Course Description

Course number | DB07 | Course name | Relational Database Management System

Author(s)      | Hanumesh V.J., hanumesh@infosys.com, E&R Dept, Bangalore Seema Acharya, seema_acharya@infosys.com, E&R Dept, Mysore

Pre-requisites for attending course: Successful completion of Computer Hardware and System Software Concepts (CHSSC), Programming Fundamentals (PF)

Suggested course duration | Seven days

Review Log

<table>
<thead>
<tr>
<th>Version Revision</th>
<th>Date</th>
<th>Author</th>
<th>Reviewed by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0a</td>
<td>06-Jun-2005</td>
<td>Hanumesh V.J. Seema Acharya</td>
<td>Sundar K.S Hanumesh V.J.</td>
<td>FP Restructure 2005 curriculum</td>
</tr>
<tr>
<td>1.0</td>
<td>24-Jun-2005</td>
<td>Seema Acharya</td>
<td></td>
<td>Review comments incorporated</td>
</tr>
</tbody>
</table>

References


2. An Introduction to Database Systems, C.J.Date, Seventh Edition 2003, Pearson Education

# Table of Contents

PURPOSE ................................................................................................................... 1

1. INTRODUCTION TO DBMS ............................................................................. 2
   1.1. WHAT IS A DATABASE? ........................................................................... 2
   1.2. WHAT IS A DATABASE MANAGEMENT SYSTEM (DBMS)? ................... 5
   1.3. FILE SYSTEM INTERFACE VERSUS DBMS INTERFACE ...................... 6
   1.4. MASTER AND TRANSACTION FILES ..................................................... 8
   1.5. TRADITIONAL APPROACH TO INFORMATION PROCESSING ............... 10
      1.5.1. Disadvantages of the Traditional Approach to Information Processing 11
   1.6. WHY DBMS? ......................................................................................... 14
   1.7. TYPES OF DATABASES ......................................................................... 15
   1.8. THREE LEVEL ARCHITECTURE FOR A DBMS ....................................... 18
   1.9. DBMS USERS ...................................................................................... 21
   1.10. DATA MODELS .................................................................................. 23
      1.10.1. Object Based Logical Model ......................................................... 23
      1.10.2. Record Based Logical Model ......................................................... 24
      1.10.2.1. Hierarchical Data Model ............................................................ 24
      1.10.2.2. Network Data Model ................................................................. 25
      1.10.2.3. Relational Data Model ............................................................... 25
      1.10.2.4. Structural Terminology .............................................................. 26
   1.11. RDBMS ............................................................................................... 26
   1.12. SOME POPULAR RDBMS PACKAGES .............................................. 27
   1.13. APPLICATION AREAS OF RDBMS .................................................... 27
   1.14. KEYS .................................................................................................. 27
   1.15. SUMMARY .......................................................................................... 34

2. ENTITY-RELATIONSHIP (E-R) MODELING ..................................................... 36
   2.1. INTRODUCTION .................................................................................... 36
   2.2. ENTITY AND RELATIONSHIP .............................................................. 37
   2.3. CARDINALITY OF A RELATIONSHIP ................................................... 38
      2.3.1. One to One Relationship ............................................................... 38
      2.3.2. One to Many Relationship ............................................................ 39
      2.3.3. Many to One Relationship ............................................................ 40
      2.3.4. Many to Many Relationship ......................................................... 40
   2.4. E-R DIAGRAM NOTATIONS ................................................................. 41
   2.5. MODELING USING E-R DIAGRAMS .................................................... 43
      2.5.1. Steps in E-R Modeling .................................................................. 43
   2.6. CASE STUDY 1: PROBLEM STATEMENT ............................................ 44
      2.6.1. Case Study 1: Solution .................................................................. 44
   2.7. CASE STUDY 2: PROBLEM STATEMENT ............................................ 47
      2.7.1. Case Study 2: Solution .................................................................. 47
   2.8. CASE STUDY 3: PROBLEM STATEMENT ............................................ 48
      2.8.1. Case Study 3: Solution .................................................................. 49
   2.9. TRANSFORMING AN E-R MODEL INTO PHYSICAL DATABASE DESIGN 52
   2.10. MERITS AND DEMERITS OF E-R MODELING ................................... 54
      2.10.1. Merits of E-R Modeling ............................................................... 54
      2.10.2. Demerits of E-R Modeling ............................................................ 55
   2.11. SUMMARY .......................................................................................... 56

3. NORMALIZATION ............................................................................................ 57
   3.1. INTRODUCTION .................................................................................... 57
   3.2. THE NEED FOR NORMALIZATION ...................................................... 57
## Table of Contents

3.3. **PROCESS OF NORMALIZATION** .................................................................................. 59  
   3.3.1. **Determinant** ........................................................................................................ 59  
   3.3.2. **Functional Dependency** ..................................................................................... 59  
   3.3.3. **Full Functional Dependency** ............................................................................. 60  
   3.3.4. **Partial Dependency** ........................................................................................... 61  
   3.3.5. **Transitive Dependency** ...................................................................................... 61  
   3.3.6. **Key attributes** .................................................................................................... 61  
   3.3.7. **Non key attributes** .............................................................................................. 62  

3.4. **TYPES OF NORMAL FORMS** .................................................................................. 62  
   3.4.1. **First Normal Form (1 NF)** .................................................................................. 62  
   3.4.2. **Second Normal Form (2 NF)** ............................................................................. 63  
   3.4.3. **Third Normal Form (3 NF)** ............................................................................... 66  
   3.4.4. **Boyce Codd Normal Form (BCNF)** .................................................................. 67  

3.5. **MERITS AND DEMERITS OF NORMALIZATION** ..................................................... 69  
   3.5.1. **Merits** ................................................................................................................ 69  
   3.5.2. **Demerits** ............................................................................................................ 69  

3.6. **SUMMARY** ............................................................................................................... 70  

3.7. **CASE STUDY** ........................................................................................................... 72  

4. **STRUCTURED QUERY LANGUAGE (SQL)** ............................................................... 75  
   4.1. **THE PURPOSE OF SQL** ......................................................................................... 75  
   4.2. **A BRIEF HISTORY OF SQL** .................................................................................. 76  
   4.3. **DATA TYPES** ......................................................................................................... 77  
   4.4. **STATEMENT TYPES** ............................................................................................. 79  
   4.5. **DATA DEFINITION LANGUAGE (DDL) STATEMENTS** ..................................... 79  
      4.5.1. **CREATE TABLE Statement** .............................................................................. 80  
      4.5.2. **ALTER TABLE statement** ............................................................................. 86  
      4.5.3. **DROP TABLE statement** .............................................................................. 89  
      4.5.4. **TRUNCATE TABLE statement** ...................................................................... 89  
      4.5.5. **CREATE INDEX statement** .......................................................................... 90  
   4.6. **DATA MANIPULATION LANGUAGE STATEMENTS** .......................................... 92  
      4.6.1. **INSERT Statement** ......................................................................................... 93  
      4.6.2. **DELETE Statement** ......................................................................................... 96  
      4.6.3. **UPDATE Statement** ....................................................................................... 98  
      4.6.4. **SELECT Statement** ......................................................................................... 98  
         4.6.4.1. **Simple SELECT Statement** ...................................................................... 99  
         4.6.4.2. **Avoiding duplicates (DISTINCT)** ............................................................... 100  
         4.6.4.3. **Row Selection (WHERE clause)** ............................................................... 100  
         4.6.4.4. **BETWEEN, IN, LIKE** .............................................................................. 103  
         4.6.4.5. **IS NULL, IS NOT NULL** .......................................................................... 105  
         4.6.4.6. **Column titles using AS** .......................................................................... 106  
         4.6.4.7. **Sorting Query Results (ORDER BY clause)** ........................................... 107  
         4.6.4.8. **Aggregate Functions / Column Functions** ............................................. 108  
         4.6.4.9. **GROUP BY** .............................................................................................. 110  
         4.6.4.10. **HAVING** ................................................................................................. 113  
         4.6.4.11. **Retrieval using UNION** .......................................................................... 115  
         4.6.4.12. **Retrieval using INTERSECT** ................................................................. 117  
      4.6.5. **Sub-Queries** .................................................................................................... 118  
         4.6.5.1. **Independent Sub-Queries** ................................................................ ...... 118  
         4.6.5.2. **Co-Related Sub-Queries** ......................................................................... 120  
      4.6.6. **JOINS** ............................................................................................................... 124  
         4.6.6.1. **SELF JOIN** .............................................................................................. 126  
         4.6.6.2. **INNER JOINS** .......................................................................................... 128  
         4.6.6.3. **OUTER JOINS** .......................................................................................... 129  
         4.6.6.4. **LEFT OUTER JOIN** ................................................................................... 130  
         4.6.6.5. **RIGHT OUTER JOIN** ................................................................................ 131  
      4.6.7. **Queries using EXISTS / NOT EXISTS** ............................................................ 132
5. ON-LINE TRANSACTION PROCESSING (OLTP) .......................................................... 149

4.6.7.1. EXISTS ........................................................................................................ 132
4.6.7.2. NOT EXISTS ................................................................................................. 132
4.6.8. The Order of Execution of a SELECT statement ................................................. 133

4.7. VIEWS ..................................................................................................................... 134
4.7.1. Horizontal View ............................................................................................... 134
4.7.2. Vertical View ..................................................................................................... 134
4.7.3. DROP VIEW Statement .................................................................................... 135
4.7.4. Joined Views .................................................................................................... 135
4.7.5. VIEW Updates ................................................................................................ 135
4.7.6. Checking View Updates (CHECK OPTION) ...................................................... 136
4.7.7. Advantages of Views ....................................................................................... 137
4.7.8. Disadvantages of Views .................................................................................. 137

4.8. DATA CONTROL LANGUAGE (DCL) ................................................................. 138
4.8.1. Granting Privileges .......................................................................................... 138
4.8.1.1. Passing Privileges (GRANT OPTION) .......................................................... 139
4.8.2. Revoking Privileges (REVOKE) ....................................................................... 139

4.9. EMBEDDED SQL .................................................................................................. 141
4.9.1. Purpose ............................................................................................................ 141
4.9.2. Why Embedded SQL? .................................................................................... 141

4.10. BEST PRACTICES ............................................................................................... 145
4.11. SUMMARY ............................................................................................................ 148

5.1. PURPOSE ................................................................................................................. 149
5.2. TRANSACTION ....................................................................................................... 149
5.3. TRANSACTION SYSTEMS ..................................................................................... 151
5.3.1. Batch Transaction Processing System .............................................................. 151
5.3.2. On-line Transaction Processing System (OLTP) .............................................. 152
5.3.3. Real time Transaction Processing System ....................................................... 152

5.4. TRANSACTION PROPERTIES ............................................................................. 152
5.5. REQUIREMENTS FOR AN OLTP SYSTEM ......................................................... 153
5.5.1. Integrity ............................................................................................................ 153
5.5.2. Concurrency ..................................................................................................... 155
5.5.2.1. Lost Update .................................................................................................. 155
5.5.2.2. Dirty Read .................................................................................................... 156
5.5.2.3. Incorrect Summary ....................................................................................... 157
5.5.2.4. Phantom Record .......................................................................................... 157

5.6. LOCKS ..................................................................................................................... 159
5.6.1. Shared Lock (S) ............................................................................................... 159
5.6.2. Exclusive Lock (X) ......................................................................................... 159

5.7. GRANULARITY OF LOCKING ............................................................................. 160

5.8. INTENT LOCKING ................................................................................................ 162
5.8.1. Intent Share (IS) ............................................................................................. 163
5.8.2. Intent Exclusive (IX) ...................................................................................... 163
5.8.3. Shared Intent Exclusive (SIX) ......................................................................... 163
5.8.4. Case study for Intent Locks ............................................................................ 166

5.9. DEADLOCK ........................................................................................................... 168
5.10. TIMESTAMPING ..................................................................................................... 168
5.11. SECURITY ............................................................................................................. 170
5.12. RECOVERY ........................................................................................................... 171
5.13. TRANSACTION LOG .......................................................................................... 172
5.13.1. Deferred update ............................................................................................. 172
5.13.2. Immediate Update ......................................................................................... 174
5.13.3. Check-Points ................................................................................................. 177

5.14. SUMMARY ............................................................................................................. 180
6. ON LINE ANALYTICAL PROCESSING (OLAP) ............................................................... 182
   6.1. DIFFERENCE BETWEEN OLTP AND OLAP ......................................................... 182
   6.2. DATA WAREHOUSE ......................................................................................... 182
       6.2.1. Why data warehouse is needed?............................................................. 183
       6.2.2. Characteristics of Data Warehouse: ......................................................... 183
       6.2.3. Data Warehousing Terminology ............................................................... 184
       6.2.4. Data Collection for Data Warehouse Applications ................................. 185
       6.2.5. Storing of data in Data warehouse ............................................................ 186
       6.2.6. Reporting of a Data warehouse application ............................................. 188
               6.2.6.1. Advantages of Data Marts: ............................................................. 188
       6.2.7. Difference between Data Warehouse and Data Mart .............................. 189
       6.2.8. Popular tools available for data warehousing ......................................... 190
   6.3. SUMMARY ...................................................................................................... 190
7. GLOSSARY ........................................................................................................ 191
8. INDEX .............................................................................................................. 195
PURPOSE

All business activities deal with a lot of data.

