{ OR } (price>300)}(\text{Book})$

The output of query 3 is shown below.

Book_Id	Title	Author	Publisher	Year	Price
B001	DBMS	Korth	Narosa	2000	250
B002	Compiler	Ulman		2004	350
B003	OODM	Rumbaugh		2003	450
B004	PPL	Sabista		2000	500

Example 4: Select the tuples for all books whose publishing year is 2000 and price is greater than 300.

Query 3: $\sigma_{(year=2000) \text{ AND } (price>300)}(\text{Book})$

The output of query 4 is shown below.

Book_Id	Title	Author	Publisher	Year	Price
B004	PPL	Sabista		2000	500

2. Project operation (Π)

The project operation selects certain columns from a table while discarding others. It removes any duplicate tuples from the result relation.

Syntax:

$\Pi_{<\text{attributelist}>}(\text{R})$

- The symbol Π (pi) is used to denote the project operation
- Attribute list of relations from the attributes of the relation R.

Example: The following are the examples of project operation on Book relation.

Example 1: Display all titles with author name.

Query 1: $\Pi_{\text{Title, Author}}(\text{Book})$

The output of query 1 is shown below.

Title	Author
DBMS	Korth
Compiler	Ulman
OODM	Rumbaugh
PPL	Sabista

Example 2: Display all book titles with authors and price.

Query 2: $\sigma_{\text{Title, Author, Price}}(\text{Book})$

The output of query 2 is shown as follows:

Title	Author	Price
DBMS	Korth	250
Compiler	Ulman	350
OODM	Rumbaugh	450
PPL	Sabista	500

Composition of select and project operations

The relational operations select and project can be combined to form a complicated query.

Example: Display the titles of books having price greater than 300.

Query: $\Pi_{\text{Title}, (\sigma_{\text{price} > 300})}(\text{Book})$

The output of query 1 is shown below.

Title
Compiler
OODM
PPL

3. Rename operation (ρ)

In relational algebra, you can rename either the relation or the attributes or both. The general rename operation can take any of the following forms:

Syntax:

- I. $\rho_s(\text{new attribute names})(R)$
- II. $\rho_s(R)$
- III. $\rho_{(\text{new attribute names})}(R)$

- The symbol ‘’ (rho) is used to denote the RENAME operator.
- ‘S’ is the new relation
- ‘R’ is original relation.

Example 1: Renames both the relation and its attributes, the second renames the relation only and the third renames as follows.

$*\rho_{\text{Temp(Bname, Aname, Pyear, Bprice)}}(\text{ Book })$

Example 2: Only the relation name is renamed.

$*\rho_{\text{Temp}}(\text{ Book })$

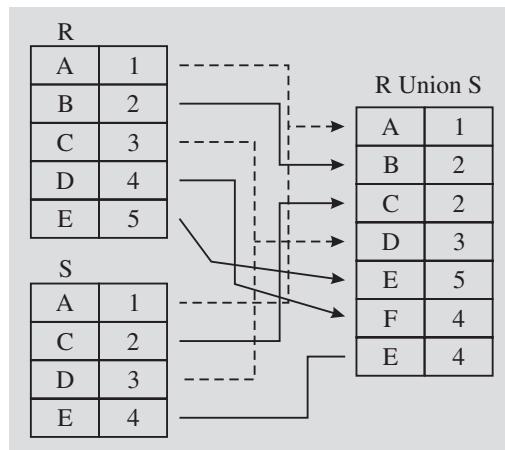
Example 3: Only the attribute names are renamed

$*\rho_{(\text{Bname, Aname, Pyear, Bprice})}(\text{ Book })$

4. Union operation (U)

The two relations/tables must contain the same number of columns (have the same degree).

Syntax: Relation1 U Relation2



Example: Consider the two relations:

The depositor relation

<i>Customer_name</i>	<i>Account_number</i>
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

The borrower relation

<i>Customer_name</i>	<i>Loan_number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

To find the names of all customers with a loan in the bank:

$$\Pi_{\text{customer-name}}(\text{borrower})$$

To find the names of all customers with an account in the bank:

$$\Pi_{\text{customer-name}}(\text{depositor})$$

Union of these two sets; that is, To find customer names that appear in either or both of the two relations.

$$\Pi_{\text{customer-name}}(\text{borrower}) \cup \Pi_{\text{customer-name}}(\text{depositor})$$

The result relation for this query:

<i>Customer_name</i>
Adams
Curry
Hayes

Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

5. Set difference operation (-)

To find tuples that is in one relation but is not in another.

Syntax: Relation1 – Relation 2

R	
A	1
B	2
C	3
D	4
E	5

R Difference S	
B	2
F	4
E	5

S	
A	1
C	2
D	3
E	4

S Difference R	
C	2
E	4

Example: find all customers of the bank who have an account but not a loan.

$$\Pi_{customer_name (depositor)} - customer_name (borrower)$$

The result relation for this query

Customer_name
Johnson
Lindsay
Turner

6. Cartesian-product operation (X)

- Cartesian product is also known as CROSS PRODUCT or CROSS JOINS.
- Cartesian product allows us to combine information from any 2 relation.

Syntax: Relation1 x Relation 2

R		R Cross S			
A	1	A	1		
B	2	A	2		
C	3	D	3		
D	4	E	4		
E	5	A	1		
S		B	2		
A	1	B	2		
C	2	B	3		
D	3	B	4		
E	4	D	1		
		D	2		
		D	3		
		D	4		
		E	1		
		E	2		
		E	3		
		E	4		

Example: Consider following two relations publisher_info and Book_info.

Publisher_Info	
Publisher_code	Name
P0001	McGraw_Hill
P0002	PHI
P0003	Pearson

Book_Info	
Book_ID	Title
B0001	DBMS
B0002	Compiler

The Cartesian product of Publisher_Info and Book_Info is given in Fig.

Publisher_Info X Book_Info			
Publisher_code	Name	Book_ID	Title
P0001	McGraw_Hill	B0001	DBMS
P0002	PHI	B0001	DBMS
P0003	Pearson	B0001	DBMS
P0001	McGraw_Hill	B0002	Compiler
P0002	PHI	B0002	Compiler
P0003	Pearson	B0002	Compiler

3.2 ADDITIONAL OPERATIONS

- (a) Set intersection
- (b) Natural join
- (c) Division
- (d) Assignment

1. Set Intersection Operation (\cap)

- The result of intersection operation is a relation that includes all tuples that are in both Relation1 and Relation2.
- The intersection operation is denoted by depositor \cap borrower.

Syntax: Relation1 \cap Relation 2

R	
A	1
B	2
C	3
D	4
E	5

R Intersection S	
A	1
D	3

S	
A	1
C	2
D	3
E	4

Example:

$$\Pi_{customer-name}(borrower) \cap \Pi_{customer-name}(depositor)$$

The result relation for this query:

Customer_name
Hayes
Jones
Smith

2. Natural Join (\bowtie)

The natural join operation performs a selection on those attributes that appear in both relation schemes and finally removes duplicate attributes.

Syntax: Relation1 \bowtie Relation 2

Example: consider the 2 relations

<i>Employee</i>		<i>Salary</i>	
<i>Emp_code</i>	<i>Emp_name</i>	<i>Emp_code</i>	<i>Salary</i>
E0001	Hari	E0001	2000
E0002	Om	E0002	5000
E0003	Smith	E0003	7000
E0004	Jay	E0004	10000

Query: $\Pi_{\text{emp_name}, \text{salary}}(\text{employee} \bowtie \text{salary})$

The output of query is:

<i>Emp_name</i>	<i>Salary</i>
Hari	2000
Om	5000
Smith	7000
Jay	10000

Inner Join (\bowtie)

Inner Join returns the matching rows from the tables that are being jointed.

Example: Consider the two relations

The Employee relation

<i>Employee_name</i>	<i>Street</i>	<i>City</i>
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

ft-works relations

<i>Employee_name</i>	<i>Branch_name</i>	<i>Salary</i>
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

The Employee \bowtie ft-works relations

<i>Employee_name</i>	<i>Street</i>	<i>City</i>	<i>Branch_name</i>	<i>Salary</i>
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500

Outer Join

The **outer-join** operation is an extension of the join operation to deal with missing information. *Outer-join* operation avoid loss of information.

Outer Joins are classified into three types namely:

- Left Outer Join
- Right Outer Join
- Full Outer Join

1. Left Outer Join ($\square \bowtie \square$)

The **left outer join** ($\square \bowtie \square$) takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation, and adds them to the result of the natural join.

Example: The Employee $\square \bowtie \square$ ft-works relations

Result:

Employee_name	Street	City	Branch_name	Salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	Null	Null

2. Right Outer Join ($\bowtie \square \square$)

The **right outer join** ($\bowtie \square \square$) is symmetric with the left outer join. It pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join.

Example: The Employee $\bowtie \square \square$ ft-works relations

Result:

Employee_name	Street	City	Branch_name	Salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	Null	Null	Redmond	5300

3. Full Outer Join ($\square \bowtie \square \square$)

The **full outer join** ($\square \bowtie \square \square$) does both of those operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join.

Example: employee $\setminus \text{ft-works}$

Result:

Employee_name	Street	City	Branch_name	Salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	Null	Null
Gates	Null	Null	Redmond	5300

3. Division operation (\div)

- Produce the tuples in one relation, r, that match all tuples in another relation, s
- Division Operation is suited to queries that include the phrase ‘for all’.

Syntax: Relation1 \div Relation 2

- r (A1, ...An, B1, ...Bm)
- s (B1 ...Bm)
- r/s, with attributes A1, ...An, is the set of all tuples $\langle a \rangle$ such that for every tuple $\langle b \rangle$ in s, $\langle a, b \rangle$ is in r

Can be expressed in terms of projection, set difference, and cross-product.

List the Ids of students who have passed all courses that were taught in spring 2000.

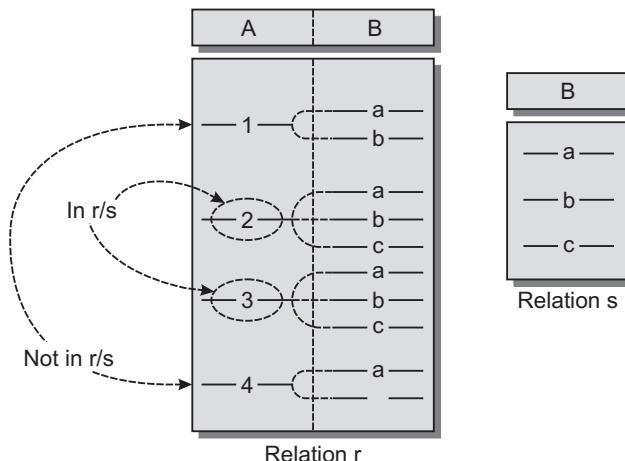
Numerator:

- StudId and CrsCode for every course passed by every student:
- $\pi_{\text{StudId}, \text{CrsCode}} (\sigma_{\text{Grade} < \text{'F'}} (\text{Transcript}))$

Denominator:

- CrsCode of all courses taught in spring 2000
- $\pi_{\text{CrsCode}} (\sigma_{\text{Semester} = \text{'S2000'}} (\text{Teaching}))$

Result is numerator/denominator



Example: Consider three relations:

Account Relation

<i>Account_number</i>	<i>Branch_name</i>	<i>Balance</i>
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

Branch Relation

<i>Branch_name</i>	<i>Branch_city</i>	<i>Assets</i>
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Depositor Relation

<i>Customer_name</i>	<i>Account_number</i>
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Suppose that we wish to find all customers who have an account at *all* the branches located in Brooklyn.

Step 1: We can obtain all branches in Brooklyn by the expression

$$r1 = \Pi_{branch-name} (\sigma_{branch-city = "Brooklyn"} (branch))$$

The result relation for this expression is shown in figure.

<i>Branch_name</i>
Brighton
Downtown

Step 2: We can find all (*customer-name*, *branch-name*) pairs for which the customer has an account at a branch by writing

$$r2 = \Pi_{customer-name, branch-name} (depositor \bowtie account)$$

Figure shows the result relation for this expression.

<i>Customer_name</i>	<i>Branch_name</i>
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

Now, we need to find customers who appear in r_2 with *every* branch name in r_1 . The operation that provides exactly those customers is the divide operation.

Thus, the query is

$$\begin{aligned} & \Pi_{\text{customer-name}, \text{branch-name}} (\text{depositor} \bowtie \text{account}) \\ & \div \Pi_{\text{branch-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn}} (\text{branch})) \end{aligned}$$

The result of this expression is a relation that has the schema (*customer-name*) and that contains the tuple (Johnson).

4. The Assignment Operation (\leftarrow)

The **assignment** operation works like assignment in a programming language.

Example:

$$\begin{aligned} \text{temp1} &\leftarrow \Pi_{R-S}(r) \\ \text{temp1} &\leftarrow \Pi_{R-S}((\text{temp1} \times s) - \Pi_{R-S,S}(r)) \\ \text{result} &\leftarrow \text{temp1} - \text{temp2} \end{aligned}$$

- Result of the expression to the right of the \leftarrow is assigned to the relation variable on the left of the \leftarrow .
- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

3.3 EXTENDED RELATIONAL-ALGEBRA OPERATIONS

1. Generalized projection
2. Aggregate operations
3. Outer join.

1. Generalized Projection

- The **generalized-projection** operation extends the projection operation by allowing arithmetic functions to be used in the projection list.
- The generalized projection operation has the form

$$\Pi_{F_1, F_2, \dots, F_n}(E)$$

Where E is any relational-algebra expression, and each of F_1, F_2, \dots, F_n is an arithmetic expression involving constants and attributes in the schema of E .

Example: Suppose we have a relation *credit-info*, as in Figure

<i>Customer_name</i>	<i>Limit</i>	<i>Credit_Balance</i>
Curry	2000	1750
Hayes	1500	1500
Jones	6000	700
Smith	2000	400

Query: $\Pi_{customer-name, (limit - credit-balance)} \text{as } credit-available \text{ (credit-info)}$

Result of this query:

<i>Customer_name</i>	<i>Credit_available</i>
Curry	250
Jones	5300
Smith	1600
Hayes	0

2. Aggregate Functions

Aggregate functions take a collection of values and return a single value as a result.

Few Aggregate Functions are,

1. Avg
2. Min and Max
3. Sum
4. Count

1. Avg: The aggregate function **avg** returns the average of the values.

Example: Use the *pt-works* relation in Figure

<i>Employee_name</i>	<i>Brance_name</i>	<i>Salary</i>
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Rao	Austin	1500
Sato	Austin	1600

Suppose that we want to find out the average of salaries.

The relational-algebra expression for this query is:

$$G_{avg(salary)}(pt\text{-}works)$$

The symbol G is the letter G in calligraphic font; read it as “calligraphic G.”

Result:

Salary
2062.5

2. Min and Max

Min: Return the minimum values in a collection

Max: Return the Maximum values in a collection

Example: branch-name $G_{\text{sum}(\text{salary}) \text{ as } \text{sum-salary}, \text{max}(\text{salary}) \text{ as } \text{max-salary}}$ (*pt-works*)

Result:

Employee_name	Sum_Salary	Max_Salary
Austin	3100	1600
Downtown	5300	2500
Perryridge	8100	5300

3. Sum: The aggregate function sumreturns the total of the values.

Example: Suppose that we want to find out the total sum of salaries.

branch-name $G_{\text{sum}(\text{salary})}$ (*pt-works*)

The symbol G is the letter G in calligraphic font; read it as “calligraphic G.”

Result:

Salary
16500

4. Count: Returns the number of the elements in the collection,

Example: $G_{\text{count-distinct}(\text{branch-name})}$ (*pt-works*)

Result: The result of this query is a single row containing the value 3.

Operation	My HTML	Symbol
Projection	PROJECT	π
Selection	SELECT	σ
Renaming	RENAME	ρ
Union	UNION	\cup
Intersection	INTERSECTION	\cap
Assignment	\leftarrow	\leftarrow

Operation	My HTML	Symbol
Cartesian Product	X	\times
Join	JOIN	\bowtie
Left outer join	LEFT OUTER JOIN	\bowtie^L
Right outer join	RIGHT OUTER JOIN	\bowtie^R
Full outer join	FULL OUTER JOIN	\bowtie^F
Semijoin	SEMIJOIN	\bowtie^S

3.4 MODIFICATION OF THE DATABASE

- The content of the database may be modified using the following operations:
 - Deletion
 - Insertion
 - Updating
- All these operations are expressed using the assignment operator.

1. Deletion

- A delete request is expressed similar to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes
- A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.

Deletion Examples:

- Delete all account records in the Perryridge branch

$$\text{account} \leftarrow \text{account} - \sigma_{\text{branch_name} = \text{"Perryridge"} }(\text{account})$$

- Delete all loan records with amount in the range of 0 to 50

$$\text{loan} \leftarrow \text{loan} - \sigma_{\text{amount} \geq 0 \text{ and } \text{amount} \leq 50 }(\text{loan})$$

- Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{\text{branch_city} = \text{"Needham"} }(\text{account branch})$$

$$r_2 \leftarrow \Pi_{\text{account_number}, \text{branch_name}, \text{balance}}(r_1)$$

$$r_3 \leftarrow \Pi_{\text{customer_name}, \text{account_number}}(r_2 \text{ depositor})$$

$$\text{account} \leftarrow \text{account} - r_2$$

$$\text{depositor} \leftarrow \text{depositor} - r_3$$

2. Insertion

- To insert data into a relation, we either:
 - specify a tuple to be inserted or
 - write a query whose result is a set of tuples to be inserted

In relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$

where r is a relation and E is a relational algebra expression.

- The insertion of a single tuple is expressed by letting E is a constant relation containing one tuple.

Insertion Examples:

- Insert information in the database specifying that Smith has \$1200 in account A-973 at the Perryridge branch.

$$\begin{aligned} account &\leftarrow account \cup \{("A-973", "Perryridge", 1200)\} \\ depositor &\leftarrow depositor \cup \{("Smith", "A-973")\} \end{aligned}$$

- Provide as a gift for all loan customers in the Perryridge branch, a \$200 savings account.
Let the loan number serve as the account number for the new savings account.

$$\begin{aligned} r_1 &\leftarrow (\pi_{branch_name = "Perryridge"}(borrower \text{ } loan)) \\ account &\leftarrow account \cup \prod_{loan_number, branch_name, 200}(r_1) \\ depositor &\leftarrow depositor \cup \prod_{customer_name, loan_number}(r_1) \end{aligned}$$

3. Updating

- A mechanism to change a value in a tuple without changing *all* values in the tuple
- Use the generalized projection operator to do this task
- Each F_i is either
 - the i^{th} attribute of r , if the i^{th} attribute is not updated, or,
 - if the attribute is to be updated F_i is an expression, involving only constants and the attributes of r , which gives the new value for the attribute

Update Examples:

- Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \prod_{account_number, branch_name, balance * 1.05}(account)$$

- Pay all accounts with balances over \$10,000 6 percent interest and pay all others 5 percent

$$\begin{aligned} account &\leftarrow \prod_{account_number, branch_name, balance * 1.06(\sigma_{BAL > 10000}(account))} \\ &\quad \cup \prod_{account_number, branch_name, balance * 1.05(\sigma_{BAL \leq 10000}(account))} \end{aligned}$$

3.5 RELATIONAL CALCULUS

- Relational Calculus is a formal query language where we can write one declarative expression to specify a retrieval request and hence there is no description of how to retrieve it.
- A calculus expression specifies what is to be retrieved rather than how to retrieve it.
- Relational Calculus is considered to be non procedural language.
- Relational Calculus can be divided into
 1. Tuple Relational Calculus
 2. Domain Relational Calculus

3.5.1 Tuple Relational Calculus

- A nonprocedural query language, where each query is of the form

$$\{t \mid P(t)\}$$

- It is the set of all tuples t such that predicate P is true for t
- t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A
- $t \in r$ denotes that tuple t is in relation r
- P is a *formula* similar to that of the predicate calculus

Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g., $<$, \leq , $=$, \neq , $>$, \geq)
3. Set of connectives: and (\wedge), or (\vee), not (\neg)
4. Implication (\Rightarrow): $x \Rightarrow y$, if x if true, then y is true

$$x \Rightarrow y \equiv \neg x \vee y$$

5. Set of quantifiers:
 - $\exists t \in r (Q(t)) \equiv$ “there exists” a tuple in t in relation r such that predicate $Q(t)$ is true
 - $\forall t \in r (Q(t)) \equiv Q$ is true “for all” tuples t in relation r

Banking Example

1. $branch(branch_name, branch_city, assets)$
2. $customer(customer_name, customer_street, customer_city)$
3. $account(account_number, branch_name, balance)$
4. $loan(loan_number, branch_name, amount)$
5. $depositor(customer_name, account_number)$
6. $borrower(customer_name, loan_number)$

Example Queries

1. Find the $loan_number$, $branch_name$, and $amount$ for loans of over \$1200

$$\{t \mid t \in loan \wedge t[amount] > 1200\}$$

2. Find the loan number for each loan of an amount greater than \$1200

$$\{t \mid \exists s \in loan (t[loan_number] = s[loan_number] \wedge s[amount] > 1200)\}$$

Notice that a relation on schema $[loan_number]$ is implicitly defined by the query

3. Find the names of all customers having a loan, an account, or both at the bank

$$\begin{aligned} &\{t \mid \exists s \in borrower (t[customer_name] = s[customer_name]) \\ &\quad \vee \exists u \in depositor (t[customer_name] = u[customer_name])\} \end{aligned}$$

4. Find the names of all customers who have a loan and an account the bank

$$\begin{aligned} &\{t \mid \exists s \in borrower (t[customer_name] = s[customer_name]) \\ &\quad \wedge \exists u \in depositor (t[customer_name] = u[customer_name])\} \end{aligned}$$

5. Find the names of all customers having a loan at the Perryridge branch

$$\begin{aligned} &\{t \mid \exists s \in borrower (t[customer_name] = s[customer_name]) \\ &\quad \wedge \exists u \in loan (u[branch_name] = "Perryridge") \\ &\quad \wedge u[loan_number] = s[loan_number]\})\} \end{aligned}$$

6. Find the names of all customers who have a loan at the Perryridge branch, but no account at any branch of the bank

$$\{t \mid \exists s \in \text{borrower} (t[\text{customer_name}] = s[\text{customer_name}]) \\ \wedge \exists u \in \text{loan} (u[\text{branch_name}] = \text{"Perryridge"}) \\ \wedge u[\text{loan_number}] = s[\text{loan_number}]) \\ \wedge \text{not } \exists v \in \text{depositor} (v[\text{customer_name}] = t[\text{customer_name}])\}$$

7. Find the names of all customers having a loan from the Perryridge branch, and the cities in which they live

$$\{t \mid \exists s \in \text{loan} (s[\text{branch_name}] = \text{"Perryridge"}) \\ \wedge \exists u \in \text{borrower} (u[\text{loan_number}] = s[\text{loan_number}]) \\ \wedge t[\text{customer_name}] = u[\text{customer_name}]) \\ \wedge \exists v \in \text{customer} (u[\text{customer_name}] = v[\text{customer_name}]) \\ \wedge t[\text{customer_city}] = v[\text{customer_city}])\}$$

8. Find the names of all customers who have an account at all branches located in Brooklyn:

$$\{t \mid \exists r \in \text{customer} (t[\text{customer_name}] = r[\text{customer_name}]) \\ \wedge (\forall u \in \text{branch} (u[\text{branch_city}] = \text{"Brooklyn"}) \\ \exists s \in \text{depositor} (t[\text{customer_name}] = s[\text{customer_name}]) \\ \wedge w \in \text{account} (w[\text{account_number}] = s[\text{account_number}]) \\ \wedge (w[\text{branch_name}] = u[\text{branch_name}]))\}$$

Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example, $\{t \mid \neg t r\}$ results in an infinite relation if the domain of any attribute of relation r is infinite.
- To guard against the problem, we restrict the set of allowable expressions to safe expressions.
- An expression $\{t \mid P(t)\}$ in the tuple relational calculus is *safe* if every component of t appears in one of the relations, tuples, or constants that appear in P .

Note: this is more than just a syntax condition.

- E.g. $\{t \mid t[A] = 5 \vee \text{true}\}$ is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in P .

3.5.2 Domain Relational Calculus

- A nonprocedural query language equivalent in power to the tuple relational calculus
- Each query is an expression of the form:

$$\{<x_1, x_2, \dots, x_n> \mid P(x_1, x_2, \dots, x_n)\}$$

- x_1, x_2, \dots, x_n represent domain variables
- P represents a formula similar to that of the predicate calculus

Example Queries

1. Find the *loan_number*, *branch_name*, and *amount* for loans of over \$1200

$$\{<l, b, a> | <l, b, a> \in \text{loan} \wedge a > 1200\}$$

2. Find the names of all customers who have a loan of over \$1200

$$\{<c> | \exists l, b, a (<c, l> \in \text{borrower} \wedge <l, b, a> \in \text{loan} \wedge a > 1200)\}$$

3. Find the names of all customers who have a loan from the Perryridge branch and the loan amount:

$$\begin{aligned} \{<c, a> | \exists l (<c, l> \in \text{borrower} \wedge \exists b (<l, b, a> \in \text{loan} \wedge b = \text{"Perryridge"}))\} \\ \{<c, a> | \exists l (<c, l> \in \text{borrower} \wedge <l, \text{"Perryridge"}, a> \in \text{loan})\} \end{aligned}$$

4. Find the names of all customers having a loan, an account, or both at the Perryridge branch:

$$\begin{aligned} \{<c> | \exists l (<c, l> \in \text{borrower} \wedge \exists b, a (<l, b, a> \in \text{loan} \wedge b = \text{"Perryridge"})) \\ \vee \exists a (<c, a> \in \text{depositor} \wedge \exists b, n (<a, b, n> \in \text{account} \wedge b = \text{"Perryridge"}))\} \end{aligned}$$

5. Find the names of all customers who have an account at all branches located in Brooklyn:

$$\begin{aligned} \{<c> | \exists s, n (<c, s, n> \in \text{customer}) \wedge \forall x, y, z (<x, y, z> \in \text{branch} \wedge y = \text{"Brooklyn"}) \\ \Rightarrow \exists a, b (<x, y, z> \in \text{account} \wedge <c, a> \in \text{depositor})\} \end{aligned}$$

Safety of Expressions

The expression: $\{<x_1, x_2, \dots, x_n> | P(x_1, x_2, \dots, x_n)\}$ is safe if all of the following hold:

- All values that appear in tuples of the expression are values from $\text{dom}(P)$ (that is, the values appear either in P or in a tuple of a relation mentioned in P).
- For every “there exists” subformula of the form $\exists x (P_1(x))$, the subformula is true if and only if there is a value of x in $\text{dom}(P_1)$ such that $P_1(x)$ is true.
- For every “for all” subformula of the form $\forall x (P_1(x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from $\text{dom}(P_1)$.

REVIEW QUESTIONS

1. Explain the basic relational algebra operation with suitable examples.
2. Explain the additional and extended relational algebra operation with suitable examples.
3. Explain Tuple Relational Calculus and Domain Relational Calculus
4. Consider the following database

employee (person-name, street, city)

works (person-name, company-name, salary)

company (company-name, city)

manages (person-name, manager-name)

- 4.1 Consider the relational database given above. Give an expression in the relational algebra for each request:
- Modify the database so that Jones now lives in Newtown.
 - Give all employees of First Bank Corporation a 10 percent salary raise.
 - Give all managers in this database a 10 percent salary raise.
 - Give all managers in this database a 10 percent salary raise, unless the salary would be greater than \$100,000. In such cases, give only a 3 percent raise.
 - Delete all tuples in the *works* relation for employees of Small Bank Corporation.
- 4.2 Consider the relational database given above. Give a relational-algebra expression for each of the following queries:
- Find the company with the most employees.
 - Find the company with the smallest payroll.
 - Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.



Chapter 4

SQL-Fundamentals

4.1 INTRODUCTION

“Structured Query Language” is abbreviated as SQL. SQL was designed to be “human-friendly”.

Most of its commands resemble spoken English. This feature of SQL makes it easy to read, understand, and learn. The SQL language may be considered one of the major reasons for the success of relational databases in the commercial world. The Structured Query Language is a relational database language. DBMS is not made by SQL. SQL is a medium used to communicate with the DBMS.

SQL is referred to as nonprocedural database language. To retrieve data from the database using nonprocedural language it is enough to tell what data to be retrieved, rather than how to retrieve it. The DBMS will take care of locating the information in the database. Commercial database management systems allow SQL to be used in two distinct ways.

1. SQL commands can be typed at the command line directly. The DBMS interprets and processes the SQL commands immediately, and the results are displayed. This method of SQL processing is called interactive SQL.
2. The second method is called programmatic SQL. Here, SQL statements are embedded in a host language such as COBOL, FORTRAN, C, etc. SQL needs a host language because SQL is not a really complete computer programming language as such because it has no statements or constructs that allow branch or loop. The host language provides the necessary looping and branching structures and the interface with the user, while SQL provides the statements to communicate with the DBMS.

4.2 ADVANTAGES OF SQL

- SQL is a high level language that provides a greater degree of abstraction than procedural languages.
- SQL enables the end-users and systems personnel to deal with a number of database management systems.

- SQL specifies what is required and not how it should be done.
- SQL is simple and easy to learn.
- SQL handles complex situations easily.
- All SQL operations are performed at a set level.
- Reduced training cost
- Enhanced productivity
- Application portability: Application portability means applications can be moved from machine to machine when each machine uses SQL.
- Application durability: A standard language tends to remain for a long time, hence there will be little pressure to rewrite old applications.
- Reduced dependence on a single vendor

4.3 PARTS OF SQL

The SQL language has several parts:

- **Data-definition language (DDL).** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- **Interactive data-manipulation language (DML).** The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus. It also includes commands to insert tuples into, delete tuples from, and modify tuples in the database.
- **View definition.** The SQL DDL includes commands for defining views.
- **Transaction control.** SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL and dynamic SQL.** Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, COBOL, Pascal, and FORTRAN.
- **Integrity.** The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **Authorization.** The SQL DDL includes commands for specifying access rights to relations and views.

4.4 DOMAIN TYPES IN SQL

In relational model the data are stored in the form of tables. A table is composed of rows and columns. When we create a table we must specify a data type for each of its columns. These data types define the domain of values that each column can take. A number of built-in data types are available. Some of the built-in data types are string data type to store characters, number data type to store numerical value, and date and time data type to store when the event happened.

1. **Char (n):** The CHAR data type specifies a fixed-length character string. The syntax of CHAR data type is:

CHAR (n) – Fixed length character data, “n” characters long.

“n” specifies the character length. If a value shorter than the column length is inserted then the input is padded with space to the column length. If a value greater than column length is inserted then an error is returned.

2. **varchar(n):** Variable length character strings, with user-specified maximum length *n*. The syntax of varchar data type is:

VARCHAR2 (n) – Variable length character of “n” length.

“n” specifies the character length.

3. **int:** Integer (a finite subset of the integers that is machine-dependent).
4. **Smallint:** Small integer (a machine-dependent subset of the integer domain type).
5. **Numeric (p,d):** fixed point number, where “p” is the total number of digits and “q” is the number of digits to the right of decimal point.
6. **Real, double precision:** Floating point and double-precision floating point numbers, with machine-dependent precision.
7. **float (n):** Floating point number, with user-specified precision of at least *n* digits.
8. **Date:** Dates, containing a (4 digit) year, month and date
Example: date ‘2005-7-27’
9. **Time:** Time of day, in hours, minutes and seconds.
Example: time ‘09:00:30’ time ‘09:00:30.75’
10. **Timestamp:** date plus time of day
Example: timestamp ‘2005-7-27 09:00:30.75’
11. **Interval:** period of time
Example: interval ‘1’ day

4.5 TERMINOLOGY

A table is the basic storage unit of DBMS. The table stores the necessary data. The common terminologies used in relational databases are as follows:

1. A **row or tuple** represents all the data required for a particular entity. Each row in a table is identified using primary key.
2. A **column or attribute** represents each column in the table. Each column represents a property or attribute of an entity. The columns can a key attribute or non key attribute.
3. A **field** can be intersection of column and a row. There can be only one value in it

4.6 DATA DEFINITION LANGUAGE (DDL)

A DDL is a language used to define data structures within a database. It is considered to be the subset of Structured Query Language. A Data Definition Language has a pre-defined syntax for describing data.

For example, to create a new table using SQL syntax, the CREATE command is used, followed by parameters for the table name and column definitions. The DDL can also define the name of each column and the associated data type.

Once a table is created, it can be modified using the ALTER command. If the table is no longer needed, the DROP command will delete the table.

Since DDL is a subset of SQL, it does not include all the possible SQL commands. For example, commands such as SELECT and INSERT are considered part of the Data Manipulation Language (DML), while access commands such as CONNECT and EXECUTE are part of the Data Control Language (DML).

The various **DDL commands** are as follows:

- CREATE: to create objects in the database
- DESC: describes the structure of the database
- ALTER: alters the structure of the database
- DROP: removes a table from the database
- TRUNCATE: removes all records from a table, including all spaces allocated for the records are removed
- RENAME: renames an object

The different structures that are created by DDL are Tables, Views, Sequences, Triggers, Indexes, etc.

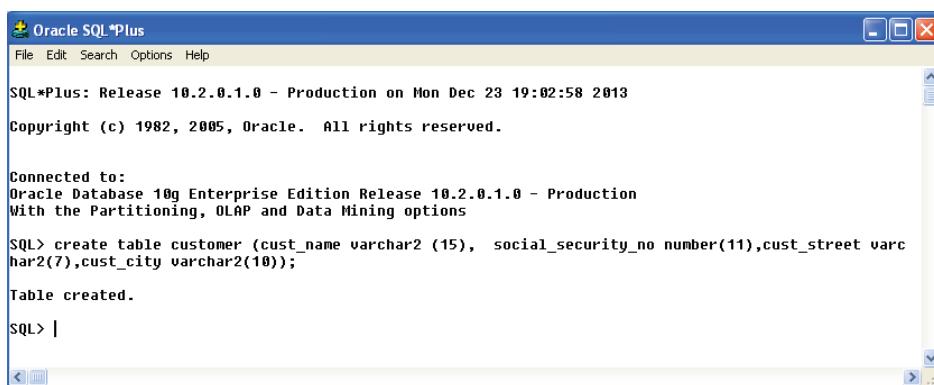
4.6.1 DDL Commands

1. **CREATE:** Command is used for creating tables.

Syntax:

```
create table<table name> (columnname 1 data type (size), Columnname 2 data
type(size),.., columnname n data type(size));
```

Example: create table customer (cust_name varchar2(15), social_security_no number(11), cust_street varchar2(7), cust_city varchar2(10));



The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following SQL command and its execution results:

```
SQL*Plus: Release 10.2.0.1.0 - Production on Mon Dec 23 19:02:58 2013
Copyright (c) 1982, 2005, Oracle. All rights reserved.

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> create table customer (cust_name varchar2 (15), social_security_no number(11),cust_street varc
har2(7),cust_city varchar2(10));
Table created.

SQL> |
```

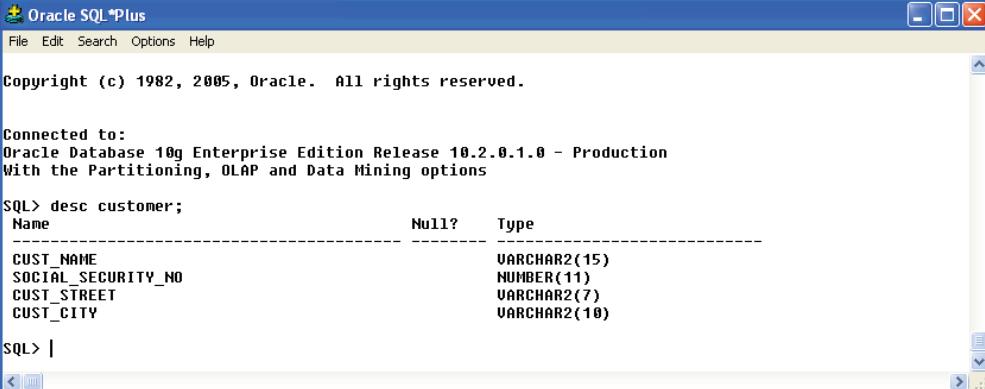
Fig. 4.1. Table creation.

2. **DESC:** Command is used to view the table structure.

Syntax:

```
desc<table name>;
```

Example: desc customer;



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following text:

```

Copyright (c) 1982, 2005, Oracle. All rights reserved.

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> desc customer;
Name          Null?    Type
-----
CUST_NAME           VARCHAR2(15)
SOCIAL_SECURITY_NO  NUMBER(11)
CUST_STREET         VARCHAR2(7)
CUST_CITY           VARCHAR2(10)

SQL> |

```

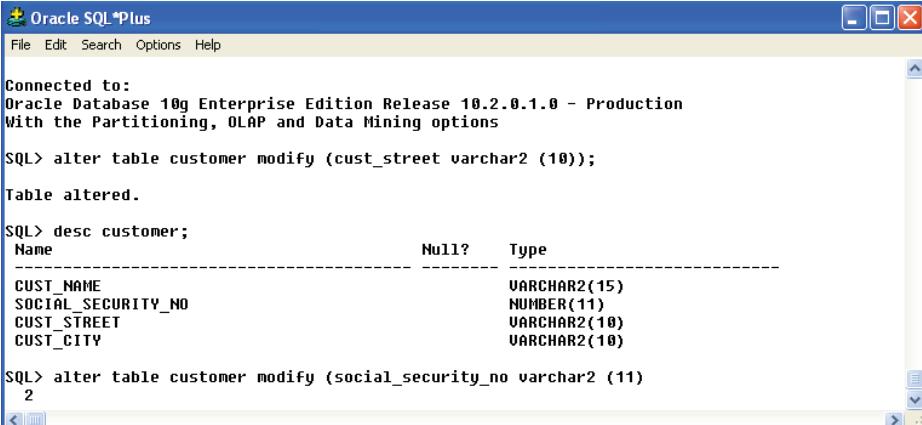
Fig. 4.2. Table description.

3. ALTER: Command is used for modifying table structure.

(i) **Syntax:**

alter table<table name>modify(columnname data type (new size));

Example: alter table customer modify (cust_street varchar2 (10));



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following text:

```

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> alter table customer modify (cust_street varchar2 (10));
Table altered.

SQL> desc customer;
Name          Null?    Type
-----
CUST_NAME           VARCHAR2(15)
SOCIAL_SECURITY_NO  NUMBER(11)
CUST_STREET         VARCHAR2(10)
CUST_CITY           VARCHAR2(10)

SQL> alter table customer modify (social_security_no varchar2 (11)
2

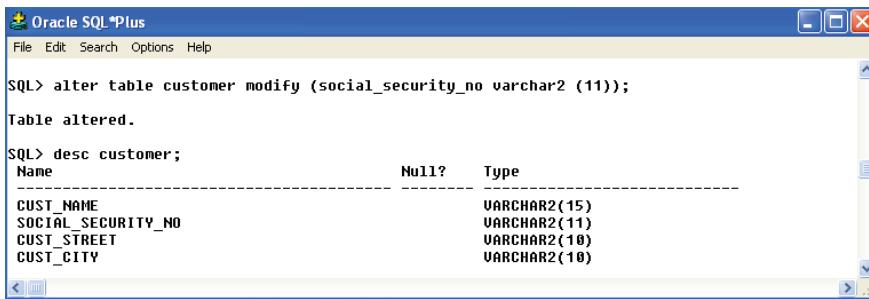
```

Fig. 4.3. Modification of size data type.

(ii) **Syntax:**

alter table <table name> modify (columnname new data type (size));

Example: alter table customer modify (social_security_no varchar2 (11));



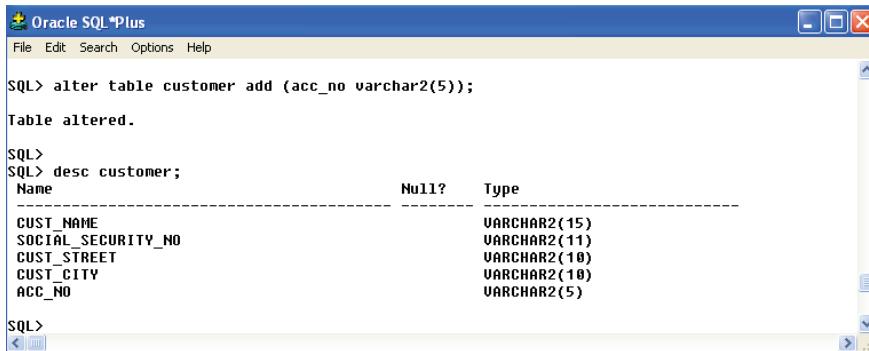
The screenshot shows the Oracle SQL*Plus interface. The command `alter table customer modify (social_security_no varchar2 (11));` is entered and executed, resulting in the message "Table altered.". A `desc customer;` command is then run, displaying the table structure:

Name	Null?	Type
CUST_NAME		VARCHAR2(15)
SOCIAL_SECURITY_NO		VARCHAR2(11)
CUST_STREET		VARCHAR2(10)
CUST_CITY		VARCHAR2(10)

Fig. 4.4. Modification of data type.**(iii) Syntax:**

$$\text{alter table} <\text{table name}> \text{add}(\text{new columnname data type (size)});$$

Example: alter table customer add (acc_no varchar2(5));



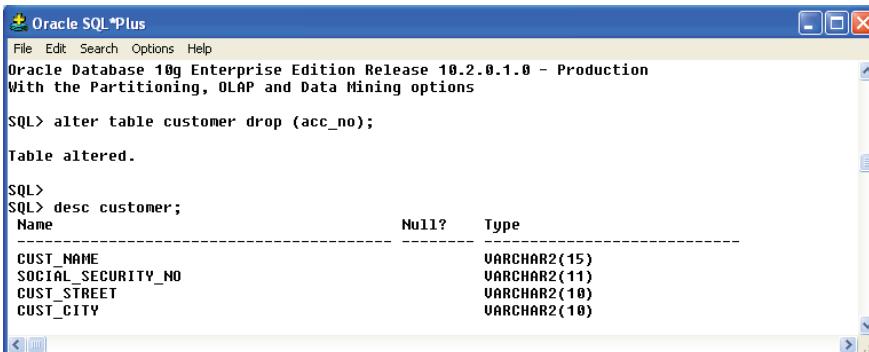
The screenshot shows the Oracle SQL*Plus interface. The command `alter table customer add (acc_no varchar2(5));` is entered and executed, resulting in the message "Table altered.". A `desc customer;` command is then run, displaying the table structure:

Name	Null?	Type
CUST_NAME		VARCHAR2(15)
SOCIAL_SECURITY_NO		VARCHAR2(11)
CUST_STREET		VARCHAR2(10)
CUST_CITY		VARCHAR2(10)
ACC_NO		VARCHAR2(5)

Fig. 4.5. Modification of column name.**(iv) Syntax:**

$$\text{alter table} <\text{table name}> \text{drop} (\text{column name});$$

Example: alter table customer drop (acc_no);



The screenshot shows the Oracle SQL*Plus interface. The command `alter table customer drop (acc_no);` is entered and executed, resulting in the message "Table altered.". A `desc customer;` command is then run, displaying the table structure:

Name	Null?	Type
CUST_NAME		VARCHAR2(15)
SOCIAL_SECURITY_NO		VARCHAR2(11)
CUST_STREET		VARCHAR2(10)
CUST_CITY		VARCHAR2(10)

Fig. 4.6. Modifying the table by dropping a column.

- 4. RENAME:** This command is used to change the name of the table.

Syntax:

rename <Old table name> to <New table name>;

Example: rename customer to customer1;

- 5. DROP:** The DROP command removes a table from the database. All the tables' rows, indexes and privileges will also be removed. No DML triggers will be fired. The operation cannot be rolled back.

Syntax:

drop table <table name>;

Example: drop table customer1;

- 6. TRUNCATE:** The TRUNCATE command removes **all rows** from a table. The operation cannot be rolled back and no triggers will be fired. TRUNCATE is faster.

Syntax:

truncate table <table name>;

Example: truncate table customer1;

The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main area displays the following SQL session:

```

File Edit Search Options Help
CUST_STREET          VARCHAR2(10)
CUST_CITY             VARCHAR2(10)

SQL> rename customer to customer1;
Table renamed.

SQL> truncate table customer1;
Table truncated.

SQL> drop table customer1;
Table dropped.

```

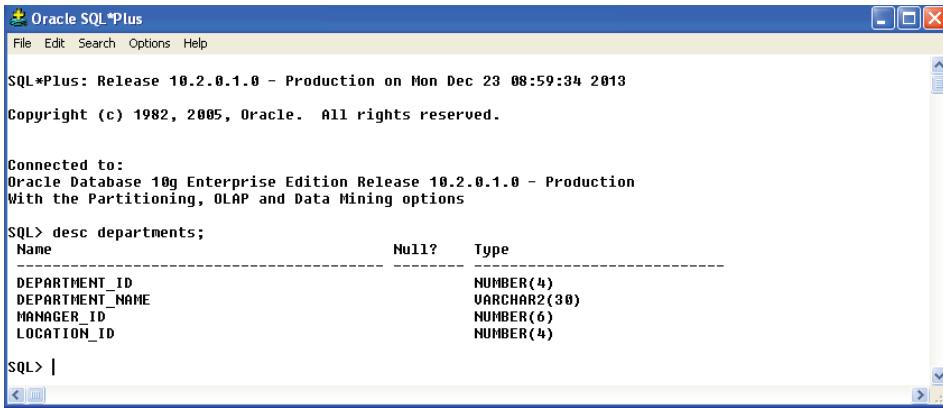
Fig. 4.7. Figure depicting rename, truncate and drop operations.

4.7 DATA MANIPULATION LANGUAGE (DML)

Data Manipulation Language (DML) statements are used for managing data within schema objects. Data Manipulation language commands let user to insert, modify and delete the data from database.

DML Commands

Consider the following table for understanding the DML commands



The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following text:

```

SQL*Plus: Release 10.2.0.1.0 - Production on Mon Dec 23 08:59:34 2013
Copyright (c) 1982, 2005, Oracle. All rights reserved.

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> desc departments;
Name          Null?    Type
-----        -----   -----
DEPARTMENT_ID      NUMBER(4)
DEPARTMENT_NAME    VARCHAR2(30)
MANAGER_ID         NUMBER(6)
LOCATION_ID        NUMBER(4)

SQL> |

```

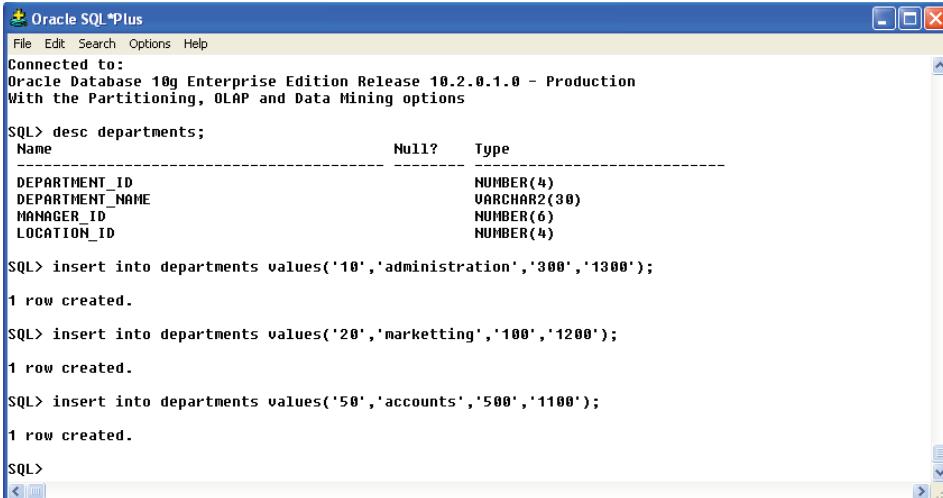
Fig. 4.8. Table departments.

1. INSERT: To insert one or more number of Rows

Syntax 1:

insert into <table name> values (List of Data Values);

Example: insert into departments values('10','administration','300','1300');



The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following text:

```

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> desc departments;
Name          Null?    Type
-----        -----   -----
DEPARTMENT_ID      NUMBER(4)
DEPARTMENT_NAME    VARCHAR2(30)
MANAGER_ID         NUMBER(6)
LOCATION_ID        NUMBER(4)

SQL> insert into departments values('10','administration','300','1300');

1 row created.

SQL> insert into departments values('20','marketting','100','1200');

1 row created.

SQL> insert into departments values('50','accounts','500','1100');

1 row created.

SQL>

```

Fig. 4.9. Inserting values into all columns of departments table.

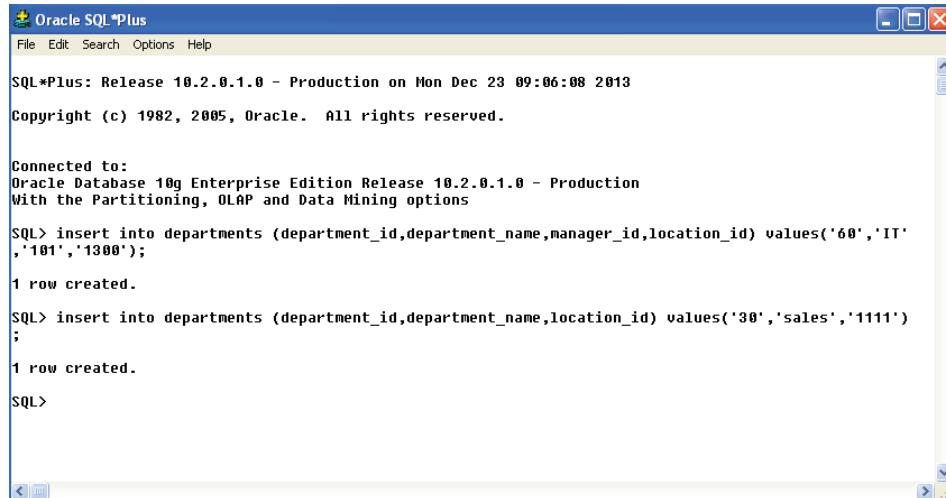
Syntax 2:

insert into <table name> (column names) values (list of data values);

Examples:

1. insert into departments (department_id,department_name,manager_id,location_id) values('60','IT','101','1300');
2. insert into departments (department_id,department_name,location_id) values('30','sales','1111');

Using this insert statement value of available columns can alone be inserted. In our first example we have inserted values for all the columns in the departments table but in the second example value for manager_id are not inserted.



```

Oracle SQL*Plus
File Edit Search Options Help
SQL*Plus: Release 10.2.0.1.0 - Production on Mon Dec 23 09:06:08 2013
Copyright (c) 1982, 2005, Oracle. All rights reserved.

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> insert into departments (department_id,department_name,manager_id,location_id) values('60','IT','101','1300');

1 row created.

SQL> insert into departments (department_id,department_name,location_id) values('30','sales','1111');

1 row created.

SQL>

```

Fig. 4.10. Inserting values into specified columns of the departments table.

Insert command using User interaction

Syntax 3:

insert into <Table name> values (&columnname1, &columnname2...);

Example: insert into departments values (&department_id,&department_name,&manager_id,&location_id);



```

Oracle SQL*Plus
File Edit Search Options Help
Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> insert into departments values(&department_id,&department_name,&manager_id,&location_id);
Enter value for department_id: 50
Enter value for department_name: 'shipping'
Enter value for manager_id: 121
Enter value for location_id: 1200
old    1: insert into departments values(&department_id,&department_name,&manager_id,&location_id)
new    1: insert into departments values(50,'shipping',121,1200)

1 row created.

SQL> /
Enter value for department_id: 100
Enter value for department_name: 'Finance'
Enter value for manager_id: 204
Enter value for location_id: 2312
old    1: insert into departments values(&department_id,&department_name,&manager_id,&location_id)
new    1: insert into departments values(100,'Finance',204,2312)

1 row created.

SQL>

```

Fig. 4.11. Inserting values into departments table using user interaction.

While inserting values into the table, values inserted should be enclosed in single quotes. For numerical values it's optional.

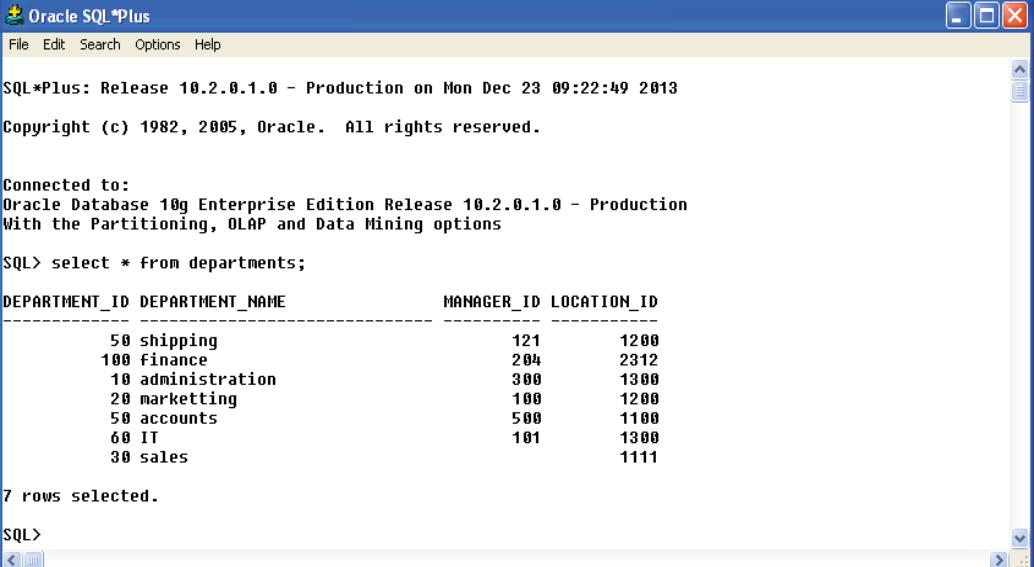
2. SELECT: To display one or more rows

Syntax 1:

```
select * from <table name>;
```

Example: select * from departments;

This query selects all the columns present in the specified table.



The screenshot shows the Oracle SQL*Plus interface. The command `select * from departments;` is entered and executed, displaying the following data:

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
50	shipping	121	1200
100	Finance	204	2312
10	administration	300	1300
20	marketing	100	1200
50	accounts	500	1100
60	IT	101	1300
30	sales		1111

7 rows selected.

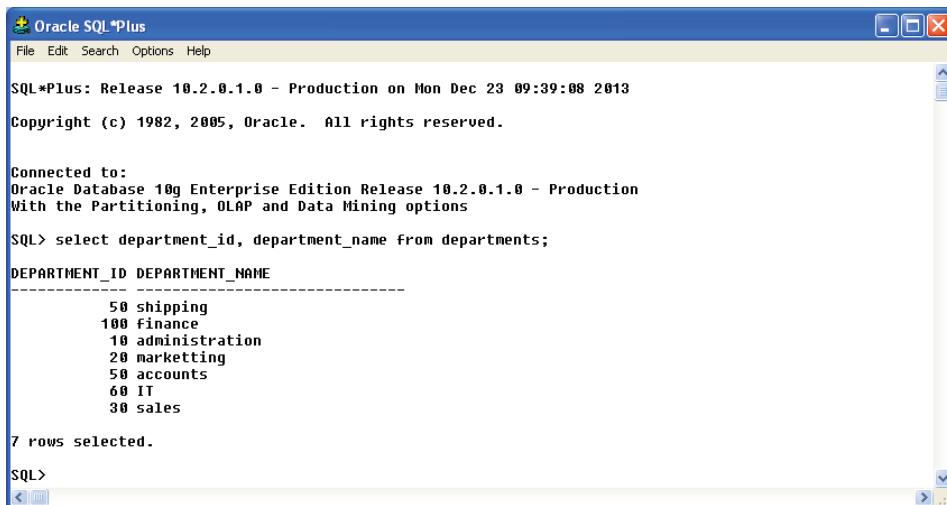
Fig. 4.12. Selecting and displaying all values present in a table.

Syntax 2:

```
select columnname1, columnname 2 from <table name>;
```

Example: select department_id, department_name from departments;

This select query helps us to select and display the required columns from the table. If the original table contains 10 columns and if the user wants to view only two or three then this type of select statement can be used. In our example columns department_id and department_name is selected from the departments table.



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> select department_id, department_name from departments;
```

The output displays the following data:

DEPARTMENT_ID	DEPARTMENT_NAME
50	shipping
100	finance
10	administration
20	marketing
50	accounts
60	IT
30	sales

7 rows selected.

SQL>

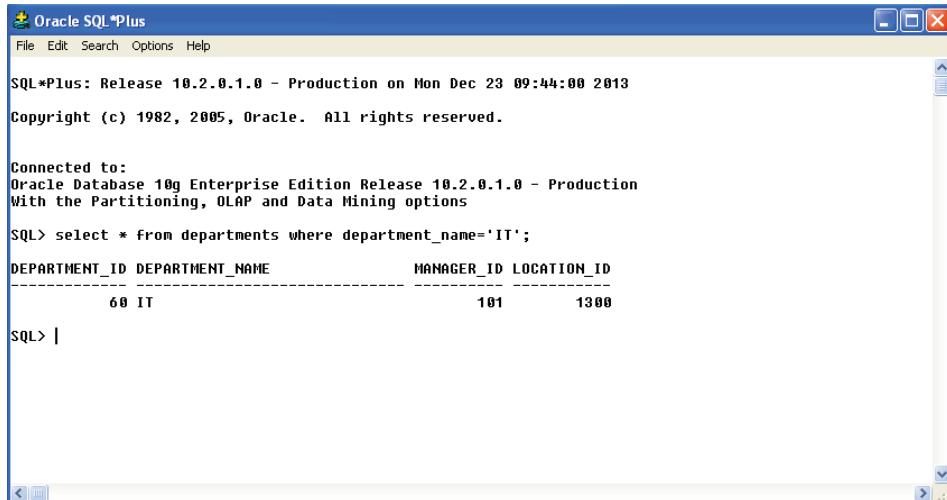
Fig. 4.13. Displaying selected columns from a table.

Syntax 3:

```
select * from <table name> where <condition>;
```

Example: select * from departments where department_name='IT';

This type of select statements helps the user to select values from the table based on some condition given in where clause. In our example we have selected the details of department_name='IT';



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> select * from departments where department_name='IT';
```

The output displays the following data:

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
60	IT	101	1300

SQL> |

Fig. 4.14. Selecting and displaying column values using some condition.

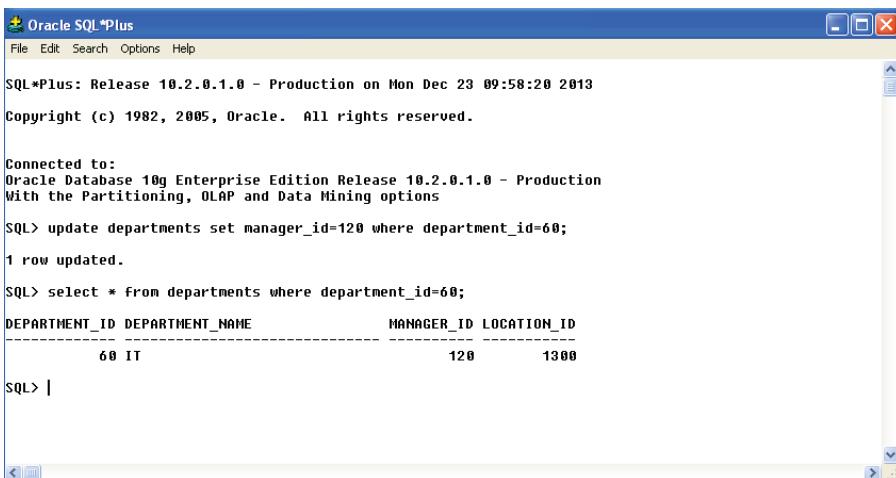
3. **UPDATE:** Used to alter the column values in a table. In the Update statement, WHERE clause identifies the rows that get affected. If you do not include the WHERE clause, column values for all the rows get affected.

Syntax:

update <table name> set column_name=new_value where <condition>;

Example: update departments set manager_id=120 where department_id=60;

In our example the manager_id is affected for the department with department_id=60.



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> update departments set manager_id=120 where department_id=60;
1 row updated.
```

After executing the update, a select statement is run to verify the changes:

```
SQL> select * from departments where department_id=60;
DEPARTMENT_ID DEPARTMENT_NAME          MANAGER_ID LOCATION_ID
-----          -----          -----
      60 IT                      120          1300
```

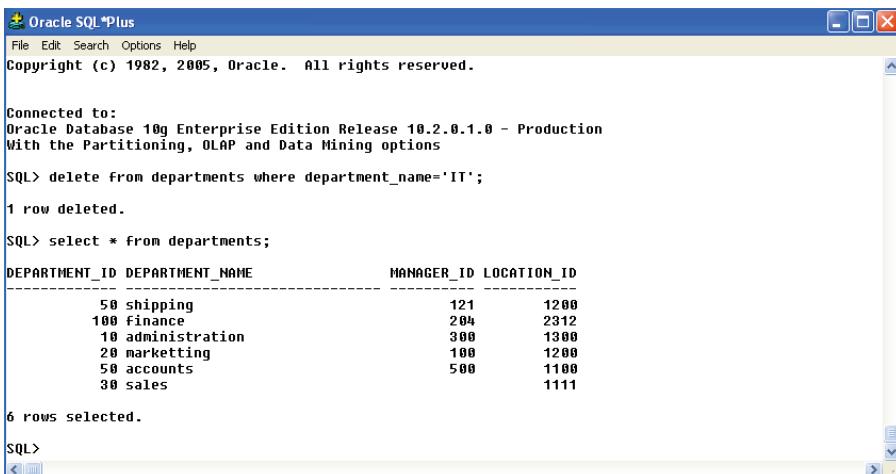
Fig. 4.15. Updating column values based on a condition.

4. DELETE: Used to delete one or more rows.**Syntax:**

delete from <table name> where <condition>;

Example: delete from departments where department_name='IT';

In our example the entire details of department with department_name='IT' will be deleted from the table departments.



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> delete from departments where department_name='IT';
1 row deleted.
```

After executing the delete, a select statement is run to show the remaining data:

```
SQL> select * from departments;
DEPARTMENT_ID DEPARTMENT_NAME          MANAGER_ID LOCATION_ID
-----          -----          -----
      50 shipping                      121          1200
      100 finance                     204          2312
      10 administration                300          1300
      20 marketing                     100          1200
      50 accounts                      500          1100
      30 sales                         111          1111
```

Fig. 4.16. Deleting a value from a table based on a condition.

4.8 BASIC STRUCTURE OF SQL EXPRESSION

SQL expression consists of three clauses:

- **Select:** The select clause corresponds to projection operation of the relational algebra. The SELECT statement is used to query the database and retrieve the fields that you specify. You can select as many fields (column names) as you want, or use the asterisk symbol “*” to select all fields.
- **From:** The from clause corresponds to the Cartesian product operation of the relational algebra. It specifies the table names that will be queried to retrieve the desired data.
- **Where:** The where clause corresponds to the selection predicate of the relations that appear in the from clause. The WHERE clause (optional) specifies which data values or rows will be returned or displayed, based on the criteria you specify.

4.8.1 General Form of SQL Query

```
Select A1, A2....., An
From R1, R2....., Rm
Where P
```

Where, A1 = represent attributes

R1 = represent relation

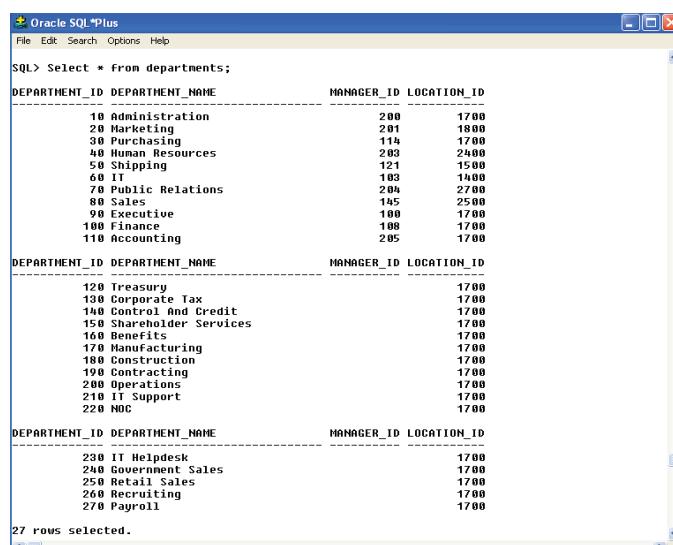
P = is a predicate

4.8.2 Selecting all Columns from a Relation

Syntax:

*Select * from <table name>;*

Example: Select * from departments;



The screenshot shows the Oracle SQL*Plus interface with the following command and its output:

```
SQL> Select * from departments;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	108	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury	1700	
130	Corporate Tax	1700	
140	Control And Credit	1700	
150	Shareholder Services	1700	
160	Quality Management	1700	
170	Manufacturing	1700	
180	Construction	1700	
190	Contracting	1700	
200	Operations	1700	
210	IT Support	1700	
220	NDC	1700	
230	IT Helpdesk	1700	
240	Government Sales	1700	
250	Retail Sales	1700	
260	Recruiting	1700	
270	Payroll	1700	

27 rows selected.

Fig. 4.17. Selecting all values from a table.

All the columns in a table can be displayed by following the SELECT keyword with an asterisk (*). Instead of using * it can also be displayed by listing all the columns after the SELECT Keyword.

For example:

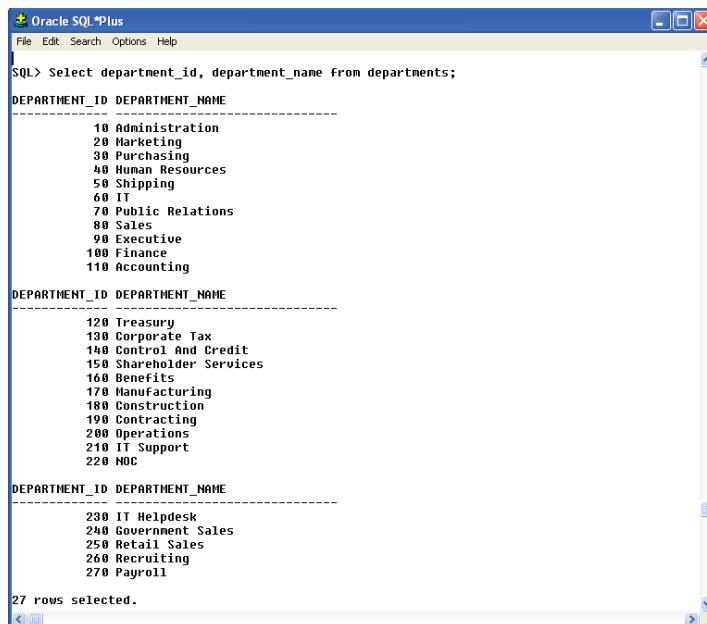
Select department_id, department_name, manager_id, location_id from departments;

4.8.3 Selecting Specific Columns

Syntax:

Select column_name1, column_name2..... from departments;

Example: Select department_id, department_name from departments;



The screenshot shows the Oracle SQL*Plus interface with a query window. The query is:

```
SQL> Select department_id, department_name from departments;
```

The output displays three distinct department groups, each with its own header:

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
30	Purchasing
40	Human Resources
50	Shipping
60	IT
70	Public Relations
80	Sales
90	Executive
100	Finance
110	Accounting

DEPARTMENT_ID	DEPARTMENT_NAME
120	Treasury
130	Corporate Tax
140	Control And Credit
150	Shareholder Services
160	Benefits
170	Manufacturing
180	Construction
190	Contracting
200	Operations
210	IT Support
220	NUC

DEPARTMENT_ID	DEPARTMENT_NAME
230	IT Helpdesk
240	Government Sales
250	Retail Sales
260	Recruiting
270	Payroll

27 rows selected.

Fig. 4.18. Selecting specific values from a table.

Specific columns of the table can be selected by specifying the column name separated by commas in the SELECT statement. The order in which the column names are listed in the query will be the order in which the names will appear in the output. In the example given above the department_id and department_name are listed from the departments table.

4.8.4 Usage of DISTINCT Keyword in SELECT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values. In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values in such case DISTINCT keyword can be used.

Syntax:

SELECT DISTINCT column_name1, column_name2,... FROM table_name;

Example: select distinct location_id from departments;

```
SQL> select distinct location_id from departments;

LOCATION_ID
-----
    1800
    2400
    1400
    2500
    1700
    2700
    1500

7 rows selected.
```

Fig. 4.19. Usage of DISTINCT keyword in SELECT statement.

4.8.5 Usage of ALL Keyword in SELECT Statement

The SELECT ALL statement is used to return all the values. In a table, a column may contain many duplicate values and if you want to list all the values without caring about the duplicate values in such cases ALL keyword can be used. If SELECT statement will use ALL keyword by default even if ALL keyword is not specified explicitly. If duplicate is to be removed then DISTINCT key should be explicitly used in the SELECT statement.

Syntax:

SELECT ALL column name1, column name2,... FROM table name;

Example: select all location id from departments;

```
Oracle SQL*Plus
File Edit Search Options Help
SQL> select all location_id from departments;
LOCATION_ID
-----
1700
1800
1700
2400
1500
1400
2700
2500
1700
1700
1700

LOCATION_ID
-----
1700
1700
1700
1700
1700
1700
1700
1700
1700
1700
1700

LOCATION_ID
-----
1700
1700
1700
1700
1700

27 rows selected.
```

Fig. 4.20. Usage of ALL keyword in SELECT statement.

4.9 COLUMN ALIAS NAME

Column Alias is nothing but renaming the columns. The SQL provides a mechanism for renaming both relations and attributes. It uses the as clause using the form:

Old-name as new-name

The keyword ‘as’ is optional. The result of the query is same whether the AS keyword is used or not. Even if the column heading is given in lowercase it to be in uppercase by default when the query displays the result. The alias name should immediately follow the column name. The alias name should be specified in double quotation if it contains spaces or special character or if it is case sensitive.

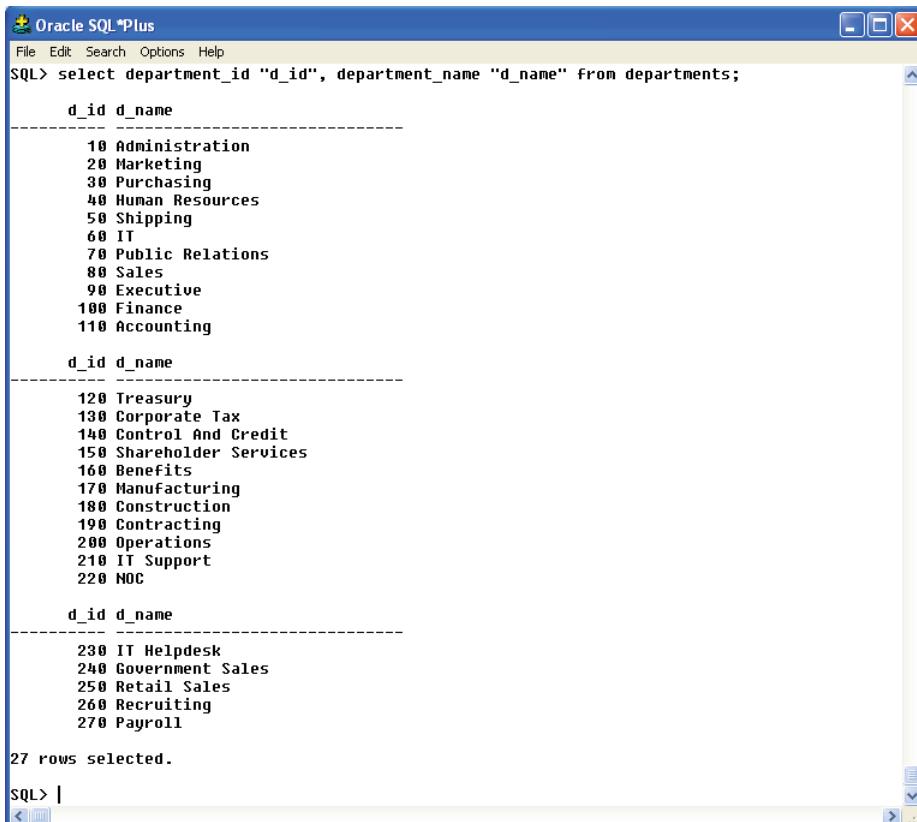
Syntax:

```
SELECT old column_name1 AS new column_name1, old column_name2 AS new
column_name2,..... FROM table_name WHERE [condition];
```

Example: select department_id as d_id, department_name as d_name from departments;

The same query can be rewritten as

```
select department_id "d_id", department_name "d_name" from departments;
```



The screenshot shows the Oracle SQL*Plus interface with a window titled 'Oracle SQL*Plus'. The window contains the following SQL query and its execution results:

```
File Edit Search Options Help
SQL> select department_id "d_id", department_name "d_name" from departments;
      d_id d_name
-----+
      10 Administration
      20 Marketing
      30 Purchasing
      40 Human Resources
      50 Shipping
      60 IT
      70 Public Relations
      80 Sales
      90 Executive
     100 Finance
     110 Accounting

      d_id d_name
-----+
     120 Treasury
     130 Corporate Tax
     140 Control And Credit
     150 Shareholder Services
     160 Benefits
     170 Manufacturing
     180 Construction
     190 Contracting
     200 Operations
     210 IT Support
     220 NOC

      d_id d_name
-----+
     230 IT Helpdesk
     240 Government Sales
     250 Retail Sales
     260 Recruiting
     270 Payroll

27 rows selected.

SQL> |
```

The output displays three distinct sets of department data, each with a different set of department IDs and names, demonstrating the use of column aliases to rename the columns.

Fig. 4.21. Query using alias name for columns in the SELECT statement.

4.10 STRING OPERATION

SQL includes a string-matching operator for comparisons on character strings. The operator “like” uses patterns that are described using two special characters:

- The percent sign (%) matches any substring
- The underscore (_) matches any character

The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character. The symbols can be used in combinations.

Syntax:

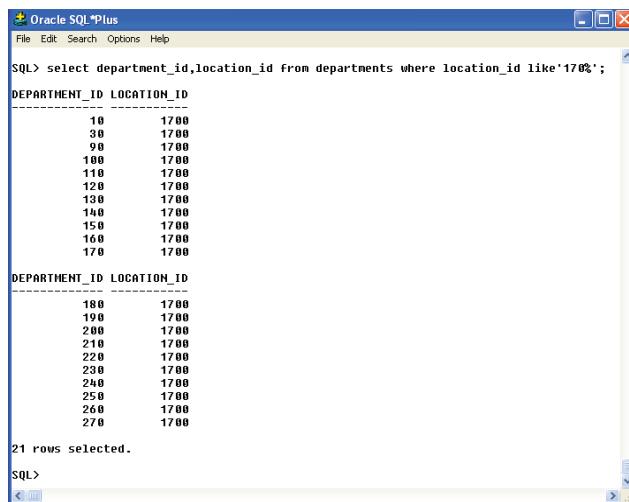
```
SELECT FROM table_name WHERE column LIKE 'XXXX%'  
or  
SELECT FROM table_name WHERE column LIKE '%XXXX%'  
or  
SELECT FROM table_name WHERE column LIKE 'XXXX_'  
or  
SELECT FROM table_name WHERE column LIKE '_XXXX'  
or  
SELECT FROM table_name WHERE column LIKE '_XXXX_'
```

You can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

Example: Here are number of examples showing WHERE part having different LIKE clause with ‘%’ and ‘_’ operators:

Statement	Description
WHERE LOCATION_ID LIKE '200%'	Finds any values that start with 200
WHERE LOCATION_ID LIKE '%200%'	Finds any values that have 200 in any position
WHERE LOCATION_ID LIKE '_00%'	Finds any values that have 00 in the second and third positions
WHERE LOCATION_ID LIKE '2_%_%'	Finds any values that start with 2 and are at least 3 characters in length
WHERE LOCATION_ID LIKE '%2'	Finds any values that end with 2
WHERE LOCATION_ID LIKE '_2%3'	Finds any values that have a 2 in the second position and end with a 3
WHERE LOCATION_ID LIKE '2__3'	Finds any values in a five-digit number that start with 2 and end with 3

Example: select department_id,location_id from departments where location_id like '170%';



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> select department_id,location_id from departments where location_id like'170%';
```

The output displays two sets of data, each with DEPARTMENT_ID and LOCATION_ID columns:

DEPARTMENT_ID	LOCATION_ID
10	1700
30	1700
90	1700
180	1700
110	1700
120	1700
130	1700
140	1700
150	1700
160	1700
170	1700

DEPARTMENT_ID	LOCATION_ID
180	1700
190	1700
200	1700
210	1700
220	1700
230	1700
240	1700
250	1700
260	1700
270	1700

21 rows selected.

SQL>

Fig. 4.22. Query using string operation.

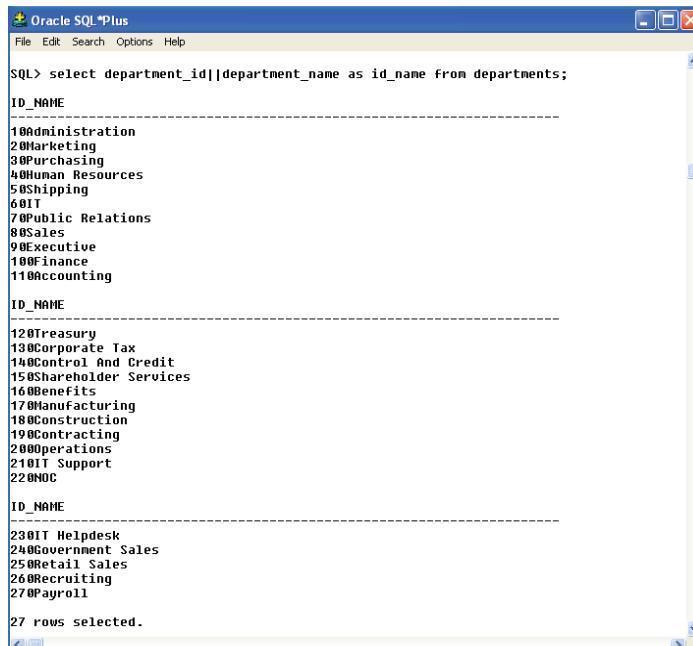
4.11 CONCATENATION OPERATION

Concatenation operation links columns or character strings to other columns. It is represented by two vertical bars (||).

Syntax:

SELECT column_name1||column_name2 as new column_name from TABLE_NAME;

Example: select department_id||department_name as id_name from departments;



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> select department_id||department_name as id_name from departments;
```

The output displays three sets of data, each with an ID_NAME column:

ID_NAME
10Administration
20Marketing
30Purchasing
40Human Resources
50Shipping
60IT
70Public Relations
80Sales
90Executive
100Finance
110Accounting

ID_NAME
120Treasury
130Corporate Tax
140Control And Credit
150Shareholder Services
160Benefits
170Manufacturing
180Construction
190Contracting
200Operations
210IT Support
220NOC

ID_NAME
230IT Helpdesk
240Government Sales
250Retail Sales
260Recruiting
270Payroll

27 rows selected.

SQL>

Fig. 4.23. Query using concatenation operation.

When the above query is executed the department_id and department_name of departments table is merged together and displayed under the column heading id_name.

Literals can also be included in the output. Literals are nothing but a character or number or date that is included in the SELECT list. Literal is not a column name or column alias. The literal will be printed once for each row. Date and character literals must be enclosed by single quotation mark (' '); number literals need not be enclosed in quotation.

Syntax:

```
SELECT column_name1||'literal'||column_name2 as new_column_name from TABLE_NAME;
```

Example: select department_id||'corresponds to'||department_name as id_name from departments;

The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> select department_id||'corresponds to'||department_name as id_name from departments;
```

The output displays three groups of rows, each starting with 'ID_NAME' followed by a dashed line. The first group contains rows 1 through 11, the second group contains rows 12 through 22, and the third group contains rows 23 through 27. Each row consists of the concatenated string 'corresponds to' followed by the department name. The final message '27 rows selected.' is shown at the bottom.

ID_NAME
10corresponds toAdministration
20corresponds toMarketing
30corresponds toPurchasing
40corresponds toHuman Resources
50corresponds toShipping
60corresponds toIT
70corresponds toPublic Relations
80corresponds toSales
90corresponds toExecutive
100corresponds toFinance
110corresponds toAccounting
ID_NAME
120corresponds toTreasury
130corresponds toCorporate Tax
140corresponds toControl And Credit
150corresponds toShareholder Services
160corresponds toBenefits
170corresponds toManufacturing
180corresponds toConstruction
190corresponds toContracting
200corresponds toOperations
210corresponds toIT Support
220corresponds toMOC
ID_NAME
230corresponds toIT Helpdesk
240corresponds toGovernment Sales
250corresponds toRetail Sales
260corresponds toRecruiting
270corresponds toPayroll

27 rows selected.

Fig. 4.24. Query using concatenation operation along with literals.

4.12 ORDERING THE DISPLAY OF TUPLES

The ORDER BY clause is used in a SELECT statement to sort results either in ascending or descending order. Oracle sorts query results in ascending order by default.

Syntax:

```
SELECT column-list FROM table_name [WHERE condition] [ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

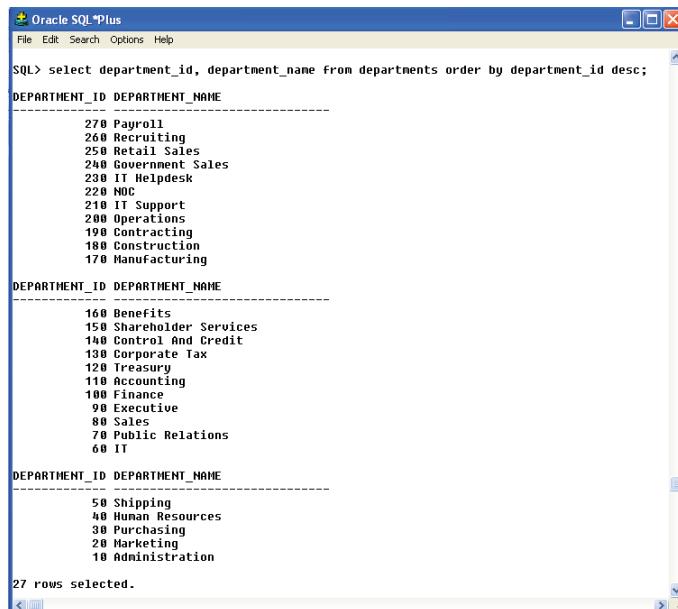
ORDER BY – Specifies the order in which rows are displayed

ASC – Orders the rows in ascending order (this is the default order)

DESC – Orders the rows in descending order

The statement enclosed in square brackets [] is optional.

Example: select department_id, department_name from departments order by department_id desc;



```
SQL> select department_id, department_name from departments order by department_id desc;
DEPARTMENT_ID DEPARTMENT_NAME
-----
270 Payroll
260 Recruiting
250 Retail Sales
240 Government Sales
230 IT Helpdesk
220 NDC
210 IT Support
200 Operations
190 Contracting
180 Construction
170 Manufacturing

DEPARTMENT_ID DEPARTMENT_NAME
-----
160 Benefits
150 Shareholder Services
140 Control And Credit
130 Corporate Tax
120 Treasury
110 Accounting
100 Finance
90 Executive
80 Sales
70 Public Relations
60 IT

DEPARTMENT_ID DEPARTMENT_NAME
-----
50 Shipping
40 Human Resources
30 Purchasing
20 Marketing
10 Administration

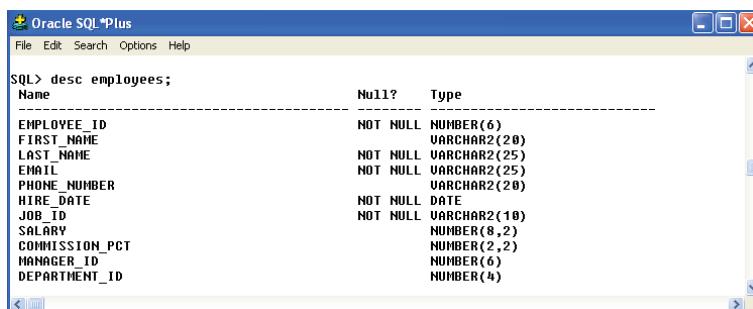
27 rows selected.
```

Fig. 4.25. Ordering the tuples in descending order.

4.13 SET OPERATIONS

Set operators combine the results of two queries into a single one. Let us consider the following tables to learn about set operations.

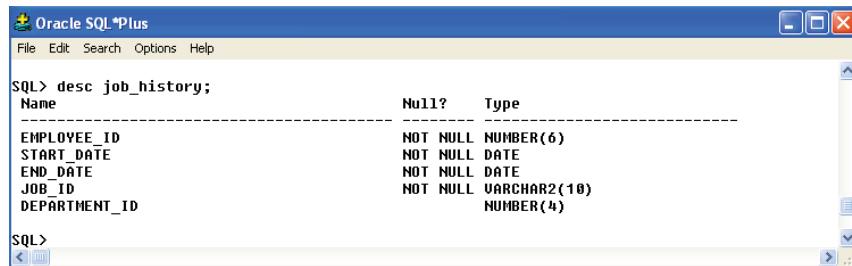
EMPLOYEES TABLE



Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(28)
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)

Fig. 4.26. Employees table.

JOB_HISTORY TABLE



The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main area displays the output of the command "SQL> desc job_history;". The output shows the following columns and their properties:

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

Fig. 4.27. Job_History table.

4.13.1 Union

The SQL **UNION** clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows. To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be the same length.

Syntax:

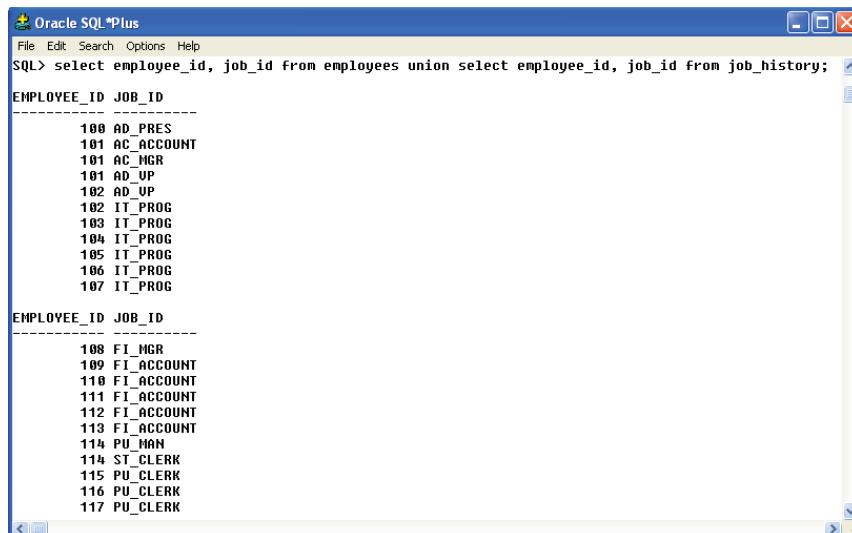
SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]

UNION

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]

Example: select employee_id, job_id from employees **union** select employee_id, job_id from job_history;

To know the current and previous job details of all employees the above query is executed.



The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The main area displays the output of the command "SQL> select employee_id, job_id from employees union select employee_id, job_id from job_history;". The output consists of two parts, each with a header "EMPLOYEE_ID JOB_ID". The first part shows data from the employees table, and the second part shows data from the job_history table.

EMPLOYEE_ID JOB_ID	
100	AD_PRES
101	AC_ACCOUNT
101	AC_MGR
101	AD_UP
102	AD_UP
102	IT_PROG
103	IT_PROG
104	IT_PROG
105	IT_PROG
106	IT_PROG
107	IT_PROG

EMPLOYEE_ID JOB_ID	
108	FI_MGR
109	FI_ACCOUNT
110	FI_ACCOUNT
111	FI_ACCOUNT
112	FI_ACCOUNT
113	FI_ACCOUNT
114	PU_MAN
114	ST_CLERK
115	PU_CLERK
116	PU_CLERK
117	PU_CLERK

Fig. 4.28. Union Operation of two tables.

4.13.2 Union All

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows. The same rules that apply to UNION apply to the UNION ALL operator. The output is not sorted by default and distinct keyword cannot be used with this operator.

Syntax:

```
SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]
```

```
UNION ALL
```

```
SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]
```

Example: select employee_id, job_id from employees **union all** select employee_id, job_id from job_history;

```
SQL> select employee_id, job_id from employees union all select employee_id, job_id from job_history
;
EMPLOYEE_ID JOB_ID
-----
198 SH_CLERK
199 SH_CLERK
200 AD_ASST
201 MK_MAN
202 MK_REP
203 HR_REP
204 PR_REP
205 AC_MGR
206 AC_ACCOUNT
100 AD_PRES
101 AD_UP

EMPLOYEE_ID JOB_ID
-----
102 AD_UP
103 IT_PROG
104 IT_PROG
105 IT_PROG
106 IT_PROG
107 IT_PROG
108 FI_MGR
109 FI_ACCOUNT
110 FI_ACCOUNT
```

Fig. 4.29. Union all of two tables.

4.13.3 Intersect

The SQL INTERSECT clause/operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements. Just as with the UNION operator, the same rules apply when using the INTERSECT operator.

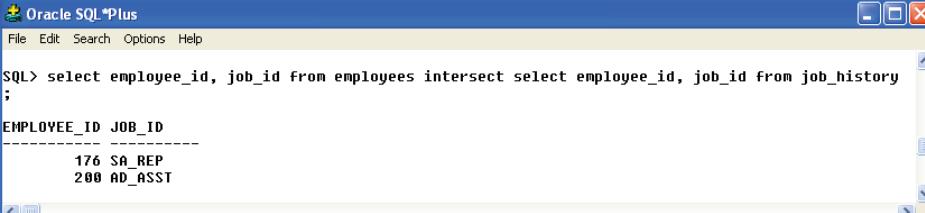
Syntax:

```
SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]
```

```
INTERSECT
```

```
SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]
```

Example: select employee_id, job_id from employees **intersect** select employee_id, job_id from job_history;



The screenshot shows the Oracle SQL*Plus interface with the following command and output:

```
SQL> select employee_id, job_id from employees intersect select employee_id, job_id from job_history;
EMPLOYEE_ID JOB_ID
-----
176 SA_REP
200 AD_ASST
```

Fig. 4.30. Intersection all of two tables.

4.13.4 Minus or Except

The SQL Minus or **EXCEPT** clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement. This means EXCEPT returns only rows, which are not available in second SELECT statement. EXCEPT operator follows the same rules as the UNION operator.

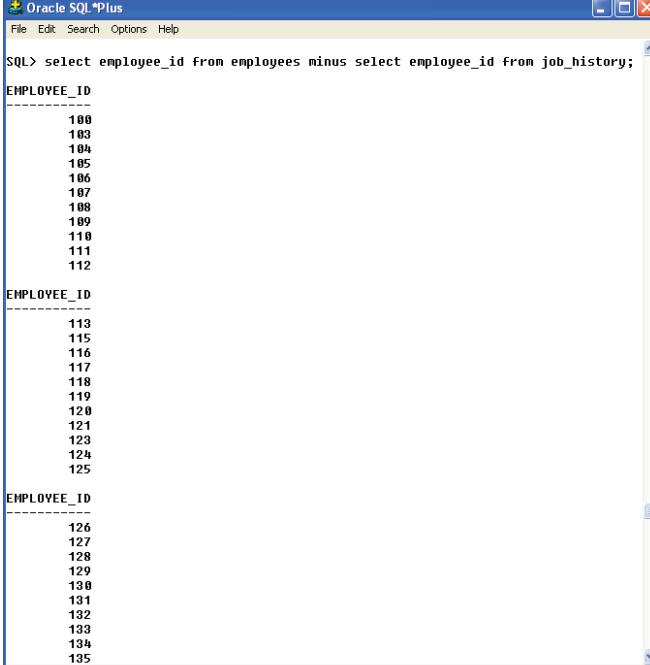
Syntax:

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]

EXCEPT

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]

Example: select employee_id from employees **minus** select employee_id from job_history;



The screenshot shows the Oracle SQL*Plus interface with the following command and output:

```
SQL> select employee_id from employees minus select employee_id from job_history;
EMPLOYEE_ID
-----
188
183
184
185
186
187
188
189
110
111
112
EMPLOYEE_ID
-----
113
115
116
117
118
119
120
121
123
124
125
EMPLOYEE_ID
-----
126
127
128
129
130
131
132
133
134
135
```

Fig. 4.31. Minus all of two tables.

4.14 WHERE CLAUSE

The WHERE Clause is used to retrieve specific information from a table excluding other irrelevant data.

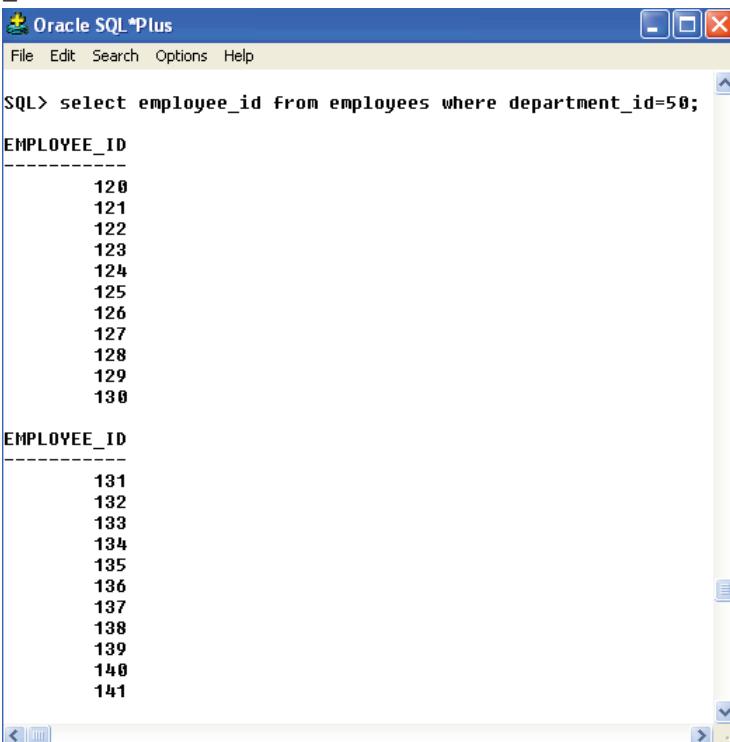
For example, when you want to see the information of all employees who work in a particular department we can remove the irrelevant information of other employees using where clause. This helps to reduce the processing time. So SQL offers a feature called WHERE clause, which we can use to restrict the data that is retrieved. The condition you provide in the WHERE clause filters the rows retrieved from the table and gives you only those rows which you expected to see. WHERE clause can be used along with SELECT, DELETE and UPDATE statements.

Syntax:

```
SELECT column_list FROM table-name WHERE condition;
```

Example: select employee_id from employees where department_id=50;

This query filters and displays the employee_id of those employees who belongs to department with department_id=50.



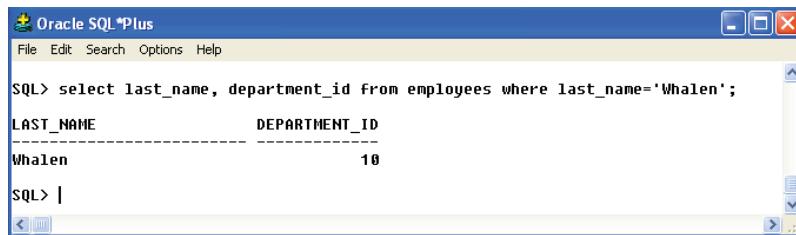
```
SQL> select employee_id from employees where department_id=50;
EMPLOYEE_ID
-----
120
121
122
123
124
125
126
127
128
129
130

EMPLOYEE_ID
-----
131
132
133
134
135
136
137
138
139
140
141
```

Fig. 4.32. Query using WHERE clause for restricting column retrieval using numerical value.

Character strings and date values can also be used in where clause but should be enclosed in single quotation marks. Character values are case sensitive and date values are format sensitive.

Example: select last_name, department_id from employees where last_name='Whalen';



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> select last_name, department_id from employees where last_name='Whalen';
```

The output is:

LAST_NAME	DEPARTMENT_ID
Whalen	10

Fig. 4.33. Query using WHERE clause for restricting column retrieval using character value.

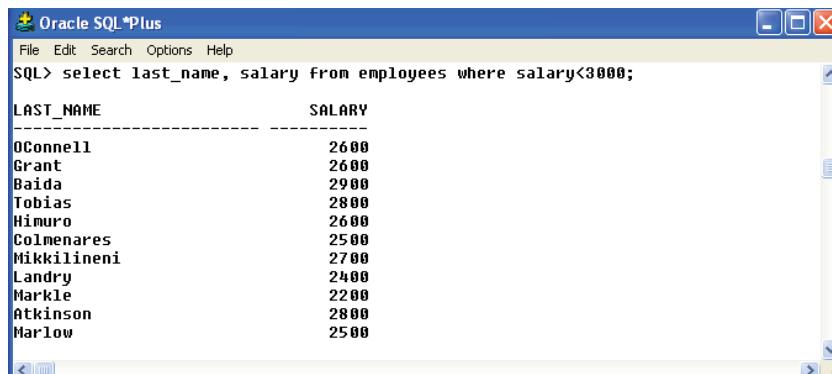
Comparison Operators and Logical Operators are used in WHERE Clause. Aliases defined for the columns in the SELECT statement cannot be used in the WHERE clause to set conditions. Only aliases created for tables can be used to reference the columns in the table. Expressions can also be used in the WHERE clause of the SELECT statement.

4.14.1 Operators in the WHERE Clause

The following operators can be used in the WHERE clause:

Operator	Description
=	Equal
<>	Not equal. Note: In some versions of SQL this operator may be written as !=
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

Example 1: select last_name, salary from employees where salary<3000;



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

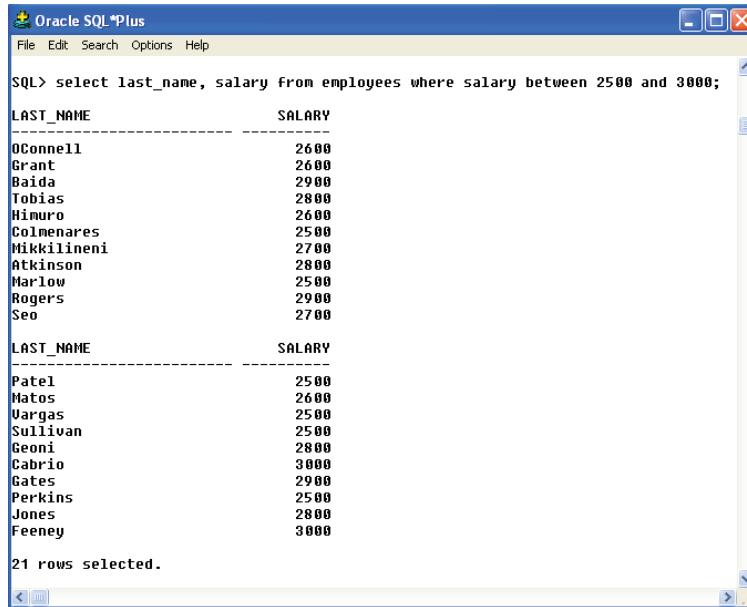
```
SQL> select last_name, salary from employees where salary<3000;
```

The output is:

LAST_NAME	SALARY
OConnell	2600
Grant	2600
Baida	2900
Tobias	2800
Himuro	2600
Colmenares	2500
Mikkilineni	2700
Landry	2400
Markle	2200
Atkinson	2800
Marlow	2500

Fig. 4.34. Query using operators in WHERE clause.

Example 2: (Between operator) select last_name, salary from employees where salary between 2500 and 3000;



The screenshot shows the Oracle SQL*Plus interface with a window titled "Oracle SQL*Plus". The menu bar includes File, Edit, Search, Options, and Help. The SQL command entered is "SQL> select last_name, salary from employees where salary between 2500 and 3000;". The output displays two sets of data, each with columns LAST_NAME and SALARY. The first set contains rows for O'Connell, Grant, Baida, Tobias, Hinuro, Colmenares, Mikkilineni, Atkinson, Marlow, Rogers, and Seo, with salaries ranging from 2500 to 2700. The second set contains rows for Patel, Matos, Vargas, Sullivan, Geoni, Cabrio, Gates, Perkins, Jones, and Feeney, with salaries ranging from 2500 to 3000. A message at the bottom indicates "21 rows selected."

```

SQL> select last_name, salary from employees where salary between 2500 and 3000;

LAST_NAME          SALARY
-----          -----
O'Connell          2600
Grant              2600
Baida              2900
Tobias              2800
Hinuro              2600
Colmenares         2500
Mikkilineni        2700
Atkinson            2800
Marlow              2500
Rogers              2900
Seo                 2700

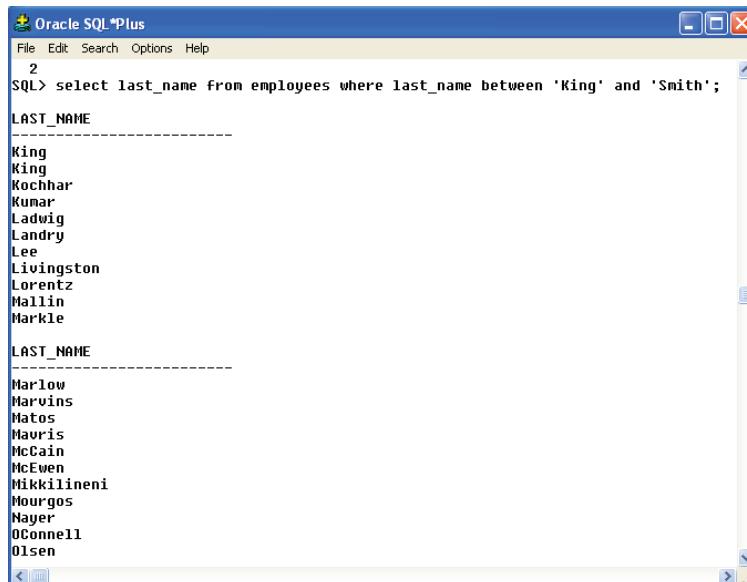
LAST_NAME          SALARY
-----          -----
Patel              2500
Matos              2600
Vargas              2500
Sullivan            2500
Geoni              2800
Cabrio              3000
Gates              2900
Perkins             2500
Jones               2800
Feeney              3000

21 rows selected.

```

Fig. 4.35. Query using BETWEEN operators using numerical values in WHERE clause.

Example 3: (Between operator) select last_name from employees where last_name between 'King' and 'Smith';



The screenshot shows the Oracle SQL*Plus interface with a window titled "Oracle SQL*Plus". The menu bar includes File, Edit, Search, Options, and Help. The SQL command entered is "SQL> select last_name from employees where last_name between 'King' and 'Smith';". The output displays two sets of data, each with column LAST_NAME. The first set contains rows for King, Kochhar, Kumar, Ludwig, Landry, Lee, Livingston, Lorentz, Mallin, and Markle. The second set contains rows for Marlow, Marvins, Matos, Mauris, McCain, McEwen, Mikkilineni, Mourgos, Nayer, O'Connell, and Olsen. A message at the top indicates "2 rows selected."

```

SQL> select last_name from employees where last_name between 'King' and 'Smith';

LAST_NAME
-----
King
King
Kochhar
Kumar
Ludwig
Landry
Lee
Livingston
Lorentz
Mallin
Markle

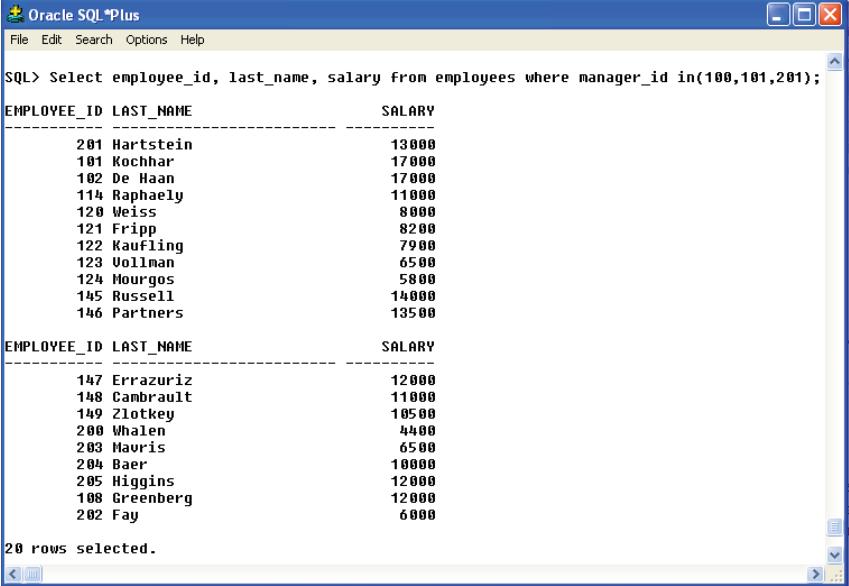
LAST_NAME
-----
Marlow
Marvins
Matos
Mauris
McCain
McEwen
Mikkilineni
Mourgos
Nayer
O'Connell
Olsen

2 rows selected.

```

Fig. 4.36. Query using BETWEEN operators using character values in WHERE clause.

Example 4: (IN) Select employee_id, last_name, salary from employees where manager_id in(100,101,201);



The screenshot shows the Oracle SQL*Plus interface with the following output:

```
SQL> Select employee_id, last_name, salary from employees where manager_id in(100,101,201);
EMPLOYEE_ID LAST_NAME          SALARY
-----  -----
  201 Hartstein           13000
  101 Kochhar            17000
  102 De Haan             17000
  114 Raphaely            11000
  120 Weiss               8000
  121 Fripp                8200
  122 Kauffling            7900
  123 Vollman              6500
  124 Mourgos              5800
  145 Russell              14000
  146 Partners             13500

EMPLOYEE_ID LAST_NAME          SALARY
-----  -----
  147 Errazuriz            12000
  148 Cambrauit            11000
  149 Zlotkey              10500
  200 Whalen                4400
  203 Mauris               6500
  204 Baer                 10000
  205 Higgins              12000
  108 Greenberg             12000
  202 Fay                  6000

20 rows selected.
```

Fig. 4.37. Query using IN operators in WHERE clause.

4.15 OPERATORS

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

4.15.1 Arithmetic Operators

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder

4.15.2. SQL Comparison Operators

Operator	Description
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.

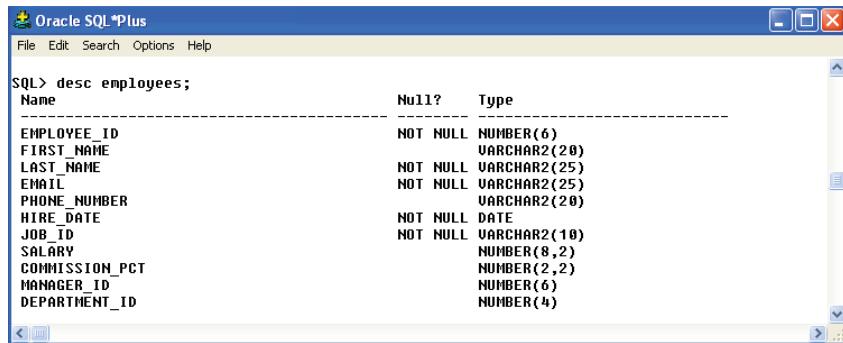
4.15.3 SQL Logical Operators

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. This is a negate operator.
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

4.16 AGGREGATE FUNCTION

Aggregate functions perform a calculation on a set of values and return a single value. Except for COUNT, aggregate functions ignore null values. Aggregate functions are frequently used with the GROUP BY clause of the SELECT statement. All aggregate functions are deterministic. This means aggregate functions return the same value any time that they are called by using a specific set of input values.

These functions operate on the multiset of values of a column of a relation, and return a value
Consider the following employees table



The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the output of the SQL command "SQL> desc employees;". The output shows the following columns and their properties:

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)

Fig. 4.38. Employees table.

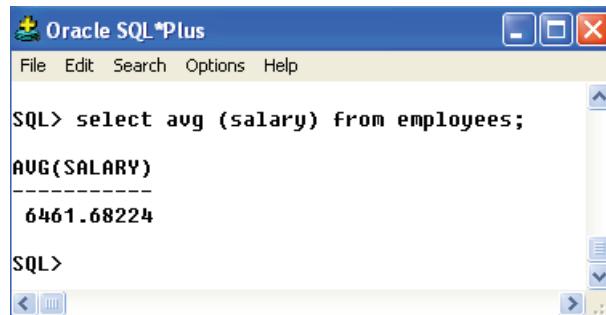
4.16.1 AVG

The AVG() function returns the average value of a numeric column.

Syntax: `SELECT AVG(column_name) FROM table_name`

Example: Find the average of salary from the employees table

Query: `select avg (salary) from employees;`



The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the output of the SQL command "SQL> select avg (salary) from employees;". The output shows the result of the AVG(SALARY) function:

Avg(Salary)
6461.68224

Fig. 4.39. Query using AVG() function.

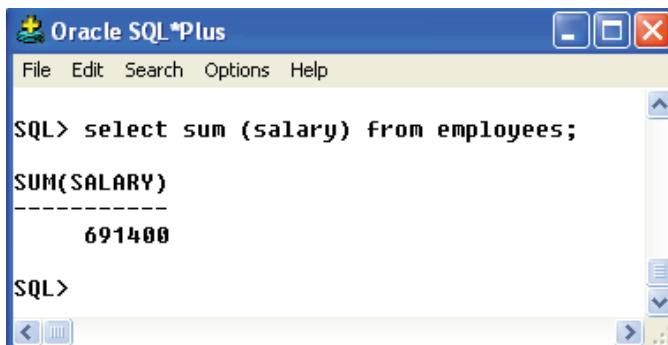
4.16.2 SUM

The SUM() function returns the total sum of a numeric column.

Syntax: `SELECT SUM(column_name) FROM table_name;`

Example: Find the sum of salary from the employees table

Query: select **sum** (salary) from employees;



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window contains the following SQL command and its output:

```
SQL> select sum (salary) from employees;
SUM(SALARY)
-----
 691400
SQL>
```

Fig. 4.40. Query using *SUM()* function.

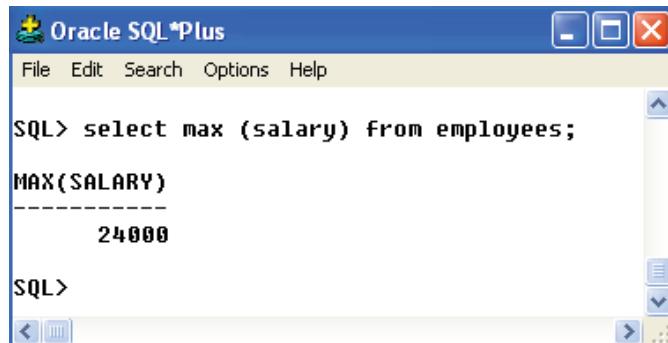
4.16.3 MAX

The MAX() function returns the largest value of the selected column.

Syntax: `SELECT MAX(column_name) FROM table_name;`

Example: Find the Maximum value of account balance from the account table.

Query: select**max**(salary) from employees;



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window contains the following SQL command and its output:

```
SQL> select max (salary) from employees;
MAX(SALARY)
-----
 24000
SQL>
```

Fig. 4.41. Query using *MAX()* function.

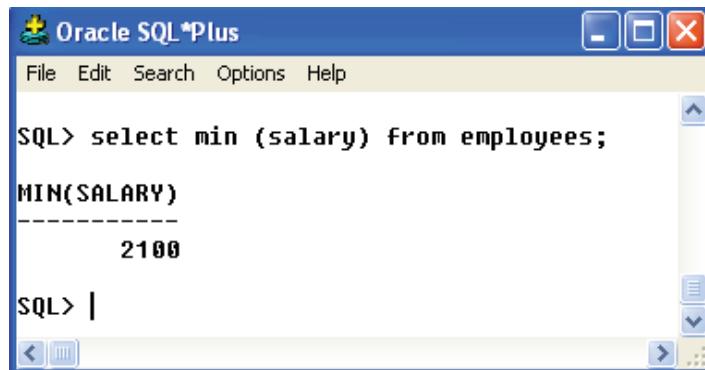
4.16.4 MIN

The MIN() function returns the smallest value of the selected column.

Syntax: `SELECT MIN(column_name) FROM table_name;`

Example: Find the Minimum value of account balance from the account table.

Query: select**min**(salary) from employees;



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following SQL query and its result:

```
SQL> select min (salary) from employees;
MIN(SALARY)
-----
2100
SQL> |
```

Fig. 4.42. Query using *MIN()* function.

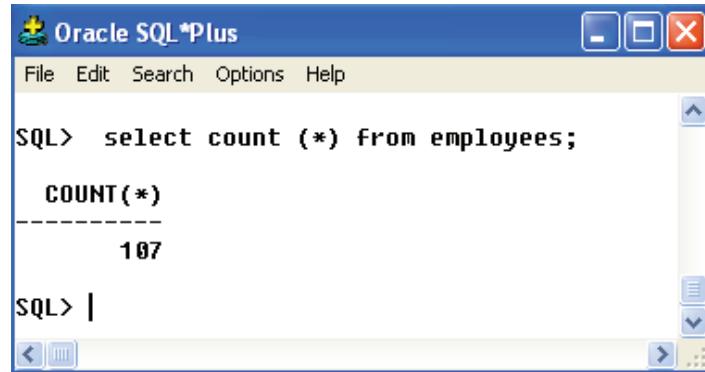
4.16.5 COUNT

The COUNT() function returns the number of rows that matches a specified criteria.

Syntax: *SELECT COUNT(column_name) FROM table_name;*

Example: Find the number rows in the employees table.

Query: *select count (*) from employees;*



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following SQL query and its result:

```
SQL> select count (*) from employees;
COUNT(*)
-----
107
SQL> |
```

Fig. 4.43. Query using *COUNT()* function.

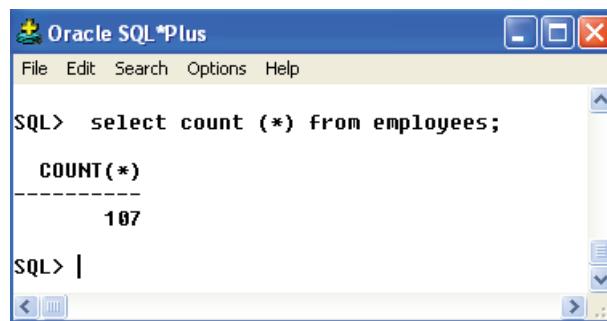
4.16.6 COUNT*

The COUNT(*) function returns the number of records in a table.

Syntax: *SELECT COUNT(*) FROM table_name;*

Example: Find the number of rows in the salary column of employees table.

Query: *select count (salary) from employees;*



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following SQL command and its result:

```
SQL> select count (*) from employees;
      COUNT(*)
      -----
           107
SQL> |
```

Fig. 4.43. Query using COUNT*() function.

4.17 GROUP BY CLAUSE

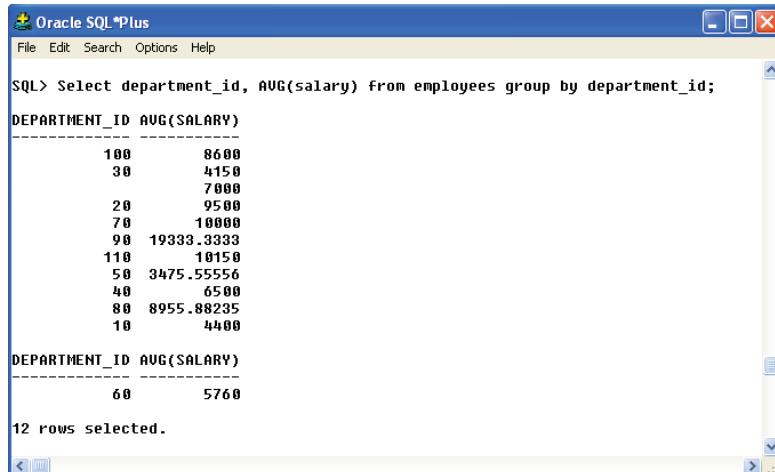
The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

In group function WHERE clause can be used to exclude rows before dividing them into groups. GROUP BY clause should include column names and column alias name cannot be included in the GROUP BY clause. The group by clause should contain all the columns in the select list expect those used along with the group functions.

Syntax:

```
SELECT column1, column2 FROM table_name WHERE [ conditions ]
[GROUP BY column1, column2]
[ORDER BY column1, column2]
```

Example: Select department_id, AVG(salary) from employees group by department_id;



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The main window displays the following SQL command and its result:

```
SQL> Select department_id, AVG(salary) from employees group by department_id;
DEPARTMENT_ID AVG(SALARY)
-----
    100      8600
     30      4150
     70      7000
     20      9500
     70      10000
     90  19333.3333
    110      10150
     50  3475.55556
     40      6500
     80  8955.88235
     10      4400
DEPARTMENT_ID AVG(SALARY)
-----
      60      5760
12 rows selected.
```

Fig. 4.44. SELECT statement with GROUP BY clause.

4.18 HAVING CLAUSE

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions. Having clause is used to filter data based on the group functions. This is similar to WHERE condition but is used with group functions. Group functions cannot be used in WHERE Clause but can be used in HAVING clause.

If you want to select the department that has total salary paid for its employees more than 25000, the sql query would be like;

Query: `select department_id, sum(salary) from employees group by department_id having sum(salary)>25000;`

DEPARTMENT_ID	SUM(SALARY)
100	51600
90	58800
50	156400
80	304500
60	28800

Fig. 4.45. SELECT statement with HAVING clause.

4.19 NESTED SUBQUERIES

A Subquery or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

There are a few rules that subqueries must follow:

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery; however, the BETWEEN operator can be used within the subquery.

4.19.1 Subqueries with the SELECT Statement:

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows:

```
SELECT column_name [, column_name ]
FROM table1 [, table2 ]
WHERE column_name OPERATOR
(SELECT column_name [, column_name ]
FROM table1 [, table2 ]
[WHERE])
```

Example: Consider the employees table

The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The menu bar includes File, Edit, Search, Options, and Help. The main area displays the output of the command "SQL> desc employees;". The table has columns for Name, Null?, and Type. The data is as follows:

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)

Fig. 4.46. Employees table.

Query: select * from employees where employee_id in (select employee_id from employees where salary >4500);

The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The main area displays the output of the query "SQL> select * from employees where employee_id in (select employee_id from employees where salary > 4500);". The data is as follows:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_HAN	13000
100				100			
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000
201				201			

Fig. 4.47. Query depicting subquery concept.

4.19.2 Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified

with any of the character, date or number functions. When adding a new row, you should ensure the datatype of the value and the column matches. The integrity constraints defined for the table should not be affected.

The basic syntax is as follows:

```
INSERT INTO table_name [(column1 [, column2 ])]  
SELECT [*|column1 [, column2 ]]  
FROM table1 [, table2 ]  
[ WHERE VALUE OPERATOR ]
```

Example: Consider a table EMPLOYEES1 with two fields employee_id and department_id. Now to copy the values of employee_id and department_id from EMPLOYEES table into EMPLOYEES1, following query is executed:

Before executing the query create a table EMPLOYEES1 with CREATE table statement and then insert the values.

Query: insert into employees1(employee_id, department_id) select employee_id, department_id from employees;

```
Oracle SQL*Plus
File Edit Search Options Help

SQL> insert into employees1(employee_id, department_id) select employee_id, department_id from employees;
187 rows created.

SQL> select * from employees1;
EMPLOYEE_ID DEPARTMENT_ID
-----
198          50
199          50
200          10
201          20
202          20
203          40
204          70
205          110
206          110
100          90
101          90
```

Fig. 4.48. Subquery with INSERT statement.

4.19.3 Subqueries with the UPDATE Statement

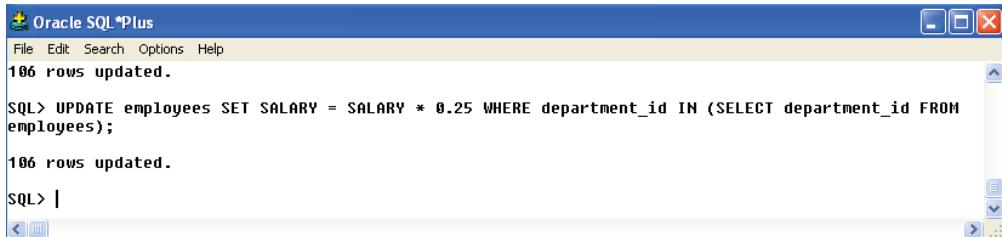
The subquery can be used with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows:

```
UPDATE table  
SET column_name = new_value  
[ WHERE OPERATOR [ VALUE ]  
(SELECT COLUMN_NAME  
FROM TABLE_NAME)  
[ WHERE])]
```

Example:

Query: update employees set salary = salary *0.25 where department_id in (select department_id from employees);



```
Oracle SQL*Plus
File Edit Search Options Help
106 rows updated.

SQL> UPDATE employees SET SALARY = SALARY * 0.25 WHERE department_id IN (SELECT department_id FROM employees);

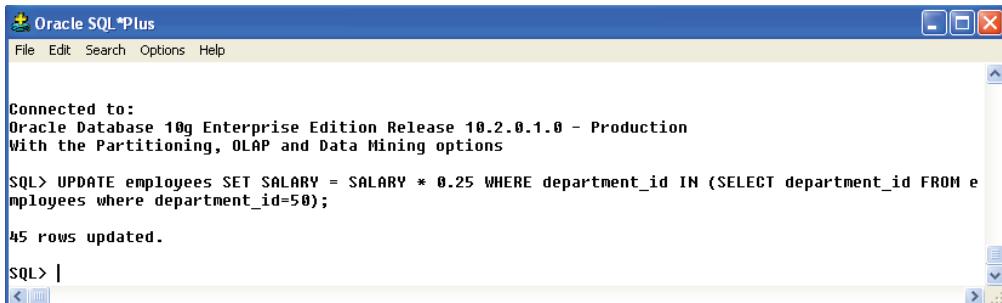
106 rows updated.

SQL> |
```

Fig. 4.49. Subquery with UPDATE statement.

Query: update employees set salary = salary *0.25 where department_id in (select department_id from employees where department_id=50);

This query is same as the previous but checks for a condition in the inner query.



```
Oracle SQL*Plus
File Edit Search Options Help

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options

SQL> UPDATE employees SET SALARY = SALARY * 0.25 WHERE department_id IN (SELECT department_id FROM employees where department_id=50);

45 rows updated.

SQL> |
```

Fig. 4.50. Subquery with UPDATE statement with condition specified in WHERE clause.

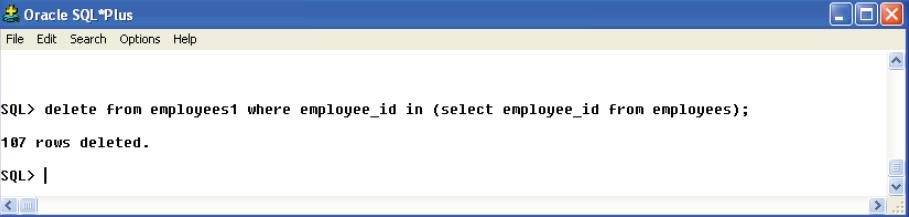
4.19.4 Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
   FROM TABLE_NAME)
  [ WHERE])]
```

Example: delete from employees1 where employee_id in (select employee_id from employees);



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window contains the following SQL command and its execution results:

```
SQL> delete from employees1 where employee_id in (select employee_id from employees);
107 rows deleted.
SQL> |
```

Fig. 4.51. Subquery with *DELETE* statement.

4.20 NULL VALUES

NULL values represent missing unknown data. By default, a table column can hold NULL values. If a column in a table is optional, we can insert a new record or update an existing record without adding a value to this column. This means that the field will be saved with a NULL value.

NULL values are treated differently from other values. NULL is used as a placeholder for unknown or inapplicable or unavailable or unassigned values. It is not possible to compare NULL and 0; they are not equivalent.

4.21 DATABASE OBJECTS

A database object in a relational database is a data structure used to either store or reference data. The most common object that most people interact with is the table.

The database objects include:

- | | | |
|-------------|------------|-------------|
| 1. Tables | 2. Views | 3. Sequence |
| 4. Triggers | 5. Indexes | |

4.21.1 Tables

Table is a relation that is used to store records of related data. It is a logical structure maintained by the database manager. It is made up of columns and rows. At the intersection of every column and row there is a specific data item called a value or field. A base table is created with the CREATE TABLE statement and is used to hold persistent user data. All queries can be performed in a Table.

Creating a basic table involves naming the table and defining its columns and each column's data type. The SQL **CREATE TABLE** statement is used to create a new table.

Syntax: Basic syntax of CREATE TABLE statement is as follows:

```
CREATE TABLE table_name(
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ....
    columnN datatype,
    PRIMARY KEY( one or more columns )
);
```

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement. Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with an example below. A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement.

Example: Following is an example, which creates a CUSTOMERS table with ID as primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table:

```
SQL> CREATE TABLE CUSTOMERS (
    ID INT      NOT NULL,
    NAME VARCHAR (20) NOT NULL,
    AGE INT      NOT NULL,
    ADDRESS CHAR (25),
    SALARY DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

The success of the created table can be verified by looking at the message displayed by the SQL server, otherwise you can use **DESC** command as follows:

```
SQL> DESC CUSTOMERS;
```

Field	Type	Null	Key	Default	Extra
ID	int(11)	NO	PRI		
NAME	varchar(20)	NO			
AGE	int(11)	NO			
ADDRESS	char(25)	YES		NULL	
SALARY	decimal(18,2)	YES		NULL	

Now, the CUSTOMERS table is available in the database which you can used to store required information related to customers.

4.21.2 Views

VIEW is a stored SQL query used as a “Virtual table” that logically represents subsets of data from one or more tables. It provides an alternative way of looking at the data in one or more tables. A view is a logical table based on a table or another view. The table on which view is based is called a base table.

It is a named specification of a result table. The specification is a SELECT statement that is executed whenever the view is referenced in an SQL statement. Consider a view to have columns and rows just like a base table. For retrieval, all views can be used just like base tables.

When the column of a view is directly derived from the column of a base table, that column inherits any constraints that apply to the column of the base table. For example, if a view includes a foreign key of its base table, INSERT and UPDATE operations using that view are subject to

the same referential constraints as the base table. Also, if the base table of a view is a parent table, DELETE and UPDATE operations using that view are subject to the same rule as DELETE and UPDATE operations on the base table.

4.21.2.1 Advantages of View

- **Restricts data access:** views restrict access to the data because the view can display selected columns from the table.
- **Makes complex query easy:** view can be used to make simple queries to retrieve the results of complicated queries. View can be used to query information from multiple tables without the user knowing how to write a join statement.
- **To present different views of the same data:** view provides data independence for ad-hoc users and application programs. One view can be used to retrieve data from several tables.
- **To provide data independence:** Views provide groups of users' access to data according to their particular criteria.

4.21.2.2 Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables, or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows:

```
CREATE VIEW view_name AS
    SELECT column_name(s)
        FROM table_name
        WHERE condition
```

Example: Consider the CUSTOMERS table having the following records:

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Query to create View

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
    SELECT name, age
        FROM CUSTOMERS;
```

When the above given query is executed View will be created in the name CUSTOMERS_VIEW. Now, we can query CUSTOMERS_VIEW in similar way as you query an actual table.

SQL > SELECT * FROM CUSTOMERS_VIEW;

This query would produce the following output

Name	Age
Ramesh	32
Khilan	25
Kaushik	23
Chaitali	25
Hardik	27
Komal	22
Muffy	24

4.21.2.3 The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition. If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following is an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION:

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should disallow the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

4.21.2.4 Updating a View

A view can be updated under certain conditions:

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.

- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the above-mentioned rules then you can update a view. Following is an example to update the age of Ramesh:

```
SQL > UPDATE CUSTOMERS_VIEW
```

```
SET AGE =35
```

```
WHERE name='Ramesh';
```

This would ultimately update the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

4.21.2.5 Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command. Here we cannot insert rows in CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in similar way as you insert them in a table.

4.21.2.6 Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE= 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW
```

```
WHERE age =22;
```

This would ultimately delete a row from the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Muffy	24	Indore	10000.00

4.21.2.7 Dropping Views

The view can be dropped if the view is no longer needed. The syntax is very simple as given below:

DROP VIEW view_name;

Following is an example to drop CUSTOMERS_VIEW from CUSTOMERS table:

DROP VIEW CUSTOMERS_VIEW;

When the above query is executed the view will be no longer present.

4.21.3 Sequences

A sequence is a database object that generates numbers in sequential order. Applications most often use these numbers when they require a unique value in a table such as primary key values. Some database management systems use an “auto number” concept or “auto increment” setting on numeric column types. Both the auto numbering columns and sequences provide a unique number in sequence used for a unique identifier. The following list describes the characteristics of sequences:

- Sequences are available to all users of the database
- Sequences are created using SQL statements
- Sequences have a minimum and maximum value (the defaults are minimum=0 and maximum= $2^{63}-1$); they can be dropped, but not reset
- Once a sequence returns a value, the sequence can never return that same value
- While sequence values are not tied to any particular table, a sequence is usually used to generate values for only one table
- Sequences increment by an amount specified when created (the default is 1)

4.21.3.1 Creating a Sequence

To create sequences, execute a CREATE SEQUENCE statement in the same way as an UPDATE or INSERT statement. The sequence information is stored in a data dictionary file in the same location as the rest of the data dictionary files for the database. If the data dictionary file does not exist, the SQL engine creates the file when it creates the first sequence.

The format for a CREATE SEQUENCE statement is as follows:

*CREATE SEQUENCE sequence_name
[INCREMENT BY #]
[START WITH #]
[MAXVALUE # | NOMAXVALUE]
[MINVALUE # | NOMINVALUE]
[CYCLE | NOCYCLE]*

Variable	Description
INCREMENT BY	The increment value. This can be a positive or negative number.
START WITH	The start value for the sequence.
MAXVALUE	The maximum value that the sequence can generate. If specifying NOMAXVALUE, the maximum value is $2^{63}-1$.
MINVALUE	The minimum value that the sequence can generate. If specifying NOMINVALUE, the minimum value is -2^{63} .
CYCLE	Specify CYCLE to indicate that when the maximum value is reached the sequence starts over again at the start value. Specify NOCYCLE to generate an error upon reaching the maximum value.

4.21.3.2 Dropping a Sequence

To drop a sequence, execute a DROP SEQUENCE statement. Use this function when a sequence is no longer useful, or to reset a sequence to an older number. To reset a sequence, first drop the sequence and then recreate it. Drop a sequence following this format:

Syntax: *DROP SEQUENCE sequence_name;*

Example: `DROP SEQUENCE my_sequence`

4.21.3.3 Using a Sequence

Use sequences when an application requires a unique identifier. INSERT statements, and occasionally UPDATE statements, are the most common places to use sequences. Two “functions” are available on sequences:

NEXTVAL: Returns the next value from the sequence.

CURVAL: Returns the value from the last call to NEXTVAL by the current user during the current connection. For example, if User A calls NEXTVAL and it returns 124, and User B immediately calls NEXTVAL getting 125, User A will get 124 when calling CURVAL, while User B will get 125 while calling CURVAL. It is important to understand the connection between the sequence value and a particular connection to the database. The user cannot call CURVAL until making a call to NEXTVAL at least once on the connection. CURVAL returns the current value returned from the sequence on the current connection, not the current value of the sequence.

Examples: To create the sequence:

CREATE SEQUENCE customer_seq INCREMENT BY 1 START WITH 100

To use the sequence to enter a record into the database:

```
INSERT INTO customer (cust_num, name, address)
VALUES (customer_seq.NEXTVAL,'John Doe','123 Main St.')
```

To find the value just entered into the database:

```
SELECT customer_seq.CURVAL AS LAST_CUST_NUM
```

4.21.4 Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur or some condition is met. Triggers are written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

4.21.4.1 Benefits of Triggers

Triggers can be written for the following purposes:

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

4.21.4.2 Creating Triggers

The syntax for creating a trigger is:

```
CREATE[OR REPLACE]TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT[OR]|UPDATE[OR]|DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN(condition)
DECLARE
    Declaration-statements
BEGIN
    Executable-statements
EXCEPTION
    Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name: Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF}: This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for the rows which the trigger would fire. This clause is valid only for row level triggers.

4.21.4.3 Types of Triggers

There are two types of triggers based on which level it is triggered.

1. **Row Level Trigger:** An event is triggered for each row updated, inserted or deleted.
2. **Statement Level Trigger:** An event is triggered for each SQL statement executed.

4.21.4.4 Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

1. BEFORE statement trigger fires first.
2. Next BEFORE row level trigger fires, once for each row affected.
3. Then AFTER row level trigger fires once for each affected row. These events will alternates between BEFORE and AFTER row level triggers.
4. Finally the AFTER statement level trigger fires.

Example: To start with, we will be using the CUSTOMERS table below

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

The following program creates a **row level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATEOR REPLACE TRIGGER display_salary_changes
BEFORE DELETEORINSERTORUPDATEON customers
FOR EACH ROW
WHEN(NEW.ID >0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff :=:NEW.salary -:OLD.salary;
    dbms_output.put_line('Old salary: '||:OLD.salary);
    dbms_output.put_line('New salary: '||:NEW.salary);
    dbms_output.put_line("Salary difference: "|| sal_diff);
END;
```

When the above code is executed at SQL prompt, it produces the following result:

Trigger created.

Here following two points are important and should be noted carefully:

- OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

4.21.4.5 Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table:

```
INSERTINTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES(7,'Kriti',22,'HP',7500.00);
```

When a record is created in CUSTOMERS table, above create trigger **display_salary_changes** will be fired and it will display the following result:

Old salary:

New salary: 7500

Salary difference:

Because this is a new record, old salary is not available and above result is displayed as null. Now, let us perform one more DML operation on the CUSTOMERS table. Here is one UPDATE statement, which will update an existing record in the table:

```
UPDATE customers SET salary = salary +500  
WHERE id =2;
```

When a record is updated in CUSTOMERS table, above create trigger **display_salary_changes** will be fired and it will display the following result:

Old salary: 1500
New salary: 2000
Salary difference: 500

4.21.4.6 Disabling Triggers

Disabling trigger statement helps in disabling the created triggers

Syntax: *Alter trigger <trigger_name> disable;*

Example: Alter trigger display_salary_changes disable;

Specific triggers on a table can be disabled as follows

Syntax: *Alter table <table_name> disable <trigger_name>;*

Example: Alter table customers disable display_salary_changes;

All triggers on a table can be disabled on a table as follows.

Syntax: *Alter table <table_name> disable all triggers;*

Example: Alter table customers disables all triggers;

4.21.4.7 Enabling Trigger

Enabling trigger statement helps in enabling the created triggers

Syntax: *Alter table <table_name> enable trigger_name;*

Example: Alter table customers enable display_salary_changes;

To enable all triggers

Syntax :*Alter table <table_name> enable all triggers;*

4.21.4.8 Dropping triggers

This statement helps to drop the created trigger. Once this statement is executed the trigger created will be no longer be present.

Syntax: *Drop trigger <trigger_name>;*

Example: Drop trigger display_salary_changes;

4.21.5 Indexes

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. An index is a pointer to data in a table basically used for performance tuning and fast retrieval. An index in a database is very similar to an index in the back of a book.

For example, if you want to refer all pages in a book that discuss a certain topic, you first refer to the index which lists all topics alphabetically and are then referred to one or more specific page numbers.

An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

4.21.5.1 The **CREATE INDEX** Command

The basic syntax of CREATE INDEX is as follows:

CREATE INDEX index_name ON table_name;

Single-Column Indexes: A single-column index is one that is created based on only one table column. The basic syntax is as follows:

*CREATE INDEX index_name
ON table_name (column_name);*

Unique Indexes: Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

*CREATE INDEX index_name
on table_name (column_name);*

Composite Indexes: A composite index is an index on two or more columns of a table. The basic syntax is as follows:

*CREATE INDEX index_name
on table_name (column1, column2);*

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions. If only one column is frequently used, then a single-column index can be used. Whereas if two or more columns are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes: Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

4.21.5.2 The **DROP INDEX** Command

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.

The basic syntax is as follows:

```
DROP INDEX index_name;
```

4.21.5.3 When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered:

- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

4.22 DCL (DATA CONTROL LANGUAGE)

DCL commands are used to enforce database security in a multiple user database environment. Two types of DCL commands are

1. GRANT
2. REVOKE.

Only Database Administrator's or owner's of the database object can provide/remove privileges on a database object.

4.22.1 GRANT Command

SQL GRANT is a command used to provide access or privileges on the database objects to the users.

The Syntax for the GRANT command is:

```
GRANT privilege_name  
    ON object_name  
    TO {user_name | PUBLIC | role_name}  
    [WITH GRANT OPTION];
```

- **privilege_name** is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
- **object_name** is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
- **user_name** is the name of the user to whom an access right is being granted.
- **PUBLIC** is used to grant access rights to all users.
- **ROLES** are a set of privileges grouped together.
- **WITH GRANT OPTION** - allows a user to grant access rights to other users.

For Example:

```
GRANT SELECT ON employee TO user1;
```

This command grants a SELECT permission on employee table to user1. You should be cautious in using the WITH GRANT option because, for example, if you GRANT SELECT privilege on employee table to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from user1, still user2 will have SELECT privilege on employee table.

4.22.2 REVOKE Command

The REVOKE command removes user access rights or privileges to the database objects.

The Syntax for the REVOKE command is:

```
REVOKE privilege_name  
ON object_name  
FROM {user_name |PUBLIC |role_name}
```

For Example:

```
REVOKE SELECT ON employee FROM user1;
```

This command will REVOKE a SELECT privilege on employee table from user1. When you REVOKE SELECT privilege on a table from a user, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it. You cannot REVOKE privileges if they were not initially granted by you.

4.22.3 Privileges and Roles

Privileges: Privileges define the access rights provided to a user on a database object. There are two types of privileges.

1. **System privileges:** This allows the user to CREATE, ALTER, or DROP database objects.
2. **Object privileges:** This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

Few CREATE system privileges are listed below:

<i>System Privileges</i>	<i>Description</i>
CREATE object	Allows users to create the specified object in their own schema.
CREATE ANY object	Allows users to create the specified object in any schema.

The above rules also apply for ALTER and DROP system privileges.

Few of the object privileges are listed below:

<i>Object Privileges</i>	<i>Description</i>
INSERT	Allows users to insert rows into a table.
SELECT	Allows users to select data from a database object.
UPDATE	Allows user to update data in a table.
EXECUTE	Allows user to execute a stored procedure or a function.

Roles: Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges. You can either create Roles or use the system roles pre-defined by oracle.

Some of the privileges granted to the system roles are as given below:

System Role	Privileges Granted to the Role
CONNECT	CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc.
RESOURCE	CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects.
DBA	ALL SYSTEM PRIVILEGES

4.22.4 Creating Roles

The Syntax to create a role is:

```
CREATE ROLE role_name
[IDENTIFIED BY password];
```

For Example: To create a role called “testing” with password as “pwd”, the code will be as follows:

```
CREATE ROLE testing
[IDENTIFIED BY pwd];
```

It’s easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user. If a role is identified by a password, then, when you GRANT or REVOKE privileges to the role, you definitely have to identify it with the password.

We can GRANT or REVOKE privilege to a role as below.

For example: To grant CREATE TABLE privilege to a user by creating a testing role:

First, create a testing Role

```
CREATE ROLE testing
```

Second, grant a CREATE TABLE privilege to the ROLE testing. You can add more privileges to the ROLE.

```
GRANT CREATE TABLE TO testing;
```

Third, grant the role to a user.

```
GRANT testing TO user1;
```

To revoke a CREATE TABLE privilege from testing ROLE, you can write:

```
REVOKE CREATE TABLE FROM testing;
```

The Syntax to drop a role from the database is as below:

```
DROP ROLE role_name;
```

For example: To drop a role called testing, you can write:

```
DROP ROLE testing;
```

4.23 TCL (TRANSACTION CONTROL LANGUAGE) ---

TCL statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

- COMMIT: save work done
- SAVEPOINT: identify a point in a transaction to which you can later roll back
- ROLLBACK: restore database to original since the last COMMIT
- SET TRANSACTION: Change transaction options like isolation level and what rollback segment to use

Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

4.23.1 The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

The **syntax** for COMMIT command is as follows:

```
COMMIT;
```

Example: Consider the CUSTOMERS table having the following records:

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Following is the example which would delete records from the table having age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS WHERE AGE =25;
SQL> COMMIT;
```

As a result, two rows from the table would be deleted and SELECT statement would produce the following result:

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
1	Ramesh	32	Ahmedabad	2000.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

4.23.2 The ROLLBACK Command:

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for ROLLBACK command is as follows:

ROLLBACK;

Example: Consider the CUSTOMERS table having the following records:

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Following is the example, which would delete records from the table having age = 25 and then ROLLBACK the changes in the database.

SQL> DELETE FROM CUSTOMERS WHERE AGE =25;

SQL> ROLLBACK;

As a result, delete operation would not impact the table and SELECT statement would produce the following result:

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

4.23.3 The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The **syntax** for SAVEPOINT command is as follows:

SAVEPOINT SAVEPOINT_NAME;

This command serves only in the creation of a SAVEPOINT among transactional statements. The ROLLBACK command is used to undo a group of transactions.

The **syntax** for rolling back to a SAVEPOINT is as follows:

ROLLBACK TO SAVEPOINT_NAME;

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state:

Example: Consider the CUSTOMERS table having the following records:

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Now, here is the series of operations:

SQL> SAVEPOINT SP1;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;

1 row deleted.

SQL> SAVEPOINT SP2;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;

1 row deleted.

SQL> SAVEPOINT SP3;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=3;

1 row deleted.

Now that the three deletions have taken place, say you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone:

SQL> ROLLBACK TO SP2;

Rollback complete.

Notice that only the first deletion took place since you rolled back to SP2:

SQL> SELECT * FROM CUSTOMERS;

<i>Id</i>	<i>Name</i>	<i>Age</i>	<i>Address</i>	<i>Salary</i>
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

6 rows selected.

4.23.4 The RELEASE SAVEPOINT Command

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The **syntax** for RELEASE SAVEPOINT is as follows:

RELEASE SAVEPOINT SAVEPOINT_NAME;

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the SAVEPOINT.

4.23.5 The SET TRANSACTION Command

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows.

For example, you can specify a transaction to be read only, or read write.

The **syntax** for SET TRANSACTION is as follows:

SET TRANSACTION [READ WRITE | READ ONLY];

REVIEW QUESTIONS

1. Define SQL.
2. Give the basic structure of SQL expression.
3. List out the different types of SQL statements.
4. Define the following and list out the comments under each of the following.
 - (i) DDL
 - (ii) DML

- (iii) DCL
 - (iv) TCL
5. List out the advantages of using SQL statements.
 6. Define database objects.
 7. What is need of HAVING clause in SQL?
 8. Define triggers.
 9. What is the use of sequence in SQL?
 10. List out the different database objects and define each of them.

Queries

1. Create a table with following table and name it as ACCOUNTS.

Name	Type
acc_no	varchar (5)
branch_name	varchar (12)
balance	number(7)

- A. Write a query to describe the structure of the ACCOUNTS table.
 - B. Write a query to determine the contents of the ACCOUNTS table.
 - C. Write a query to display the branch_name.
 - D. Find branch_name having balance>30,000.
 - E. Change the data type of acc_no to number(5).
 - F. Include a new column branch_id to the ACCOUNTS table.
 - G. Change the size of balance field from 7 to 10.
 - H. Truncate the table.
 - I. Drop the created table.
2. Create the following table and insert the given values into the table:

Customer Table:

cid	Cust_name	Cust_street	Cust_city
1	Jones	Main	Harrison
2	Smith	North	Rye
3	Hays	Main	Princeton
4	Curry	North	Pittsfield
5	Lindsay	Park	Palo Alto

For the questions below consider the employees table used in chapter 4

3. Display the last name concatenated with job_id and name the column Employee and Title.
4. Display the last_name and salary of employees in employees table who earn more than 10,000.

5. Display the last_name and department number for employee having department_id=100.
6. Display the names of all the employees whose salary lies between 15,000 and 25,000.
7. Find the names of employees whose employee_id is either 100, 102 or 104.
8. Display the names and id of all employees whose name starts with letter ‘A’.
9. Display all employees last name in which the third letter of the name is “a.”.
10. Display the last name of all employees who have both a and s in their last name.
11. Create a view for employees table and name it as Employees1.
12. Copy the values present in the employees table to the view Employees1 using subquery concept.
13. Create a report to display all details of employees who earn More than \$12,000.
14. Display the last_name and salary for any employee whose salary is not in \$5000to \$12000.
15. Find the Average, minimum and maximum salary of the employees in the employee table.
16. Fine the average salary of the employees in the employee table group by department_id.
17. Fine the minimum salary of the employees in the employee table group by department_id and check that the minimum salary does not fall below 1000.
18. Consider the employees table and departments table from our 4th chapter and perform all the set operations.
19. Create the following Library and summary tables.

Library table:

Name	Type
BNO	NUMBER(1)
BNAME	VARCHAR2(20)
NCOPY	NUMBER(2)

Summary Table:

Name	Type
BNO	NUMBER(1)
NCOPY	NUMBER(2)

Inert values in to the library table and write a trigger such that BNO and NCOPY are automatically updated in the summary table.

20. Create sequenceseq1 increment by 1 start with 100 maxvalue 105 and invalue 100.



Chapter 5

Joins, Constraints and Advanced SQL

5.1 JOINS

The SQL Joins clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

The Syntax for joining two tables is:

```
SELECT col1, col2, col3...
  FROM table_name1, table_name2
 WHERE table_name1.col2 = table_name2.col1;
```

Consider the following two tables;

(a) CUSTOMER and the values in the CUSTOMER table is as follows:

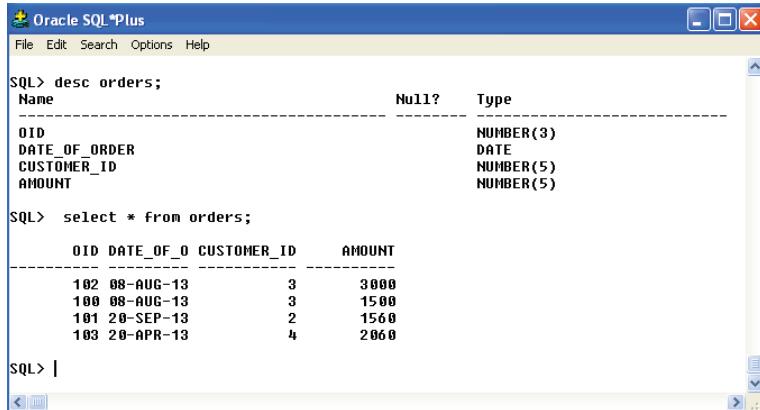
The screenshot shows the Oracle SQL*Plus interface. At the top, there's a menu bar with File, Edit, Search, Options, Help. Below it, the SQL prompt shows two commands: 'desc customer;' and 'select * from customer;'. The 'desc customer;' command displays the structure of the table with columns ID, NAME, AGE, ADDRESS, and SALARY, each with their respective data types (NUMBER(5), VARCHAR2(10), NUMBER(3), VARCHAR2(10), and NUMBER(5)). The 'select * from customer;' command displays the data for 7 rows, with columns ID, NAME, AGE, ADDRESS, and SALARY. The data is as follows:

ID	NAME	AGE	ADDRESS	SALARY
1	ramesh	32	Ahmedabad	2000
2	Akhilan	25	delhi	1500
3	kaushik	23	kota	2000
4	ravi	25	mumbai	6500
5	hardik	27	bhopal	8500
6	komal	22	MP	4500
7	sethu	24	tamilnadu	10000

7 rows selected.

Fig. 5.1. Customer table.

(b) Another table is ORDERS and its values are as follows:



The screenshot shows the Oracle SQL*Plus interface. The command `desc orders;` is run, displaying the structure of the table with columns: Name, Null?, and Type. The table has four columns: OID (NUMBER(3)), DATE_OF_ORDER (DATE), CUSTOMER_ID (NUMBER(5)), and AMOUNT (NUMBER(5)). The command `select * from orders;` is run, displaying the data with columns: OID, DATE_OF_O, CUSTOMER_ID, and AMOUNT. The data is as follows:

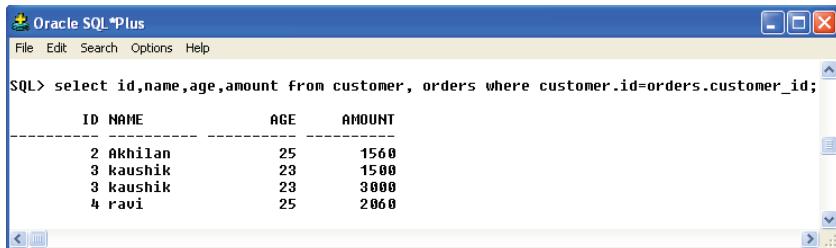
OID	DATE_OF_O	CUSTOMER_ID	AMOUNT
102	08-AUG-13	3	3000
100	08-AUG-13	3	1500
101	20-SEP-13	2	1560
103	20-APR-13	4	2060

Fig. 5.2. Orders table.

Now, let us join these two tables in our SELECT statement as follows:

```
SQL> SELECT ID, NAME, AGE, AMOUNT
      FROM CUSTOMERS, ORDERS
     WHERE CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:



The screenshot shows the Oracle SQL*Plus interface. The command `select id,name,age,amount from customer, orders where customer.id=orders.customer_id;` is run, displaying the joined data with columns: ID, NAME, AGE, and AMOUNT. The data is as follows:

ID	NAME	AGE	AMOUNT
2	Akhilan	25	1560
3	kaushik	23	1500
3	kaushik	23	3000
4	ravi	25	2060

Fig. 5.3. Join operation.

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as `=`, `<`, `>`, `<>`, `<=`, `>=`, `!=`, `BETWEEN`, `LIKE`, and `NOT`; they can all be used to join tables. However, the most common operator is the equal symbol.

5.1.1 Joins Types

There are different types of joins available in SQL:

- **INNER JOIN:** returns rows when there is a match in both tables.
- **LEFT JOIN:** returns all rows from the left table, even if there are no matches in the right table.
- **RIGHT JOIN:** returns all rows from the right table, even if there are no matches in the left table.

- **FULL JOIN:** returns rows when there is a match in one of the tables.
- **SELF JOIN:** is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- **CARTESIAN JOIN:** returns the Cartesian product of the sets of records from the two or more joined tables.

5.1.1.1 Inner Join

The most commonly used and important of the joins is the INNER JOIN. It is also referred to as an EQUIJOIN. The INNER JOIN creates a new result table by combining column values of two tables based upon the join-condition. The query compares each row of the first table with each row of second table to be joint together to find all pairs of rows which satisfy the join-condition. When the join-condition is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax:

The basic syntax of **INNER JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

Now, let us join these two tables using INNER JOIN as follows:

```
SQL> SELECT ID, NAME, AMOUNT, DATE_OF_ORDER
  FROM CUSTOMER
INNER JOIN ORDERS
  ON CUSTOMER.ID = ORDERS.CUSTOMER_ID
```

ID	NAME	AMOUNT	DATE_OF_O
2	Akhilan	1560	20-SEP-13
3	Kaushik	1500	08-AUG-13
3	Kaushik	3000	08-AUG-13
4	Ravi	2060	20-APR-13

Fig. 5.4. Inner Join operation.

5.1.1.2 Left Join

The SQL LEFT JOIN returns all rows from the left table, even if there are no matches in the right table. This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join condition.

Syntax:

The basic syntax of LEFT JOIN is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

Consider the following example

```
SQL> SELECT ID, NAME, AMOUNT, DATE_OF_ORDER
      FROM CUSTOMER
      LEFT JOIN ORDERS
      ON CUSTOMER.ID = ORDERS.CUSTOMER_ID;
```

ID	NAME	AMOUNT	DATE_OF_O
3	kaushik	3000	08-AUG-13
3	kaushik	1500	08-AUG-13
2	Akhilan	1560	20-SEP-13
4	ravi	2060	20-APR-13
5	hardik		
1	ramesh		
6	komal		
7	sethu		

8 rows selected.

SQL>

Fig. 5.5. Left Join operation.

5.1.1.3 Right Join

The SQL RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table. This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join condition.

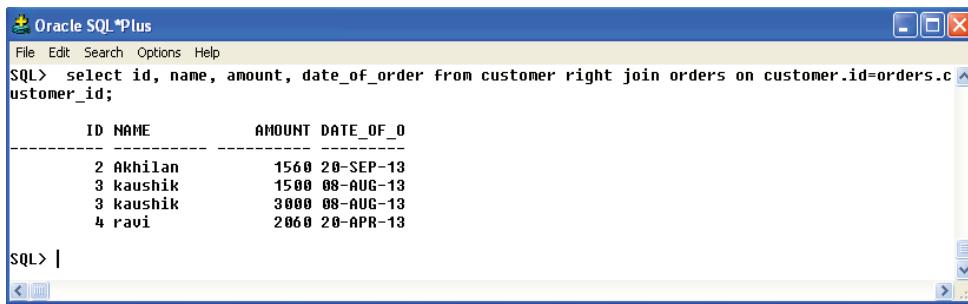
Syntax:

The basic syntax of **RIGHT JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

Consider the following example

```
SQL> SELECT ID, NAME, AMOUNT, DATE_OF_ORDER
      FROM CUSTOMER
      RIGHT JOIN ORDERS
      ON CUSTOMER.ID = ORDERS.CUSTOMER_ID;
```



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> select id, name, amount, date_of_order from customer right join orders on customer.id=orders.customer_id;
```

The resulting output is:

ID	NAME	AMOUNT	DATE_OF_O
2	Akhilan	1560	20-SEP-13
3	kaushik	1500	08-AUG-13
3	kaushik	3000	08-AUG-13
4	ravi	2060	20-APR-13

Fig. 5.6. Right Join operation.

5.1.1.4 Full Join

The SQL FULL JOIN combines the results of both left and right outer joins. The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

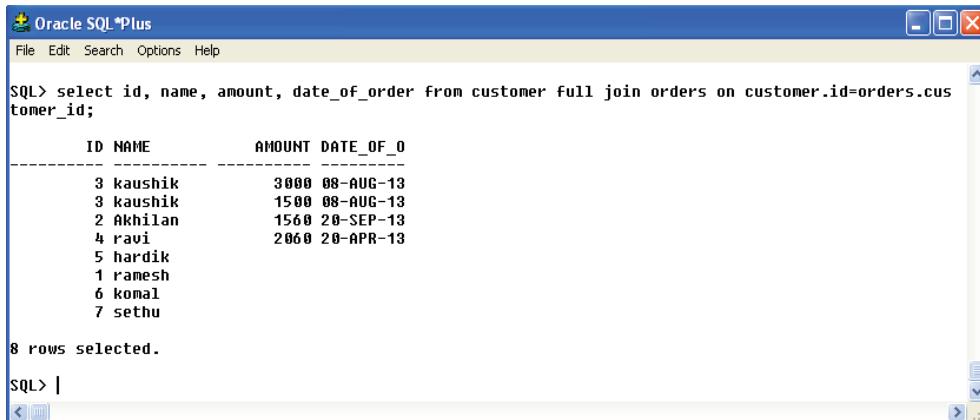
Syntax:

The basic syntax of FULL JOIN is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

Consider the following example

```
SQL> SELECT ID, NAME, AMOUNT, DATE_OF_ORDER
      FROM CUSTOMER
      FULL JOIN ORDERS
      ON CUSTOMER.ID = ORDERS.CUSTOMER_ID;
```



The screenshot shows the Oracle SQL*Plus interface. The command entered is:

```
SQL> select id, name, amount, date_of_order from customer full join orders on customer.id=orders.customer_id;
```

The resulting output is:

ID	NAME	AMOUNT	DATE_OF_O
3	kaushik	3000	08-AUG-13
3	kaushik	1500	08-AUG-13
2	Akhilan	1560	20-SEP-13
4	ravi	2060	20-APR-13
5	hardik		
1	rameesh		
6	komal		
7	sethu		

8 rows selected.

Fig. 5.7. Full Join operation.

5.1.1.5 Self Join

The SQL SELF JOIN is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

Syntax:

The basic syntax of SELF JOIN is as follows:

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

Here, WHERE clause could be any given expression based on your requirement.

Consider the following example

```
SQL> SELECT a.ID, b.NAME, a.SALARY
      FROM CUSTOMER a, CUSTOMER b
     WHERE a.SALARY < b.SALARY;
```

ID	NAME	SALARY
2	ramesh	1500
2	kaushik	1500
2	komal	1500
2	ravi	1500
2	hardik	1500
2	sethu	1500
1	komal	2000
1	ravi	2000
1	hardik	2000
1	sethu	2000
3	komal	2000

ID	NAME	SALARY
3	ravi	2000
3	hardik	2000
3	sethu	2000
6	ravi	4500
6	hardik	4500
6	sethu	4500
4	hardik	6500
4	sethu	6500
5	sethu	8500

20 rows selected.

Fig. 5.8. Self Join operation.

5.1.1.6 Cartesian Join

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from the two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to True or where the join-condition is absent from the statement.

Syntax:

The basic syntax of INNER JOIN is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1, table2 [, table3 ]
```

Consider the following example

```
SQL> SELECT ID, NAME, AMOUNT, DATE_OF_ORDER
  FROM CUSTOMER, ORDERS;
```

The screenshot shows the Oracle SQL*Plus interface with three distinct result sets displayed vertically. Each set has columns: ID, NAME, AMOUNT, and DATE_OF_O_0.

ID	NAME	AMOUNT	DATE_OF_O_0
1	ramesh	3000	08-AUG-13
2	Akhilan	3000	08-AUG-13
3	kaushik	3000	08-AUG-13
4	ravi	3000	08-AUG-13
5	hardik	3000	08-AUG-13
6	komal	3000	08-AUG-13
7	sethu	3000	08-AUG-13
1	ramesh	1500	08-AUG-13
2	Akhilan	1500	08-AUG-13
3	kaushik	1500	08-AUG-13
4	ravi	1500	08-AUG-13
5	hardik	1500	08-AUG-13
6	komal	1500	08-AUG-13
7	sethu	1500	08-AUG-13
1	ramesh	1560	20-SEP-13
2	Akhilan	1560	20-SEP-13
3	kaushik	1560	20-SEP-13
4	ravi	1560	20-SEP-13
5	hardik	1560	20-SEP-13
6	komal	1560	20-SEP-13
7	sethu	1560	20-SEP-13
1	ramesh	2060	20-APR-13
2	Akhilan	2060	20-APR-13
3	kaushik	2060	20-APR-13
4	ravi	2060	20-APR-13
5	hardik	2060	20-APR-13
6	komal	2060	20-APR-13
7	sethu	2060	20-APR-13

28 rows selected.

Fig. 5.9. Cartesian Join operation.

5.2 CONSTRAINTS

Constraints are used to specify rules for the data in a table. Before one can start to implement the database tables, one must define the integrity constraints. Integrity means something like 'be right' and consistent. This constraint ensures whether the data in a database is right and is in good condition. Constraints also provide mechanism to prevent invalid data entry in the table and accidental damages to the databases.

Constraints can be defined in two ways

1. The constraints can be specified immediately after the column definition. This is called column-level definition. i.e., constraints specified when the table is created (inside the create table statement)
2. The constraints can be specified after all the columns are defined. This is called table-level definition. i.e., after the table is created (inside the alter table statement)

General syntax for creating constraints

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

Dropping Constraints

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

Syntax:

```
ALTER TABLE TABLE_NAME DROP CONSTRAINT CONSTRAINT_NAME;
```

5.2.1 Types of Constraints

The constraints fall under three different categories

1. Domain integrity Constraints
2. Entity integrity Constraints
3. Referential integrity Constraints

5.2.1.1 DOMAIN Integrity Constraints

A domain is a set of values that may be assigned to an attribute. All values that appear in a column of a relation (table) must be taken from the same domain. Domain restricts the values of attributes in the relation and it is a constraint of the relational model. We need more specific ways to state what data values are not allowed and what format is suitable for an attributes. The SQL constraints that fall under this category are

- **Not null:** indicates that a column cannot store null value
- **Check:** ensures that the value in a column meets a specific condition

5.2.1.1.1 Not NULL Constraints

By default, a column can hold NULL values. If we do not want a column to have a NULL value, then we have to define NOT NULL constraint on that particular column or columns. This constraint will not allow NULL values for the specified column. A NULL is not the same as no data, rather, it represents unknown data.

Syntax to define a Not Null constraint:

```
CREATE TABLE<TABLE NAME>(COLUMNNAME DATA TYPE (SIZE)
CONSTRAINT CONSTRAINT_NAME NOT NULL);
```

Example: For example, the following SQL creates a new table called CUSTOMERS and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

```
CREATE TABLE CUSTOMERS (
    ID INT NOT NULL,
    NAME VARCHAR (20) NOT NULL,
    AGE INT NOT NULL,
    ADDRESS CHAR (25),
    SALARY DECIMAL (18, 2),
    KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to SALARY column, the following statement is used:

```
ALTER TABLE CUSTOMERS
MODIFY SALARY DECIMAL (18, 2) NOT NULL;
```

5.2.1.1.2 Check Constraints

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered into the table.

Syntax to define a Check constraint:

```
CREATE TABLE <TABLE NAME>(COLUMNNAME DATA TYPE (SIZE)
CONSTRAINT CONSTRAINT_NAMECHECK(CHECK_CONDITION));
```

Example: For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you cannot have any CUSTOMER below 18 years:

```
CREATE TABLE CUSTOMERS (
    ID INT NOT NULL,
    NAME VARCHAR (20) NOT NULL,
    AGE INT NOT NULL CHECK (AGE >= 18),
    ADDRESS CHAR (25),
    SALARY DECIMAL (18, 2),
    KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
MODIFY AGE INT NOT NULL CHECK (AGE >= 18);
```

You can also use following syntax, which supports naming the constraint in multiple columns as well:

```
ALTER TABLE CUSTOMERS
ADD CONSTRAINT myCheckConstraint CHECK (AGE >= 18);
```

DROP a CHECK Constraint:

To drop a CHECK constraint, use the following SQL.

```
ALTER TABLE CUSTOMERS
DROP CONSTRAINT myCheckConstraint;
```

5.2.1.2 Entity Integrity Constraints

The entity integrity constraints state that no primary key value can be null. This is because the primary key value is used to identify individual tuples in a relation. The SQL constraints that fall under this category are

- **Unique:** ensures that each row must have a unique value
- **Primary key:** a combination of a not null and unique.

5.2.1.2.1 Unique Constraints

This constraint ensures that a column or a group of columns in each row have a distinct value. A column(s) can have a null value but the values cannot be duplicated.

Syntax for unique constraints:

```
CREATE TABLE <TABLE NAME>(COLUMNNAME DATA TYPE (SIZE)
CONSTRAINT CONSTRAINT_NAME UNIQUE);
```

Assigning unique constraints for multiple columns is called composite unique constraints. The syntax is as follows

```
CREATE TABLE <TABLE NAME>(COLUMNNAME1 DATA TYPE (SIZE),
COLUMNNAME2 DATA TYPE (SIZE), CONSTRAINT CONSTRAINT_NAME
UNIQUE (COLUMNNAME1, COLUMNNAME2));
```

Example: For example, the following statement creates a new table called customers and adds five columns. Here, age column is set to unique, so that you cannot have two records with same age:

```
CREATE TABLE CUSTOMERS(
    ID INT                      NOT NULL,
    NAME VARCHAR (20)            NOT NULL,
    AGE INT                      NOT NULL UNIQUE,
    ADDRESS CHAR (25),
    SALARY DECIMAL (18, 2),
    KEY (ID)
);
```

If customers table has already been created, then to add a unique constraint to age column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
MODIFY AGE INT NOT NULL UNIQUE;
```

You can also use following syntax, which supports naming the constraint in multiple columns as well:

```
ALTER TABLE CUSTOMERS
ADD CONSTRAINT MYUNIQUECONSTRAINT UNIQUE(AGE, SALARY);
```

Drop a unique constraint:

To drop a unique constraint, use the following statement:

```
ALTER TABLE CUSTOMERS
DROP CONSTRAINT MYUNIQUECONSTRAINT;
```

5.2.1.2.2 Primary Key Constraints

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values. A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a **composite key**.

Syntax for primary key constraints

```
TABLE<TABLE NAME>(COLUMNNAME DATA TYPE (SIZE) CONSTRAINT
CONSTRAINT_NAME PRIMARY KEY);
```

Composite primary key

```
CREATE TABLE<TABLE NAME>(COLUMNNAME1 DATA TYPE (SIZE),
COLUMNNAME2 DATA TYPE (SIZE), CONSTRAINT CONSTRAINT_NAME
PRIMARY KEY (COLUMNNAME1, COLUMNNAME2));
```

Create Primary Key

Here is the syntax to define ID attribute as a primary key in a CUSTOMERS table.

```
CREATE TABLE CUSTOMERS(
    ID INT NOT NULL,
    NAME VARCHAR (20) NOT NULL,
    AGE INT NOT NULL,
    ADDRESS CHAR (25) ,
    SALARY DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

To create a PRIMARY KEY constraint on the "ID" column when CUSTOMERS table already exists, use the following SQL syntax:

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);
```

To use ALTER TABLE statement to add a primary key to an already existing table, the column to which the primary key is to be assigned should have NOT NULL constraint when the table was first created. If null values are present, primary key cannot be assigned.

For defining a PRIMARY KEY constraint on multiple columns, use the following SQL statement:

```
CREATE TABLE CUSTOMERS(
    ID INT          NOT NULL,
    NAME VARCHAR (20) NOT NULL,
    AGE INT          NOT NULL,
    ADDRESS CHAR (25),
    SALARY DECIMAL (18, 2),
    PRIMARY KEY (ID, NAME)
);
```

To create a PRIMARY KEY constraint on the "ID" and "NAME" columns when CUSTOMERS table already exists, use the following SQL syntax:

```
ALTER TABLE CUSTOMERS
ADD CONSTRAINT PK_CUSTID PRIMARY KEY (ID, NAME);
```

Delete Primary Key:

To remove the primary key constraint use the following syntax:

```
ALTER TABLE CUSTOMERS DROP PRIMARY KEY;
```

5.2.1.3 Referential Integrity Constraints

This ensures that a value appearing in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called referential integrity.