Examples:

- Schools, colleges and universities store data about students, courses, trainers, etc.
- Banks store data about their customers, transactions (deposits, withdrawals), loans, etc.

A Database Management System (DBMS) provides an efficient storage and data management mechanism.

All real life software projects use databases to store huge volumes of data. It is extremely important for a software engineer to understand the concepts of DBMS. The knowledge of DBMS enables a software engineer to:

- Store data
- Access data
- Modify data
- Delete data
- Share data among the different users
- Ensure security of the data

In short, DBMS concepts and techniques help in the efficient management of data.

---

1 Transaction: A group of processing steps that are treated as a single activity to achieve a desired result. In DBMS, collections of operations that form a single logical unit of work are called transactions. A database system ensures proper execution of transactions despite failures – either the entire transaction executes, or none of it does.
1. Introduction to DBMS

1.1. What is a Database?

A database is an organized collection of interrelated data.

**Example:** Consider a bank database. The bank stores data about their customers in a file known as *Customer_Details*. The *Customer_Details* file has the following fields:

- **Cust_ID**: The customer’s identification number
- **Cust_Last_Name**: The customer’s last name
- **Cust_Mid_Name**: The customer’s middle name or initials
- **Cust_First_Name**: The customer’s first name
- **Account_No**: The customer’s account number
- **Account_Type**: The type of account that the customer has in the bank (Savings or Checking etc).
- **Bank_Branch**: The name of the bank branch
- **Cust_Email**: The customer’s email ID

The bank also stores data about the loan(s) taken by its customers. The loan details are stored in the file *Customer_Loan*. The *Customer_Loan* file has the following fields:

- **Cust_ID**: The customer’s identification number
- **Loan_No**: The loan number to identify the loan
- **Amount_in_Dollars**: The amount loaned by the bank to the customer

One customer can avail multiple loans from the bank.

The data stored in the two files, *Customer_Details* and *Customer_Loan* constitutes interrelated data.

Refer to Figure 1-1.
The data in the database is integrated which means that the database is a collection of several distinct\(^2\) files. These distinct files may have some duplicate data but the duplication of data is kept to the minimum.

**Example:** Figure 1-1 shows two files, Customer_Details and Customer_Loan. The two files are distinct in the sense, Customer_Details file contains details about the bank’s customers and the Customer_Loan file contains details about all the loans taken by the customers of the bank. Both the files have the Cust_ID field.

In order to sanction a loan to a customer, the bank requires the account number (Account_No) of the customer. There is no need to include the account number information again in the Customer_Loan file, because it can always be discovered by referring to the Customer_Details file.

The data in the database can be shared. **Sharing** means individual pieces of data in the database can be shared among different users. Each of those users can have access to the same piece of data. They can use the data for different purposes.

Refer to Figure 1-2.

---

\(^2\) **Distinct:** Not identical.
As depicted in Figure 1-2, the Account_No information from the Customer_Details file is being accessed by the users in the bank’s Fixed Deposit Department and the users in the bank’s Loan Department. The information would typically be used for different purposes by the two classes of users.

The users can even concurrently access the database. Concurrent access implies that different users can access the same piece of data at the same time.

**Points to Remember:**

- A database is an organized collection of interrelated data
- Data in the database:
  - Is integrated
  - Can be shared
  - Can be concurrently accessed
1.2. What is a Database Management System (DBMS)?

A Database Management System (DBMS) is a collection of interrelated files and a set of programs that allow users to access and modify these files. The primary goal of a DBMS is to provide a convenient and efficient way to store, retrieve and modify information.

Figure 1-3 shows an end user\(^3\) working with data from the Customer_Details file, maintained in the bank database.

The database systems are designed to:
- Define structures for the storage of data
- Provide mechanisms for the manipulation\(^4\) of data
- Ensure the safety of the data stored, despite system crashes or attempts at unauthorized access
- Share data among the different users

In short, database systems are designed to manage large volumes of data.

---

\(^3\) **End User**: The person for whom a system is being developed. Example: a bank teller or a bank manager is an end user of a bank system.

\(^4\) **Manipulation**: Data manipulation refers to the addition of new data, modification of existing data, etc.
1.3. File System Interface versus DBMS Interface

In the traditional file approach, data is stored in flat files\(^5\) which are maintained by the file system, under the operating system's control.

Refer to Figure 1-4.

The end users use the application programs to perform specific tasks. For example, personnel in the bank’s Loan Department make use of the Loan_Processing system to process the loan(s) of customer(s).

All application programs go through the file system to access the data stored in these flat files.

---

\(^5\) **Flat Files**: A flat file is a file containing records that has no structured interrelationship. Files used in programming fundamentals (PF) projects were essentially flat files.

\(^6\) **SQL**: (Structured Query Language). A language used by relational databases to query, update and manage data. Relational database is explained in Section 1.10.2.3.
The application programs are written in some programming language such as COBOL, PL/1, C++ or in some higher level *fourth generation language*\(^7\). The standard SQL interface is provided as an integral part of the database system software to access the database.

![Diagram of DBMS handling requests for access to the database](image)

The **DBMS** acts as a layer of *abstraction*\(^8\) over the file system.

---

\(^7\) **Fourth Generation Language (4GL):** A 4GL is typically non-procedural and designed so that end users can specify what they want without having to know how the computer will process their requirement.

\(^8\) **Abstraction:** A simplified representation of something that is potentially quite complex. It is often not necessary to know the exact details of how something works, is represented or is implemented, because it can still be used in its simplified form.
**Example:** As depicted in Figure 1-6, in the file system interface, the end user uses an application program written in a high level language such as COBOL, to access the data from the Customer_Details file. The files are maintained by the file system under the operating systems control. In the DBMS interface, the end user uses an SQL interface to place a request to the DBMS to retrieve data from the Customer_Details table.

![Diagram of File System Interface versus DBMS Interface](image)

**1.4. Master and Transaction Files**

A master file is used to store relatively static data about some entity. A transaction file contains relatively transient data about a particular data processing task.

---

9 **Table:** A table has a specified number of columns but can have any number of rows. Rows stored in a table are structurally equivalent to records from flat files.

10 **Entity:** An entity is a “thing” or “object” in the real world that is distinguishable from other objects. Example: each person is an entity, and bank accounts can be considered to be entities.
Example: Consider the banking system consisting of two files, the Customer_Details and the Customer_Transaction file.

In Figure 1-7, Customer_Details is the master file containing all the information about the bank’s customers. In Figure 1-8, Customer_Transaction is the transaction file containing information about all the transactions that a customer makes with the bank.

The Customer_Details file is modified rarely. For example, when a new account is created or whenever the existing details of a customer changes. However, for every deposit or withdrawal made by the customer(s), the Customer_Transaction file is updated.

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Account_No</th>
<th>Account_Type</th>
<th>Bank_Branch</th>
<th>Cust_Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
<td>1020</td>
<td>Savings</td>
<td>Downtown</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
</tr>
<tr>
<td>102</td>
<td>Smith</td>
<td>S.</td>
<td>Graham</td>
<td>2348</td>
<td>Checking</td>
<td>Bridgewater</td>
<td><a href="mailto:Smith_Graham@rediffmail.com">Smith_Graham@rediffmail.com</a></td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>S.</td>
<td>Justin</td>
<td>3421</td>
<td>Savings</td>
<td>Plainsboro</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
</tr>
<tr>
<td>104</td>
<td>Quails</td>
<td>D.</td>
<td>Jack</td>
<td>2367</td>
<td>Checking</td>
<td>Downtown</td>
<td><a href="mailto:Quails_Jack@yahoo.com">Quails_Jack@yahoo.com</a></td>
</tr>
<tr>
<td>105</td>
<td>Jones</td>
<td>E.</td>
<td>Simon</td>
<td>2385</td>
<td>Checking</td>
<td>Brighton</td>
<td><a href="mailto:Jones_Simon@rediffmail.com">Jones_Simon@rediffmail.com</a></td>
</tr>
</tbody>
</table>

Customer_Details records from Customer_Details file

Figure 1-7: Example of master file - Customer_Details

<table>
<thead>
<tr>
<th>Account_No</th>
<th>Transaction_Date</th>
<th>Transaction_Type</th>
<th>Transaction_Amount_in_Dollars</th>
<th>Total_Available_Balance_in_Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>1020</td>
<td>12-Jan-2005</td>
<td>Deposit</td>
<td>5000.00</td>
<td>10000.00</td>
</tr>
<tr>
<td>2348</td>
<td>14-Jan-2005</td>
<td>Withdrawal</td>
<td>2500.00</td>
<td>13500.00</td>
</tr>
<tr>
<td>3421</td>
<td>14-Jan-2005</td>
<td>Deposit</td>
<td>2000.00</td>
<td>27234.00</td>
</tr>
<tr>
<td>2367</td>
<td>16-Jan-2005</td>
<td>Withdrawal</td>
<td>1200.00</td>
<td>12456.00</td>
</tr>
<tr>
<td>1020</td>
<td>17-Jan-2005</td>
<td>Withdrawal</td>
<td>1500.00</td>
<td>8500.00</td>
</tr>
</tbody>
</table>

Customer_Transaction records from Customer_Transaction file

Figure 1-8: Example of transaction file - Customer_Transaction

Points to Remember:

- A master file
  - Stores relatively static data about an entity
  - Changes rarely

- A transaction file
  - Stores relatively transient data about a particular data processing task
  - Changes frequently as transactions happen more periodically and in large numbers
1.5. Traditional Approach to Information Processing

In the traditional file approach each application maintains its own master file and generally has its own set of transaction files. Files are custom-designed for each application and there is little sharing of data among the various applications.

Application programs are data-dependent. It is impossible to change the physical representation (how the data is physically represented in storage) or access technique (how it is physically accessed) without affecting the application.

Refer to Figure 1-9.

Although the traditional, file-oriented approach is still widely used, it has some serious disadvantages. The next section deals with the drawbacks of the traditional approach to information processing.
1.5.1. Disadvantages of the Traditional Approach to Information Processing

The disadvantages of the traditional approach to information processing are discussed below:

- **Data Security**: The data as maintained in the flat file(s) is easily accessible and therefore not secure.

**Example**: Consider the banking system. The Customer Transaction file has details about the total available balance of all customers. A customer wants information about his account balance. In a file system it is difficult to give the customer access to only his data in the file. Thus, enforcing security constraints\(^{11}\) for the entire file or for certain data items are difficult.

- **Data Redundancy**: Often the same information is duplicated in two or more files. Refer to Figure 1-10.

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Account_No</th>
<th>Account_Type</th>
<th>Bank_Branch</th>
<th>Cust_Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
<td>1020</td>
<td>Savings</td>
<td>Downtown</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
</tr>
<tr>
<td>102</td>
<td>Smith</td>
<td>S.</td>
<td>Graham</td>
<td>2346</td>
<td>Checking</td>
<td>Bridgewater</td>
<td><a href="mailto:Smith_Graham@rediffmail.com">Smith_Graham@rediffmail.com</a></td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>D.</td>
<td>Justin</td>
<td>3421</td>
<td>Savings</td>
<td>Plainsboro</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
</tr>
<tr>
<td>104</td>
<td>Qualis</td>
<td>E.</td>
<td>Jack</td>
<td>2367</td>
<td>Checking</td>
<td>Downtown</td>
<td><a href="mailto:Qualis_Jack@yahoo.com">Qualis_Jack@yahoo.com</a></td>
</tr>
<tr>
<td>105</td>
<td>Jones</td>
<td>E.</td>
<td>Simon</td>
<td>2389</td>
<td>Checking</td>
<td>Brighton</td>
<td><a href="mailto:Jones_Simon@rediffmail.com">Jones_Simon@rediffmail.com</a></td>
</tr>
</tbody>
</table>

Customer_Detail records from Customer_Details file

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Cust_Email</th>
<th>Fixed_Deposit_No</th>
<th>Amount_In_Dollars</th>
<th>Rate_of_Interest_In_Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
<td>2011</td>
<td>8055.00</td>
<td>6.5</td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>D.</td>
<td>Justin</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
<td>2015</td>
<td>2060.00</td>
<td>6.5</td>
</tr>
<tr>
<td>104</td>
<td>Qualis</td>
<td>D.</td>
<td>Jack</td>
<td><a href="mailto:Qualis_Jack@yahoo.com">Qualis_Jack@yahoo.com</a></td>
<td>3010</td>
<td>3050.00</td>
<td>6.5</td>
</tr>
</tbody>
</table>

Customer_Fixed_Deposit records from Customer_Fixed_Deposit file

**Figure 1-10: Data Redundancy in files**

This duplication of data (redundancy) leads to higher storage and access cost. In addition, it may lead to data inconsistency\(^{12}\). **Example**: Assume that the same data is repeated in two or more files. If change is made to data in one file, it is required that the change be made to the data in the other file as well. If this is not done, it will lead to error during access to the data.

**Example**: As depicted in Figure 1-10, customer’s details such as Cust_Last_Name, Cust_Mid_Name, Cust_First_Name, Cust_Email are stored both in the Customer_Details file and the Customer_Fixed_Deposit file. If the email ID of one

\(^{11}\) **Constraints**: Restriction, limitation.

\(^{12}\) **Inconsistency**: Lacking uniformity or agreement.
customer, for example, Langer G. Justin changes from Langer_Justin@yahoo.com to Langer_Justin@ediffmail.com, the Cust_Email must be updated in both the files; otherwise it will lead to inconsistent data.

However, one can design file systems with minimal redundancy. Data redundancy is sometimes preferred. **Example:** Assume that the customer’s details such as Cust_Last_Name, Cust_Mid_Name, Cust_First_Name and Cust_Email are not stored in the Customer_Fixed_Deposit file. If it is required to get this information about the customer along with his fixed deposit details, it would mean that the details be retrieved from two files. This would mean an increased overhead. It is thus preferred to store the information in the Customer_Fixed_Deposit file itself.

- **Data Isolation:** Data isolation means that all the related data is not available in one file. Generally, the data is scattered in various files, and the files may be in different formats, therefore writing new application programs to retrieve the appropriate data is difficult.

- **Program/Data Dependence:** Under the traditional file approach, application programs are data-dependent. It is impossible to change the physical representation (how the data is physically represented in storage) or access technique (how it is physically accessed) without affecting the application. Changes in the physical format of the master file(s), such as addition of a data field requires that the change be made in all the application programs that accesses the master file. Consequently, for each of the application programs that a programmer writes or maintains, the programmer must also focus on data management issues. There can be no centralized execution of the data management functions. Data management is scattered among all the application programs.

**Example:** Consider the banking system. The master file, Customer_Fixed_Deposit contains details about the customers fixed deposit accounts. Refer to Figure 1-11. A customer’s fixed deposit record is described by:

- Cust_ID
- Cust_Last_Name
- Cust_Mid_Name
- Cust_First_Name
- Cust_Email
- Fixed_Deposit_No
- Amount_in_Dollars
- Rate_of_Interest_in_Percent

An application program is available to display all the details about the fixed deposit accounts of all the customers. Assume that a new data field, the **Centralized:** Systems where decision making, flow of data, or the beginning of activities are initiated at the same central point and disseminated to remote points in the organization.
Fixed_Deposit_Maturity_Date is added to the master file. The application program must also be altered because it depends on the master file.

If for example, the physical format of the master/transaction file such as the field delimiter or record delimiter is changed, it necessitates that the application program which depends on it, also be altered.

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Cust_Email</th>
<th>Fixed_Deposit_No</th>
<th>Amount_in_Dollars</th>
<th>Rate_of_Interest_in_Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
<td>2011</td>
<td>8055.00</td>
<td>6.5</td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>D.</td>
<td>Justin</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
<td>2015</td>
<td>2060.00</td>
<td>6.5</td>
</tr>
<tr>
<td>104</td>
<td>Qualis</td>
<td>D.</td>
<td>Jack</td>
<td><a href="mailto:Qualis_Jack@yahoo.com">Qualis_Jack@yahoo.com</a></td>
<td>3010</td>
<td>3050.00</td>
<td>6.5</td>
</tr>
</tbody>
</table>

Customer_Fixed_Deposit records from Customer_Fixed_Deposit file

**Figure 1-11: Master file - Customer_Fixed_Deposit**

- **Lack of Flexibility:** The traditional systems are able to retrieve information for predetermined requests for data. If the management needs unanticipated data, the information can perhaps be provided if it is in the files of the system. Extensive programming is however required which may result in a delay. By the time the information is made available, it may no longer be required or useful.

**Example:** Consider the banking system. An application program is available to generate a list of customer names in a particular area of the city. However the bank manager requires a list of those customers who have an account balance greater than $10,000.00 and reside in a particular area of the city. An application program for this purpose does not exist. The bank manager has two choices:

- To print the list of customer names in a particular area of the city and then manually find out those with an account balance greater than $10,000.00
- Hire an application programmer to write the application program for the same

Both the solutions are cumbersome.

- **Concurrent Access Anomalies:** Many traditional systems allow multiple users to access and update the same piece of data simultaneously. But the interaction of concurrent updates may result in inconsistent data. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult because data may be accessed by many different application programs and these application programs may not have been coordinated previously.

**Example:** Consider the banking system. Assume that the bank manager is analyzing all the transactions made by the customers. At about the same time, a customer accesses his account to make a withdrawal. The account is both read by the bank manager and updated by the customer at the same time. This is called concurrent access. Because the customer’s account is being updated at the same time, there is a possibility of the bank manager reading an incorrect balance.
These difficulties prompted the development of database systems.

**Points to Remember:**

Disadvantages of the traditional file approach:
- **Data Security** – Data easily accessible by all and therefore not secure
- **Data Redundancy** – Same data is duplicated in two or more files which may lead to update anomalies
- **Data Isolation** – All the related data is not available in one file. Thus writing a new application program is difficult
- **Program / Data Dependence** – Application programs are data-dependent. It is impossible to change the physical representation (how the data is physically represented in storage) or access technique (how it is physically accessed) without affecting the application
- **Lack of Flexibility** – Only pre-determined requests for information can be met. It is not flexible enough to satisfy unanticipated queries
- **Concurrent Access Anomalies** – Same piece of data is allowed to be updated simultaneously which leads to inconsistencies

### 1.6. Why DBMS?

DBMS ensures the following:

- Application programs and *queries*\(^{14}\) are data-independent. They do not depend on any one particular physical representation of data in secondary storage or access technique

- Allows for sharing of data among different users. Users are also able to access the database concurrently without facing the issues of inconsistent data

- Controls redundancy and inconsistency

- Provides secure access to the database

- Enforces *integrity constraints*\(^{15}\) (also known as business rules) by preventing the entry of invalid information into the database

- Enables backup and recovery from system crashes

---

\(^{14}\) **Queries**: A query is essentially a request that a user makes on the database.

\(^{15}\) **Integrity Constraints**: A set of rules to ensure the correctness and accuracy of data.
### 1.7. Types of Databases

There are two generic database architectures: centralized and distributed. The basic differences between the two architectures are:

<table>
<thead>
<tr>
<th>Centralized</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refer to Figure 1-12.</td>
<td>Refer to Figure 1-13.</td>
</tr>
<tr>
<td>All data is located at a single site</td>
<td>The database is stored on several computers - from personal computers up to mainframe systems</td>
</tr>
<tr>
<td>Allows for greater control over accessing and updating data</td>
<td>Computers in a distributed system communicate with one another through various communication media, such as high speed networks or telephone lines</td>
</tr>
<tr>
<td>Vulnerable to failure as they depend on the availability of resources at the central site</td>
<td>Distributed databases are geographically separated and managed</td>
</tr>
<tr>
<td></td>
<td>Distributed databases are separately administered</td>
</tr>
<tr>
<td></td>
<td>Distributed databases have a slower interconnection</td>
</tr>
</tbody>
</table>

**Example:** The account information of customers is stored in a particular branch office of a bank. This information must be shared across all Automated Teller Machines (ATM), so that customers can withdraw money from their accounts. Instead of storing the customer information in every ATM machine it can be stored at a common place (the branch office of the bank) and shared over a network.

**Example:** Consider the bank system. The bank’s head office is located at Chicago and the branch offices are at Melbourne and Tokyo. The bank database is distributed across the branch offices. The branch offices are connected through a network.
Figure 1-12: Centralized Database

Figure 1-13: Distributed Databases
The distributed databases can be classified as *homogeneous*\(^{16}\) or *heterogeneous*\(^{17}\).

In a distributed system, it is easy to distinguish between local and global transactions. A local transaction is one that accesses data in the single site at which the transaction was initiated. A global transaction, on the other hand, is one which either accesses data in a site different from the one at which the transaction was initiated, or access data in several different sites.

**Example:** Consider the bank system. The bank’s head office is located at Chicago and the branch offices are at Melbourne and Tokyo. The branch offices are connected through a network. Each branch has its own computer, with a database consisting of all the accounts maintained at that branch. Refer to Figure 1-14.

The head office maintains information about all the branches of the bank. Consider a transaction to add $50 to account number 1020 located at the Downtown bank branch in Tokyo. If the transaction was initiated at the Downtown bank branch in Tokyo, then it is considered local; otherwise, it is considered global. A transaction to transfer $50 from account 1020 to account 2389, which is located at the Brighton bank branch in Melbourne, is a global transaction, since accounts in two different sites are accessed as a result of its execution.

Thus in a distributed database system:

- The various sites are aware of one another
- Each site provides an environment to execute both local and global transactions
- If each site runs the same distributed database management software, it is called homogeneous distributed database systems
- If different sites run different database management software, it is difficult to manage global transactions. Such systems are called multi-database systems or heterogeneous distributed database systems

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Account_No</th>
<th>Account_Type</th>
<th>Bank.Branch</th>
<th>Cust_Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
<td>1020</td>
<td>Savings</td>
<td>Downtown</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
</tr>
<tr>
<td>102</td>
<td>Smith</td>
<td>S.</td>
<td>Graham</td>
<td>2348</td>
<td>Checking</td>
<td>Bridgewater</td>
<td><a href="mailto:Smith_Graham@rediffmail.com">Smith_Graham@rediffmail.com</a></td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>G.</td>
<td>Justin</td>
<td>3421</td>
<td>Savings</td>
<td>Plainsboro</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
</tr>
<tr>
<td>104</td>
<td>Quails</td>
<td>D.</td>
<td>Jack</td>
<td>2367</td>
<td>Checking</td>
<td>Downtown</td>
<td><a href="mailto:Quails_Jack@yahoo.com">Quails_Jack@yahoo.com</a></td>
</tr>
<tr>
<td>105</td>
<td>Jones</td>
<td>E.</td>
<td>Simon</td>
<td>2385</td>
<td>Checking</td>
<td>Brighton</td>
<td><a href="mailto:Jones_Simon@rediffmail.com">Jones_Simon@rediffmail.com</a></td>
</tr>
</tbody>
</table>

Customer_Detail records from Customer_Details file

Figure 1-14: Customer_Details file

---

\(^{16}\) **Homogeneous:** All the same; uniform; harmonized.

\(^{17}\) **Heterogeneous:** Varied; mixed; diverse.
1.8. Three level architecture for a DBMS

Most commercial databases are based on a three-level architecture model called the ANSI/SPARC (American National Standards Institute/Standard Planning and Requirements Committee) model.

Refer to Figure 1-15.

The overall design of the database is called database schema. Schemas are not changed frequently. In general, database systems support one internal schema, one conceptual schema and several external schemas.

**External / View level:** Many users of the database system are not concerned with all the information in the database. Instead, they need to access only a part of the database. The external level of abstraction simplifies the end user’s interaction with the system. The system may provide many views for the same database.

**Conceptual / Logical level:** The conceptual level describes what data are stored in the database, and what relationships exist among those data. This level is used by the Database Administrator\(^\text{18}\)'s (DBA), who in turn decide what information must be kept in the database.

---

\(^{18}\) **Database Administrator:** The DBA administers the DBMS and is in charge of creating, maintaining and modifying all the three levels of the DBMS. The DBA also controls the allocation of system resources, grants/revokes access privileges to/from users and ensures the consistency of the database.
Internal / Physical level: The internal level is the lowest level of abstraction and describes the data storage and access methods. Database Administrator(s) may be aware of certain details of the physical organization of the data.

Example: Consider a banking system. It uses:
A bank database
- Customer_Details table
- Customer_Transaction table

At the internal level, a Customer_Details or Customer_Transaction record can be described as a block of consecutive storage locations (for example, words or bytes). The language compiler hides this level of detail from programmers. Similarly, the database system hides the lowest-level storage details (how the data is stored and accessed) from database programmers.

At the conceptual level, the table definition (the attribute\(^\text{19}\) data type and width definition) and the interrelationship among the data is described.

Finally at the external level, several views\(^\text{20}\) of the database are defined, and database end users are able to see these views. In addition to hiding details of the conceptual level of the database, the views also provide a security mechanism to prevent users from accessing parts of the database. For example, tellers in the bank will be able to see only that part of the database that has information on customer accounts; they cannot access information concerning salaries of bank employees.