Reference key (foreign key): It represents relationships between tables. Foreign key is a column whose values are derived from the primary key of the same or some other table.

The referential integrity constraint is specified between two tables and it is used to maintain the consistency among rows between the two tables.

The rules are:

1. Deleting a record from a primary table is not possible if matching records exist in a related table.
2. Changing a primary key value in the primary table is not possible if that record has related records.
3. Entering a value in the foreign key field of the related table is not possible if it doesn't exist in the primary key of the primary table.
4. However, entering a Null value in the foreign key is possible, specifying that the records are unrelated.

5.2.1.3.1 Foreign Key Constraints

Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table. The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table. If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

Syntax:

```
CREATE TABLE <TABLE NAME>(COLUMNNAME DATA TYPE (SIZE)
CONSTRAINT CONSTRAINT_NAME REFERENCES PARENT_TABLE_NAME);
```

Example: Consider the structure of the two tables as follows:

CUSTOMERS table:

```
CREATE TABLE CUSTOMERS(
    ID INT NOT NULL,
    NAME VARCHAR (20) NOT NULL,
    AGE INT NOT NULL,
    ADDRESS CHAR (25),
    SALARY DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

ORDERS table:

```
CREATE TABLE ORDERS (
    ID INT NOT NULL,
    DATE DATETIME,
    CUSTOMER_ID INT references CUSTOMERS(ID),
    AMOUNT double,
    PRIMARY KEY (ID)
);
```

If ORDERS table has already been created, and the foreign key has not yet been set, use the syntax for specifying a foreign key by altering a table.

```
ALTER TABLE ORDERS
ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

DROP a FOREIGN KEY Constraint:

To drop a FOREIGN KEY constraint, use the following syntax:

```
ALTER TABLE ORDERS
DROP FOREIGN KEY;
```

5.2.1.4 Index Constraint

The INDEX is used to create and retrieve data from the database very quickly. Index can be created by using single or group of columns in a table. When index is created, it is assigned a ROWID for each row before it sorts out the data.

Proper indexes are good for performance in large databases, but you need to be careful while creating index. Selection of fields depends on what you are using in your SQL queries.

Example: For example, the following SQL creates a new table called CUSTOMERS and adds five columns:

```
CREATE TABLE CUSTOMERS(
    ID INT NOT NULL,
    NAME VARCHAR (20) NOT NULL,
    AGE INT NOT NULL,
    ADDRESS CHAR (25),
    SALARY DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

Now, you can create index on single or multiple columns using the following syntax:

```
CREATE INDEX index_name
ON table_name ( column1, column2.....);
```

To create an INDEX on AGE column, to optimize the search on customers for a particular age, following is the SQL syntax:

```
CREATE INDEX idx_age
    ON CUSTOMERS (AGE);
```

DROP an INDEX Constraint:

To drop an INDEX constraint, use the following syntax:

```
ALTER TABLE CUSTOMERS
    DROP INDEX idx_age;
```

5.2.1.5 Default Constraint

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

Example: For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, SALARY column is set to 5000.00 by default, so in case INSERT INTO statement does not provide a value for this column, then by default this column would be set to 5000.00.

```
CREATE TABLE CUSTOMERS(
    ID INT NOT NULL,
    NAME VARCHAR (20) NOT NULL,
    AGE INT NOT NULL,
    ADDRESS CHAR (25),
    SALARY DECIMAL (18, 2) DEFAULT 5000.00,
    PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a DEFAULT constraint to SALARY column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
    MODIFY SALARY DECIMAL (18, 2) DEFAULT 5000.00;
```

Drop Default Constraint:

To drop a DEFAULT constraint, use the following syntax:

```
ALTER TABLE CUSTOMERS  
    ALTER COLUMN SALARY DROP DEFAULT;
```

5.2.1.6 Assertion

An assertion is a predicate expressing a condition we wish the database to always satisfy. Domain constraints, functional dependency and referential integrity are special forms of assertion. Where a constraint cannot be expressed in these forms, we use an assertion.

Syntax:

```
CREATE ASSERTION ASSERTION-NAME CHECK PREDICATE
```

When an assertion is created, the system tests it for validity. If the assertion is valid, any further modification to the database is allowed only if it does not cause that assertion to be violated. This testing may result in significant overhead if the assertions are complex. Hence assertions should be used with great care. Some system developers omit support for general assertions or provide specialized form of assertions that are easier to test.

5.3 SECURITY

Security of data is an important concept in DBMS because it is essential to safeguard the data against any unwanted users. It is a protection from malicious attempts to steal or modify data. Database management systems are increasingly being used to store information about all aspects of an enterprise. The data stored in a DBMS is often vital to the business interests of the organization and is regarded as a corporate asset. In addition to protecting the intrinsic value of the data, corporations must consider ways to ensure privacy and to control access to data that must not be revealed to certain groups of users for various reasons.

There are five different levels of security

1. Database system level

Authentication and authorization mechanism allows specific users access only to required data.

2. Operating level

- Protection from invalid logins
- File-level access protection
- Protection from improper use of “superuser” authority.
- Protection from improper use of privileged machine instructions.

3. Network level

- Each site must ensure that it communicates with trusted sites.
- Links must be protected from theft or modification of messages.

Mechanisms used

- Identification protocol (password based)
- Cryptography

4. Physical level

- Protection of equipment from floods, power failure etc.
- Protection of disks from theft, erasure, physical damage etc.
- Protection of network and terminal cables from wire tapes, non-invasive electronic eavesdropping, physical damage, etc.

Solution:

- Replication hardware-mirrored disks, dual busses etc.
- Multiple access paths between every pair of devices.
- Physical security by locks, police etc.
- Software techniques to detect physical security breaches

5. Human level

- Protection from stolen passwords, sabotage, etc.

Solution:

1. Frequent change of passwords.
2. Use of “non-guessable” passwords.
3. Log all invalid access attempts.
4. Data audits.
5. Careful hiring practices.

5.3.1 Authorization

Forms of authorization on parts of the database:

- **Read authorization:** allows reading, but not modification of data.
- **Insert authorization:** allows insertion of new data, but not modification of existing data.
- **Update authorization:** allows modification, but not deletion of data.
- **Delete authorization:** allows deletion of data.

Forms of authorization to modify the database schema:

- **Index authorization:** allows creation and deletion of indices.
- **Resources authorization:** allows creation of new relations.
- **Alteration authorization:** allows addition or deletion of attributes in a relation.
- **Drop authorization:** allows deletion of relations.
- **The grant statement** is used to give authorization

Syntax:

GRANT <PREVILEGE LIST> ON <RELATION NAME OR VIEW NAME> TO <USER/ROLE LIST>

Example:

```
grant select on account to john, mary;
```

// this query grants database users john and mary with select authorization.

- **Revoke statement:** It gets back the granted privilege

Syntax:

```
REVOKE <PREVILEGE LIST> ON <RELATION NAME OR VIEW NAME> FROM  
<USER/ROLE LIST>[RESTRICT|CASCADE]
```

Example:

```
revoke select on branch from john, mary;
```

- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the revoke.
- We can prevent cascading by specifying restrict:
 - revoke select on *branch* from U_1, U_2, U_3 restrict
- With restrict, the revoke command fails if cascading revokes are required.

5.3.2 Roles

- Roles permit common privileges for a class of users. Privileges can be specified for a class of users just once by creating a corresponding “role”
- Privileges can be granted to or revoked from roles, just like user
- Roles can be assigned to users, and even to other roles

Example:

- create role *teller*
create role *manager*
- grant select on *branch* to *teller*
grant update (*balance*) on *account* to *teller*
grant all privileges on *account* to *manager*
grant *teller* to *manager*
grant *teller* to *alice, bob*
grant *manager* to *avi*

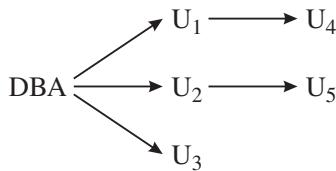
5.3.3 Authorization and Views

- Users can be given authorization on views, without being given any authorization on the relations used in the view definition
- Ability of views to hide data serves both to simplify usage of the system and to enhance security by allowing users access only to data they need for their job
- A combination of relational-level security and view-level security can be used to limit a user’s access to precisely the data that user needs.

5.3.4 Granting of Privileges

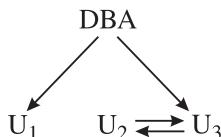
- The passage of authorization from one user to another may be represented by an authorization grant graph.
- The nodes of this graph are the users.
- The root of the graph is the database administrator.
- Consider graph for update authorization on loan.
- An edge $U_i \rightarrow U_j$ indicates that user U_i has granted update authorization on loan to U_j .

Authorization Grant Graph

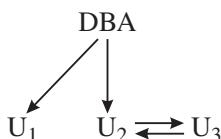


5.3.4.1 Authorization Grant Graph

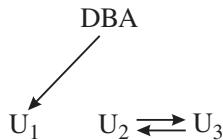
- Requirement:** All edges in an authorization graph must be part of some path originating with the database administrator
- If DBA revokes grant from U_1 :
 - Grant must be revoked from U_4 since U_1 no longer has authorization
 - Grant must not be revoked from U_5 since U_5 has another authorization path from DBA through U_2
- Must prevent cycles of grants with no path from the root:
 - DBA grants authorization to U_2
 - U_2 grants authorization to U_3
 - U_3 grants authorization to U_2



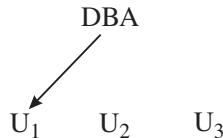
- If the database administrator revokes authorization from U_2 , U_2 retains authorization through U_3 .



- If authorization is revoked subsequently from U_3 , U_3 appears to retain authorization through U_2 .



- When the database administrator revokes authorization from both U_2 and U_3 , the edges from U_3 to U_2 and from U_2 to U_3 are no longer parts of a path starting with the database administrator.
- The edges between U_2 and U_3 are deleted, and the resulting authorization graph is



5.3.5 Audit Trails

- An audit trail is a log of all changes (inserts/deletes/updates) to the database along with information such as which user performed the change, and when the change was performed.
- Used to track erroneous/fraudulent updates.
- Can be implemented using triggers, but many database systems provide direct support.

5.3.6 Limitations of SQL Authorization

- SQL does not support authorization at a tuple level
 - E.g. we cannot restrict students to see only their own grades
- With the growth in Web access to databases, database accesses come primarily from application servers. End users don't have database user ids, they are all mapped to the same database user id
- All end-users of an application (such as a web application) may be mapped to a single database user
- The task of authorization in above cases falls on the application program, with no support from SQL
 - **Benefit:** fine grained authorizations, such as to individual tuples, can be implemented by the application.
 - **Drawback:** Authorization must be done in application code, and may be dispersed all over an application
 - Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code

5.3.7 Encryption

- *Data Encryption Standard* (DES) substitutes characters and rearranges their order on the basis of an encryption key which is provided to authorize users via a secure mechanism. Scheme is no more secure than the key transmission mechanism since the key has to be shared.

- Advanced Encryption Standard (AES) is a new standard replacing DES, and is based on the Rijndael algorithm, but is also dependent on shared secret keys
- *Public-key encryption* is based on each user having two keys:
 - **public key:** publicly published key used to encrypt data, but cannot be used to decrypt data
 - **private key:** key known only to individual user, and used to decrypt data. Need not be transmitted to the site doing encryption.
- Encryption scheme is such that it is impossible or extremely hard to decrypt data given only the public key.
- The RSA public-key encryption scheme is based on the hardness of factoring a very large number (100's of digits) into its prime components.

5.3.8 Authentication (Challenge Response System)

- Password based authentication is widely used, but is susceptible to sniffing on a network
- Challenge-response systems avoid transmission of passwords
 - DB sends a (randomly generated) challenge string to user
 - User encrypts string and returns result.
 - DB verifies identity by decrypting result
 - Can use public-key encryption system by DB sending a message encrypted using user's public key, and user decrypting and sending the message back
- **Digital signatures** are used to verify authenticity of data
 - Private Key is used to sign data and the signed data is made public.
 - Anyone can read the data with public key but cannot generate data without private key.
 - Digital signatures also help ensure nonrepudiation: sender cannot later claim to have not created the data

5.3.9 Digital Certificates

Digital certificates are used to verify authenticity of public keys.

Problem: when you communicate with a web site, how do you know if you are talking with the genuine web site or an imposter?

Solution: use the public key of the web site

Problem: how to verify if the public key itself is genuine?

Solution:

- Every client (e.g. browser) has public keys of a few root-level certification authorities
- A site can get its name/URL and public key signed by a certification authority: signed document is called a certificate
- Client can use public key of certification authority to verify certificate
- Multiple levels of certification authorities can exist. Each certification authority presents its own public-key certificate signed by a higher level authority, and uses its private key to sign the certificate of other web sites/authorities

5.4 EMBEDDED SQL

Embedded SQL provides an environment to develop application programs. The basic idea behind embedded SQL is to allow SQL statements in a program written in a high-level programming language such as C or C++. A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded SQL*. By embedding SQL statements in a C/C++ program, one can now write application programs in C/C++ that interact with the database.

Oracle provides a tool, called Pro*C/C++, which allows for applications to be developed in the C or C++ language with embedded SQL statements. The Pro*C/C++ preprocessor parses the embedded program and converts all SQL statements to system calls in C/C++ and produces a C/C++ program as its output. This C/C++ program can be compiled in the usual manner to produce the executable version of the application program. The embedded SQL program should be preprocessed prior to compilation. The preprocessor replaces embedded SQL requests with host language declarations and procedure calls. The resulting program is compiled by host language compiler.

EXEC SQL statement is used to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement> END_EXEC

Note: this varies by language (for example, the Java embedding uses # SQL { }; , C language uses semicolon instead of END_EXEC)

5.4.1 Host Variables

Since SQL statements are to be embedded within the C program, there is a need for a mechanism to pass values between the C program environment and the SQL statements that communicate with the Oracle database server. Special variables, called host variables, are defined in the embedded program for this purpose. These host variables are defined between the begin declare section and end declare section directives of the preprocessor as follows:

```
EXEC SQL begin declare section;
    int cno;
    varchar cname[31];
    varchar street[31];
    int zip;
    char phone[13];
```

EXEC SQL end declare section;

The data types of the host variables must be compatible with the data types of the columns of the tables in the database. The following table shows data types in C that are compatible with commonly used data types in Oracle.

<i>Oracle Data Type</i>	<i>C Data Type</i>
char	char
char(N)	char array[N+1]

varchar(N)	varchar array[N+1]
date	char array[10]
number(6)	int
number(10)	long int
number(6,2)	float

The varchar array in C is declared as

```
varchar cname[31];
```

and the Pro*C preprocessor produces the following C code corresponding to the above declaration:

```
/* varchar cname[31]; */
struct {
    unsigned short len;
    unsigned char arr[31];
} cname;
```

Note that the varchar array variable cname has been transformed into a structure (with the same name) containing two fields: arr and len. The arr field will store the actual string and the len field will store the length of the character string. When sending a varchar value to the database, it is the responsibility of the programmer to make sure that both fields, arr and len, are assigned proper values. When receiving such a value from the database, both fields are assigned appropriate values by the system. The date data type is mapped to a fixed-length (10 characters, corresponding to the default date format in Oracle) character string in C. The numeric data types are appropriately mapped to small int, int, long int, float, or double, depending on their precisions in Oracle.

The host variables are used in the usual manner within C language constructs; however, when they are used in the embedded SQL statements, they must be preceded by a colon (:). Some examples of their usage are shown in the following code fragment.

```
scanf("%d",&cno);
EXEC SQL select cname
    into :cname
    from customers
    where cno = :cno;
scanf("%d%s%s%d%s",&cno,cname.arr,street.arr,&zip,phone);
cname.len = strlen(cname.arr);
street.len = strlen(street.arr);
EXEC SQL insert into customers
    values (:cno, :cname, :street, :zip, :phone);
```

The select statement in the above example has an additional clause, the into clause, which is required in embedded SQL since the results of the SQL statements must be stored in some place. The select into statement can be used only if it is guaranteed that the query returns exactly one or zero rows. A different technique is used to process queries that return more than one row. Note that

all occurrences of the host variables within the embedded SQL statements are preceded by a colon. Also, the len fields of all varchar arrays in C are set to the correct lengths before sending the host variables to the database.

5.4.2 Indicator Variables

A null value in the database does not have an equivalent in the C language environment. To solve the problem of communicating null values between the C program and Oracle, embedded SQL provides indicator variables, which are special integer variables used to indicate if a null value is retrieved from the database or stored in the database. Consider the orders table of the mail-order database.

5.4.3 SQL Communications Area (SQLCA)

Immediately after the Oracle database server executes an embedded SQL statement, it reports the Status of the execution in a variable called sqlca, the SQL communications area. This variable is a structure with several fields, the most commonly used one being sqlcode. Typical values returned in this field are shown below.

To include the sqlca definition, the following statement must appear early in the program:

```
EXEC SQL include sqlca;
```

The SQL connect statement is used to establish a connection with Oracle. Such a connection must be established before any embedded SQL statements can be executed. The following code fragment illustrates the use of the connect statement.

```
EXEC SQL begin declare section;
      varchar userid[10], password[15];
EXEC SQL end declare section;
      int loginok=FALSE, logintries=0;
```

To disconnect from the database, which should be done at the end of the program, the following statement is used:

```
EXEC SQL commit release;
```

This commits any changes that were made to the database and releases any locks that were placed during the course of the execution of the program.

The **open** statement causes the query to be evaluated

```
EXEC SQL opencEND_EXEC
```

The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

```
EXEC SQL fetch c into :cn, :cc END_EXEC
```

The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

```
EXEC SQL closec END_EXEC
```

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

5.4.4 Cursors

When an embedded SQL select statement returns more than one row in its result, the simple form of the select into statement cannot be used anymore. To process such queries in embedded SQL, the concept of a cursor is used. It is a mechanism that allows the C program to access the rows of a query one at a time. The cursor declaration associates a cursor variable with an SQL select statement.

To start processing the query, the cursor must first be opened. It is at this time that the query is evaluated. It is important to note this fact, since the query may contain host variables that could change while the program is executing. Also, the database tables may be changing while the program is executing. Once the cursor is opened, the fetch statement can be used several times to retrieve the rows of the query result, one at a time. Once the query results are all retrieved, the cursor should be closed.

The syntax for cursor declaration is

```
EXEC SQL declare (cur-name) cursor for
    (select-statement)
    [for {read only | update [of _column-list]}];
```

where *_cur-name_* is the name of the cursor and *_select-statement_* is any SQL select statement associated with the cursor. The select statement, which may involve host variables, is followed by one of the following two optional clauses:

for read only
or
for update [of _column-list_]

The for update clause is used in cases of *positioned* deletes or updates. The for read only clause is used to prohibit deletes or updates based on the cursor. If the optional for clause is left out, the default is for read only.

5.4.5 Transaction Control

A transaction is a sequence of database statements that must execute as a whole to maintain consistency. If for some reason one of the statements fails in the transaction, all the changes caused by the statements in the transaction should be undone to maintain a consistent database state. Oracle provides the following statements to create transactions within the embedded SQL program.

- **EXEC SQL set transaction read only.** These are typically used before a sequence of select or fetch statements that do only a read from the database. The statement begins a read-only transaction. At the end of the sequence of statements, the commit statement is executed to end the transaction.
- **EXEC SQL set transaction read write.** This is typically used before a sequence of statements that performs some updates to the database. This statement begins a read-write transaction. The transaction is terminated by either a commit statement that makes the changes permanent in the database or a rollback statement that undoes all the changes made by all the statements within the transaction.

- **EXEC SQL commit.** This statement commits all the changes made in the transaction and releases any locks placed on tables so that other processes may access them.
- **EXEC SQL rollback.** This statement undoes all the changes made in the course of the transaction and releases all locks placed on the tables.

5.5 DYNAMIC SQL

Dynamic SQL is a programming methodology for generating and executing SQL statements at run time. Dynamic SQL is generally used for sending SQL statements to the database manager from interactive query building graphical user interfaces and SQL command line processors as well as from applications where the complete structure of queries is not known at application compilation time and the programming API supports dynamic SQL.

There are many situations in which the content of an SQL statement is not known by a user or programmer in advance. For example, suppose a spreadsheet allows a user to enter a query, which the spreadsheet then sends to the database management system to retrieve data. The contents of this query obviously cannot be known to the programmer when the spreadsheet program is written. To solve this problem, the spreadsheet uses a form of embedded SQL called dynamic SQL.

Unlike static SQL statements, which are hard-coded in the program, Dynamic SQL statements can be built at run time and placed in a string host variable. They are then sent to the database management system for processing. Because the database management system must generate an access plan at run time for dynamic SQL statements, dynamic SQL is generally slower than static SQL.

When a program containing dynamic SQL statements is compiled, the dynamic SQL statements are not stripped from the program, as in static SQL. Instead, they are replaced by a function call that passes the statement to the DBMS. Since the actual creation of dynamic SQL statements is based upon the flow of programming logic at application run time, they are more powerful than Static SQL statements. However, because the DBMS must go through each of the multiple steps of processing a SQL statement for each dynamic request, Dynamic SQL tends to be slower than Static SQL.

The Dynamic SQL allows programs to construct and submit SQL queries at run time. Dynamic SQL can be executed immediately or can be used later. Two principle Dynamic SQL statements are

1. PREPARE
2. EXECUTE

Example:

```
SQLSOURCE = 'Delete from account where amount<10000;  
EXEC SQL PREPARE SQLPREPPED FROM: SQL SOURCE;  
EXEC SQL EXECUTE SQLPREPPED;
```

- SQLSOURCE specifies the programming language variable.
- SQLPREPPED identifies the SQL variables. It holds the compiled version of SQL statement whose source form is given in SQLSOURCE.
- The prepare statement takes the source statement and prepares it to produce an executable version, which is stored in SQLPREPPED.

- EXECUTE statement executes the SQLPREPPED version.
- EXECUTE IMMEDIATE statement combines the functions of PREPARE and EXECUTE in a single operation.

Call Level Interface

The SQL Call Level Interface (SQL/CLI) is based on Microsoft's OpenSource DataBase Connectivity (ODBC). They allow the applications to be written from which the exact SQL code is not known until run time. Two principle reason for using SQL/CLI

- Dynamic SQL is a source code statement. Dynamic SQL requires some kind of SQL compiler to process the operations like PREPARE, EXECUTE. SQL/CLI does not require any special compiler instead it uses the host language compiler. It is in object code form.
- SQL/CLI is DBMS independent i.e., it allows creation of several applications with different DBMS.

Example for SQL/CLI:

```
strcpy (sqlsource, " Delete from account where amount>10000);  
rc = SQLEexecDirect(hstmt,(SQLCHAR*)sqlsource,SQL.NTS);
```

- Strcpy is used to copy the source form of delete statement into sqlsource variable.
- SQLEexecDirect executes the SQL Statement contained in sqlsource and assigns the return code to the variable rc.

Two standards connect an SQL database and perform queries and updates.

- OpenSource DataBase Connectivity (ODBC) was initially developed for C language and extended to other languages like C++, C# and Visual Basic.
- Java DataBase Connectivity (JDBC) is an application program interface for java language.
- The user and application connects to an SQL server establishing a session, executes a series of statements and finally disconnects the session.
- In addition to normal SQL commands, a session can also contain commands to commit the work carried out or rollback the work carried out in a session.

5.6 DISTRIBUTED DATABASES

A distributed database looks like a single database for the user but is, in fact, a set of databases stored on multiple computers. The data on several computers can be simultaneously accessed and modified using a network. They do not share main memory or disks. Each database server in the distributed database is controlled by its local DBMS, and each cooperates to maintain the consistency of the global database.

A database server is the software managing a database, and a client is an application that requests information from a server. Each computer in a system is a node. A node in a distributed database system can be a client, a server, or both.

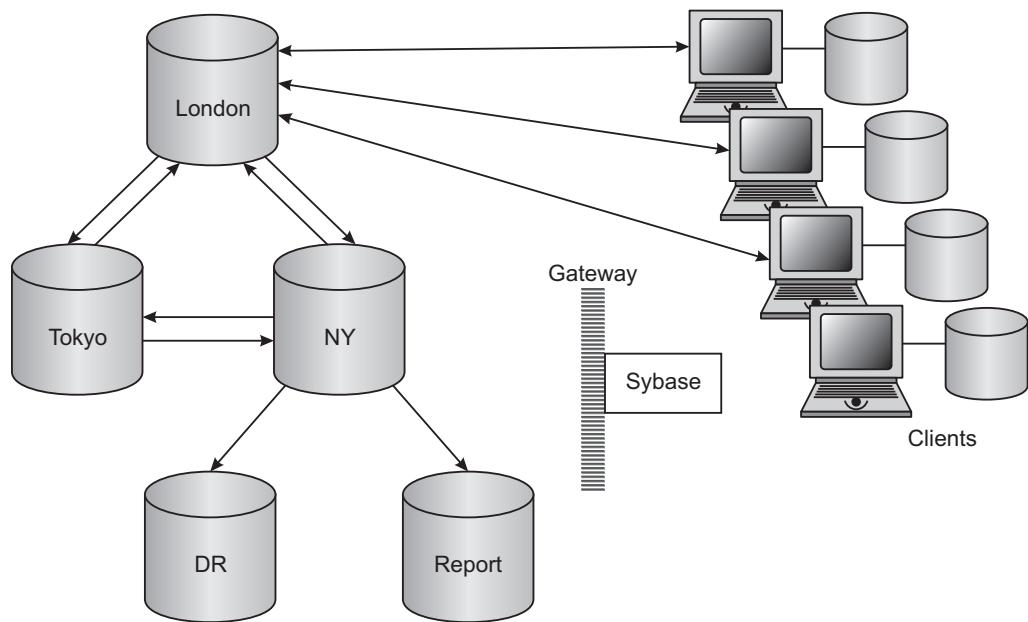


Fig. 5.10. Distributed database.

5.6.1 Features of Distributed Databases

A Distributed DBMS may have a number of local applications, but it has at least one global application. Thus, a distributed DBMS has the following features:

- (i) A distributed DBMS is a collection of logically related shared data.
- (ii) The data in a distributed DBMS is split into a number of fragments or partitions.
- (iii) Fragments may be replicated in a distributed system.
- (iv) Fragments/replicas are allocated to different sites.
- (v) In a distributed system, the sites are linked by communications network.
- (vi) The data at each site is under the control of a DBMS.
- (vii) The DBMS at each site has its own right, that is, it can handle local applications independently.
- (viii) Each DBMS in a distributed system participates in at least one global application.

5.6.2 Types of Distributed Databases

A distributed DBMS may be classified

1. Homogeneous distributed databases
2. Heterogeneous distributed databases

In an ideal distributed database system, the sites would share a common global schema, all sites would run the same database management software and the sites are aware of the existence of other sites. In a distributed system, if all sites use the same DBMS product, it is called a **homogenous distributed database system**. However, in reality, a distributed database has to be constructed

by linking multiple already-existing database systems together, each with its own schema and possibly running different database management software. Such systems are called **heterogeneous distributed database systems**. In a heterogeneous distributed database system, sites may run different DBMS products that need not be based on the same underlying data model, and thus, the system may be composed of relational, network, hierarchical, and object-oriented DBMSs.

Homogeneous distributed DBMS provides several advantages such as simplicity, ease of designing and incremental growth. It is much easier to design and manage a homogeneous distributed DBMS than a heterogeneous one. In a homogeneous distributed DBMS, making the addition of a new site to the distributed system is much easier, thereby providing incremental growth. These systems also improve performance by exploiting the parallel processing capability of multiple sites.

5.6.3 Components of a Distributed Databases

A Distributed DBMS controls the storage and efficient retrieval of logically interrelated data that are physically distributed among several sites. Therefore, a distributed DBMS includes the following components.

- (a) **Computer Workstations (sites or nodes):** A distributed DBMS consists of a number of computer workstations that form the network system. The distributed database system must be independent of the computer system hardware.
- (b) **Network Components (both hardware and software):** Each workstation in a distributed system contains a number of network hardware and software components. These components allow each site to interact and exchange data with each other site. Network system independence is a desirable property of the distributed system.
- (c) **Communication Media:** In a distributed system, any type of communication (data transfer, information exchange) among nodes is carried out through communication media. This is a very important component of a distributed DBMS. It is desirable that a distributed DBMS be communication media independent, that is, it must be able to support several types of communication media.
- (d) **Transaction Processor (TP):** A TP is a software component that resides in each computer connected with the distributed system and is responsible for receiving and processing both local and remote applications' data requests. This component is also known as the application processor (AP) or the transaction manager (TM).
- (e) **Data Processor (DP):** A DP is also a software component that resides in each computer connected with the distributed system and stores and retrieves data located at that site. The DP is also known as the data manager (DM). In a distributed DBMS, a DP may be a centralized DBMS.

5.6.4 Distributed Data Storage

Consider a relation 'r' that is to be stored in the database. There are two approaches storing this relation in the distributed database.

Replication: the system maintains several identical copies of the relation, and stores each replica at a different site.

Fragmentation: the system partitions the relation into several fragments and stores each fragment at a different site.

5.6.4.1 Data Replication

There are a number of advantages and disadvantages to replication

Availability: If one of the site containing the relation ‘r’ fails, then the relation ‘r’ can be found in another site, thus, the system can continue to process queries involving ‘r’, despite the failure of one site.

Increased Parallelism: In the case where the majority of accesses to the relation ‘r’ result in only the reading of the relation, then several sites can process queries involving ‘r’ in parallel. The more replicas of ‘r’ there are, greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites.

Increased overhead on update: The system must ensure that all replicas of a relation ‘r’ are consistent. Otherwise, erroneous computations may result. Thus, wherever ‘r’ is updated, the update must be propagated to all sites containing replicas. The result is increased overhead.

5.6.4.2 Data Fragmentation

If relation ‘r’ is fragmented, ‘r’ is divided into a number of fragments $r_1, r_2, r_3, \dots, r_n$. These fragments contain sufficient information to allow reconstruction of the original relation r.

There are two types of fragmentation in a relation

1. Horizontal fragmentation
2. Vertical fragmentation

Horizontal fragmentation: It splits the relation by assigning each tuple of ‘r’ to one or more fragments.

Vertical fragmentation: It splits the relation by decomposing the schema R of relation ‘r’.

5.6.4.3 Transparency

The user of a distributed database system should not be required to know either where the data are physically located or how the data can be accessed at the specific local site. This characteristic is called as data transparency.

Fragmentation Transparency: Users are not required to know how a relation has been fragmented.

Replication Transparency: Users view each data objects as logically unique. The distributed system may replicate an object to increase either system performance or data availability. Users do not have to be concerned with what data objects have been replicated or where replicas have been placed.

Location Transparency: Users are not required to know the physical location of the data. The Distributed database system should be able to find any data as long as data identifiers are supplied by the user transaction.

5.6.5 Advantages of Distributed System

1. **Sharing Data:** using distributed system, users at one site are able to access the data residing to the site.
2. **Autonomy:** Each site is able to retain a degree of control over data that are stored locally.
3. **Availability:** If one site fails in distributed system, the remaining sites may be able to continue operating.

5.6.6 Disadvantages of Distributed System

1. **Software Development Cost:** It is more difficult to implement a distributed database system so it is more costly.
2. **Greater Potential for Bugs:** Since the sites that constitute the distributed system operate in parallel, it is harder to ensure the correctness of algorithms, especially during failures of part of the system, and recovery from failures.
3. **Increased Processing Overhead:** The exchange of messages and the additional computation required to achieve inter site co-ordination are a form of overhead that does not arise in centralized systems.

5.7 CLIENT/SERVER DATABASES

Client/Server architecture is defined as a software based architecture which enables distributed computing resources on a network to share common resources among group of users at intelligent workstations.

In centralized database systems, the database system, application programs, and user-interface all are executed on a single system and dummy terminals are connected to it. The processing power of single system is utilized and dummy terminals are used only to display the information.

As the personal computers became faster, more powerful, and cheaper, the database system started to exploit the available processing power of the system at the user's side, which led to the development of client/server architecture. In client/server architecture, the processing power of the computer system at the user's end is utilized by processing the user-interface on that system.

A client is a computer system that sends request to the server connected to the network, and a server is a computer system that receives the request, processes it, and returns the requested information back to the client. Client and server are usually present at different sites. The end users work on client computer system and database system runs on the server. Servers can be of several types, for example, file servers, printer servers, web servers, database servers, etc. The client machines have user interfaces that help users to utilize the servers. It also provides users the local processing power to run local applications on the client side.

There are two approaches to implement client/server architecture.

1. Two-tier architecture
2. Three-tier architecture

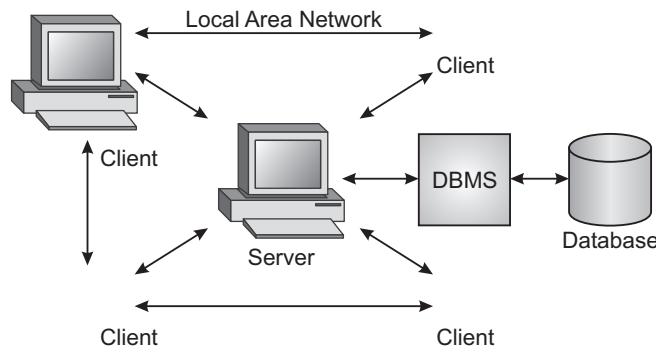


Fig. 5.11. Client/Server database.

5.7.1 Two-tier Architecture

If the user interface and application programs are placed on the client side and the database system on the server side then it is called two-tier architecture. The application programs that reside at the client side invoke the DBMS at the server side. The application program interface standards like Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) are used for the interaction between client and server.

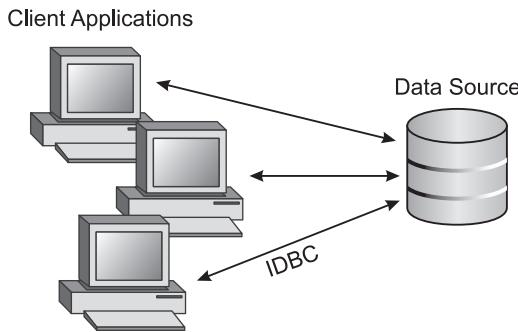


Fig. 5.12. Two tier architecture.

5.7.2 Three-tier Architecture

The three-tier architecture is primarily used for web-based applications. It adds intermediate layer known as application server between the client and the database server. The client communicates with the application server, which in turn communicates with the database server. The application server stores the business rules including procedures and constraints used for accessing data from database server. It checks the client's credentials before forwarding a request to database server. Hence, it improves database security.

When a client requests for information, the application server accepts the request, processes it, and sends corresponding database commands to database server. The database server sends the result back to application server which is converted into GUI format and presented to the client.

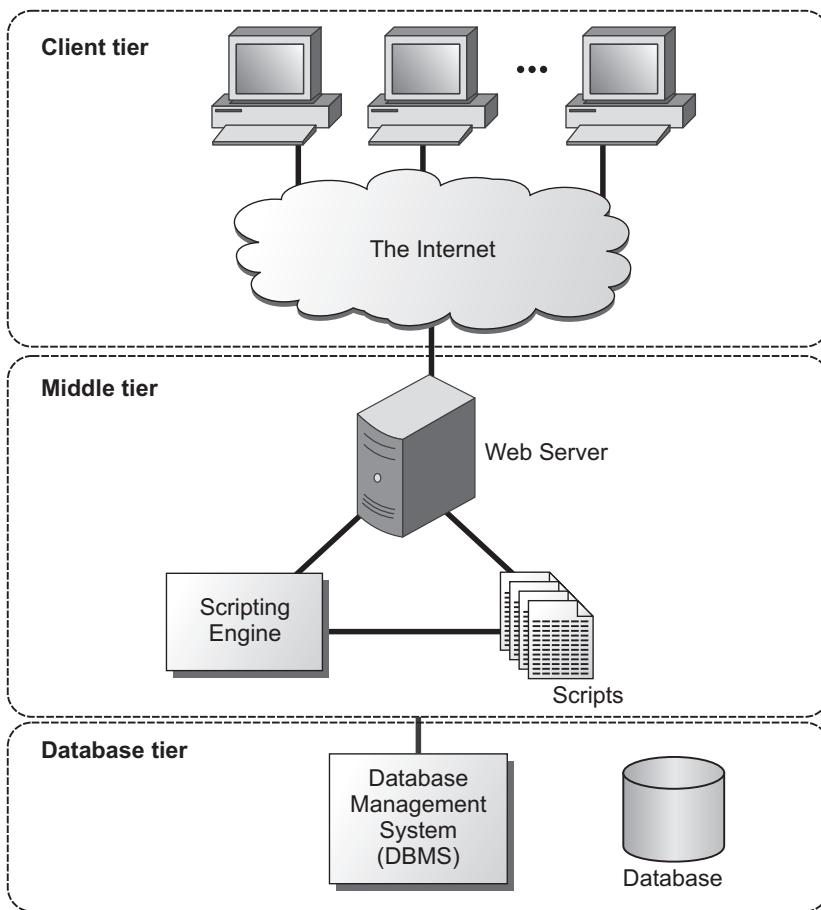


Fig. 5.13. Three-tier architecture.

5.7.3 Merits of Client/Server Computing

1. **Platform Independence:** It is the most required to be locked into a single vendor's proprietary hardware or software environments.
2. **Improved Performance:** With more processing power scattered throughout, the enterprise, the information is processed with faster response time.
3. **Reduced Operating Costs:** Client/Server computing replaces expensive large systems with less expensive smaller ones networked together.
4. **Fast Application Development:** Advances in client/server tools have led rapid application development.
5. **Easier Data Access and Processing:** More users are able to access more data more quickly than ever before.

REVIEW QUESTIONS

1. Define join. Explain the need of join operation in database management systems.
2. List out different types of joins and explain each of the joins with suitable examples.
3. Why are constraints important in a database?
4. What is the need for primary key in a relation and explain how to assign primary key and composite primary key for a relation.
5. Explain how to assign constraints for an already created table without affecting the table structure.
6. Explain referential integrity and the role of foreign key.
7. Define assertion.
8. Why is security needed in a database?
9. Explain the concept of authorization with the help of authorization grant graph.
10. Explain embedded SQL.
11. Write a short note on dynamic SQL.
12. Explain distributed databases and client/server databases.
13. Explain homogeneous and heterogeneous databases with respect to distributed databases.
14. Discuss the circumstances and reasons when and why an organization with a centralized database system would prefer to move on to a distributed database environment. What are the possible disadvantages of such a decision?
15. Describe the relative advantages and disadvantages of distributed DBMSS.
16. Differentiate homogeneous and heterogeneous distributed DBMSS with examples.



Chapter 6

Relational Database Design

6.1 INTRODUCTION

The goal of a relational data base design is to generate a set of relation schema that allows us to store information without unnecessary redundancy. It should allow us to retrieve information easily.

A relational data base design may lead to:

- (i) Repetition of data
- (ii) Inadequacy to represent certain information.

Design Goals

- (a) Avoid redundant data.
- (b) Ensure that relationships among attributes are represented.
- (c) Facilitate the checking of updates for violation of database integrity constraints.

6.2 ANOMALIES IN DATABASES

Database anomalies, are really just unmatched or missing information caused by limitations or flaws within a given database. Databases are designed to collect data and sort or present it in specific ways to the end user. Entering or deleting information or update new record can cause some problems leading the database to inconsistent state.

There are three types of anomalies. They are:

1. Insert Anomalies
2. Update Anomalies
3. Delete Anomalies

1. Insert Anomalies

- The inability to insert part of information into a relational schema due to the unavailability of part of the remaining information is called Insert Anomalies.
- **Example:** If there is a guide having no student registered under him, then we cannot insert the guide's information in the schema project.

2. Update Anomalies

- Updating a relation schema with redundant data may lead to update anomalies.
- **Example:** If a person changes his address, the updation should be carried out wherever the copies occur. If it is not updated properly then data inconsistency arises.

3. Delete Anomalies

- If the deletion of some information leads to loss of some other information, then we say there is a deletion anomaly.
- **Example:** If a guide guides one student and if the student discontinues the course then the information about the guide will be lost.

6.3 REDUNDANT INFORMATION

Storing the same information several times is called redundancy. Redundancy leads to

- Increase in size of the database
- Wastage of storage space
- Inconsistencies

Consider the following student table

<i>Student_Id</i>	<i>student_Name</i>	<i>student_Address</i>	<i>students_Semester</i>	<i>mark1</i>	<i>mark2</i>	<i>mark3</i>
CS101	Raghul	Chennai	First	75	83	84
CS101	Raghul	Chennai	Second	82	67	87
CS101	Raghul	Chennai	Third	76	79	89
CS102	Renjith	Madurai	First	87	54	63
CS102	Renjith	Madurai	Second	73	84	52
CS102	Renjith	Madurai	Third	65	84	90

The details of students like student_id, student_address are repeated while recording marks of different semesters. the repeated data is called redundant. If any modification is made to the repeating data, the change should be reflected everywhere if not the database will be led to data inconsistency. This redundant data also will lead to increased use of disk space.

6.4 FUNCTIONAL DEPENDENCY

A functional dependency is defined as a constraint between two sets of attributes in a relation from a database. Functional dependencies are the relationships among the attributes within a Relation. Functional dependencies provide a formal mechanism to express constraints between attributes. If attribute A functionally depends on attribute B, Then for every instance of B you will know the respective value of A.

6.4.1 Notation of Functional Dependency

The notation of functional dependency is $A \rightarrow B$.

The meaning of this notation is:

1. "A" determines "B"
2. "B" is functionally dependent on "A"
3. "A" is called determinant
4. "B" is called object of the determinant

Example: $\text{Student_ID} \rightarrow \text{GPA}$, The meaning is the grade point average (GPA) can be determined if we know the student ID.

$A \rightarrow B$ does not imply $B \rightarrow A$

For example, in a student relation the value of an attribute "GPA" is known then the value of an attribute "Student_ID" cannot be determined.

A functional dependency $A \rightarrow B$ is said to be trivial if $B \subseteq A$.

6.4.2 Compound Determinants

If more than one attribute is necessary to determine another attribute in an entity then such a determinant is termed as composite determinant. For example, the internal marks and the external marks scored by the student determine the grade of the student in a particular subject.

Internal mark, external mark \rightarrow grade

Since more than one attribute is necessary to determine the attribute grade it is an example of compound determinant.

6.4.3 Types of Functional Dependency

There are three types of functional dependency:

- (a) Full functional dependency
 - (b) Partial functional dependency
 - (c) Transitive functional dependency
- (a) **Full functional dependency:** In a relation R, X and Y are attributes. X functionally determines Y. Subset of X should not functionally determine Y.

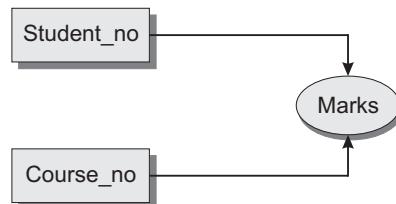


Fig. 6.1. Full functional dependency.

In the above example marks is fully functionally dependent on student_no and course_no together and not on subset of {student_no, course_no}.

This means marks cannot be determined either by student_no or course_no alone. It can be determined only using student_no and course_no together. Hence marks are fully functionally dependent on {student_no, course_no}.

- (b) **Partial functional dependency:** Attribute Y is partially dependent on the attribute X only if it is dependent on a subset of attribute X.

For example course_name, Instructor_name are partially dependent on composite attributes {student_no, course_no} because course_no alone defines course_name, Instructor_name.

- (c) **Transitive functional dependency:** X, Y and Z are 3 attributes in the relation R.

$$\boxed{\begin{array}{l} X \rightarrow Y \\ Y \rightarrow Z, \text{ then} \\ X \rightarrow Z \end{array}}$$

For example, grade depends on marks and in turn mark depends on {student_no course_no}, hence Grade depends transitively on {student_no & course_no}.

6.4.4 Uses of Functional Dependency

Functional dependency is used in the following cases:

1. Test relations to see if they are legal under a given set of functional dependencies. If a relation r is legal under a set F of functional dependencies, we say that r satisfies F .
2. To specify constraints on the set of legal relations. We say that F holds on R if all legal relations on R satisfy the set of functional dependencies F .

6.5 CLOSURE OF A SET OF FUNCTIONAL DEPENDENCIES

Given a set F of functional dependencies for a relation R , F^+ , the closure of F , is the set of all functional dependencies that are logically implied by F .

Given a set of attributes A and a set of functional dependencies, the closure of the set of attributes A under F , written as A^+ , is the set of attributes B that can be derived from A by applying the inference axioms to the functional dependencies of F . The closure of A is always nonempty set because $A \rightarrow A$ by the axiom of reflexivity.

Armstrong's axioms are sufficient to compute all of F^+ , which means if we apply Armstrong's rules repeatedly, we can find all the functional dependencies in F^+ .

6.5.1 Armstrong's Axioms

There are three rules to find the logically implied functional dependencies, called the Armstrong's rule.

1. **Reflexivity Rule:** If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
2. **Augmentation Rule:** If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
3. **Transitivity Rule:** If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds then $\alpha \rightarrow \gamma$ holds.

These rules are sound because they do not generate any incorrect functional dependencies and Complete because they generate all functional dependencies that hold.

6.5.2 Additional Rules

In addition to these three basic rules there are three additional rules to simplify manual computation of F^+ .

1. **Union Rule:** If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds
2. **Decomposition Rule:** If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds
3. **Pseudo transitivity Rule:** If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds

Example: Consider the schema $R = (A, B, C, G, H, I)$

Set of functional dependency $F = \{A \rightarrow B$

$$\begin{aligned} & A \rightarrow C \\ & CG \rightarrow H \\ & CG \rightarrow I \\ & B \rightarrow H \end{aligned}$$

Some members of F^+

- $A \rightarrow H$
 - By transitivity from $A \rightarrow B$ and $B \rightarrow H$ then $A \rightarrow H$ holds.
- $AG \rightarrow I$
 - By augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$ we get $AG \rightarrow I$.

(OR)

- By Pseudo transitivity rule $A \rightarrow C$ and $CG \rightarrow I$, then $AG \rightarrow I$ holds.
- $CG \rightarrow HI$
 - By union rule of $CG \rightarrow H$ and $CG \rightarrow I$, $CG \rightarrow HI$ holds.

The left-hand and right-hand sides of a functional dependency are both subsets of R .

Procedure for Computing F^+ : To compute the closure of a set of functional dependencies F :

$$F^+ = F$$

repeat

for each functional dependency f in F^+
 apply reflexivity and augmentation rules on f
 add the resulting functional dependencies to F^+
for each pair of functional dependencies f_1 and f_2 in F^+
if f_1 and f_2 can be combined using transitivity
then add the resulting functional dependency to F^+
until F^+ does not change any further

A set of size n has 2^n subsets, there are a total of $2 \times 2^n = 2^{n+1}$ possible functional dependencies, where n is the number of attributes in R . Each iteration of the loop except the last iteration adds at least one functional dependency to F^+ .

6.5.3 Closure of Attribute Sets

An attribute β is functionally determined by α if $\alpha \rightarrow \beta$ holds. Given a set of attributes α , define the closure of α (α^+) under F as the set of attributes that are functionally determined by α under F

Algorithm to compute α^+ , the closure of α under F

```

result := a;
while (changes to result) do
    for each  $\beta \rightarrow \gamma$  in  $F$  do
        begin
            if  $\beta \subseteq result$  then result := result  $\cup$   $\gamma$ 
        end

```

Example of Attribute Set Closure

$$\begin{aligned}
R &= (A, B, C, G, H, I) \\
F &= \{A \rightarrow B \\
&\quad A \rightarrow C \\
&\quad CG \rightarrow H \\
&\quad CG \rightarrow I \\
&\quad B \rightarrow H\}
\end{aligned}$$

To show the working of the algorithm let us compute closure of A , $(AG)^+$

The algorithm start with result = AG

1. $result = AG$
2. $A \rightarrow B$ includes B in $result$. Since $A \rightarrow B$ is in F and $A \subseteq result$, so $result := result \cup B$.
3. $result = ABCG$ ($A \rightarrow C$)
4. $result = ABCGH$ ($CG \rightarrow H$ and $CG \subseteq AGBC$)
4. $result = ABCGHI$ ($CG \rightarrow I$ and $CG \subseteq AGBCH$)

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for super key:
 - To test if α is a super key, we compute α^+ , and check if α^+ contains all attributes of R .
- Testing functional dependencies
 - To check if a functional dependency $\alpha \rightarrow \beta$ holds, just check if $\beta \subseteq \alpha^+$.
 - That is, we compute α^+ by using attribute closure, and then check if it contains β .
 - Is a simple and cheap testing procedure, and very useful

6.6 CANONICAL COVER

If a relational schema R has a set of functional dependencies. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies. The system must roll back the update if it violates any functional dependencies in the set F . The violation can be checked by testing a simplified set of functional dependencies. If simplified set of functional dependency is satisfied then the original functional dependency is satisfied and vice versa. Sets of functional dependencies may have redundant dependencies that can be inferred from the others.

A canonical cover of F is a “minimal” set of functional dependencies equivalent to F , having no redundant dependencies or redundant parts of dependencies.

Definition of Canonical Cover

A *canonical cover* for F is a set of dependencies F_c such that

- F logically implies all dependencies in F_c , and
- F_c logically implies all dependencies in F , and
- No functional dependency in F_c contains an extraneous attribute, and
- Each left side of functional dependency in F_c is unique.

To compute a canonical cover for F : $F_c = F$

repeat

Use the union rule to replace any dependencies in F

$$\alpha_1 \rightarrow \beta_1 \text{ and } \alpha_1 \rightarrow \beta_2 \text{ with } \alpha_1 \rightarrow \beta_1\beta_2$$

Find a functional dependency $\alpha \rightarrow \beta$ with an extraneous attribute either in α or in β .

If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ **until** F does not change

6.6.1 Extraneous Attributes

An attribute of a functional dependency is said to be extraneous if we can remove it without changing the closure of the set of functional dependencies.

Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .

- Attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
 - Attribute A is extraneous in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F .
- Example:* Given $F = \{A \rightarrow C, AB \rightarrow C\}$
- B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$
- Example:* Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
- C is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting C .

Testing if an Attribute is Extraneous

Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .

To test if attribute $A \in \alpha$ is extraneous in α

1. compute $(\{\alpha\} - A)^+$ using the dependencies in F
2. check that $(\{\alpha\} - A)^+$ contains β ; if it does, A is extraneous in α

To test if attribute $A \in \beta$ is extraneous in β

3. compute α^+ using only the dependencies in $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
4. check that α^+ contains A ; if it does, A is extraneous in β

Example: F contains $AB \rightarrow CD$, $A \rightarrow E$ and $E \rightarrow C$. To check C is extraneous in $AB \rightarrow CD$, we compute the attribute closure of AB under

$$F' = \{ AB \rightarrow CD, A \rightarrow E, E \rightarrow C \}$$

The closure is $ABCDE$, which includes CD . So C is extraneous.

6.6.2 Computing a Canonical Cover

Consider the following set F of functional dependencies on schema $R = (A, B, C)$

$$\begin{aligned} F = & \{ A \rightarrow BC \\ & B \rightarrow C \\ & A \rightarrow B \\ & AB \rightarrow C \} \end{aligned}$$

By union rule combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$

- Set is now $\{ABC, B \rightarrow C, AB \rightarrow C\}$

A is extraneous in $AB \rightarrow C$

- Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
 - After deleting A from $AB \rightarrow C$ the resultant set will be $\{A \rightarrow BC, B \rightarrow C, B \rightarrow C\}$
 - $B \rightarrow C$ is already present in the set.
- So the resultant set is now $\{A \rightarrow BC, B \rightarrow C\}$

C is extraneous in $A \rightarrow BC$

- Removing C from $A \rightarrow BC$ we get $\{A \rightarrow B, B \rightarrow C\}$

The canonical cover is:

$$\begin{aligned} & A \rightarrow B \\ & B \rightarrow C \end{aligned}$$

Canonical cover might not be unique

6.7 NORMALIZATION

Normalization of data is a process of analyzing the given relational schema based on their functional dependencies and primary key to achieve the desirable properties of

- Minimize redundancy
- Minimize insert, delete and update anomalies during database activities

Normalization is an essential part of database design. The concept of normalization helps the designer to build efficient design.

Normalization includes creating tables and establishing relationships between those tables according to rules designed both to protect the data and to make the database more flexible by eliminating two factors: redundancy and inconsistent dependency.

Redundant data wastes disk space and creates maintenance problems. If data that exists in more than one place must be changed, the data must be changed in exactly the same way in all

locations. Inconsistent dependencies can make data difficult to access; the path to find the data may be missing.

Normalization is the analysis of functional dependencies between attributes. It is the process of decomposing relations with anomalies to produce well-structured relations. Well-structured relation contains minimal redundancy and allows insertion, modification, and deletion without errors or inconsistencies.

Normalization is a formal process for deciding which attributes should be grouped together in a relation. It is the primary tool to validate and improve a logical design so that it satisfies certain constraints that avoid unnecessary duplication of data. A relational table is said to be a particular normal form if it satisfied a certain set of constraints.

6.7.1 Purpose of Normalization

- Minimize redundancy in data.
- Remove insert, delete and update anomaly during database activities.
- Reduce the need to reorganize data when it is modified or enhanced.
- Normalization reduces a complex user view to a set of small and stable subgroups of fields/relations.
- This process helps to design a logical data model known as conceptual data model.

6.7.2 Normalization Forms

Different normalization forms are:

1. **First normal form (1NF):** A relation is said to be in the first normal form if it is already in unnormalized form and it has **no repeating group**.
2. **Second normal form (2NF):** A relation is said to be in second normal form if it is already in the first normal form and it has **no partial dependency**.
3. **Third normal form (3NF):** A relation is said to be in third normal form if it is already in second normal form and it has **no transitive dependency**.
4. **Boyce-Codd normal form(BCNF):** A relation is said to be in Boyce-Codd normal form if it is already in third normal form and **every determinant is a candidate key**. It is a stronger version of 3NF
5. **Fourth normal form (4NF):** A relation is said to be in fourth normal form if it is already in BCNF and it has **no multi valued dependency**.
6. **Fifth normal form (5NF):** A relation is said to be in 5NF if it is already in 4NF and has **no join dependency**.

6.7.3 First Normal Form (1NF)

1NF does not allow multi valued attribute or composite attribute and their combinations. It states that domain of the attribute includes only single value, atomic or indivisible value. 1NF does not allow relation within relation.

Example: Consider the following schema Department

Department

<u>Department_number</u>	<u>Department_name</u>	<u>Department_Location</u>
--------------------------	------------------------	----------------------------

Department Table

<u>Department_number</u>	<u>Department_name</u>	<u>Department_Location</u>
1	Nilgiris	{Coimbatore, Chennai}
2	Subiksha	{Chennai, Tirunelveli}
3	Krishna	Trichy
4	Kannan	Coimbatore

In our example Department relation is not in 1NF because Dlocation has multi valued attributes. There are 3 main techniques to achieve 1NF for such relation.

1. Remove the Department_Location that violates 1NF and place it in a separate relation Department_Location along with primary key Department_number of department. The primary key of this relation is the combination of {Department_number, Department_Location}.

Dept_location

<u>Department_number</u>	<u>Department_Location</u>
1	Coimbatore
1	Chennai
2	Chennai
2	Tirunelveli
3	Trichy
4	Coimbatore

2. Expand the key so that there will be separate tuple in the original department relation. The primary key becomes {Department_number, Department_location}. This solution has the disadvantage of introducing redundancy in the relation.

<u>Department_name</u>	<u>Department_number</u>	<u>Department_Location</u>
1	Nilgiris	Coimbatore
1	Nilgiris	Chennai
2	Subiksha	Chennai
2	Subiksha	Tirunelveli
3	Krishna	Trichy
4	Kannan	Coimbatore

3. If a maximum number of values is known for the multi valued attribute. For example, if it is known that at most three locations can exist for a department, and then replace Location by

Department_location1, Department_location2, and Department_location3. This solution has the disadvantage of introducing null values if most departments have fewer than three locations.

<i>Department_Number</i>	<i>Department_name</i>	<i>Department_Location1</i>	<i>Department_Location2</i>
1	Nilgiris	Coimbatore	Chennai
2	Subiksha	Chennai	Tirunelveli
3	Krishna	Trichy	NIL
4	Kannan	Coimbatore	NIL

6.7.4 Second Normal Form (2NF)

A relation is in Second Normal Form (2NF) when it meets the requirement of being in First Normal Form (1NF) and additionally:

- Does not have a composite primary key. Meaning that the primary key cannot be subdivided into separate logical entities.
- All the non-key columns are functionally dependent on the entire primary key.
- A row is in second normal form if, and only if, it is in first normal form and every non-key attribute is fully dependent on the key.
- 2NF eliminates functional dependencies on a partial key by putting the fields in a separate table from those that are dependent on the whole key.

2NF is based on the concept of full functional dependency. A functional dependency $X \rightarrow Y$ is **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more.

i.e., any attribute $A \in X, (X - \{A\})$ does not functionally determine Y .

A functional dependency $X \rightarrow Y$ is **partial functional dependency** if some attribute $A \in X$ removed from X and the functional dependency still holds.

i.e., for some $A \in X, (X - \{A\}) \rightarrow Y$ holds

Example:

$\{\text{eid}, \text{Pnumber}\} \rightarrow \text{hours}$ is fully functional dependency

$\{\text{eid}, \text{Pnumber}\} \rightarrow \text{Ename}$ is partial functional dependency because $\text{eid} \rightarrow \text{Ename}$ holds

The test for 2NF involves testing for FDs whose LHS attribute are parts of the Primary Key. If the Primary Key contains a single attribute, the test need not be applied at all. A relational schema R is in 2NF if every nonprime attribute A in R is full functional dependent on the Primary Key of R.

Prime attribute: An attribute of a relational schema R is called a Prime attribute of R if it is a member of some candidate key of R. **Nonprime attribute:** An attribute is called a nonprime attribute if it is not a prime attribute. i.e., if it is not a member of any candidate key.

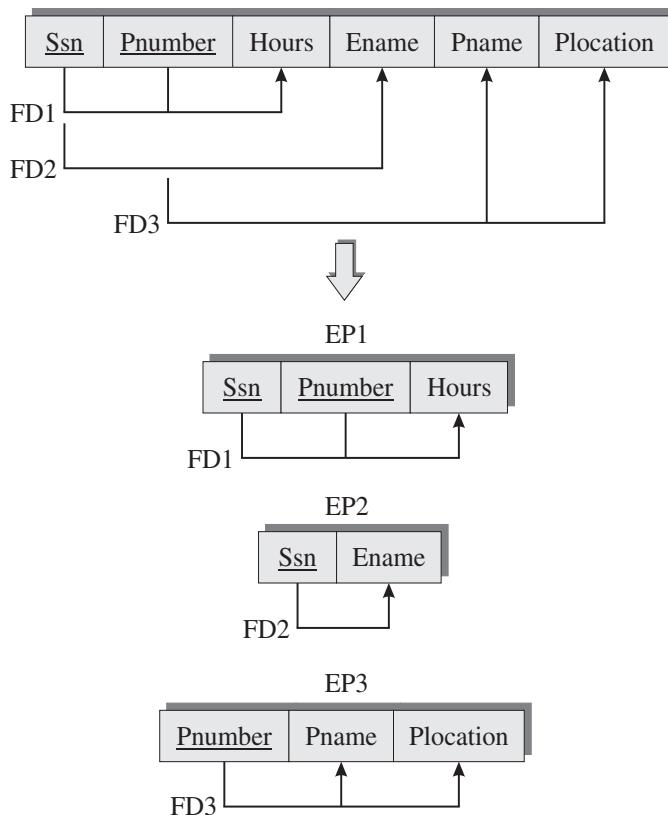
Example:

EMP_PROJ

<i>Ssn</i>	<i>Pnumber</i>	<i>Hours</i>	<i>Ename</i>	<i>Pname</i>	<i>Plocation</i>
------------	----------------	--------------	--------------	--------------	------------------

- In the above example EMP_PROJ. Ssn and Pnumber are primary key.
- The table is in 1NF.

EMP_PROJ



- FD1 is in 2NF but FD2 and FD3 violates 2NF.
- The *Ename*, *Pname*, *Plocation* in FD2 and FD3 are partially dependent on the primary key attributes *Ssn* and *Pnumber*.
- A relation which is not in second normal form can be made to be in 2NF by decomposing the relation into a number such that each nonprime attribute is fully functional dependent on the primary key.
- So the above table (EMP_PROJ) can be decomposed into three tables.

Consider the following example

Customer Table

<i>Customer_id</i>	<i>Customer_name</i>	<i>Order_id</i>	<i>Order_name</i>	<i>Sale_detail</i>
101	Ravi	10	Order 1	Sale 1
101	Ravi	11	Order 2	Sale 2
102	Alex	12	Order 3	Sale 3
103	George	13	Order 4	Sale 4

In the Customer table *Customer_id* and *Order_id* is the primary Key. Customer table is in first normal form but not in second normal form because there is a partial dependency of columns on Primary Key. *Customer_name* is only dependent on *Customer_id* similarly *Order_name* is dependent on *Order_id* and there is no link between *sale_detail* and *Customer_name*.

To convert the Customer table to second normal form, the table is divided into three different tables.

Customer_details Table

<i>Customer_id</i>	<i>Customer_name</i>
101	Ravi
101	Ravi
102	Alex
103	George

Order_Details Table

<i>Order_id</i>	<i>Order_name</i>
10	Order 1
11	Order 2
12	Order 3
13	Order 4

Sale_Details Table

<i>Customer_id</i>	<i>Order_id</i>	<i>Sale_detail</i>
101	10	Sale 1
101	11	Sale 2
102	12	Sale 3
103	13	Sale 4

Now all the three tables act in accordance with second normal form.

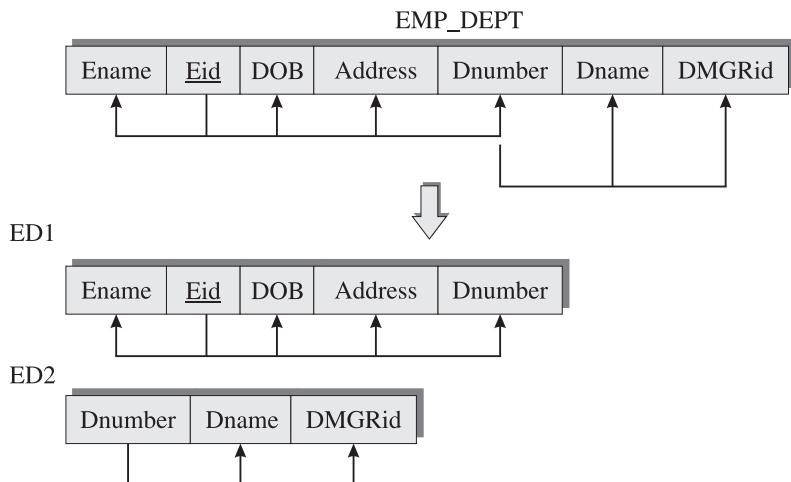
6.7.5 Third Normal Form (3NF)

A relational schema R is in 3NF if it satisfies 2NF and no nonprime attribute in relation R is transitively dependent on the primary key and additionally:

- Functional dependencies on non-key fields are eliminated by putting them in a separate table. At this level, all non-key fields are dependent on the primary key.
- A row is in third normal form if and only if it is in second normal form and if attributes that do not contribute to a description of the primary key are moved into a separate table.

Third Normal Form is based on the concept of transitive dependency. A functional dependency $X \rightarrow Y$ in a relational schema R is a transitive dependency if there is a set of attributes Z that is neither a candidate key or a subset of any key R, and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

Example:



- The dependency $Eid \rightarrow DMGRid$ is transitive through $Dnumber$ in **EMP_DEPT**, because both the dependencies $Eid \rightarrow Dnumber$ and $Dnumber \rightarrow DMGRid$ hold.
- $Dnumber$ is neither a key itself nor a subset of key of **EMP_DEPT**. Therefore the **EMP_DEPT** relational schema is not in 3NF.
- The relation is in 2NF because there is no partial dependencies on the key attribute.
- We can normalize **EMP_DEPT** by decomposing it into two 3NF relational schemas **ED1** and **ED2**.

Consider the student details table

Student_detail table

<i>Student_id</i>	<i>Student_name</i>	<i>DOB</i>	<i>Street</i>	<i>City</i>	<i>State</i>	<i>Zip_code</i>
-------------------	---------------------	------------	---------------	-------------	--------------	-----------------

In this table *Student_id* is primary key, but *street*, *city* and *state* depends upon *zip_code*. The dependency between *zip_code* and other fields is called transitive dependency. To normalize the table to third normal form, *street*, *city* and *state* are moved to new table and *zip_code* is assigned as primary key to that new table.

<i>New Student_detail table</i>			
<i>Student_id</i>	<i>Student_name</i>	<i>DOB</i>	<i>Zip_code</i>
<i>Address table</i>			
<i>Zip_code</i>	<i>Street</i>	<i>City</i>	<i>State</i>

6.7.5.1 Summary of Normal Forms

<i>Normal form</i>	<i>Test</i>	<i>Remedy</i>
1NF	Relation should have no multi valued attributes or nested relations.	Forms new relations for each multi valued attributes or nested relations.
2NF	For relations where primary key contains multiple attribute, no non key attribute should be functionally dependent on a part of primary key.	Decomposes and set up a new relation for each partial key with its dependent attributes. Make sure to keep a relation with the original primary key and any attributes that are fully FD on it.
3NF	Relations should not have a non key attribute functionally determined by another non key attribute. i.e., there should be no transitive dependency of a non key attribute on the primary key.	Decompose and set up a relation that includes the non key attributes that functionally determines other non-key attributes.

6.8 LOSS LESS DECOMPOSITION

Let R be a relational schema and F be a set of functional dependencies on R. Let R1 and R2 form a decomposition of R. Let r(R) be a relation with schema R.

The decomposition is lossless decomposition if

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

If natural join is computed on R1 and R2 then we get the relation r. A decomposition that is not a lossless decomposition is called lossy decomposition. The lossless join decomposition is also called lossless decomposition and the lossy join decomposition is called lossy decomposition.

R1 and R2 form a lossless decomposition of R if at least one of the following functional dependencies is in F⁺

$$R_1 \cap R_2 \rightarrow R_1$$

$$R_1 \cap R_2 \rightarrow R_2$$

If $R_1 \cap R_2$ forms a super key of either R1 or R2, the decomposition of R is a lossless decomposition. Attribute closure can be used to calculate super key.

Example: Consider the following schema

$$\text{bor_loan} = (\text{customer_id}, \text{loan_number}, \text{amount})$$

If it is decomposed into

$$\text{borrower} = (\text{customer_id}, \text{loan_number})$$

$$\text{loan} = (\text{loan_number}, \text{amount})$$

Here $\text{borrower} \cap \text{loan} = \text{bor_loan}$ and the decomposition satisfies lossless decomposition rule.

6.9 DEPENDENCY PRESERVATION

Let F be a set of functional dependencies on a schema R , and let R_1, R_2, \dots, R_n be a decomposition of R . The **restriction** of F to R_i is the set F_i of all functional dependencies in F^+ that include *only* attributes of R_i .

Example:

$$F = \{A \rightarrow B, B \rightarrow C\}$$

The restriction of F is $A \rightarrow C$, since $A \rightarrow C$ is in F^+ , even though it is not in F .

Even though $F' \neq F$, $F'^+ = F^+$ where $F' = F_1 \cup F_2 \cup F_3 \dots \cup F_n$. The decomposition having the property $F'^+ = F^+$ is a **dependency-preserving decomposition**.

Algorithm to test dependency preservation

```

compute  $F^+$ ;
for each schema  $R_i$  in  $D$  do
    begin
         $F_i :=$  the restriction of  $F^+$  to  $R_i$ ;
    end
 $F' := \emptyset$ 
for each restriction  $F_i$  do
    begin
         $F' = F' \cup F_i$ 
    end
compute  $F'^+$ ;
if ( $F'^+ = F^+$ ) then return (true)
else return (false);

```

The input to the algorithm is a set of decomposed relational schemas $D = \{R_1, R_2, R_3, \dots, R_n\}$ and a set F of functional dependencies. This algorithm is expensive since it requires the computation of F^+ .

6.10 BOYCE CODD NORMAL FORM (BCNF)

A relational schema R is in BCNF with respect to a set F of functional dependencies, if for all FD in F^+ of the form $\{\alpha \rightarrow \beta\}$, where $\alpha \subseteq R$ and $\beta \subseteq R$. at least one of the following holds:

- $\{\alpha \rightarrow \beta\}$ is a trivial dependency (i.e., $\beta \subseteq \alpha$)
- α is a super key for the schema R .

It is the strongest version of 3NF. A relation should be in 3NF and every determinant is a candidate key. It should be trivial. A candidate key is a unique identifier of each tuple. If a relation has only one candidate key then the relation is in 3NF and BCNF.

The difference between the 3NF and BCNF is that for a FD $A \rightarrow B$, the 3NF allows this dependency in a relation if B is a primary key and A is not a candidate key. But, BCNF allows this dependency in a relation only when A is a candidate key.

Example: Consider the client interview table.

Interview Table

<i>Client_no</i>	<i>Interviewdate</i>	<i>Interviewtime</i>	<i>Staffno</i>	<i>Roomno</i>
CR76	13-may-14	10.30	SG5	G101
CR 56	13-may-14	12.00	SG37	G102

The client interview relation has 3 candidate keys. They are,

1. {Client_no, Interview date}
2. {Staffno, Interview date, Interviewtime}
3. {Roomno, Interview date, Interviewtime}

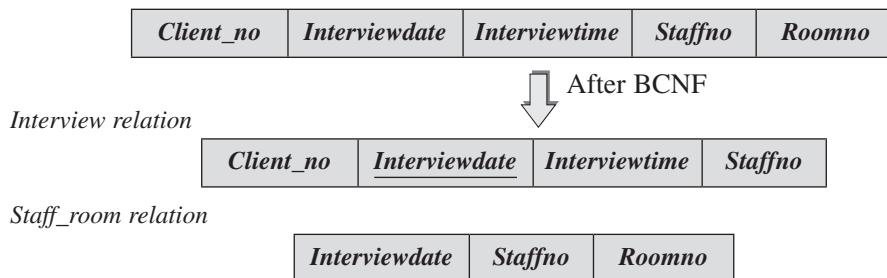
The Client_no and Interviewdate are selected as primary key. The dependency in this relation is,

- (i) {Client_no, interview date} → {Interviewtime, staffno, Roomno}
- (ii) {Staffno, Interviewtime, Interviewdate} → Client_no
- (iii) {Roomno, Interviewdate, Interviewtime} → {staffno, Client_no}
- (iv) {Staffno, Interview date} → Roomno.

The FD (i), (ii), and (iii) are all candidate keys of this relation and these dependencies won't cause problem for the relation.

But this relation is not in the BCNF form due to the presence of the {Staffno & Interviewdate} determinant and they are not a candidate key for the relation.

The basic requirement of the BCNF is all the determinants in a relation must be a candidate key.



Consider the relation project given below

Project

<i>ECode</i>	<i>Ename</i>	<i>ProjectCode</i>	<i>Hours</i>
101	Laka	P2	48
102	Sudha	P5	100
103	Subha	P6	250
104	Mathu	P7	300
104	Mathu	P8	100
101	Laka	P7	200

Candidate key can qualify as primary key and it can be single column or combination of columns

<i>ECode</i>	<i>Ename</i>	<i>ProjectCode</i>	<i>Hours</i>
--------------	--------------	--------------------	--------------

{ECode, ProjectCode} is the primary key. Therefore it acts as candidate key also. There are two candidate key in the above table

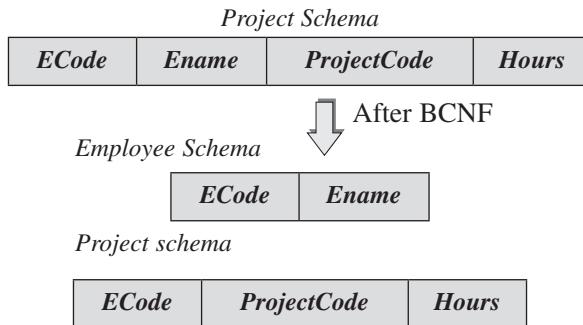
1. {ECode, ProjectCode}
2. {Ename, ProjectCode}

The relation has the following functional dependencies:

1. {ECode, ProjectCode} → Hours
2. {Ename, ProjectCode} → Hours
3. {Ename} → ECode
4. {ECode} → Ename

The multi valued candidate keys are {ECode, ProjectCode} and {Ename, ProjectCode}. They are composite in nature. The candidate key overlap since the attribute ProjectCode is common.

Find and remove the overlapping candidate keys. Place the part of the candidate key and the attribute dependent on it in a different table. Group the remaining attributes into a table.



So the above given project table is decomposed into:

1. Employee Relation
2. Project Relation, as follows

Employee Relation

<i>ECode</i>	<i>Ename</i>
101	Laka
102	Sudha
103	Subha
104	Mathu
104	Mathu
101	Laka

Project Relation

<i>ECode</i>	<i>ProjectCode</i>
101	P2
102	P5
103	P6
104	P7
104	P8
101	P7

6.10.1 Testing Decomposition of BCNF

To test if a relation is in BCNF the following can be done:

1. To check if a nontrivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF, compute ∞^+ and verify that it includes all attributes of R ; that is; it is a super key of R .
2. To check if a relation schema R is in BCNF, it is sufficient to check only if the dependencies in the set F do not violate BCNF, rather than to check all dependencies in F^+ .

If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF. It is not true when a relation is decomposed.

6.11 MULTI VALUED DEPENDENCIES & FOURTH NORMAL FORM (4NF)

An entity is in Fourth Normal Form (4NF) when it meets the requirement of being in Third Normal Form (3NF) and has no multi-valued dependencies.

6.11.1 Multi Valued Dependencies (MVD)

A Multi-valued Dependencies $X \rightarrow\rightarrow Y$ on relation schema R , where X and Y are both subset of R , specifies the following constraints any relation state r of R .

If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should exist in r with the following properties:

$$\begin{aligned}t_3[X] &= t_4[X] = t_1[X] = t_2[X] \\t_3[Y] &= t_1[Y] \text{ and } t_4[Y] = t_2[Y] \\t_3[Z] &= t_2[Z] \text{ and } t_4[Z] = t_1[Z]\end{aligned}$$

where Z is used to denote $(R - (X \cup Y))$

- $X \rightarrow\rightarrow Y$ holds, we say that X multi determines Y .
- A MVD is trivial Multi-valued Dependencies if
 - (a) Y is a subset of X .
 - (b) $X \cup Y = R$
- In MVD $X \rightarrow\rightarrow Y$, $X \rightarrow\rightarrow Z$ can be written as $X \rightarrow Y/Z$.

There are two types of Multi valued dependencies

Trivial MVD: A Multi valued dependency $A \rightarrow\rightarrow B$ in a relation R is said to be trivial if B is a subset of A .

Non trivial MVD: A Multi valued dependency $A \rightarrow\rightarrow B$ in a relation R is said to be non-trivial if B is not a subset of A .

Inference rules for functional and multi valued dependencies

- | | |
|--|--|
| 1. IR1 (reflexive rule for FDs) | : If $X \supseteq Y$, then $X \rightarrow Y$ |
| 2. IR2 (augmentation rule for FDs) | : $\{X \rightarrow Y\} = XZ \rightarrow YZ$ |
| 3. IR3 (transitive rule for FDs) | : $\{X \rightarrow Y, Y \rightarrow Z\} = X \rightarrow Z$ |
| 4. IR4 (complementation rule for FDs) | : $\{X \rightarrow\rightarrow Y\} = \{X \rightarrow\rightarrow (R - X \cup Y)\}$ |
| 5. IR5 (augmentation rule for MVDs) | : if $X \rightarrow\rightarrow Y$ and $w \supseteq Z$, then $wX \rightarrow\rightarrow Z$ |

- 6. IR6** (transitive rule for MVDs) : if $\{X \rightarrow \rightarrow Y, X \rightarrow \rightarrow Z\} = X \rightarrow \rightarrow (Z-Y)$
- 7. IR7** (replication rule for MVDs) : $\{X \rightarrow \rightarrow Y\} = \{x \rightarrow \rightarrow y\}$
- 8. IR8** (coalescence rule for FDs and MVDs): if $\{X \rightarrow \rightarrow Y\}$ and there exists w with the properties that
- w \cap y is empty
 - w \rightarrow z and
 - y \supseteq z, then x \rightarrow z

6.11.2 Fourth Normal Form

A relational schema R is in 4NF with respect to a set of dependency F if for every non trivial multi valued dependency $X \rightarrow \rightarrow Y$ in F^+ , X is a super key for R.

Example: Consider the following table Course_details

Course_details

<u>Course</u>	<u>Teacher</u>	<u>Book</u>
OOPS	Ravi	Programming in C++
	Sankar	C++ Complete Reference
	Ram	C++ Skills
	Ravi	C & C++
	Sankar	OOPS with C++
	Ram	Balagurusamy

The above table depicts that OOPs is taught by the teachers Ravi, Sankar and Ram. The teacher refers the book as given in the table. There is no association between teacher and book which means that the two attributes are independent.

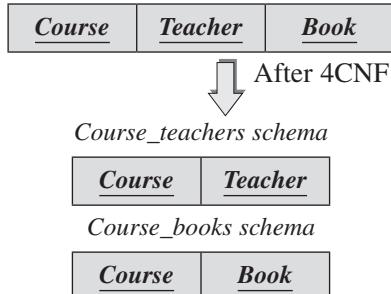
The MVD in the Course_details is

1. $\text{Course} \rightarrow \rightarrow \text{Teacher}$
2. $\text{Course} \rightarrow \rightarrow \text{Book}$

It can also be represented as **Course \rightarrow Teacher/Book**.

When multi valued dependency exist, adding new values to the table is difficult. So to reduce the table to fourth normal form, the MVDs available in the table are removed and placed in a new table along with the copy of the determinants.

So the Course_details table is decomposed



By applying 4 NF we can eliminate MVD. In order to remove the MVD in the Course_details table, the table is decomposed into two tables

1. Course_teachers table
2. Course_books table

Course_teachers

<u>Course</u>	<u>Teacher</u>
OOPS	Ravi
OOPS	Sankar
OOPS	Ram

Course_books

<u>Course</u>	<u>Book</u>
OOPS	Programming in C++
OOPS	C++ Complete Reference
OOPS	C++ Skills
OOPS	C & C++
OOPS	OOPS with C++
OOPS	Balagurusamy

Consider the following EMP table

EMP

<u>Ename</u>	<u>Pname</u>	<u>Dname</u>
Smith	X	CSE
Smith	Y	IT
Smith	X	IT
Smith	Y	CSE

The dependencies are

1. Ename → → Pname
2. Ename → → Dname

EMP is have MVD so to eliminate that 4NF is applied and the EMP relation is decomposed to

1. EMP_PROJECTS
2. EMP_DEPARTMENTS, as follows

EMP_PROJECTS

<u>Ename</u>	<u>Pname</u>
Smith	X
Smith	Y

EMP_DEPARTMENTS

<u>Ename</u>	<u>Dname</u>
Smith	CSE
Smith	IT

If the relation has nontrivial MVDs, then insert, delete and update operations on single tuple may cause additional tuples to be modified. To overcome these anomalies the relation is decomposed into 4NF.

Procedure for 4NF:

Input: A universal relation R and a set of functional and multi valued dependencies F

Set D := { R }

While there is a relation schema Q in D that is not in 4NF, do

{

Choose a relation schema Q in D that is not in 4NF;

Find a nontrivial MVD $X \rightarrow\!\!\!\rightarrow Y$ that violates 4NF;

Replace Q in D by two relation schemas (Q-Y) and (X ∪ Y);

};

6.12 JOIN DEPENDENCIES & FIFTH NORMAL FORM (5NF) ——————

6.12.1 Join Dependencies

A join dependency (JD) denoted by $\text{JD}(R_1, R_2, R_3 \dots R_n)$, specified on relation schema R, specifies a constraint on the state r of R. The constraint states that every legal state r of R should have non additive join decomposition into $R_1, R_2, R_3 \dots R_n$ i.e., for every such relation r we have

$$(\Pi_{R_1}(r), \Pi_{R_2}(r) \dots \Pi_{R_n}(r)) = r$$

JD denoted as $\text{JD}(R_1, R_2)$ implies an MVD $(R_1 \cap R_2) \rightarrow\!\!\!\rightarrow (R_1 - R_2)$.

Whenever a table is decomposed into two relations, the resulting relations have loss less join property. According to this property when a decomposed relation is rejoined then the resulting relation produces the original relation.

6.12.2 Fifth Normal Form (5NF)

A relational schema R is in fifth normal form or Project Join Normal Form (PJNF) with respect to a set F of functional, multivalued and join dependency if, for every nontrivial join dependency JD $(R_1, R_2, R_3 \dots R_n)$ in F^+ , every R_i is a superkey of R.

Example: Consider the Supply relation

Supply

<u>Sname</u>	<u>Part_name</u>	<u>Proj_name</u>
Smith	Bolt	X
Smith	Nut	Y
Adamsky	Bolt	Y
Walton	Nut	Z
Adamsky	Nail	X
Adamsky	Bolt	X
Smith	Bolt	Y

This supply relation does not have MVD since the attributes are not independent to each other. As the relation contains a join dependency, the relation is not in 5NF. In order to remove the join dependency, the relation is decomposed into three relations as follows:

R1

<u>Sname</u>	<u>Part_name</u>
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R2

<u>Sname</u>	<u>Proj_name</u>
Smith	X
Smith	Y
Adamsky	Y
Walton	Z
Adamsky	X

R3

<u>Part_name</u>	<u>Proj_name</u>
Bolt	X
Nut	Y
Bolt	Y
Nut	Z
Nail	X

Consider the following Department_subject_student relation

Department_subject_student relation

<u>Department</u>	<u>Subject</u>	<u>Student</u>
CSE	CS46	Ravi
IT	CS48	Sankar
EEE	EE52	Ram
CSE	CS43	Kumar
ECE	EC33	Arun
CSE	CS42	Arjune

The Department_subject_student relation contain a join dependency but does not contain MVD, since the attributes subject and student are not independent. In order to remove the join dependency, the relation is decomposed into three relations as follows:

Department_subject Schema

<u>Department</u>	<u>Subject</u>
CSE	CS46
IT	CS48
EEE	EE52
CSE	CS43
ECE	EC33
CSE	CS42

Department_subject Relation

<u>Department</u>	<u>Subject</u>
CSE	CS46
IT	CS48
EEE	EE52
CSE	CS43
ECE	EC33
CSE	CS42

Department_student Schema

<u>Department</u>	<u>Student</u>
CSE	Ravi
IT	Sankar
EEE	Ram
CSE	Kumar
ECE	Arun
CSE	Arjune

Department_student Relation

<u>Department</u>	<u>Student</u>
CSE	Ravi
IT	Sankar
EEE	Ram
CSE	Kumar
ECE	Arun
CSE	Arjune

Subject_student Scheme

<u>Subject</u>	<u>Student</u>
CS46	Ravi
CS48	Sankar
EE52	Ram
CS43	Kumar
EC33	Arun
CS42	Arjune

Subject_student Relation

<u>Subject</u>	<u>Student</u>
CS46	Ravi
CS48	Sankar
EE52	Ram
CS43	Kumar
EC33	Arun
CS42	Arjune

REVIEW QUESTIONS

1. Discuss the pitfalls of relational database design.
2. Define anomalies. Explain the different types of anomalies and illustrate how it affects the database.
3. Why is the concept of normalization needed in relational database design?
4. Explain the different Normal forms with suitable example.
5. Explain BCNF.
6. How BCNF differ from third Normal form.
7. Explain the concept of functional dependency with example.
8. Explain loss less decomposition and dependency preservation.
9. State Armstrong's axioms for functional dependency.
10. Explain closures of functional dependency and attribute closure.
11. Explain extraneous attributes.
12. How do extraneous attributes help in calculating canonical cover?



Chapter 7

Transaction Processing

7.1 INTRODUCTION

A transaction is a discrete unit of work that must be completely processed or not processed at all.

Example: transferring funds from a checking account to a saving account.

A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes. The concept of transaction provides a mechanism for describing logical units of database processing. Transaction processing systems are systems with large databases and hundreds of concurrent users that are executing database transactions. Transaction processing divides information process into individual, indivisible operations, called transactions that complete or fail as a whole; a transaction can't remain in an intermediate, incomplete, state. So other processes can't access the transaction's data until either the transaction has completed or it has been "rolled back" after failure.

Examples includes: systems for reservations, banking, credit card processing, stock markets, supermarket checkout, and other similar systems. They require high availability and fast response time for hundreds of concurrent users.

Transaction processing is designed to maintain database integrity of related data items in a consistent state. Transaction processing databases are databases that have been designed specifically to optimize the performance of transaction processing, which is often referred to as OLTP (Online Transaction Processing).

7.1.1 Transaction Concepts

A transaction is a logical unit of work. It begins with the execution of a BEGIN TRANSACTION operation and ends with the execution of a COMMIT or ROLLBACK operation.

A Sample Transaction (Pseudo code)

BEGIN TRANSACTION

 UPDATE ACC123 (BALANCE: =BALANCE-\$100);

 If any error occurred THEN GOTO UNDO;

 END IF;