Detailed system architecture (Figure 1-16).

The database management system (DBMS) is the software that handles all access to the database. Conceptually, what happens is the following:
- A user issues an access request (typically using SQL)
- The DBMS intercepts that request and analyzes it
- The DBMS inspects the external schema for that user, the corresponding external/conceptual mapping, the conceptual schema, the conceptual/internal mapping and the storage structure definition
- The DBMS executes the necessary operations on the stored database

\(^\text{19}\) Attribute: The literal meaning is quality; characteristic; trait or feature. Entities are described in a database by a set of attributes. For example, in the bank system, Cust_ID, Cust_Email, etc. describe Customer-Detail entity set.

\(^\text{20}\) View: A view is a virtual table in the database defined by a query. For more details on views see Chapter 4.
Figure 1-16: Detailed System Architecture

Figure 1-17: An example of the three levels

The above figure depicts the three levels of DBMS architecture. The external view is how the customer, Mike A Smith views it. The conceptual view is how the DBA views it. The internal view is how the data is actually stored.
1.9. DBMS Users

The DBMS users, depending on their level of interaction with the system, fall into one of the three categories.

- **End User**: End users deal only with the highest level of abstraction. End users may not be concerned with or even aware of the details of the DBMS. Typically, the end user is involved in updates to the database or queries on the database.

- **Application Programmer**: Application programmer is responsible for writing database application programs in some programming language such as COBOL, PL/I, C++, or some higher-level fourth generation language. The application programs access the database by issuing the appropriate request to the DBMS.

- **Database Administrator (DBA)**: The DBA can be a single person or a group of persons. The functions of the DBA include the following:
  
  - **Defining the Conceptual Schema**: It is the DBA’s job to decide exactly what information is to be held in the database. The DBA identifies the entities and the information to be recorded about those entities. This process is usually referred to as logical database design. Once the DBA has decided the content of the database at an abstract level, he creates the corresponding conceptual schema.
  
  - **Defining the Internal Schema**: The DBA must also decide how the data is to be represented in the database. This process is usually referred to as physical database design. Having done the physical design, the DBA must then create the corresponding storage structure definition. In addition, the DBA must also define the associated conceptual/ internal mapping.
  
  - **Liaising with users**: The DBA liaises with users to ensure that the data they need is available and to write the necessary external schema. In addition, the DBA must also define the associated external/ conceptual mapping.
  
  - **Granting of authorization for data access**: The granting of different types of authorizations (read, write, etc.) allows the DBA to regulate which parts of the database various users can access.
  
  - **Defining Integrity constraints**: The data values stored in the database must satisfy certain consistency constraints.

Refer to Figure 1-18.
Figure 1-18: Users of DBMS

- **External Level**
  - End User
  - Works at the highest level of abstraction.
  - Deals with updates and queries

- **Conceptual Level**
  - Application Programmers
  - Writes application programs

- **Internal Level**
  - Data Base Administrator (DBA)
  - Defines the Conceptual, Internal and External schema, controls access privileges to users and ensures consistency of the database
1.10. **Data Models**

A data model is a conceptual tool to describe data, data relationships, data semantics and consistency constraints.

Two of the widely used data models are discussed in the next sections.

### 1.10.1. Object Based Logical Model

Entity-Relationship Model (E-R Model) is a widely known object based logical model. They are used to describe data at the conceptual and the view level. The E-R Model is based on the perception of the real world that consists of a collection of basic objects called entities, and of relationships among these objects. The E-R Model is covered in detail in Chapter 2.

![Entity Relationship Model Diagram]

**Figure 1-19:** Entity Relationship Model

### 1.10.2. Record Based Logical Model

They are used to describe data at the conceptual and the view level. They are used both to specify the overall logical structure of the database and to provide a higher-level description of the implementation. There are three widely accepted record based logical models.
1.10.2.1. Hierarchical Data Model

The hierarchical data model organizes data in a tree structure. There is a hierarchy of parent and child data segments. This structure implies that a record can have repeating information, generally in the child data segments.

Data is represented by a collection of records (record types). A record type is the equivalent of a table in the relational model, and with the individual records being the equivalent of rows. To create links between these record types, the hierarchical model uses parent-child relationships.

In a hierarchical database the parent-child relationship is one to many. This restricts a child segment to having only one parent segment.

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

Example: Consider the banking system. Figure 1-20 shows the hierarchical representation of Customer_Details and Customer_Loan records from Customer_Details and Customer_Loan files respectively.

Note: Loan (Loan_No: 1011) is shown as taken jointly by Mike A. Smith and Graham S. Smith to explain the difference between hierarchical and network Model.

![Figure 1-20: Hierarchical Model](image)

1.10.2.2. Network Data Model

The network model permitted the modeling of many-to-many relationships in data. In 1971, the Conference on Data Systems Languages (CODASYL) formally defined the network model.
Data in the network model is represented by a collection of records and the relationships among data are represented by links (pointers). The records in the database are organized as collections of graphs. **Example:** IDMS

**Example:** Refer to Figure 1-21. Assume that loan (Loan_No:1011) is taken jointly by two customers (Mike A. Smith and Graham S. Smith).

In the hierarchical model (Figure 1-20), the loan information has to be repeated for each customer individually because it does not permit many to many relationship. The parent-child relationship is one to many.

However in the network model, because it allows many to many relationship, the loan information is stored only once and both the customers can refer to it.

```
101  Smith  A. Mike  1020  Savings  Downtown  Smith_Mike@yahoo.com
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
102  Smith  S. Graham  2348  Checking  Bridgewater  Smith_Graham@rediffmail.com
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
103  Langer  G. Justin  3421  Savings  Plainsboro  Langer_Justin@yahoo.com
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
104  Quails  D. Jack  2367  Checking  Downtown  Quails_Jack@yahoo.com
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
     |    |          |           |              |
105  Jones  E. Simon  2389  Checking  Brighton  Jones_Simon@rediffmail.com
```

**Figure 1-21: Network Model**

### 1.10.2.3. Relational Data Model

The relational model uses a collection of tables (relations), each of which is assigned a unique name, to represent both data and the relationships among those data.

A table has a specified number of columns but can have any number of rows. Rows stored in a table resemble records from flat files. A row in a table represents a relationship among a set of values. Refer to Figure 1-22, a row in the Customer_Loan table gives the details of a loan taken by a customer. **Example:** Customer (Cust_ID: 101) has taken a loan (Loan_No: 1011) of amount (Amount_in_Dollars: 8755.00)

Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name.
### 1.10.2.4. Structural Terminology

<table>
<thead>
<tr>
<th>Formal Relational Term</th>
<th>Informal Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation</td>
<td>Table</td>
</tr>
<tr>
<td>Tuple</td>
<td>Row or Record</td>
</tr>
<tr>
<td>Cardinality of a Relation</td>
<td>Number of rows</td>
</tr>
<tr>
<td>Attribute</td>
<td>Column or Field</td>
</tr>
<tr>
<td>Degree of a Relation</td>
<td>Number of Columns</td>
</tr>
<tr>
<td>Primary Key</td>
<td>Unique Identifier</td>
</tr>
<tr>
<td>Domain</td>
<td>A pool of values from which the values of specific attributes of specific relations are taken</td>
</tr>
</tbody>
</table>

### 1.11. RDBMS

**Relational Database**: Any database for which the logical organization is based on relational data model.

**RDBMS**: A DBMS that manages the relational database.

An RDBMS is a type of DBMS that stores data in the form of related tables.
1.12. Some Popular RDBMS packages

<table>
<thead>
<tr>
<th>RDBMS Package</th>
<th>Company / Corporation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>Oracle Corp.</td>
</tr>
<tr>
<td>Sybase</td>
<td>Sybase Inc.</td>
</tr>
<tr>
<td>Informix</td>
<td>Informix Software Inc.</td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
</tr>
<tr>
<td>DB2</td>
<td>IBM</td>
</tr>
<tr>
<td>Ingres II</td>
<td>Computer Associates International Inc.</td>
</tr>
<tr>
<td>SQL Server</td>
<td>Microsoft</td>
</tr>
</tbody>
</table>

1.13. Application Areas of RDBMS

Databases are widely used in real life applications such as:

1. **Airlines**: For reservations and schedule information.

2. **Banking**: For customer information, accounts, loans and banking transactions.

3. **Universities**: For student information, course registrations and grades.

4. **Telecommunications**: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards and storing information about the communication networks.

5. **Sales**: For customer, product and purchase information in any industry.

1.14. Keys

**Candidate Key**: A candidate key is a set of one or more attributes that can uniquely identify a row in a given table.

**Example**: Consider the Customer_Details table shown in Figure 1-23.

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Account_No</th>
<th>Account_Type</th>
<th>Bank_Branch</th>
<th>Cust_Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
<td>102</td>
<td>Savings</td>
<td>Downtown</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
</tr>
<tr>
<td>102</td>
<td>Smith</td>
<td>S.</td>
<td>Graham</td>
<td>2348</td>
<td>Checking</td>
<td>Bridgewater</td>
<td><a href="mailto:Smith_Graham@rediffmail.com">Smith_Graham@rediffmail.com</a></td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>G.</td>
<td>Justin</td>
<td>3421</td>
<td>Savings</td>
<td>Plainsboro</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
</tr>
<tr>
<td>104</td>
<td>Quails</td>
<td>D.</td>
<td>Jack</td>
<td>2367</td>
<td>Checking</td>
<td>Downtown</td>
<td><a href="mailto:Quails_Jack@yahoo.com">Quails_Jack@yahoo.com</a></td>
</tr>
<tr>
<td>105</td>
<td>Jones</td>
<td>E.</td>
<td>Simon</td>
<td>2388</td>
<td>Checking</td>
<td>Brighton</td>
<td><a href="mailto:Jones_Simon@rediffmail.com">Jones_Simon@rediffmail.com</a></td>
</tr>
</tbody>
</table>

Customer_Detail records from Customer_Details table

**Figure 1-23: Customer_Details Table**
**Assumptions:**

- **One customer can have only one account**  
  **Example:** As depicted in Figure 1-23, customer Mike A. Smith has a Savings account with Account_No: 1020. Similarly, customer Justin G. Langer has a Savings account with Account_No: 3421.

- **An account can belong to only one customer**  
  **Example:** Account_No: 2367 belongs to Jack Quails.

- **Two rows cannot have the same values in the attributes, Cust_Last_Name and Cust_First_Name attributes. If two rows have the same value for Cust_Last_Name, they differ in their values for Cust_First_Name**

In the Customer_Details table, there are four candidate Keys. Out of the four, three are simple candidate keys and one is a composite candidate key. They are:

**Simple Candidate Key:** A candidate key comprising of one attribute only.  
**Example:**  
- Account_No  
- Cust_ID  
- Cust_Email

**Composite Candidate Key:** A candidate key comprising of two or more attributes.  
**Example:** \{Cust_Last_Name, Cust_First_Name\}.  
Cust_Last_Name alone is not sufficient to distinguish between rows in the table. But along with Cust_First_Name it can distinguish between rows in the table. Their combination constitutes a candidate key.

**Invalid Candidate Key:** A candidate key should be comprised of a set of attributes that can uniquely identify a row. A subset of the attributes should not possess the unique identification property.  
**Example:** the combination of \{Account_No, Account_Type\} is an invalid candidate key. Although the attributes Account_No and Account_Type together can distinguish rows, their combination does not form a candidate key, since the attribute Account_No alone is a candidate key.

Candidate keys are identified during the design of the database.

**Primary Key:** During the creation of the table (the implementation phase), the database designer chooses one of the candidate key from amongst the several available, to uniquely identify rows in the Customer_Details table. The candidate key so chosen is called the primary key.

**Example:** The database designer chooses Account_No to differentiate between rows in the Customer_Details table. Account_No is the primary key for the Customer_Details table. Refer to Figure 1-24.
**Entity integrity constraint:** The primary key of a table is always not null and unique. The attributes which constitute the primary key cannot have duplicate values in the rows of the table. It is mandatory to provide input for the primary key attributes. This constraint is referred to as the entity integrity constraint. A null value is an unknown value. It is not a blank character or a zero value.

**It is preferable to select a candidate key with a minimal number of attributes to function as a primary key.**

**Example:** It is preferable to select the candidate key (Account_No) as the primary key as opposed to the candidate key (Cust_Last_Name, Cust_First_Name).