```

UPDATE ACC456 (BALANCE: =BALANCE+$100);
If any error occurred THEN GOTO UNDO;
END IF;
COMMIT;
GOTO FINISH;
UNDO;
ROLLBACK;
FINISH;
RETURN;

```

In our example an amount of \$100 is transferred from account 123 to 456. It is not a single atomic operation; it involves two separate updates on the database. Transaction involves a sequence of database update operation. The purpose of this transaction is to transform a correct state of database into another correct state, without preserving correctness at all intermediate points.

Transaction management guarantees a correct transaction and maintains the database in a correct state. It guarantees that if the transaction executes some updates and then a failure occurs before the transaction reaches its planned termination, then those updates will be undone. Thus the transaction either executes entirely or totally cancelled. The system component that provides this atomicity is called transaction manager or transaction processing monitor or TP monitor. ROLLBACK and COMMIT are keys to the way it works.

1. COMMIT

- The COMMIT operation signals successful end of transaction.
- It tells the transaction manager that a logical unit of work has been successfully completed and database is in correct state and the updates can be recorded or saved.

2. ROLLBACK

- By contrast, the ROLLBACK operation signals unsuccessful end of transaction.
- It tells the transaction manager that something has gone wrong, the database might be in incorrect state and all the updates made by the transaction should be undone.

3. IMPLICIT ROLLBACK

- Explicit ROLLBACK cannot be issued in all cases of transaction failures or errors. So the system issues implicit ROLLBACK for any transaction failure.
- If the transaction does not reach the planned termination then we ROLLBACK the transaction else it is COMMITTED.

4. MESSAGE HANDLING

- A typical transaction will not only update the database, it will also send some kind of message back to the end user indicating what has happened.

Example: “Transfer done” if the COMMIT is reached, or “Error—transfer not done”

5. RECOVERY LOG

- The system maintains a log or journal or disk on which all particular about the updation is maintained.

- The values of before and after updation is also called as before and after images.
- This log is used to bring the database to the previous state in case of some undo operation.
- The log consist of two portions
 - an *active* or online portion
 - an *archive* or offline portion.
- The **online portion** is the portion used during normal system operation to record details of updates as they are performed and it is normally kept on disk.
- When the online portion becomes full, its contents are transferred to the **offline portion**, which can be kept on tape.

6. STATEMENT ATOMICITY

- The system should guarantee that individual statement execution must be atomic.

7. PROGRAM EXECUTION

- Program execution is a sequence of transactions.
- COMMIT and ROLLBACK terminate the *transaction*, not the application program.
- A single program execution will consist of a *sequence* of several transactions running one after another. A single program execution will consist of a sequence of several transactions running one after another as illustrated below

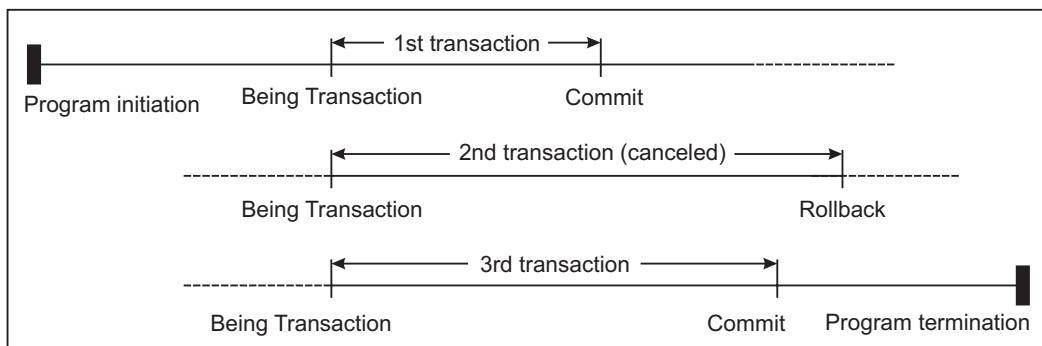


Fig. 7.1. Program execution is a sequence of transactions.

8. NO NESTED TRANSACTIONS

- An application program can execute a BEGIN TRANSACTION statement only when it has no transaction currently in progress.
- i.e., no transaction has other transactions nested inside itself.

9. CORRECTNESS

- Consistent means “not violating any known integrity constraint.”
- Consistency and correctness of the system should be maintained.
- If T is a transaction that transforms the database from state $D1$ to state $D2$, and if $D1$ is correct, then $D2$ is correct as well.

10. MULTIPLE ASSIGNMENT

- Multiple assignments allow any number of individual assignments (i.e., updates) to be performed “simultaneously.”

Example: UPDATE ACC 123 {BALANCE: = BALANCE - \$100}

UPDATE ACC 456 {BALANCE: = BALANCE + \$100}

- Multiple assignments would make the statement atomic.
- Current products do not support multiple assignments.

7.2 REASONS FOR TRANSACTION FAILURES

Failures can be classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. **A computer failure (system crash):** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures. Example: main memory failure.
2. **A transaction or system error:** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In some cases, the user may interrupt the transaction during its execution.
3. **Local errors or exception conditions detected by the transaction:** During transaction execution there may be occurrence of certain condition which may require the cancellation of the transaction. For example, data for the transaction may not be found. Notice that an exceptional condition such as insufficient account balance in a banking database may cause a transaction, such as a fund withdrawal, to be canceled. This exception should be programmed in the transaction itself, and hence would not be considered a failure.
4. **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.
5. **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. **Physical problems and disaster:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

7.3 TRANSACTION AND SYSTEM CONCEPTS

A transaction is an atomic unit of work that is either completed entirely or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts. The recovery manager is responsible to keep track of those details.

7.3.1 Transaction States and Additional Operations

- **BEGIN_TRANSACTION:** This marks the beginning of transaction execution.
- **READ or WRITE:** These specify read or write operations on the database items that are executed as part of a transaction.
- **END_TRANSACTION:** This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates serializability or for some other reason.
- **COMMIT_TRANSACTION:** This signals a successful end of the transaction so that any changes executed by the transaction can be safely committed to the database and will not be undone.
- **ROLLBACK (or ABORT):** This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

7.3.2 State Transition of a State Transition Diagram

Whenever a transaction is submitted to a DBMS for execution, either it executes successfully or fails due to some reasons. During its execution, a transaction passes through various states that are active, partially committed, committed, failed, and aborted. The following figure shows the state transition diagram that describes how a transaction passes through various states during its execution.

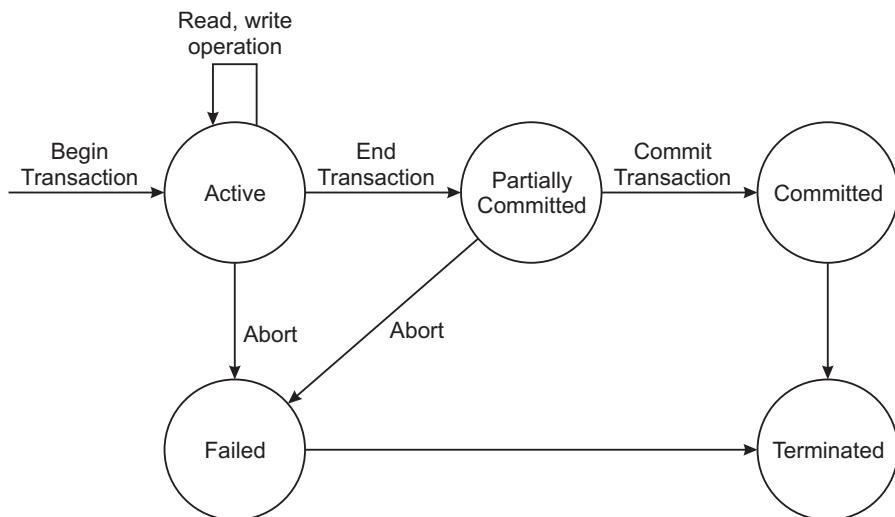


Fig. 7.2. State transition diagram of a transaction.

State transition diagram describes how a transaction moves through its execution states. A transaction goes into an active state immediately after it starts execution, where it can issue READ

and WRITE operations. When the transaction ends, it moves to the partially committed state. At this point, some recovery protocols need to ensure that a system failure does not occur. Once this check is successful, the transaction is said to have reached its commit point and enters the committed state.

Once a transaction is committed, the execution is completed successfully and all its changes can be recorded permanently in the database.

A transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

The terminated state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be restarted later, either automatically or after being resubmitted by the user.

7.4 THE SYSTEM LOG

To recover the system from transaction failure, the system maintains a log. This log keeps track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from failures. The log is kept on disk, so it is not affected by any type of failure except for disk or disastrous failure.

The log is periodically backed up to archival storage like tape to guard against such disastrous failures. The entry made in such log is called log records. The log records and the action performed are as follows:

1. **[start_transaction, T]:** Indicates that transaction T has started execution.
2. **[write_item, T , X , old_value , new_value]:** Indicates that transaction T has changed the value of database item X from old_value to new_value .
3. **[read_item, T , X]:** Indicates that transaction T has read the value of database item X .
4. **[commit, T]:** Indicates that transaction T has completed successfully, and affirms that its effect can be committed to the database.
5. **[abort, T]:** Indicates that transaction T has been aborted.

T refers to a unique transaction-id that is generated automatically by the system and is used to uniquely identify each transaction.

All READ operations are not required to be written to the system log. However, if the log is also used for other purposes like keeping track of all database operations then such entries can be included.

If the system crashes, we can recover to a consistent database state by examining the log. Because the log contains a record of every WRITE operation that changes the value of some database item, it is possible to undo the effect of these WRITE operations of a transaction T by tracing backward through the log and resetting all items changed by a WRITE operation of T to their old_values . Redoing the operations of a transaction may also be needed if all its updates are recorded in the log but a failure occurs before we can be sure that all these new_values have been written permanently in the actual database. Redoing the operations of transaction T is applied by

tracing forward through the log and setting all items changed by a WRITE operation of T to their new_values.

7.5. ACID PROPERTIES OR TRANSACTION PROPERTIES

Transactions possess several properties and these properties are often called as the ACID properties. These properties are responsible for enforcing the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed entirely or not performed at all.

The atomicity property requires the complete execution of the transaction. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

2. **Consistency preservation:** A transaction is consistency preserving if its complete execution take the database from one consistent state to another.

The preservation of consistency is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. State of a database is a collection of all the stored data items in the database at a given point in time. A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints that should hold on the database. A database program should be written in a way that guarantees the consistent state of the database before and after the execution of the transaction, assuming that there is no occurrence of interference with other transactions.

3. **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

Isolation is enforced by the concurrency control subsystem of the DBMS. If every transaction does not make its updates visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks. There have been attempts to define the *level of isolation* of a transaction. A transaction is said to have level 0 isolation if it does not overwrite the dirty reads of higher-level transactions. A level 1 isolation transaction has no lost updates; and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation or true isolation has no lost updates, no dirty reads and repeatable reads.

4. **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

Finally, the durability property is the responsibility of the recovery subsystem of the DBMS.

7.6 SERIALIZABILITY

Serializability is the generally accepted "criterion for correctness" for the interleaved execution of a set of transactions. Interleaved execution is considered to be correct if and only if it is serializable. The execution of a given set of transactions is serializable and correct if and only if it is equivalent to some serial execution of the same transactions, where:

- A serial execution is one in which the transactions run one at a time in some sequence.
- Guaranteed means that the given serial execution of the transaction always produces the same result as each other, no matter what the initial state of the database might be.

The definition is justified as follows:

1. Individual transactions are assumed to be correct: i.e., they are assumed to transform a correct state of the database into another correct state.
 2. Running the transactions one at a time in any serial order is correct ("any" serial order is correct because individual transactions are assumed to be independent of one another).
 3. It is thus reasonable to define an interleaved execution to be correct if and only if it is equivalent to some serial execution.
 4. Some times nonserializable interleaved execution of a transaction may also lead to correct result in some specific initial state of the database.
- It is found in the examples of lost update problem, uncommitted dependency problem and inconsistent analysis problem (discussed in next chapter) that interleaved execution was not serializable.
 - The use of strict two phase locking protocol forced serializable but it led to another problem called deadlock.

Given a set of transactions, any execution of those transactions, interleaved or otherwise, is called a schedule. Executing the transactions one at a time, with no interleaving, constitutes a serial schedule. Two schedules are said to be equivalent if and only if they produce the same result, no matter about the initial state of the database.

Two different serial schedules involving the same transactions will produce different results and hence the two different interleaved schedules involving those same transactions might also produce different results, and yet both are correct.

For example, suppose transaction A is of the form "Add 1 to x" and transaction B is of the form "Double x". Suppose that the initial value of x is 10. Then the serial schedule A then B gives x=22, the serial schedule B then A gives x=21.

The two phase locking theorem states that "*if all transactions obey two phase protocol, then all possible interleaved schedules are serializable*". The two phase locking protocol is as follows.

- Before operating on any object (e.g., a database tuple), a transaction must acquire a lock on that object.
- After releasing a lock, a transaction must never go on to acquire any more locks.

A transaction that obeys this protocol has two phases, a lock acquisition or "growing" phase and a lock releasing or "shrinking" phase. Shrinking phase is compressed into the single operation of COMMIT or RDLLBACK at end-of-transaction. Let I be an interleaved schedule involving

some set of transactions T_1, T_2, \dots, T_n . If I is serializable, then there exists at least one serial-schedule S involving T_1, T_2, \dots, T_n such that I is equivalent to S . S is said to be a serialization of I .

If A and B are any two transactions involved in some serializable schedule, then either A logically precedes B or B logically precedes A in that schedule. Conversely if the effect is not as if either A ran before B or B ran before A , then the schedule is not serializable and not correct.

In order to reduce resource utilization and thereby improving performance and throughput, real-world systems allows the construction of transactions that are not two-phase i.e., transactions "release locks early" (prior to COMMIT) and then acquire more locks. However, such transactions are a risky.

7.6.1 Schedules and Serializability

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a schedule or history. The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before a task is assigned to the database, first thing to be decided is the schedule that maintains the database in consistent state.

Transactions are programs; it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. Let us consider two operations: read and write. Between a read(Q) instruction and a write(Q) instruction on a data item Q , a transaction may perform an random sequence of operations on the copy of Q that is residing in the local buffer of the transaction. Therefore only read and write instructions are shown in any schedules, as shown in the figure below.

T_1	T_2
Read(A) Write(A)	
	Read(A) Write(A)

T_1	T_2
Read(B) Write(B)	
	Read(B) Write(B)

Fig. 7.3. Schedule 1: Schedule showing read and write operations.

7.6.2 CONFLICT SERIALIZABILITY

Let us consider a schedule S in which there are two consecutive instructions I_i and I_j , of transactions T_i and T_j , respectively ($i \neq j$). If I_i and I_j refer to different data items, then we can swap I_i and I_j without affecting the results of any instruction in the schedule. However, if I_i and I_j refer to the same data item Q , then the order of the two steps may matter. There are four cases to consider:

1. $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$. The order of I_i and I_j does not matter, since the same value of Q is read by T_i and T_j , regardless of the order.
2. $I_i = \text{read}(Q)$, $I_j = \text{write}(Q)$. If I_i comes before I_j , then T_i does not read the value of Q that is written by T_j in instruction I_j . If I_j comes before I_i , then T_i reads the value of Q that is written by T_j . Thus, the order of I_i and I_j matters.
3. $I_i = \text{write}(Q)$, $I_j = \text{read}(Q)$. The order of I_i and I_j matters for reasons similar to those of the previous case.
4. $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$. Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j . However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database.

If there is no other write(Q) instruction after I_i and I_j in S , then the order of I_i and I_j directly affects the final value of Q in the database state that results from schedule S . Thus, only in the case where both I_i and I_j are read, instructions do the relative order of their execution, not matter. We say that I_i and I_j conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

To illustrate the concept of conflicting instructions, we consider schedule in Fig 7.3. The write (A) instruction of T_1 conflicts with the read (A) instruction of T_2 . However, the write (A) instruction of T_2 does not conflict with the read (B) instruction of T_1 , because the two instructions access different data items. Let I_i and I_j be consecutive instructions of a schedule S .

If I_i and I_j are instructions of different transactions and I_i and I_j do not conflict, then we can swap the order of I_i and I_j to produce a new schedule S' . We expect S to be equivalent to S' , since all instructions appear in the same order in both schedules except for I_i and I_j , whose order does not matter. Since the write (A) instruction of T_2 in schedule of Figure 7.3 does not conflict with the read (B) instruction of T_1 , we can swap these instructions to generate an equivalent schedule as in figure 7.4. Regardless of the initial system state, both schedules (1 and 2) produce the same final system state. We continue to swap non conflicting instructions:

- Swap the read (B) instruction of T_1 with the read (A) instruction of T_2 .
- Swap the write (B) instruction of T_1 with the write (A) instruction of T_2 .
- Swap the write (B) instruction of T_1 with the read (A) instruction of T_2 .

T_1	T_2
Read(A)	
Write(A)	
	Read(A)
Read(B)	
	Write(A)
Write(B)	
	Read(B)
	Write(B)

Fig. 7.3. Schedule 2: Schedule showing read and write operations after swapping.

The final result of these swaps is a serial schedule and it's shown in Fig. 7.4. Schedule 3. This equivalence implies that, regardless of the initial system state, schedule 1 will produce the same final state as some other serial schedule. If a schedule S can be transformed into a schedule S' by a series of swaps of non conflicting instructions, then S and S' are conflict equivalent.

T_1	T_2
Read(A) Write(A) Read(B) Write(B)	Read(A) Write(A) Read(B) Write(B)

Fig. 7.4. Schedule 3: A serial schedule that is equivalent to schedule 1.

7.6.3 View Serializability

Consider two schedules S and S' , where the same set of transactions participates in both schedules. The schedules S and S' are said to be view equivalent if three conditions are met:

1. For each data item Q , if transaction T_i reads the initial value of Q in schedule S , then transaction T_i must, in schedule S' , also read the initial value of Q .
2. For each data item Q , if transaction T_i executes read(Q) in schedule S , and if that value was produced by a write(Q) operation executed by transaction T_j , then the read(Q) operation of transaction T_i must, in schedule S' , also read the value of Q that was produced by the same write(Q) operation of transaction T_j .
3. For each data item Q , the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S' .

The concept of view equivalence leads to the concept of view serializability. Schedule S is said to be view serializable if it is view equivalent to a serial schedule. Every conflict-serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable. The schedule in figure 7.5 is view-serializable but not conflict serializable

$T3$	$T4$	$T6$
Read(Q) Write(Q)	Write(Q)	Write(Q)

Fig. 7.5. Schedule 4: View Serializability.

7.6.4 Testing for Serializability

Algorithm can be used to test a schedule for conflict serializability. The algorithm looks at only the read and writes item operations in a schedule to construct a precedence graph. The precedence graph is a directed graph $G=(N,E)$ that consists of set of nodes $N=\{T_1, T_2, \dots, T_n\}$ and a set of directed edges $E=\{e_1, e_2, \dots, e_m\}$. For each transaction in the schedule there is one node in the graph.

Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$ where T_j is the starting node of e_i and T_k is the ending node of e_i . Such an edge is created if one of the operations in T_j appears in the schedule before some conflicting operation in T_k .

Algorithm

1. For each transaction T_i participating in schedule S, create a node labeled in the precedence graph.
2. For each case in S where T_j executes a read item(x) after T_i , executes a write-item(x), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a write-item(x) after T_i executes a read-item(x) create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where T_j executes a write-item(x) after T_i executes a write-item(x) create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
5. The schedule is serializable if and only if the precedence graph has no cycle.

REVIEW QUESTIONS

1. Explain the concept of transaction.
2. Explain the different states of a transaction with a neat diagram.
3. Explain ACID properties.
4. What is the need for system log in a transaction management system?
5. Explain serializability.
6. Explain the types of serializability with an example schedule.
7. Illustrate the algorithm to test for serializability.



Chapter 8

Concurrency Control

8.1 INTRODUCTION

Database management systems allow any transaction to access the same database at the same time. Such access is called concurrent access. In order to ensure that concurrent access does not interfere with each other, concurrency control mechanism is needed. Concurrency control can be defined as the process of managing simultaneous execution of transactions in a shared database to ensure the serializability of transactions.

8.2 THREE CONCURRENCY PROBLEMS

In order to implement a control mechanism to prevent the concurrent transactions interfering with each other, first thing to be known is the problem that will occur when the transactions interfere with each other. Concurrent transactions can lead to three concurrency problems

1. The lost update problem
2. The uncommitted dependency problem
3. The inconsistent analysis problem

8.2.1 The Lost Update Problem

To understand the lost update problem, consider two transactions, transaction A and transaction B. The tuple values accessed by the transactions A and B are as follows and diagrammatically given in figure 8.1.

- Transaction A retrieves some tuple t at time t_1 .
- Transaction B retrieves that same tuple t at t_2 .
- Transaction A updates the tuple at time t_3 on basis of values seen at time t_1 .
- Transaction B updates the same tuple at time t_4 based on values read at time t_2 .
- Transaction A's value is lost at time t_4 because Transaction B overwrites it without even looking at it.
- This is called lost update problem.

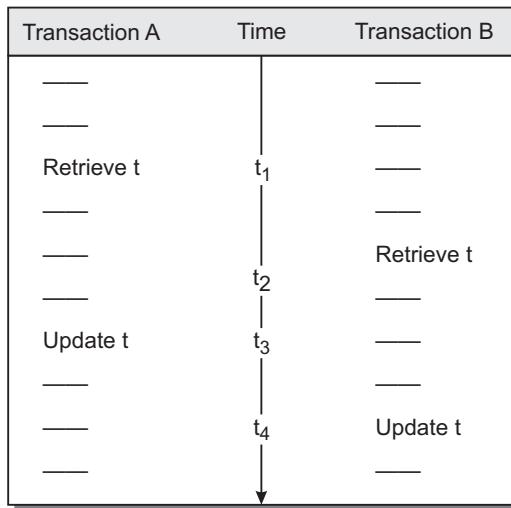


Fig. 8.1. Transaction A loses an update at time t_4 .

8.2.2 The Uncommitted Dependency Problem

The uncommitted dependency problem arises if one transaction is allowed to retrieve or update a tuple that has been updated by another transaction that has not yet been committed by other transaction. If the transaction is not committed there is a possibility for that transaction to Rollback. In such case the first transaction will have seen some data that no longer exists.

Example 1: Consider the following figure 8.2.

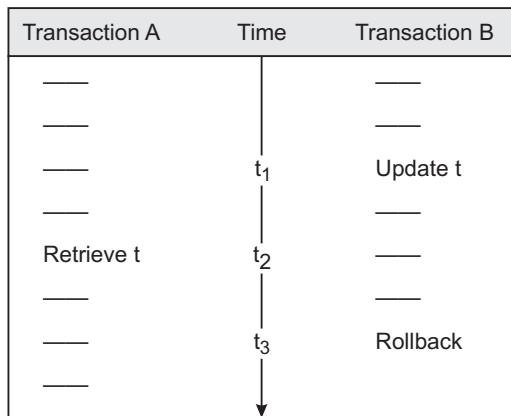


Fig. 8.2. Transaction A dependent on an uncommitted change at time t_2 .

- Transaction A sees an uncommitted update at time t_2 and depends on that uncommitted change.
- That update is undone by transaction B at time t_3 .

- Transaction A is therefore operating on a false assumption. A assumes that the tuple t has the same value as seen in time t_2 .
- 'A' might produce an incorrect result.
- The rollback of Transaction B might be due to system failure or crash.
- Transaction A might have terminated before knowing that transaction B is crashed and A have to rollback to previous stage.

Example 2: Consider the figure 8.3

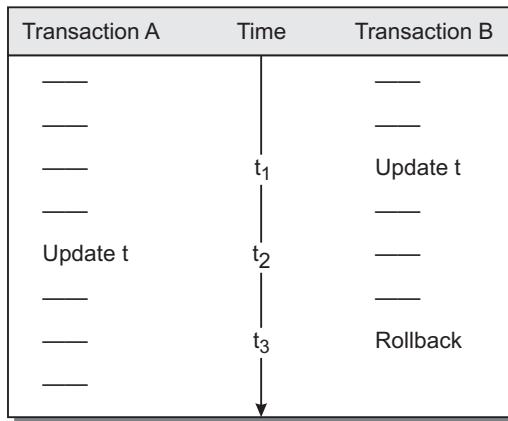


Fig. 8.3. Transaction A updates an uncommitted change at time t_2 .

- Transaction A updates an uncommitted change at time t_2 and losses that update at time t_3 .
- Transaction A becomes dependent on an uncommitted change at time t_2 and losses the update at time t_3 .
- The rollback at t_3 causes the tuple to be restored to its value prior to time t_1 .

8.2.3 The Inconsistent Analysis Problem

In the following figure 8.4., transaction A leads to inconsistent analysis problem.

- Two transactions A and B are operating on an account tuple.
- Transaction A is summing account balances; transaction B is transferring an amount 10 from account 3 to account 1.
- The result produced by transaction A (110) is wrong, if transaction A write the result back to the database then the database will be in inconsistent state.
- Transaction A has seen an inconsistent state of the database and has performed an inconsistent analysis.
- Transaction B commits all of the transaction before transaction A sees. This is not uncommitted dependency.
- Inconsistent analysis of transaction A has lead the database to inconsistent state.

ACC 1 40	ACC 2 50	ACC 13 30
Transaction A	Time	Transaction B
Retrieve ACC 1: Sum = 40	t_1	
Retrieve ACC 2: Sum = 90	t_2	
	t_3	Retrieve ACC 3
	t_4	Update ACC 3: 30 → 20
	t_5	Retrieve ACC 1
	t_6	Update ACC 1: 40 → 50
	t_7	Commit
Retrieve ACC 3: Sum = 110, not 120	t_8	

Fig. 8.4. Inconsistent Analysis problem created by Transaction A.

8.2.4 Primary Operations Leading to Concurrency Problems

The primary operations of database are retrievals and updates. They are also referred to as read and write operation respectively. If A and B are two concurrent operations, problem can occur if A and B want to read and write the same database object, say tuple t.

There are four possibilities.

1. **RR:** A and B both want to read t. Read does not interfere with each other. So there will be no problem in this case.
2. **RW:** A wants to read tuple t and B wants to write t. If B is allowed to write, then it may lead to inconsistent analysis problem. After B has performed write, if A reads t again, it will find a different value. This problem is called non repeatable read. This is also due to RW conflict.
3. **WR:** A writes t and B wants to read t. If B is allowed to perform its read, it may lead to uncommitted dependency problem. If B's read is allowed, then it is said to be dirty read. WR leads to uncommitted dependency problem and dirty read.

4. **WW:** A writes t and B also wants to write t. If B is allowed to perform its write then it may lead to lost update problem. If B is allowed to write, it is said to be dirty write. WW leads to lost update problem and dirty write.

8.3 LOCKING

The problems that occur due to concurrency can be solved by concurrency control mechanism called locking. The idea behind locking is, when some transaction A needs a tuple from being updated by any other transaction then a lock can be used. When a lock is acquired on an object, it remains in the stable state as long as required.

The most commonly used locks are:

1. Exclusive Lock or X lock or write lock
2. Shared lock or S lock or read lock

Rules in granting locks to transactions

- If a transaction A holds an exclusive lock (X) on tuple t, then a request from some another transaction B for a lock of either X lock or S lock cannot be immediately granted.
- If transaction A holds a shared lock (S) on tuple t, then
 - A request from another transaction B for X lock on tuple t can be granted immediately.
 - A request from another transaction B for S lock on tuple t can be granted immediately.

This rule can be depicted by means of a lock type compatibility matrix.

	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

Fig. 8.5. Compatibility matrix.

The matrix is constructed as follows:

For any tuple t, the column heading indicates the lock held by the transaction A. Left side entries indicate the request for lock of another transaction B. A dash indicates no lock. N indicates a conflict, i.e., B's request cannot be granted immediately. Y indicates compatibility, i.e., that B's request can be granted immediately.

8.3.1 Locking Protocol or Data Access Protocol

The concurrency problems like last update problem, uncommitted dependency problem and inconsistent analysis problem can be overcome by locking protocol or data access protocol. This protocol makes use of the X and S locks to solve the concurrency problem.

1. A transaction that wishes to retrieve a tuple must first acquire an S lock on that tuple.
2. A transaction that wishes to update a tuple must first acquire an X lock on that tuple. If it already holds an S lock, then to perform update operation it must upgrade level of S lock to X lock.

3. If the lock requested by the transaction cannot be immediately granted because of some conflict with the lock held by some other transaction A, B goes to wait state. B will wait until lock is granted. In some case B may not be granted lock as soon A releases the lock because there may be some other transaction waiting for the lock. The system should check that B is not waiting forever. Transaction waiting for a lock forever may lead to starvation. In order to overcome problems like starvations system should satisfy the request made by the transactions for the lock should be granted on first come first served basis.
4. X locks are released at the end of transaction. S locks can be released immediately.
5. Request to locks are implicit. Similarly request to upgrade the lock level is also implicit.

8.4 STRICT TWO PHASE LOCKING PROTOCOL

The strict two phase locking protocol solves the concurrency problems. The control mechanism for the three concurrency problems like last update problem, uncommitted dependency problem and inconsistent analysis problem through locking protocol are discussed below:

8.4.1 Last Update Problem

Consider the following figure 8.6.

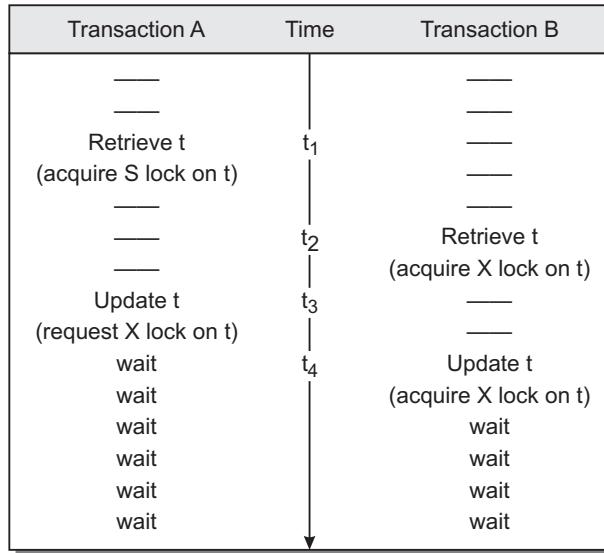


Fig. 8.6. Resolving lost update problem but leading to deadlock.

Transaction A's update at time t_3 is not accepted, because it is an implicit request for an X lock on t and will conflict with S lock already held by transaction B. So A goes to waiting state. Due to some reason B also goes to waiting state at time t_4 . Now both the transactions are unable to proceed. Here the lost update problem is solved but it leads to another problem called deadlock.

8.4.2 The Uncommitted Dependency Problem

Consider the following figures 8.7 and 8.8.

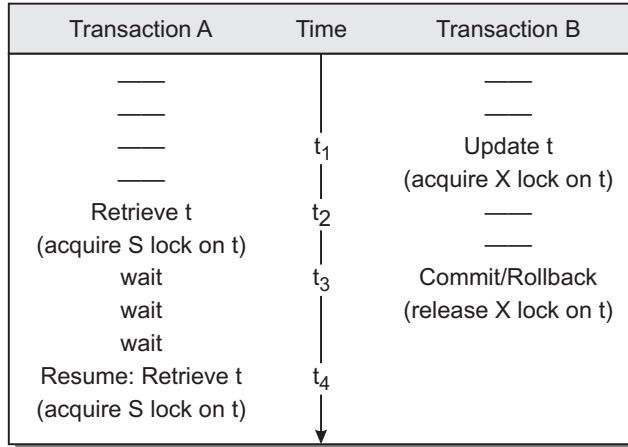


Fig. 8.7. Resolving uncommitted dependency in case of retrieval.

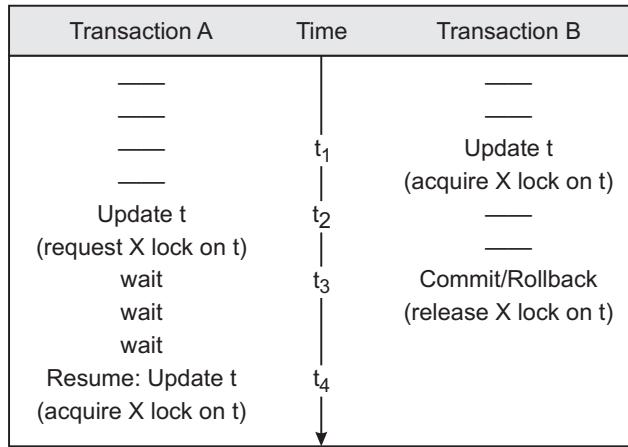


Fig. 8.8. Resolving uncommitted dependency in case of updation.

In both the cases the transaction A's operation is not accepted because it is an implicit request for a lock on tuple t. If such request is granted it conflicts with the X locks that is already held by B. So A goes to waiting state. A remains in waiting state until B reaches its termination. When B's lock is released and A is able to proceed, A sees a committed value and does not depend on uncommitted value. Therefore the uncommitted dependency problem is solved.

8.4.3 The Inconsistent Analysis Problem

Consider the figure 8.9.

ACC 1 40	ACC 2 50	ACC 13 30
Transaction A	Time	Transaction B
—		—
—		—
Retrieve ACC 1: (acquire S lock on ACC 1)	t_1	—
Sum = 40		—
—		—
Retrieve ACC 2: (acquire S lock ACC t)	t_2	—
Sum 90		—
—		—
—	t_3	Retrieve ACC 3 (acquire S lock on t)
—		—
—	t_4	Update ACC 3: (acquire X lock on ACC 3) 30 → 20
—		—
—	t_5	Retrieve ACC 1 (acquire S lock on ACC 1)
—		—
—	t_6	Update ACC 1: (request X lock on ACC 3) wait
—		wait
—	t_7	wait
—		wait
Retrieve ACC 3: (acquire S lock on ACC 3)		
Sum = 110, not 120		
wait		
wait		

Fig. 8.9. Resolving inconsistent analysis problem but leading to deadlock.

Transactions B's update at time t_6 is not accepted, because it is an implicit request for an X lock on ACC 1. Such a lock conflicts with the S lock held by A. So B goes to a waiting state. Transaction A's retrieval operation at time t_7 is also not accepted, because it is an implicit request for an S lock on ACC 3, and such a request conflicts with the X lock held by B. So A goes to waiting state. This approach solves the original inconsistent analysis phase but leads to a new problem called deadlock.

8.5 DEADLOCK

The two phase locking protocol discussed in the previous section can be used to solve the three concurrency problem using the help of X and S locks but introduces a new problem called deadlock.

A deadlock is a condition in which two or more transactions in a set are waiting simultaneously for locks held by some other transaction in the set. Neither transaction can continue because each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. A deadlock is also called a circular waiting condition where two transactions are waiting directly or in-directly for each other.

8.5.1 Deadlock Detection

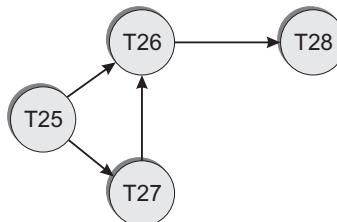
A simple way to detect a state of deadlock is with the help of wait-for graph. This graph is constructed and maintained by the system. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction T1 is waiting to lock an item X that is currently locked by a transaction T2, a directed edge ($T1 \rightarrow T2$) is created in the wait-for graph. When T2 releases the lock(s) on the items that T1 was waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. Then each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

To illustrate these concepts, consider the following wait-for graph in figure 8.10. Here:

Transaction T25 is waiting for transactions T26 and T27.

Transactions T27 is waiting for transaction T26.

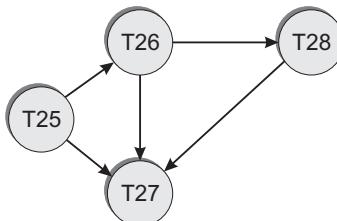
Transaction T26 is waiting for transaction T28.



Representing no deadlock state

Fig. 8.10. Wait for graph with no deadlock.

This wait-for graph has no cycle, so there is no deadlock state. Suppose now that transaction T28 is requesting an item held by T27. Then the edge $T28 \rightarrow T27$ is added to the wait -for graph, resulting in a new system state as shown in figure 8.11.



Representing a deadlock state

Fig. 8.11. Wait for graph with deadlock.

This time the graph contains the cycle.

T26 → T28 → T27 → T26

It means that transactions T26, T27 and T28 are all deadlocked.

Invoking Deadlock Detection Algorithm

The invoking of deadlock detection algorithm depends on two factors:

1. How often does a deadlock occur?
2. How many transactions will be affected by the deadlock?

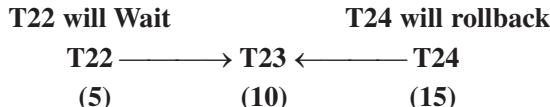
If deadlocks occur frequently, then the detection algorithm should be invoked more frequently than usual. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

8.5.2 Deadlock Avoidance

Avoiding deadlock is better than dealing with them when they occur. Deadlock avoidance is possible by modifying the locking protocol in various ways. The Wait-Die and Wound-Wait deadlock detection strategies can be explained as follows:

- Every transaction is time stamped with its start time.
- When transaction A requests a lock on a tuple that is already locked by transaction B, then
 - **Wait-Die:** Transaction A waits if it is older than transaction B, otherwise it dies. i.e., transaction A is rolled back and restarted.

Example: Suppose that transactions T22, T23 and T24 have timestamps 5, 10 and 15 respectively. If T22 requests a data item held by T23 then T22 will wait. If T24 requests a data item held by T23 then T24 will be rolled back.

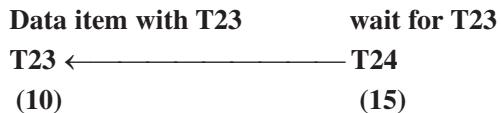


The wait-die scheme is non-preemptive scheme because only a transaction requesting a lock can be aborted. As a transaction grows older and its priority increases, it tends to wait for more and more younger transactions.

Wound-Wait: Transaction A waits if it is younger than transaction B, otherwise it wounds transaction B, i.e., transaction B is rolled back and restarted.

Example: Returning to our previous example, with transactions T22, T23 and T24, if T22 requests a data item held by T23 then the data item will be preempted from T23 and T23 will be rolled back. If T24 requests a data item held by T23, and then T24 will wait.





This scheme is based on a preemptive technique and is a counterpart to the wait-die scheme. In the wait-die scheme, lower priority transactions can never wait for higher priority transactions. In the wound-wait scheme, higher priority transactions never wait for lower priority transactions. In either case no deadlock cycle can develop.

When a transaction is aborted and restarted, it should not be given the same timestamp that it had originally. Reissuing timestamps in this way ensures that each transaction will eventually become the oldest transaction, and thus the one with the highest priority, and will get the locks that it requires.

If a transaction has to be restarted, it retains its original timestamp.

The terms Wait and Wound in each strategy indicates what happens if transaction A is older than transaction B. Wait-Die means all waits consists of older transactions waiting for younger ones, Wound-wait means all waits consist of younger transactions waiting for older ones. Whatever strategy is followed it is easy to avoid deadlock from being occurring. This avoidance technique also guarantees all the transaction to reach proper termination. There will be no starvation, and no transaction will be restarted again and again forever. The only demerit with the approach is that it does too many rollbacks.

8.5.2.1 Other Technique that can be used to Prevent the Deadlock Situation

Avoiding holding multiple locks: If no transaction attempts to hold more than one lock, then no deadlock can occur since the Hold and Wait condition is invalidated. In some cases it is possible but in some other cases there may be genuine needs to lock multiple objects at the same time. In such cases, it is necessary to minimize the situations where transactions are in need of multiple locks.

Using coarser-grained locks: One way to modify a program to avoid holding multiple locks is to replace uses of multiple locks by a single lock. This can also sometimes avoid the Circular Wait Condition even when threads hold multiple locks. Combining into coarser-grained locks may result in threads waiting when they do not really need to; hence concurrency is reduced, which may be undesirable. We can always resolve the new deadlocks by combining more locks, until we ultimately use only a single lock for the whole program. But, this reduces the concurrency so much that it is generally undesirable, and anyway library code will often use its own locks. On the other hand, if transaction hold locks for very short periods of time, then having a single lock is reasonable.

Using finer-grained locks: A common use of this technique is to replace a lock on a whole object by a number of locks for its parts. E.g., instead of locking a collection object like an array, the individual objects in the collection may be locked as appropriate. Just like coarser-grained, this may introduce deadlocks. Finer-grained locking tends to be more complex, so this method is only recommended where it is quite clear that it will work.

Minimizing the holding of locks: The default style in Java holds locks on objects whenever a method for the object is executing. Often some methods can be safely unsynchronized, and this may prevent deadlocks. It is not safe to do this if the method accesses multiple instance variables

of the object, or the same variable many times, and it assumes that the object has not changed in between. Similarly, this is not safe if the object is modified temporarily to an inconsistent state without holding the lock. Often only a part of a method needs to be synchronized, since the other parts do not access instance variables, or do so only safely.

Reordering lock acquisition: If we require transactions to always acquire locks in a particular order, then no deadlock can occur. The Circular Wait Condition is avoided, since we cannot have a circular chain if transactions can only wait for locks which come after the locks they've already acquired. This may be easy, or it may require major restructuring the code. Or, often it is practically impossible, since there is no practical way to predict which locks may be acquired in future.

Releasing and reacquiring locks in order: The basic idea is to always acquire locks in a particular order.

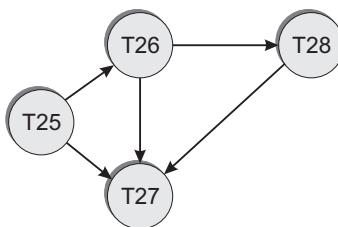
8.5.3 Deadlock Recovery

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Choosing which transaction to abort is known as Victim Selection.

8.5.3.1 Choice of Deadlock Victim

In the figure 8.11, Transactions T26, T28 and T27 are deadlocked. In order to remove deadlock one of the transaction out of these three transactions must be roll backed. We should roll back those transactions that will incur the minimum cost. When a deadlock is detected, the choice of which transaction to abort can be made using following criteria:

- The transaction which have the fewest locks
- The transaction that has done the least work
- The transaction that is farthest from completion



Representing a deadlock state

Fig. 8.12. Wait for graph.

Rollback: Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back. The simplest solution is a total rollback; Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. But this method requires the system to maintain additional information about the state of all the running system.

Problem of Starvation: In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result this

transaction never completes can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks as the cost factor.

8.5.3.2 Detection Versus Prevention

In prevention-based schemes, the abort mechanism is used preemptively in order to avoid deadlocks. On the other hand in detection-based schemes, the transactions in a deadlock cycle hold locks that prevent other transactions from making progress. Deadlock detection is preferable, if different transactions rarely access the same items at the same time. This can happen if the transactions are short and each transaction locks only few items or if the transaction load is light. On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is quite heavy, it may advantageous to use deadlock prevention scheme.

8.6 ISOLATION LEVEL

In a concurrent environment isolation is a property that defines how and when the changes made by one operation become visible to other concurrent operations. Isolation gives the querying user the feeling that he owns the database. It does not matter that hundreds or thousands of concurrent users work with the same database and the same schema. These other users can generate new data, modify existing data or perform any other action. The querying user must be able to get a complete, consistent picture of the data, unaffected by other users' actions.

Suppose a transaction in one program updates data in another database, but before the transaction commits, another program reads the same data. Will the second program read the new but uncommitted data, or will it read the old data? In any database such critical situation is handled by isolation level defined in the database.

If the transaction allows other programs to read uncommitted data, the second program will read the updated data. This may improve performance because the second program doesn't have to wait until the transaction ends to read the data. If the second program reads data that is not committed, it would have either read incorrect data or data that doesn't exist.

To allow users to choose what is best for their application, DBMSs provide isolation levels defining how subsequent processes are to be isolated from other concurrently executing processes. Isolation levels can be set by the database administrator and apply to all transactions within the database. They can also be set within an application or before an individual transaction. An isolation level specifies:

- The degree to which the rows read and updated by the application are available to other concurrently executing processes
- The degree to which updates from other concurrently executing application processes are available to the application

DBMSs allow four isolation levels. The isolation levels listed below are from lowest to highest isolation. Each level adds to the features of the previous level. Higher isolation levels offer a greater degree of data integrity, but at the cost of decreased concurrency since they hold locks longer.

1. **UNCOMMITTED READ:** This allows minimum isolation from concurrent transactions. The transaction can read data that has been changed by concurrent transactions even before they commit. This isolation level is also called *dirty reads*. This isolation level ensures

the quickest performance, as data is read directly from the table's blocks with no further processing, verifications or any other validation.

2. **COMMITTED READ:** The transaction will read committed data only: it will not read uncommitted data.
3. **REPEATABLE READ:** Places an additional restriction that applies when the same rows are read multiple times during the course of the transaction. This ensures that when the same rows are read a subsequent time, they are the same.
4. **SERIALIZABLE:** This isolation level offers the highest degree of isolation from concurrent transactions. All reads in the transaction only see data committed before the transaction began and never see concurrent transaction changes committed during transaction execution.

8.6.1 Phantoms

Special problems that occur when transactions operate at less than the maximum isolation level are called phantoms.

Example: If Transaction A's task is to calculate average account balance of a customer. In case if there are three accounts with a balance of Rs.100, the average will be Rs.100. Suppose another transaction say transaction B is adding another account to customer with balance of Rs.300 and commits the transaction. Assume that new account is added after A has completed its calculation.

Now if transaction A decides to scan the account again. The result will be different. Both the transactions has followed strict two phase locking but still there is some mistake, transaction A sees something that did not exist first. This is called phantoms.

To avoid phantom access path should be locked. In the above example access path will be index entry of customer. If it is locked transaction B cannot add new entry. Access path should be locked with X lock. So that phantoms can be prevented.

8.7 INTENT LOCKING

The locking protocol explained in the previous sections describes working procedure and need for lock on an individual tuple. When locks are used to lock the entire relation or database then there will be no need for individual lock for each tuple. So locks can be either large to lock the entire relation or database or it can be small to lock the tuple or specific component of a tuple.

For example, if transaction A has an X lock on the entire relation then there is no need to set X locks on individual tuples within that table. Moreover, no concurrent transaction will be able to obtain any lock on that table or tuples within that table.

Suppose transaction T request for an X lock on some relation R. when the request of T is received the system must be capable of telling the details of already available lock details on the tuples of the relation. Only with those details it can be decided whether the lock requested by T can be granted without any conflict at that time or we have to make T to wait. It is undesirable to examine ever tuple in R to find the existence of any lock. In order to solve such situation another protocol called intent locking protocol is introduced. According to this protocol no lock can be acquired on the tuple before acquiring an intent lock on the entire relation that contains the tuple.

In addition to X and S locks, three more intent locks are introduced. Intent locks are used for relations and not on individual tuples. The types of intent locks are:

1. Intent shared (IS) locks
2. Intent exclusive (IX) locks
3. Shared Intent exclusive (SIX) locks

Intent shared (IS) locks: T intends to set S locks on individual tuples in R, in order to guarantee the stability of those tuples while they are being processed.

Intent exclusive (IX) locks: Same as IS, plus T might update individual tuples in R and will therefore set X locks on those tuples.

Shared (S) locks: T can tolerate concurrent readers, but not concurrent updaters, in R.

Shared Intent exclusive (SIX) locks: Combines S and IX, that is, T tolerate concurrent readers, but not concurrent updaters, in R. In addition T might update individual tuples in R and will therefore set X on those tuples.

Exclusive (X) locks: T cannot tolerate any concurrent access to R at all.

The definitions of the locks are given in a lock type compatibility matrix

	X	SIX	IX	S	IS	-
X	N	N	N	N	N	Y
SIX	N	N	N	N	Y	Y
IX	N	N	Y	N	Y	Y
S	N	N	N	Y	Y	Y
IS	N	Y	Y	Y	Y	Y
-	Y	Y	Y	Y	Y	Y

Important things to be considered with Intent locking protocol

- (i) Before a given transaction acquires an S lock on a given tuple, it must first acquire an IS or stronger lock on the relation containing that tuple.
- (ii) Before a given transaction acquires an X lock on a given tuple, it must first acquire an IX or stronger lock on the relation containing that tuple.

Lock escalation is implemented in many systems to help in balancing the conflicting requirements of high concurrency and low lock management overhead. The idea behind lock escalation is that when some prescribed threshold is reached, the system automatically replaces a collection of locks of fine granularity by a single lock of coarse granularity.

8.8 DROPPING ACID

First let us have a fast recap of the ACID properties discussed earlier in section 7.5. ACID stands for Atomicity, Consistency, Isolation and Durability

Atomicity: Any given transaction executes as a whole or will not execute.

Correctness: Any given transaction transforms a correct state of the database into another correct state, without necessarily preserving correctness at all intermediate state.

Isolation: Any given transaction's update is hidden from all other transactions, until the given transaction commits.

Durability: Once a given transaction commits, its updates survives in the database even if there is a subsequent system crash.

8.8.1 Immediate Check for Constraint

All the integrity constraints must be checked immediately at the end of statement, it should not be postponed to end of transaction due to the following reasons:

1. Database has a collection of data. If there are no inconsistencies then the system is the best. Results of an inconsistent database cannot be trusted. Isolation property states that no more than one transaction will ever see any particular inconsistency. Inconsistency cannot be tolerated even if they are never visible to more than one transaction at a time. Therefore constraints must be enforced in the first place.
2. Guarantee cannot be given that an inconsistent transaction is seen by just only one transaction. Only if certain protocols are enforced then we can guarantee that transactions are isolated from one another. For example, transaction A sees an inconsistent state of the database and so writes inconsistent data to some file F, and transaction B then reads that information from file F, then A and B are not isolated from each other.
3. The relational constraints are checked immediately but the constraints set to the database are checked only at the end of transaction. The database constraints must also be checked immediately.
4. The databases should be in consistent state at all times not only at the transaction boundaries.

The conventional wisdom is that database constraint checks have to be deferred. Consider suppliers and parts database with constraint that "Supplier S1 and part P2 are in the same city". If supplier S1 moves, say from Bangalore to Chennai, then part P1 also should move from Bangalore to Chennai. The conventional solution to this problem is to wrap the two updates up into a single transaction

```
BEGIN TRANSACTION;
UPDATE SUPPLIERS WHERE SUPPLIER_NUMBER
    = SUPPLIER_NUMBER ('S1'{CITY:='CHENNAI'});
UPDATE PARTS WHERE PART_NUMBER
    = PART_NUMBER ('P1'{CITY:='CHENNAI'});
COMMIT;
```

In this conventional solution, the constraint is checked at COMMIT, and the database is inconsistent between the two update operations. If the transaction performing the UPDATEs were to ask the question "Are supplier S1 and part P1 in the same city?" between the two update operations, the answer will be no.

In case of multiple assignment operators several assignments will be performed as a single operation without any integrity checking being done until all of the assignments have been executed. In the above example, we should perform the two update operation as a single operation

```
UPDATE SUPPLIERS WHERE SUPPLIER_NUMBER
    = SUPPLIER_NUMBER ('S1'){CITY:='CHENNAI'} ;
UPDATE PARTS WHERE PART_NUMBER
    = PART_NUMBER ('P1'){CITY:='CHENNAI'} ;
```

In this case no integrity is done until both updates have been done. There is no possibility for the transactions to see an inconsistent state of the database between the two UPDATES. If multiple assignments were supported there should be no need for deferred checking in the traditional sense.

To better understand the ACID properties and the need of them, consider a banking system consisting of several accounts and a set of transactions that access and updates those accounts. Access to the database is accomplished by two operations given below:

- (i) **Read(X):** this operation transfers the data item 'X' from the database to a local buffer belonging to the transaction that executed the read operation.
- (ii) **Write(X):** this operation transfers the data item 'X' from the local buffer of the transaction that executed the write operation to the database.

Let T be a transaction that transfers Rs. 5,000/- from account A to account B. This transaction can be defined as

T: Read (A);