<table>
<thead>
<tr>
<th>Points to Remember:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A candidate key is a set of one or more attributes that can uniquely identify a row in a given table</td>
</tr>
<tr>
<td>There can be more than one candidate keys in a table</td>
</tr>
<tr>
<td>Candidate keys are identified during the design phase</td>
</tr>
<tr>
<td>While creating the table, the database designer chooses one candidate key from amongst the several available, to serve as a primary key</td>
</tr>
<tr>
<td>It is preferred to select a candidate key with a minimal number of attributes to function as a primary key</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Guidelines to select a primary key:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Give preference to numeric column(s). The search algorithm performs better when the primary key is numeric</td>
</tr>
<tr>
<td>Give preference to a single attribute. The search algorithm gives better output with a single attribute primary key than with a composite attribute primary key</td>
</tr>
<tr>
<td>Give preference to the minimal composite key. A composite key is a collection of two or more attributes. <strong>Example:</strong> if the candidate keys are {x1,x2,x3} and {y1,y2}, the composite key {y1,y2} is the minimal composite key and will therefore be chosen as the primary key</td>
</tr>
<tr>
<td>Primary keys are chosen according to business convenience</td>
</tr>
</tbody>
</table>
**Foreign Key:** A foreign key is a set of attribute(s) the values of which are required to match the values of a candidate key in the same or another table. The foreign key attribute(s) can have duplicate or null values.

**Example:** Consider the banking system. The details of all the customers of the bank are stored in the **Customer_Details** table. Whenever the customer makes a transaction for example a deposit or withdrawal of funds from the bank, the transaction is recorded in the **Customer_Transaction** table.

A transaction is allowed only if the customer has an account in the bank. The account number information is stored in the **Customer_Details** table. This information is referred to for every transaction. If the account number does not exist, the transaction will not be allowed.

### Figure 1-24: Customer_Details Table with Account_No as Primary Key

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Account_No</th>
<th>Account_Type</th>
<th>Bank_Branch</th>
<th>Cust_Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A</td>
<td>Mike</td>
<td>1029</td>
<td>Savings</td>
<td>Downtown</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
</tr>
<tr>
<td>102</td>
<td>Smith</td>
<td>S</td>
<td>Graham</td>
<td>2348</td>
<td>Checking</td>
<td>Bridgewater</td>
<td><a href="mailto:Smith_Graham@rediffmail.com">Smith_Graham@rediffmail.com</a></td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>S</td>
<td>Justin</td>
<td>3421</td>
<td>Savings</td>
<td>Plainsboro</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
</tr>
<tr>
<td>104</td>
<td>Quails</td>
<td>D</td>
<td>Jack</td>
<td>2367</td>
<td>Checking</td>
<td>Downtown</td>
<td><a href="mailto:Quails_Jack@yahoo.com">Quails_Jack@yahoo.com</a></td>
</tr>
<tr>
<td>105</td>
<td>Jones</td>
<td>E</td>
<td>Simon</td>
<td>2388</td>
<td>Checking</td>
<td>Brighton</td>
<td><a href="mailto:Jones_Simon@rediffmail.com">Jones_Simon@rediffmail.com</a></td>
</tr>
</tbody>
</table>

### Figure 1-25: An example to demonstrate Referential Integrity

<table>
<thead>
<tr>
<th>Account_No</th>
<th>Transaction_Type</th>
<th>Transaction_Date</th>
<th>Transaction_Amount_In_Dollars</th>
<th>Total_Available_Balance_In_Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>1029</td>
<td>Deposit</td>
<td>12-Jan-2005</td>
<td>5000.00</td>
<td>10000.00</td>
</tr>
<tr>
<td>2348</td>
<td>Withdrawal</td>
<td>14-Jan-2005</td>
<td>2500.00</td>
<td>13500.00</td>
</tr>
<tr>
<td>3421</td>
<td>Deposit</td>
<td>14-Jan-2005</td>
<td>2000.00</td>
<td>27234.00</td>
</tr>
<tr>
<td>2367</td>
<td>Withdrawal</td>
<td>16-Jan-2005</td>
<td>1200.00</td>
<td>12496.00</td>
</tr>
<tr>
<td>2388</td>
<td>Withdrawal</td>
<td>17-Jan-2005</td>
<td>1500.00</td>
<td>8500.00</td>
</tr>
</tbody>
</table>
In the example above, Account_No attribute of Customer_Transaction table is the foreign key referring to Account_No of Customer_Details table. The foreign key attributes can have duplicate values. In the example above, Account_No 1020 occurs in two rows of the table. The foreign key attributes can have NULL values.

The problem of ensuring that the database does not include any invalid foreign key values is therefore known as the referential integrity problem.

The constraint that values of a given foreign key must match the values of the corresponding candidate key is known as a referential constraint.

The relation that contains the foreign key is the referencing relation (also called the child table) and the relation that contains the corresponding candidate key is the referenced relation (also called the parent table).

Invalid foreign key value: A value of 1050 in the Account_No attribute of Customer_Transaction table is invalid because a value of 1050 is not present in the Account_No attribute of the Customer_Details table.

Self Referencing: A table might include a foreign key, the values of which are required to match the value of a candidate key in the same table. This is known as self referencing.

![Employee Table](image)

**Figure 1-26: An example of Self Referencing**

As can be seen from Figure 1-26, each employee belongs to a department. Each department has a manager. For example Cindy S. Atherton, Henry A. George and Matt G. Jackson are the managers of the HR, Finance and Design department respectively. A NULL value means an unknown value or inapplicable value. It does not mean a blank value or zero.
All employees including the manager have a unique Employee_ID. The Manager_ID attribute of the Employee_Manager table can only have any existing value from the Employee_ID attribute. Manager_ID is therefore the foreign key referencing the candidate key, Employee_ID.

**Example:** Assume that the employees in the organization have to undertake a course. Details of the courses are available in the table Course_Description as shown in Figure 1-27. Some courses have a prerequisite, for example an employee must go through the Computer Hardware and System Software Concepts course before taking up the Programming Fundamentals course. Here, the attribute Prerequisite, is the foreign key referencing the candidate key, Course_ID.

![Figure 1-27: Another Example of Self Referencing Table](image)

**Super Key:** Any superset of a candidate key is a super key.

**Example:** Consider the following sets comprising of attributes from the Customer_Details table.
- \{Account_No\}
- \{Account_No, Account_Type\}
- \{Account_No, Account_Type, Bank_Branch\}

\{Account_No\} is a candidate key for the Customer_Details table. \{Account_No, Account_Type\} is a superset of \{Account_No\} and therefore it is a super key for the Customer_Details table. Same is the case for the set \{Account_No, Account_Type, Bank_Branch\}.

A super key may have unnecessary attribute(s).

Although the combination of \{Account_No, Account_Type\} is a super key but \{Account_Type\} is an unnecessary attribute as \{Account_No\} is sufficient to uniquely distinguish between rows in the Customer_Details table.

**Non-Key Attributes:** The attributes other than the primary key attributes in a table/relation are called non-key attributes.
**Example:** `Cust_Last_Name, Cust_Mid_Name, Cust_First_Name, Bank_Branch, etc. are non-key attributes in the Customer_Details table.`

**Points to Remember:**

- A foreign key is a set of attributes of a table, the values of which are required to match values of some candidate key in the same or another table.
- The constraint that values of a given foreign key must match the values of the corresponding candidate key is known as referential constraint.
- A table which has a foreign key referring to its own candidate key is known as self-referencing table.
- The foreign key attribute(s) can have duplicate or null values.
- Any superset of a candidate key is a super key.
- The attributes other than the primary key attributes in a table/ relation are called non-key attributes.
1.15. Summary

- A database is an organized collection of interrelated data
- **Data in the database:**
  - Is integrated
  - Can be shared
  - Can be concurrently accessed
- **The database systems are designed to:**
  - Define structures for the storage of data
  - Provide mechanisms for the manipulation of data
  - Ensure the safety of the data stored, despite system crashes or attempts at unauthorized access
  - Share data among the different users
- **A master file**
  - Stores relatively static data about an entity
  - Changes rarely
- **A transaction file**
  - Stores relatively transient data about a particular data processing task
  - Changes frequently as transactions happen more periodically and in large numbers
- **Disadvantages of the traditional file approach:**
  - Data security
  - Data redundancy
  - Data isolation
  - Program / data dependence
  - Lack of flexibility
  - Concurrent access anomalies
- **DBMS ensures the following:**
  - Data independence
  - Allows for sharing of data among the different users
  - Allows concurrent access to the database
  - Controls redundancy and inconsistency
  - Provides a secure access to the database
  - Enforces integrity constraints by preventing the entry of invalid information into the database
  - Enables backup and recovery from system crashes
- **Centralized Database:** All data is located at a single site
- **Distributed Database:** The database is stored on several computers
- **Three level architecture for a DBMS**
  - **External/View Level:** Enables users to view/access only a part of the database
  - **Conceptual/Logical level:** Describes what data is stored in the database and what relationships exist among those data
  - **Internal/Physical level:** Describes the data storage and access methods
- **DBMS Users:**
- **End Users:** Works at the external level and generally makes updates to the database or executes queries on the database
- **Application Programmer:** Writes application programs
- **Database Administrator:** Defines the conceptual, internal and external schema, control access privileges to/from users and ensures the consistency of the database

**Data Models:** Is a conceptual tool which can be used to describe data, data relationships, data semantics and consistency constraints
- **Object Based Logical Model:** E-R Model
- **Record Based Logical Model:**
  - **Hierarchical Data Model:** IMS
  - **Network Model:** IDMS
  - **Relational Data Model:** Relational data model uses a collection of tables (relations) to represent data and the relationships among those data. *Example:* Oracle, Sybase

**Relational Database:** Any database for which the logical organization is based on relational data model

**RDBMS:** A DBMS that manages the relational database

**Keys:**
- A candidate key is a set of one or more attributes that can uniquely identify a row in a given table
- A table can have multiple candidate keys
- Candidate keys are identified during the design phase
- While creating tables the database designer chooses one candidate key from amongst the several available, to serve as a primary key
- It is preferable to select a candidate key with a minimal number of attributes to function as a primary key in the table
- A foreign key is a set of attribute(s) of a table, the values of which are required to match the values of a candidate key in the same table or in a different table
- The constraint that values of a given foreign key must match the values of the corresponding candidate key is known as referential integrity constraint
- A table which has a foreign key referring to its own candidate key is known as self-referencing table
- Any superset of a candidate key is a super key
- The attributes other than the primary key attributes in a table/relation are called non-key attributes
2. Entity-Relationship (E-R) Modeling

“A picture is worth a thousand words”
-Anonymous

2.1. Introduction

Generally the business scenarios are complex in nature. A software application designer\textsuperscript{21} who is not an expert in a particular business domain may fail to capture the exact business requirement to build a software application. This is one of the prominent causes of software project failure. Figure 2-1 explains how miscommunications between different users could create chaos in software development projects.

\textsuperscript{21}Software application designer: The person who designs software applications.
Reviewing of requirement specification\textsuperscript{22} document with the business users\textsuperscript{23} (who are experts in their respective business domains) will not yield the expected results because of the following reasons:

- The application designer usually writes the requirement specification documents using software technology jargons\textsuperscript{24} which are difficult to understand by the business users.

- Usually these documents are quite lengthy and the business users will not be able to devote enough time to read and review the complete document.

It is always better to represents all the business rules\textsuperscript{25} in pictorial format so that the business users can understand and review the business rules easily and correctly.

One such technique, which is commonly used for designing of the databases, is Entity-Relationship Modeling (E-R Modeling). The diagram used in this technique is called Entity Relationship Diagram (ERD).

In Infosys, 60\% to 70\% of projects use this technique to capture the requirement specification for the database application design and development.

Entity Relationship Diagram (ERD) was first defined in 1976 by Peter Chen. Since then Charles Bachman and James Martin have added some small refinements to the basic ERD principles. Due to its simplicity and ease of use, this technique attracted considerable attention during early 1990’s in both industry and research community.

### 2.2. Entity and Relationship

Before learning E-R diagrams technique in detail let us understand Entity and Relationship.

**Entity**

Entity is a common word anything real or abstract\textsuperscript{26}, about which we want to store data. Entity types fall into five categories: roles, events, locations, tangible\textsuperscript{27} things or concepts.

Some examples of entities are employee, hockey match, campus, book and department. Department is an entity, and Education & Research, HR, Finance, etc., are instances\textsuperscript{28} of the department entity.

---

\textsuperscript{22} Requirement specification: A document which contains requirement for a specific application.

\textsuperscript{23} Business users: The users who owns the application.

\textsuperscript{24} Jargons: The specialized or technical language of a trade, profession, or similar group.

\textsuperscript{25} Business rules: The rules/policies which govern the functioning of the application.

\textsuperscript{26} Abstract: Conceptual/ theoretical object.

\textsuperscript{27} Tangible: Physical object.

\textsuperscript{28} Instance: Occurrence.
Similarly Henry, Luther, Crystal, Jane etc., are instances of **employee** entity.

**Attribute**

An attribute is a characteristic property of an entity. An entity could have multiple attributes.

**Example:** For an entity car, the attributes would be the color, model number, number of doors, right or left hand drive etc.

**Relationship**

Relationship is a natural association that exists between one or more entities.

**Example:** Employee **borrows** books from the library.

### 2.3. Cardinality of a relationship

Cardinality of relationship defines the type of relationship between two participating entities.

**Example:** One employee can take many books from library. One book can be taken by only one employee. Cardinality of relationship between employee and book is “**one to many**”.

One person can sit on only one chair at any point of time. One chair can accommodate only one person in a given point of time. This relationship has “**one to one**” cardinality.

**Note:** Cardinality of relationship is different from cardinality of Relation (Cardinality of relation was discussed in chapter one which refers to number of rows in given relation).

There are **four** types of cardinality relationship.

#### 2.3.1. One to One Relationship

In this relationship, one instance of entity is related to another instance of the entity. Both participating entities have a one to one relationship.

**Example1:** One person (P1,P2,P3,P4) can sit on only one chair at any point of time. And also one chair (C1,C2,C3,C4) can accommodate a
maximum of one person at any given time. In this relationship both the participating entities have one-to-one relationship.

![Figure 2-2: One to One Relationship](image)

**Example2:** One country can have only one citizen as its president and one citizen can become president of only one country.

### 2.3.2. One to Many Relationship

One instance of entity is related to multiple instance of another entity.

**Example1:** One organization (O1, O2, O3) can have many employees but one employee (E1, E2, E3, E4, E5) can work only for one organization.

![Figure 2-3: One to Many Relationship](image)

**Example2:** One warehouse (W1, W2, W3) can be used to store many parts but one part (P1, P2, P3..) can be stored only in one warehouse. In this example one instance of warehouse accommodates many parts. Hence the relationship between warehouse and part is one-to-many.

![Figure 2-4: Another example of One to Many Relationship](image)
2.3.3. Many to One Relationship

This is the reverse of the one to many relationship.

Example: Many employees (E1, E2, E3...) can work for only one department but one department (D1, D2, D3) can have many employees. The relationship between employee and department is many to one.

![Figure 2-5: Many to One Relationship](image)

2.3.4. Many to Many Relationship

In this many to many relationship multiple instances of one Entity are related to multiple instances of another Entity.

Example1: One student (S1, S2, S3, S4) is enrolled for many courses (C1, C2, C3, C4) and one course is enrolled by many students.

![Figure 2-6: Many to Many Relationship](image)

Example2: One student (S1, S2, S3, S4) trained by many instructors (I1, I2, I3, I4) and one instructor trains many students.

![Figure 2-7: Another example of Many to Many Relationship](image)
Many to many relationship is superset of all the above mentioned relationships. All other relationships are special case of **many to many** relationships.

### 2.4. E-R Diagram Notations

**Entity**

An entity is an object or concept about which business user wants to store information.

**Weak entity**

A weak entity is dependent on another entity to exist. Example bank branch depends upon bank name for its existence. Without bank name it is impossible to identify bank branch uniquely.

**Attributes**

Attributes are the properties or characteristics of an entity.

**Key attribute**

A key attribute is the unique, distinguishing characteristic of the entity. For example, an employee’s employee number might be the employee’s key attribute.

**Multivalued attribute**

A multivalued attribute can have more than one value. For example, an employee entity can have multiple skill values.

**Derived attribute**

A derived attribute is based on another attribute. For example, an employee’s monthly salary is based on the employee’s basic salary and house rent allowance.

**Relationships**

Relationships illustrate how two entities share information in the database structure.
Cardinality
Cardinality specifies how many instances of an entity relate to one instance of another entity. M, N both represent ‘MANY’ and 1 represents ‘ONE’ cardinality.

Recursive relationship
In some cases, entities can be self-linked. For example, employees can supervise other employees.

Figure 2-8: E-R Diagram Notations
2.5. Modeling using E-R Diagrams

A *model* is an abstract form of any system or process that hides the unnecessary details, while highlighting those details important to the application. We have noticed the model of huge campuses or buildings which help to visualize the structure, before they are built. On similar lines, we can also model our software applications before they are developed. This will help the business users to visualize the application before it is developed and suggest changes, if it is not as per their requirement.

Modeling the databases using E-R diagrams is called as E-R Modeling. This technique is also called as **Top-Down approach**, because one need **not** identify all the attributes to model the system using this technique.

2.5.1. Steps in E-R Modeling

Usually the following six steps are followed to generate E-R Models.

a) **Identify the entities**: Look for general nouns in requirement specification document which are of business interest to business users

b) **Find relationships**: Identify the natural relationship and their cardinalities between the entities

c) **Identify the key attributes for every entity**: Identify the attribute or set of attributes which can identify instance of entity uniquely

d) **Identify other relevant attributes**: Identify other attributes which are interest to business users and want to store the information in database

e) **Complete E-R diagram**: Draw complete E-R diagram with all attributes including primary key

f) **Review your results with your business users** - Look at the list of attributes associated with each entity to see if anything has been omitted.

Note that while this is an **iterative** approach and one cannot come to a final E-R model in a single step. It requires a great deal of patience and numerous revisions before the model is created.

---

29 *Model*: A representation or a scaled down structure of an object.

30 *Iterative*: process of repeating the same task.
2.6. Case Study 1: Problem Statement

Let us apply the above methodology to model, University database application.

- An university has many departments
- Each department has multiple instructors; one among them is the head of the department
- An instructor belongs to only one department
- Each department offers multiple courses, each of which is taught by a single instructor
- A student may enroll for many courses offered by different departments

2.6.1. Case Study 1: Solution

Step 1: Identify the Entities
Generally the entities will have multiple instances in a given business scenario. As per this guideline, we can identify the following entities:

1. DEPARTMENT
2. COURSE
3. INSTRUCTOR
4. STUDENT

“Head of the department” is NOT an Entity. It is a relationship between the Instructor and department entities.

Note: One may be tempted to identify “university” as an entity, but it is a false entity because it has only one instance.

Step 2: Find relationships
We can derive the following relationships:

1. The department offers multiple courses and each course belongs to only one department, hence cardinality between department and course is one to many.

   ![Diagram](Diagram1)

   - Department
   - Offers
   - Course

2. One course is enrolled by multiple students and one student enrolls for multiple courses, hence the relationship is many to many.
3. One department has multiple instructors and one instructor belongs to one and only one department, hence the relationship is one to many.

4. Each department has one “Head of Department” and one Instructor is “Head of Department” for only one department, hence the relationship is one to one.

5. One course is taught by only one instructor but one instructor teaches many courses, hence the relationship between course and instructor is many to one.

The relationship between instructor and student need **NOT** be defined for following reasons:
1. There is no business significance in this relationship.
2. We can always derive this relationship indirectly through course and instructor, and course and students.

**Step 3: Identify the key attributes**
1. **DName** (Department Name) which identifies the department uniquely will be the key attribute for “DEPARTMENT” entity.
2. **STUDENT#** (Student Number) is the key attribute for “STUDENT” entity.
3. **IName** (Instructor Name) is the key attribute for “INSTRUCTOR” entity.
4. **COURSE#** (Course Number) is the key attribute for COURSE entity.

**Step 4: Identify other relevant attributes**
1. For the “Department” entity, the relevant attribute other than “Department Name” is “Location”.
2. For the “Course” entity, the relevant attributes other than “Course Number” are “Course Name”, “Duration” and “Pre Requisite”.
3. For the “Instructor” entity, the relevant attributes other than “Instructor Name” are “Room Number” and “Telephone Number”.
4. For the “Student” entity, the relevant attributes other than “Student Number” are “Student Name” and “Date of Birth”.

**Step 5: Complete E-R diagram**

After considering all the above mentioned guidelines one can generate the E-R Model for the university database as shown in Fig 2-9.

![Figure 2-9: E-R Diagram for University](image-url)
2.7. Case Study 2: Problem Statement

Let us consider a university library scenario for developing the E-R model.

Assume in a university

- There are multiple libraries and each library has multiple student members
- Students can become members to multiple libraries by paying appropriate membership fee
- Each library has its own set of books. Within the library these books are identified by a unique number
- Students can borrow multiple books from subscribed library
- Students can order books using inter-library loan. This can be useful if a student wishes to borrow books from a library where s/he is not a member. The student orders the books through a library where s/he is a member

2.7.1. Case Study 2: Solution

Step 1: Identify the entities

Generally the entities will have multiple instances in a given business scenario. As per this guideline, we can identify the following entities:

1. LIBRARY
2. BOOK
3. STUDENT

In this business scenario BOOK is a weak entity because without knowing the library details, book cannot be identified independently. Book is always associated with its library.

Step 2: Find Relationships

We can derive the following relationships:

1. One library has many member students and each student can become member of many libraries, hence the cardinality between library and student is many to many.
2. One book belongs to only one library and one library can have multiple books, hence cardinality between library and book is one to many.
3. One library can loan multiple books and each book can be loaned to only one library, hence the cardinality between library and book is one to many.
4. One student can borrow multiple books and one book can be borrowed by only one student, hence the cardinality between student and book is many to one.

Step 3: Identify the key attributes

1. Library# (library ID) is the key attribute for the entity “Library”, as it identifies the library uniquely.
2. Book# (book ID) and Library# are together key attributes for “Book” entity.
3. Student# (student number) is the key attribute for “Student” entity.
Step 4: Identify other relevant attributes

1. For the “Library” entity, the relevant attributes other than “Library#” would be “Library Name” and “Location”.
2. For the “Book” entity, the relevant attributes other than “Book#” would be “Title” and “ISBN”.
3. For the “Student” entity, the relevant attribute other than “Student#” would be “Student Name” and “Date of Birth”.

Step 5: Complete E-R diagram

After considering all the above mentioned guidelines, one can generate the E-R Model for above mentioned university library business scenario as shown in Figure 2-10.

![Figure 2-10: E-R Diagram for University Library](image)

2.8. Case Study 3: Problem Statement

Let us consider a banking business scenario for developing the E-R model.
Assume in a city
- There are multiple banks and each bank has many branches. Each branch has multiple customers
- Customers have various types of accounts
- Some customers also had taken different types of loans from these bank branches
- One customer can have multiple accounts and loans

2.8.1. Case Study 3: Solution

**Step 1: Identify the entities**
Generally the entities will have multiple instances in a given business scenario. As per this guideline, we can identify the following entities:

1. BANK
2. BRANCH
3. LOAN
4. ACCOUNT
5. CUSTOMER

BRANCH is considered a weak entity because without knowing the BANK, we cannot define the BRANCH independently. BRANCH is always associated with its BANK name.

*Example:* Citi bank Branch, ICICI Bank Branch, Wells Fargo Branch or State Bank of India branch.

**Note:** One may be tempted to identify “City” as an entity, but it is a false entity because it has only one instance.

**Step 2: Find Relationships**
We can derive the following relationships:

1. One bank has many branches and each branch belongs to only one bank, hence the cardinality between bank and branch is one to many.
2. One branch offers many loans and each loan is associated with one branch, hence the cardinality between branch and loan is one to many.
3. One branch maintains multiple accounts and each account is associated to one and only one branch, hence the cardinality between branch and account is one to many.
4. One Loan can be availed by multiple customers, and each customer can avail multiple loans, hence the cardinality between loan and customer is many to many.
5. One customer can hold multiple accounts, and each account can be held by multiple customers, hence the cardinality between customer and account is many to many.
Step 3: Identify the key attributes
1. BankCode (Bank Code) is the key attribute for the entity “Bank”, as it identifies the bank uniquely.
2. Branch# (Branch Number) and BankCode (Bank Code) are together key attributes for “Branch” entity.
3. Customer# (Customer Number) is the key attribute for “Customer” entity.
4. Loan# (Loan Number) is the key attribute for “Loan” entity.
5. Account# (Account Number) is the key attribute for “Account” entity.

Step 4: Identify other relevant attributes
1. For the “Bank” entity, the relevant attributes other than “BankCode” would be “Name” and “Address”.
2. For the “Branch” entity, the relevant attributes other than “Branch#” would be “Name” and “Address”.
3. For the “Loan” entity, the relevant attribute other than “Loan#” would be “Loan Type”.
4. For the “Account” entity, the relevant attribute other than “Account#” would be “Account Type”.
5. For the “Customer” entity, the relevant attributes other than “Customer#” would be “Name”, “Phone” and “Address”.

Step 5: Complete E-R diagram
After considering all the above mentioned guidelines, one can generate the E-R Model for above mentioned banking business scenario as shown in Figure 2-11.
Figure 2-11: E-R Diagram for Bank
2.9. Transforming an E-R Model into Physical Database Design

E-R model helps mainly in capturing and analyzing the requirements. It can also be used during the design of the physical database. The following is a set of guidelines for converting an E-R model into a physical database design.

1. Each entity represented in the E-R model can be defined as a table in the relational schema. All attributes of the entity will become columns of the table. As per this guideline we can translate BANK, BRANCH, LOAN, ACCOUNT and CUSTOMER entities to following tables. Additional columns can be added to the below tables as per the business requirements at the later stage.