```
A: = A - 5000;
Write (A);
Read (B);
B = B + 5000;
Write (B);
```

It will be better if we choose the order of ACID property as CIDA

Consistency

The consistency requirement here is that the sum of A and B is unchanged by the execution of the transaction. Before the execution of the transaction and after the execution of the transaction, the sum of A and B should be the same. Ensuring the consistency for the individual transaction is the responsibility of the application programmer who codes the transaction.

Isolation

If several transactions are executed concurrently, operations may be interleaved in some undesirable way resulting in an inconsistent state.

For example, the database is temporarily inconsistent while the transaction transfer fund from A to B is executed. Amount is deducted from transaction A and increased to transaction B. if another transaction reads A and B at an intermediate point and computes A+B, it will arrive at an inconsistent value. Further more if that transaction performs updates on A and B, the database may be left in inconsistent state even after both transactions have completed.

To avoid the problem of concurrent execution, transactions should be executed in isolation. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that could have been obtained if the transactions are executed serially. Ensuring the isolation property is the responsibility of concurrent control component of DBMS.

Durability

The durability property guarantees that, once a transaction completes successfully, all updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. Durability is guaranteed by ensuring either.

- (i) The updates carried out by the transaction have been written to disk before the transaction completes.
- (ii) Information about the updates carried out by transaction and written to disk is sufficient for the database to reconstruct the updates when the database system is restarted after the failure.

Ensuring the durability is the responsibility of the recovery-management component of the DBMS.

Atomicity

Let us assume that before transaction takes place, the balance in the account A is ₹ 50,000 and that in the account B is ₹ 25,000. Now during the execution of the transaction, a failure occurred that prevented the successful completion of the transaction. The failure occurred after the Write (A) operation was executed but before the execution of Write (B). In this case the value of the accounts A and B reflected in the database are ₹ 45,000 and ₹ 25,000 respectively. The 5000 rupees that we took from account A is lost thus leading the database to inconsistent state. Such inconsistencies should not be permitted in the database.

To ensure atomicity the database system keeps track of the old values of any data on which a transaction performs a write. If the transaction does not complete its execution, the database restores the old values. Ensuring atomicity is the responsibility of the transaction management component of the DBMS.

8.9 SQL FACILITIES

SQL standard does not provide any explicit locking facilities. It provides guarantee against the interference of transaction among the concurrently executing transactions. The updates made by transaction T1 should not be made visible to any other transaction T2 until and unless transaction T1 commits. It is assumed that all transactions executes at isolation level read committed, repeatable read or serializable. Transaction executing at read uncommitted level are allowed to perform dirty read but are required to be read only.

There are four possibilities in isolation level, Serializable, Repeatable read, Read committed and Read uncommitted. Serializable is the default case. Other level should be specified. If all transactions execute at isolation level Serializable then the interleaved execution of any set of concurrent transactions is guaranteed to be serializable. If any transaction executes at a lesser isolation level, then Serializability can be violated in a variety of different ways. Three main

violations are dirty read, nonrepeatable read and phantoms. The violations of various isolation levels are summarized as follows:

<i>Isolation Level</i>	<i>Dirty Read</i>	<i>Nonrepeatable Read</i>	<i>Phantoms</i>
Read uncommitted	Y	Y	Y
Read committed	N	Y	Y
Repeatable read	N	N	Y
Serializable	N	N	N

REVIEW QUESTIONS

1. Define concurrency. Explain with suitable examples the problem associated with concurrent transactions.
2. Explain the primary operations leading to concurrency problem.
3. What is need for a Lock. Explain the locking protocol in detail.
4. Explain strict two phase locking protocol and how this protocols solves the concurrency problem.
5. Define deadlock. Explain deadlock detection, avoidance and recovery techniques.
6. Explain the need for introducing isolation level.
7. Explain intent locking and how it differs from strict two phase locking.



Chapter 9

Database Recovery Techniques

9.1 INTRODUCTION

Database recovery is the process of restoring the database to a correct state in the event of a failure. In other words, it is the process of restoring the database to the most recent consistent state that existed shortly before the time of system failure. The failure may be the result of human error, hardware failure, incorrect or invalid data, program errors, computer viruses, or natural catastrophes. Recovery restores a database from a given state, usually inconsistent, to a previously consistent state.

A transaction is considered as a single unit of work in which all operations must be applied and completed to produce a consistent database. If, for some reason, any transaction operation cannot be completed, the transaction must be aborted and any change to the database must be rolled back. Thus, transaction recovery reverses all the changes that the transaction has made to the database before it was aborted. Since the organization depends on its database, the database management system must provide mechanisms for restoring a database quickly and accurately after loss or damage.

9.2 DIFFERENT TYPES OF DATABASE FAILURES

A failure in database can occur due to various reasons like wrong input data value or loss of data or destruction of entire database. Some common errors are:

- (i) System crashes, resulting in loss of main memory
- (ii) Media failures, resulting in loss of parts of secondary storage
- (iii) Application software errors
- (iv) Natural physical disasters
- (v) Carelessness or unintentional destruction of data or facilities
- (vi) Sabotage

9.2.1 Recovery Facilities

DBMS should provide facility to recover from failures in order to make the system stay in a consistent stage and for the efficient usage of the database. DBMS provides the following facilities to assist with recovery:

- (i) Backup mechanism, which makes periodic backup copies of database
- (ii) Logging facilities, which keep track of current state of transactions and Database changes
- (iii) Checkpoint facility, which enables updates to database in progress to be made permanent
- (iv) Recovery manager, which allows DBMS to restore the database to a consistent State following a failure

Backup Mechanism

The DBMS should provide backup facilities that produce a backup copy of the entire database. A backup copy is produced at least once per day. The copy should be stored in a secured location such that it is protected from loss or damage. The backup copy is used to restore the database in the case of hardware failure, catastrophic loss, or damage. In case of large databases, regular backups may be impractical, as the time required to perform the backup may exceed that available. As a result, backups can be taken for dynamic data regularly and backups of static data, which do not change frequently, may be taken less often.

Logging Facilities

Basically there are two types of log:

- (i) Transaction log
- (ii) Database change log

A transaction log is a record of the essential data for each transaction that is processed against the database. In database change log, there are before and after images of records that have been modified.

Transaction log: Transaction log contains a record of the essential data for each transaction that is processed against the database. Data that are Typically recorded for each transaction include the transaction code or identification, Action or type of transaction, time of the transaction, user ID, input data values, table and records accessed, records modified, and the old and new field values.

Database change log: The database change log contains before and after Images of records that have been modified by transactions. A before-image is a copy of a record before it has been modified, and an after-image is a copy of the same record after it has been modified.

Checkpoint Facility

A checkpoint facility in a DBMS periodically refuses to accept any new Transactions. All transactions in progress are completed, and the journal files are brought up to date. At this point, the system is in a quiet state, and the database and transaction logs are synchronized. The DBMS writes a special record to the log file, which is like a snapshot of the state of the database and that record is called the checkpoint record.

The checkpoint record contains information necessary to restart the system. Any dirty data blocks are written from memory to disk storage, thus ensuring that all changes made prior to

taking the checkpoint have been written to long-term storage. A DBMS may perform Checkpoints automatically or in response to commands in user application programs. Checkpoints should be taken frequently.

Recovery Manager

The recovery manager is a module of the DBMS which restores the database to a correct condition when a failure occurs and which resumes processing user requests. The recovery manager uses the logs to restore the database.

9.2.2 Main Recovery Techniques

The recovery techniques provided by the database management system are:

- (i) Deferred update
- (ii) Immediate update
- (iii) Shadow paging

Deferred update: Deferred updates are not written to the database only after the transaction commits its change. If transaction fails before commit, it will not modify the database and so no undo of changes are required. Deferred update may be necessary to redo updates of committed transactions as their effect may not have reached database.

Immediate update: In the case of immediate update, updates are applied to database as they occur. There is a need to redo updates of committed transactions following a failure. There may be need to undo effects of transactions that had not committed at time of failure. It is essential that log records are written before write to database. Undo operations are performed in reverse order in which they were written to log.

Shadow paging: Shadow paging maintains two page tables during life of a transaction, current page and shadow page table. When transaction starts, two pages are the same. Shadow page table is never changed thereafter and is used to restore database in the event of failure. During transaction, current page table records all updates to database. When transaction completes, current page table becomes shadow page table.

9.3 TRANSACTION RECOVERY

Transaction recovery overcomes the short comings of traditional recoveries by eliminating downtime and avoiding the loss of consistent data. Transaction recovery is the process of removing the undesired effects of specific transactions from the database. Traditional recovery is at object level. For example at the data space, table space or index level. When performing a traditional recovery, a specific database object is chosen. Then a backup copy of that objects is applied, followed by reapplying log entries for changes that occurred after the image copy was taken. As all changes made to a relational database are captured in the database log, the change details can be read from the log, recovery can be achieved by reversing the impact of the logged changes.

A transaction begins by executing a BEGIN TRANSACTION operation and ends by executing either a COMMIT or a ROLLBACK operation. COMMIT establishes a commit point or synch point. A commit point corresponds to the successful end of a transaction and the database will be

in a correct state. ROLLBACK rolls the database back to the previous commit point. There will be several transactions executing in parallel in a database.

When a commit point is established:

1. When a program is committed, the change is made permanent. i.e., they are guaranteed to be recorded in the database. Prior to the commit point updates are tentative i.e., they can be subsequently be undone.
2. All database positioning is lost and all tuple locks are released. Database positioning means at the time of execution each program will typically have addressability to certain tuples in the database, this addressability is lost at a COMMIT point.

Transactions are not only a unit of work but also a unit of recovery. If a transaction successfully commits, then the system updates will be permanently recorded in the database, even if the system crashes the very next moment. If the system crashes before the updates are written physically to the database, the system's restart procedure will still record those updates in the database. The values can be discovered from the relevant records in the log. The log must be physically written before the COMMIT processing can complete. This is called write-ahead log rule. The restart procedure helps in recovering any transactions that completed successfully but not physically written prior to the crash.

Implementation issues

1. Database updates are kept in buffers in main memory and not physically written to disk until the transaction commits. That way, if the transaction terminates unsuccessfully, there will be no need to undo any disk updates.
2. Database updates are physically written to the disk after COMMIT operation. That way, if the system subsequently crashes, there will be no need to redo any disk updates.

If there is no enough disk space then a transaction may steal buffer space from another transaction. They may also force updates to be written physically at the time of COMMIT.

Write ahead log rule is elaborated as follows:

1. The log record for a given database update must be physically written to the log before that update is physically written to the database.
2. All other log records for a given transaction must be physically written to the log before the COMMIT log record for that transaction is physically written to the log.
3. COMMIT processing for a given transaction must not complete until the COMMIT log record for that transaction is physically written to the log.

9.4 SYSTEM RECOVERY

The system must be recovered not only from local failures such as an individual transaction, but also from “global” failures. A local failure affects only the transaction in which the failure has actually occurred. A global failure affects all of the transactions in progress at the time of the failure. The failures fall into two broad categories:

System failures (e.g., power outage), which affect all transactions currently in progress but do not physically damage the database. A system failure is sometimes called a soft crash.

Media failures (e.g., head crash on the disk), which cause damage to the database or some portion of it. A media failure is sometimes called a hard crash.

9.4.1 System Failure and Recovery

During system failures the contents of main memory is lost. The transaction at the time of the failure will not be successfully completed, so transactions must be undone i.e., rolled back when the system restarts. It is necessary to redo certain transactions at the time of restart that is not successfully completed prior to the crash but did not manage to get their updates transferred from the buffers in main memory to the physical database.

Whenever some prescribed number of records has been written to the log the system automatically takes a checkpoint. The checkpoint record contains a list of all transactions that were in progress at the time the checkpoint was taken.

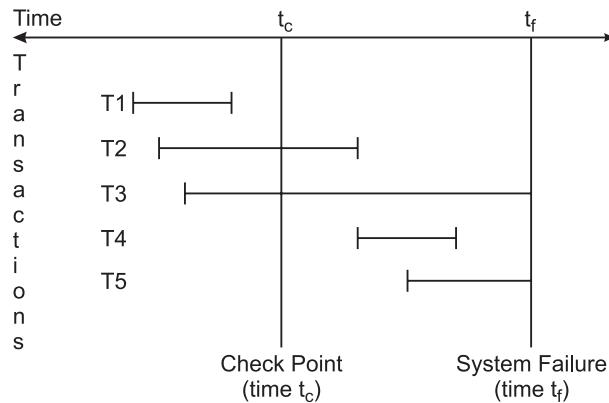


Fig. 9.1. Check Point Working Procedure.

Working of check point

- A system failure has occurred at time t_f
- The most recent checkpoint prior to time t_f was taken at time t_c .
- Transactions of type T_1 completed (successfully) prior to time t_c .
- Transactions of type T_2 started prior to time t_c and completed (successfully) after time t_c and before time t_f .
- Transactions of type T_3 also started prior to time t_c but did not complete by time t_f .
- Transactions of type T_4 started after time t_c and completed (successfully) before time t_f .
- Finally, transactions of type T_5 also started after time t_c but did not complete by time t_f .

The transactions of types T_3 and T_5 must be undone, and transactions of types T_2 and T_4 must be redone. At restart time, the system first goes through the following procedure:

1. Start with two lists of transactions, the UNDO list and the REDO list.
2. Set the UNDO list equal to the list of all transactions given in the most recent checkpoint record and the REDO list to empty.
3. Search forward through the log, starting from the checkpoint record.

4. If a BEGIN TRANSACTION log record is found for transaction T , add T to the UNDO list.
5. If a COMMIT log record is found for transaction T , move T from the UNDO list to the REDO list.
6. When the end of the log is reached, the UNDO and REDO lists are identified.

The system now works backward through the log, undoing the transactions in the UNDO list. Then it works forward, redoing the transactions in the REDO list. Restoring the database to a correct state by redoing work is sometimes called forward recovery. Restoring the database to a correct state by undoing work is called backward recovery. When the recovery activity is complete, the system is ready to accept new work.

9.5 MEDIA RECOVERY

Media recovery is different from transaction and system recovery. A media failure is a failure such as a disk head crash or a disk controller failure in which some portion of the database has been physically destroyed. Recovery from such a failure basically involves reloading or restoring the database from a backup or dump copy and then using the log. There is no need to undo transactions that were still in progress at the time of the failure. The dump portion of that utility is used to make backup copies of the database on demand. Such copies can be kept on tape or other archival storage, it is not necessary that they be on direct access media. After a media failure, the restore portion of the utility is used to recreate the database from a specified backup copy.

9.6 CRASH RECOVERY

Crash recovery is the process of protecting the database from catastrophic system failures and media failures. Recovery manager of a DBMS is responsible for ensuring transaction atomicity and durability. Atomicity is attained by undoing the actions of transactions that do not commit. Durability is attained by making sure that all actions of committed transactions survive system crashes.

9.7 RECOVERY MANAGER

Recovery manager is responsible for ensuring transaction atomicity and durability. To save the state of database for the period of times it performs few operations. They are:

1. Saving checkpoints
2. Stealing frames
3. Forcing pages

Saving checkpoints: It will save the status of the database in the period of time duration. So if any crashes occur, then database can be restored into last saved check point.

Steal approach: In this case, the object can be written into disk before the transaction which holds the object is committed. This happens when the buffer manager chooses the same place to

replace by some other page, and at the same time another transaction requires the same page. This method is called as stealing frames.

Forcing pages: In this case, once if the transaction completed the entire objects associated with it should be forced to disk or written to the disk. But it will result in more I/O cost. So normally we use only no-force approach.

9.8 ARIES ALGORITHM

ARIES is a recovery algorithm designed to work with a steal, no-force approach. It is more simple and flexible than other algorithms. ARIES algorithm is used by the recovery manager which is invoked after a crash. Recovery manager will perform restart operations. Earlier recovery system performs UNDO before REDO operations. ARIES scheme performs REDO before UNDO operation.

9.8.1 Terminologies Used in Aries

Log: These are all files which contain several records. These records contain the information about the state of database at any time. These records are written by the DBMS while any changes are done in the database. Normally copy of the log file is placed in the different parts of the disk for safety.

LSN: The full form of LSN is log sequence number. It is the ID given to each record in the log file. It will be in ascending order.

Page LSN: For recovery purpose, every page in the database contains the LSN of the most recent log record that describes a change to this page. This LSN is called the page LSN.

CLR: Compensation log record is abbreviated as CLR. It is written just before the change recorded in an update log record is undone.

WAL: The full form of WAL is write-ahead log. Before updating a page to disk, every update log record that describes a change to this page must be forced to stable storage.

There are three phases in restarting process, they are:

1. Analysis
2. Redo
3. Undo

Analysis: In this phase, it will identify whether any page present in the buffer pool is not written into disk and activate the transactions which are inactive at the time of crash. Build the REDO and UNDO lists.

Redo: In this phase, all the operations are restarted and the state of database at the time of crash is obtained. This is done with the help of log files.

Undo: In this phase, the actions of uncommitted transactions are undone. So only committed transactions are taken into account.

Example: Consider the following log history as shown in Fig. 9.2.

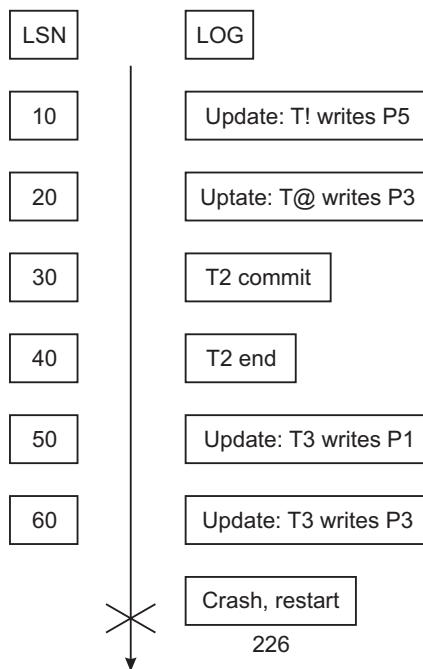


Fig. 9.2. Sample flow of a transaction.

When the system is restarted after the crash, the analysis phase identifies T1 and T3 as transactions active at the time of crash and therefore to be undone; T2 as a committed transaction, and all its actions therefore to be written to disk, and P1, P3, and P5 as potentially dirty pages. All the updates are reapplied in the order shown during the redo phase.

Finally the actions of T1 and T3 are undone in reverse order during the undo phase; that is, T3's write of P3 is undone, T3's write of P1 is undone, and then T1's write of P5 is undone.

ARIES algorithm has three main principles:

1. Write-ahead logging
2. Repeating history during redo
3. Logging changes during undo

Write-ahead logging: This states that any change to a database object is first recorded in the log; the record in the log must be written to stable storage before the change to the database object is written to disk.

Repeating history during redo: On restart after a crash, ARIES retraces all actions of the DBMS before the crash and brings the system back to the exact state that it was in at the time of crash. Then, it undoes the actions of transactions still active at the time of crash.

Logging changes during undo: Changes made to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated restarts.

9.8.2 Elements of Aries

The elements of ARIES includes the following:

1. The log
2. Tables
3. The write-ahead log protocol
4. Check pointing

1. The Log

It records a history of actions that are executed by DBMS. The most recent portion of the log is called as log tail. This page will be kept at main memory and periodically it will be forced to disk. It contains several records. Each record is uniquely identified by LSN.

A log record is written for each of the following actions:

- (i) Updating a page
- (ii) Commit
- (iii) Abort
- (iv) End
- (v) Undoing an update

Updating a page: After modifying the page, an update type record is appended to the log tail. The page LSN of this page is then set to the update log record as illustrated in the Fig. 9.3. Before-image is the value of the changed bytes before the change. After-image is the value of the changed bytes after the change.

Previous LSN	Transaction ID	Type	Page ID	Length	Offset	Before-Image	After-Image
--------------	----------------	------	---------	--------	--------	--------------	-------------

Fig. 9.3. Format of update type record.

Previous LSN	Transaction ID	Type	Page ID	Length	Offset	Before-Image	After-Image
	T1	Update	P1	4	15	HIL	SIL
	T2	Update	P2	4	23	BIL	WIL
	T2	Update	P1	4	14	TIL	VIL
	T1	Update	P4	4	15	RIL	JIL

Fig. 9.4. Log record.

2. Tables

In addition of log ARIES, it maintains the following two tables to sustain recovery related information:

- (i) Transaction table
- (ii) Dirty page table

Transaction table: This table contains one entry for each active transaction. The entry contains the transaction ID, the status and a field called last LSN, which is the LSN of the most recent log record for this transaction. The status of a transaction can be that it is in progress, committed, or aborted.

Transaction ID	Last LSN
T1	
T2	

Fig. 9.5. Transaction table.

Dirty page table: This table contains one entry for each dirty page in the buffer pool, i.e., each page with changes not yet reflected on disk. The entry contains a field record LSN, which is the LSN of the first log record that caused the page to become dirty.

Page ID	Record LSN
P1	
P2	
P4	

Fig. 9.6. Dirty page table.

Record LSN in the dirty page table and last LSN in the transaction table are pointing to the corresponding records in the log table.

3. Write-Ahead Log Protocol

WAL is the fundamental rule that ensures that a record of every change to the database is available while attempting to recover from crash. If a transaction made a change and committed, the no-force approach means that some of these changes may not have been written to disk at the time of a subsequent crash. Without a record of these changes, there would be no way to ensure that the changes of a committed transaction survive crashes. According to its rules when a transaction is completed its log tail is forced to disk, even a no-force approach is used.

4. Check Pointing

A checkpoint is used to reduce amount of work to be done during restart in the event of a subsequent crash. Check pointing in ARIES has three steps:

- (i) Begin checkpoint
- (ii) End checkpoint
- (iii) Fuzzy checkpoint

Begin checkpoint: It is written to indicate the start of a checkpoint.

End checkpoint: It is written at the end of a checkpoint. It contains current contents of transaction table and the dirty page table, and appended to the log.

Fuzzy checkpoint: It is written after end checkpoint is forced to the disk. While the end checkpoint is being constructed, the DBMS continues executing transactions and writing other log

records, the only guarantee we have is that the transaction table and dirty page table are accurate as the time of the begin checkpoint.

9.9 TWO PHASE COMMIT

When a transaction involves multiple distributed resources, for example, a database server on each of two different network hosts, the commit process is somewhat complex because the transaction includes operations that span two distinct software systems, each with its own resource manager, log records, and so no. In this case, the distributed resources are the database servers.

Two-phase commit is important whenever a given transaction can interact with several independent “resource managers”. With a two-phase commit protocol, the distributed transaction manager employs a coordinator to manage the individual resource managers.

Example:

- Consider a transaction running on an IBM mainframe that updates both an IMS database and a DB2 database. If the transaction completes successfully, then both IMS data and DB2 data are committed.
- Conversely, if the transaction fails, then both the updates must be rolled back.
- It is not possible to commit one database update and rollback the other. If done so the atomicity will not be maintained in the system.
- Therefore, the transaction issues a single “global” or system-wide COMMIT or ROLLBACK.
- That COMMIT or ROLLBACK is handled by a system component called the coordinator.
- Coordinators task is to guarantee the resource managers commit or roll back.
- It should also guarantee even if the system fails in the middle of the process.
- The two-phase commit protocol is responsible for maintaining such a guarantee.

9.9.1 Working

Assume that the transaction has completed and a COMMIT is issued. On receiving the COMMIT request, the coordinator goes through the following two-phase process:

Phase I - Prepare:

- The resource manager should get ready to “go either way” on the transaction.
- The participants in the transaction should record all updates performed during the transaction from temporary storage to permanent storage.
- In order to perform either COMMIT or ROLLBACK as necessary.
- Resource manager now replies “OK” to the coordinator or “NOT OK” based on the write operation.

Phase II - Commit:

- When the coordinator has received replies from all participants, it takes a decision regarding the transaction and records it in the physical log.
- If all replies were “OK,” then the decision is “commit”; if any reply was “Not OK,” the decision is “rollback.”
- The coordinator informs its decision to all the participants.

- Each participant must then commit or roll back the transaction locally, as instructed by the coordinator.

If the system fails at some point during the process, the restart procedure looks for the decision of the coordinator. If the decision is found then the two phase commit can start processing from where it has left off. If the decision is not found then it assumes that the decision is ROLLBACK and the process can complete appropriately. If the participants are from several systems like in distributed system, then some participants should wait for long time for the coordinator's decision. Data communication manager (DC manager) can act as a resource manager in case of a two-phase commit process.

9.10 SAVEPOINTS

Transactions cannot be nested with in another transaction. Transactions cannot be broken down into smaller sub transactions. Transactions establish intermediate savepoints while executing. If there is a roll back operation executed in the transaction, instead of performing roll back all the way to the beginning we can roll back to the previous savepoint. Savepoint is not the same as performing a COMMIT, updates made by the transaction are still not visible to other transactions until the transaction successfully executes a COMMIT.

Multiple savepoints can exist within a single transaction. Savepoints are useful for implementing complex error recovery in database applications. If an error occurs in the midst of a multiple statement transaction, the application may be able to recover from the error by rolling back to a savepoints without nesting to abort the entire transactions.

The syntax for creating savepoints is

SQL> SAVEPOINT <SAVEPOINTNAME>

Example: Suppose that we want to delete the employee records whose age is above 60, and we are not sure whether we are given work to actually delete the records of employees whose age is above 60 years or 58 years. In such case of dilemma, to proceed further we should create savepoints and then use delete command to delete the employee records.

SQL> savepoint Delete Employee;

SQL> delete from employee where Emp_age>60;

The above command deletes the records of those employees whose age is above 60 years. Though the changes are reflected on database they are not actually saved in the database, rather they are stored in temporary area.

After some time, if manager orders not to delete any employee's record, then we can undo the earlier changes by using the roll back to save point command as

SQL> rollback to Delete Employee;

It will rollback the changes made to the employee table.

9.11 SQL FACILITIES FOR RECOVERY

SQL supports transactions and transaction-based recovery. All executable SQL statements are atomic except CALL and RETURN. SQL provides BEGIN TRANSACTION, COMMIT, and ROLLBACK, called START TRANSACTION, COMMIT WORK, and ROLLBACK WORK, respectively.

Syntax for START TRANSACTION:

START TRANSACTION <option commalist>;

The *<option commalist>* specifies an access mode, an isolation level, or both. The **access mode** is either READ ONLY or READ WRITE. If neither is specified, READ WRITE is assumed. If READ WRITE is specified, the isolation level must not be READ UNCOMMITTED.

The isolation level takes the form

ISOLATION LEVEL <isolation>

where *<isolation>* can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

COMMIT

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command was issued.

The syntax for COMMIT:

COMMIT [WORK] [AND [NO] CHAIN];

AND CHAIN causes a START TRANSACTION to be executed automatically after the COMMIT; AND NO CHAIN is the default.

ROLLBACK

The ROLLBACK command is the transactional control command used to undo transactions that have not already been saved to the database. The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for COMMIT:

ROLLBACK [WORK] [AND [NO] CHAIN];

A CLOSE is executed automatically for every open cursor except for the cursors declared WITH HOLD. A cursor declared WITH HOLD is not automatically closed at COMMIT

SAVEPOINTS

SQL also supports savepoints.

The syntax for savepoints

SAVEPOINT <savepoint name>;

This syntax creates a savepoint with the specified user-chosen name. For more details on save points refer section 9.10.

Syntax for roll back:

ROLLBACK TO <savepoint name>;

This statement undoes all updates done since the specified savepoint.

Syntax for releasing savepoints:

RELEASE <savepoint name>;

This statement drops the specified savepoint. All savepoints are automatically dropped at transaction termination.

REVIEW QUESTIONS

1. List out the different types of database failures and the recovery facilities provided by the DBMS.
2. Write short notes on:
 - (i) Transaction failure
 - (ii) System failure
 - (iii) Media failure
 - (iv) Crash failure
3. What is the role of recovery manager in database recovery?
4. Explain in detail ARIES algorithm.
5. Write down the working procedure for two-phase commit protocol and explain the need for two-phase commit protocol.
6. Explain the importance of save points.



Chapter 10

Record Storage

10.1 INTRODUCTION

As all of us know database is a collection of data, this computerized data must be stored physically on a computerized storage media. DBMS is a collection of interrelated data and a set of program to access those data. The set of programs in the DBMS can be used to retrieve, update, and process this data as and when needed by the user. Computer storage media form a storage hierarchy that includes three main categories:

Primary storage: The memory storage, which is directly accessible by the computer's central processing unit (CPU), comes under this category. CPU's internal memory (registers), fast memory (cache) and main memory (RAM) are directly accessible to CPU as they all are placed on the motherboard. Primary storage usually provides fast access to data but is of limited storage capacity. They are expensive and have less storage capacity than secondary and tertiary storage devices. This storage needs continuous power supply in order to maintain its state, i.e. in case of power failure all data are lost.

Secondary storage: Primary memory cannot store large amount of data but in general there is a need to store large data for longer amount of time and to preserve it even after the power supply is interrupted, this requirement gave to the development of secondary data storage. All memory devices, which are not part of CPU chipset or motherboard comes under this category. It includes magnetic disks, optical disks (CD-ROMs, DVDs, and other similar storage media), and tapes. Hard disk drives, which contain the operating system and generally not removed from the computers, are considered secondary storage and all others are called tertiary storage.

Tertiary storage: Third level in memory hierarchy is called tertiary storage. This is used to store huge amount of data. This storage is external to the computer system. Tertiary memories are slower in speed. These storage devices are mostly used to back up the entire system. Optical disk and magnetic tapes are widely used storage devices as tertiary storage.

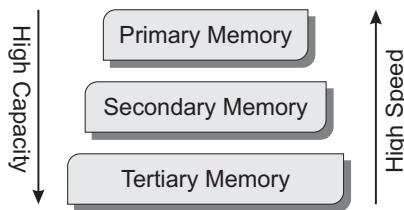


Fig. 10.1. Computer memory hierarchy.

As discussed above, the data in database management system (DBMS) is stored on physical storage devices such as main memory, secondary memory or tertiary storage. The overall performance of a database system is determined by the physical database organization. Thus, it is important that the physical database is properly designed to increase data processing efficiency and minimize the time required by users to interact with the information system.

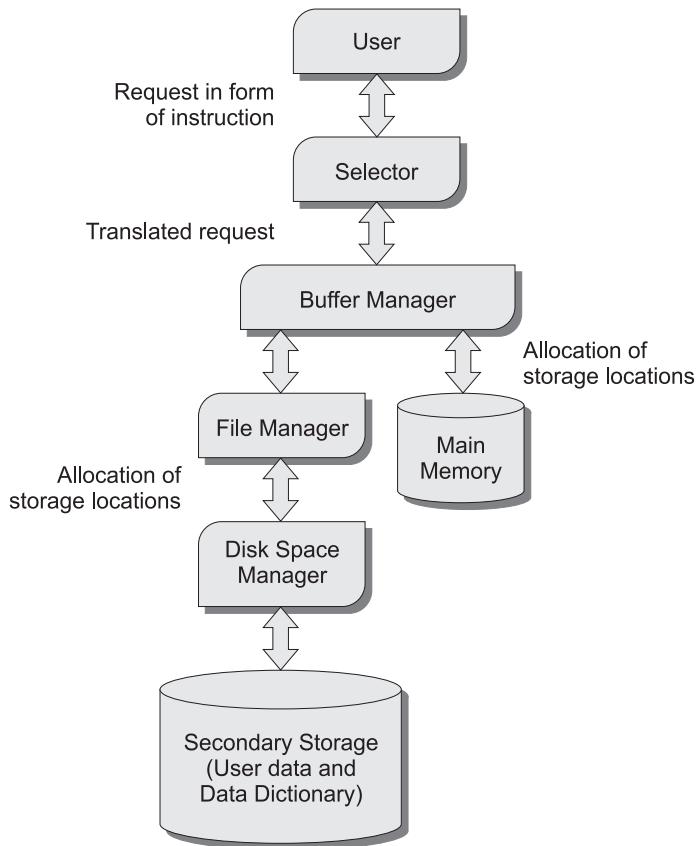


Fig. 10.2. Physical access of the database.

10.2 PHYSICAL STORAGE MEDIA OVERVIEW

Computer system provides a variety of data storage. These storage media are classified based on few categories like speed of data access, cost per unit of data to buy the medium and reliability of the storage. Few of the storage media are described below:

Cache: Cache memory is extremely fast memory that is built into a computer's central processing unit (CPU), or located next to it on a separate chip. The CPU uses cache memory to store instructions that are repeatedly required to run programs, improving overall system speed. The advantage of cache memory is that the CPU does not have to use the motherboard's system bus for data transfer. Whenever data must be passed through the system bus, the data transfer speed slows to the motherboard's capability. The size of the cache memory is very small.

Main memory: Main memory is where programs and data are kept when the processor is actively using them. When programs and data become active, they are copied from secondary memory into main memory where the processor can interact with them. A copy remains in secondary memory. Main memory is intimately connected to the processor, so moving instructions and data into and out of the processor is very fast. The general-purpose machine instructions operate on main memory. Main memory is too small and expensive for storing the entire database contents. The contents of main memory are usually lost if a power failure or system crash occurs.

Flash memory: Flash memory is also known as electrically erasable programmable read-only memory (EEPROM). Flash memory differs from main memory where data survive power failure. Reading data from flash memory takes less than 100 nanoseconds which are as fast as reading data from main memory. However, writing data to flash memory is more complicated: data can be written once, which takes about 4 to 10 microseconds, but cannot be overwritten directly. To overwrite memory that has been written already, we have to erase an entire bank of memory at once. It is then ready to be written again. A drawback of flash memory is that it can support only a limited number of erase cycles, ranging from 10,000 to 1 million. Flash memory has found popularity as a replacement for magnetic disks for storing small volumes of data in low-cost computer systems, such as computer systems that are embedded in other devices, in hand-held computers, and in other digital electronic devices such as digital cameras.

Magnetic-disk storage: The primary medium for the long term on line storage of data is the magnetic disk. The entire database is stored on magnetic disk. To access the data it has to be moved from disk to main memory. After all the operations, the data that have been modified must be written to disk. Disk storage survives power failures and system crashes. Disk-storage devices themselves fail sometimes and thus destroy data. Such failures usually occur less frequently than the system crashes.

Optical storage: The most popular forms of optical storage are the compact disk (CD). As the name implies data are stored optically on a disk, and are read by a laser. The optical disks used in read-only compact disks (CD-ROM) or read-only digital video disk (DVD-ROM) cannot be written, they are supplied with data prerecorded.

Record once versions of compact disk (CD-R) and digital video disk (DVD-R) can be written only once. These disks are also called write-once, read-many (WORM) disks.

Multiple-write versions of compact disk (CD-RW) and digital video disk (DVD-RW and DVD-RAM) can be written multiple times.

Tape storage: Tape storage is used primarily for backup and archival data. Although magnetic tape is much cheaper than disks, access to data is much slower, because the tape must be accessed sequentially from the beginning. For this reason, tape storage is referred to as sequential access storage. In contrast, disk storage is referred to as direct access storage because it is possible to read data from any location on disk.

Tapes have a high capacity, and can be removed from the tape drive, facilitating cheap archival storage. Tape jukeboxes are used to hold exceptionally large collections of data, such as remote sensing data from satellites, which include large amount of data.

10.2.1 Storage Device Hierarchy

The various storage media can be organized in a hierarchy according to their speed and their cost shown in figure 10.3. The higher levels are expensive, but are fast. The cost per bit decreases as we move down and access time increases.

The fastest storage media like cache and main memory are referred to as primary storage. The media in the next level of the hierarchy, the magnetic disks, are referred to as secondary storage, or online storage. The media in the lowest level of the hierarchy, including magnetic tape and optical-disk jukeboxes, are referred to as tertiary storage, or offline storage.

In addition to the speed and cost of the various storage systems, there is also the issue of storage volatility. Volatile storage loses its contents when the power to the device is removed. In the hierarchy, the storage systems from main memory up are volatile, whereas the storage systems below main memory are nonvolatile.

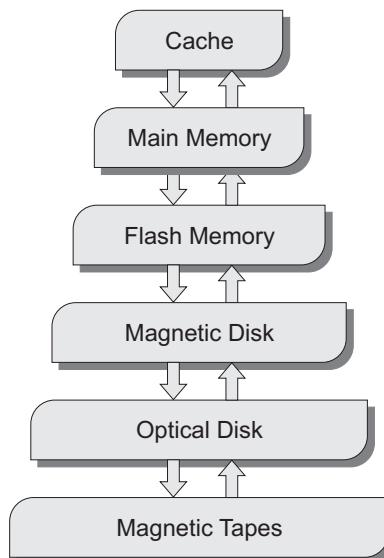


Fig. 10.3. Physical storage hierarchy.

10.3 MAGNETIC DISK

Most of the modern computers rely on the magnetic disks for their huge secondary storage. Disk capacities are growing day by day. A large database may require hundreds of disks.

10.3.1 Physical Characteristics

Magnetic disks are used for storing large amounts of data. The most basic unit of data on the disk is a single bit of information. To code information, bits are grouped into bytes. Each disk platter has a flat circular shape. Its two surfaces are covered with a magnetic material and information is recorded on the surfaces. Platters are made from rigid metal or glass and are covered with magnetic recording material on both sides.

The disk surface is logically divided into tracks, which are subdivided into sectors. A sector is the smallest unit of information that can be read from or written to the disk. There are over 16,000 tracks on each platter, and 2 to 4 platters per disk. The inner tracks are of smaller length and the outer tracks contain more sectors than the inner tracks.

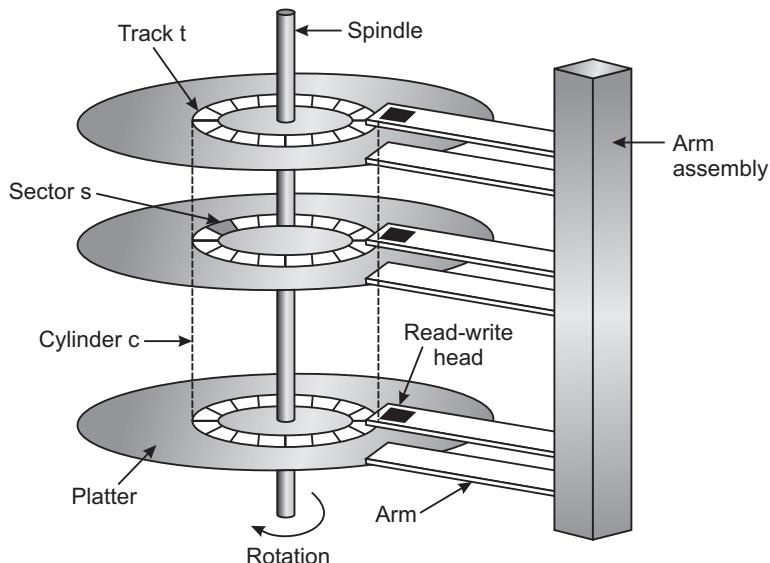


Fig. 10.4. Magnetic Disc Assembly.

When the disk is in use, a drive motor spins it at a constant high speed. There is a read–write head positioned just above the surface of the platter. This read–write head stores information on a sector. Each side of a platter of a disk has a read–write head, which moves across the platter to access different tracks. The platters and the read–write heads of all the tracks are mounted on a single assembly called a disk arm.

The disk platters mounted on a spindle and the heads mounted on a disk arm are together known as head–disk assemblies. Since the heads on all the platters move together, when the head on one platter is on the i th track, the heads on all other platters are also on the i th track of their respective platters. Hence, the i th tracks of all the platters together are called the i th cylinder.

The read–write heads are kept as close as possible to the disk surface to increase the recording density. The head typically floats from the disk surface. If the head contacts the disk surface, the head can scrape the recording medium off the disk, destroying the data that had been there. Head crashes can be a problem. A head crash results in failure of the entire disk, which must then be replaced.

A fixed-head disk has a separate head for each track. This arrangement allows the computer to switch from track to track quickly, without having to move the head assembly, but because of the large number of heads, the device is extremely expensive.

10.3.2 Disk Controller

A disk controller interfaces between the computer system and the actual hardware of the disk drive. It accepts high-level commands to read or write a sector, and initiates actions, such as moving the disk arm to the right track and actually reading or writing the data. Disk controllers also attach checksums to each sector to check errors in the data read from the disk.

The checksum is computed from the data written to the sector. When the sector is read back, the controller computes the checksum again from the retrieved data and compares it with the stored checksum; if the data are corrupted, with a high probability the newly computed checksum will not match the stored checksum. If such an error occurs, the controller will retry the read several times; if the error continues to occur, the controller will signal a read failure.

Another functionality of disk controllers is to remap bad sectors. If the controller detects that a sector is damaged when the disk is initially formatted, or when an attempt is made to write the sector, it can logically map the sector to a different physical location.

Like other storage units, disks are connected to a computer system or to a controller through a high speed interconnection. Figure 10.5 shows how disks are connected to a computer system.

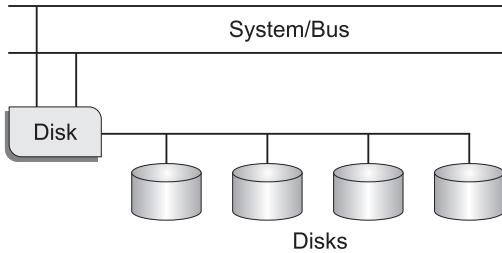


Fig. 10.5. Disk Subsystem.

In modern disk systems, functions of the disk controller like controlling the disk arm, computing and verification of checksums, and remapping of bad sectors, are implemented within the disk drive unit. In the storage area network (SAN) architecture, large numbers of disks are connected by a high-speed network to a number of server computers.

10.3.3 Disk Performance Measurement

The qualities that measure the performance of a disk are:

1. Capacity
2. Access time

3. Data-transfer rate
4. Reliability

Access time is the time from when a read or write request is issued to the time from when data transfer begins. To access data on a given sector of a disk, the arm first must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates. The time for repositioning the arm is called the **seek time**. Typical seek times range from 2 to 30 milliseconds, depending on how far the track is from the initial arm position.

The **average seek time** is the average of the seek times, measured over a sequence of random requests. Once the seek has started, the time spent waiting for the sector to be accessed to appear under the head is called the **rotational latency time**. Thus, the **average latency time** of the disk is one-half the time for a full rotation of the disk. The access time is then the sum of the seek time and the latency, and ranges from 8 to 20 milliseconds. Once the first sector of the data to be accessed has come under the head, data transfer begins.

The **data-transfer rate** is the rate at which data can be retrieved from or stored to the disk. Current disk systems claim to support maximum transfer rates of about 25 to 40 megabytes per second.

The final commonly used measure of a disk is the **mean time to failure (MTTF)**, which is a measure of the reliability of the disk. The mean time to failure of a disk is the amount of time that, on average, we can expect the system to run continuously without any failure. According to vendors' claims, the mean time to failure of disks today ranges from 30,000 to 1,200,000 hours—about 3.4 to 136 years.

10.3.4 Optimization of Disk-Block Access

Data are transferred between disk and main memory in units of blocks. A block is a contiguous sequence of sectors from a single track of one platter. Blocks are addressed using block number. Block sizes range from 512 bytes to several kilobytes. The lower levels of the file-system manager convert block addresses into the hardware-level cylinder, surface, and sector number. Few techniques to optimize disk block access are discussed below:

Scheduling: If several blocks from a cylinder need to be transferred from disk to main memory, we may be able to save access time by requesting the blocks in the order in which they will pass under the heads.

Disk-arm-scheduling algorithms attempt to order accesses to tracks in a fashion that increases the number of accesses that can be processed. A commonly used algorithm is the **elevator algorithm**, which works in the same way as many elevators do.

Suppose that, initially, the arm is moving from the innermost track toward the outside of the disk. Under the elevator algorithms control, for each track with an access request, the arm stops, services requests for the track, and then continues moving outward until there are no waiting requests for tracks farther out. At this point, the arm changes direction, and moves towards the inside, again stopping at each track for which there is a request, until it reaches a track where there is no request for tracks farther toward the center. Now, it reverses direction and starts a new cycle.

File organization. To reduce block-access time, we can organize blocks on disk in a way that corresponds closely to the way we expect data to be accessed. For example, if we expect a file to be

accessed sequentially, then we should ideally keep all the blocks of the file sequentially on adjacent cylinders. Over time, a sequential file may become **fragmented**; that is, its blocks become scattered all over the disk. To reduce fragmentation, the system can make a backup copy of the data on disk and restore the entire disk. The restore operation writes back the blocks of each file contiguously. Some systems have utilities that scan the disk and then move blocks to decrease the fragmentation. The performance increases realized from these techniques can be large.

Nonvolatile write buffers: Since the contents of main memory are lost in a power failure, information about database updates has to be recorded on disk to survive possible system crashes. For this reason, the performance of update-intensive database applications, such as transaction-processing systems, is heavily dependent on the speed of disk writes. We can use **nonvolatile random-access memory** (NV-RAM) to speed up disk writes drastically. The contents of nonvolatile RAM are not lost in power failure. A common way to implement nonvolatile RAM is to use battery-backed-up RAM. The idea is that, when the database system requests that a block be written to disk, the disk controller writes the block to a nonvolatile RAM buffer, and immediately notifies the operating system that the write completed successfully.

Log disk: Another approach to reduce write latencies is to use a log disk, a disk devoted to writing a sequential log. All access to the log disk is sequential and several consecutive blocks can be written at once, making writes to the log disk several times faster than random writes.

The data have to be written to their actual location on disk as well, but the log disk can do the write later, without the database system having to wait for the write to complete. Furthermore, the log disk can reorder the writes to minimize disk arm movement. If the system crashes before some writes to the actual disk location have completed, when the system comes back up it reads the log disk to find those writes that had not been completed, and carries them out then.

File systems that support log disks as above are called **journaling file systems**. Journaling file systems can be implemented even without a separate log disk, keeping data and the log on the same disk. Doing so reduces the monetary cost, at the expense of lower performance. The **log-based file system** is an extreme version of the log-disk approach. Data are not written back to their original destination on disk; instead, the file system keeps track of where in the log disk the blocks were written most recently, and retrieves them from that location. The log disk itself is compacted periodically, so that old writes that have subsequently been overwritten can be removed. This approach improves write performance, but generates a high degree of fragmentation for files that are updated often.

10.4 RAID

Disk are potential bottlenecks for system performance and storage system reliability. Even though disk performance has been improving continuously, microprocessor performance has advanced much more rapidly. Since disks contain mechanical elements, they have much higher failure rates than electronic parts of a computer system. If a disk fails, all the data stored on it will be lost.

A disk array is an arrangement of several disks, organized so as to increase performance and improve reliability of the resulting storage system. Performance is increased through data striping. Data striping distributes data over several disks to give the impression of having a single large, very fast disk. Reliability is improved through redundancy. Instead of having a single copy of

the data, redundant information is maintained. The redundant information is carefully organized so that in case of a disk failure, it can be used to reconstruct the contents of the failed disk. Disk arrays that implement a combination of data striping and redundancy are called redundant arrays of independent disks, or in short, RAID. Several RAID organizations, referred to as RAID levels, have been proposed. Each RAID level represents a different trade-off between reliability and performance.

10.4.1 Data Striping

A disk array gives the user the abstraction of having a single, very large disk. If the user issues an I/O request, we first identify the set of physical disk blocks that store the data requested. These disk blocks may reside on a single disk in the array or may be distributed over several disks in the array. Then the set of blocks is retrieved from the disks involved. Thus, how we distribute the data over the disks in the array influences how many disks are involved when an I/O request is processed.

In data striping, the data is segmented into equal-size partitions that are distributed over multiple disks. The size of a partition is called the striping unit. The partitions are usually distributed using a round robin algorithm. If the disk array consists of D disks, then partition i is written onto disk $i \bmod D$. The requests are processed in parallel by all disks leading to increase in the transfer rate.

There are two type of stripping:

1. Bit-level striping stripes bits of data across multiple disks
2. Block-level striping stripes blocks across multiple disks.

10.4.2 Redundancy

Increase in number of disk increases the storage system performance and reduces the storage system reliability. Assume that the mean-time-to-failure, or MTTF, of a single disk is 50,000 hours. Then, the MTTF of an array of 100 disks is only $50,000/100 = 500$ hours or about 21 days, assuming that failures occur independently and that the failure probability of a disk does not change over time.

Reliability of a disk array can be increased by storing redundant information. If a disk failure occurs, the redundant information is used to reconstruct the data on the failed disk. Redundancy can immensely increase the MTTF of a disk array. When incorporating redundancy into a disk array design, we have to make two choices.

First, we have to decide where to store the redundant information. We can either store the redundant information on a small number of check disks or we can distribute the redundant information uniformly over all disks.

The second choice we have to make is how to compute the redundant information. Most disk arrays store parity information. In the parity scheme, an extra check disk contains information that can be used to recover from failure of any one disk in the array. Assume that we have a disk array with D disks and consider the first bit on each data disk. Suppose that i of the D data bits are one. The first bit on the check disk is set to one if i is odd, otherwise it is set to zero. This bit on the check disk is called the parity of the data bits. The check disk contains parity information for each set of corresponding D data bits.

To recover the value of the first bit of a failed disk we first count the number of bits that are one on the $D - 1$ non failed disks; let this number be j . If j is odd and the parity bit is one, or if j is even and the parity bit is zero, then the value of the bit on the failed disk must have been zero. Otherwise, the value of the bit on the failed disk must have been one. Thus, with parity we can recover from failure of any one disk. Reconstruction of the lost information involves reading all data disks and the check disk.

10.4.3 Raid Levels

RAID 0: In this level a striped array of disks is implemented. The data is broken down into blocks and all blocks are distributed among all disks. Each disk receives a block of data to write/read in parallel. This enhances the speed and performance of storage device. There is no parity and backup in Level 0.

Advantages

- Very simple design and easy to implement.
- No parity calculation overhead is involved.
- Best performance is achieved when data are striped across multiple controllers with only one drive per controller.
- Input/output performance is greatly improved by spreading the input/output load across many channels and drives.

Disadvantages

- RAID Level 0 is not a “true” RAID because it is not fault-tolerant.
- The failure of just one drive will result in all data in an array being lost.
- RAID Level 0 should never be used in mission critical environments.

Applications

- Image, video editing
- Prepress applications
- Applications that require high bandwidth

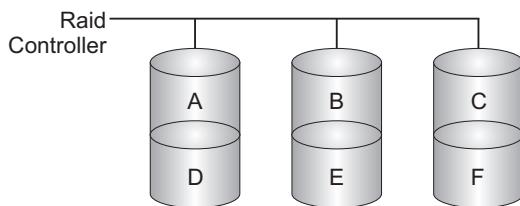


Fig. 10.6. RAID 0.

RAID 1: A RAID Level 1 system is the most expensive solution. Instead of having one copy of the data, two identical copies of the data on two different disks are maintained. This type of redundancy is often called mirroring. Every write of a disk block involves a write on both disks. These writes may not be performed simultaneously, since a global system failure (e.g., due to a power outage) could occur while writing the blocks and then leave both copies in an inconsistent

state. Therefore, we always write a block on one disk first and then write the other copy on the mirror disk. Since two copies of each block exist on different disks, we can distribute reads between the two disks and allow parallel reads of different disk blocks that conceptually reside on the same disk. RAID Level 1 does not stripe the data over different disks.

Advantages

- Simplest RAID storage subsystem design.
- Under certain circumstances, RAID 1 can sustain multiple simultaneous drive failures.
- One hundred percent redundancy of data means no rebuild is necessary in case of a disk failure, just a copy to the replacement disk.

Disadvantages

- Highest disk overhead.
- May not support hot swap of failed disk when implemented in “software.”
- Hardware implementation is strongly recommended.

Applications

- Accounting
- Payroll
- Financial
- Any application requiring high availability

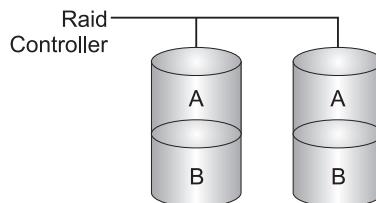


Fig. 10.7. RAID 1.

RAID 2: This level records the Error Correction Code using Hamming distance for its data striped on different disks. Like level 0, each data bit in a word is recorded on a separate disk and ECC codes of the data words are stored on different set disks. Because of its complex structure and high cost, RAID 2 is not commercially available.

Advantages

- Extremely high data transfer rates possible.
- Relatively simple controller design compared to RAID Levels 3–5.

Disadvantages

- Entry level cost is very high.
- No practical use; same performance can be achieved by RAID 3 at lower cost.

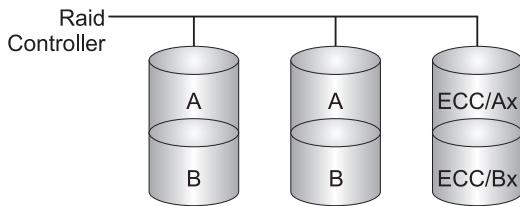


Fig. 10.8. RAID 2.

RAID 3 (Bit-Interleaved Parity): This level also stripes the data onto multiple disks in array. The parity bit generated for data word is stored on a different disk. This technique makes it to overcome single disk failure and a single disk failure does not impact the throughput.

Advantages

- Very high data transfer rate.
- Disk failure has an insignificant impact on throughput.
- High efficiency because of low ratio of parity disks to data disks.

Disadvantages

- Controller design is fairly complex.
- Transaction rate is equal to that of a single disk drive at best.
- Very difficult and resource intensive to implement a “software” RAID.

Applications

- Video production and live streaming
- Image editing, video editing
- Any application requiring high throughput

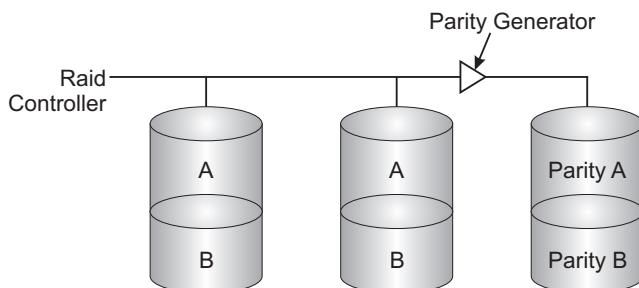


Fig. 10.9. RAID 3.

RAID 4 (Block-Interleaved Parity): In this level an entire block of data is written onto data disks and then the parity is generated and stored on a different disk. The prime difference between level 3 and 4 is, level 3 uses byte level striping whereas level 4 uses block level striping. Both level 3 and 4 requires at least 3 disks to implement RAID.

Advantages

- Very high Read data transaction rate.
- Low ratio of parity disks to data disks means high efficiency.

- High aggregate Read transfer rate.

Disadvantages

- Quite complex controller design.
- Worst write transaction rate and Write aggregate transfer rate.
- Difficult and inefficient data rebuild in the event of disk failure.

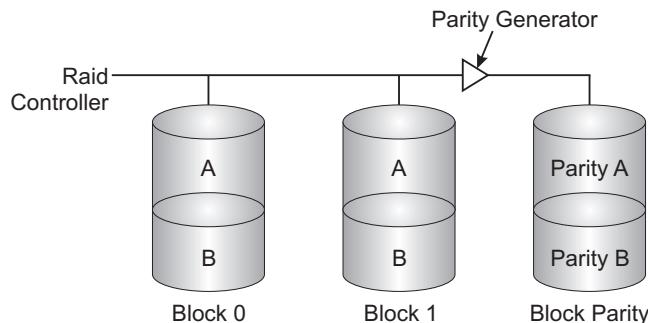


Fig. 10.10. RAID 4.

RAID 5 (Block-Interleaved Distributed Parity): This level also writes whole data blocks on to different disks but the parity generated for data block stripe is not stored on a different dedicated disk, but is distributed among all the data disks. This distribution has two advantages. First, several write requests can potentially be processed in parallel, since the bottleneck of a unique check disk has been eliminated. Second, read requests have a higher level of parallelism. Since the data is distributed over all disks, read requests involve all disks, whereas in systems with a dedicated check disk the check disk never participates in reads.

Advantages

- Highest Read transaction rate
- Medium Write data transaction rate
- Good aggregate transfer rate

Disadvantages

- Most complex controller design
- Difficult to rebuild in the event of disk failure
- Individual block transfer rate is same as single disk

Applications

- File and application servers
- Database servers
- Intranet servers

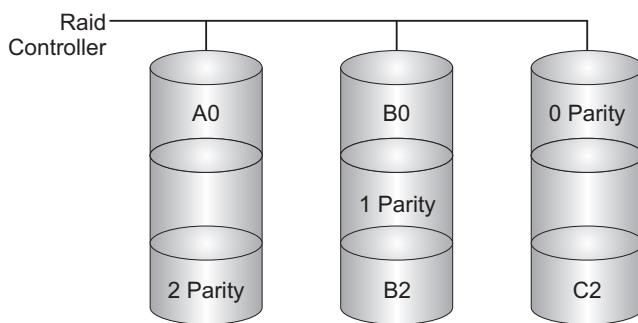


Fig. 10.11. RAID 5.

RAID 6 (P+Q Redundancy): This level is an extension of level 5. In this level two independent parities are generated and stored in distributed fashion among disks. Two parities provide additional fault tolerance. This level requires at least 4 disk drives to be implemented.

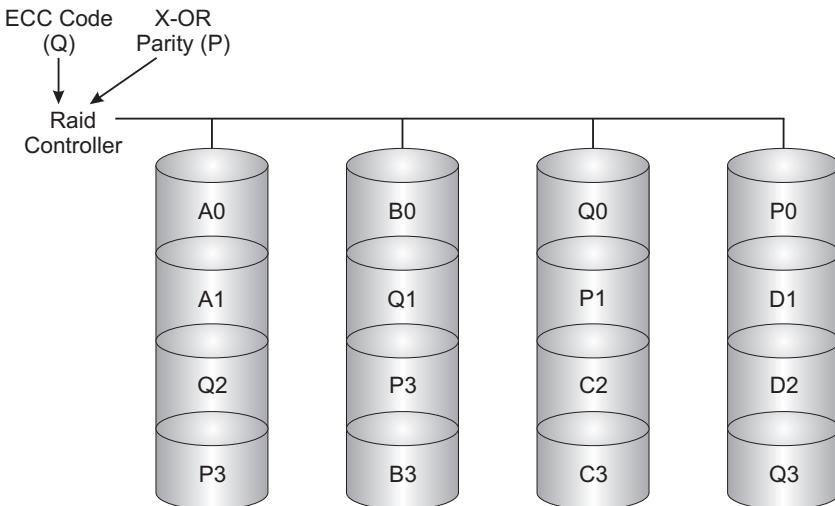


Fig. 10.12. RAID 6.

Advantages

- RAID Level 6 provides high fault tolerance and can sustain multiple simultaneous drive failures.
- Perfect solution for critical applications.

Disadvantages

- Some of the drawbacks of RAID Level 6 are:
- More complex controller design.
- Controller overhead to compute parity addresses is extremely high.

Applications

- File and application servers

- Web and E-mail servers
- Intranet servers

10.4.4 Choice of Raid Level

The factors to be taken into account when choosing a RAID level are:

- Monetary cost of extra disk storage requirements
- Performance requirements in terms of number of I/O operations
- Performance when a disk has failed
- Performance during rebuild

The time to rebuild the data of a failed disk varies with the RAID level that is used. Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data of a failed disk.

RAID level 0 is used in high-performance applications where data safety is not critical. RAID levels 2 and 4 are subsumed by RAID levels 3 and 5, the choice of RAID levels is restricted to the remaining levels. Bit striping (level 3) is rarely used since block striping (level 5) gives as good data transfer rates for large transfers, while using fewer disks for small transfers.

In fact, level 3 may perform worse than level 5 for a small transfer, since the transfer completes only when corresponding sectors on all disks have been fetched; the average latency for the disk array thus becomes very close to the worst-case latency for a single disk, negating the benefits of higher transfer rates. Level 6 is not supported currently by many RAID implementations, but it offers better reliability than level 5 and can be used in applications where data safety is very important.

The choice between RAID level 1 and level 5 is harder to make. RAID level 1 is popular for applications such as storage of log files in a database system, since it offers the best write performance. RAID level 5 has a lower storage overhead than level 1, but has a higher time overhead for writes. For applications where data are read frequently, and written rarely, level 5 is the preferred choice.

RAID level 5, which increases the number of I/O operations needed to write a single logical block, pays a significant time penalty in terms of write performance. RAID level 1 is therefore the RAID level of choice for many applications with moderate storage requirements, and high I/O requirements.

10.4.5 Hardware Issues

Another issue in the choice of RAID implementations is at the level of hardware. RAID can be implemented with no change at the hardware level, using only software modification. Such RAID implementations are called software RAID. Hardware RAID implementations can use nonvolatile RAM to record writes that need to be executed, in case of power failure before a write is completed, when the system comes back up, it retrieves information about incomplete writes from nonvolatile RAM and then completes the writes. Some hardware RAID implementations permit hot swapping; that is, faulty disks can be removed and replaced by new ones without turning power off. Hot swapping reduces the mean time to repair, since replacement of a disk does not have to wait until a time when the system can be shut down.

Further, many RAID implementations assign a spare disk for each array. If a disk fails, the spare disk is immediately used as a replacement. As a result, the mean time to repair is reduced greatly, minimizing the chance of any data loss. The failed disk can be replaced at leisure.

The power supply, or the disk controller, or even the system interconnection in a RAID system could become a single point of failure that could stop functioning of the RAID system. To avoid this possibility, good RAID implementations have multiple redundant power supplies. Such RAID systems have multiple disk controllers, and multiple interconnections to connect them to the computer system. Thus, failure of any single component will not stop the functioning of the RAID system.

10.5 TERTIARY STORAGE

Tertiary storage aims in providing huge storage capacity at low cost. Some of the commonly used tertiary storage includes:

1. Optical disks
2. Magnetic tapes

These storage devices are composed of fixed storage drives and removable media units. The storage drives are fixed to the computer system. The removable media unit can be removed from the drives so that the storage capacity can be expanded with more media units.

10.5.1 Optical Disks

Compact disks are a popular medium for distributing software, multimedia data such as audio and images, and other electronically published information. They have a fairly large capacity and they are cheap to mass-produce. Digital video disks (DVDs) are replacing compact disks in applications that require very large amounts of data.

CD and DVD drives have much longer seek times than magnetic-disk drives, since the head assembly is heavier. Rotational speeds are typically lower than those of magnetic disks, although the faster CD and DVD drives have rotation speeds of about 3000 rotations per minute, which is comparable to speeds of lower-end magnetic-disk drives.

Data transfer rates are somewhat less than for magnetic disks. Current CD drives read at around 3 to 6 megabytes per second, and current DVD drives read at 8 to 15 megabytes per second. Like magnetic disk drives, optical disks store more data in outside tracks and less data in inner tracks.

The transfer rate of optical drives is characterized as $n\times$, which means the drive supports transfers at n times the standard rate; rates of around 50 \times for CD and 12 \times for DVD are now common. The record-once versions of optical disks are popular for distribution of data and particularly for archival storage of data because they have a high capacity, have a longer lifetime than magnetic disks, and can be removed and stored at a remote location. Since they cannot be overwritten, they can be used to store information that should not be modified, such as audit trails. The multiple write versions (CD-RW, DVD-RW, and DVD-RAM) are also used for archival purposes.

Jukeboxes are devices that store large number of optical disks and load them automatically on demand to one of a small number of drives. The aggregate storage capacity of such a system can be many terabytes.

10.5.2 Magnetic Tapes

Magnetic tapes are relatively permanent, and can hold large volumes of data. They are slow compared to magnetic and optical disks. Magnetic tapes are limited to sequential access. Tapes are used mainly for backup, for storage of infrequently used information, and as an offline medium for transferring information from one system to another.

Tapes are also used for storing large volumes of data, such as video or image data that either do not need to be accessible quickly or are so voluminous that magnetic disk storage would be too expensive.

A tape is kept in a spool, and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take seconds or even minutes. Capacities vary, depending on the length and width of the tape and on the density at which the head can read and write.

Data transfer rates are of the order of a few to tens of megabytes per second. Tape devices are reliable, and good tape drive systems perform a read of the just-written data to ensure that it has been recorded correctly. Tapes, however, have limits on the number of times that they can be read or written reliably. Some tape formats support faster seeks times. This feature is important for applications that need quick access to very large amounts of data, larger than what would fit economically on a disk drive.

Most other tape formats provide larger capacities, at the cost of slower access; such formats are ideal for data backup, where fast seeks is not important. Tape jukeboxes, like optical disk jukeboxes, hold large numbers of tapes, with a few drives onto which the tapes can be mounted; they are used for storing large volumes of data, ranging up to many terabytes with access times on the order of seconds to a few minutes.

REVIEW QUESTIONS

1. Explain the need for storage system in a computer system.
2. Explain the hierarchy of storage structure.
3. Explain the various physical storage media available
4. Explain the physical properties of a magnetic disk.
5. Explain the role of a disk controller.
6. How will you measure disk performance?
7. How will you optimize the access to disk block?
8. Explain the concept of RAID.
9. How will you choose the appropriate RAID level?
10. Explain the need tertiary storage device.



Chapter 11

Physical Database Design

11.1 INTRODUCTION

Physical database design describes the storage structures and access methods used in system. The goal of physical database design is to specify all identifying and operational characteristics of the data that will be recorded in the information system. The physical database design specifies how database records are stored, accessed, and related to ensure adequate performance. The physical database design specifies the base relations, file organizations, and indexes used to achieve efficient access to the data. The physical organization of data has a major impact on database system performance because it is the level at which actual implementation takes place in physical storage.

The goal of physical database design is to create a design providing the best response time at the lowest cost. Response time refers to the time required to access the data, and the cost associated with CPU, memory disk input/output. The main goals of good physical database design are:

- A good physical database design should achieve high packing density, which implies minimum wastage space.
- A good physical database design should achieve fast response time.
- The physical database design should also support a high volume of transactions.

11.2 PHYSICAL DESIGN STEPS

The various steps in physical database design are:

1. Stored record format design
2. Stored record clustering
3. Access method design
4. Program design

Stored Record Format Design

The visible component of the physical database structure is the stored record format design. Stored record format design addresses the problem of formatting stored data by analysis of the characteristics of data item types, distribution of their values, and their usage by various applications. Decisions

on redundancy of data, derived and explicitly stored values of data, and data compression are made here. Certain data items are often accessed far more frequently than others. Each time when a particular piece of data is needed, the entire stored records are accessed. Record partitioning defines an allocation of individual data items to separate physical devices of the same or different type, or separate extents on the same device, so that total cost of accessing data for a given set of user applications is minimized.

Stored Record Clustering

One of the most important physical design considerations is the physical allocations of stored records, as a whole, to physical extents. Record clustering refers to the allocation of records of different types into physical clusters to take advantage of physical sequentiality whenever possible. Associated with both record clustering and record partitioning is the selection of physical block size. Blocks in a given clustered extent are influenced to some extent by stored record size, storage characteristics of the physical devices. Larger blocks are typically associated with sequential processing and smaller blocks with random processing.

Access Method Design

The critical components of an access method are storage structure and search mechanisms. Storage structure defines the limits of possible access paths through indexes and stored records, and the search mechanisms define which paths are to be taken for a given applications. Access method design is often defined in terms of primary and secondary access path structure. The primary access paths are associated with initial record loading, or placement, and usually involve retrieval via the primary key. Individual files are first designed in this manner to process the dominant application most efficiently. Access time can be greatly reduced through secondary indexes, but at the expense of increased storage space overhead and index maintenance.

Program Design

Standard DBMS routines should be used for all accessing, and query or update transaction optimization should be performed at the systems software level. Application program design should be completed when the logical database structure is known.

11.3 OPERATIONS IN FILES

Operations on files are usually grouped into:

- (i) Retrieval operations
- (ii) Update operations.

Retrieval operations do not change any data in the file, but only locate certain records so that their field values can be examined and processed. Update operations change the file by insertion or deletion of records or by modification of field values. In both the cases one or more records are selected for retrieval, deletion, or modification based on a selection condition.

When several file records satisfy a search condition, the first record with respect to the physical sequence of file records is initially located and designated as the current record. Subsequent search operations commence from this record and locate the next record in the file that satisfies the condition. Actual operations for locating and accessing file records vary from system to system.

File operations performed in high level programs such as DBMS software programs are described below:

Open: Prepares the file for reading or writing; Allocates appropriate buffers to hold file blocks from disk, and retrieves the file header; Sets the file pointer to the beginning of the file.

Reset: Sets the file pointer of an open file to the beginning of the file.

Find or Locate: Searches for the first record that satisfies a search condition; Transfers the block containing that record into a main memory buffer.

Read or Get: Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.

Find Next: Searches for the next record in the file that satisfies the search condition; Transfers the block containing that record into a main memory buffer. The record is located in the buffer and becomes the current record.

Delete: Deletes the current record and updates the file on disk to reflect the deletion.

Modify: Modifies some field values for the current record and updates the file on disk to reflect the modification.

Insert: Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer, writing the record into the buffer, and writing the buffer to disk to reflect the insertion.

Close: Completes the file access by releasing the buffers and performing any other needed cleanup operations.

Scan: If the file has just been opened or reset, *Scan* returns the first record; otherwise it returns the next record. If a condition is specified with the operation, the returned record is the first or next record satisfying the condition.

Additional higher-level operations are as follows:

Find All: Locates *all* the records in the file that satisfy a search condition.

Find or Locate *n*: Searches for the first record that satisfies a search condition and then continues to locate the next $n - 1$ records satisfying the same condition. Transfers the blocks containing the *n* records to the main memory buffer.

Find Ordered: Retrieves all the records in the file in some specified order.

Reorganize: Starts the reorganization process.

File organization: Refers to the organization of the data of a file into records, blocks, and access structures. This includes the way records and blocks are placed on the storage medium and interlinked.

Access method: It specifies the group of operations that can be applied to a file.

11.4 FILE ORGANIZATION

The method of mapping file records to disk blocks is defined as file organization. A file is organized logically as a sequence of records. These records are mapped onto disk blocks. The following are the types of file organization:

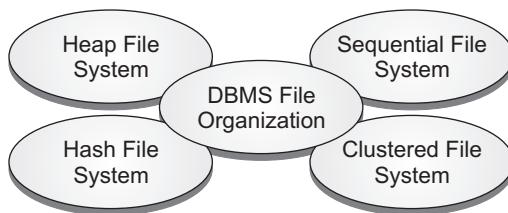


Fig. 11.1. File Organization types.

Heap File Organization: When a file is created using Heap File Organization mechanism, the Operating Systems allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of software to manage the records. Heap File does not support any ordering, sequencing or indexing on its own.

Sequential File Organization: Every file record contains a data field to uniquely identify that record. In sequential file organization mechanism, records are placed in the file in the some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.

Hash File Organization: This mechanism uses a Hash function computation on some field of the records. As we know, that file is a collection of records, which has to be mapped on some block of the disk space allocated to it. This mapping is defined by hash computation using a hash function. The output of hash determines the location of disk block where the records may exist.

Clustered File Organization: Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in a same disk block, that is, the ordering of records is not based on primary key or search key. This organization helps to retrieve data easily based on particular join condition.

11.4.1 Fixed-Length Records

Let us consider a file of student record as an example. Each record of this file is defined as:

```

type student = record
  Assignment_number : char(10);
  Student_name : char (22);
  Mark_total : real;
end
  
```

If we assume that each character occupies 1 byte and that a real occupies 8 bytes, our student record is 40 bytes long. A simple approach is to use the first 40 bytes for the first record, the next 40 bytes for the second record, and so on is shown in Figure 11.2.

The problems with this simple approach is as follows

- It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

- The block size happens to be a multiple of 40 but some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would require two block accesses to read or write such a record.

Record 0	RCS001	Peter	450
Record 1	RCS307	Rajesh	375
Record 2	RCS215	Mary	710
Record 3	RCS101	Dinesh	550
Record 4	RCS223	Ramesh	750
Record 5	RCS202	Peter	910
Record 6	RCS219	Bindhu	725
Record 7	RSC117	Dinesh	630
Record 8	RCS220	Peter	700

Fig. 11.2. File containing student record.

When a record is deleted, the record that came after it is moved into the space formerly occupied by the deleted record, and so on. All the record following the deleted record has to be moved ahead as shown in Figure 11.4. This approach requires moving a large number of records.

Record 0	RCS001	Peter	450
Record 1	RCS307	Rajesh	375
Record 3	RCS101	Dinesh	550
Record 4	RCS223	Ramesh	750
Record 5	RCS202	Peter	910
Record 6	RCS219	Bindhu	725
Record 7	RSC117	Dinesh	630
Record 8	RCS220	Peter	700

Fig. 11.3. Record 2 deleted in Fig. 11.2.

Record 0	RCS001	Peter	450
Record 1	RCS307	Rajesh	375
Record 3	RCS101	Dinesh	550
Record 4	RCS223	Ramesh	750
Record 5	RCS202	Peter	910
Record 6	RCS219	Bindhu	725
Record 7	RSC117	Dinesh	630
Record 8	RCS220	Peter	700

Fig. 11.4. All the records after the deleted record is moved one step ahead.

Another easier and simple method is to move the final record of the file into the space occupied by the deleted record as shown in Figure 11.5.

Record 0	RCS001	Peter	450
Record 1	RCS307	Rajesh	375
Record 8	RCS220	Peter	700
Record 3	RCS101	Dinesh	550
Record 4	RCS223	Ramesh	750
Record 5	RCS202	Peter	910
Record 6	RCS219	Bindhu	725
Record 7	RSC117	Dinesh	630

Fig. 11.5. Final record moved to the deleted record's space.

It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space.

To efficiently use the available free space due to the deletion of records, at the beginning of the file a certain number of bytes is used as a file header. The header will contain a variety of information about the file. The record will contain the address of the first record whose contents are deleted. This first record stores the address of the second deleted record, and so on. The deleted records form a linked list, which is referred to as a free list. Figure 11.6 shows the file of Figure 11.2, with the free list, after records 2, 4, and 7 have been deleted.

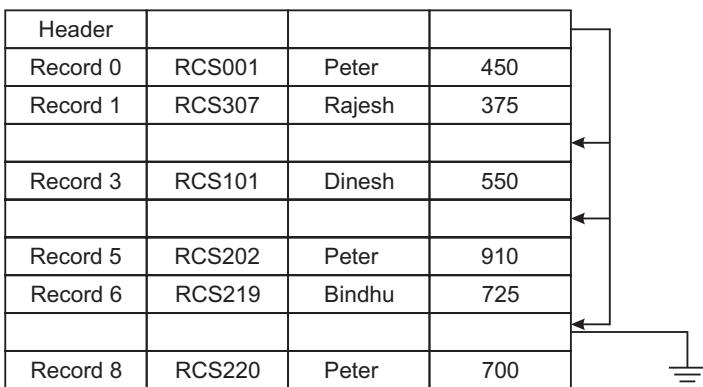


Fig. 11.6. Final record with free list after deleting record 2, 4 and 7.

On insertion of a new record, the header is made to point the next available record. If no space is available, new record is added to the end of the file. Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record. If variable-length record is inserted then this method cannot be used.

11.4.2 Variable-Length Records

Database system includes variable-length records in many cases which include:

- Storage of multiple record types in a file
- Record types that allow variable lengths for one or more fields
- Record types that allow repeating fields

Different techniques for implementing variable-length records exist. Consider the following record:

```
type Student_list = record
  Student_name : char (22);
  Student_info: array [1 ..∞] of
    record;
    Assignment_number : char(10);
    Total_marks : real;
  end
end
```

Student-info as an array with arbitrary number of elements. The type definition does not limit the number of elements in the array. There is no limit on how large a record can be.

11.4.2.1 Byte-String Representation

A simple method for implementing variable-length records is to attach a special *end-of-record* (\perp) symbol to the end of each record. Each record is stored as a string of consecutive bytes. Figure 11.6 shows such an organization to represent the file of fixed-length records of Figure 11.2 as variable-length records.

Record 0	RCS001	Peter	450	RCS202	910	RCS220	700	\perp
Record 1	RCS307	Rajesh	375	\perp				
Record 2	RCS215	Mary	710	\perp				
Record 3	RCS101	Dinesh	550	RSC117	630		\perp	
Record 4	RCS223	Ramesh	750	\perp				
Record 5	RCS219	Bindhu	725	\perp				

Fig. 11.6. Bytes string representation of variable length records.

In some cases instead of using an end-of-record symbol, byte-string representation stores the record length at the beginning of each record. Disadvantages of byte-string representation are as follows:

- It is not easy to reuse space occupied formerly by a deleted record.
- Techniques exist to manage insertion and deletion but it leads to a large number of small fragments of disk storage that are wasted.

- There is no space for records to grow longer. If a variable-length record becomes longer, it must be moved.
- Movement is costly if pointers to the record are stored elsewhere in the database, since the pointers must be located and updated.

Thus, the basic byte-string representation described here is not used for implementing variable-length records. Instead, a modified form of the byte-string representation called the slotted-page structure is commonly used for organizing records within a single block.

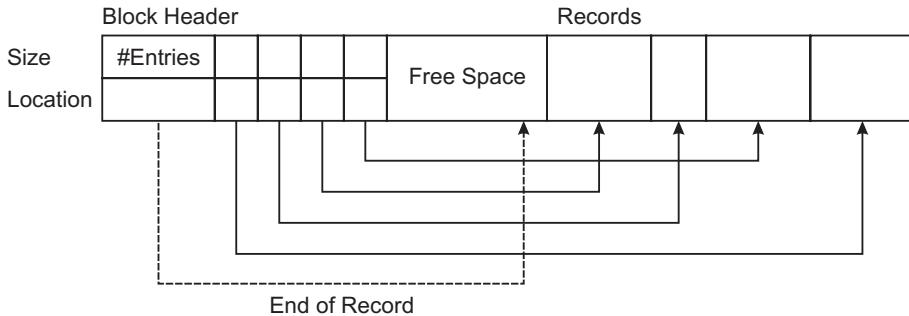


Fig. 11.7. Slotted-page structure.

Header at the beginning of each block contains the following information:

- (i) The number of record entries in the header
- (ii) The end of free space in the block
- (iii) An array whose entries contain the location and size of each record

The actual records are allocated contiguously in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array, and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

If a record is deleted, the occupied space is freed and its entry is deleted. The records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied, and all free space is again between the final entry in the header array and the first record. The end-of-free-space pointer in the header is appropriately updated as well.

Records can be grown or shrunk by similar techniques, as long as there is space in the block. The slotted-page structure does not require pointers that point directly to records. Instead, pointers are made to point the entry in the header that contains the actual location of the record. This prevents fragmentation of space inside a block.

11.4.2.2 Fixed-Length Representation

Variable-length records can be efficiently implemented in a file system by using one or more fixed-length records to represent one variable-length record. This implementation can be done in two ways:

1. **Reserved space.** If the records do not exceed a maximum record length then fixed-length records of that length can be used. Unused space is filled with a special null or end-of-record symbol.
2. **List representation.** Variable-length records can be represented by a list of fixed length records, chained together by pointers.

In order to use the reserved-space method, first thing to be done is to choose the maximum length of the record. In our example let us choose a maximum of three assignments and total marks for each student. Those records with less than three values are represented using end-of-record symbol. The reserved-space method is useful when most of the records have length closer to the maximum size chosen. Otherwise, a significant amount of space may be wasted.

Record 0	RCS001	Peter	450	RCS202	910	RCS220	700
Record 1	RCS307	Rajesh	375	⊥	⊥	⊥	⊥
Record 2	RCS215	Mary	710	⊥	⊥	⊥	⊥
Record 3	RCS101	Dinesh	550	RSC117	630	⊥	⊥
Record 4	RCS223	Ramesh	750	⊥	⊥	⊥	⊥
Record 5	RCS219	Bindhu	725	⊥	⊥	⊥	⊥

Fig. 11.8. Reserved space method for variable length record.

To represent the file by the linked list method pointer are added to the field. A disadvantage is that space is wasted in all records except the first in a chain. The first record needs to have the Student_name value, but subsequent records do not.

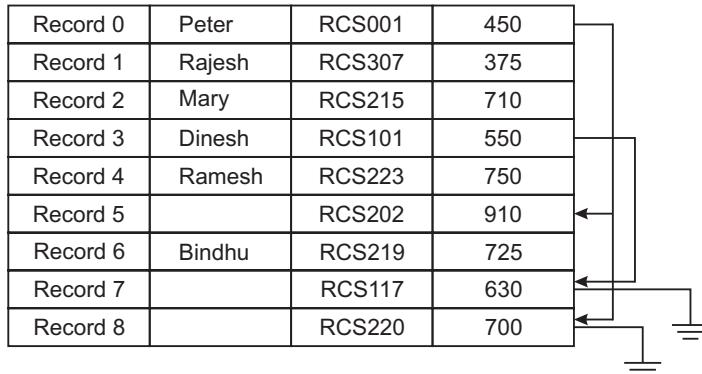


Fig. 11.9. Variable length record using linked list.

To overcome this problem, two kinds of blocks are used:

- **Anchor block**, which contains the first record of a chain
- **Overflow block**, which contains records other than those that are the first record of a chain

Thus, all records within a block have the same length, even if all records in the file do not have the same length.

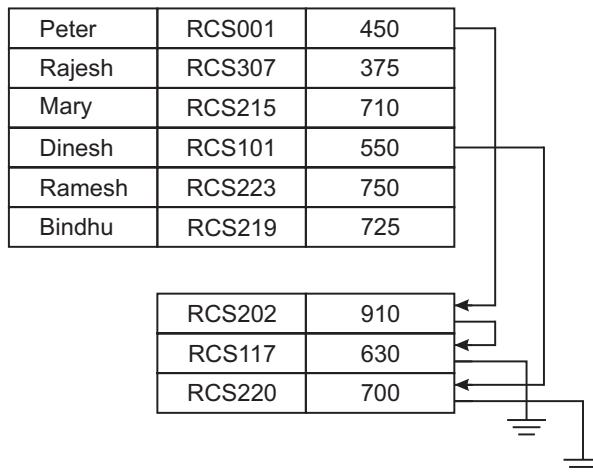


Fig. 11.10. Anchor block and overflow block structures.

11.5 ORGANIZATION OF RECORDS IN FILES

This topic discusses about the possible ways of organizing records in files:

Heap file organization: Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.

Sequential file organization: Records are stored in sequential order, according to the value of a “search key” of each record.

Hashing file organization: A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed. Separate file is used to store records of each relation.

Clustering file organization: Records of several different relations are stored in the same file. So that one I/O operation fetches related records from all the relations.

11.5.1 Sequential File Organization

A sequential file is designed for efficient processing of records in sorted order based on some search-key. A search key is any attribute or set of attributes; it need not be the primary key, or even a super key. To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the next record in search-key order. To minimize the number of block accesses in sequential file processing, records are stored physically in search-key order or as close to search-key order as possible.

Figure 11.11 shows a sequential file of *student* records. In that example, the records are stored in search-key order, using *student_name* as the search key. The sequential file organization allows records to be read in sorted order.

RCS219	Bindhu	725		
RCS101	Dinesh	550		←
RCS117	Dinesh	630		←
RCS215	Mary	710		←
RCS001	Peter	450		←
RCS202	Peter	910		←
RCS220	Peter	700		←
RCS307	Rajesh	375		←
RCS223	Ramesh	750		←

Fig. 11.11. Sequential file of records.

It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion. Deletion can be done efficiently using pointers. For insertion, the following rules are applied:

- (i) Locate the record in the file that comes before the record to be inserted in search-key order.
- (ii) If there is a free record within the same block as this record, insert the new record there. Otherwise, insert the new record in an overflow block. In either case, adjust the pointers so as to chain together the records in search-key order.

Figure 11.12 shows the file of Figure 11.11 after the insertion of the record (Nitin, RCS-880, 800). The structure in Figure 11.12 allows fast insertion of new records, but forces sequential file-processing applications to process records in an order that does not match the physical order of the records.

RCS219	Bindhu	725		
RCS101	Dinesh	550		←
RCS117	Dinesh	630		←
RCS215	Mary	710		←
RCS001	Peter	450		←
RCS202	Peter	910		←
RCS220	Peter	700		←
RCS307	Rajesh	375		←
RCS223	Ramesh	750		←
RCS880	Nitin	800		←

Fig. 11.12. Sequential file of records after insertion of new record.

If relatively few records need to be stored in overflow blocks, this approach works well. The association between search-key order and physical order may be totally lost, in such case sequential processing will become much less efficient. At this point, the file should be reorganized so that it is

once again physically in sequential order. Such reorganizations are costly, and must be done when the system load is low.

11.5.2 Clustering File Organization

Relational-database systems store each relation in a separate file, so that they can take full advantage of the file system that the operating system provides. Usually, tuples of a relation can be represented as fixed-length records. This simple implementation of a relational database system is well suited for low-cost database implementations like embedded systems or portable devices. In such systems, the size of the database is small.

A simple file structure reduces the amount of code needed to implement the system. This simple approach to relational-database implementation becomes less satisfactory as the size of the database increases. Large-scale database systems do not rely directly on the underlying operating system for file management. Instead, one large operating-system file is allocated to the database system. The database system stores all relations in this one file, and manages the file itself. To see the advantage of storing many relations in one file, consider the following SQL query for the bank database:

```
select assignment_number, student_name, street, city
from student_assignment_details, student_personal_details
where student_assignment_details.student_name = customer.student_name
```

This query computes a join of the *student_assignment_details* and *student_personal_details* relations. Thus, for each tuple of *student_assignment_details*, the system must locate the *student_personal_details* tuples with the same value for *student_name*.

Student_name	Assignment_name
Helen	RCS101
Helen	RCS232
Helen	RCS501
Tyson	RCS300

Fig. 11.13. *Student_assignment_details*.

Student_name	Street	City
Helen	West	Tirunelveli
Tyson	East	Chennai

Fig. 11.14. *Student_personal_details*.

Helen	West	Tirunelveli
Helen	RCS101	
Helen	RCS232	
Helen	RCS501	
Tyson	East	Chennai
Tyson	RCS300	

Fig. 11.15. Clustering file structure.



Fig. 11.16. Clustering file structure with pointers.

A clustering file organization is a file organization that stores related records of two or more relations in each block. Allows to read records that would satisfy the join condition by using one block read. Thus, such queries are processed more efficiently. Clustering has enhanced processing of a particular join (*student_assignment_details* join *student_personal_details* relations), but it results in slowing processing of other types of query. For example,

```
select *from student_assignment_details
```

requires more block accesses than when processing each relation in a separate file. Instead of several *student_assignment_details* records appearing in one block, each record is located in a distinct block. Finding all the *student_assignment_details* records is not possible without some additional structure. To locate all tuples of the *student_assignment_details* relation pointers are used, as in Figure 11.16. Need for clustering depends on the types of query that the database designer believes to be most frequent. Careful use of clustering can produce significant performance gains in query processing.

REVIEW QUESTIONS

1. What is the need for physical database design?
2. Explain different file organization.
3. Explain the steps involved in physical database design.
4. How will you organize records in file?
5. List and explain the different operations that can be performed in a file.
6. What are the ways to represent variable sized record?
7. What is the need for anchor blocks and overflow blocks?
8. Explain slotted page structure.
9. Discuss the drawbacks of byte string representation.



Chapter 12

Indexing and Hashing

12.1 INTRODUCTION

Indexing is a data structure technique to efficiently retrieve records from database files based on some attributes or set of attributes on which the indexing has been done. An attribute or set of attributes used to look up records in a file is called a search key. Indexing in database systems is similar to the one we see in books. It is used to speed up access to desired data.

An index is a small table having only two columns. The first column contains a copy of the primary or candidate key of a table and the second column contains a set of pointers holding the address of the disk block where that particular key value can be found.

The advantage of using index is that index makes search operation perform very fast. Suppose that a table has several rows of data, each row being 20 bytes wide. If you want to search for the record number 100, the management system must thoroughly read each and every row and after reading $99 \times 20 = 1980$ bytes it will find record number 100. If we have an index, the management system starts to search for record number 100 not from the table, but from the index. The index, containing only two columns, may be just 4 bytes wide in each of its rows. After reading only $99 \times 4 = 396$ bytes of data from the index the management system finds an entry for record number 100, reads the address of the disk block where record number 100 is stored and directly points at the record in the physical storage device. The result is a much quicker access to the record.

The minor disadvantages of using index is that it takes up a little more space than the main table and amount of I/O increases with the size of index. Additionally, index needs to be updated periodically for insertion or deletion of records in the main table. However, the advantages are so huge that these disadvantages can be considered negligible.

Types of Indices

Ordered indices: Based on a sorted ordering of the values.

Hash indices: Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a hash function.

Several techniques are available for both ordered indexing and hashing. No one technique is the best. Each technique is evaluated on the basis of the following factors:

Access types: The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.

Access time: The time taken to find a particular data item, or set of items, using the technique.
Insertion time: The time taken to insert a new data item. This value includes the time taken to find the correct place to insert the new data item, as well as the time taken to update the index structure.

Deletion time: The time taken to delete a data item. This value includes the time taken to find the item to be deleted, as well as the time taken to update the index structure.

Space overhead: The additional space occupied by an index structure. If the amount of additional space required is moderate then that space can be provided to improve the performance.

12.2 ORDERED INDICES

Index structure provides fast access to records in files and an index structure is associated with a particular search key. An ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a primary index is an index whose search key also defines the sequential order of the file. Primary indices are also called clustering indices. The search key of a primary index is usually the primary key, although that is not necessarily so. Indices whose search key specifies an order different from the sequential order of the file are called secondary indices, or nonclustering indices.

12.2.1 Primary Index

In primary index, there is a one-to-one relationship between the entries in the index table and the records in the main table. Primary index can be of two types:

1. Dense primary index
2. Sparse primary index

All files are ordered sequentially on some search key. Files with primary index are called as index-sequential files. This method is the oldest index schemes used in database systems. They are designed for applications that require both sequential processing of the entire file and random access to individual records.

Dense primary index: The number of entries in the index table is the same as the number of entries in the main table.

An index record appears for every search-key value in the file. In a dense primary index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search key-value would be stored sequentially after the first record.

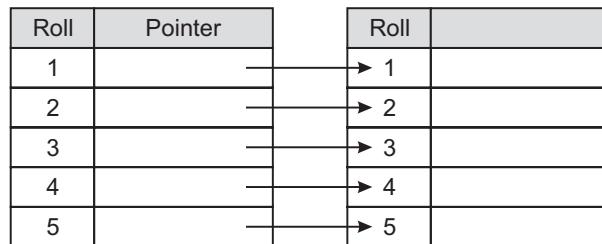


Fig. 12.1. Dense primary index.

Sparse or Non-Dense Primary Index: In case of large database the size of dense primary index will be large in size. To keep the size of the index smaller, an index record appears for only some of the search-key values. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

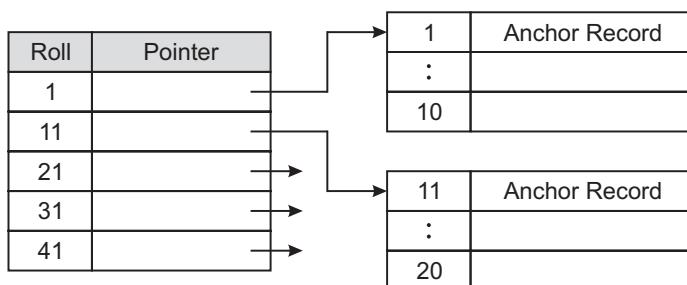


Fig. 12.2. Sparse primary index.

In the figure 12.2, the data blocks have been divided into several blocks, each containing a fixed number of records. The pointer in the index table points to the first record of each data block, which is known as the Anchor Record. If you are searching for roll 14, the index is first searched to find out the highest entry which is smaller than or equal to 14. We have 11. The pointer leads us to roll 11 where a short sequential search is made to find out roll 14.

12.2.2 Secondary Indices

Generally the index table is kept in the primary memory. The main table is kept in secondary memory because of its size. A table may contain millions of records (E.g. telephone directory), for which even a sparse index becomes so large in size that we cannot keep it in the primary memory. And if we cannot keep the index in the primary memory, then we lose the advantage of the speed of access. For very large table, it is better to organize the index in multiple levels as shown in figure 12.3.

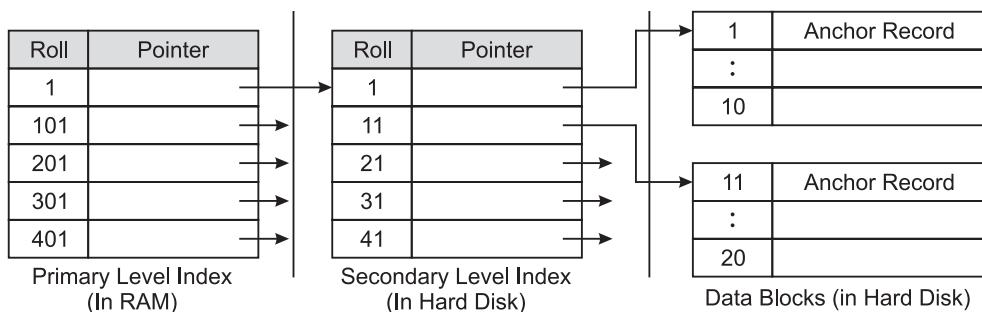


Fig. 12.3. Secondary indices.

In this scheme, the primary level index is kept in the RAM for quick reference. If you need to find out the record of roll 14 now, the index is first searched to find out the highest entry which is smaller than or equal to 14. We have 1. The adjoining pointer leads us to the anchor record of the corresponding secondary level index, where another similar search is conducted. This finally leads us to the actual data block whose anchor record is roll 11. We now come to roll 11 where a short sequential search is made to find out roll 14.

12.2.3 Multilevel Indices

Indices with two or more levels are called multilevel indices. Index records comprise search-key value and data pointers. This index itself is stored on the disk along with the actual database files. As the size of database grows, the size of indices also grows. There is an immense need to keep the index records in the main memory so that the search can speed up. If single level index is used then a large size index cannot be kept in memory as whole and this leads to multiple disk accesses. Multi-level Index helps breaking down the index into several smaller indices in order to make the outer most level so small that it can be saved in single disk block which can easily be accommodated anywhere in the main memory.

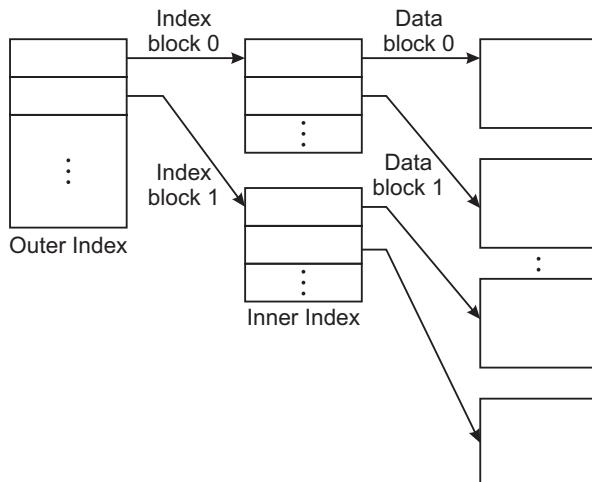


Fig. 12.4. Multilevel indices.

Multi-level Index helps breaking down the index into several smaller indices in order to make the outer most level so small that it can be saved in single disk block which can easily be accommodated anywhere in the main memory.

12.3 B⁺ TREE

B⁺ Tree indexes are an alternative to index sequential files. In case of index sequential files performance degrades as sequential file grows, because many overflow blocks are created. Periodic reorganization of entire file is required. B⁺ Tree index file automatically reorganizes itself with small, local changes in the case of insertions and deletions.

The B⁺ Tree index structure is the most widely used. A B⁺ Tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length.

12.3.1 Structure of B⁺ Tree

A B⁺-Tree is a rooted tree satisfying the following properties:

- All paths from the root to leaf have the same length. B⁺ Tree is a balanced tree.
- Each node that is not the root or a leaf node has between $[n/2]$ and n children, where n is fixed for a particular tree.
- A leaf node has between $[(n - 1)/2]$ and n – 1 values.
- If the root is not a leaf, it has at least 2 children. If the root is a leaf, it can have between 0 and n – 1 values.
- The search keys in a node are ordered, i.e., $K_1 < K_2 < K_3 \dots < K_{n-1}$
- Structure of a B⁺ Tree node:

P ₁	K ₁	P ₂	...	P _{n-1}	K _{n-1}	P _n
----------------	----------------	----------------	-----	------------------	------------------	----------------

Fig. 12.5. Node structure of a B⁺ tree.

Where

K_i are the search key values

P_i are pointers to non-leaf nodes or pointers to records or buckets of records

- For $i = 1, 2, \dots, n - 1$, pointer P_i either points to a file record with search key value K_i, or to a bucket of pointers to file records, each record having search key value K_i.
- If L_i, L_j are leaf nodes and $i < j$, L_i's search key values are less than L_j's search key values.
- P_n points to next leaf node in search key order.
- The non-leaf levels of the B⁺ Tree form a hierarchy of sparse indices.
- The B⁺ Tree contains a relatively small number of levels, thus searches can be done efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

Example: B^+ tree

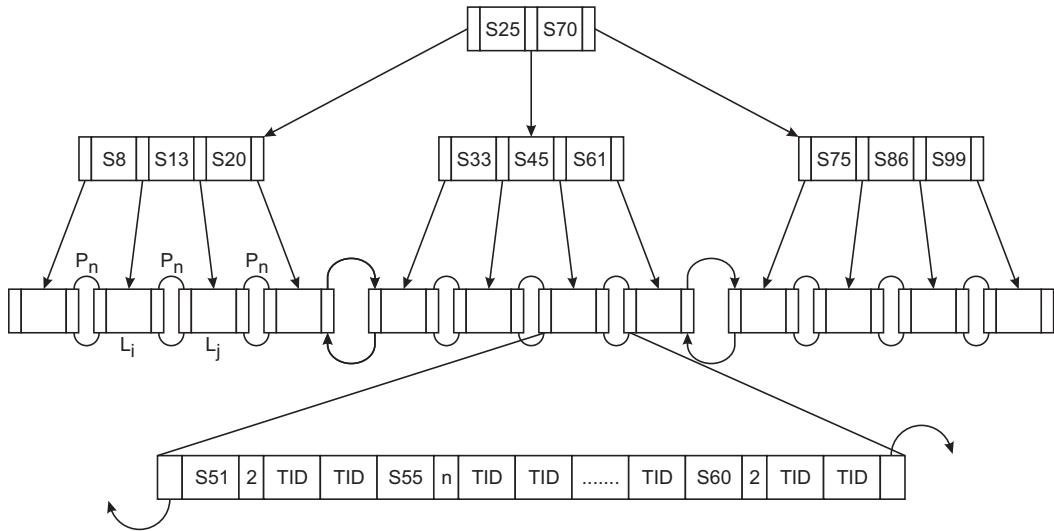


Fig. 12.6. B^+ tree Example.

12.3.2 Queries on B^+ Trees

Find all records with a search key value of k

- Start with the root node
- Examine the node for the smallest search key value > k.
- If such a value exists, assume it as K_i . Then follow P_i to the child node.
- Otherwise, $k \geq K_{m-1}$, where there are m pointers in the node. Then follow P_m to the child node.
- If the node is reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
- Eventually reach a leaf node. Scan entries K_i in the leaf node. If $K_i = k$, follow pointer P_i to the desired record or bucket. Otherwise no record with search key value k exists.
- If there are V search key values in the file, the path from the root to a leaf node is no longer than $\lfloor \log_{[n/2]}(V) \rfloor$.
- In general a node has the same size as a disk block, typically 4KB, and $n \approx 100$ (40 bytes per index entry).
- With 1,000,000 search key values and $n = 100$, at most $\log_{50}(1,000,000) = 4$ nodes are accessed.
- Index file degradation problem is solved by using B^+ Tree indices. Data file degradation problem is solved by using a B^+ Tree file organization.
- Leaf nodes in a B^+ Tree file organization can store records instead of just pointers.

12.4 HASHING TECHNIQUES

File organizations based on hashing technique allows avoiding accessing an index structure. Hashing provides a way of constructing indices. In a hash file organization, the address of the disk block containing a desired record is obtained directly by computing a function on the search-key value of the record. In hashing the storage unit used to store one or more records is called as a bucket. A bucket is typically a disk block.

Let K denote the set of all search-key values, and let B denote the set of all bucket addresses. A hash function h is a function from K to B . Let h denote a hash function. To insert a record with search key K_i , $h(K_i)$ is computed. This gives the address of the bucket for that record.

If there is space in the bucket to store the record, then the record is stored in that bucket. Suppose if two search keys, K_3 and K_5 , have the same hash value, i.e., $h(K_3) = h(K_5)$. The bucket $h(K_3)$ contains records with search-key values K_3 and records with search key values of K_5 . Thus, the search-key value of every record is checked in the bucket to verify that the required record is selected.

Deletion is equally straightforward. If the search-key value of the record to be deleted is K_i , then $h(K_i)$ is computed, then search the corresponding bucket for that record, and delete the record from the bucket.

12.4.1 Hash Functions

An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records. The worst possible hash function maps all search-key values to the same bucket. Such a function is undesirable because all the records have to be kept in the same bucket. A lookup has to examine every such record to find the one desired.

A hash function has to be chosen such that it assigns search-key values to buckets in such a way that the distribution has these qualities:

- The distribution is uniform. That is, the hash function assigns each bucket the same number of search-key values from the set of all possible search-key values.
- The distribution is random. That is, in the average case, each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search-key values. More precisely, the hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function will appear to be random.

Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well-designed function gives an average-case lookup time that is a constant, independent of the number of search keys in the file.

12.4.2 Handling of Bucket Overflows

When a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a bucket overflow is said to occur. Bucket overflow can occur for several reasons:

Insufficient buckets: The number of buckets, which we denote n_B , must be chosen such that $n_B > n_r/f_r$, where n_r denotes the total number of records that will be stored, and f_r denotes the number of records that will fit in a bucket. This designation, of course, assumes that the total number of records is known when the hash function is chosen.

Skew: Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket skew. Skew can occur for two reasons:

1. Multiple records may have the same search key.
2. The chosen hash function may result in non uniform distribution of search keys.

So the probability of bucket overflow is reduced, the number of buckets is chosen to be $(n_r/f_r) * (1 + d)$, where d is a fudge factor, around 0.2. About 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.

Bucket overflow can also be handled by using overflow buckets. If a record must be inserted into a bucket b , and b is already full, the system provides an overflow bucket for b , and inserts the record into the overflow bucket. If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked list, as in Figure 12.7. Overflow handling using such a linked list is called overflow chaining.

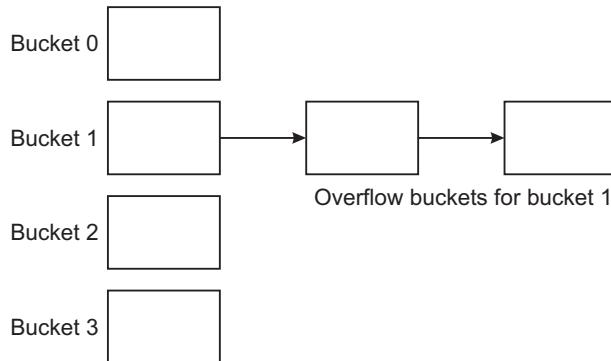


Fig. 12.7. Overflow chaining.

The lookup algorithm of overflow chaining should be changed. As before, the system uses the hash function on the search key to identify a bucket b . The system must examine all the records in bucket b to see whether they match the search key, as before. In addition, if bucket b has overflow buckets, the system must examine the records in all the overflow buckets also. The form of hash structure that we have just described is sometimes referred to as closed hashing.

In another method called open hashing, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets B . One policy is to use the next bucket that has space; this policy is called linear probing. Other policies, such as computing further hash functions, are also used.

Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems. The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations

on their symbol tables. However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open hashing is of only minor importance in database implementation. An important drawback to the form of hashing is to choose the hash function when we implement the system, because it cannot be changed easily if the file being indexed grows or shrinks. Since the function h maps search-key values to a fixed set B of bucket addresses, space is wasted if B is made large to handle future growth of the file. If B is too small, the buckets contain records of many different search-key values, and bucket overflows can occur. As the file grows, performance suffers.

12.4.3 Hash Indices

Hashing can be used for index-structure creation. A hash index organizes the search keys, with their associated pointers, into a hash file structure. Hash index is constructed as follows. A hash function is applied on a search key to identify a bucket, and store the key and its associated pointers in the bucket or in overflow buckets. Figure 12.8 shows a secondary hash index on the instructor file, for the search key ID. The hash function in the figure computes the sum of the digits of the ID modulo

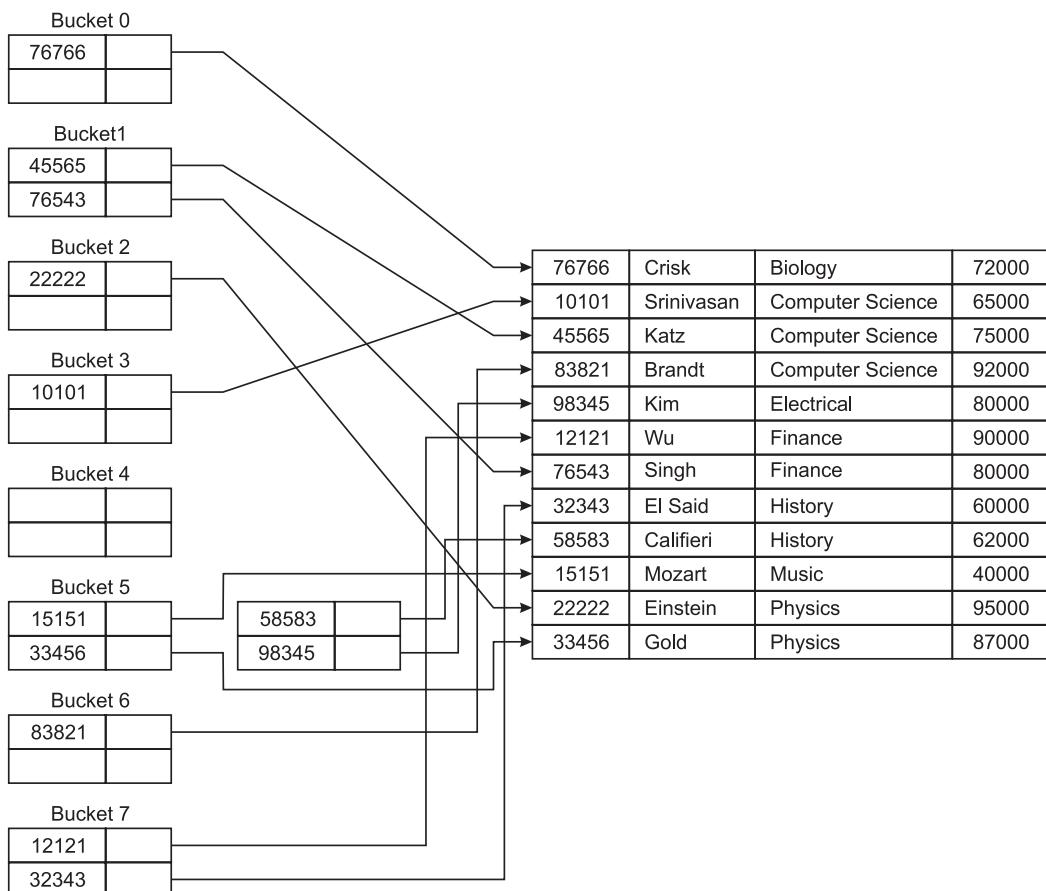


Fig. 12.8. Hash Index.

8. The hash index has eight buckets, each of size 2. One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, ID is a primary key for instructor, so each search key has only one associated pointer. In general, multiple pointers can be associated with each key. Hash index is used to denote hash file structures as well as secondary hash indices. Hash indices are only secondary index structures. A hash index is never needed as a primary index structure because if a file itself is organized by hashing, there is no need for a separate hash index structure on it.

12.4.4 Dynamic Hashing

Most databases grow larger over time. If static hashing is used for such a database then:

1. Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
2. Choose a hash function based on the expected size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.
3. Periodically reorganize the hash structure in response to file growth. Such reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments. This reorganization is a massive, time-consuming operation.

These problems can be avoided by using dynamic hashing technique. Dynamic hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database. One form of dynamic hashing is extendable hashing.

12.4.5 Extendable Hashing

Extendable hashing copes with changes in database size by splitting and joining together buckets as the database grows and shrinks. As a result, space efficiency is retained. Moreover, since the reorganization is performed on only one bucket at a time, the resulting performance overhead is acceptably low.

Extendable hashing chooses a hash function h with the desirable properties of uniformity and randomness. Buckets are not created for each hash value. Instead, buckets are created on demand, as records are inserted into the file. Initially the entire b bits of the hash value is not used. At any point, only i bits are used, where $0 \leq i \leq b$. These i bits are used as an offset into an additional table of bucket addresses. The value of i grows and shrinks with the size of the database. Figure 12.9 shows a general extendable hash structure.

The i appearing above the bucket address table in the figure indicates that i bits of the hash value $h(K)$ are required to determine the correct bucket for K . This number will change as the file grows. Although i bits are required to find the correct entry in the bucket address table, several consecutive table entries may point to the same bucket. All such entries will have a common hash prefix, but the length of this prefix may be less than i . Therefore, each bucket is prefixed with an integer giving the length of the common hash prefix.

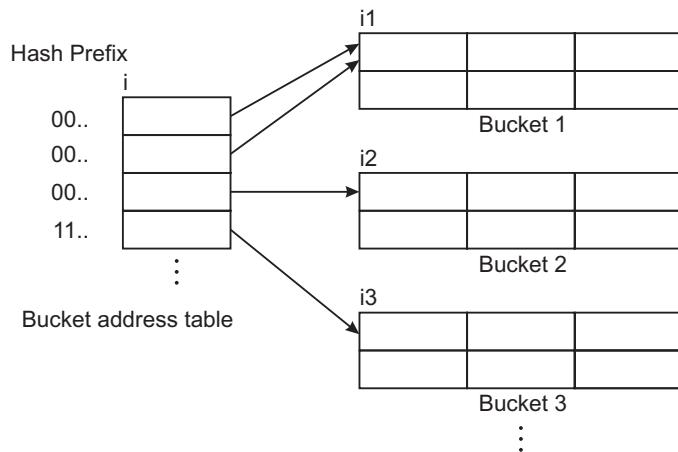


Fig. 12.9. Extendable Hash Structure.

12.4.6 Queries and Updates

To locate the bucket containing search-key value K_p , the system takes the first i high-order bits of $h(K_p)$, looks at the corresponding table entry for this bit string, and follows the bucket pointer in the table entry.

To insert a record with search-key value K_p , the system follows the same procedure for lookup as before, ending up in some bucket say, j . If there is room in the bucket, the system inserts the record in the bucket. If, on the other hand, the bucket is full, it must split the bucket and redistribute the current records, plus the new one. To split the bucket, the system must first determine from the hash value whether it needs to increase the number of bits that it uses.

Dept_number	Instructor_name	Dept_name	Salary
10101	Srinivasan	Computer Science	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Computer Science	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crisk	Biology	72000
83821	Brandt	Computer Science	92000
98345	Kim	Electrical	80000

Fig. 12.10. Instructor Table.

To delete a record with search-key value K_i , the system follows the same procedure for lookup as before, ending up in some bucket say, j . It removes both the search key from the bucket and the record from the file. The bucket too is removed if it becomes empty. At this point, several buckets can be combined together.

Consider the instructor file in Figure 12.10. Dept_number is assumed to be the search key value. The 32-bit hash values on dept_number are shown in Figure 12.11. Initially, the file is empty, as in Figure 12.12. The records are inserted one by one.

Dept_name	$h(\text{dept_name})$
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Computer Science	1111 0001 0010 0100 1001 0011 0110 1101
Electrical	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Fig. 12.11. Hash function for Dept_name.

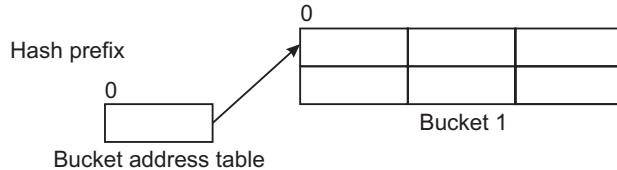
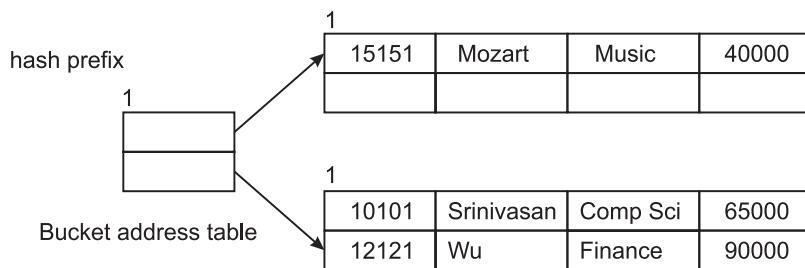


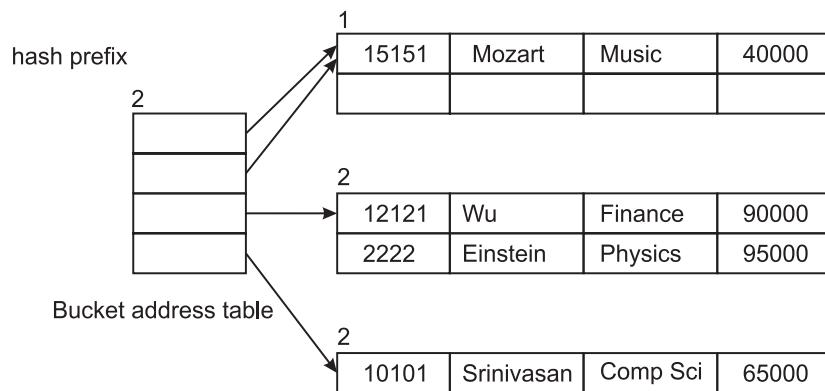
Fig. 12.12. Initial Hash structure.

When the record (10101, Srinivasan, Computer Science, 65000) is inserted. The bucket address table contains a pointer to the one bucket, and the system inserts the record. Next, the record (12121, Wu, Finance, 90000) is inserted. The system places this record also in the one bucket of the structure.

When an attempt is made to insert the next record (15151, Mozart, Music, 40000), it is noticed that the bucket is full. Since $i = i_0$, there is a need to increase the number of bits that is used from the hash value. Now 1 bit is used which allows $2^1 = 2$ buckets. This increase in the number of bits necessitates doubling the size of the bucket address table to two entries. The system splits the bucket, placing in the new bucket those records whose search key has a hash value beginning with 1, and leaving in the original bucket the other records. Figure 12.13 shows the state of structure after the split.

**Fig. 12.13.** Hash structure after 3 insertions.

Then the next record (22222, Einstein, Physics, 95000) is inserted. It is noticed that the first bit of $h(\text{Physics})$ is 1, this record must be inserted into the bucket pointed to by the “1” entry in the bucket address table. Once again it is found that the bucket full and $i = i_1$. The number of bits that is used is increased from the hash to 2. This increase in the number of bits necessitates doubling the size of the bucket address table to four entries, as in Figure 12.14.

**Fig. 12.14.** Hash structure after 4 insertions.

For each record in the bucket of Figure 12.11 for hash prefix 1, the system examines the first 2 bits of the hash value to determine which bucket of the new structure should hold it.

Next, (32343, El Said, History, 60000) is inserted which hashes into the same bucket as that of Computer Science. Next record, (33456, Gold, Physics, 87000) results in a bucket overflow, leading to an increase in the number of bits, and a doubling of the size of the bucket address table as in Figure 12.15.

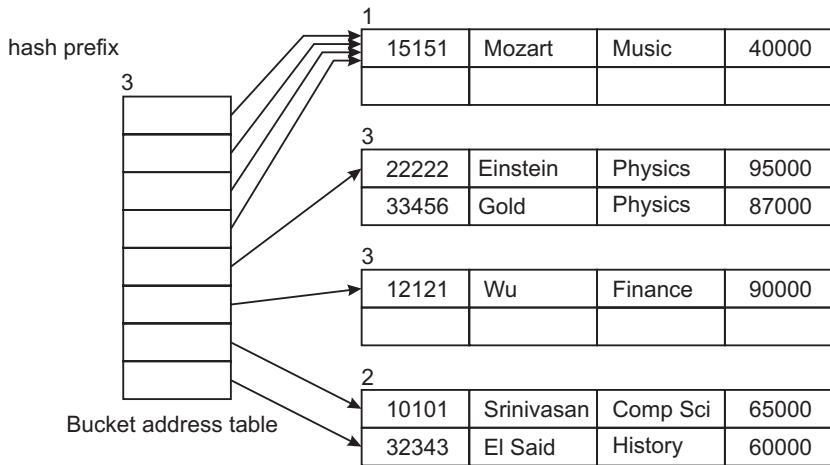


Fig. 12.15. Hash structure after 6 insertions.

The insertion of (45565, Katz, Computer Science, 75000), leads to another overflow. This overflow cannot be handled by increasing the number of bits, since there are three records with exactly the same hash value. Hence, the system uses an overflow bucket, as in Figure 12.16.

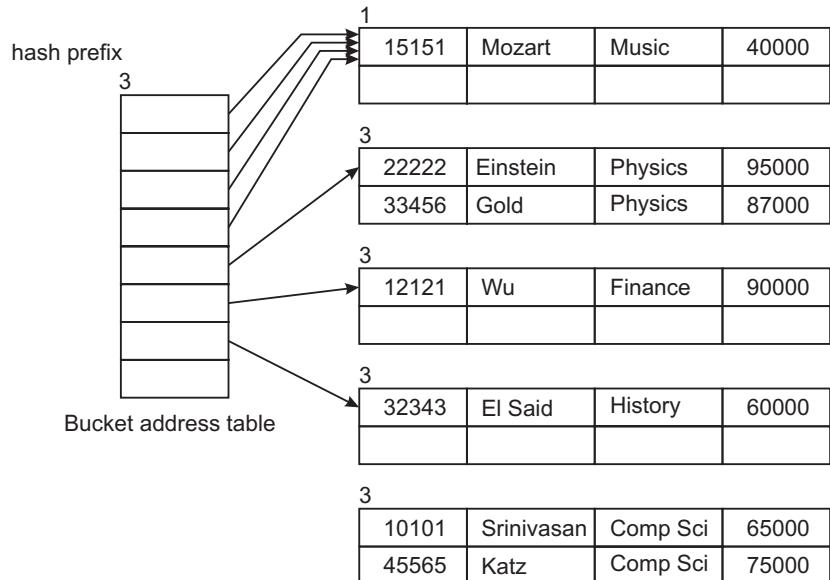


Fig. 12.16. Hash structure after 7 insertions.

Next the insertion of “Califieri”, “Singh” and “Crisk” are carried out without any bucket overflow. The insertion of the record (83821, Brandt, Computer Science, 92000) leads to another overflow. This overflow cannot be handled by increasing the number of bits, since there are three

records with exactly the same hash value. Therefore, the system uses an overflow bucket, as in figure 12.17.

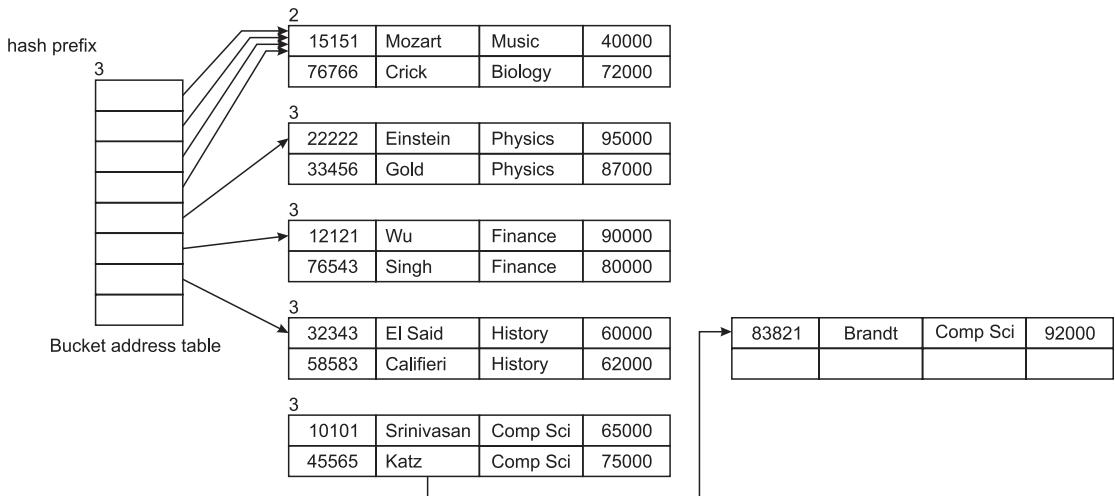


Fig. 12.17. Hash structure after 11 insertions.

The same procedure is followed for all the records. Finally after inserting all the records of the instructor table, the hash structure appears as in Figure 12.18.

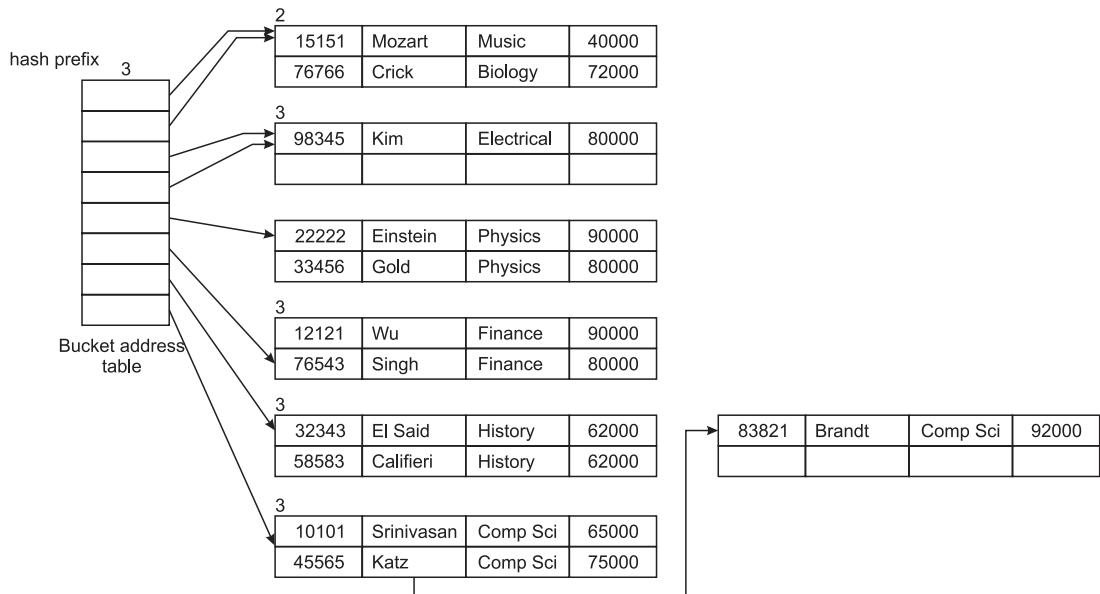


Fig. 12.18. Extendible Hash structure for the instructor table.

12.4.7 Comparison with Other Schemes

The advantages and disadvantages of extendable hashing are compared with the other schemes. The main advantage of extendable hashing is that performance does not degrade as the file grows and there is minimal space overhead.

Although the bucket address table deserves additional overhead, it contains one pointer for each hash value for the current pre fix length. This table is thus small. The main space saving of extendable hashing over other forms of hashing is that no buckets need to be reserved for future growth; rather, buckets can be allocated dynamically.

A disadvantage of extendable hashing is that lookup involves an additional level of indirection, since the system must access the bucket address table before accessing the bucket itself. This extra reference has only a minor effect on performance. Extendable hashing appears to be a highly attractive technique, provided that the added complexity involved in its implementation is accepted.

REVIEW QUESTIONS

1. Explain the different types of indices.
2. When is it preferable to use a dense index rather than a sparse index?
3. What is the difference between a primary index and a secondary index?
4. Explain in detail the use of B tree as an indexing technique. Compare B tree and B⁺ tree.
5. Describe the structure of B⁺ tree and how update operations are performed on B⁺ tree.
6. Describe briefly static and dynamic hashing.
7. How to reduce bucket overflow in hash file organization?



Chapter 13

Query Processing

13.1 INTRODUCTION

The activities involved in retrieving data from the database are called as query processing. The aims of query processing are to transform a query written in a high-level language typically SQL, into a correct and efficient execution strategy expressed in a low-level language implementing relational algebra, and to execute the strategy to retrieve the required data.

An important aspect of query processing is Query Optimization. The activity of choosing an efficient execution strategy for processing a query is called as query optimization. As there are many equivalent transformations of the same high-level query, the aim of query optimization is to choose the one that minimizes the resource usage. A DBMS uses different techniques to process, optimize, and execute high level queries (SQL). A query expressed in high-level query language must be first scanned, parsed, and validated. The scanner identifies the language components (tokens) in the text of the query, while the parser checks the correctness of the query syntax. The query is also validated (by accessing the system catalog) whether the attribute names and relation names are valid. An internal representation (tree or graph) of the query is created. Queries are parsed and then presented to a query optimizer, which is responsible for identifying an efficient plan. The optimizer generates alternative plans and chooses the plan with the least estimated cost.

13.2 STEPS IN QUERY PROCESSING

The steps involved in processing a query are shown in Figure 13.1. The basic steps are:

1. Parsing and translation
2. Optimization
3. Evaluation

The query processing begins only after the system has translated the query into usable form. Database languages like SQL will be suitable for users. These languages act as an interface between the user and the database servers. But it does not help in the system's internal representation of a query. Extended relational algebra is used to represent the query in system's internal representation.

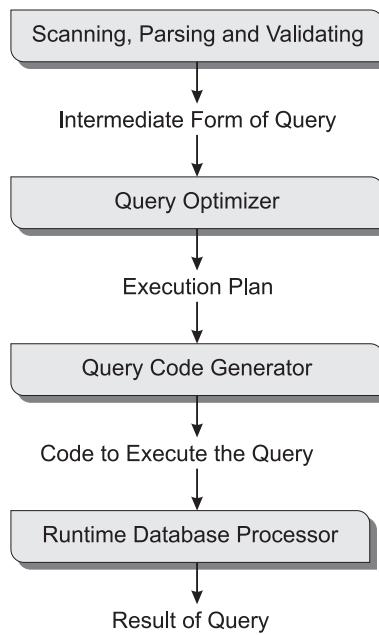


Fig. 13.1. Steps in Query processing.

The first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression.

Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query. There are several ways to evaluate relational-algebra expressions. As an example, consider the following query

*Select salary
From instructor
Where salary < 75000;*

This query can be translated into either of the following relational-algebra expressions:

- $\sigma_{\text{salary} < 75000} (\Pi_{\text{salary}}(\text{instructor}))$
- $\Pi_{\text{salary}}(\sigma_{\text{salary} < 75000} (\text{instructor}))$

Apart from representing the query in relational algebra form, it also needs to be interpreted with instructions specifying how to evaluate each operation. A relational-algebra operation interpreted with instructions on how to evaluate it is called an evaluation primitive. A sequence of primitive operations that can be used to evaluate a query is a query execution plan or query-evaluation plan. Figure 13.2 illustrates an evaluation plan for the above specified example query.

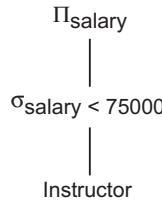


Fig. 13.2. *Query evaluation plan.*

The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query. The different evaluation plans for a given query can have different costs. The system takes the responsibility to construct a query-evaluation plan that minimizes the cost of query evaluation. Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output. Not all databases follow exactly the same steps.

13.3 MEASURES OF QUERY COST

Multiple evaluation plans are possible for a query. Therefore, it is necessary to compare the alternatives in terms of their cost to choose the best ones. Comparison is performed by estimating the cost of the individual operations and then the calculated individual costs are combined together to estimate the cost of the query evaluation plan.

The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication. The response time for a query-evaluation plan could be used as a good measure of the cost of the plan.

In large database systems disk accesses are considered to be the most important cost, since disk accesses are slow compared to in-memory operations. Moreover, CPU speeds have been improving much faster than disk speeds. Thus, it is likely that the time spent in disk activity will continue to dominate the total time to execute a query. Finally, estimating the CPU time is relatively hard compared to estimating the disk-access cost. Therefore, most people consider the disk-access cost a reasonable measure of the cost of a query-evaluation plan.

The number of block transfers from disk is a measure of the actual cost. To simplify the computation of disk-access cost, it is assumed that all transfers of blocks have the same cost. These assumptions ignore the variance arising from rotational latency and seek.

To get more precise numbers, it is necessary to distinguish between sequential I/O, where the blocks read are contiguous on disk, and random I/O, where the blocks are noncontiguous, and an extra seek cost must be paid for each disk I/O operation. It is also needed to distinguish between reads and writes of blocks, since it takes more time to write a block to disk than to read a block from disk. A more accurate measure would therefore estimate the following:

1. The number of seek operations performed
2. The number of blocks read
3. The number of blocks written

Then it adds up these numbers after multiplying them by the average seek time, average transfer time for reading a block, and average transfer time for writing a block, respectively. Real life query optimizers also take CPU costs into account when computing the cost of an operation.

13.4 SELECTION OPERATION

In query processing, the file scan is the lowest-level operation to access data. File scans are search algorithms that locate and retrieve records that fulfill a selection condition. In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.

13.4.1 Basic Algorithms

Consider a selection operation on a relation whose tuples are stored together in one file. Two scan algorithms to implement the selection operation are:

Linear search based algorithm: In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. For a selection on a key attribute, the system can terminate the scan if the required record is found, without looking at the other records of the relation. The cost of linear search, in terms of number of I/O operations, is b_r , where b_r denotes the number of blocks in the file. Selections on key attributes have an average cost of $b_r/2$, but still have a worst-case cost of b_r . Although it may be slower than other algorithms for implementing selection, the linear search algorithm can be applied to any file, regardless of the ordering of the file, or the availability of indices, or the nature of the selection operation. Other algorithms cannot be applicable in all cases, but when applicable they are generally faster than linear search.

Binary search based algorithm: If the file is ordered on an attribute, and the selection condition is an equality comparison on the attribute, then binary search is used to locate records that satisfy the selection. The system performs the binary search on the blocks of the file. The number of blocks that need to be examined to find a block containing the required records is $(\log_2(b_r))$, where b_r denotes the number of blocks in the file. If the selection is on a non key attribute, more than one block may contain required records, and the cost of reading the extra blocks has to be added to the cost estimate.

13.4.2 Selections Using Indices

Index structures are referred to as access paths, since they provide a path through which data can be located and accessed. A primary index is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file. An index that is not a primary index is called a secondary index. Search algorithms that use an index are referred to as index scans. Search algorithms that use an index are:

Primary index, equality on key algorithm: For an equality comparison on a key attribute with a primary index, the index is used to retrieve a single record that satisfies the corresponding equality condition. If a B+-tree is used, the cost of the operation, in terms of I/O operations, is equal to the height of the tree plus one I/O to fetch the record.

Primary index, equality on non key algorithm: Multiple records are retrieved by using a primary index when the selection condition specifies an equality comparison on a non key attribute, A.

The only difference from the previous case is that multiple records may need to be fetched. However, the records would be stored consecutively in the file since the file is sorted on the search key. The cost of the operation is proportional to the height of the tree, plus the number of blocks containing records with the specified search key.

Secondary index, equality algorithm: Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may get retrieved if the indexing field is not a key. In the first case, only one record is retrieved, and the cost is equal to the height of the tree plus one I/O operation to fetch the record. In the second case each record may be resident on a different block, which may result in one I/O operation per retrieved record. The cost could become even worse than that of linear search if a large number of records are retrieved.

13.4.3 Selections Involving Comparisons

Consider a selection of the form $\sigma_{A \leq v}(r)$. This selection is implemented either by using a linear or binary search or by using indices in one of the following ways:

Primary index, comparison Algorithm: A primary ordered index can be used when the selection condition is a comparison. For comparison conditions of the form $A > v$ or $A \geq v$, a primary index on A can be used to direct the retrieval of tuples, as follows. For $A \geq v$, the value v is looked in the index to find the first tuple in the file that has a value of $A = v$. A file scan starting from that tuple up to the end of the file returns all tuples that satisfy the condition. For $A > v$, the file scan starts with the first tuple such that $A > v$. For comparisons of the form $A < v$ or $A \leq v$, an index lookup is not required. For $A < v$, a simple file scan is used starting from the beginning of the file, and continuing up to the first tuple with attribute $A = v$. The case of $A \leq v$ is similar, except that the scan continues up to the first tuple with attribute $A > v$. In either case, the index is not useful.

Secondary index, comparison Algorithm: A secondary ordered index is used to guide retrieval for comparison conditions involving $<$, \leq , \geq , or $>$. The lowest level index blocks are scanned, either from the smallest value up to v , or from v up to the maximum value. The secondary index provides pointers to the records, but the actual record is fetched by using pointers.

13.4.4 Implementation of Complex Selections

The above discussed algorithms in the previous sections considered only simple selection conditions of the form $A op B$, where op is an equality or comparison operation. This section deals with more complex selection predicates.

Conjunction: A conjunctive selection is a selection of the form

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$$

Disjunction: A disjunctive selection is a selection of the form

$$\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$$

A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θ_i .

Negation: The result of a selection $\sigma_\theta(r)$ is the set of tuples of r for which the condition θ evaluates to false. In the absence of nulls, this set is simply the set of tuples that are not in $\sigma_\theta(r)$.

Selection operation involving either a conjunction or a disjunction of simple conditions is implemented by using one of the following algorithms:

Conjunctive selection using one index: First thing to be determined is to check whether an access path is available for an attribute in one of the simple conditions. If present then any one of the selection algorithms discussed in Selections Using Indices or Selections Involving Comparisons can be used to retrieve records satisfying that condition.

Conjunctive selection using composite index: An appropriate composite index may be available for some conjunctive selections. If the selection specifies an equality condition on two or more attributes, and a composite index exists on these combined attribute fields, then the index can be searched directly. The type of index determines which of algorithms to be used.

Conjunctive selection by intersection of identifiers: Another alternative for implementing conjunctive selection operations involves the use of record pointers or record identifiers. This algorithm requires indices with record pointers, on the fields involved in the individual conditions. The algorithm scans each index for pointers to tuples that satisfy an individual condition. The intersection of all the retrieved pointers is the set of pointers to tuples that satisfy the conjunctive condition. The algorithm then uses the pointers to retrieve the actual records. If indices are not available on all the individual conditions, then the algorithm tests the retrieved records against the remaining conditions. The cost of algorithm is the sum of the costs of the individual index scans, plus the cost of retrieving the records in the intersection of the retrieved lists of pointers. This cost can be reduced by sorting the list of pointers and retrieving records in the sorted order.

Disjunctive selection by union of identifiers: If access paths are available on all the conditions of a disjunctive selection, each index is scanned for pointers to tuples that satisfy the individual condition. The union of all the retrieved pointers yields the set of pointers to all tuples that satisfy the disjunctive condition. Then the pointers are used to retrieve the actual records. However, if even one of the conditions does not have an access path, then linear scan of the relation is performed to find tuples that satisfy the condition. Therefore, if there is one such condition in the disjunct, the most efficient access method is a linear scan, with the disjunctive condition tested on each tuple during the scan.

13.5 SORTING

Sorting of data in a database system is important for the following two reasons:

1. Output can be displayed in sorted manner
2. Efficient implementation of relational operations

Sorting can be performed by building an index on the sort key, then using that index to read the relation in sorted order. This process orders the relation logically and not physically. This may lead to one disk block access for each tuple. This is expensive as the number of records can be much larger than the number of blocks. For this reason, it may be desirable to order the records physically. The sorting can be applied to the relations that fit in main memory and to the relations that do not fit in main memory. For relations that fit in main memory techniques such as quick sort can be used. For relations that do not fit in memory external sorting is used. The most commonly used external sorting technique is external sort merge algorithm.

13.5.1 External Sort Merge Algorithm

The external sort-merge algorithm is described as follows. Let M denote the number of page frames in the main-memory buffer.

1. Create number of sorted runs. Initially let i be 0.

Repeatedly do the following till the end of the relation:

- (i) Read the M blocks of relation into the memory.
- (ii) Sort the in memory blocks.
- (iii) Write sorted data to run file R_i , increment i .

2. Merge the run. It is assumed that total number of runs, N , is less than M .

- (i) Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page.
- (ii) Repeat
 - (a) Select the first record among all buffer pages.
 - (b) Write the record to the output buffer. If the output buffer is full write it to disk.
 - (c) Delete the record from its input buffer page.

If the buffer page becomes empty then

Read the next block of the run into the buffer.

- (iii) Until all input buffer pages are empty

3. If $I \geq M$, several merge passes are required.

- (i) In each pass, contiguous groups of $M-1$ runs are merged.
- (ii) A pass reduces the number of runs by a factor of $M-1$ and creates runs longer by the same factor.
 - (a) For example: if $M=11$ and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs.
 - (iii) Repeated passes are performed till all runs have been merged into one.

The output of the merge stage is the sorted relation. The output file is buffered to reduce the number of disk write operations. In general, if the relation is much larger than memory, there may be M or more runs generated in the first stage, and it is not possible to allocate a page frame for each run during the merge stage. In this case, the merge operation proceeds in multiple passes. Since there is enough memory for $M - 1$ input buffer pages, each merge can take $M - 1$ runs as input.

The initial pass merges the first $M - 1$ runs to get a single run for the next pass. Then, it merges the next $M - 1$ runs similarly, and so on, until it has processed all the initial runs. The number of runs will be reduced by a factor of $M - 1$. If this reduced number of runs is still greater than or equal to M , another pass is made, with the runs created by the first pass as input. Each pass reduces the number of runs by a factor of $M - 1$. The passes repeat as many times as required, until the number of runs is less than M ; a final pass then generates the sorted output. Following figure 13.3 shows an example for external merge sort. It is assumed that only one tuple fits in a block and memory holds at most three page frames. During the merge stage, two page frames are used for input and one for output.

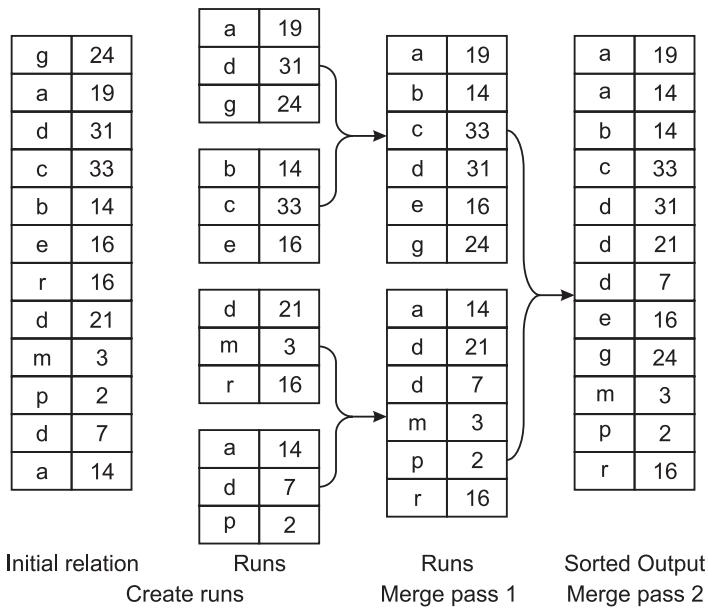


Fig. 13.3. External sorting – Example.

The number of block transfers required for the external sort merge is computed as given below:

- Let b_r denote the number of blocks containing records of relation r . The first stage reads every block of the relation and writes them out again, giving a total of $2b_r$ disk accesses. The initial number of runs is $[b_r/M]$. Since the number of runs decreases by a factor of $M-1$ in each merge pass, the total number of merge passes required is $\lceil \log M - 1(b_r/M) \rceil$. Each of these passes reads every block of the relation once and writes it out once, with two exceptions.
 - The final pass can produce the sorted output without writing its result to disk.
 - There may be runs that are not read in or written out during a pass.
- The total number of disk accesses for external sorting of the relation is $b_r(2[\log M - 1(b_r/M)] + 1)$.

13.6 JOIN OPERATION

This section deals with different join algorithms of relations and analyzes the respective cost of the join algorithms.

Consider the expression

student \bowtie takes

The following information are considered about the two relations:

Number of records of student, $n_{\text{student}} = 5,000$

Number of blocks of student, $b_{\text{student}} = 100$

Number of records of takes, $n_{\text{take}} = 10,000$

Number of blocks of takes, $b_{\text{take}} = 400$

13.6.1 Nested-Loop Join

```

for each tuple  $t_r$  in r do begin
    for each tuple  $t_s$  in s do begin
        test pair  $(t_r, t_s)$  to see if they satisfy the join condition  $\theta$ 
        if they do, add  $t_r \cdot t_s$  to the result
    end
end

```

Above code shows a simple algorithm to compute the theta join of two relations r and s. This algorithm is called the nested-loop join algorithm, since it basically consists of a pair of nested for loops. Relation r is called the outer relation and relation s the inner relation of the join, since the loop for r encloses the loop for s.

The algorithm uses the notation $t_r \cdot t_s$, where t_r and t_s are tuples; $t_r \cdot t_s$ denotes the tuple constructed by concatenating the attribute values of tuples t_r and t_s . The nested-loop join algorithm requires no indices. The nested-loop join algorithm is expensive, since it examines every pair of tuples in the two relations.

The join can be extended to compute the natural join. The natural join can be expressed as a theta join followed by elimination of repeated attributes by a projection. The only change required is an extra step of deleting repeated attributes from the tuple $t_r \cdot t_s$, before adding it to the result.

Consider the cost of the nested-loop join algorithm. The number of pairs of tuples to be considered is $n_r * n_s$, where n_r denotes the number of tuples in r and n_s denotes the number of tuples in s. For each record in r, we have to perform a complete scan on s.

In the worst case, the buffer can hold only one block of each relation, and a total of $n_r * b_s + b_r$ block accesses would be required, where b_r and b_s denote the number of blocks containing tuples of r and s respectively. In the best case, there is enough space for both relations to fit simultaneously in memory, so each block would have to be read only once; hence, only $b_r + b_s$ block accesses would be required. If one of the relations fits entirely in main memory, it is beneficial to use that relation as the inner relation, since the inner relation would then be read only once.

Now consider the natural join of student and takes. Consider there are no indices for both the relation. Student relation is considered to be the outer relation and takes relation is considered to be the inner relation in the join. Then totally $5000 * 10000 = 50 * 10^6$ pairs of tuples have to be examined. In the worst case, the number of block accesses is $5000 * 400 + 100 = 2,000,100$. In the best-case scenario, both relations are read only once for computation. This computation requires at most $100+400 = 500$ block accesses. If the relation takes has been used for outer loop and student has been taken as the inner loop then, then the worst-case cost of final strategy would be $10000 * 100 + 400 = 1,000,400$ block transfers.

13.6.2 Block Nested-Loop Join

If the buffer is too small to hold either relation entirely in memory, still block access can be saved if the relations are processed on a per-block basis, rather than on a per-tuple basis.

```

for each block  $B_r$  of  $r$  do begin
  for each block  $B_s$  of  $s$  do begin
    for each tuple  $t_r$  in  $B_r$  do begin
      for each tuple  $t_s$  in  $B_s$  do begin
        Test pair  $(t_r, t_s)$  to see if they satisfy the join condition
        if they do, add  $t_r . t_s$  to the result
      End
    End
  End
End

```

Above code shows a block nested-loop join, which is a modification of the nested-loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, every tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples. As before, all pairs of tuples that satisfy the join condition are added to the result.

The primary difference in cost between the block nested-loop join and the basic nested-loop join is that, in the worst case, each block in the inner relation s is read only once for each block in the outer relation, instead of once for each tuple in the outer relation. Thus, in the worst case, there will be a total of $b_r * b_s + b_r$ block accesses, where b_r and b_s denote the number of blocks containing records of r and s respectively. In the best case, there will be $b_r + b_s$ block accesses.

Now let us consider the example relations, student and takes for computing the block nested-loop join algorithm. In the worst case each block of takes should be read once for each block of student. Thus, in the worst case, a total of $100 * 400 + 100 = 40,100$ block accesses are required. This cost is a significant improvement over the $5000 * 400 + 100 = 2,000,100$ block accesses needed in the worst case for the basic nested-loop join. The number of block accesses in the best case remains the same, i.e., $100 + 400 = 500$.

The performance of the nested-loop and block nested-loop procedures can be further improved:

- If the join attributes in a natural join or an equi-join form a key on the inner relation, then for each outer relation tuple the inner loop can terminate as soon as the first match is found.
- In the block nested-loop algorithm, instead of using disk blocks as the blocking unit for the outer relation, the biggest size that can fit in memory can be used, while leaving enough space for the buffers of the inner relation and the output.
- Reuse the blocks stored in the buffer, which reduces the number of disk accesses needed.
- If an index is available on the inner loop's join attribute, replace file scans with more efficient index lookups.

13.6.3 Indexed Nested-Loop Join

In a nested-loop join, if an index is available on the inner loop's join attribute, index lookups can replace file scans. For each tuple t_r in the outer relation r , the index is used to look up tuples in s that will satisfy the join condition with tuple t_r . This join method is called an indexed nested-loop join. It can be used with existing indices, as well as with temporary indices created for evaluating the join.

For example, consider student \bowtie takes. Suppose if there is a student tuple with ID “00128”. Then, the relevant tuple’s in takes are those that satisfy the selection “ID=00128”.

The cost of an indexed nested-loop join can be computed as follows. For each tuple in the outer relation r , a lookup is performed on the index for s , and the relevant tuples are retrieved. In the worst case, there is space in the buffer for only one page of r and one page of the index. Then, b_r disk accesses are needed to read relation r , where b_r denotes the number of blocks containing records of r . For each tuple in r , an index lookup on s is performed. Then, the cost of the join can be computed as $b_r + n_r * c$, where n_r is the number of records in relation r , and c is the cost of a single selection on s using the join condition.

For example, consider an indexed nested-loop join of student \bowtie takes, with student as the outer relation. Suppose takes also has a primary B^+ -tree index on the join attribute ID, which contains 20 entries on an average in each index node. Since takes has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data. Since n_{student} is 5000, the total cost is $100 + 5000 * 5 = 25,100$ disk accesses. This cost is lower than the 40,100 accesses needed for a block nested-loop join.

13.6.4 Merge Join or Sort-merge Join

The merge join algorithm can be used to compute natural joins and equi-joins. Let $r(R)$ and $s(S)$ be the relations whose natural join is to be computed, and let $R \cap S$ denote their common attributes. Suppose that both relations are sorted on the attributes $R \cap S$. Then, their join can be computed by a process much like the merge stage in the merge sort algorithm. Merge join algorithm is given below:

```

 $pr :=$  address of first tuple of  $r$ ;
 $ps :=$  address of first tuple of  $s$ ;
while ( $ps \neq \text{null}$  and  $pr \neq \text{null}$ ) do
    begin
         $t_s :=$  tuple to which  $ps$  points;
         $S_s := \{t_s\}$ ;
        set  $ps$  to point to next tuple of  $s$ ;
         $done := false$ ;
        while (not  $done$  and  $ps \neq \text{null}$ ) do
            begin
                 $t_s' :=$  tuple to which  $ps$  points;
                if ( $t_s'[JoinAttrs] = t_s[JoinAttrs]$ )
                    then begin
                         $S_s := S_s \cup \{t_s'\}$ ;
                        set  $ps$  to point to next tuple of  $s$ ;
                    end
                else  $done := true$ ;
            end
    
```

```

 $t_r :=$  tuple to which  $pr$  points;
while ( $pr \neq \text{null}$  and  $t_r[\text{JoinAttrs}] < t_s[\text{JoinAttrs}]$ ) do
  begin
    set  $pr$  to point to next tuple of  $r$ ;
     $t_r :=$  tuple to which  $pr$  points;
  end
while ( $pr \neq \text{null}$  and  $t_r[\text{JoinAttrs}] = t_s[\text{JoinAttrs}]$ ) do
  begin
    for each  $t_s$  in  $S_s$  do
      begin
        add  $t_s \bowtie t_r$  to result ;
      end
    set  $pr$  to point to next tuple of  $r$ ;
     $t_r :=$  tuple to which  $pr$  points;
  end
end

```

In the algorithm, JoinAttrs refers to the attributes in $R \cap S$, and $t_r \bowtie t_s$, where t_r and t_s are tuples that have the same values for JoinAttrs, denotes the concatenation of the attributes of the tuples, followed by projecting out repeated attributes.

The merge join algorithm associates one pointer with each relation. These pointers point initially to the first tuple of the respective relations. As the algorithm proceeds, the pointers move through the relation. A group of tuples of one relation with the same value on the join attributes is read into S_s . The algorithm specified above requires every set of tuples S_s to fit in main memory. Then, the corresponding tuples of the other relation are read in, and are processed as they are read.

13.6.5 Hash Join

Like the merge join algorithm, the hash join algorithm can be used to implement natural joins and equi-joins. In the hash join algorithm, a hash function h is used to partition tuples of both relations. The basic idea is to partition the tuples of each of the relations into sets that have the same hash value on the join attributes. It is assumed that

- h is a hash function mapping JoinAttrs values to $\{0, 1, \dots, n_h\}$, where JoinAttrs denotes the common attributes of r and s used in the natural join.
- $H_{r0}, H_{r1}, \dots, H_{rn}$ denote partitions of r tuples, each initially empty. Each tuple $t_r \in r$ is put in partition H_{ri} , where $i = h(t_r[\text{JoinAttrs}])$.
- $H_{s0}, H_{s1}, \dots, H_{sn}$ denote partitions of s tuples, each initially empty. Each tuple $t_s \in s$ is put in partition H_{si} , where $i = h(t_s[\text{JoinAttrs}])$.

The hash function h should have the properties of randomness and uniformity. Figure 13.4 depicts the partitioning of the relations. The idea behind the hash join algorithm is that, if an r tuple and an s tuple satisfy the join condition, then the tuples will have the same value for the join

attributes. If that value is hashed to some value i , the r tuple has to be in H_{ri} and the s tuple in H_{si} . Therefore, r tuples in H_{ri} need only to be compared with s tuples in H_{si} ; it need not be compared with s tuples in any other partition.

For example, if d is a tuple in student, c a tuple in takes, and h a hash function on the ID attributes of the tuples, then d and c must be tested only if $h(c) = h(d)$. If $h(c) \neq h(d)$, then c and d must have different values for ID. However, if $h(c) = h(d)$, then the values of c and d must be tested to check whether the values in their join attributes are the same, since it is possible that c and d have different IDs that have the same hash value.

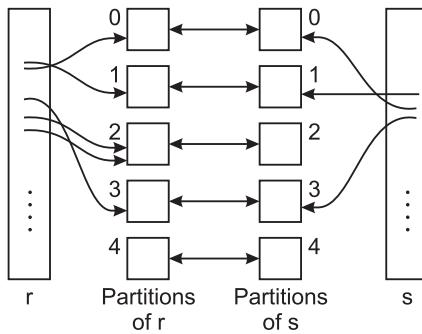


Fig. 13.4. Hash partitioning of relations.

```

/* Partition s */
for each tuple  $t_s$  in  $s$  do begin
     $i := h(t_s[JoinAttrs]);$ 
     $H_{si} := H_{si} \cup \{t_s\};$ 
end
/* Partition r */
for each tuple  $t_r$  in  $r$  do begin
     $i := h(t_r[JoinAttrs]);$ 
     $H_{ri} := H_{ri} \cup \{t_r\};$ 
end
/* Perform join on each partition */
for  $i := 0$  to  $n_h$  do begin
    read  $H_{si}$  and build an in-memory hash index on it
    for each tuple  $t_r$  in  $H_{ri}$  do begin
        probe the hash index on  $H_{si}$  to locate all tuples  $t_s$ 
        such that  $t_s[JoinAttrs] = t_r[JoinAttrs]$ 
        for each matching tuple  $t_s$  in  $H_{si}$  do begin
            add  $t_r \bowtie t_s$  to the result
        end
    end
end
end

```

Above algorithm shows the details of the hash join algorithm to compute the natural join of relations r and s . As in the merge join algorithm, $t_r \bowtie t_s$ denotes the concatenation of the attributes of tuples t_r and t_s , followed by projecting out repeated attributes. After the partitioning of the relations, the rest of the hash join code performs a separate indexed nested-loop join on each of the partition pairs i , for $i = 0, \dots, n_h$. To do so, it first builds a hash index on each H_{si} , and then probes with tuples from H_{ri} . The relation s is the build input, and r is the probe input. The hash index on H_{si} is built in memory, so there is no need to access the disk to retrieve the tuples. The hash function used to build this hash index is different from the hash function h used earlier, but is still applied to only the join attributes. In the course of the indexed nested-loop join, the system uses this hash index to retrieve records that will match records in the probe input.

The build and probe phases require only a single pass through both the build and probe inputs. It is straightforward to extend the hash join algorithm to compute general equi-joins. The value n_h must be chosen to be large enough such that, for each i , the tuples in the partition H_{si} of the build relation, along with the hash index on the partition, will fit in memory. It is not necessary for the partitions of the probe relation to fit in memory. Clearly, it is best to use the smaller input relation as the build relation.

13.7 DATABASE TUNING

Database tuning describes a group of activities used to optimize and homogenize the performance of a database. It usually overlaps with query tuning, but refers to design of the database files, selection of the database management system, Operating system and CPU the DBMS runs on.

The goal is to maximize use of system resources to perform work as efficiently and rapidly as possible. Most systems are designed to manage work efficiently, but it is possible to greatly improve the performance by customizing setting and the configuration for the database and the DBMS being tuned.

DBMS tuning refers to tuning of the DBMS and the configuration of the memory and processing resources of the computer running the DBMS. This is typically done through configuring the DBMS, but the resources involved are shared with the host system.

Tuning the database management system can involve setting the recovery interval, assigning parallelism and network protocols used to communicate with database consumers.

Memory is allocated for data, execution plans, procedure cache and work space. It is much faster to access data in memory than data on storage, so maintaining a sizable cache of data makes activities perform faster. The same consideration is given to work space. Caching execution and procedures means that they are reused instead of recompiled when needed. It is important to take as much memory as possible, while leaving enough for other processes and the OS to use without excessive paging of memory to storage.

Processing resources are sometimes assigned to specific activities to improve concurrency. On a server with eight processors, size could be reserved for DBMS to maximize available processing resources for the database.

13.7.1 Types of Database Tuning

Different kinds of database tuning are:

1. Tuning indexes

- The initial choice of indexes may be refined for several reasons. The simplest reason is that the observed workload reveals that some queries and updates considered important in the initial workload specification are not very frequent.
- The observed workload may also identify some new queries and updates that are important. The initial choice of indexes has to be reviewed for this new information.
- Some of the original indexes may be dropped and new ones added.
- Some indexes may be reorganized. For example, a static index. Drop such index and rebuild it, which in turn improves access times.

For a dynamic structure such as B+- tree, if the implementation does not merge pages on deletes, space occupancy can decrease, which in turn makes size of index larger than necessary and could increase the height and therefore the access time. Rebuilding the index should be considered in such cases.

Extensive updates to a clustered index might also lead to overflow pages being allotted, thereby decreasing the degree of clustering. Again rebuilding the index is advantageous.

2. Tuning the Conceptual Schema

- Sometimes the relational schema designed for the database does not meet performance objectives for the given workload. In such cases, it is necessary to redesign the conceptual schema.
- While designing the conceptual schema consider the queries and updates in workload, issues of redundancy that motivate normalization.
- Several options that must be considered while tuning the conceptual schema are
 - Use 3NF design instead of a BCNF design
 - If there are two ways to decompose a given schema into 3NF or the BCNF, choice should be guided by workload.
 - Sometimes it is necessary to decompose a relation that is already in BCNF.
 - In some situations relations are denormalized. In most cases it is chosen from a larger relation with original relation, even though it suffers from some redundancy problem. Alternatively it is chosen to add some fields to certain relations to speed up some important queries, even if this leads to a redundant storage of some information.
 - Use normalization using decomposition technique. For decomposing the given relation either use vertical partitioning or horizontal partitioning, which would lead to have two relations with identical schemas.

3. Tuning Queries and Views

- Sometimes a query runs slower than expected. In such cases, examine the query carefully to find the problem. Rewriting of query with some index tuning may fix the problem. Similar tuning may be called for if queries on some view run slower than expected.

- While tuning a query, first thing to check is to verify whether the system is using the expected plan or not. Perhaps the system is not finding the best plan for a variety of reasons. If the optimizer is not smart enough to find the best plan, some systems allow users to guide the choice of a plan by providing hints to the optimizer, for example, users might be able to force the use of a particular index or choose the join order. A user who wishes to guide optimization in this manner should have a thorough understanding of both optimization and the capabilities of the given DBMS.

13.8 EVALUATION OF EXPRESSIONS

The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order. The result of each evaluation is materialized in a temporary relation for subsequent use. A disadvantage to this approach is the need to construct the temporary relations, which must be written to disk. An alternative approach is to evaluate several operations simultaneously in a pipeline, with the results of one operation passed on to the next, without the need to store a temporary relation.

13.8.1 Materialization

Consider the expression

$$\Pi_{\text{name}} (\sigma_{\text{building} = \text{"Watson"}} (\text{department}) \bowtie \text{instructor})$$

Diagrammatic representation

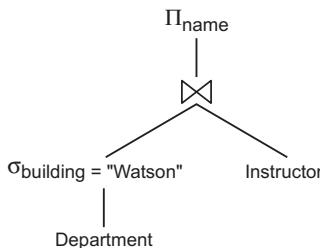


Fig. 13.5. Pictorial representation of an expression.

If materialization approach is applied, evaluation is applied from the lowest-level operations in the expression. In the above example, there is only one such operation; the selection operation on department. The inputs to the lowest-level operations are relations in the database. These operations are executed by using algorithms and the results are stored in temporary relations. These temporary relations are used to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database. In the above example, the inputs to the join are the instructor relation and the temporary relation created by the selection on department. The join can now be evaluated, creating another temporary relation.

By repeating the process, eventually the operation at the root of the tree is evaluated, giving the final result of the expression. In the above example, the final result is got by executing the projection operation at the root of the tree, using as input the temporary relation created by the join.

Evaluation described in this session is called materialized evaluation, since the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level

operations. The cost of a materialized evaluation is not simply the sum of the costs of the operations involved. To compute the cost of evaluating an expression, it is needed to add the costs of all the operations, as well as the cost of writing the intermediate results to disk.

13.8.2 Pipelining

Query-evaluation efficiency can be improved by reducing the number of temporary files that are produced. This reduction can be achieved by combining several relational operations into a pipeline of operations, in which the results of one operation are passed along to the next operation in the pipeline such evaluation is called pipelined evaluation. Combining operations into a pipeline eliminates the cost of reading and writing temporary relations. For example, consider the expression $(\Pi_{a1,a2}(r \bowtie s))$. If materialization were applied, evaluation would involve creating a temporary relation to hold the result of the join, and then reading back in the result to perform the projection. These operations can be combined: When the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. By combining the join and the projection, creation of intermediate result can be avoided and instead creates the final result directly.

13.8.3 Implementation of Pipelining

Pipeline can be implemented by constructing a single, complex operation that combines the operations that constitute the pipeline. Although this approach may be feasible for various frequently occurring situations, it is desirable in general to reuse the code for individual operations in the construction of a pipeline. Therefore, each operation in the pipeline is modeled as a separate process or thread within the system, which takes a stream of tuples from its pipelined inputs, and generates a stream of tuples for its output. For each pair of adjacent operations in the pipeline, the system creates a buffer to hold tuples being passed from one operation to the next.

In the example of Figure 13.5, all three operations can be placed in a pipeline, which passes the results of the selection to the join as they are generated. In turn, it passes the results of the join to the projection as they are generated. The memory requirements are low, since results of an operation are not stored for long. However, as a result of pipelining, the inputs to the operations are not available all at once for processing.

Pipelines can be executed in either of two ways:

1. Demand driven
2. Producer driven

1. Demand-driven Pipeline

In a demand-driven pipeline, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple to be returned, and then returns that tuple. If the inputs of the operation are not pipelined, the next tuple(s) to be returned can be computed from the input relations, while the system keeps track of what has been returned so far. If it has some pipelined inputs, the operation also makes requests for tuples from its pipelined inputs. Using the tuples received from its pipelined inputs, the operation computes tuples for its output, and passes them up to its parent.

2. Producer-driven Pipeline

In a producer-driven pipeline, operations do not wait for requests to produce tuples, but instead generate the tuples eagerly. Each operation at the bottom of a pipeline continually generates output tuples, and puts them in its output buffer, until the buffer is full. An operation at any other level of a pipeline generates output tuples when it gets input tuples from lower down in the pipeline, until its output buffer is full. Once the operation uses a tuple from a pipelined input, it removes the tuple from its input buffer. In either case, once the output buffer is full, the operation waits until its parent operation removes tuples from the buffer, so that the buffer has space for more tuples. At this point, the operation generates more tuples, until the buffer is full again. The operation repeats this process until all the output tuples have been generated. It is necessary for the system to switch between operations only when an output buffer is full, or an input buffer is empty and more input tuples are needed to generate any more output tuples.

REVIEW QUESTIONS

1. With suitable diagram explain the various steps involved in query processing.
2. Explain the sorting technique in detail.
3. Write short notes on query optimization.
4. How cost of query is measured? Explain.
5. What is meant by complex selection? Explain its implementation with suitable example.
6. What is the need for database tuning?
7. Explain different types of database tuning methods.
8. Explain how pipelining improves query evaluation efficiency.
9. Explain the algorithm for hash join.



Chapter 14

PL/SQL

14.1 INTRODUCTION

PL/SQL stands for Procedural Language/Structured Query Language. PL/SQL is a combination of SQL along with the procedural features of programming languages.

PL/SQL has the following features:

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports developing web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages:

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. Dynamic SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.

- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for Developing Web Applications and Server Pages.

14.1.1 Structure if PL/SQL

PL/SQL offers all features of advanced programming language such as portability, security, data encapsulation, information hiding, etc. A PL/SQL program may consist of more than one SQL statements, while execution of a PL/SQL program makes only one call to Oracle engine, thus it helps in reducing the database overheads. With PL/SQL, one can use the SQL statements together with the control structures for data manipulation. Besides this, user can define his/her own error messages to display. Thus we can say that PL/SQL combines the data manipulation power of SQL with data processing power of procedural language. PL/SQL is a block structured language. This means a PL/SQL program is made up of blocks, where block is a smallest piece of PL/SQL code having logically related statements and declarations. A block consists of three sections namely:

- (i) Declare
- (ii) Begin
- (iii) Exception followed by an End statement.

Declare Section: Declare section declares the variables, constants, processes, functions, etc., to be used in the other parts of program. It is an optional section.

Begin Section: It is the executable section. It consists of a set of SQL and PL/SQL statements, which is executed when PL/SQL block runs. It is a compulsory section.

Exception Section: This section handles the errors, which occurs during execution of the PL/SQL block. This section allows the user to define error messages. This section executes only when an error occurs. It is an optional section.

End Section: This section indicates the end of PL/SQL block.

Every PL/SQL program must consist of at least one block, which may consist of any number of nested sub-blocks. Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using BEGIN and END.

Basic structure of a PL/SQL block:

```

DECLARE
    <declarations section>
BEGIN
    <executable command(s)>
EXCEPTION
    <exception handling>
END;

```

Example:



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window contains the following PL/SQL code and its output:

```

SQL> DECLARE
  2   message  varchar2(20):= 'Hello World!!!!!!';
  3   BEGIN
  4     dbms_output.put_line(message);
  5   END;
  6 /
Hello World!!!!!!
PL/SQL procedure successfully completed.

SQL> |

```

Fig. 14.1. Hello World – Example.

14.2 PL/SQL LANGUAGE ELEMENTS

Like all programming languages PL/SQL also has specific character sets, operators, indicators, punctuations, identifiers, comments, etc.

Character Set

A PL/SQL program consists of text having specific set of characters. Character set may include the following characters:

- Alphabets, both in upper case [A–Z] and lower case [a–z]
- Numeric digits [0–9]
- Special characters () + – * / <> = ! ~ ^ ; : . ‘ @ % , “ # \$ & { } ? []
- Blank spaces, tabs, and carriage returns.

PL/SQL is not case sensitive, so lowercase letters are equivalent to corresponding uppercase letters except within string and character literals.

Lexical Units

A line of PL/SQL program contains groups of characters known as lexical units, which can be classified as follows:

- (i) Delimiters
- (ii) Identifiers
- (iii) Literals
- (iv) Comments

(i) Delimiters

A *delimiter* is a simple or compound symbol that has a special meaning to PL/SQL. Simple symbol consists of one character, while compound symbol consists of more than one character. Following is the list of delimiters in PL/SQL:

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, '=, ~=, ^=	Different versions of NOT EQUAL

(ii) Identifiers

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. Any other characters like hyphens, slashes, blank spaces, etc. are illegal. Identifiers are not case-sensitive. Reserved keywords cannot be used as an identifier.

Example: A, A1, Share\$price, e_mail, phone# – number sign is permitted

The following identifiers are illegal:

mine&yours – ampersand is illegal

debit-amount – hyphen is illegal

on/off – slash is illegal

user id – space is illegal

However, PL/SQL allows space, slash, hyphen, etc. except double quotes if the identifier is enclosed within double quotes. An identifier cannot be a reserve word, i.e., the words that have special meaning for PL/SQL. For example, DECLARE cannot be used as identifier since it is a keyword.

(iii) Literals

A literal is an explicitly defined character, string, numeric, or Boolean value, which is not represented by an identifier.

Numeric Literals: A numeric literal is an integer or a real value. An integer literal may be a positive, negative, or unsigned whole number without a decimal point. A real literal is a positive, negative, or unsigned whole or fractional number with a decimal point. Besides integer and real literals, numeric literals can also contain exponential.

E.g. 100, 2, -19, 3.14, 2.0E-3

Character Literals: A character literal is an individual character enclosed by single quotes. Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols. PL/SQL is case sensitive within character literals. For example, PL/SQL considers the literals “A” and “a” to be different.

E.g. “A”, “@”, “3”

String Literals: A character string can be represented by an identifier or explicitly written as a string literal. A string literal is enclosed within single quotes and may consist of one or more characters.

E.g. “Good Morning!”, “Data Base”, “10000”

All string literals are of character data type. PL/SQL is case sensitive within string literals. For example, PL/SQL considers the following literals to be different:

- “DATA”
- “data”

Boolean Literals: Boolean literals are the predefined values TRUE, FALSE, and NULL. Boolean literals are values, not strings. For example a condition: if ($y = 20$) is TRUE only for the value of y equal to 20, for any other value of y it is FALSE and for no value of y it is NULL.

(iv) Comments

Comments are used in the PL/SQL program to improve the readability and understandability of a program. A comment can appear anywhere in the program code. The compiler ignores comments. A PL/SQL comment may be a single-line or multiline.

Single-Line Comments: Single-line comments begin with a double hyphen (--) anywhere on a line and extend to the end of the line.

Example: --start program

Multiline Comments: Multiline comments begin with a slash-asterisk /*) and end with an asterisk slash (*/), and can span multiple lines.

Example: /* This is an example of multiline comments in PL/SQL */

14.3 PL/SQL SCALAR DATA TYPES

PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and valid range of values.

Category	Description
Scalar	Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.
Large Object (LOB)	Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
Composite	Data items that have internal components that can be accessed individually. For example, collections and records.
Reference	Pointers to other data items.

PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories:

Date Type	Description
Numeric	Numeric values on which arithmetic operations are performed.
Character	Alphanumeric values that represent single characters or strings of characters.
Boolean	Logical values on which logical operations are performed.
Datetime	Dates and times.

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. Subtypes provide compatibility.

PL/SQL Numeric Data Types and Subtypes

PL/SQL pre-defined numeric data types and their sub-types are as follows:

Data Type	Description
PLS_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_FLOAT	Single-precision IEEE 754-format floating-point number
BINARY_DOUBLE	Double-precision IEEE 754-format floating-point number
NUMBER(prec, scale)	Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0.
DEC(prec, scale)	ANSI specific fixed-point type with maximum precision of 38 decimal digits.
DECIMAL(prec, scale)	IBM specific fixed-point type with maximum precision of 38 decimal digits.
NUMERIC(pre, scale)	Floating type with maximum precision of 38 decimal digits.
DOUBLE PRECISION	ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
FLOAT	ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)

INT	ANSI specific integer type with maximum precision of 38 decimal digits
INTEGER	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
SMALLINT	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
REAL	Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

PL/SQL Character Data Types and Subtypes

PL/SQL pre-defined character data types and their sub-types are as follows:

Data Type	Description
CHAR	Fixed-length character string with maximum size of 32,767 bytes
VARCHAR2	Variable-length character string with maximum size of 32,767 bytes
RAW	Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
NCHAR	Fixed-length national character string with maximum size of 32,767 bytes
NVARCHAR2	Variable-length national character string with maximum size of 32,767 bytes
LONG	Variable-length character string with maximum size of 32,760 bytes
LONG RAW	Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
ROWID	Physical row identifier, the address of a row in an ordinary table
UROWID	Universal row identifier (physical, logical, or foreign row identifier)

PL/SQL Boolean Data Types

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL. However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in:

- SQL statements
- Built-in SQL functions
- PL/SQL functions invoked from SQL statements

PL/SQL Date Time and Interval Types

The DATE data type to store fixed-length date times, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD. The default date format is 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year, for example, 01-OCT-12. Each DATE includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field:

Field Name	Valid Date time Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable

PL/SQL Large Object (LOB) Data Types

Large object (LOB) data types refer large to data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types:

Data Type	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

PL/SQL User-Defined Subtypes

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values. PL/SQL predefines several subtypes in package STANDARD. For example, PL/SQL predefines the subtypes CHARACTER and INTEGER as follows:

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

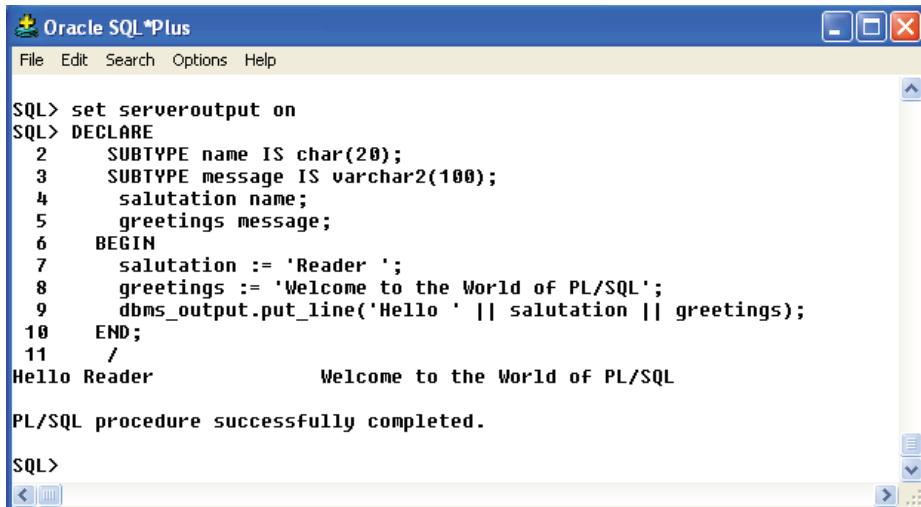
Users can define and use their own subtypes. The following program illustrates defining and using a user-defined subtype:

```

DECLARE
    SUBTYPE name IS char(20);
    SUBTYPE message IS varchar2(100);
        salutation name;
        greetings message;
BEGIN
    salutation := 'Reader';
    greetings := 'Welcome to the World of PL/SQL';
    dbms_output.put_line('Hello ' || salutation || greetings);
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



```

Oracle SQL*Plus
File Edit Search Options Help

SQL> set serveroutput on
SQL> DECLARE
  2      SUBTYPE name IS char(20);
  3      SUBTYPE message IS varchar2(100);
  4          salutation name;
  5          greetings message;
  6      BEGIN
  7          salutation := 'Reader';
  8          greetings := 'Welcome to the World of PL/SQL';
  9          dbms_output.put_line('Hello ' || salutation || greetings);
 10      END;
 11  /
Hello Reader           Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.

SQL>

```

Fig. 14.2. User defined Data type – Example.

NULLS in PL/SQL

PL/SQL NULL values represent missing or unknown data and they are not an integer, a character, or any other specific data type. Note that NULL is not the same as an empty data string or the null character value '\0'. A null can be assigned but it cannot be equated with anything, including itself.

14.4 VARIABLE DECLARATION IN PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When a variable is declared, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

Syntax:

variable_name [CONSTANT] datatype [NOTNULL][:=|DEFAULT initial_value]

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type.

Example:

```
sales number(10,2);
pi CONSTANT doubleprecision:=3.1415;
name varchar2(25);
address varchar2(100);
```

When a size is provided with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example:

```
sales number(10,2);
name varchar2(25);
address varchar2(100);
```

14.4.1 Initializing Variables in PL/SQL

Whenever a variable is declared, PL/SQL assigns it a default value of NULL. If the variable should be initialized with a value other than the NULL value, then during the declaration use one of the following:

- The **DEFAULT** keyword
- The **assignment operator**

For example:

```
counter binary_integer :=0;
greetings varchar2(20)DEFAULT 'Have a Good Day';
```

NOT NULL constraints can be used to restrict the variable from using NULL values. It is a good programming practice to initialize variables properly otherwise, sometimes program would produce unexpected result. Try the following example which makes use of various types of variables:

```
DECLARE
    a integer :=10;
    b integer :=20;
    c integer;
    f real;
BEGIN
    c := a + b;
    dbms_output.put_line('Value of c: '|| c);
    f :=70.0/3.0;
    dbms_output.put_line('Value of f: '|| f);
END;
/
```

When the above code is executed, it produces the following result:

Fig. 14.3. Initializing Variables – Example.

14.4.2 Variable Scope in PL/SQL

PL/SQL allows the nesting of Blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer Block, it is also accessible to all nested inner Blocks.

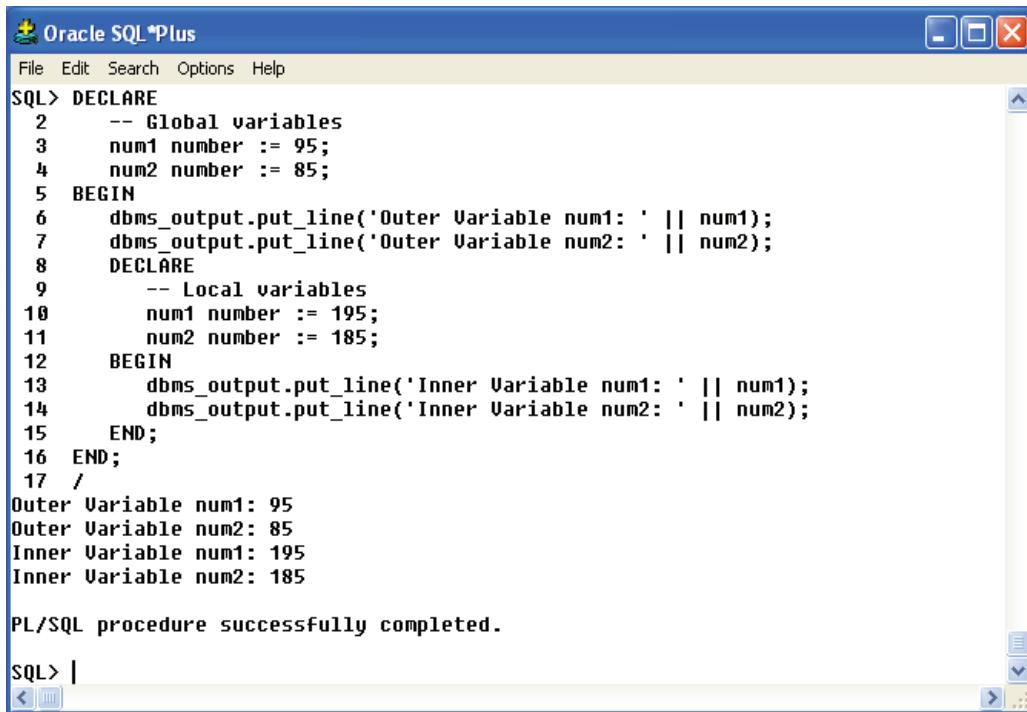
There are two types of variable scope:

- **Local variables:** Variables declared in an inner block and not accessible to outer blocks.
 - **Global variables:** Variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form:

```
DECLARE
-- Global variables
    num1 number :=95;
    num2 number :=85;
BEGIN
    dbms_output.put_line('Outer Variable num1: ''|| num1);
    dbms_output.put_line('Outer Variable num2: ''|| num2);
DECLARE
-- Local variables
    num1 number :=195;
    num2 number :=185;
BEGIN
    dbms_output.put_line('Inner Variable num1: ''|| num1);
    dbms_output.put_line('Inner Variable num2: ''|| num2);
END;
END;/
```

When the above code is executed, it produces the following result:



```

SQL> DECLARE
  2      -- Global variables
  3      num1 number := 95;
  4      num2 number := 85;
  5  BEGIN
  6      dbms_output.put_line('Outer Variable num1: ' || num1);
  7      dbms_output.put_line('Outer Variable num2: ' || num2);
  8      DECLARE
  9          -- Local variables
 10         num1 number := 195;
 11         num2 number := 185;
 12     BEGIN
 13         dbms_output.put_line('Inner Variable num1: ' || num1);
 14         dbms_output.put_line('Inner Variable num2: ' || num2);
 15     END;
 16 END;
 17 /
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.

SQL> |

```

Fig. 14.4. Program explaining scope of variables.

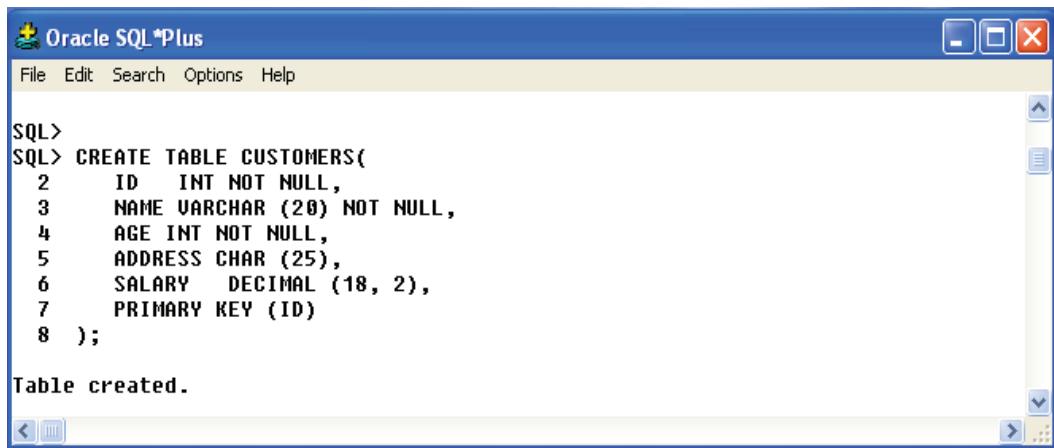
14.4.3 Assigning SQL Query Results to PL/SQL Variables

The SELECT INTO statement of SQL is used to assign values to PL/SQL variables. For each item in the SELECT list, there must be a corresponding, type-compatible variable in the INTO list. The following example illustrates the concept: Let us create a table named CUSTOMERS:

```

CREATE TABLE CUSTOMERS(
    ID INT NOTNULL,
    NAME VARCHAR (20)NOTNULL,
    AGE INT NOTNULL,
    ADDRESS CHAR (25),
    SALARY DECIMAL (18,2),
    PRIMARYKEY(ID)
);

```



The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following SQL code:

```
SQL> CREATE TABLE CUSTOMERS(
 2   ID INT NOT NULL,
 3   NAME VARCHAR (20) NOT NULL,
 4   AGE INT NOT NULL,
 5   ADDRESS CHAR (25),
 6   SALARY DECIMAL (18, 2),
 7   PRIMARY KEY (ID)
 8 );
```

Below the code, the message "Table created." is displayed. The bottom of the window shows standard Windows-style scroll bars.

Fig. 14.5. Table Creation.

Next, let us insert some values in the table:

```
INSERTINTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES(1,'Ramesh',32,'Ahmedabad',2000.00);
INSERTINTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES(2,'Khilan',25,'Delhi',1500.00);
INSERTINTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES(3,'kaushik',23,'Kota',2000.00);
INSERTINTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES(4,'Chaitali',25,'Mumbai',6500.00);
INSERTINTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES(5,'Hardik',27,'Bhopal',8500.00);
INSERTINTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES(6,'Komal',22,'MP',4500.00);
```

The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The menu bar includes File, Edit, Search, Options, and Help. The main window displays the following SQL session:

```

SQL> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
  2  VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
1 row created.

SQL> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
  2  VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
1 row created.

SQL> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
  2  VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
1 row created.

SQL> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
  2  VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );
1 row created.

SQL> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
  2  VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );
1 row created.

SQL> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
  2  VALUES (6, 'Komal', 22, 'MP', 4500.00 );
1 row created.

SQL> select * from customers;

```

The output of the SELECT statement is displayed as a table:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500

6 rows selected.

Fig. 14.6. Inserting values into customer table.

The following program assigns values from the above table to PL/SQL variables using the SELECT INTO clause of SQL:

```

DECLARE
  c_id customers.id%type :=1;
  c_name customers.name%type;
  c_addr customers.address%type;
  c_sal customers.salary%type;
BEGIN
  SELECT name, address, salary INTO c_name, c_addr, c_sal

```

```

FROM customers
WHERE id = c_id;

    dbms_output.put_line
('Customer ''||c_name||' from ''|| c_addr ||' earns ''|| c_sal);
END;
/

```

When the above code is executed, it produces the following result:

```

SQL> DECLARE
  2      c_id customers.id%type := 1;
  3      c_name  customers.name%type;
  4      c_addr  customers.address%type;
  5      c_sal   customers.salary%type;
  6  BEGIN
  7      SELECT name, address, salary INTO c_name, c_addr, c_sal
  8      FROM customers
  9      WHERE id = c_id;
 10
 11      dbms_output.put_line
 12      ('Customer ' ||c_name|| ' from ' || c_addr || ' earns ' || c_sal);
 13  END;
 14 /
Customer Ramesh from Ahmedabad          earns 2000
PL/SQL procedure successfully completed.

SQL>

```

Fig. 14.7. Assigning SQL Query Results to PL/SQL Variables.

14.4.4 Declaring a Constant

A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the NOT NULL constraint. A constant is declared using the CONSTANT keyword. It requires an initial value and does not allow that value to be changed. For example:

```
PI CONSTANT NUMBER := 3.141592654;
```

Example:

```

DECLARE
    -- constant declaration
    pi constant number := 3.141592654;
    -- other declarations
    radius number(5,2);
    dia number(5,2);

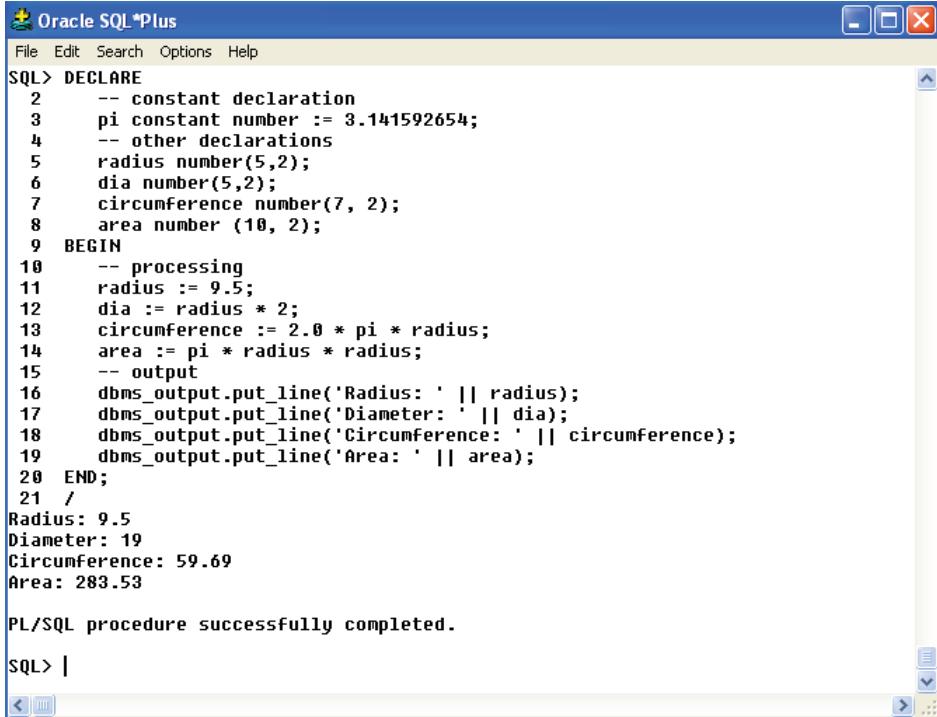
```

```

circumference number(7, 2);
area number (10, 2);
BEGIN
    -- processing
    radius := 9.5;
    dia := radius * 2;
    circumference := 2.0 * pi * radius;
    area := pi * radius * radius;
    -- output
    dbms_output.put_line('Radius: ' || radius);
    dbms_output.put_line('Diameter: ' || dia);
    dbms_output.put_line('Circumference: ' || circumference);
    dbms_output.put_line('Area: ' || area);
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



```

SQL> DECLARE
  2  -- constant declaration
  3  pi constant number := 3.141592654;
  4  -- other declarations
  5  radius number(5,2);
  6  dia number(5,2);
  7  circumference number(7, 2);
  8  area number (10, 2);
  9 BEGIN
10  -- processing
11  radius := 9.5;
12  dia := radius * 2;
13  circumference := 2.0 * pi * radius;
14  area := pi * radius * radius;
15  -- output
16  dbms_output.put_line('Radius: ' || radius);
17  dbms_output.put_line('Diameter: ' || dia);
18  dbms_output.put_line('Circumference: ' || circumference);
19  dbms_output.put_line('Area: ' || area);
20 END;
21 /
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53

PL/SQL procedure successfully completed.

SQL> |

```

Fig. 14.8. Declaring Constant Variables.

14.5 PL/SQL OPERATOR

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language provides the following types of operators:

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

14.5.1 Arithmetic Operators

PL/SQL provides the following arithmetic operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 5 then:

<i>Operator</i>	<i>Description</i>	<i>Example</i>
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

14.5.2 Relational Operators

Relational operators compare two expressions or values and return a Boolean result. PL/SQL provides the following relational operators. Assume variable A holds 10 and variable B holds 20, then:

<i>Operator</i>	<i>Description</i>	<i>Example</i>
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
~=		
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

14.5.3 Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either TRUE, FALSE OR NULL.

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that $x \geq a$ and $x \leq b$.	If $x = 10$ then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If $x = 'm'$ then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If $x = 'm'$, then 'x is null' returns Boolean false.

14.5.4 Logical Operators

Logical operators supported by PL/SQL are as follows. All these operators work on Boolean operands and produces Boolean results. Assume variable A holds true and variable B holds false, then:

Operator	Description	Example
And	Called logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
Or	Called logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
Not	Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.

14.5.5 PL/SQL Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator. Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

<i>Operator</i>	<i>Operation</i>
$**$	exponentiation
$+, -$	identity, negation
$\ast, /$	multiplication, division
$+, -, \parallel$	addition, subtraction, concatenation
$=, <, >, \leq, \geq, \neq, \sim, \approx,$ IS NULL, LIKE, BETWEEN, IN	comparison
NOT	logical negation
AND	conjunction
OR	inclusion

14.6 PL/SQL CONTROL STRUCTURE

Control structure is an essential part of any programming language. It controls the flow of process. Control structure is broadly divided into two categories:

1. Conditional control
2. Iterative control

14.6.1 Conditional Control

A conditional control structure tests a condition to find out whether it is true or false and accordingly executes the different blocks of SQL statements. Conditional control is generally performed by IF statement. There are three forms of IF statement.

- (i) IF-THEN
- (ii) IF-THEN-ELSE
- (iii) IF-THEN-ELSEIF
- (iv) CASE
- (v) SEARCHED CASE
- (vi) NESTED IF-THEN-ELSE

(i) IF-THEN

It is the simplest form of IF control statement, frequently used in decision making and changing the control flow of the program execution. The IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF. If the condition is TRUE, the statements get executed, and if the condition is FALSE or NULL, then the IF statement does nothing.

Syntax:

```
IF condition THEN
  Sequence of statement;
ENDIF;
```

Where condition is a Boolean or relational condition. Example of an IF-THEN statement is:

```
IF(a <=20)THEN
```

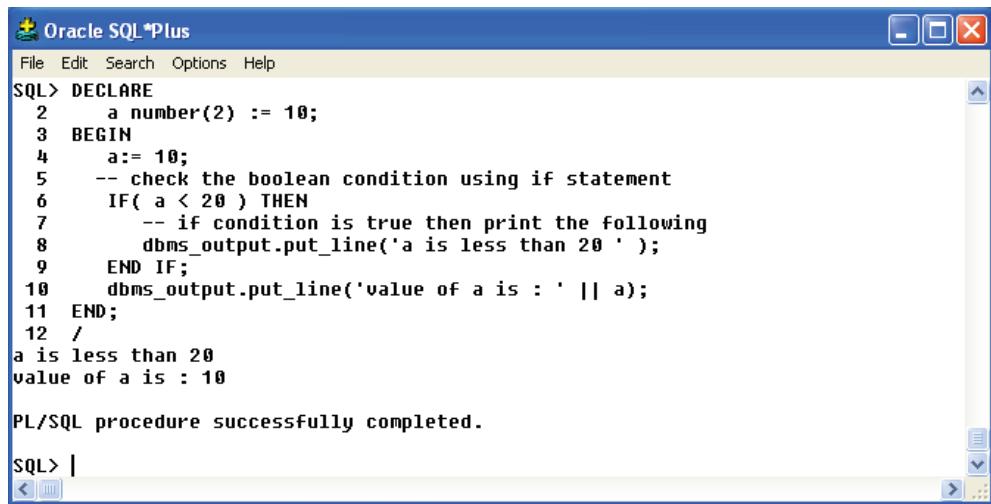
```
c:= c+1;
ENDIF;
```

If the Boolean expression condition evaluates to true then the block of code inside the IF statement will be executed. If Boolean expression evaluates to false then the first set of code after the end of the IF statement will be executed.

Example 1:

```
DECLARE
    a number(2):=10;
BEGIN
    a:=10;
    -- check the boolean condition using if statement
    IF( a <20)THEN
        -- if condition is true then print the following
        dbms_output.put_line('a is less than 20 ');
    ENDIF;
    dbms_output.put_line('value of a is : '|| a);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



```
Oracle SQL*Plus
File Edit Search Options Help
SQL> DECLARE
  2   a number(2) := 10;
  3 BEGIN
  4   a:= 10;
  5   -- check the boolean condition using if statement
  6   IF( a < 20 ) THEN
  7       -- if condition is true then print the following
  8       dbms_output.put_line('a is less than 20 ');
  9   END IF;
 10   dbms_output.put_line('value of a is : ' || a);
 11 END;
 12 /
a is less than 20
value of a is : 10

PL/SQL procedure successfully completed.

SQL> |
```

Fig. 14.9. IT-THEN statement – Example.

(ii) IF-THEN-ELSE

A sequence of IF-THEN statements can be followed by an optional sequence of ELSE statements, which execute when the condition is FALSE.

Syntax:

```
IF condition THEN
    Sequence of statements 1;
ELSE
    Sequence of statements 2;
END IF;
```

In the IF-THEN-ELSE statements, when the test condition is TRUE, the statement Sequence of statements 1 is executed and Sequence of statements 2 is skipped; when the test condition is FALSE, then Sequence of statements 1 is bypassed and statement Sequence of statements 2 is executed. For example:

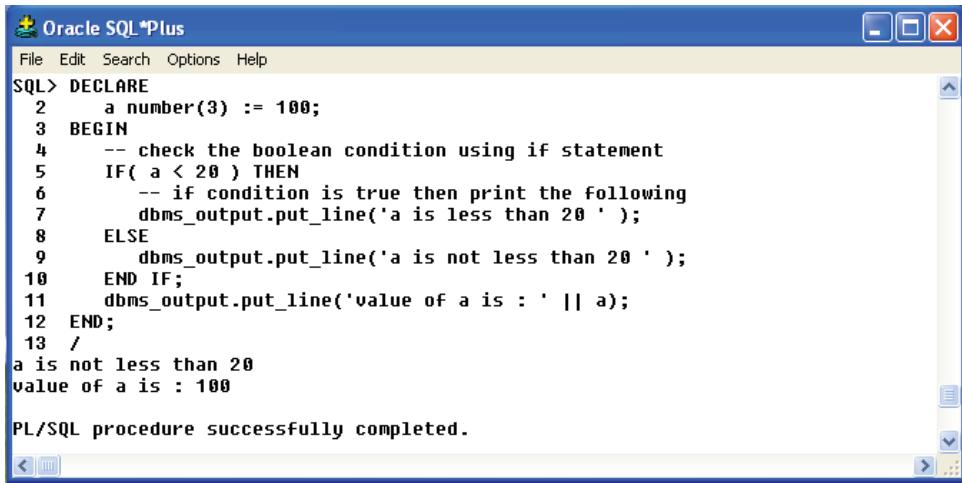
```
IF color = red THEN
    dbms_output.put_line('You have chosen a red car')
ELSE
    dbms_output.put_line('Please choose a color for your car');
END IF;
```

If the Boolean expression condition evaluates to true, then the if-then block of code will be executed, otherwise the else block of code will be executed.

Example:

```
DECLARE
    a number(3) := 100;
BEGIN
    -- check the boolean condition using if statement
    IF( a < 20 ) THEN
        -- if condition is true then print the following
        dbms_output.put_line('a is less than 20 ');
    ELSE
        dbms_output.put_line('a is not less than 20 ');
    END IF;
    dbms_output.put_line('value of a is : ' || a);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. In the command window, a PL/SQL block is being run:

```

SQL> DECLARE
  2      a number(3) := 100;
  3  BEGIN
  4      -- check the boolean condition using if statement
  5      IF( a < 20 ) THEN
  6          -- if condition is true then print the following
  7          dbms_output.put_line('a is less than 20 ' );
  8      ELSE
  9          dbms_output.put_line('a is not less than 20 ' );
 10     END IF;
 11     dbms_output.put_line('value of a is : ' || a);
 12 END;
 13 /
a is not less than 20
value of a is : 100

PL/SQL procedure successfully completed.

```

The output window shows the results of the execution.

Fig. 14.10. IT-THEN-ELSE statement – Example.

(iii) IT-THEN-ELSIF

The IF-THEN-ELSIF statement allows you to choose between several alternatives. An IF-THEN statement can be followed by an optional ELSIF...ELSE statement. The ELSIF clause lets you add additional conditions. When using IF-THEN-ELSIF statements there are few points to keep in mind.

- It's ELSIF, not ELSEIF
- An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
- An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

Syntax:

```

IF(boolean_expression 1)THEN
    Sequence of statement1; -- Executes when the boolean expression 1 is true
ELSIF( boolean_expression 2) THEN
    Sequence of statement2; -- Executes when the boolean expression 2 is true
ELSIF( boolean_expression 3) THEN
    Sequence of statement 3; -- Executes when the boolean expression 3 is true
ELSE
    Sequence of statement 4; -- executes when the none of the above condition is true
END IF;

```

Example:

```

DECLARE
    a number(3) := 100;

```

```

BEGIN
  IF ( a = 10 ) THEN
    dbms_output.put_line('Value of a is 10' );
  ELSIF ( a = 20 ) THEN
    dbms_output.put_line('Value of a is 20' );
  ELSIF ( a = 30 ) THEN
    dbms_output.put_line('Value of a is 30' );
  ELSE
    dbms_output.put_line('None of the values is matching');
  END IF;
  dbms_output.put_line('Exact value of a is: '|| a );
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:

```

Oracle SQL*Plus
File Edit Search Options Help
SQL> set serveroutput on
SQL> DECLARE
2   a number(3) := 100;
3   BEGIN
4     IF ( a = 10 ) THEN
5       dbms_output.put_line('Value of a is 10' );
6     ELSIF ( a = 20 ) THEN
7       dbms_output.put_line('Value of a is 20' );
8     ELSIF ( a = 30 ) THEN
9       dbms_output.put_line('Value of a is 30' );
10    ELSE
11      dbms_output.put_line('None of the values is matching');
12    END IF;
13    dbms_output.put_line('Exact value of a is: '|| a );
14  END;
15 /
None of the values is matching
Exact value of a is: 100

PL/SQL procedure successfully completed.

SQL> |

```

Fig. 14.11. IF-THEN-ELSIF statement – Example.