<table>
<thead>
<tr>
<th>BANK</th>
<th>BRANCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>BankCode</td>
<td>BankCode</td>
</tr>
<tr>
<td>Name</td>
<td>Branch#</td>
</tr>
<tr>
<td>Address</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td>Address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOAN</th>
<th>ACCOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loan#</td>
<td>Account#</td>
</tr>
<tr>
<td>LoanType</td>
<td>AccountType</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUSTOMER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer#</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Telephone#</td>
</tr>
<tr>
<td>Address</td>
</tr>
</tbody>
</table>

Figure 2-12: Entity based tables

Weak entity types are converted into a table of their own, with the primary key of the strong entity acting as a foreign key in the table. This foreign key along with the key of the weak entity form the composite primary key of this table. As per this guideline, a “Branch” table is created with the above mentioned structure, with BankCode and Branch# together as composite primary key.
2. Each relationship can be defined as separate table in relational schema. Key attributes of participating entities\(^{31}\) will become key attribute of the relationship. As per this guideline we can define LOAN_OFFERING_DETAILS table between BRANCH and LOAN entities, BRANCH_ACCOUNTDETAILS between BRANCH and ACCOUNT entities, LOAN_DETAILS table between LOAN and CUSTOMER entities, ACCOUNT_DETAILS tables between ACCOUNT and CUSTOMER entities. BANK and BRANCH relationship table is not defined because this information is already captured in BRANCH table. Usually relationship based tables will have their own attributes in addition to prime attributes of participating entities. For example, LOAN_DETAILS table contain prime attributes from LOAN and CUSTOMER table (which together act as composite primary key) in addition to other attributes such as DateofSanction, IntRate, LoanAmount, Duration etc.

<table>
<thead>
<tr>
<th>LOAN_OFFERING_DETAILS</th>
<th>BRANCH_ACCOUNT_DETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BankCode</td>
<td>Branch#</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOAN_DETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loan#</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACCOUNT_DETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account#</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Figure 2-13: Relationship based tables

\(^{31}\) Participating entities: The entities which are joined by the relation.
Note: At this stage entities and relationships are converted to tables hence table does not have any data.

Actual database table designs are driven by business requirements. The principles of data base design from ERD might be subjected to small changes depending on the requirements. We will be exposed to these details once we get a real life project experience. Some of the aspects we might have to keep in mind are:

- In one to one and one to many cases, we may not always have separate tables for the participating entities and their relationship. One combined table for both the participating entities and related attributes of relationship may be sufficient.
- In a many to many relationship, it is mandatory to create separate tables for participating entities and relationships. For example, entities and relationship defined in Figure 2-11, CUSTOMER and LOAN entities have a many to many relationship. Hence one should create separate tables for CUSTOMER, LOANS and LOAN DETAILS. Here LOAN DETAILS refers to relationship table.

2.10. Merits and Demerits of E-R Modeling

The following sections discuss the merit and demerits of E-R modeling.

2.10.1. Merits of E-R Modeling

2. Intuitive\(^{32}\) and helps in physical database creation.
3. Can be generalized and specialized based on needs.
5. Gives a higher level abstraction of the system.

\(^{32}\) Intuitive: Natural
2.10.2. Demerits of E-R Modeling

1. Physical design derived from E-R Model may have some amount of ambiguities or inconsistency.
2. Sometime diagrams may lead to misinterpretations.

*Example:*

In a real situation, there could be several types of borrowing, for example long-term, normal and short-term. It is not immediately clear whether the above diagram represents all or some of these only. If this aspect is not clarified, then people could come to a wrong conclusion. Giving proper description of the relationship is extremely important for ensuring better understanding.

In the following set of figures, the relationship is described clearly to overcome misinterpretation.
2.11. SUMMARY

- Most of the application errors are because of miscommunication between the application user and the designer and between the designer and the developer.
- It is always better to represent business findings in terms of picture to avoid miscommunication.
- It is practically impossible to review the complete requirement document by business users.
- An E-R diagram is one of the many ways to represent business findings in pictorial format.
- Four types of cardinality of relationships are:
  a. one to one
  b. one to many
  c. many to one
  d. many to many
- E-R Modeling will also help the database design.
- E-R modeling has some amount of inconsistency and anomalies associated with it.
3. Normalization

“No human investigation can be called real science if it can not be demonstrated mathematically.”

- Leonardo da Vinci

3.1. Introduction

Usually in the software industry designers use E-R modeling as a requirement analysis tool. Database design using E-R diagram is a by-product.

Database designed based on the E-R model may have some amount of inconsistency, ambiguity\(^{33}\) and redundancy. To resolve these issues we have to do some amount of refinement is required. This refinement process is called as Normalization.

As normalization involves building structures (like table/tables), starting from the stage of identifying the columns (attributes) associated in the table, it is also called “Bottom-Up” approach.

This normalization technique is based on a strong mathematical foundation.

Basically normalization eliminates the duplicate data and makes insert, update and delete operations much more efficient in terms of performance and space requirement to store the data.

In Infosys, almost all the database designs are initially based on E-R modeling and later refined using normalization techniques before they are physically created.

3.2. The need for Normalization

Consider a university scenario, where in the data associated with the students, courses and their results are maintained in a table called “Student_Course_Result”.

\(^{33}\) Ambiguity: Uncertainty.
Table: Student_Course_Result Table

<table>
<thead>
<tr>
<th>Student_Details</th>
<th>Course_Details</th>
<th>Result_Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>101 Davis</td>
<td>M4 Applied Mathematics</td>
<td>7</td>
</tr>
<tr>
<td>101 Davis</td>
<td>H6 American History</td>
<td>4</td>
</tr>
<tr>
<td>103 Sandra</td>
<td>C3 Bio Chemistry</td>
<td>11</td>
</tr>
<tr>
<td>104 Evelyn</td>
<td>B3 Botany</td>
<td>8</td>
</tr>
<tr>
<td>102 Daniel</td>
<td>P3 Nuclear Physics</td>
<td>13</td>
</tr>
<tr>
<td>105 Susan</td>
<td>H6 American History</td>
<td>4</td>
</tr>
<tr>
<td>103 Sandra</td>
<td>B4 Zoology</td>
<td>5</td>
</tr>
<tr>
<td>105 Susan</td>
<td>P3 Nuclear Physics</td>
<td>13</td>
</tr>
<tr>
<td>104 Evelyn</td>
<td>M4 Applied Mathematics</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 3-1: Data file in table format

If we observe the table shown in Figure 3-1 closely, we would find that the table has many anomalies. They are:

**Insert Anomaly**

In some cases, insertion of new data is difficult.

*Example:* We cannot insert prospective course which does not have any registered student or we cannot insert student details who is yet to register for any course.

**Update Anomaly**

In some cases, updation of existing data is difficult.

*Example:* If we want to update the course M4’s name we need to do this operation three times. Similarly we may have to update student 103’s name twice if it changes.

**Delete Anomaly**

In some cases, deletion of existing data is not possible.

*Example:* If we want to delete a course M4, in addition to M4 course details, other critical details of student also will be deleted. This kind of deletion is harmful to business. Moreover, M4 appears thrice in above table and needs to be deleted thrice.

**Duplicate Data**

The table has lots of duplicate data.

*Example:* Course M4’s data is stored thrice and student 102’s data stored twice. This redundancy will increase as the number of course offering and students increases.

Hence we need to refine our design so that we make an efficient database in terms of storage space and Inserts, Updates and Deletes operations. This refining technique is called as normalization.

34 Anomalies: Irregularities.
### 3.3. Process of Normalization

As mentioned previously, normalization technique is based on strong mathematical foundation.

Basically in software industry four normal forms are used to design the database.

Before getting to know the normalization techniques in detail, let us define a few building blocks which are used to define normal forms.

#### 3.3.1. Determinant

Attribute X can be defined as determinant if it uniquely defines the attribute value Y in a given relationship or entity. To qualify as determinant attribute need **NOT** be a key attribute. Usually dependency of an attribute is represented as $X \rightarrow Y$, which means attribute $X$ decides attribute $Y$.

**Example:** In `RESULT` relation, Marks attribute may decide the grade attribute. This is represented as $Marks \rightarrow Grade$ and read as Marks decides Grade.

![Figure 3-2: Determinant](image)

In the `RESULT` relation, Marks attribute is not a key attribute. Hence it can be concluded that key attributes are determinants but not all the determinants are key attributes.

#### 3.3.2. Functional Dependency

Consider the following Relation

$\text{REPORT} (\text{Student\#}, \text{Course\#}, \text{CourseName}, \text{IName}, \text{Room\#}, \text{Marks}, \text{Grade})$

Where:

- **Student\#** - Student Number
- **Course\#** - Course Number
- **CourseName** - Course Name
- **IName** - Name of the instructor who delivered the course
- **Room\#** - Room number which is assigned to respective instructor
- **Marks** - Scored in course Course\# by student Student\#
- **Grade** - Obtained by student Student\# in course Course\#
Student# Course# together (called composite attribute) defines EXACTLY ONE value of marks. This can be symbolically represented as

$$ \text{Student# Course#} \rightarrow \text{Marks} $$

This type of dependency is called as functional dependency. In above example marks is functionally dependent on Student# Course#.

Other functional dependencies in above examples are:
- Course# → CourseName,
- Course# → IName (Assuming one course is taught by one and only one instructor)
- IName → Room# (Assuming each instructor has his/her own and non-shared room)
- Marks → Grade.

Formally we can define functional dependency as: In a given relation R, X and Y are attributes. Attribute Y is functionally dependent on attribute X if each value of X determines EXACTLY ONE value of Y. This is represented as:

$$ X \rightarrow Y $$

However X may be composite in nature.

### 3.3.3. Full Functional Dependency

In above example Marks is fully functionally dependent on Student# Course# and not on sub set of Student# Course#. This means Marks can not be determined either by Student# OR Course# alone. It can be determined only using Student# AND Course# together. Hence Marks is fully functionally dependent on Student# Course#.

CourseName is not fully functionally dependent on Student# Course# because one of the subset Course# determines the CourseName and Student# does not have any role in deciding CourseName. Hence CourseName is not fully functionally dependent on Student# Course#.

![Figure 3-3: Full Functional Dependency](image-url)
Formal definition of full functional dependency is: In a given relation $R$, $X$ and $Y$ are attributes. $Y$ is fully functionally dependent on attribute $X$ only if it is not functionally dependent on sub-set of $X$. However $X$ may be composite in nature.

### 3.3.4. Partial Dependency

In the above relationship $\text{CourseName}$, $\text{IName}$, $\text{Room\#}$ are partially dependent on composite attributes $\text{Student\# Course\#}$ because $\text{Course\#}$ alone defines the $\text{CourseName}$, $\text{IName}$, $\text{Room\#}$.

![Figure 3-4: Partial Dependency](image)

Formal definition of partial dependency is: In a given relation $R$, $X$ and $Y$ are attributes. attribute $Y$ is partially dependent on the attribute $X$ only if it is dependent on subset of attribute $X$. However $X$ may be composite in nature.

### 3.3.5. Transitive Dependency

In above example, $\text{Room\#}$ depends on $\text{IName}$ and in turn $\text{IName}$ depends on $\text{Course\#}$. Hence $\text{Room\#}$ transitively depends on $\text{Course\#}$.

![Figure 3-5: Transitive Dependency](image)

Similarly $\text{Grade}$ depends on $\text{Marks}$, in turn $\text{Marks}$ depends on $\text{Student\# Course\#}$ hence $\text{Grade}$ fully transitively $^{35}$ depends on $\text{Student\# Course\#}$.

### 3.3.6. Key attributes

In a given relationship $R$, if the attribute $X$ uniquely defines all other attributes, then the attribute $X$ is a Key attribute which is nothing but the candidate key which is defined in Chapter One.

---

$^{35}$ Transitive: In-direct
**Example 1:** Student # Course# together is a composite key attribute which determines all attributes in relationship REPORT (Student#, Course#, CourseName, IName, Room#, Marks, Grade) uniquely. Hence Student# and Course# are key attributes.

**Example 2:** Student # and EMailID also can be considered as candidate keys for entity student STUDENT(Student#, Student Name, Date of Birth, EMailID). Student # or EMailID uniquely defines all other attributes of student entity.

### 3.3.7. Non key attributes

In a given relationship R, all the attributes which are not key attributes are considered as non-key attributes.

### 3.4. Types of Normal Forms

#### 3.4.1. First Normal Form (1 NF)

A relation R is said to be in the first normal form (1NF) if and only if all the attributes of the relation R are atomic\(^{36}\) in nature.

Consider the Student_Course_Result table which is reproduced from the Section 3.2: The need for Normalization.