(iv) CASE

Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression, whose value is used to select one of several alternatives.

Syntax:

```

CASE selector
    WHEN 'value1' THEN Sequence of statement 1;
    WHEN 'value2' THEN Sequence of statement 2;
    WHEN 'value3' THEN Sequence of statement 3;
    ...
    ELSE Sequence of statement n; -- default case
END CASE;

```

Example:

```

DECLARE
    grade char(1) := 'A';
BEGIN
    CASE grade
        when 'A' then dbms_output.put_line('Excellent');
        when 'B' then dbms_output.put_line('Very good');
        when 'C' then dbms_output.put_line('Well done');
        when 'D' then dbms_output.put_line('You passed');
        when 'F' then dbms_output.put_line('Better try again');
        else dbms_output.put_line('No such grade');
    END CASE;
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:

```

Oracle SQL*Plus
File Edit Search Options Help
SQL> DECLARE
 2   grade char(1) := 'A';
 3 BEGIN
 4   CASE grade
 5     when 'A' then dbms_output.put_line('Excellent');
 6     when 'B' then dbms_output.put_line('Very good');
 7     when 'C' then dbms_output.put_line('Well done');
 8     when 'D' then dbms_output.put_line('You passed');
 9     when 'F' then dbms_output.put_line('Better try again');
10     else dbms_output.put_line('No such grade');
11   END CASE;
12 END;
13 /
Excellent

PL/SQL procedure successfully completed.

```

Fig. 14.12. CASE statement – Example.

(v) SEARCHED CASE

The searched CASE statement has no selector and its WHEN clauses contain search conditions that give Boolean values.

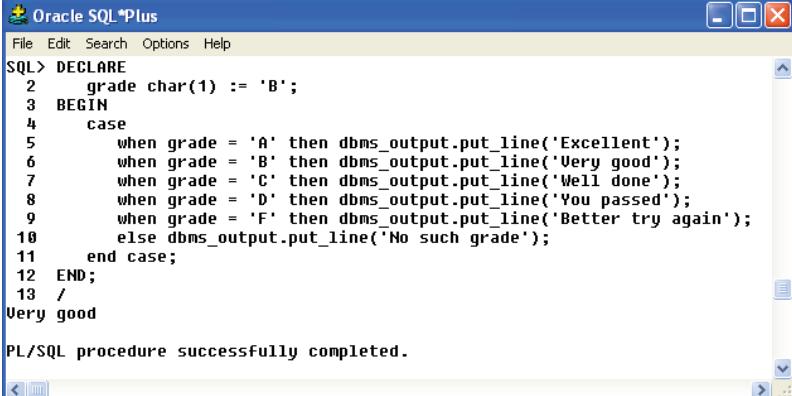
Syntax:

```
CASE
    WHEN selector = 'value1' THEN Sequence of statement 1;
    WHEN selector = 'value2' THEN Sequence of statement 2;
    WHEN selector = 'value3' THEN Sequence of statement 3;
    ...
    ELSE Sequence of statement n; -- default case
END CASE;
```

Example:

```
DECLARE
grade char(1) := 'B';
BEGIN
case
when grade = 'A' then dbms_output.put_line('Excellent');
when grade = 'B' then dbms_output.put_line('Very good');
when grade = 'C' then dbms_output.put_line('Well done');
when grade = 'D' then dbms_output.put_line('You passed');
when grade = 'F' then dbms_output.put_line('Better try again');
else dbms_output.put_line('No such grade');
end case;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



```
Oracle SQL*Plus
File Edit Search Options Help
SQL> DECLARE
 2   grade char(1) := 'B';
 3 BEGIN
 4   case
 5     when grade = 'A' then dbms_output.put_line('Excellent');
 6     when grade = 'B' then dbms_output.put_line('Very good');
 7     when grade = 'C' then dbms_output.put_line('Well done');
 8     when grade = 'D' then dbms_output.put_line('You passed');
 9     when grade = 'F' then dbms_output.put_line('Better try again');
10     else dbms_output.put_line('No such grade');
11   end case;
12 END;
13 /
Very good
PL/SQL procedure successfully completed.
```

Fig. 14.13. SEARCHED CASE statement – Example.

(vi) NESTED IF-THEN-ELSE

It is always legal in PL/SQL programming to nest IF-ELSE statements, which means you can use one IF or ELSE IF statement inside another IF or ELSE IF statement(s).

Syntax:

```

IF( boolean_expression 1)THEN
    -- executes when the boolean expression 1 is true
    IF(boolean_expression 2) THEN
        -- executes when the boolean expression 2 is true
        sequence-of-statements;
    END IF;
ELSE
    -- executes when the boolean expression 1 is not true
    else-statements;
END IF;

```

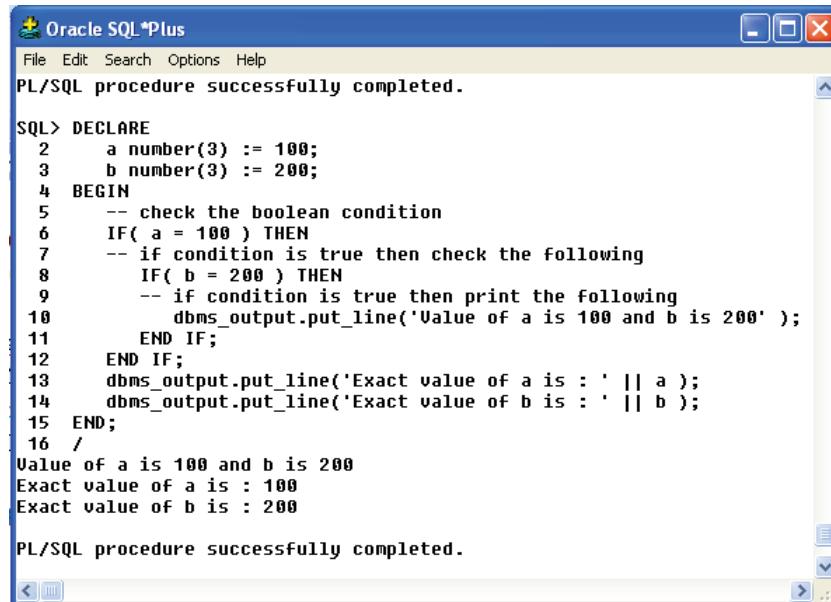
Example:

```

DECLARE
    a number(3) := 100;
    b number(3) := 200;
BEGIN
    -- check the boolean condition
    IF( a = 100 ) THEN
        -- if condition is true then check the following
        IF( b = 200 ) THEN
            -- if condition is true then print the following
            dbms_output.put_line('Value of a is 100 and b is 200');
        END IF;
    END IF;
    dbms_output.put_line('Exact value of a is : ' || a );
    dbms_output.put_line('Exact value of b is : ' || b );
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



```

+ Oracle SQL*Plus
File Edit Search Options Help
PL/SQL procedure successfully completed.

SQL> DECLARE
  2      a number(3) := 100;
  3      b number(3) := 200;
  4  BEGIN
  5      -- check the boolean condition
  6      IF( a = 100 ) THEN
  7          -- if condition is true then check the following
  8          IF( b = 200 ) THEN
  9              -- if condition is true then print the following
 10              dbms_output.put_line('Value of a is 100 and b is 200' );
 11          END IF;
 12      END IF;
 13      dbms_output.put_line('Exact value of a is : ' || a );
 14      dbms_output.put_line('Exact value of b is : ' || b );
 15  END;
 16 /
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200

PL/SQL procedure successfully completed.

```

Fig. 14.14. NESTED-IF-THEN-ELSE statement – Example.

14.6.2 Iterative Control

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths. There are mainly three types of loop statements:

- (i) BASIC LOOP
- (ii) WHILE-LOOP
- (iii) FOR-LOOP
- (iv) NESTED LOOP

(i) BASIC LOOP

Basic loop structure encloses sequence of statements in between the LOOP and END LOOP statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax:

LOOP
Sequence of statements;
END LOOP;

Here, sequence of statement(s) may be a single statement or a block of statements. An EXIT statement or an EXIT WHEN statement is required to break the loop.

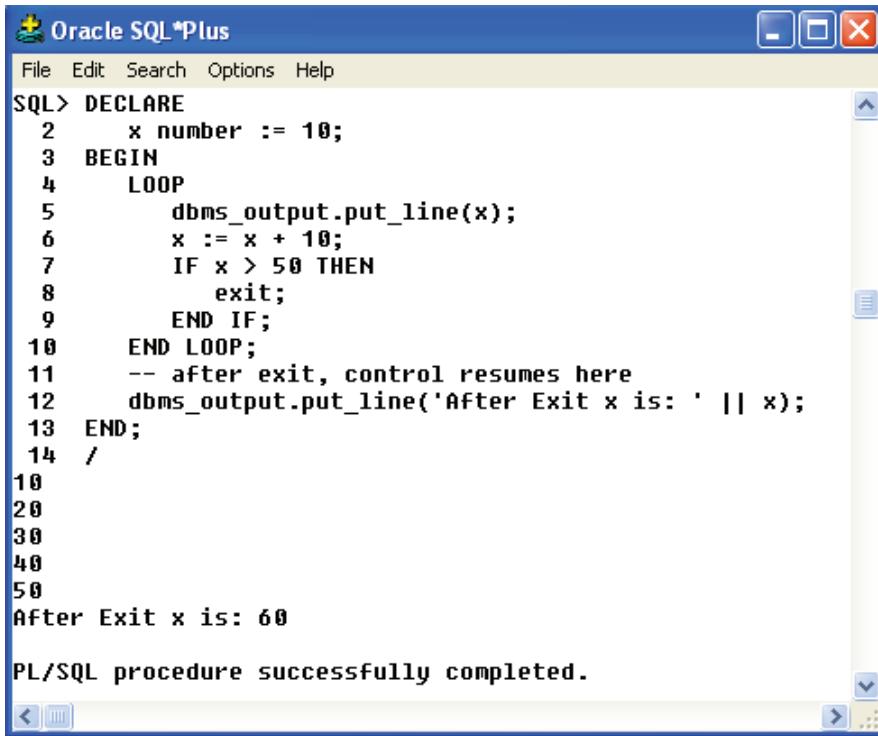
Example:

```

DECLARE
    x number := 10;
BEGIN
    LOOP
        dbms_output.put_line(x);
        x := x + 10;
        IF x > 50 THEN
            exit;
        END IF;
    END LOOP;
    -- after exit, control resumes here
    dbms_output.put_line('After Exit x is: ' || x);
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The command window displays the following PL/SQL code:

```

SQL> DECLARE
  2      x number := 10;
  3  BEGIN
  4      LOOP
  5          dbms_output.put_line(x);
  6          x := x + 10;
  7          IF x > 50 THEN
  8              exit;
  9          END IF;
 10      END LOOP;
 11      -- after exit, control resumes here
 12      dbms_output.put_line('After Exit x is: ' || x);
 13  END;
 14 /

```

The output window shows the execution results:

```

10
20
30
40
50
After Exit x is: 60

PL/SQL procedure successfully completed.

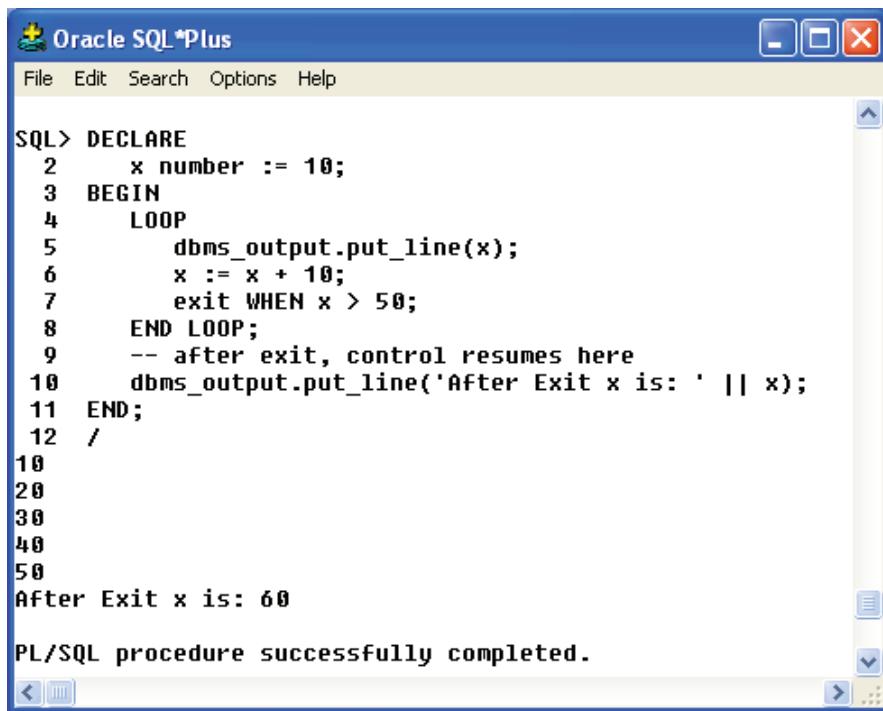
```

Fig. 14.15. Basic loop statement – Example with Exit.

You can use the EXIT WHEN statement instead of the EXIT statement:

```
DECLARE
    x number := 10;
BEGIN
    LOOP
        dbms_output.put_line(x);
        x := x + 10;
        exit WHEN x > 50;
    END LOOP;
    -- after exit, control resumes here
    dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The command window displays the following PL/SQL code:

```
SQL> DECLARE
  2      x number := 10;
  3  BEGIN
  4      LOOP
  5          dbms_output.put_line(x);
  6          x := x + 10;
  7          exit WHEN x > 50;
  8      END LOOP;
  9      -- after exit, control resumes here
10      dbms_output.put_line('After Exit x is: ' || x);
11  END;
12  /
```

Below the code, the output window shows the values of 'x' from 10 to 50, followed by the message 'After Exit x is: 60', and finally 'PL/SQL procedure successfully completed.'

Fig. 14.16. Basic loop statement – Example with Exit When.

(ii) WHILE LOOP

A WHILE LOOP statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

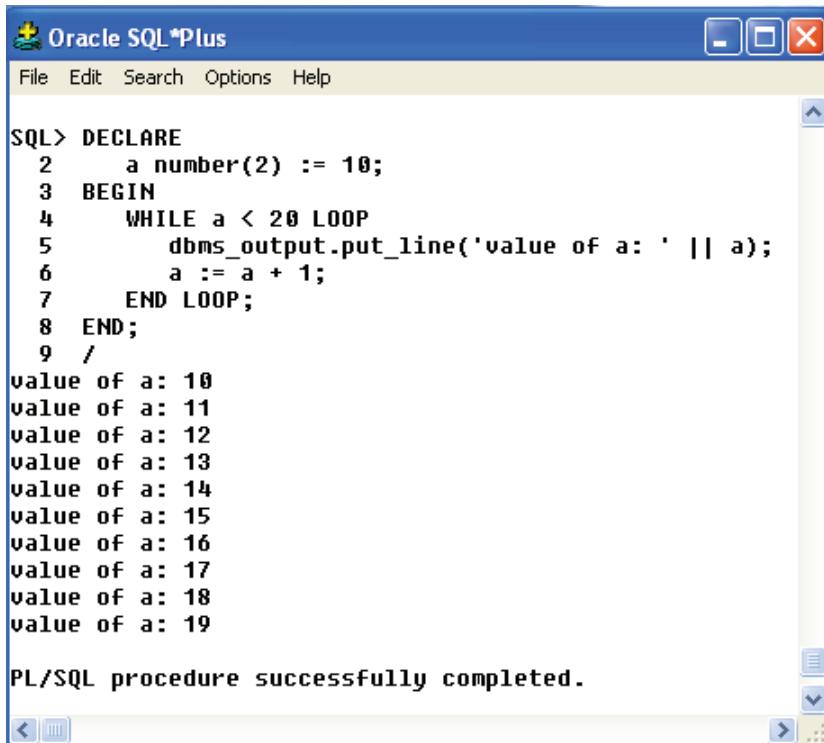
Syntax:

```
WHILE condition LOOP
    sequence_of_statements
END LOOP;
```

Example:

```
DECLARE
    a number(2) := 10;
BEGIN
    WHILE a < 20 LOOP
        dbms_output.put_line('value of a: ' || a);
        a := a + 1;
    END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following PL/SQL code and its output:

```
SQL> DECLARE
  2      a number(2) := 10;
  3  BEGIN
  4      WHILE a < 20 LOOP
  5          dbms_output.put_line('value of a: ' || a);
  6          a := a + 1;
  7      END LOOP;
  8  END;
  9 /
```

The output window shows the values of 'a' from 10 to 19, each on a new line:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

At the bottom of the output window, the message "PL/SQL procedure successfully completed." is displayed.

Fig. 14.17. While loop – Example.

(iii) FOR LOOP

A FOR LOOP is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
FOR counter IN initial_value .. final_value LOOP  
    sequence_of_statements;  
END LOOP;
```

Here is the flow of control in for loop:

- The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- Next, the condition, i.e., initial_value .. final_value is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and flow of control jumps to the next statement just after the FOR-LOOP.
- After the body of the FOR-LOOP executes, the value of the counter variable is increased or decreased.
- The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself. After the condition becomes FALSE, the FOR-LOOP terminates.

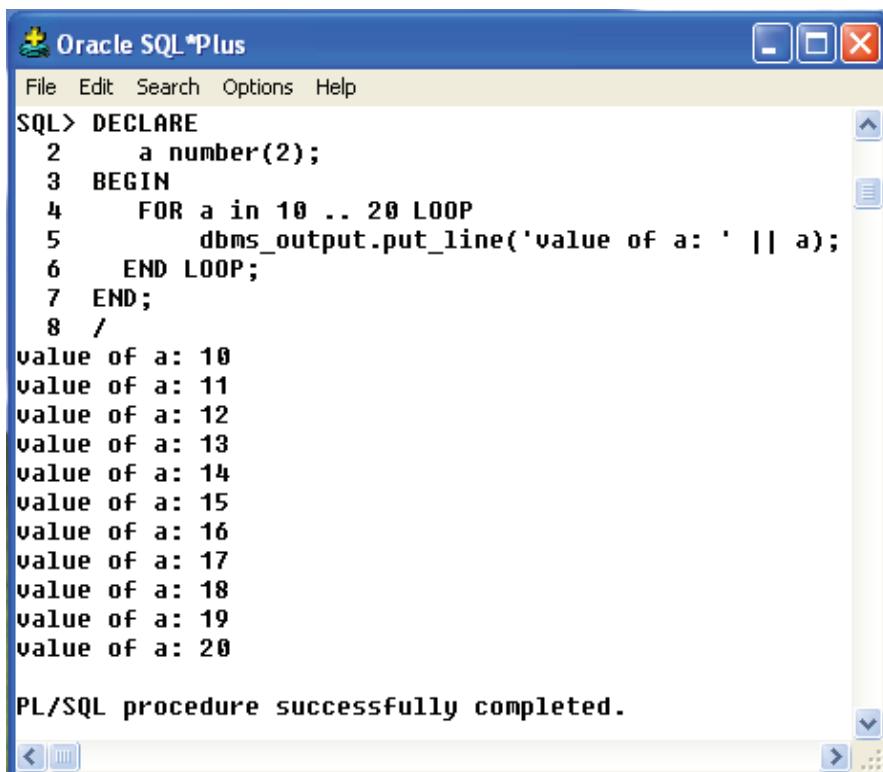
Following are some special characteristics of PL/SQL for loop:

- The initial_value and final_value of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.
- The initial_value need not to be 1; however, the loop counter increment must be 1.
- PL/SQL allows determine the loop range dynamically at run time.

Example:

```
DECLARE  
    a number(2);  
BEGIN  
    FOR a in 10 .. 20 LOOP  
        dbms_output.put_line('value of a: ' || a);  
    END LOOP;  
END;  
/
```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The SQL prompt "SQL>" is followed by a PL/SQL block:

```

SQL> DECLARE
  2      a number(2);
  3  BEGIN
  4      FOR a in 10 .. 20 LOOP
  5          dbms_output.put_line('value of a: ' || a);
  6      END LOOP;
  7  END;
  8 /

```

The output window displays the values of 'a' from 10 to 20, each on a new line:

```

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20

```

At the bottom, a message indicates the procedure was successfully completed:

```

PL/SQL procedure successfully completed.

```

Fig. 14.18. For loop – Example.

Reverse FOR LOOP Statement

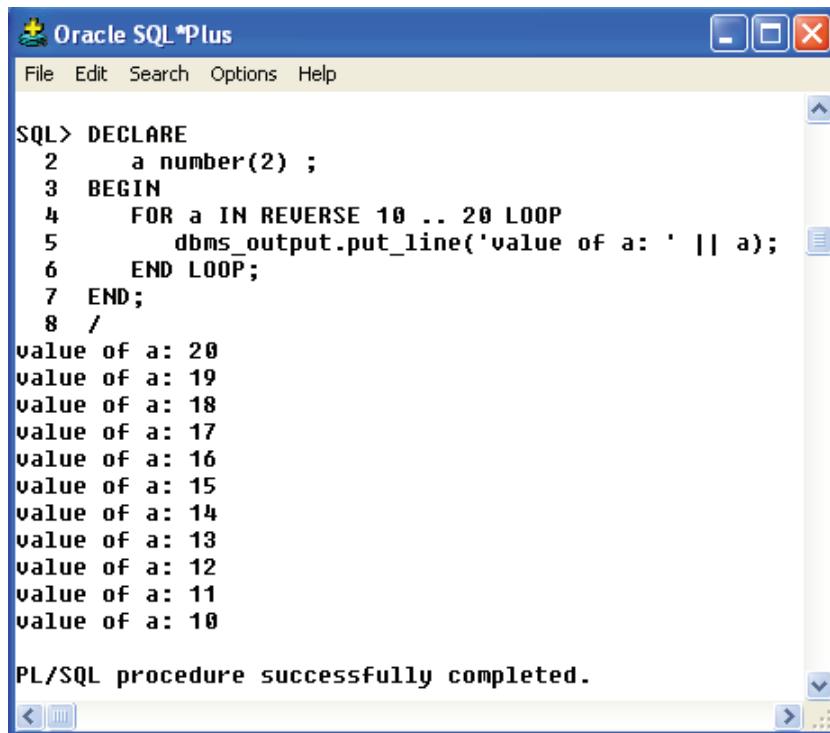
By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the REVERSE keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented. However, you must write the range bounds in ascending order. The following program illustrates this:

```

DECLARE
    a number(2) ;
BEGIN
    FOR a IN REVERSE 10 .. 20 LOOP
        dbms_output.put_line('value of a: ' || a);
    END LOOP;
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. In the command window, a PL/SQL block is executed:

```

SQL> DECLARE
  2    a number(2) ;
  3  BEGIN
  4    FOR a IN REVERSE 10 .. 20 LOOP
  5      dbms_output.put_line('value of a: ' || a);
  6    END LOOP;
  7  END;
  8 /

```

The output window displays the values of 'a' from 20 down to 10:

```

value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10

```

At the bottom, a message indicates the procedure was successfully completed.

Fig. 14.19. Reverse For loop – Example.

(iv) NESTED LOOP

PL/SQL allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

```


$$\begin{array}{c} \textit{LOOP} \\ \quad \textit{Sequence of statements1} \\ \quad \textit{LOOP} \\ \quad \quad \textit{Sequence of statements2} \\ \quad \textit{END LOOP;} \\ \textit{END LOOP;} \end{array}$$


```

The syntax for a nested FOR LOOP statement in PL/SQL is as follows:

```


$$\begin{array}{c} \textit{FOR counter1 IN initial\_value1 .. final\_value1 LOOP} \\ \quad \textit{sequence\_of\_statements1} \\ \quad \textit{FOR counter2 IN initial\_value2 .. final\_value2 LOOP} \\ \quad \quad \textit{sequence\_of\_statements2} \\ \quad \textit{END LOOP;} \\ \textit{END LOOP;} \end{array}$$


```

The syntax for a nested WHILE LOOP statement in Pascal is as follows:

```
WHILE condition1 LOOP
    sequence_of_statements1
    WHILE condition2 LOOP
        sequence_of_statements2
    END LOOP;
END LOOP;
```

Example:

The following program uses a nested basic loop to find the prime numbers from 2 to 100:

```
DECLARE
    i number(3);
    j number(3);
BEGIN
    i := 2;
    LOOP
        j:= 2;
        LOOP
            exit WHEN ((mod(i, j) = 0) or (j = i));
            j := j +1;
        END LOOP;
        IF (j = i ) THEN
            dbms_output.put_line(i || ' is prime');
        END IF;
        i := i + 1;
        exit WHEN i = 50;
    END LOOP;
END;
```

/

When the above code is executed at SQL prompt, it produces the following result:

The screenshot shows the Oracle SQL*Plus interface with a blue title bar containing the logo and the text "Oracle SQL*Plus". Below the title bar is a menu bar with "File", "Edit", "Search", "Options", and "Help". The main window displays a PL/SQL script and its execution results.

```
SQL> DECLARE
  2      i number(3);
  3      j number(3);
  4  BEGIN
  5      i := 2;
  6      LOOP
  7          j := 2;
  8          LOOP
  9              EXIT WHEN ((mod(i, j) = 0) OR (j = i));
 10              j := j + 1;
 11          END LOOP;
 12      IF (j = i) THEN
 13          dbms_output.put_line(i || ' is prime');
 14      END IF;
 15      i := i + 1;
 16      EXIT WHEN i = 50;
 17      END LOOP;
 18  END;
 19 /
```

2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime

PL/SQL procedure successfully completed.

Fig. 14.20. Nested loop – Example.

14.6.2.1 Labeling a PL/SQL Loop

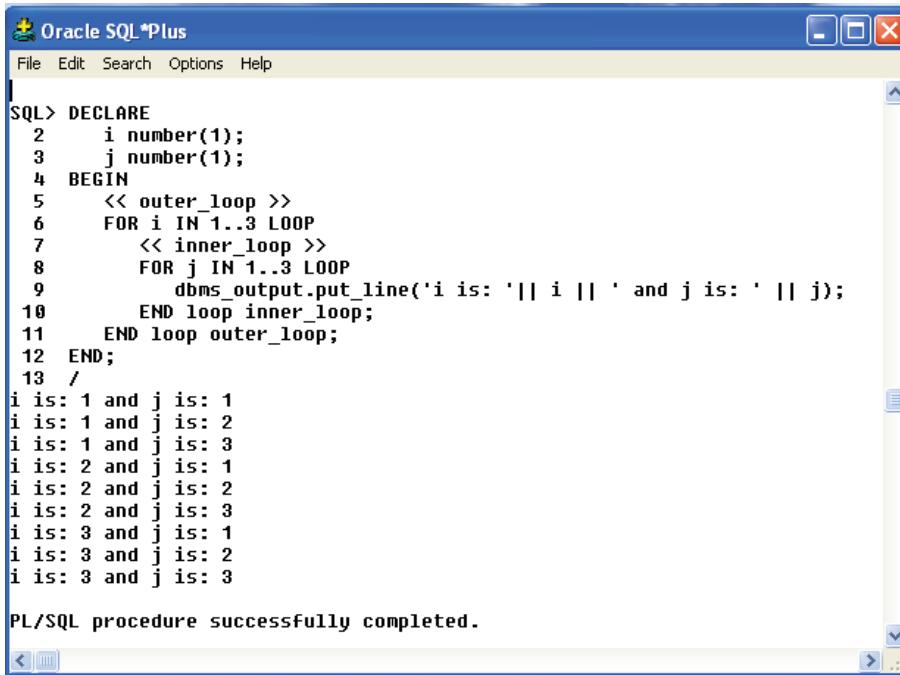
PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop. The following program illustrates the concept:

```

DECLARE
    i number(1);
    j number(1);
BEGIN
    << outer_loop >>
    FOR i IN 1..3 LOOP
        << inner_loop >>
        FOR j IN 1..3 LOOP
            dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
        END loop inner_loop;
    END loop outer_loop;
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays PL/SQL code and its execution results. The code is as follows:

```

SQL> DECLARE
  2      i number(1);
  3      j number(1);
  4  BEGIN
  5      << outer_loop >>
  6      FOR i IN 1..3 LOOP
  7          << inner_loop >>
  8          FOR j IN 1..3 LOOP
  9              dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
 10          END loop inner_loop;
 11      END loop outer_loop;
 12  END;
 13 /

```

Below the code, the output shows the execution of the loops:

```

i is: 1 and j is: 1
i is: 1 and j is: 2
i is: 1 and j is: 3
i is: 2 and j is: 1
i is: 2 and j is: 2
i is: 2 and j is: 3
i is: 3 and j is: 1
i is: 3 and j is: 2
i is: 3 and j is: 3

```

At the bottom, a message indicates the procedure was successfully completed.

Fig. 14.21. Nested loop – Example.

14.6.2.2 The Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. PL/SQL supports the

following control statements. Labeling loops also helps in taking the control outside a loop. Loop control statement includes

- (i) Exit
- (ii) Exit When
- (iii) Goto

(i) Exit

The EXIT statement in PL/SQL programming language has following two usages:

- When the EXIT statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- If you are using nested loops (i.e. one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

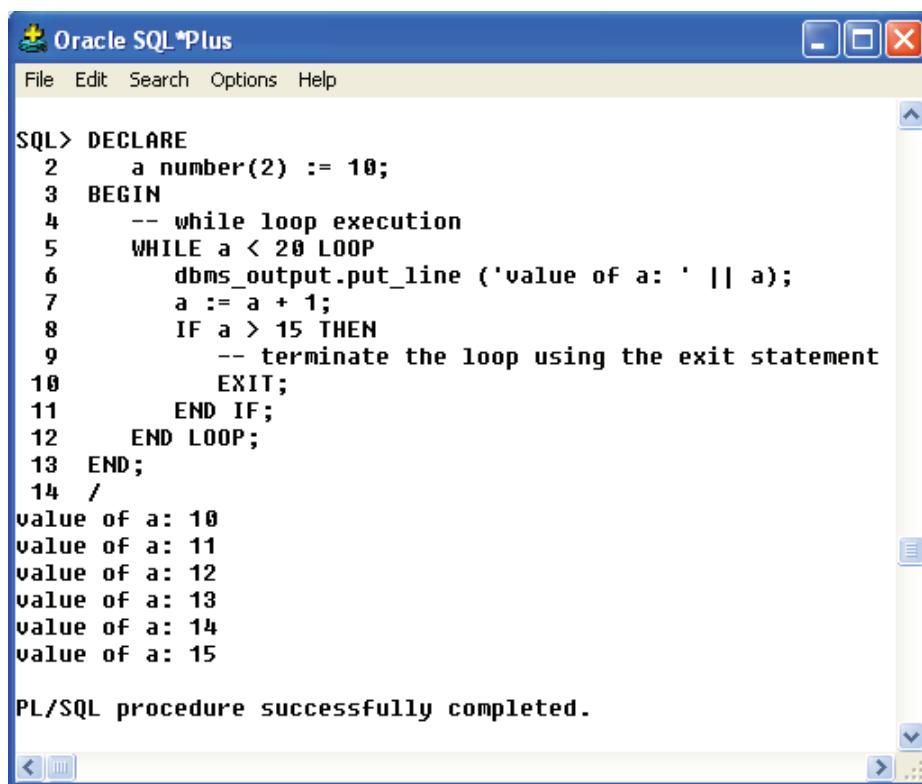
Syntax:

EXIT;

Example:

```
DECLARE
    a number(2) := 10;
BEGIN
    -- while loop execution
    WHILE a < 20 LOOP
        dbms_output.put_line ('value of a: ' || a);
        a := a + 1;
        IF a > 15 THEN
            -- terminate the loop using the exit statement
            EXIT;
        END IF;
    END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



```

SQL> DECLARE
 2      a number(2) := 10;
 3  BEGIN
 4      -- while loop execution
 5      WHILE a < 20 LOOP
 6          dbms_output.put_line ('value of a: ' || a);
 7          a := a + 1;
 8          IF a > 15 THEN
 9              -- terminate the loop using the exit statement
10              EXIT;
11          END IF;
12      END LOOP;
13  END;
14 /
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

PL/SQL procedure successfully completed.

```

Fig. 14.22. Exit – Example.

(ii) EXIT WHEN Statement

The EXIT-WHEN statement allows the condition in the WHEN clause to be evaluated. If the condition is true, the loop completes and control passes to the statement immediately after END LOOP. The EXIT WHEN statement replaces a conditional statement like if-then used with the EXIT statement. Following are two important aspects for the EXIT WHEN statement:

- Until the condition is true, the EXIT-WHEN statement acts like a NULL statement, except for evaluating the condition, and does not terminate the loop.
- A statement inside the loop must change the value of the condition.

Syntax:

EXIT WHEN *condition*;

Example:

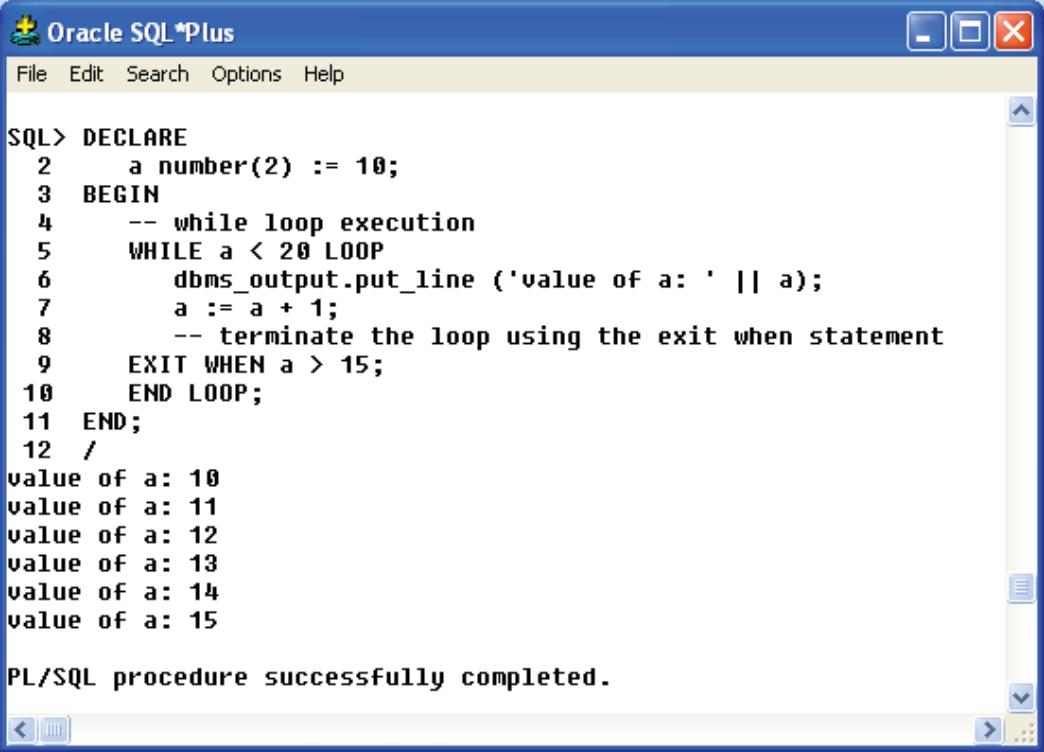
```

DECLARE
    a number(2) := 10;
BEGIN
    -- while loop execution
    WHILE a < 20 LOOP

```

```
        dbms_output.put_line ('value of a: ' || a);
        a := a + 1;
        -- terminate the loop using the exit when statement
        EXIT WHEN a > 15;
    END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following PL/SQL code and its output:

```
SQL> DECLARE
  2      a number(2) := 10;
  3  BEGIN
  4      -- while loop execution
  5      WHILE a < 20 LOOP
  6          dbms_output.put_line ('value of a: ' || a);
  7          a := a + 1;
  8          -- terminate the loop using the exit when statement
  9      EXIT WHEN a > 15;
 10     END LOOP;
 11 END;
 12 /
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

PL/SQL procedure successfully completed.
```

Fig. 14.23. Exit When – Example.

(iii) GOTO

A GOTO statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram. Use of GOTO statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Syntax:

GOTO label;

..

..

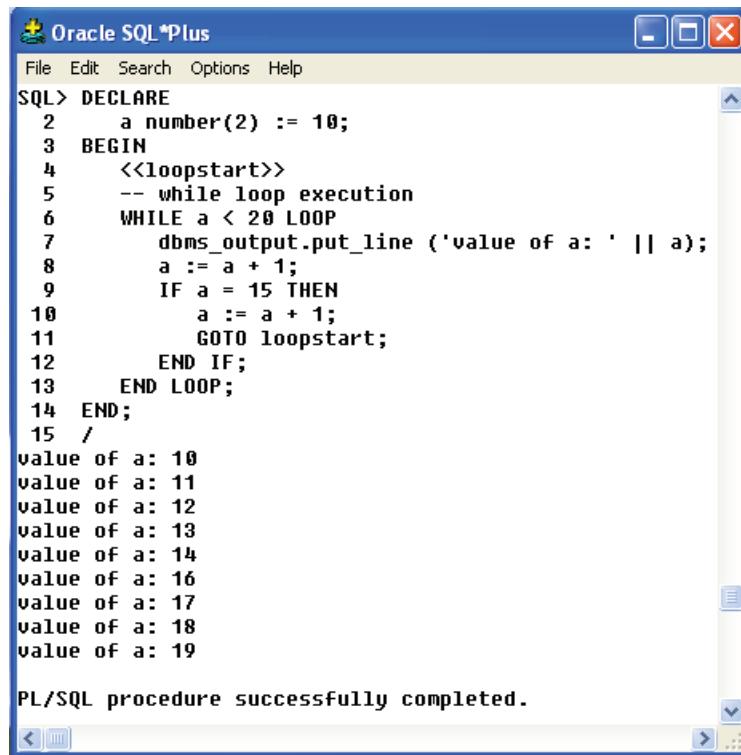
<< label >>

statement;

Example:

```
DECLARE
    a number(2) := 10;
BEGIN
    <<loopstart>>
    -- while loop execution
    WHILE a < 20 LOOP
        dbms_output.put_line ('value of a: ' || a);
        a := a + 1;
        IF a = 15 THEN
            a := a + 1;
            GOTO loopstart;
        END IF;
    END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The command line displays a PL/SQL block. The code uses a GOTO statement to skip the IF block when the value of variable 'a' reaches 15. The output window shows the values of 'a' from 10 to 19, with the value 15 being skipped. The message "PL/SQL procedure successfully completed." is displayed at the bottom.

```

SQL> DECLARE
  2   a number(2) := 10;
  3   BEGIN
  4     <<loopstart>>
  5     -- while loop execution
  6     WHILE a < 20 LOOP
  7       dbms_output.put_line ('value of a: ' || a);
  8       a := a + 1;
  9       IF a = 15 THEN
 10         a := a + 1;
 11         GOTO loopstart;
 12       END IF;
 13     END LOOP;
 14   END;
 15 /
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.

```

Fig. 14.24. GOTO statement – Example.

Restrictions with GOTO Statement

GOTO Statement in PL/SQL imposes the following restrictions:

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
- A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.
- A GOTO statement cannot branch from an outer block into a sub-block.
- A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.
- A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

14.7 PL/SQL STRINGS

The string in PL/SQL is a sequence of characters with an optional size specification. The characters can be numeric, letters, blank, special characters or a combination of all these. PL/SQL offers three kinds of strings:

- **Fixed-length strings:** In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
- **Variable-length strings:** In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.
- **Character large objects (CLOBs):** These are variable-length strings that can be up to 128 terabytes.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example,

'This is a string literal.' Or 'hello world'

To include a single quote inside a string literal, you need to type two single quotes next to one another, like:

'this isn't what it looks like'

14.7.1 Declaring String Variables

Oracle database provides numerous string data types like, CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The data types prefixed with an 'N' are 'national character set' data types, that store Unicode character data.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables:

```

DECLARE
    name varchar2(20);
    company varchar2(30);
    introduction clob;
    choice char(1);

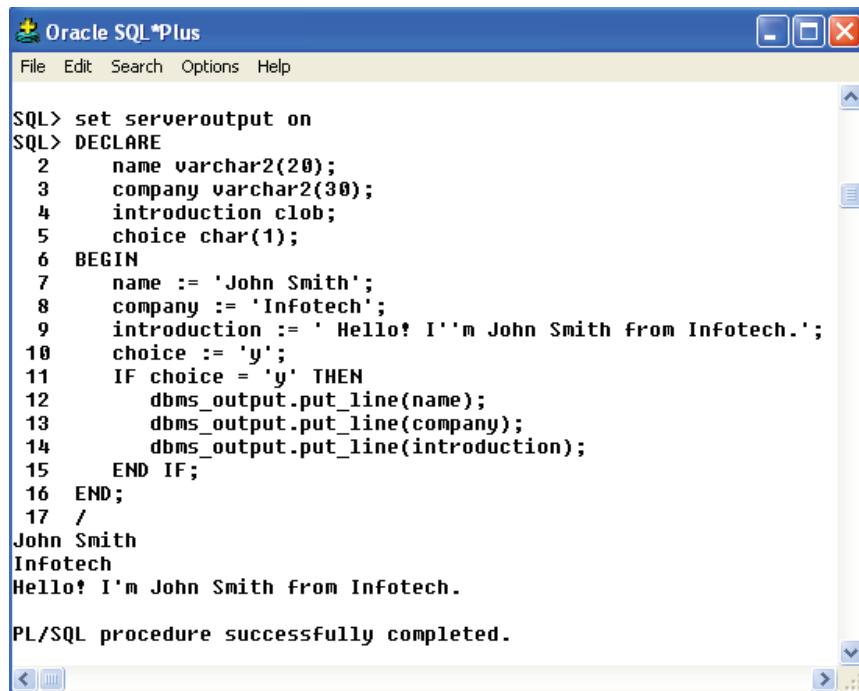
BEGIN
    name := 'John Smith';
    company := 'Infotech';
    introduction := 'Hello! I'm John Smith from Infotech.';
    choice := 'y';

    IF choice = 'y' THEN
        dbms_output.put_line(name);
        dbms_output.put_line(company);
        dbms_output.put_line(introduction);
    END IF;

END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The command line displays:

```
SQL> set serveroutput on
SQL> DECLARE
  2      name varchar2(20);
  3      company varchar2(30);
  4      introduction clob;
  5      choice char(1);
  6  BEGIN
  7      name := 'John Smith';
  8      company := 'Infotech';
  9      introduction := 'Hello! I''m John Smith from Infotech.';
 10      choice := 'y';
 11      IF choice = 'y' THEN
 12          dbms_output.put_line(name);
 13          dbms_output.put_line(company);
 14          dbms_output.put_line(introduction);
 15      END IF;
 16  END;
 17 /
John Smith
Infotech
Hello! I'm John Smith from Infotech.

PL/SQL procedure successfully completed.
```

Fig. 14.25. String declaration – Example.

To declare a fixed-length string, use the CHAR data type. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. So following two declarations below are identical:

```
red_flag CHAR(1) := 'Y';
red_flag CHAR    := 'Y';
```

14.7.2 PL/SQL String Functions and Operators

PL/SQL offers the concatenation operator (||) for joining two strings. The following table provides the string functions provided by PL/SQL:

S.N.	Function & Purpose
1	ASCII(x); Returns the ASCII value of the character x.
2	CHR(x); Returns the character with the ASCII value of x.
3	CONCAT(x, y); Concatenates the strings x and y and return the appended string.
4	INITCAP(x); Converts the initial letter of each word in x to uppercase and returns that string.

5	INSTR(x, find_string [, start] [, occurrence]); Searches for find_string in x and returns the position at which it occurs.
6	INSTRB(x); Returns the location of a string within another string, but returns the value in bytes.
7	LENGTH(x); Returns the number of characters in x.
8	LENGTHB(x); Returns the length of a character string in bytes for single byte character set.
9	LOWER(x); Converts the letters in x to lowercase and returns that string.
10	LPAD(x, width [, pad_string]); Pads x with spaces to left, to bring the total length of the string up to width characters.
11	LTRIM(x [, trim_string]); Trims characters from the left of x.
12	NANVL(x, value); Returns value if x matches the NaN special value (not a number), otherwise x is returned.
13	NLS_INITCAP(x); Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT.
14	NLS_LOWER(x); Same as the LOWER function except that it can use a different sort method as specified by NLSSORT.
15	NLS_UPPER(x); Same as the UPPER function except that it can use a different sort method as specified by NLSSORT.
16	NLSSORT(x); Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used.
17	NVL(x, value); Returns value if x is null; otherwise, x is returned.
18	NVL2(x, value1, value2); Returns value1 if x is not null; if x is null, value2 is returned.
19	REPLACE(x, search_string, replace_string); Searches x for search_string and replaces it with replace_string.
20	RPAD(x, width [, pad_string]); Pads x to the right.
21	RTRIM(x [, trim_string]); Trims x from the right.

22	SOUNDEX(x) ; Returns a string containing the phonetic representation of x.
23	SUBSTR(x, start [, length]); Returns a substring of x that begins at the position specified by start. An optional length for the substring may be supplied.
24	SUBSTRB(x); Same as SUBSTR except the parameters are expressed in bytes instead of characters for the single-byte character systems.
25	TRIM([trim_char FROM) x); Trims characters from the left and right of x.
26	UPPER(x); Converts the letters in x to uppercase and returns that string.

The following examples illustrate some of the above-mentioned functions and their use:

Example 1

```

DECLARE
    greetings varchar2(11) := 'hello world';
BEGIN
    dbms_output.put_line(UPPER(greetings));

    dbms_output.put_line(LOWER(greetings));

    dbms_output.put_line(INITCAP(greetings));

    /* retrieve the first character in the string */
    dbms_output.put_line ( SUBSTR (greetings, 1, 1));

    /* retrieve the last character in the string */
    dbms_output.put_line ( SUBSTR (greetings, -1, 1));

    /* retrieve five characters,
       starting from the seventh position. */
    dbms_output.put_line ( SUBSTR (greetings, 7, 5));

    /* retrieve the remainder of the string,
       starting from the second position. */
    dbms_output.put_line ( SUBSTR (greetings, 2));

```

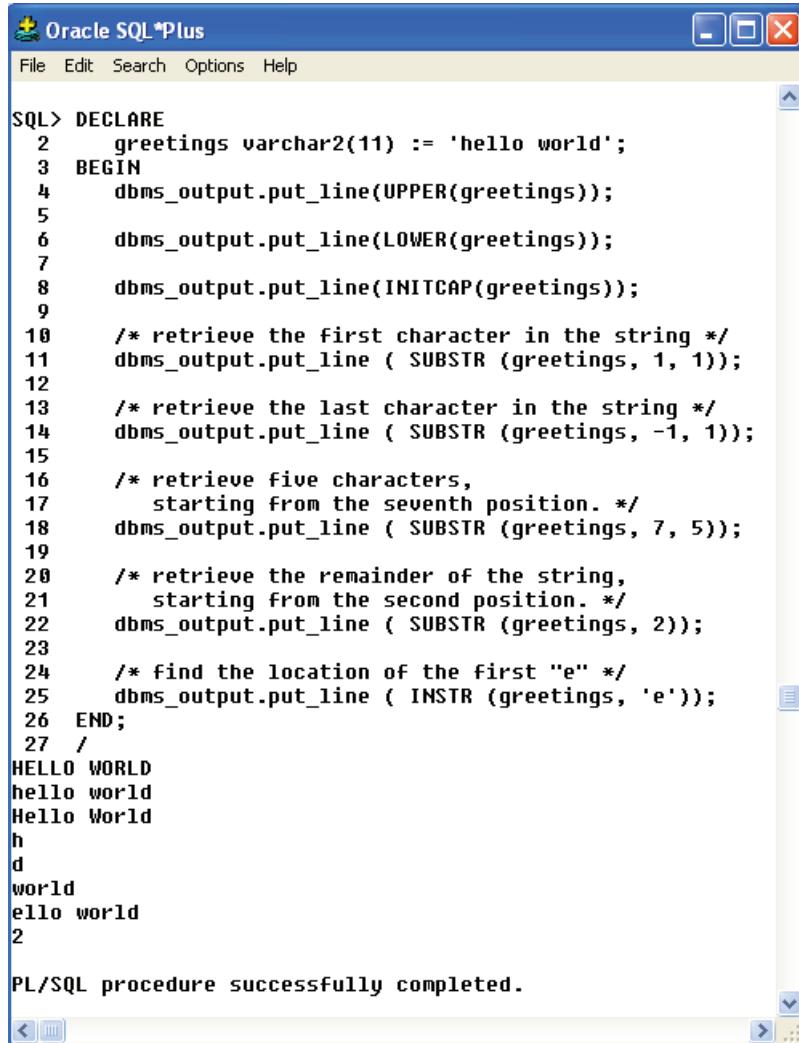
```

/* find the location of the first "e" */
dbms_output.put_line ( INSTR (greetings, 'e'));

END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following PL/SQL code and its output:

```

SQL> DECLARE
  2      greetings varchar2(11) := 'hello world';
  3  BEGIN
  4      dbms_output.put_line(UPPER(greetings));
  5
  6      dbms_output.put_line(LOWER(greetings));
  7
  8      dbms_output.put_line(INITCAP(greetings));
  9
 10     /* retrieve the first character in the string */
 11     dbms_output.put_line ( SUBSTR (greetings, 1, 1));
 12
 13     /* retrieve the last character in the string */
 14     dbms_output.put_line ( SUBSTR (greetings, -1, 1));
 15
 16     /* retrieve five characters,
 17        starting from the seventh position. */
 18     dbms_output.put_line ( SUBSTR (greetings, 7, 5));
 19
 20     /* retrieve the remainder of the string,
 21        starting from the second position. */
 22     dbms_output.put_line ( SUBSTR (greetings, 2));
 23
 24     /* find the location of the first "e" */
 25     dbms_output.put_line ( INSTR (greetings, 'e'));
 26  END;
 27 /
HELLO WORLD
hello world
Hello World
h
d
world
ello world
2

PL/SQL procedure successfully completed.

```

Fig. 14.26. String functions and operators – Example 1.

Example 2

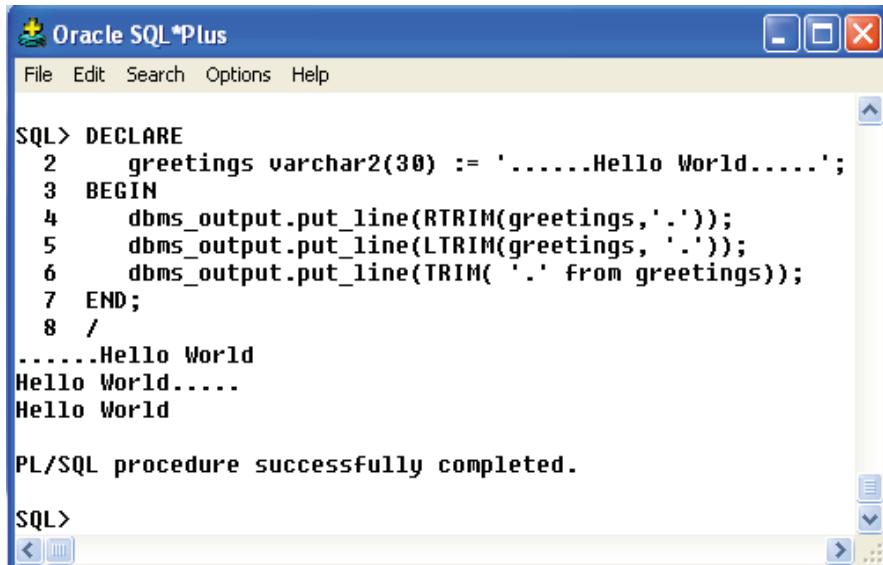
```

DECLARE
    greetings varchar2(30) := '.....Hello World.....';

```

```
BEGIN
    dbms_output.put_line(RTRIM(greetings,'.'));
    dbms_output.put_line(LTRIM(greetings, '.')); 
    dbms_output.put_line(TRIM( '.' from greetings));
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



```
SQL> DECLARE
  2      greetings varchar2(30) := '.....Hello World.....';
  3  BEGIN
  4      dbms_output.put_line(RTRIM(greetings,'.'));
  5      dbms_output.put_line(LTRIM(greetings, '.')); 
  6      dbms_output.put_line(TRIM( '.' from greetings));
  7  END;
  8 /
.....Hello World
Hello World.....
Hello World

PL/SQL procedure successfully completed.
```

Fig. 14.27. String functions and operators – Example 2.

14.8 PL/SQL ARRAYS

PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type. A VARRAY is used to store an ordered collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All VARRAYs consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. An array is a part of collection type data and it stands for variable-size arrays. Each element in a VARRAY has an index associated with it. It also has a maximum size that can be changed dynamically.

14.8.1 Creating a VARRAY Type

A VARRAY type is created with the CREATE TYPE statement. The maximum size and the type of elements stored in the varray must be specified.

Syntax:

CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>

Where,

- *varray_type_name* is a valid attribute name,
- *n* is the number of elements (maximum) in the varray,
- *element_type* is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);

The basic syntax for creating a VRRAY type within a PL/SQL block is:

TYPE varray_type_name IS VARRAY(n) of <element_type>

For example:

TYPE namearray IS VARRAY(5) OF VARCHAR2(10);

Type grades IS VARRAY(5) OF INTEGER;

Example 1

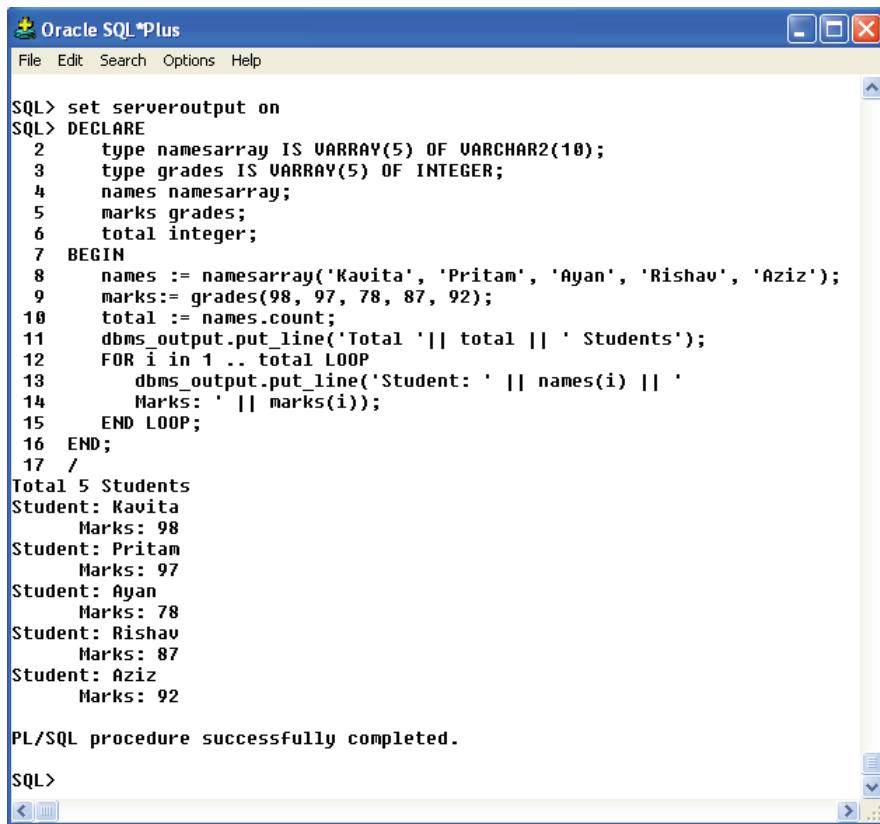
The following program illustrates using varrays:

```

DECLARE
    type namesarray IS VARRAY(5) OF VARCHAR2(10);
    type grades IS VARRAY(5) OF INTEGER;
    names namesarray;
    marks grades;
    total integer;
BEGIN
    names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
    marks:= grades(98, 97, 78, 87, 92);
    total := names.count;
    dbms_output.put_line('Total '|| total || ' Students');
    FOR i in 1 .. total LOOP
        dbms_output.put_line('Student: ' || names(i) || '
        Marks: ' || marks(i));
    END LOOP;
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface with the following PL/SQL code and its execution results:

```

SQL> set serveroutput on
SQL> DECLARE
  2      type namesarray IS VARRAY(5) OF VARCHAR2(10);
  3      type grades IS VARRAY(5) OF INTEGER;
  4      names namesarray;
  5      marks grades;
  6      total integer;
  7  BEGIN
  8      names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
  9      marks:= grades(98, 97, 78, 87, 92);
 10      total := names.count;
 11      dbms_output.put_line('Total '|| total || ' Students');
 12      FOR i in 1 .. total LOOP
 13          dbms_output.put_line('Student: ' || names(i) || ' '
 14          Marks: ' || marks(i));
 15      END LOOP;
 16  END;
 17 /

```

Total 5 Students
 Student: Kavita
 Marks: 98
 Student: Pritam
 Marks: 97
 Student: Ayan
 Marks: 78
 Student: Rishav
 Marks: 87
 Student: Aziz
 Marks: 92

PL/SQL procedure successfully completed.

Fig. 14.28. SQL Array – Example.

14.9 PL/SQL SUBPROGRAM

A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This approach is called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

A subprogram can be created:

- At schema level
- Inside a package
- Inside a PL/SQL block

A schema level subprogram is a standalone subprogram. It is created with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms:

Functions: these subprograms return a single value, mainly used to compute and return a value.

Procedures: these subprograms do not return a value directly, mainly used to perform an action.

14.9.1 Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name and a parameter list. The subprograms will have the following three parts:

1. Declarative Part: It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

2. Executable Part: This is a mandatory part and contains statements that perform the designated action.

3. Exception-handling: This is again an optional part. It contains the code that handles run-time errors.

14.9.1.1 PL/SQL Procedure

A procedure is a subprogram that performs some specific task, and stored in the data dictionary. A procedure must have a name, so that it can be invoked or called by any PL/SQL program that appears within an application. Procedures can take parameters from the calling program and perform the specific task. Before the procedure or function is stored, it gets parsed and compiled. When a procedure is created, the Oracle automatically performs the following steps:

1. Compiles the procedure
2. Stores the procedure in the data dictionary

If an error occurs during creation of procedure, Oracle displays a message that procedure is created with compilation errors, but it does not display the errors. To see the errors following statement is used:

SELECT * FROM user_errors;

When the function is invoked, the Oracle loads the compiled procedure in the memory area called system global area (SGA). Once loaded in the SGA other users can also access the same procedure provided they have granted permission for this.

Benefits of Procedures and Functions

Stored procedures and functions have many benefits in addition to *modularizing* application development.

1. It modifies one routine to affect multiple applications.
2. It modifies one routine to eliminate duplicate testing.

3. It ensures that related actions are performed together, or not at all, by doing the activity through a single path.
4. It avoids PL/SQL parsing at runtime by parsing at compile time.
5. It reduces the number of calls to the database and database network traffic by bundling the commands.

14.9.1.2 Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement.

Syntax:

```
CREATE[OR REPLACE]PROCEDURE procedure_name  
[(parameter_name [IN| OUT |IN OUT] type [...])]  
{IS|AS}  
BEGIN  
<procedure_body>  
END procedure_name;
```

Where,

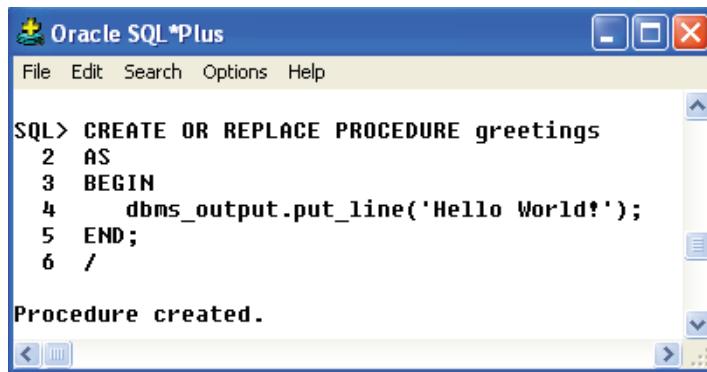
- procedure-name specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- procedure-body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example:

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATEOR REPLACE PROCEDURE greetings  
AS  
BEGIN  
    dbms_output.put_line('Hello World!');  
END;  
/
```

When above code is executed using SQL prompt, it will produce the following result:



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following PL/SQL code:

```
SQL> CREATE OR REPLACE PROCEDURE greetings
  2  AS
  3  BEGIN
  4    dbms_output.put_line('Hello World!');
  5  END;
  6 /
```

Below the code, the message "Procedure created." is displayed. The bottom of the window has standard scroll and search controls.

Fig. 14.29. PL/SQL Procedure creation.

Executing a Standalone Procedure

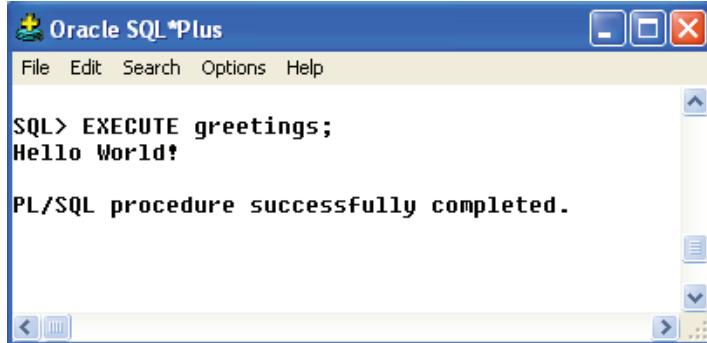
A standalone procedure can be called in two ways:

- Using the EXECUTE keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named 'greetings' can be called with the EXECUTE keyword as:

EXECUTE greetings;

The above call would display:



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The main window displays the following command and its output:

```
SQL> EXECUTE greetings;
Hello World!
```

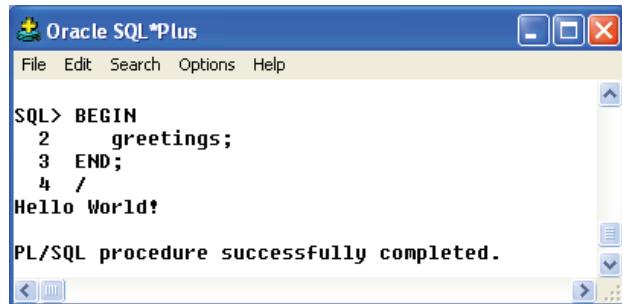
Below the output, the message "PL/SQL procedure successfully completed." is displayed. The bottom of the window has standard scroll and search controls.

Fig. 14.30. PL/SQL Procedure execution.

The procedure can also be called from another PL/SQL block:

```
BEGIN
  greetings;
END;
/
```

The above call would display:



```

SQL> BEGIN
 2   greetings;
 3 END;
 4 /
Hello World!

PL/SQL procedure successfully completed.

```

Fig. 14.31. PL/SQL Procedure execution using PL/SQL block.

14.9.1.3 Deleting a Standalone Procedure

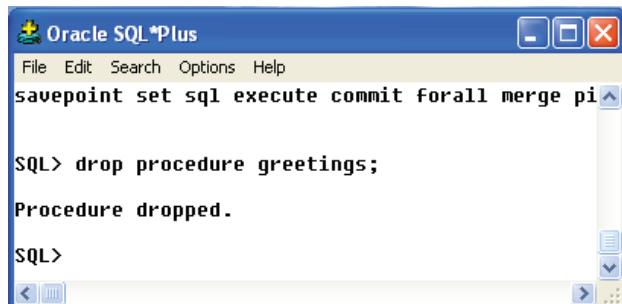
A standalone procedure is deleted with the DROP PROCEDURE statement.

Syntax:

DROP PROCEDURE procedure-name;

Drop greetings procedure by using the following statement:

`DROP PROCEDURE greetings;`



```

savepoint set sql execute commit forall merge pi

SQL> drop procedure greetings;
Procedure dropped.

SQL>

```

Fig. 14.32. PL/SQL Procedure deletion.

14.9.1.4 Parameter Modes in PL/SQL Subprograms

IN: Specifies that a value for the argument must be specified when calling the procedure.

OUT: Specifies that the procedure pass a value for this argument back to its calling environment after execution.

IN OUT: Specifies that a value for the argument must be specified when calling the procedure and that the procedure passes a value for this argument back to its calling environment after execution. If no value is specified then it takes the default value IN.

Methods for Passing Parameters: Actual parameters could be passed in three ways:

- Positional notation
- Named notation
- Mixed notation

Positional Notation: In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on.

Named Notation: In named notation, the actual parameter is associated with the formal parameter using the arrow symbol (\Rightarrow).

Mixed Notation: In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

14.9.2 PL/SQL FUNCTIONS

A Function is similar to procedure except that it returns a value to the calling program. Besides this, a function can be used as part of SQL expression, whereas the procedure cannot.

Creating a Function

A standalone function is created using the CREATE FUNCTION statement.

Syntax:

```
CREATE[OR REPLACE]FUNCTION function_name
[(parameter_name [IN| OUT |IN OUT] type [,....])]
RETURN return_datatype
{IS|AS}
BEGIN
<function_body>
END[function_name];
```

Where,

- function-name specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents that the value will be passed from outside, and OUT represents that this parameter will be used to return a value outside of the procedure.
- The function must contain a return statement.
- RETURN clause specifies that data type you are going to return from the function.
- Function-body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example: The following example illustrates creating and calling a standalone function. This function returns the total number of CUSTOMERS in the customers table. Let us use the following customer table

```
Select*from customers;
```

The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main area displays the following SQL command and its results:

```
SQL> select * from customers;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	3000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500

6 rows selected.

Fig. 14.33. Entries in Customers Table.

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
    total number(2):=0;
BEGIN
    SELECT count(*) into total
    FROM customers;

    RETURN total;
END;
/
```

When above code is executed using SQL prompt, it will produce the following result:

The screenshot shows the Oracle SQL*Plus interface with the title bar "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main area displays the following SQL commands and their execution results:

```
SQL> set serveroutput on
SQL> CREATE OR REPLACE FUNCTION totalCustomers
  2  RETURN number IS
  3      total number(2) := 0;
  4  BEGIN
  5      SELECT count(*) into total
  6      FROM customers;
  7
  8      RETURN total;
  9  END;
 10 /
```

Function created.

Fig. 14.34. PL/SQL Function Creation.

Calling a Function

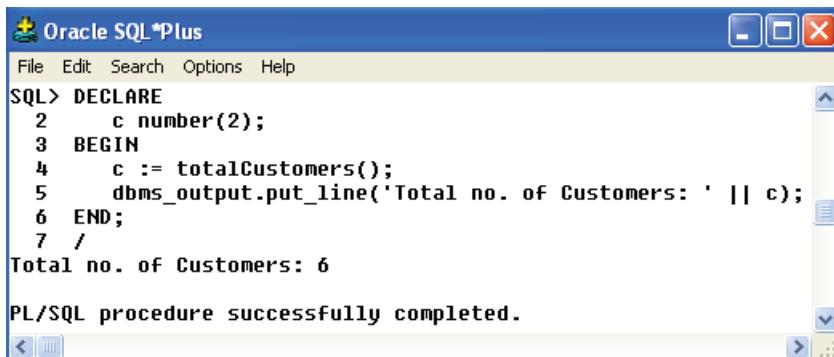
While creating a function, a definition is given for the function stating what the function has to do. The defined function has to be called in order to make the function perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when the end statement is reached, it returns program control back to the main program.

To call a function, the required parameters are passed along with function name and if function returns a value it is stored. Following program calls the function totalCustomers from an anonymous block:

```
DECLARE
    c number(2);
BEGIN
    c := totalCustomers();
    dbms_output.put_line("Total no. of Customers: " || c);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



```
Oracle SQL*Plus
File Edit Search Options Help
SQL> DECLARE
  2      c number(2);
  3  BEGIN
  4      c := totalCustomers();
  5      dbms_output.put_line('Total no. of Customers: ' || c);
  6  END;
  7 /
Total no. of Customers: 6
PL/SQL procedure successfully completed.
```

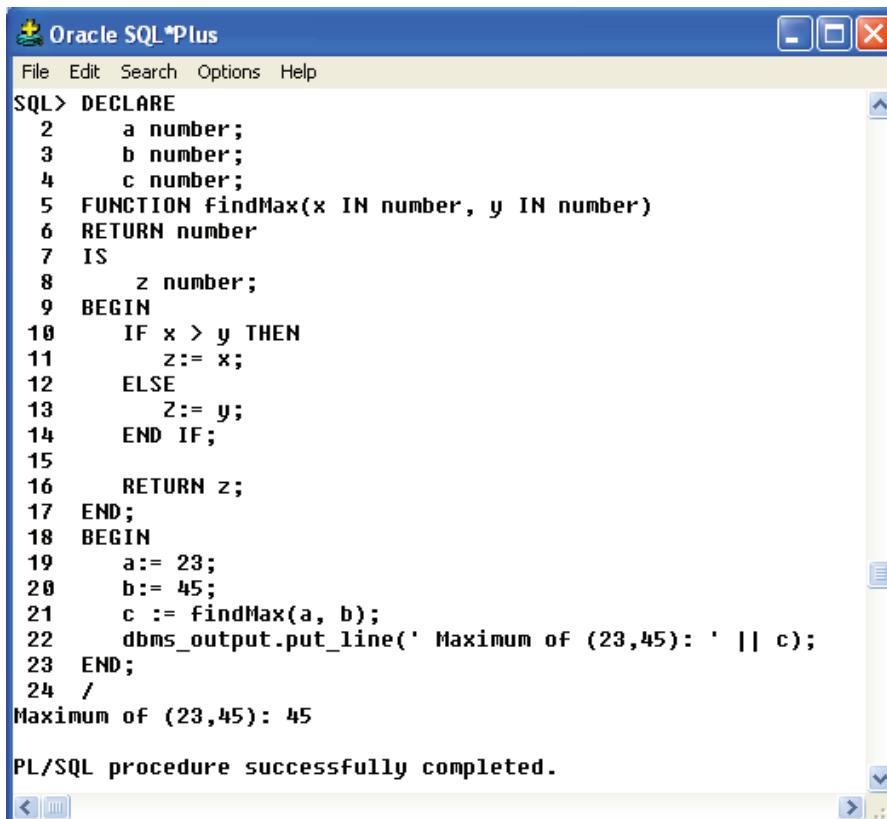
Fig. 14.35. PL/SQL Function Call.

Example: The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
    a number;
    b number;
    c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
    IF x > y THEN
```

```
        Z:= x;
    ELSE
        Z:= y;
    ENDIF;
    RETURN z;
END;
BEGIN
    a:=23;
    b:=45;
    c := findMax(a, b);
    dbms_output.put_line(' Maximum of (23,45): '|| c);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following PL/SQL code:

```
SQL> DECLARE
  2      a number;
  3      b number;
  4      c number;
  5  FUNCTION FindMax(x IN number, y IN number)
  6  RETURN number
  7  IS
  8      z number;
  9  BEGIN
10      IF x > y THEN
11          z:= x;
12      ELSE
13          z:= y;
14      END IF;
15
16      RETURN z;
17  END;
18  BEGIN
19      a:= 23;
20      b:= 45;
21      c := findMax(a, b);
22      dbms_output.put_line(' Maximum of (23,45): ' || c);
23  END;
24 /
```

Below the code, the output is shown:

```
Maximum of (23,45): 45
PL/SQL procedure successfully completed.
```

Fig. 14.36. PL/SQL Function definition, declaration and call.

14.9.2.1 PL/SQL Recursive Functions

A program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as:

$$\begin{aligned} n! &= n \cdot (n-1)! \\ &= n \cdot (n-1) \cdot (n-2)! \\ &\dots \\ &= n \cdot (n-1) \cdot (n-2) \cdot (n-3) \dots 1 \end{aligned}$$

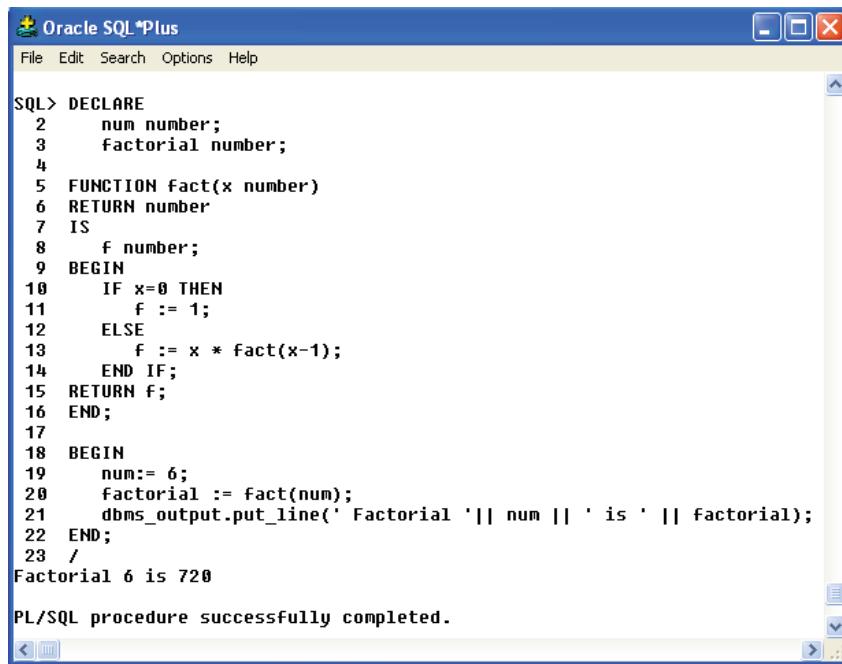
The following program calculates the factorial of a given number by calling itself recursively:

```

DECLARE
    num number;
    factorial number;
FUNCTION fact(x number)
RETURN number
IS
    f number;
BEGIN
    IF x=0 THEN
        f :=1;
    ELSE
        f := x * fact(x-1);
    END IF;
    RETURN f;
END;
BEGIN
    num:=6;
    factorial := fact(num);
    dbms_output.put_line(' Factorial ''|| num ||' is '|| factorial);
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



```

SQL> DECLARE
 2      num number;
 3      factorial number;
 4
 5  FUNCTION fact(x number)
 6  RETURN number
 7  IS
 8      f number;
 9  BEGIN
10      IF x=0 THEN
11          f := 1;
12      ELSE
13          f := x * fact(x-1);
14      END IF;
15      RETURN f;
16  END;
17
18  BEGIN
19      num:= 6;
20      factorial := fact(num);
21      dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
22  END;
23 /
Factorial 6 is 720
PL/SQL procedure successfully completed.

```

Fig. 14.37. PL/SQL Recursive Function.

14.9.2.2 Parameters

Parameters are the link between a subprogram code and the code calling the subprogram. It is absolutely necessary to know more about parameters, their modes, their default values, and how subprograms can be called without passing all the parameters. Parameter modes define the behavior of formal parameters of subprograms. There are three types of parameter modes:

1. IN
2. OUT
3. IN/OUT

IN Mode

IN mode is used to pass values to the called subprogram. In short this is an input to the called subprogram. Inside the called subprogram, an IN parameter acts like a constant and hence it cannot be assigned a new value. The IN parameter in actual parameter list can be a constant, literal, initialized variable, or an expression. IN parameters can be initialized to default values, which is not the case with IN/OUT or OUT parameters. It is important to note that IN mode is the default mode of the formal parameters. If we do not specify the mode of a formal parameter it will be treated as an IN mode parameter.

OUT Mode

An OUT parameter returns a value back to the caller subprogram. Inside the subprogram, the parameter specified with OUT mode acts just like any locally declared variable. Its value can be

changed or referenced in expressions, just like any other local variables. The points to be noted for an OUT parameter are:

- The parameter corresponding to OUT parameter must be a variable; it cannot be a constant or literal.
- Formal OUT parameters are by default initialized to NULL, so we cannot constraint the formal OUT parameters by NOT NULL constraint.
- The parameter corresponding to OUT parameter can have a value before a call to subprogram, but the value is lost as soon as a call is made to the subprogram.

IN/OUT

An IN/OUT parameter performs the duty of both IN parameter as well as OUT parameter. It first passes input value to the called subprogram and then inside subprogram it receives a new value which will be assigned finally to the actual parameter. In short, inside the called subprogram, the IN/OUT parameter behaves just like an initialized local variable. Like OUT parameter, the parameter in the actual argument list that corresponds to IN/OUT parameter, must be a variable, it cannot be a constant or an expression. If the subprogram exits successfully, PL/SQL assigns value to actual parameters, however, if the subprogram exits with unhandled exception, PL/SQL does not assign values to actual parameters.

14.10 PL/SQL CURSORS

For processing an SQL statement Oracle creates a memory area, known as context area. This context area contains all information needed for processing the statement. A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows returned by a SQL statement. The set of rows the cursor holds is referred to as the active set. There are two types of cursors:

- Implicit cursors
- Explicit cursors

14.10.1 Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it. Whenever a DML statement is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, the most commonly used recent implicit cursor is the **SQL cursor**, which contains attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The description of the attributes is as follows:

%FOUND: Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND: It can be useful in reporting or processing when no data is affected. The logical

opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

%ISOPEN: %ISOPEN is used to determine if a cursor is already open. Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

%ROWCOUNT: This attribute is used to determine the number of rows that are processed by an SQL statement. It returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

SQL cursor attribute can be accessed using

sql%attribute_name

Example: Let us consider the following CUSTOMERS table.

The screenshot shows the Oracle SQL*Plus interface with a title bar 'Oracle SQL*Plus'. The menu bar includes 'File', 'Edit', 'Search', 'Options', and 'Help'. The main area displays the following SQL query and its results:

```
SQL> Select * from customers;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	3000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500

6 rows selected.

Fig. 14.38. Customer Table.

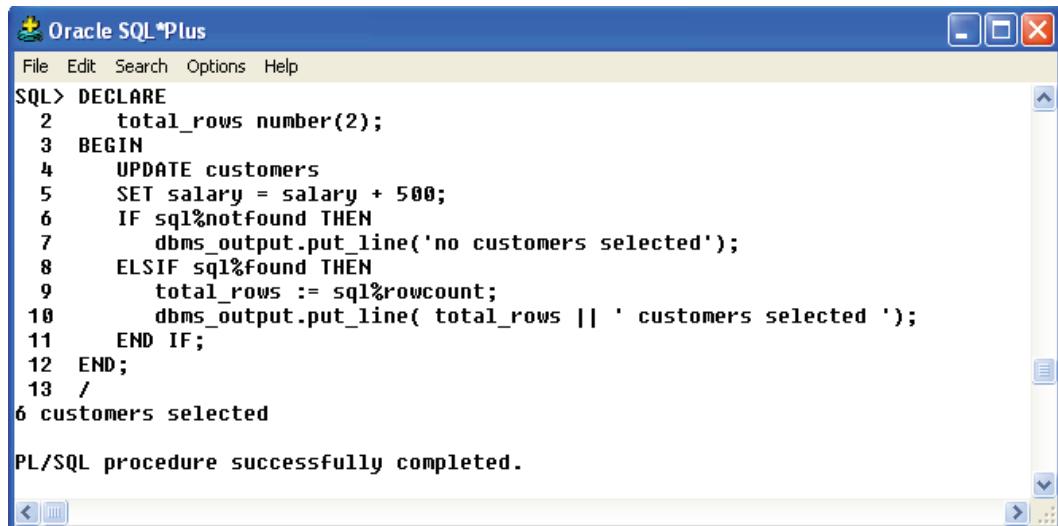
The following program would update the table and increase salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected:

```

DECLARE
    total_rows number(2);
BEGIN
    UPDATE customers
    SET salary = salary + 500;
    IF sql%notfound THEN
        dbms_output.put_line('no customers selected');
    ELSIF sql%found THEN
        total_rows := sql%rowcount;
        dbms_output.put_line( total_rows || ' customers selected ' );
    END IF;
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The command line displays the following PL/SQL code:

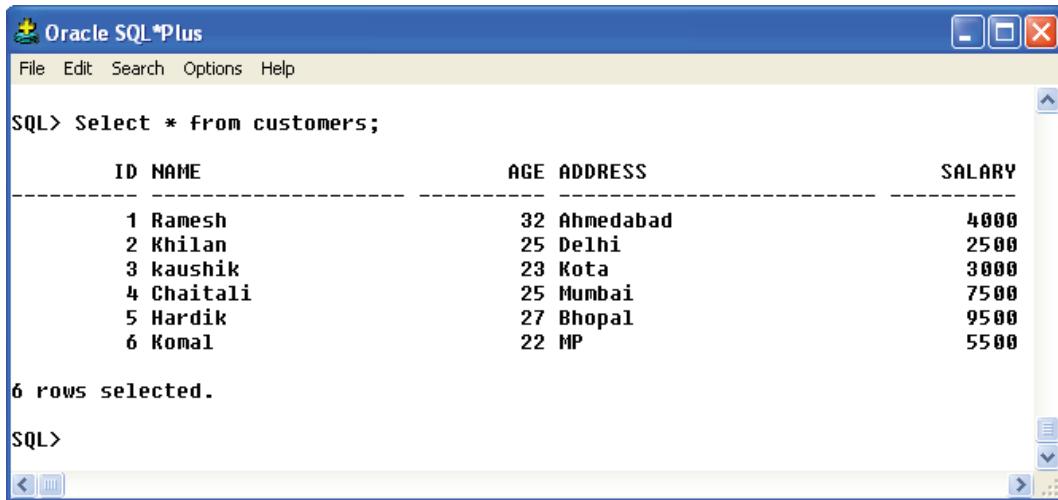
```
SQL> DECLARE
 2      total_rows number(2);
 3  BEGIN
 4      UPDATE customers
 5      SET salary = salary + 500;
 6      IF sql%notfound THEN
 7          dbms_output.put_line('no customers selected');
 8      ELSIF sql%found THEN
 9          total_rows := sql%rowcount;
10      dbms_output.put_line( total_rows || ' customers selected ');
11  END IF;
12 END;
13 /
6 customers selected

PL/SQL procedure successfully completed.
```

The output window shows the message "6 customers selected" and "PL/SQL procedure successfully completed.".

Fig. 14.39. Program illustrating the use of *SQL%ROWCOUNT* attribute.

If the records in customers table are checked, it list out the rows updated:



The screenshot shows the Oracle SQL*Plus interface. The command line displays the following SQL query:

```
SQL> Select * from customers;
```

The output window displays the updated data in the customers table:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	4000
2	Khilan	25	Delhi	2500
3	Kaushik	23	Kota	3000
4	Chaitali	25	Mumbai	7500
5	Hardik	27	Bhopal	9500
6	Komal	22	MP	5500

The message "6 rows selected." is shown below the table.

Fig. 14.40. Updated Customer Table.

14.10.2 Explicit Cursors

Explicit cursors are programmer defined cursors to gain more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

Syntax:

CURSOR cursor_name IS select_statement;

Steps to create Explicit Cursor:

- Declaring the cursor for initializing in the memory
- Opening the cursor for allocating memory
- Fetching the cursor for retrieving data
- Closing the cursor to release allocated memory

Declaring the Cursor: Declaring the cursor defines the cursor with a name and the associated SELECT statement.

Syntax:

```
CURSOR <cursor name>IS  
SELECT . . .
```

For example:

```
CURSOR c_customers IS  
SELECT id, name, address FROM customers;
```

Opening the Cursor: Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it.

Syntax:

```
OPEN <cursor name>;
```

Example:

```
OPEN c_customers;
```

Fetching the Cursor: Fetching the cursor involves accessing one row at a time.

Syntax:

```
FETCH <cursor name>INTO <variable name>, <variable name>. . .
```

Example:

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor: Closing the cursor means releasing the allocated memory.

Syntax:

```
CLOSE <cursor name>;
```

Example:

```
CLOSE c_customers;
```

Example Explicit Cursor:

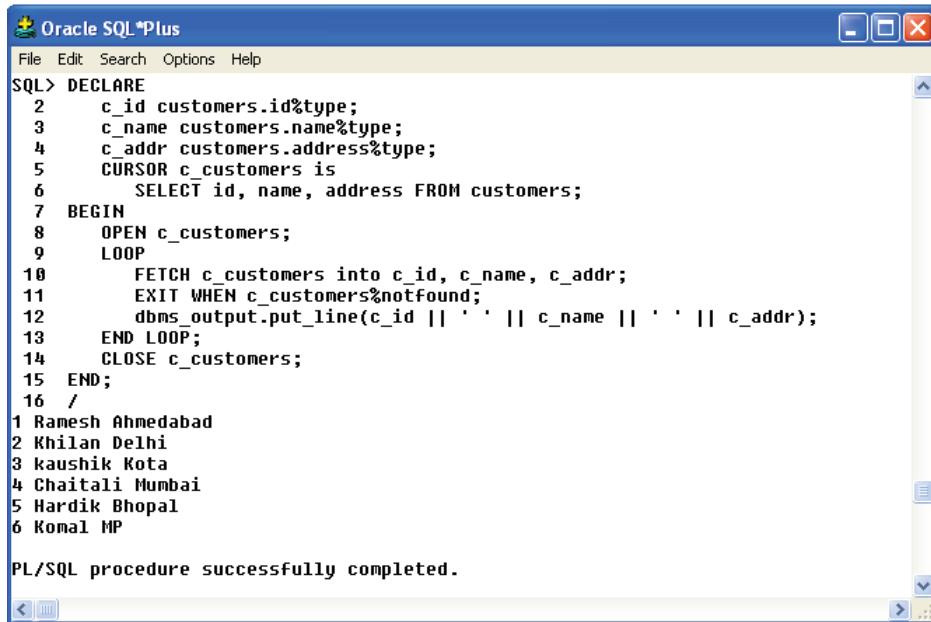
```
DECLARE  
    c_id customers.id%type;  
    c_name customers.name%type;  
    c_addr customers.address%type;  
    CURSOR c_customers is
```

```

SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
    FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:



```

Oracle SQL*Plus
File Edit Search Options Help
SQL> DECLARE
 2   c_id customers.id%type;
 3   c_name customers.name%type;
 4   c_addr customers.address%type;
 5   CURSOR c_customers IS
 6     SELECT id, name, address FROM customers;
 7   BEGIN
 8     OPEN c_customers;
 9     LOOP
10       FETCH c_customers into c_id, c_name, c_addr;
11       EXIT WHEN c_customers%notfound;
12       dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
13     END LOOP;
14     CLOSE c_customers;
15   END;
16 /
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP
PL/SQL procedure successfully completed.

```

Fig. 14.41. Explicit Cursor – Example.

14.11 PL/SQL PACKAGES

A package is defined as a collection of related program objects such as procedures, functions, and associated cursors and variables together as a unit in the database. A package contains two separate parts:

- (i) The package specification
- (ii) The package body.

The package specification and package body are compiled separately and stored in the data dictionary as two separate objects. The package body is optional and need not be created if the package specification does not contain any procedures or functions. Applications or users can call packaged procedures and functions explicitly similar to standalone procedures and functions.

Advantages of Packages

1. Stored packages allow us to group related stored procedures, variables, and data types stored unit in the database. This provides orderliness during the development process.
2. Grouping of related procedures, functions, etc. in a package also makes privilege management easier. Granting the privilege to use a package makes all components of the package accessible to the grantee.
3. Package helps in achieving data abstraction.
4. An entire package is loaded into memory when a procedure within the package is called for the first time. This results in faster and efficient operation of programs.
5. Packages provide better performance than stored procedures and functions because public package variables persist in memory for the duration of a session. So they can be accessed by all procedures and functions that try to access them.
6. Packages allow overloading of its member modules. More than one function in a package can be of same name. The functions are differentiated, depending upon the type and number of parameters it takes.

14.11.1 Creating a Package

A package consists of package specification and package body. Hence creation of a package involves creation of the package specification and then creation of the package body.

The package specification is declared using the CREATE PACKAGE command.

Syntax:

CREATE[OR REPLACE] PACKAGE <package name>

[AS/IS]

PL/SQL package specification

All the procedures, sub programs, cursors declared in the CREATE PACKAGE command are described and implemented fully in the package body.

Syntax:

CREATE[OR REPLACE] PACKAGE BODY <package name>

[AS/IS]

PL/SQL package body

Member functions and procedures can be declared in a package and can be made public or private member using the keywords public and private. Use of all the private members of the package is restricted within the package while the public members of the package can be accessed and used outside the package.

14.11.2 Referencing Package Subprograms

Once the package body is created with all members as public, we can access them from outside the program. To access these members outside the packages we have to use the dot operator, by prefixing the package object with the package name.

Syntax:

<PACKAGE NAME>.<VARIABLE NAME>

To reference procedures

Syntax:

EXECUTE <package name>.<procedure name(variables)>;

But the package member can be referenced by only its name if we reference the member within the package. Moreover the EXECUTE command is not required if procedures are called within PL/SQL. Functions can be referenced similar to that of procedures from outside the package using the dot operator.

14.11.3 Removing a Package

A package can be dropped from the database just like any other table or database object. To drop a package a user either must own the package or he should have DROP ANY PACKAGE privilege.

Syntax:

DROP PACKAGE <PACKAGE NAME>;

14.11.4 Parts of Package

A Package has two parts. They are:

- Package specification
- Package body

Package Specification

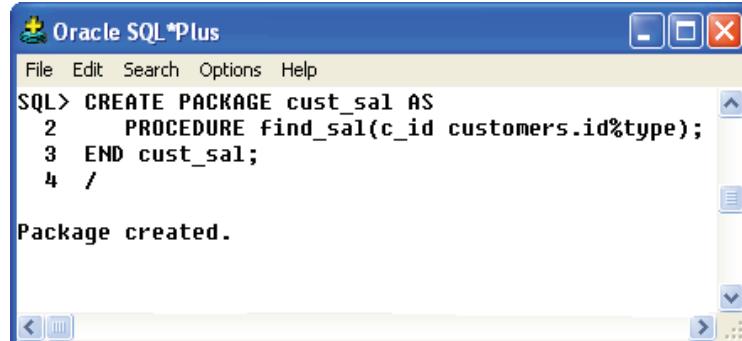
The specification declares the types, variables, constants, exceptions, cursors, and subprograms that are public, and thus available for use outside the package. In case, if there is only types, constants, exception, or variables in the package specification declaration, then there is no need for the package body because package specification is sufficient for them. Package body is required when there are subprograms like cursors, functions, etc.

In other words, it contains all information about the content of the package, but excludes the code for the subprograms. All objects placed in the specification are called public objects. Any subprogram that is not in the package specification but coded in the package body is called a private object.

The following example code shows a package specification having a single procedure.

```
CREATE PACKAGE cust_sal AS
  PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The title bar says "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window contains the following SQL code:
SQL> CREATE PACKAGE cust_sal AS
2 PROCEDURE find_sal(c_id customers.id%TYPE);
3 END cust_sal;
4 /

Package created.

Fig. 14.42. Package Specification.

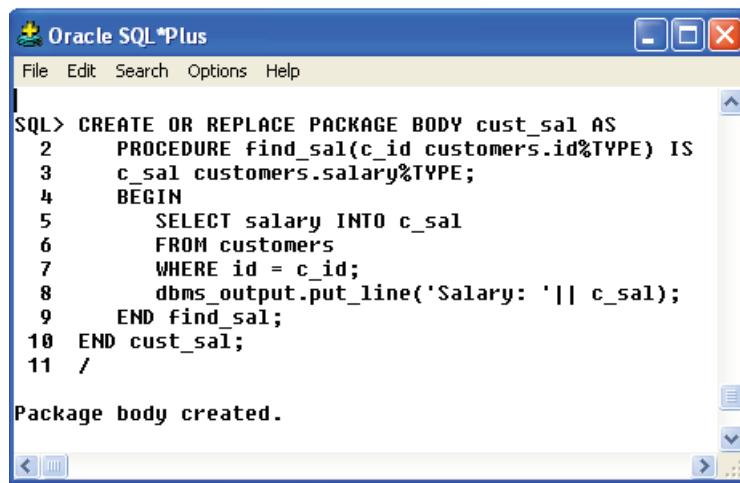
Package Body

The package body fully defines subprograms such as cursors, functions, and procedures. All the private declarations of the package are included in the package body. It implements the package specification. A package specification and the package body are stored separately in the database. This allows calling objects to depend on the specification only, not on both. This separation enables to change the definition of program object in the package body without causing Oracle to interfere with other objects that call or reference the program object. Oracle invalidates the calling object if the package specification is changed.

The CREATE PACKAGE BODY Statement is used for creating the package body. The following example code shows the package body declaration for the *cust_sal* package created above. The customers table used in the package is already created.

```
CREATEOR REPLACE PACKAGE BODY cust_sal AS  
  PROCEDURE find_sal(c_id customers.id%TYPE)IS  
    c_sal customers.salary%TYPE;  
  BEGIN  
    SELECT salary INTO c_sal  
    FROM customers  
    WHERE id = c_id;  
    dbms_output.put_line('Salary: '|| c_sal);  
  END find_sal;  
END cust_sal;  
/
```

When the above code is executed at SQL prompt, it produces the following result:



The screenshot shows the Oracle SQL*Plus interface. The title bar reads "Oracle SQL*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following PL/SQL code:

```

SQL> CREATE OR REPLACE PACKAGE BODY cust_sal AS
  2      PROCEDURE find_sal(c_id customers.id%TYPE) IS
  3          c_sal customers.salary%TYPE;
  4      BEGIN
  5          SELECT salary INTO c_sal
  6          FROM customers
  7          WHERE id = c_id;
  8          dbms_output.put_line('Salary: '|| c_sal);
  9      END find_sal;
10  END cust_sal;
11 /

```

Below the code, the message "Package body created." is displayed. The bottom of the window shows standard SQL*Plus navigation icons.

Fig. 14.43. Package Body.

14.12 PL/SQL EXCEPTION

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

In absence of exceptions, unless the error checking is disabled, a program will exit abnormally whenever some runtime error occurs. But with exceptions, if some error occurs, the exceptional handler will report an appropriate error/warning message and will continue the execution of program and finally come out of the program successfully. An exception handler is a code block in memory that attempts to resolve the current exception condition. Each exception handler has codes attached to it that attempts to resolve the exception condition.

Syntax:

EXCEPTION
WHEN exception name THEN
User defined action to be carried out.

Detailed Syntax:

DECLARE
<declarations section>
BEGIN
<executable command(s)>
EXCEPTION
<exception handling goes here >

```
WHEN exception1 THEN  
    exception1-handling-statements  
WHEN exception2 THEN  
    exception2-handling-statements  
WHEN exception3 THEN  
    exception3-handling-statements  
.....  
WHEN others THEN  
    exception3-handling-statements  
END;
```

Exceptions can be internally defined or user defined. Internally defined exceptions are raised implicitly by the run-time system. User-defined exceptions must be raised explicitly by RAISE statements, which can also raise internally defined exceptions. Raised exceptions are handled by separate routines called exception handlers. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

Advantages of Using Exceptions

1. Control over abnormal exits of executing programs helps in maintaining the reliability of the application.
2. Meaningful messages can be flagged so that the developer can become aware of error and warning conditions and act upon them.
3. In traditional error checking system, if same error is to be checked at several places, you are required to code the same error check at all those places. But with exception handling technique, we will write the exception for that particular error only once in the entire code. Whenever that type error occurs at any place in code, the exceptional handler will automatically raise the defined exception.
4. PL/SQL exceptions can be coded and isolated like procedures and functions. This improves the overall readability of a PL/SQL program.

14.12.1 Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command RAISE.

Syntax:

```
DECLARE  
    exception_name EXCEPTION;  
BEGIN  
    IF condition THEN  
        RAISE exception_name;  
    END IF;
```

```
EXCEPTION
  WHEN exception_name THEN
    statement;
  END;
```

14.12.2 User-defined Exceptions

PL/SQL allows users to define their own exceptions according to the need of their program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

Syntax:

```
DECLARE
  my-exception EXCEPTION;
```

14.12.3 Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. Descriptions for predefined exceptions are as follows:

- **ACCESS_INTO_NULL:** It is raised when a null object is automatically assigned a value.
- **CASE_NOT_FOUND:** It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
- **COLLECTION_IS_NULL:** It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
- **DUP_VAL_ON_INDEX:** It is raised when duplicate values are attempted to be stored in a column with unique index.
- **INVALID_CURSOR:** It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
- **INVALID_NUMBER:** It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
- **LOGIN_DENIED:** It is raised when a program attempts to log on to the database with an invalid username or password.
- **NO_DATA_FOUND:** It is raised when a SELECT INTO statement returns no rows.
- **NOT_LOGGED_ON:** It is raised when a database call is issued without being connected to the database.
- **PROGRAM_ERROR:** It is raised when PL/SQL has an internal problem.
- **ROWTYPE_MISMATCH:** It is raised when a cursor fetches value in a variable having incompatible data type.
- **SELF_IS_NULL:** It is raised when a member method is invoked, but the instance of the object type was not initialized.
- **STORAGE_ERROR:** It is raised when PL/SQL run out of memory or memory was corrupted.

- **TOO_MANY_ROWS:** It is raised when a SELECT INTO statement returns more than one row.
- **VALUE_ERROR:** It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs.
- **ZERO_DIVIDE:** It is raised when an attempt is made to divide a number by zero.

REVIEW QUESTIONS

1. List out the advantages of using PL/SQL?
2. Explain the structure of PL/SQL programming with suitable example.
3. Explain the lexical unit in PL/SQL.
4. What is Delimiter?
5. Explain the elements of PL/SQL.
6. With suitable example explain how to create user defined sub types in PL/SQL.
7. How to declare variable in PL/SQL? Explain how a query result is assigned to a variable?
8. Give the syntax for single-line comments and multiline comments?
9. List and explain the different types of operators supported by PL/SQL.
10. With suitable example explain the case statement in PL/SQL.
11. Explain the different String functions and operations in PL/SQL.
12. Write a program to illustrate the concept of varray in PL/SQL.
13. Explain the concept of Subprogram.
14. Mention the facilities available for iterating the statements in PL/SQL?
15. What is cursor and mention its types in Oracle?
16. Mention some implicit and explicit cursor attributes and explain the same.
17. What is Procedure in PL/SQL?
18. Mention any four advantages of procedures and function?
19. What are Packages?
20. Mention how exception handling is done in Oracle?



Chapter 15

Data Mining & Data Warehousing

15.1 INTRODUCTION

Data mining has become very famous in the recent years in information industry due to the availability of large amount of data and the need to transfer this data into useful information and knowledge.

15.1.1 What is Data Mining?

Data mining refers to extracting or mining knowledge from large amounts of data.

There are many other terms related to data mining, such as knowledge mining, knowledge extraction, data/pattern analysis, data archaeology, and data dredging.

15.1.2 Essential Step in the Process of Knowledge Discovery in Databases

Knowledge discovery as a process is depicted in following figure and consists of an iterative sequence of the following steps:

1. **Data Cleaning:** to remove noise or irrelevant data
2. **Data Integration:** where multiple data sources may be combined
3. **Data Selection:** where data relevant to the analysis task are retrieved from the database
4. **Data Transformation:** where data are transformed or consolidated into forms appropriate for mining by performing summary or aggregation operations
5. **Data Mining:** an essential process where intelligent methods are applied in order to extract data patterns
6. **Pattern Evaluation:** to identify the truly interesting patterns representing knowledge based on some interestingness measures
7. **Knowledge Presentation:** where visualization and knowledge representation techniques are used to present the mined knowledge to the user.

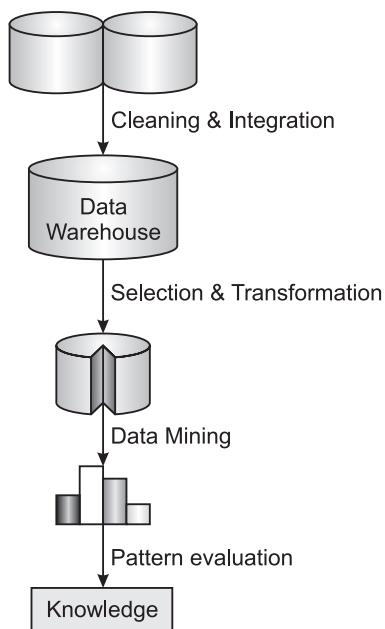


Fig. 15.1. Steps in KDD process.

15.1.3 Architecture of a Typical Data Mining System

Data mining is the process of discovering interesting knowledge from large amounts of data stored either in databases, data warehouses, or other information repositories. Major components of data mining system are as follows:

1. **A database, data warehouse, or other information repository**, which consists of the set of databases, data warehouses, spreadsheets, or other kinds of information repositories.
2. **A database or data warehouse server** which fetches the relevant data based on users' data mining requests.
3. **A knowledge base** that contains the domain knowledge used to guide the search or to evaluate the interestingness of resulting patterns.
4. **A data mining engine**, which consists of a set of functional modules for tasks such as classification, association, classification, cluster analysis, and evolution and deviation analysis.
5. **A pattern evaluation** module that works in phase with the data mining modules by employing interestingness measures to search.
6. **A graphical user interface** that allows the user an interactive approach to the data mining system.

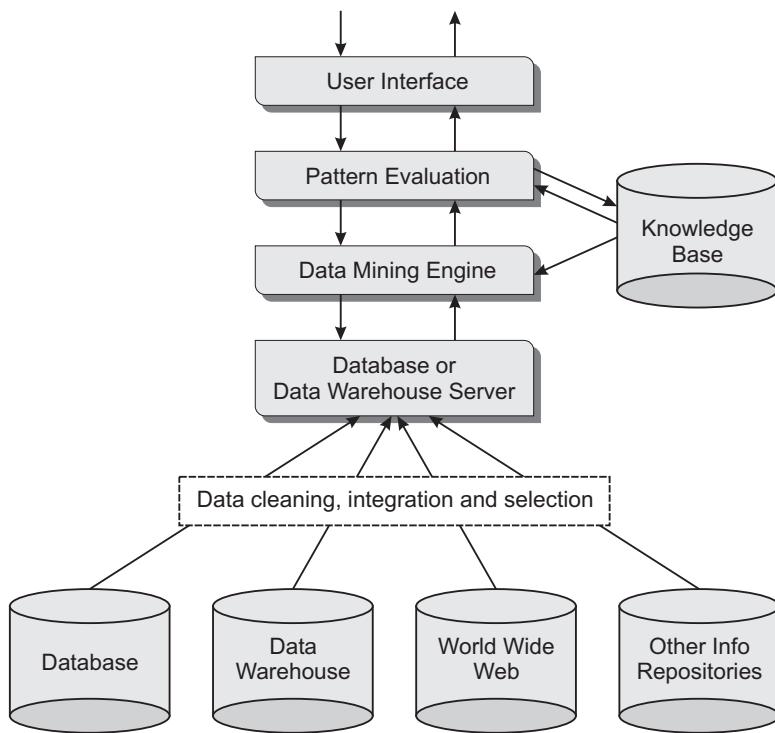


Fig. 15.2. Architecture of a typical data mining system.

15.2 DATA IN DATA MINING

Data mining should be applicable to any kind of information repository. This includes:

- Relational databases
- Data warehouses
- Transactional databases
- Advanced database systems
- Flat files
- World-Wide Web.

Advanced database systems include:

- object-oriented
- object-relational databases
- special application-oriented databases, such as spatial databases, time-series databases, text databases, and multimedia databases.

Flat files: Flat files are actually the most common data source for data mining algorithms, especially at the research level. Flat files are simple data files in text or binary format with a structure known by the data mining algorithm to be applied. The data in these files can be transactions, time-series data, scientific measurements, etc.

Relational Databases: A relational database consists of a set of tables containing either values of entity attributes, or values of attributes from entity relationships. Tables have columns and rows, where columns represent attributes and rows represent tuples. A tuple in a relational table corresponds to either an object or a relationship between objects. The most commonly used query language for relational database is SQL.

Data warehouses: A data warehouse is a repository of information collected from multiple sources, stored under a unified schema, and which usually resides at a single site. Data warehouses are constructed via a process of data cleansing, data transformation, data integration, data loading, and periodic data refreshing.

Transactional databases: A transactional database consists of a flat file where each record represents a transaction. A transaction typically includes a unique transaction identity number, and a list of the items making up the transaction.

15.2.1 Advanced Database Systems and Advanced Database Applications

- An **objected-oriented database** is designed based on the object-oriented programming paradigm where data are a large number of objects organized into classes and class hierarchies. Each entity in the database is considered as an object. The object contains a set of variables that describe the object, a set of messages that the object can use to communicate with other objects or with the rest of the database system and a set of methods where each method holds the code to implement a message.
- A **spatial database and spatio temporal databases:** **spatial databases** contain spatial-related data which may be represented in the form of raster or vector data. Raster data consists of n-dimensional bit maps or pixel maps, and vector data are represented by lines, points, polygons or other kinds of processed primitives, some examples of spatial databases include geographical (map) databases. **Spatio temporal database**, are spatial database that stores spatial objects that changes with time. E.g. Bioterrorist attack
- **Temporal, sequence and Time-Series Databases:** **Temporal databases**, stores relational data that include time related attributes. A **sequence database**, stores sequence of ordered events. **Time-series databases**, stores sequence of values or events obtained over repeated measurements of time.
- **A text database** is a database that contains text documents or other word descriptions in the form of long sentences or paragraphs, such as product specifications, error or bug reports, warning messages, summary reports, notes, or other documents.
- **A multimedia database** stores images, audio, and video data, and is used in applications such as picture content-based retrieval, voice-mail systems, video-on-demand systems, the World Wide Web, and speech-based user interfaces.
- **The World-Wide Web** provides rich, world-wide, on-line information services, where data objects are linked together to facilitate interactive access. Some examples of distributed information services associated with the World-Wide Web include America Online, Yahoo!, AltaVista, and Prodigy.

- A **heterogeneous database** consists of a set of interconnected, autonomous component databases.
- A **legacy database** is a group of heterogeneous databases that combines different kinds of data systems, such as relational or object-oriented databases, hierarchical databases, network databases, spreadsheets, multimedia databases, or file systems.

15.3 DATA MINING FUNCTIONALITIES

Data mining functionalities are used to specify the kind of patterns to be found in data mining tasks. In general, data mining tasks can be classified into two categories:

- Descriptive
- Predictive

Descriptive mining tasks characterize the general properties of the data in the database.

Predictive mining tasks perform inference on the current data in order to make predictions.

1. Concept/class Description: Characterization and Discrimination: Data can be associated with classes or concepts. It describes a given set of data in a concise and summarative manner, presenting interesting general properties of the data. These descriptions can be derived via

- (i) Data Characterization
- (ii) Data Discrimination

(i) Data characterization: It is a summarization of the general characteristics or features of a target class of data. The results are presented in pie chart, bar chart, cures, multi dimensional cubes or multi dimensional tables, etc. The resulting descriptions can also be presented as generalized relations, or in rule form called characteristic rules.

(ii) Data Discrimination is a comparison of the general features of target class data objects with the general features of objects from one or a set of contrasting classes. Discrimination descriptions expressed in rule form are referred to as discriminant rules.

2. Mining Frequent Patterns, Associations, and Correlations: Frequent patterns are patterns that occur frequently in data. There are many kinds of frequent patterns, including itemsets, subsequences, and substructures.

A **frequent itemset** is a set of items that frequently appear together in a transactional data set, such as Computer and Software.

Mining frequent patterns leads to the discovery of interesting associations and correlations within data.

3. Classification and Prediction: Classification can be defined as the process of finding a model (or function). This derived model is based on the analysis of a set of training data.

A classification model can be represented in various forms, such as

- (i) **IF-THEN rules,**

student (class , "undergraduate") AND concentration (level, "high") ==> class A

student (class , "undergraduate") AND concentration (level,"low") ==> class B

student (class , "post graduate") ==> class C

(ii) Decision tree

A **decision tree** is a flow-chart-like tree structure, where each node denotes a test on an attribute value, each branch represents an outcome of the test, and tree leaves represent classes or class distributions. Decision trees can easily be converted to classification rules.

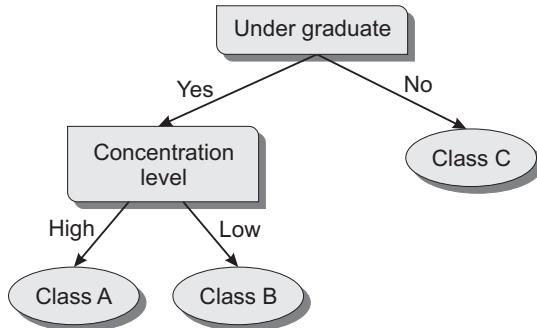


Fig. 15.3. Decision tree.

(iii) Neural network

A **neural network**, when used for classification, is typically a collection of neuron-like processing units with weighted connections between the units.

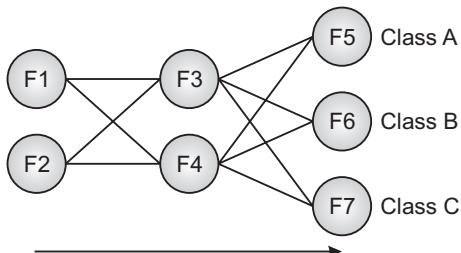


Fig. 15.4. Neural network.

Prediction: Find some missing or unavailable data values rather than class labels are referred to as prediction.

4. Clustering Analysis: Clustering analyzes data objects without consulting a known class label. The objects are clustered or grouped based on the principle of maximizing the intraclass similarity and minimizing the interclass similarity. Each cluster that is formed can be viewed as a class of objects.

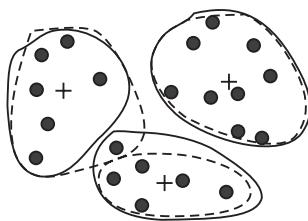


Fig. 15.5. Cluster.

5. Outlier Analysis: A database may contain data objects that do not comply with general model of data. These data objects are outliers.

The data objects which do not fall within the cluster will be called as outlier data objects. Noisy data or exceptional data are also called as outlier data. The analysis of outlier data is referred to as outlier mining.

6. Data Evolution Analysis describes and models regularities or trends for objects whose behavior changes over time.

7. Correlation Analysis: Correlation analysis is a technique used to measure the association between two variables. A **correlation coefficient (r)** is a statistic used for measuring the strength of a supposed linear association between two variables. Correlations range from -1.0 to $+1.0$ in value.

A correlation coefficient of 1.0 indicates a perfect positive relationship in which high values of one variable are related perfectly to high values in the other variable, and conversely, low values on one variable are perfectly related to low values on the other variable.

A correlation coefficient of 0.0 indicates no relationship between the two variables. That is, one cannot use the scores on one variable to tell anything about the scores on the second variable.

A correlation coefficient of -1.0 indicates a perfect negative relationship in which high values of one variable are related perfectly to low values in the other variables, and conversely, low values in one variable are perfectly related to high values on the other variable.

15.4 CLASSIFICATION OF DATA MINING SYSTEMS

There are many data mining systems available or being developed. Data mining systems can be categorized according to various criteria among other classification are the following:

- **Classification according to the type of data source mined:** This classification categorizes data mining systems according to the type of data handled such as spatial data, multimedia data, time-series data, text data, World Wide Web, etc.
- **Classification according to the data model drawn on:** This classification categorizes data mining systems based on the data model involved such as relational database, object-oriented database, data warehouse, transactional, etc.
- **Classification according to the kind of knowledge discovered:** this classification categorizes data mining systems based on the kind of knowledge discovered or data mining functionalities, such as characterization, discrimination, association, classification, clustering, etc. Some systems tend to be comprehensive systems offering several data mining functionalities together.
- **Classification according to mining techniques used:** Data mining systems employ and provide different techniques. This classification categorizes data mining systems according to the data analysis approach used such as machine learning, neural networks, genetic algorithms, statistics, visualization, database oriented or data warehouse-oriented, etc.

The classification can also take into account the degree of user interaction involved in the data mining process such as query-driven systems, interactive exploratory systems, or autonomous systems.

A comprehensive system would provide a wide variety of data mining techniques to fit different situations and options, and offer different degrees of user interaction.

15.5 PRIMITIVES FOR SPECIFYING A DATA MINING TASK

A data mining task can be specified in the form of a data mining query, which is input to the data mining system. A data mining query is defined in terms of data mining task primitives. These primitives allow the user to interactively communicate with the data mining system.

- **Task-relevant data:** This primitive specifies the data upon which mining is to be performed. It involves specifying the database and tables or data warehouse containing the relevant data, conditions for selecting the relevant data, the relevant attributes or dimensions for exploration, and instructions regarding the ordering or grouping of the data retrieved.
- **Knowledge type to be mined:** This primitive specifies the specific data mining function to be performed, such as characterization, discrimination, association, classification, clustering, or evolution analysis. As well, the user can be more specific and provide pattern templates that all discovered patterns must match. These templates or Meta patterns, can be used to guide the discovery process.
- **Background knowledge:** This primitive allows users to specify knowledge they have about the domain to be mined. Such knowledge can be used to guide the knowledge discovery process and evaluate the patterns that are found.
- **Pattern interestingness measure:** This primitive allows users to specify functions that are used to separate uninteresting patterns from knowledge and may be used to guide the mining process, as well as to evaluate the discovered patterns. This allows the user to confine the number of uninteresting patterns returned by the process, as a data mining process may generate a large number of patterns. Interestingness measures can be specified for such pattern characteristics as simplicity, certainty, utility and novelty.
- **Visualization of discovered patterns:** This primitive refers to the form in which discovered patterns are to be displayed. In order to make data mining effective in conveying knowledge to users, data mining systems should be able to display the discovered patterns in multiple forms such as rules, tables, cross tabs, pie or bar charts, decision trees, cubes or other visual representations.

15.6 INTEGRATION OF A DATA MINING SYSTEM WITH A DATA WAREHOUSE

Good system architecture will facilitate the data mining system to make best use of the software environment.

If a data mining (DM) system works as a stand-alone system or is embedded in an application program, there are no database (DB) or data warehouse (DW) systems with which it has to communicate. This simple scheme is called **no coupling**.

When a DM system works in an environment that requires communication with other information system components, such as DB and DW systems, possible integration schemes is needed. The integration scheme include:

1. No coupling
2. Loose coupling
3. Semi tight coupling
4. Tight coupling.

1. No coupling: *No coupling* means that a DM system will not utilize any function of a DB or DW system. It may fetch data from a particular source (such as a file system), process data using some data mining algorithms, and then store the mining results in another file.

Few draw backs of No coupling system is as follows:

A DB system provides a great deal of flexibility and efficiency at storing, organizing, accessing, and processing data. Without using a DB/DW system, a DM system may spend a substantial amount of time finding, collecting, cleaning, and transforming data. In DB and/or DW systems, data tend to be well organized, indexed, cleaned, integrated, or consolidated, so that finding the task-relevant, high-quality data becomes an easy task.

There are many tested, scalable algorithms and data structures implemented in DB and DW systems. It is feasible to realize efficient, scalable implementations using such systems. Moreover, most data have been or will be stored in DB/DW systems. Without any coupling of such systems, a DM system will need to use other tools to extract data, making it difficult to integrate such a system into an information processing environment. Thus, no coupling represents a poor design.

2. Loose coupling: *Loose coupling* means that a DM system will use some facilities of a DB or DW system. Fetching data from a data repository managed by these systems, performing data mining, and then storing the mining results either in a file or in a designated place in a database or data warehouse.

Loose coupling is better than no coupling because it can fetch any portion of data stored in databases or data warehouses by using query processing, indexing, and other system facilities. It incurs some advantages of the flexibility, efficiency, and other features provided by such systems. However, many loosely coupled mining systems are main memory-based.

3. Semitight coupling: *Semitight coupling* means that besides linking a DM system to a DB/DW system, efficient implementations of a few essential data mining primitives can be provided in the DB/DW system. These primitives can include sorting, indexing, aggregation, histogram analysis, multi way join, and precomputation of some essential statistical measures, such as sum, count, max, min, standard deviation, and so on. Frequently used intermediate mining results can be precomputed and stored in the DB/DW system. This design will enhance the performance of a DM system.

4. Tight coupling: *Tight coupling* means that a DM system is smoothly integrated into the DB/DW system. The data mining subsystem is treated as one functional component of an information system. Data mining queries and functions are optimized based on mining query analysis, data structures, indexing schemes, and query processing methods of a DB or DW system.

This approach is highly desirable because it facilitates efficient implementations of data mining functions, high system performance, and an integrated information processing environment.

15.7 ISSUES IN DATA MINING

Major issues in data mining is regarding mining methodology, user interaction, performance, and diverse data types.

1. Mining methodology and user-interaction issues

- **Mining different kinds of knowledge in databases:** Since different users can be interested in different kinds of knowledge, data mining should cover a wide spectrum of data analysis and knowledge discovery tasks, including data characterization, discrimination, association, classification, clustering, trend and deviation analysis, and similarity analysis. These tasks may use the same database in different ways and require the development of numerous data mining techniques.
- **Interactive mining of knowledge at multiple levels of abstraction:** Since it is difficult to know exactly what can be discovered within a database, the data mining process should be interactive.
- **Incorporation of background knowledge:** Background knowledge, or information regarding the domain under study, may be used to guide the discovery patterns. Domain knowledge related to databases, such as integrity constraints and deduction rules, can help focus and speed up a data mining process, or judge the interestingness of discovered patterns.
- **Data mining query languages and ad-hoc data mining:** Knowledge in Relational query languages (such as SQL) required since it allow users to pose ad-hoc queries for data retrieval.
- **Presentation and visualization of data mining results:** Discovered knowledge should be expressed in high-level languages, visual representations, so that the knowledge can be easily understood and directly usable by humans.
- **Handling outlier or incomplete data:** The data stored in a database may reflect outliers: noise, exceptional cases, or incomplete data objects. These objects may confuse the analysis process, causing over fitting of the data to the knowledge model constructed. As a result, the accuracy of the discovered patterns can be poor. Data cleaning methods and data analysis methods which can handle outliers are required.
- **Pattern evaluation: refers to interestingness of pattern:** A data mining system can uncover thousands of patterns. Many of the patterns discovered may be uninteresting to the given user, representing common knowledge or lacking novelty. Several challenges remain regarding the development of techniques to assess the interestingness of discovered patterns,

2. Performance issues

These include efficiency, scalability, and parallelization of data mining algorithms.

- **Efficiency and scalability of data mining algorithms:** To effectively extract information from a huge amount of data in databases, data mining algorithms must be efficient and scalable.
- **Parallel, distributed, and incremental updating algorithms:** Such algorithms divide the data into partitions, which are processed in parallel. The results from the partitions are then merged.

3. Issues relating to the diversity of database types

- **Handling of relational and complex types of data:** Since relational databases and data warehouses are widely used, the development of efficient and effective data mining systems for such data is important.
- **Mining information from heterogeneous databases and global information systems:** Local and wide-area computer networks (such as the Internet) connect many sources of data, forming huge, distributed, and heterogeneous databases. The discovery of knowledge from different sources of structured, semi-structured, or unstructured data with diverse data semantics poses great challenges to data mining.

15.8 DATA PREPROCESSING

Data preprocessing describes the type of processing performed on raw data to prepare it for another processing procedure. Data preprocessing is a commonly used preliminary data mining practice. Data preprocessing transforms the data into a format that will be more easily and effectively processed for the purpose of the user

Why Data Preprocessing?

Data in the real world is dirty. It can be in incomplete, noisy and inconsistent form. These data need to be preprocessed in order to help improve the quality of the data, and quality of the mining results.

- If there is no quality data, then no quality mining results. The quality decision is always based on the quality data.
- If there is much irrelevant and redundant information or noisy and unreliable data, then knowledge discovery during the training phase is more difficult.

Incomplete data: lacking attribute values, lacking certain attributes of interest, or containing only aggregate data. e.g., occupation=“ ”.

Noisy data: containing errors or outliers data. e.g., Salary=“-10”

Inconsistent data: containing discrepancies in codes or names. e.g., Age=“42” Birthday=“03/07/1997”

- Incomplete data may come from
 - “Not applicable” data value when collected
 - Different considerations between the time when the data was collected and when it is analyzed.
 - Human/hardware/software problems
- Noisy data (incorrect values) may come from
 - Faulty data collection by instruments
 - Human or computer error at data entry
 - Errors in data transmission
- Inconsistent data may come from
 - Different data sources
 - Functional dependency violation (e.g., modify some linked data)

Major Tasks in Data Preprocessing

- **Data cleaning:** Fill in missing values, smooth noisy data, identify or remove outliers, and resolve inconsistencies
- **Data integration:** Integration of multiple databases, data cubes, or files
- **Data transformation:** Normalization and aggregation
- **Data reduction:** Obtains reduced representation in volume but produces the same or similar analytical results
- **Data discretization:** Part of data reduction but with particular importance, especially for numerical data

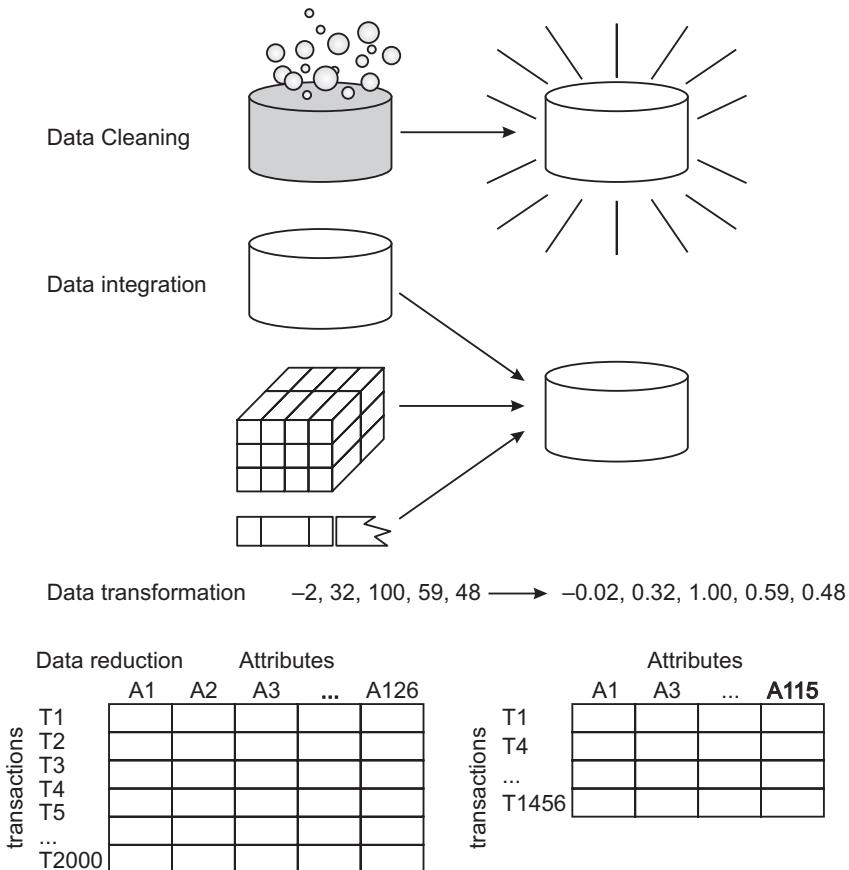


Fig. 15.6. Forms of Data Preprocessing.

15.8.1 Data Cleaning

Data cleaning routines attempt to fill in missing values, smooth out noise while identifying outliers, and correct inconsistencies in the data.

15.8.1.1 Missing Values

Many tuples have no recorded value for several attributes, such as customer *income*. The various methods for handling the problem of missing values in data tuples include:

- (a) **Ignoring the tuple:** This is usually done when the class label is missing. This method is not very effective unless the tuple contains several attributes with missing values. It is especially poor when the percentage of missing values per attribute varies considerably.
- (b) **Manually filling in the missing value:** In general, this approach is time-consuming and may not be a feasible for large data sets with many missing values, especially when the value to be filled in is not easily determined.
- (c) **Using a global constant to fill in the missing value:** Replace all missing attribute values by the same constant, such as a label like “Unknown,” or $-\infty$. If missing values are replaced by, say, “Unknown,” then the mining program may mistakenly consider that they form an interesting concept, since they all have a value in common — that of “Unknown.” Hence, although this method is simple, it is not recommended.
- (d) **Using the attribute mean for quantitative (numeric) values or attribute mode for categorical (nominal) values, for all samples belonging to the same class as the given tuple:** For example, if classifying customers according to credit risk, replace the missing value with the average income value for customers in the same credit risk category as that of the given tuple.
- (e) **Using the most probable value to fill in the missing value:** This may be determined with regression, inference-based tools using Bayesian formalism, or decision tree induction. For example, using the other customer attributes in your data set, you may construct a decision tree to predict the missing values for income.

15.8.1.2 Noisy Data

Noise is a random error or variance in a measured variable. Data smoothing technique is used for removing such noisy data.

Several Data smoothing techniques:

1. **Binning methods:** Binning methods smooth a sorted data value by consulting the neighborhood", or values around it. The sorted values are distributed into a number of 'buckets', or bins. Because binning methods consult the neighborhood of values, they perform local smoothing.

In this technique,

- (i) First sort the data
- (ii) Then place the sorted list partitioned into equi-depth of bins.
- (iii) Then one can smooth by bin means, smooth by bin median, smooth by bin boundaries, etc.
 - (a) **Smoothing by bin means:** Each value in the bin is replaced by the mean value of the bin.
 - (b) **Smoothing by bin medians:** Each value in the bin is replaced by the bin median.

- (c) **Smoothing by boundaries:** The min and max values of a bin are identified as the bin boundaries. Each bin value is replaced by the closest boundary value.
2. **Clustering:** Outliers in the data may be detected by clustering, where similar values are organized into groups, or ‘clusters’. Values that fall outside of the set of clusters may be considered outliers.

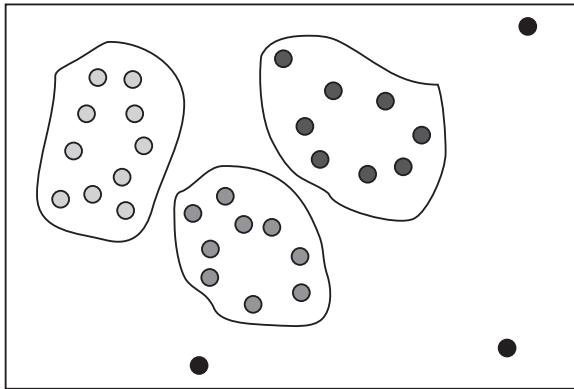


Fig. 15.7. Outliers.

3. **Regression:** smooth by fitting the data into regression functions.
- Linear regression involves finding the best of line to fit two variables, so that one variable can be used to predict the other.
 - Multiple linear regression is an extension of linear regression, where more than two variables are involved and the data are fit to a multidimensional surface.

Using regression to find a mathematical equation to fit the data helps smooth out the noise.

15.8.1.3 Data Cleaning as a Process

Missing values, noise, and inconsistencies contribute to inaccurate data. Cleaning process includes two steps:

- (i) Discrepancy detection
- (ii) Data transformations

15.8.1.3.1 Discrepancy detection

- Discrepancies can be caused by several factors, including poorly designed data entry forms that have many optional fields, human error in data entry, deliberate errors and data decay (e.g., outdated addresses).
- Discrepancies may also arise from inconsistent data representations and the inconsistent use of codes.
- Errors in instrumentation devices that record data, and system errors, are another source of discrepancies. Discrepancies can be overcome by the use of any knowledge you may already have regarding properties of the data. Such knowledge or “data about data” is referred to as metadata.
- Field overloading is a kind of source of errors that typically occurs when developers compress new attribute definitions into unused portions of already defined attributes.

- Following rules can be used to detect Discrepancies:
 - **Unique rule** states that each value of the given attribute must be different from all other values of that attribute.
 - **Consecutive rule** states that there can be no missing values between the lowest and highest values of the attribute and that all values must also be unique.
 - **Null rule** specifies the use of blanks, question marks, special characters or other strings that may indicate the null condition and how such values should be handled.

Tools to Detect Discrepancy

- **Data scrubbing tools** use simple domain knowledge to detect errors and make corrections in the data. These tools rely on parsing and fuzzy matching techniques when cleaning data from multiple sources.
- **Data auditing tools** find discrepancies by analyzing the data to discover rules and relationships, and detecting data that violate such conditions.

15.8.1.3.2 Data transformations

This is the second step in data cleaning as a process. After finding the discrepancies, we typically need to define and apply transformations to correct them.

Data Transformation Tools

- **Data migration tools** allow simple transformations to be specified, such as to replace the string “gender” by “sex”.
- **ETL (extraction/transformation/loading) tools** allow users to specify transforms through a graphical user interface (GUI). These tools typically support only a restricted set of transforms.

15.8.1.4 Disadvantages in data cleaning process

- (a) Nested discrepancies
- (b) Lack of interactivity
- (c) Increased interactivity

15.8.2 Data Integration

It combines data from multiple sources into a coherent store. These sources may include multiple databases, data cubes, or flat files. There are number of issues to consider during data integration.

Issues in data integration:

- **Schema integration:** It refers to the integration of metadata from different sources.
- **Entity identification problem:** Identifying entity in one data source which is similar to entity in another table. For example, customer_id in one db and customer_no in another db refer to the same entity.
- **Detecting and resolving data value conflicts:** Attribute values from different sources can be different due to different representations, different scales. E.g. metric vs. British units
- **Redundancy:** It is another issue confronted while performing data integration. Redundancy can occur due to the following reasons:

- **Object identification:** The same attribute may have different names in different db
- **Derived Data:** one attribute may be derived from another attribute.

15.8.2.1 Handling Redundant Data in Data Integration

Correlation analysis: Some redundancy can be identified by correlation analysis. The correlation between two variables A and B can be measured by

$$r_{A, B} = \frac{\sum (A - \bar{A})(B - \bar{B})}{(n - 1)\sigma_A \sigma_B}$$

\bar{A}, \bar{B} are respective mean values of A and B

σ_A, σ_B are respective standard deviation of A and B

n is the number of tuples

- The result of the equation is > 0 , then A and B are positively correlated, which means the value of A increases as the values of B increases. The higher value may indicate redundancy that may be removed.
- The result of the equation is $= 0$, then A and B are independent and there is no correlation between them.
- If the resulting value is < 0 , then A and B are negatively correlated where the values of one attribute increase as the value of one attribute decrease which means each attribute may discourage each other.

15.8.3 Data Transformation

In *data transformation*, the data are transformed or consolidated into forms appropriate for mining. Data transformation can involve the following:

- **Smoothing:** which works to remove noise from the data.
- **Aggregation:** where summary or aggregation operations are applied to the data. For example, the daily sales data may be aggregated so as to compute weekly and annual total scores.
- **Generalization of the data:** where low-level or “primitive” (raw) data are replaced by higher-level concepts through the use of concept hierarchies. For example, categorical attributes, like street, can be generalized to higher-level concepts, like city or country.
- **Normalization:** where the attribute data are scaled so as to fall within a small specified range, such as -1.0 to 1.0 , or 0.0 to 1.0 .
- **Attribute construction (feature construction):** this is where new attributes are constructed and added from the given set of attributes to help the mining process.

15.8.3.1 Normalization

In normalization, data are scaled to fall within a small, specified range, useful for classification algorithms involving neural networks, distance measurements such as nearest neighbor classification and clustering. There are 3 methods for data normalization. They are:

- min-max normalization

- z-score normalization
- normalization by decimal scaling

Min-max normalization: performs linear transformation on the original data values. It can be defined as,

$$v' = \frac{v - \min_A}{\max_A - \min_A} (\text{new_max}_A - \text{new_min}_A) + \text{new_min}_A$$

v is the value to be normalized

\min_A, \max_A are minimum and maximum values of an attribute A

$\text{new_max}_A, \text{new_min}_A$ are the normalization range.

Z-score normalization / zero-mean normalization: In which values of an attribute A are normalized based on the mean and standard deviation of A. It can be defined as,

$$v' = \frac{v - \text{mean}_A}{\text{stand_dev}_A}$$

This method is useful when min and max value of attribute A are unknown or when outliers dominate min-max normalization.

Normalization by decimal scaling: This refers to normalization by moving the decimal point of values of attribute A. The number of decimal points moved depends on the maximum absolute value of A. A value v of A is normalized to v' by computing,

$$v' = \frac{v}{10^j} \text{ where } j \text{ is smallest integer such that } \text{Max}(|v'|) < 1$$

15.8.4 Data Reduction Techniques

These techniques can be applied to obtain a reduced representation of the data set that is much smaller in volume but closely maintains the integrity of the original data. Data reduction includes,

1. **Data cube aggregation**, where aggregation operations are applied to the data in the construction of a data cube.
2. **Attribute subset selection**, where irrelevant, weakly relevant or redundant attributes or dimensions may be detected and removed.
3. **Dimensionality reduction**, where encoding mechanisms are used to reduce the data set size. Examples: Wavelet Transforms Principal Components Analysis
4. **Numerosity reduction**, where the data are replaced or estimated by alternative, smaller data representations such as parametric models or nonparametric methods such as clustering, sampling, and the use of histograms.

Discretization and concept hierarchy generation, where raw data values for attributes are replaced by ranges or higher conceptual levels. Data discretization is a form of numerosity reduction that is very useful for the automatic generation of concept hierarchies.

15.8.4.1 Data Cube Aggregation

Reduce the data to the concept level needed in the analysis. Queries regarding aggregated information should be answered using data cube when possible. Data cubes store multidimensional aggregated

information. The following figure shows a data cube for multidimensional analysis of sales data with respect to annual sales per item type for each branch.

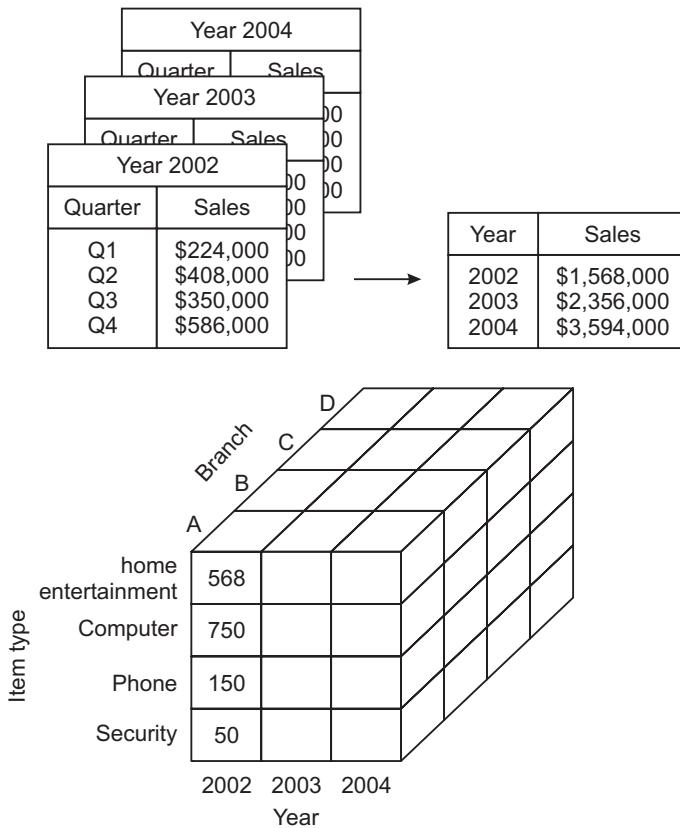


Fig. 15.8. Multi dimensional data cube.

Each cell holds an aggregate data value, corresponding to the data point in multidimensional space. Data cubes provide fast access to precomputed, summarized data, thereby benefiting on-line analytical processing as well as data mining.

1. The cube created at the lowest level of abstraction is referred to as the **base cuboid**.
 2. Data cubes created for varying levels of abstraction are called **lattice of cuboids or cudoids**.
 3. A cube for the highest level of abstraction is the **apex cuboid**.
 4. The lowest level of a **data cube (base cuboid)**.
- Each higher level of abstraction further reduces the resulting data size.

15.8.4.2 Attribute Sub Selection

Attribute subset selection reduces the data set size by removing irrelevant or redundant attributes (or dimensions). The goal of attribute subset selection is to find a minimum set of attributes. Mining

on a reduced set of attributes has an additional benefit. It reduces the number of attributes appearing in the discovered patterns, helping to make the patterns easier to understand.

Basic heuristic methods of attribute subset selection include the following techniques,

1. **Step-wise forward selection:** The procedure starts with an empty set of attributes. The best of the original attributes is determined and added to the set. At each subsequent iteration or step, the best of the remaining original attributes is added to the set.
2. **Step-wise backward elimination:** The procedure starts with the full set of attributes. At each step, it removes the worst attribute remaining in the set.
3. **Combination forward selection and backward elimination:** The step-wise forward selection and backward elimination methods can be combined, where at each step one selects the best attribute and removes the worst from among the remaining attributes.
4. **Decision tree induction:** Decision tree induction constructs a flow-chart-like structure where each internal (non-leaf) node denotes a test on an attribute, each branch corresponds to an outcome of the test, and each external (leaf) node denotes a class prediction. At each node, the algorithm chooses the “best” attribute to partition the data into individual classes. When decision tree induction is used for attribute subset selection, a tree is constructed from the given data. All attributes that do not appear in the tree are assumed to be irrelevant. The set of attributes appearing in the tree form the reduced subset of attributes.

<i>Forward Selection</i>	<i>Backward Selection</i>	<i>Decision Tree Induction</i>
Initial attribute set: {A1, A2, A3, A4, A5, A6}	Initial attribute set: {A1, A2, A3, A4, A5, A6}	Initial attribute set; {A1, A2, A3, A4, A5, A6}
Initial reduced set: {} {A1} {A1, A4}	Initial reduced set: {A1, A2, A3, A4, A5, A6} {A1, A4, A5, A6}	<pre> graph TD A4[A4?] -- Y --> A1[A1?] A4 -- N --> A6[A6?] A1 -- Y --> C1_1((Class 1)) A1 -- N --> C1_2((Class 2)) A6 -- Y --> C2_1((Class 1)) A6 -- N --> C2_2((Class 2)) </pre>
Reduced attribute set: {A1, A4, A6}	Reduced attribute set: {A1, A4, A6}	Reduced attribute set: {A1, A4, A6}

Fig. 15.9. Greedy (heuristic) methods for attribute subset selection.

15.8.4.3 Dimensionality Reduction

In *dimensionality reduction*, data encoding or transformations are applied so as to obtain a reduced or “compressed” representation of the original data. If the original data can be reconstructed from the compressed data without any loss of information, the data reduction is called **lossless**. If, instead, we can reconstruct only an approximation of the original data, then the data reduction is called **lossy**.

Two popular and effective methods of lossy dimensionality reduction:

1. Wavelet transforms
2. Principal components analysis.

15.8.4.3.1 Wavelet transforms

Wavelet transforms is a form of data compression well suited for image compression. The discrete wavelet transform (DWT) is a linear signal processing technique that, when applied to a data vector D, transforms it to a numerically different vector, D0, of wavelet coefficients.

The general algorithm for a discrete wavelet transform is as follows.

1. The length, L, of the input data vector must be an integer power of two. This condition can be met by padding the data vector with zeros, as necessary.
2. Each transform involves applying two functions:
 - data smoothing
 - calculating weighted difference
3. The two functions are applied to pairs the input data, resulting in two sets of data of length L/2.
4. The two functions are recursively applied to the sets of data obtained in the previous loop, until the resulting data sets obtained are of desired length.
5. A selection of values from the data sets obtained in the above iterations are designated the wavelet coefficients of the transformed data.

If wavelet coefficients are larger than some user-specified threshold then it can be retained. The remaining coefficients are set to 0.

15.8.4.3.2 Principal Component Analysis (PCA)

PCA is also called as Karhunen-Loeve (K-L) method. PCA searches for k n -dimensional orthogonal vectors that can best be used to represent the data, where $k \leq n$. The original data are thus projected onto a much smaller space, resulting in dimensionality reduction.

The basic procedure is as follows:

1. The input data are normalized.
2. PCA computes k ortho normal vectors that provide a basis for the normalized input data. These are unit vectors that each point in a direction perpendicular to the others. These vectors are referred to as the principal components. The input data are a linear combination of the principal components.
3. The principal components are sorted in order of decreasing “significance” or strength.

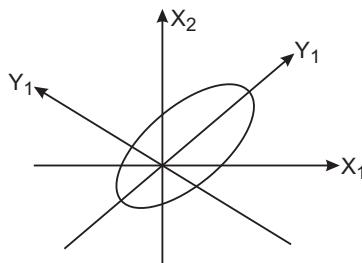


Fig. 15.10. Principal components analysis.

For example, Figure shows the first two principal components, Y_1 and Y_2 , for the given set of data originally mapped to the axes X_1 and X_2 . This information helps identify groups or patterns within the data.

4. Because the components are sorted according to decreasing order of “significance,” the size of the data can be reduced by eliminating the weaker components.

Advantages of PCA

- PCA is computationally inexpensive.
- Can be applied to ordered and unordered attributes
- Can handle sparse data and skewed data.
- Multidimensional data of more than two dimensions can be handled by reducing the problem to two dimensions.
- Principal components may be used as inputs to multiple regression and cluster analysis.

15.8.4.4 Numerosity Reduction

Data volume can be reduced by choosing alternative smaller forms of data. Techniques used in numerosity reduction

- Parametric method
- Non parametric method

Parametric: In this model only the data parameters need to be stored, instead of the actual data.

Non parametric: This method stores reduced representation of the data. It includes histogram, clustering and sampling.

Parametric Model:

1. **Linear Regression and log linear models:** In linear regression, the data are modeled to fit a straight line. For example, a random variable, y , can be modeled as a linear function of another random variable, x , with the equation

$$y = wx + b,$$

where the variance of y is assumed to be constant. In the context of data mining, x and y are numerical database attributes. The coefficients, w and b (called *regression coefficients*), specify the slope of the line and the Y -intercept, respectively.

2. **Multiple linear regression** is an extension of (simple) linear regression, which allows a response variable, y , to be modeled as a linear function of two or more predictor variables.
3. **Log-linear models:** Log-linear models can be used to estimate the probability of each point in a multidimensional space for a set of discretized attributes, based on a smaller subset of dimensional combinations.

This allows a higher-dimensional data space to be constructed from lower dimensional spaces. Log-linear models are therefore also useful for dimensionality reduction and data smoothing.

Nonparametric Model

1. Histogram

- Divide data into buckets and store average (sum) for each bucket

- A bucket represents an attribute-value/frequency pair
- It can be constructed optimally in one dimension using dynamic programming
- It divides up the range of possible values in a data set into classes or groups. For each group, a rectangle (bucket) is constructed with a base length equal to the range of values in that specific group, and an area proportional to the number of observations falling into that group.
- The buckets are displayed in a horizontal axis while height of a bucket represents the average frequency of the values.

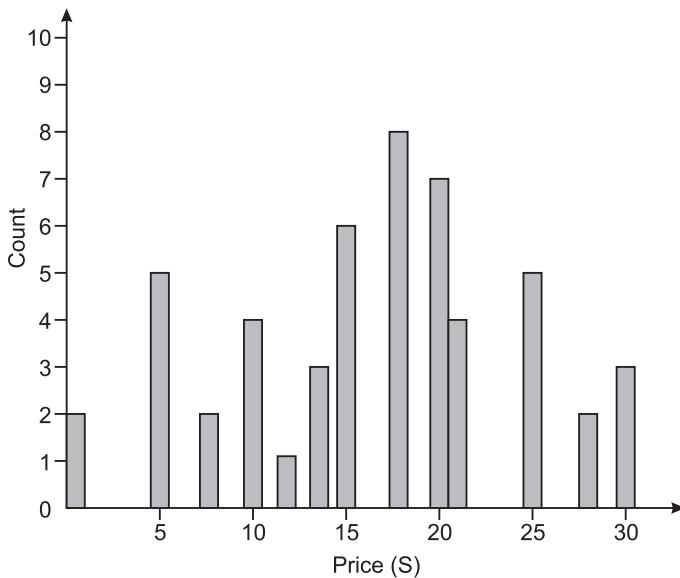


Fig. 15.11. Histogram.

The buckets can be determined based on the following partitioning rules, including the following.

1. **Equi-width:** histogram with bars having the same width
2. **Equi-depth:** histogram with bars having the same height
3. **V-Optimal:** histogram with least variance
4. **MaxDiff:** bucket boundaries defined by user specified threshold

V-Optimal and MaxDiff histograms tend to be the most accurate and practical. Histograms are highly effective at approximating both sparse and dense data, as well as highly skewed, and uniform data.

2. **Clustering techniques** consider data tuples as objects. They partition the objects into groups or clusters, so that the objects within a cluster are "similar" to one another and are "dissimilar" to the objects in other clusters. Similarity is commonly defined in terms of how "close" the objects are in space, based on a distance function.

Quality of clusters is measured by their diameter (max distance between any two objects in the cluster) or centroid distance (avg. distance of each cluster object from its centroid).

- 3. Sampling:** Sampling can be used as a data reduction technique since it allows a large data set to be represented by a much smaller random sample (or subset) of the data.

Advantages of Sampling:

1. An advantage of sampling for data reduction is that the cost of obtaining a sample is proportional to the size of the sample, n , as opposed to N , the data set size. Hence, sampling complexity is potentially sub-linear to the size of the data.
2. When applied to data reduction, sampling is most commonly used to estimate the answer to an aggregate query.

15.8.5 Data Discretization and Concept Hierarchies

Data Discretization: Discretization techniques can be used to reduce the number of values for a given continuous attribute, by dividing the range of the attribute into intervals. Interval labels can then be used to replace actual data values.

Data Discretization Categories: Discretization techniques can be categorized based on how the discretization is performed.

If the discretization process uses class information, then we say it is **supervised discretization**. Otherwise, it is **unsupervised**.

If the process starts by first finding one or a few points (called split points or cut points) to split the entire attribute range, and then repeats this recursively on the resulting intervals, it is called **top-down discretization or splitting**. This contrasts with **bottom-up discretization or merging**, which starts by considering all of the continuous values as potential split-points, removes some by merging neighborhood values to form intervals, and then recursively applies this process to the resulting intervals.

Concept Hierarchy: A concept hierarchy for a given numeric attribute defines a discretization of the attribute. Concept hierarchies can be used to reduce the data by collecting and replacing low level concepts (such as numeric values for the attribute age) by higher level concepts (such as young, middle-aged, or senior).

15.8.5.1 Concept Hierarchy Generation for Category Data

A concept hierarchy defines a sequence of mappings from set of low-level concepts to higher-level, more general concepts. It organizes the values of attributes or dimension into gradual levels of abstraction. They are useful in mining at multiple levels of abstraction

Categorical data are discrete data. Categorical attributes have a finite (but possibly large) number of distinct values, with no ordering among the values. Examples include *geographic location*, *job category*, and *itemtype*. There are several methods for the generation of concept hierarchies for categorical data.

Specification of a partial ordering of attributes explicitly at the schema level by users or experts: Concept hierarchies for categorical attributes or dimensions typically involve a group of attributes. A user or expert can easily define a concept hierarchy by specifying a partial or total ordering of the attributes at the schema level. For example, a relational database or a dimension *location* of a data warehouse may contain the following group of attributes: *street*, *city*, *province or state*, and

country. A hierarchy can be defined by specifying the total ordering among these attributes at the schema level, such as *street < city < state < country*.

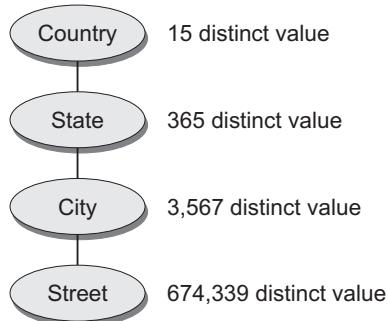


Fig. 15.12. Automatic generation of a schema concept hierarchy based on the number of distinct attribute values.

Specification of a portion of a hierarchy by explicit data grouping: This is essentially the manual definition of a portion of a concept hierarchy. In a large database, it is unrealistic to define an entire concept hierarchy. But it is easy to specify explicit groupings for a small portion of intermediate-level data.

Specification of a set of attributes, but not of their partial ordering: A user may specify a set of attributes forming a concept hierarchy, but omit to explicitly state their partial ordering. The system can then try to automatically generate the attribute ordering so as to construct a meaningful concept hierarchy.

Specification of only a partial set of attributes: Sometimes a user can be sloppy when defining a hierarchy, or have only a vague idea about what should be included in a hierarchy. Consequently, the user may have included only a small subset of the relevant attributes in the hierarchy specification. For example, instead of including all of the hierarchically relevant attributes for location, the user may have specified only street and city. To handle such partially specified hierarchies, it is important to embed data semantics in the database schema so that attributes with tight semantic connections can be pinned together.

15.9 DATA WAREHOUSE INTRODUCTION

A data warehouse is a collection of data marts representing historical data from different operations in the company. This data is stored in a structure optimized for querying and data analysis as a data warehouse. Table design, dimensions and organization should be consistent throughout a data warehouse so that reports or queries across the data warehouse are consistent. A data warehouse can also be viewed as a database for historical data from different functions within a company.

Data Warehouse can be defined as “a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management's decision making process”. *Subject Oriented Data* gives information about a particular subject instead of a company's ongoing operations.

Integrated: Data is gathered into the data warehouse from a variety of sources and merged into a coherent whole.

Time-variant: All data in the data warehouse are identified with a particular time period.

Non-volatile: Data is stable in a data warehouse. More data is added but data is never removed.

This enables management to gain a consistent picture of the business. It is a single, complete and consistent store of data obtained from a variety of different sources made available to end users in what they can understand and use in a business context. It can be:

- used for decision Support
- used to manage and control business
- used by managers and end-users to understand the business and make judgments

Data Warehousing is an architectural construct of information systems that provides users with current and historical decision support information that is hard to access or present in traditional operational data stores.

15.10 TERMINOLOGY

Enterprise Data warehouse: It collects all information about subjects (*customers, products, sales, assets, personnel*) that span the entire organization.

Data Mart: Departmental subsets that focus on selected subjects. A data mart is a segment of a data warehouse that can provide data for reporting and analysis on a section, unit, department or operation in the company, e.g. sales, payroll, production. Data marts are sometimes complete individual data warehouses which are usually smaller than the corporate data warehouse.

Decision Support System (DSS): Information technology that helps the knowledge worker (executive, manager, and analyst) make faster & better decisions.

Drill-down: Traversing the summarization levels from highly summarized data to the underlying current or old detail.

Metadata: Data about data. Containing location and description of warehouse system components: names, definition, structure.

15.11 BENEFITS OF DATA WAREHOUSING

- Data warehouses are designed to perform well with aggregate queries running on large amounts of data.
- The structure of data warehouses is easier for end users to navigate, understand and query against unlike the relational databases primarily designed to handle lots of transactions.
- Data warehouses enable queries that cut across different segments of a company's operation. E.g. production data could be compared against inventory data even if they were originally stored in different databases with different structures.
- Queries that would be complex in very normalized databases could be easier to build and maintain in data warehouses, decreasing the workload on transaction systems.
- Data warehousing is an efficient way to manage and report on data that is from a variety of sources, non uniform and scattered throughout a company.
- Data warehousing is an efficient way to manage demand for lots of information from lots of users.

- Data warehousing provides the capability to analyze large amounts of historical data for nuggets of wisdom that can provide an organization with competitive advantage.

15.12 DATA WAREHOUSE CHARACTERISTICS

Data in the Data Warehouse is integrated from various, heterogeneous operational systems like database systems, flat files, etc. Before the integration, structural and semantic differences have to be reconciled, i.e., data have to be “homogenized” according to a uniform data model. Furthermore, data values from operational systems have to be cleaned in order to get correct data into the data warehouse. Since a data warehouse is used for decision making, it is important that the data in the warehouse be correct. However, large volumes of data from multiple sources are involved; there is a high probability of errors and anomalies in the data. Therefore, tools that help to detect data anomalies and correct them can have a high payoff. Some examples where data cleaning becomes necessary are: inconsistent field lengths, inconsistent descriptions, inconsistent value assignments, missing entries, and violation of integrity constraints.

The need to access historical data is one of the primary incentives for adopting the data warehouse approach. Historical data are necessary for business trend analysis which can be expressed in terms of understanding the differences between several views of the real-time data. Maintaining historical data means that periodical snapshots of the corresponding operational data are propagated and stored in the warehouse without overriding previous warehouse states. However, the potential volume of historical data and the associated storage costs must always be considered in relation to their business benefits.

Data warehouse usually contains additional data, not explicitly stored in the operational sources, but derived through some process from operational data. For example, operational sales data could be stored in several aggregation levels in the warehouse.

15.13 DATA WAREHOUSE ARCHITECTURE AND ITS SEVEN COMPONENTS

Data warehouse is an environment, not a product which is based on relational database management system that functions as the central repository for informational data. The central repository information is surrounded by number of key components designed to make the environment is functional, manageable and accessible.

The data source for data warehouse comes from operational applications. The data is entered into the data warehouse, transformed into an integrated structure and format. The transformation process involves conversion, summarization, filtering and condensation. The data warehouse must be capable of holding and managing large volumes of data as well as different structure of data over the time.

The data warehouse architecture has seven components. The components are as follows:

1. Data sourcing, cleanup, transformation, and migration tools
2. Metadata repository
3. Warehouse/database technology
4. Data marts
5. Data query, reporting, analysis, and mining tools

6. Data warehouse administration and management
7. Information delivery system

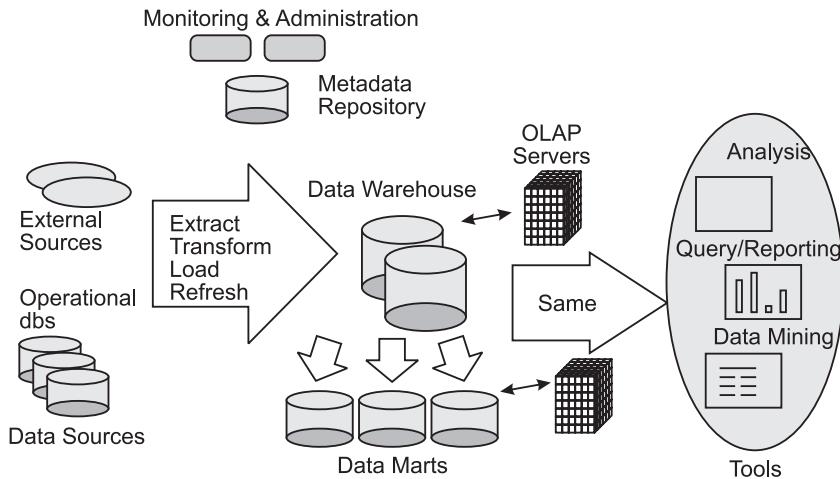


Fig. 15.13. Data warehouse architecture.

1. Data Warehouse Database: This is the central part of the data warehousing environment. This is implemented based on RDBMS technology.

2. Sourcing, Acquisition, Clean up, and Transformation Tools: Sourcing, Acquisition, Clean up, and Transformation Tools perform conversions, summarization, key changes, structural changes and condensation. The data transformation is required so that the information can be used by decision support tools. The transformation produces programs, control statements, JCL code, COBOL code, UNIX scripts, and SQL DDL code etc., to move the data into data warehouse from multiple operational systems.

The functionalities of these tools are listed below:

- To remove unwanted data from operational db
- Converting to common data names and attributes
- Calculating summaries and derived data
- Establishing defaults for missing data

3. Meta data: It is data about data. It is used for maintaining, managing and using the data warehouse. It is classified into two:

Technical Meta data: It contains information about data warehouse, data used by warehouse designer, administrator to carry out development and management tasks. It includes:

- Info about data stores
- Transformation descriptions. That is mapping methods from operational db to warehouse db
- Warehouse Object and data structure definitions for target data
- The rules used to perform clean up, and data enhancement
- Data mapping operations

- Access authorization, backup history, archive history, info delivery history, data acquisition history, data access etc.,

Business Meta data: It contains info that gives info stored in data warehouse to users. It includes,

- Subject areas, and info object type including queries, reports, images, video, audio clips etc.
- Internet home pages
- Info related to info delivery system
- Data warehouse operational info such as ownerships, audit trails etc.,

Meta data helps the users to understand content and find the data. Meta data are stored in a separate data stores which is known as informational directory or Meta data repository which helps to integrate, maintain and view the contents of the data warehouse. The following lists the characteristics of info directory/ Meta data:

- It is the gateway to the data warehouse environment
- It supports easy distribution and replication of content for high performance and availability
- It should be searchable by business oriented key words
- It should act as a launch platform for end user to access data and analysis tools
- It should support the sharing of info
- It should support scheduling options for request
- It should support and provide interface to other applications
- It should support end user monitoring of the status of the data warehouse environment

4. Access tools: Its purpose is to provide info to business users for decision making. There are five categories:

- 1) Data query and reporting tools
- 2) Application development tools
- 3) Executive info system tools (EIS)
- 4) OLAP tools
- 5) Data mining tools

Query and reporting tools are used to generate query and report. There are two types of reporting tools. They are:

- Production reporting tool used to generate regular operational reports
- Desktop report writer are inexpensive desktop tools designed for end users.

Managed Query tools: This tool is used to generate SQL query. It uses Meta layer software in between users and databases which offers a point-and-click creation of SQL statement. This tool is a preferred choice of users to perform segment identification, demographic analysis, territory management and preparation of customer mailing lists etc.

Application Development Tools: This is a graphical data access environment which integrates OLAP tools with data warehouse and can be used to access all db systems

OLAP Tools: These are used to analyze the data in multi dimensional and complex views. To enable multidimensional properties it uses MDDB and MRDB where MDDB refers multi dimensional data base and MRDB refers multi relational data bases.

Data Mining Tools: These are used to discover knowledge from the data warehouse data also can be used for data visualization and data correction purposes.

5. Data marts: Data marts are Departmental subsets that focus on selected subjects. They are independently used by dedicated user group. They are used for rapid delivery of enhanced decision support functionality to end users. Data mart is used in the following situation:

- Extremely urgent user requirement
- The absence of a budget for a full scale data warehouse strategy
- The decentralization of business needs
- The attraction of easy to use tools and mind sized project

Data mart presents two problems:

(i) **Scalability:** A small data mart can grow quickly in multi dimensions. So that while designing it, the organization has to pay more attention on system scalability, consistency and manageability issues

(ii) Data integration

6. Data Warehouse Admin and Management: The management of data warehouse includes,

- Security and priority management
- Monitoring updates from multiple sources
- Data quality checks
- Managing and updating meta data
- Auditing and reporting data warehouse usage and status
- Purging data
- Replicating, sub setting and distributing data
- Backup and recovery
- Data warehouse storage management which includes capacity planning, hierarchical storage management and purging of aged data etc.,

7. Information Delivery System

- It is used to enable the process of subscribing for data warehouse info.
- Delivery to one or more destinations according to specified scheduling algorithm

15.14 CLASSIFICATION OF DATA WAREHOUSE DESIGN

The data warehouse design can be broadly classified into two categories:

1. Logical design
2. Physical design.

15.14.1 Logical Design

The logical design is more conceptual and abstract than physical design. In the logical design, the emphasis is on the logical relationship among the objects. One technique that can be used to model organization's logical information requirements is entity-relationship modeling. Entity-relationship modeling involves identifying the things called entities, the properties of these things called the attributes, and how they are related to one another called the relationships.

The process of logical design involves arranging data into a series of logical relationships called entities and attributes. An entity represents a chunk of information. In relational databases, an entity often maps to a table. An attribute is a component of an entity that helps define the uniqueness of the entity. In relational databases, an attribute maps to a column. Whereas entity-relationship diagramming has traditionally been associated with highly normalized models such as OLTP applications, the technique is still useful for data warehouse design in the form of dimensional modeling.

In dimensional modeling, instead of seeking to discover atomic units of information and all the relationship between them, the focus is on identifying which information belongs to a central fact table and which information belongs to its associated dimension tables. The logical design should result in

- (a) A set of entities and attributes corresponding to fact tables and dimension tables and
- (b) A model of operational data from the source into subject-oriented information in target data warehouse schema.

Data Warehousing Schemas

A schema is a collection of database objects, including tables, views, indexes, and synonyms. The arrangement of schema objects in the schema models designed for data warehouse can be done in a variety of ways. Most data warehouses use a dimensional model.

Star Schema

The star schema is the simplest data warehouse schema. It is called a star schema because the diagram resembles a star, with points radiating from the center. The center of the star consists of one or more fact tables and the points of the star are the dimension tables as illustrated in Fig. 15.14.

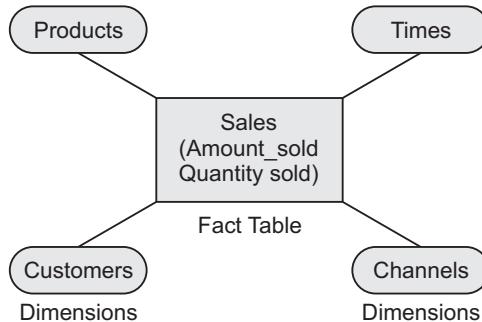


Fig. 15.14. Star schema.

A star schema optimizes performance by keeping queries simple and providing fast response time. All the information about each level is stored in one row. The most natural way to model a data warehouse is star schema, only one join establishes the relationship between the fact table and any one of the dimension tables. Another schema that is sometimes useful is the snowflake schema, which is a star schema with normalized dimensions in a tree structure.

Data Warehouse Objects

Two types of objects used in the dimensional data warehouse schemas are Fact and dimension tables.

Fact Tables: Fact tables are the large tables in warehouse schema that store business measurements. Fact tables typically contain facts and foreign keys to the dimension tables. Fact tables represent data, usually numeric and additive that can be analyzed and examined.

A fact table typically has two types of columns: those that contain numeric facts and those that are foreign keys to dimension tables. A fact table contains either detail-level facts or facts that have been aggregated.

Fact tables that contain aggregated facts are often called summary tables. A fact table usually contains facts with the same level of aggregation. Though most facts are additive, they can also be semi additive or nonadditive. Additive facts can be aggregated by simple arithmetical addition. Semi additive facts can be aggregated along some of the dimensions and not along others.

Dimension Tables: Dimension tables, also known as lookup or reference tables; contain the relatively static data in the warehouse. Dimension tables stores the information that is normally used to contain queries. Dimension tables are usually textual and descriptive.

A dimension is a structure, often composed of one or more hierarchies, that categorizes data. Dimensional attributes help to describe the dimensional value. They are commonly descriptive, textual values. Several distinct dimensions, combined with facts, enable one to answer business questions. Dimensional data are typically collected at the lowest level of detail and then aggregated into higher level totals that are more useful for analysts. These natural rollups or aggregations within a dimension table are called hierarchies.

Hierarchies: Hierarchies are logical structures that use ordered levels as a means of organizing data. A hierarchy can be used to define data aggregation. For example, in a time dimension, a hierarchy might aggregate data from the month level to the quarter level to the year level.

A hierarchy can also be used to define a navigational drill path and to establish a family structure. Within a hierarchy, each level is logically connected to the levels above and below it. Data values at lower levels aggregate into the data values at higher levels.

A dimension can be composed of more than one hierarchy. Hierarchies impose a family structure on dimension values. For a particular level value, a value at the next higher level is its parent, and values at the next lower level are its children. These familial relationships enable analysts to access data quickly.

15.14.2 Physical Design

During the physical design process the data gathered during the logical design phase is converted into a description of the physical database structure. Physical design decisions are mainly driven by query performance and database maintenance aspects.

Physical Design Structures

Some of the physical design structures that are going to be discussed in this section include:

1. Table spaces
2. Tables and Partitioned Tables
3. Data Segment Compression
4. Views
5. Integrity Constraints
6. Indexes and Partitioned Indexes
7. Dimensions

1. Table Spaces: A table space consists of one or more data files, which are physical structures within the operating system. A data file is associated with only one table space. From the design perspective, table spaces are containers for physical design structures. Table spaces need to be separated by differences. For example, tables should be separated from their indexes and small tables should be separated from large tables. Table spaces should also represent logical business units.

2. Tables and Partitioned Tables: Tables are the basic unit of data storage. They are the container for the expected amount of raw data in the data warehouse. Using partitioned tables instead of nonpartitioned ones addresses the key problem of supporting very large data volumes by allowing you to decompose them into smaller and more manageable pieces. The main design criterion for partitioning is manageability.

3. Data Segment Compression: Disk space can be saved by compressing heap-organized tables. A typical type of heap-organized table that one should consider for data segment compression is partitioned tables. Data segment compression can also speed up query execution. There is, however, a cost in CPU overhead. Data segment compression should be used with highly redundant data, such as tables with many foreign keys.

4. Views: A view is a personalized presentation of the data contained in one or more tables or other views. A view takes the output of a query and treats it as a table. Views do not require any space in the database.

5. Integrity Constraints: Integrity constraints are used to enforce business rules associated with the database and to prevent having invalid information in the tables. Integrity constraints in data warehousing differ from constraints in OLTP environments. In OLTP environments, they primarily prevent the insertion of invalid data into a record, which is not a big problem in data warehousing environments because accuracy has already been guaranteed.

In data warehousing environments, constraints are only used for query rewrite. NOT NULL constraints are particularly common in data warehouses. Under some specific circumstances, constraints need space in the database. These constraints are in the form of the underlying unique index.

6. Indexes and Partitioned Indexes: Indexes are optional structures associated with tables or clusters. In addition to the classical B-tree indexes, bitmap indexes are very common in data warehousing environments. Bitmap indexes are optimized index structures for set-oriented

operations. Additionally, they are necessary for some optimized data access methods such as star transformations.

7. Dimensions: A dimension is a schema object that defines hierarchical relationships between columns or column sets. A hierarchical relationship is a functional dependency from one level of a hierarchy to the next one. A dimension is a container of logical relationships and does not require any space in the database. A typical dimension is city, state (or province), region, and country.

REVIEW QUESTIONS

1. What is the need for data mining?
2. Explain the steps of knowledge discovery in database.
3. Explain the architecture of data mining system.
4. Explain various tasks in data mining.
5. Explain the components of data warehousing.
6. What are the functionalities of data mining system?
7. What are the performance issues of data mining?
8. What is data preprocessing and state its need?
9. Explain data preprocessing techniques in detail.
10. Explain the classification of data warehousing system.
11. List out the benefits of using data warehousing and data mining.
12. Explain how data mining system is integrated with a data warehouse system.
13. Categorise data mining system and explain the same.