<table>
<thead>
<tr>
<th>Student_Details</th>
<th>Course_Details</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>104 Evelyn 2/22/1986</td>
<td>B3 Botany 8</td>
<td>11/26/2004 77 B</td>
</tr>
<tr>
<td>105 Susan 8/31/1985</td>
<td>P3 Nuclear Physics Basic Physics 13</td>
<td>11/12/2004 69 A</td>
</tr>
<tr>
<td>105 Susan 8/31/1985</td>
<td>H6 American History 4</td>
<td>11/22/2004 87 A</td>
</tr>
</tbody>
</table>

Figure 3-6: Data file in table format

Table shown in Figure 3-6, Student_Details, Course_Details and Results attributes can be further divided. Student_Details attribute is divided into Student# (Student Number), StudentName (Student Name) and DateofBirth (Date of Birth). Course_Details attribute is divided into Course# (Course Number), CourseName,

\(^{36}\) Atomic: the smallest level to which data may be broken down and remain meaningful
To make above table 1NF compliant, it is re-designed as shown below.

<table>
<thead>
<tr>
<th>Student#</th>
<th>Student Name</th>
<th>Dateof Birth</th>
<th>Course #</th>
<th>CourseName</th>
<th>Pre Requisite</th>
<th>Duration InDays</th>
<th>DateOf Exam</th>
<th>Marks</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Davis</td>
<td>04-Nov-1986</td>
<td>M4</td>
<td>Applied Mathematics</td>
<td>Basic Mathematics</td>
<td>7</td>
<td>11-Nov-2004</td>
<td>82</td>
<td>A</td>
</tr>
<tr>
<td>101</td>
<td>Davis</td>
<td>04-Nov-1986</td>
<td>H6</td>
<td>American History</td>
<td></td>
<td>4</td>
<td>22-Nov-2004</td>
<td>79</td>
<td>B</td>
</tr>
<tr>
<td>104</td>
<td>Evelyn</td>
<td>22-Feb-1986</td>
<td>B3</td>
<td>Botany</td>
<td>Basic Physics</td>
<td>8</td>
<td>26-Nov-2004</td>
<td>77</td>
<td>B</td>
</tr>
<tr>
<td>102</td>
<td>Daniel</td>
<td>05-Nov-1986</td>
<td>P3</td>
<td>Nuclear Physics</td>
<td>Basic Physics</td>
<td>13</td>
<td>12-Nov-2004</td>
<td>68</td>
<td>B</td>
</tr>
<tr>
<td>105</td>
<td>Susan</td>
<td>31-Aug-1985</td>
<td>P3</td>
<td>Nuclear Physics</td>
<td>Basic Physics</td>
<td>13</td>
<td>12-Nov-2004</td>
<td>89</td>
<td>A</td>
</tr>
<tr>
<td>105</td>
<td>Susan</td>
<td>31-Aug-1985</td>
<td>H6</td>
<td>American History</td>
<td></td>
<td>4</td>
<td>22-Nov-2004</td>
<td>87</td>
<td>A</td>
</tr>
<tr>
<td>104</td>
<td>Evelyn</td>
<td>22-Feb-1986</td>
<td>M4</td>
<td>Applied Mathematics</td>
<td>Basic Mathematics</td>
<td>7</td>
<td>11-Nov-2004</td>
<td>65</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure 3-7: First Normal Form

In the new form, all the attributes are atomic, meaning they are not further decomposable. You cannot divide Student#, StudentName etc further into smaller attributes. Hence this table is in 1NF.

Let us re-visit the issues we had with un-normalized table. Even at this stage, it is difficult to add prospective course or student information. Still it is difficult to update or delete either Course or Student information. Hence anomalies in inserts, updates and deletes are still to be resolved.

Unfortunately first normal form has all the problems which we faced in un-normalized table.

3.4.2. Second Normal Form (2 NF)

A Relation is said to be in Second Normal Form if and only if:
- It is in the First normal form, and
- No partial dependency exists between non-key attributes and key attributes.

Let us re-visit 1NF table structure.

---

37 Decomposable: further split or reduce.
- Student# is key attribute for Student.
- Course# is key attribute for Course.
- Student# Course# together form the composite key attributes for Result relationship.
- Other attributes like StudentName (Student Name), DateofBirth, CourseName, PreRequisite, DurationInDays, DateofExam, Marks and Grade are non-key attributes.

To make this table 2NF compliant, we have to remove all the partial dependencies.
- StudentName and DateofBirth depends only on Student#.
- CourseName, PreRequisite and DurationInDays depends only on Course#.
- DateofExam depends only on Course#.

To remove this partial dependency we need to split Student_Course_Result table into four separate tables, STUDENT, COURSE, RESULT and EXAM_DATE tables as shown in Figure 3-8.

**STUDENT TABLE**

<table>
<thead>
<tr>
<th>Student#</th>
<th>StudentName</th>
<th>DateofBirth</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Davis</td>
<td>04-Nov-1986</td>
</tr>
<tr>
<td>102</td>
<td>Daniel</td>
<td>06-Nov-1987</td>
</tr>
<tr>
<td>103</td>
<td>Sandra</td>
<td>02-Oct-1988</td>
</tr>
<tr>
<td>104</td>
<td>Evelyn</td>
<td>22-Feb-1986</td>
</tr>
<tr>
<td>105</td>
<td>Susan</td>
<td>31-Aug-1985</td>
</tr>
<tr>
<td>106</td>
<td>Mike</td>
<td>04-Feb-1987</td>
</tr>
<tr>
<td>107</td>
<td>Juliet</td>
<td>09-Nov-1986</td>
</tr>
<tr>
<td>108</td>
<td>Tom</td>
<td>07-Oct-1986</td>
</tr>
<tr>
<td>109</td>
<td>Catherine</td>
<td>06-Jun-1984</td>
</tr>
</tbody>
</table>

**COURSE TABLE**

<table>
<thead>
<tr>
<th>Course#</th>
<th>CourseName</th>
<th>PreRequisite</th>
<th>DurationInDays</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>Basic Mathematics</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>M4</td>
<td>Applied Mathematics</td>
<td>M1</td>
<td>7</td>
</tr>
<tr>
<td>H6</td>
<td>American History</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>C1</td>
<td>Basic Chemistry</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>C3</td>
<td>Bio Chemistry</td>
<td>C1</td>
<td>11</td>
</tr>
<tr>
<td>B3</td>
<td>Botany</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>P1</td>
<td>Basic Physics</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>P3</td>
<td>Nuclear Physics</td>
<td>P1</td>
<td>13</td>
</tr>
<tr>
<td>B4</td>
<td>Zoology</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

**RESULT Table**

<table>
<thead>
<tr>
<th>Student#</th>
<th>Course#</th>
<th>Marks</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>M4</td>
<td>82</td>
<td>A</td>
</tr>
<tr>
<td>102</td>
<td>M4</td>
<td>62</td>
<td>C</td>
</tr>
<tr>
<td>101</td>
<td>H6</td>
<td>79</td>
<td>B</td>
</tr>
<tr>
<td>103</td>
<td>C3</td>
<td>65</td>
<td>B</td>
</tr>
<tr>
<td>104</td>
<td>B3</td>
<td>77</td>
<td>B</td>
</tr>
<tr>
<td>102</td>
<td>P3</td>
<td>68</td>
<td>B</td>
</tr>
<tr>
<td>105</td>
<td>P3</td>
<td>89</td>
<td>A</td>
</tr>
<tr>
<td>103</td>
<td>B4</td>
<td>54</td>
<td>D</td>
</tr>
<tr>
<td>105</td>
<td>H6</td>
<td>87</td>
<td>A</td>
</tr>
<tr>
<td>104</td>
<td>M4</td>
<td>65</td>
<td>B</td>
</tr>
</tbody>
</table>

**EXAM_DATE Table**

<table>
<thead>
<tr>
<th>Course#</th>
<th>DateOfExam</th>
</tr>
</thead>
<tbody>
<tr>
<td>M4</td>
<td>11-Nov-2004</td>
</tr>
<tr>
<td>H6</td>
<td>22-Nov-2004</td>
</tr>
<tr>
<td>C3</td>
<td>16-Nov-2004</td>
</tr>
<tr>
<td>B3</td>
<td>26-Nov-2004</td>
</tr>
<tr>
<td>P3</td>
<td>12-Nov-2004</td>
</tr>
<tr>
<td>B4</td>
<td>27-Nov-2004</td>
</tr>
</tbody>
</table>

Figure 3-8: Second Normal Form
In the first table (STUDENT), the key attribute is Student# and all other non-key attributes, StudentName and DateofBirth are fully functionally dependant on the key attribute.

In the second table (COURSE), Course# is the key attribute and all the non-key attributes, CourseName, PreRequisite and DurationInDays are fully functionally dependant on the key attribute.

In third table (RESULT) Student# Course# together are key attributes and all other non key attributes, Marks and Grade are fully functionally dependant on the key attributes.

In the fourth table (EXAM_DATE) Course# is the key attribute and the non-key attribute, DateOfExam is fully functionally dependant on the key attribute.

These four tables are also compliant with the First Normal Form definition. Hence these four tables are in Second Normal Form (2NF).

At first look it appears like all our anomalies are taken away! Now we are storing Student 103 and M4 record only once. We can insert prospective students and courses at our will. We will update only once if we need to change any data in STUDENT, COURSE tables. We can get rid of any course or student details by deleting just one row.

Let us analyze the following table.

<table>
<thead>
<tr>
<th>Student#</th>
<th>Course#</th>
<th>Marks</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>M4</td>
<td>82</td>
<td>A</td>
</tr>
<tr>
<td>102</td>
<td>M4</td>
<td>62</td>
<td>C</td>
</tr>
<tr>
<td>101</td>
<td>H6</td>
<td>79</td>
<td>B</td>
</tr>
<tr>
<td>103</td>
<td>C3</td>
<td>65</td>
<td>B</td>
</tr>
<tr>
<td>104</td>
<td>B3</td>
<td>77</td>
<td>B</td>
</tr>
<tr>
<td>102</td>
<td>P3</td>
<td>68</td>
<td>B</td>
</tr>
<tr>
<td>105</td>
<td>P3</td>
<td>89</td>
<td>A</td>
</tr>
<tr>
<td>103</td>
<td>B4</td>
<td>54</td>
<td>D</td>
</tr>
<tr>
<td>105</td>
<td>H6</td>
<td>87</td>
<td>A</td>
</tr>
<tr>
<td>104</td>
<td>M4</td>
<td>65</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure 3-9: RESULT Table

We already concluded that:
- All the attributes are atomic in nature.
- No partial dependency exists between the key attributes and non-key attributes.
- RESULT table is in **Second Normal form (2NF)**

Assume, at present, as per the university evaluation policy,
- Students who score more than or equal to 80 marks are awarded with “A” grade.
- Students who score more than or equal to 65 up till 79 gets “B” grade.
- Students who score marks more than or equal to 50 up till 64 fetches “C” grade.
- Students who score marks less than 50 is only “D” grade.
The university management which is committed to improve the quality of education, wants to change the existing grading system to a new grading system as given below.

- “A+” grade for 95 and above
- “A” grade for 85 to 94
- “B” grade for 70 to 84
- “B-” grade for 65 to 69
- “C” grade for 55 to 64
- “D” grade for 45 to 54
- “E” grade for less than 40

In the present RESULT table structure,
- We do not have an option to introduce new grades like A+, B- and E.
- We need to do multiple updates on the existing records to bring them to the new grading definition.
- We will not be able to take away “D” grade if we want to.
- 2NF does not take care of all the anomalies and inconsistencies.

### 3.4.3. Third Normal Form (3 NF)

A relation R is said to be in the **Third Normal Form (3NF)** if and only if
- It is in 2NF and
- No transitive dependency exists between non-key attributes and key attributes

In the above RESULT table Student# and Course# are the key attributes. All other attributes, except grade are non-partially, non-transitively dependent on key attributes. The “Grade” attribute is dependant on “Marks” and in turn “Marks” is dependent on Student# Course#. To bring this table to third normal form we need to take off this **transitive dependency**.

After taking this transitive dependency we can infer the following table structures which are in 3NF.

<table>
<thead>
<tr>
<th>Student#</th>
<th>Course#</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>M4</td>
<td>82</td>
</tr>
<tr>
<td>102</td>
<td>M4</td>
<td>62</td>
</tr>
<tr>
<td>101</td>
<td>H6</td>
<td>79</td>
</tr>
<tr>
<td>103</td>
<td>C3</td>
<td>65</td>
</tr>
<tr>
<td>104</td>
<td>B3</td>
<td>77</td>
</tr>
<tr>
<td>102</td>
<td>P3</td>
<td>68</td>
</tr>
<tr>
<td>105</td>
<td>P3</td>
<td>89</td>
</tr>
<tr>
<td>103</td>
<td>B4</td>
<td>54</td>
</tr>
<tr>
<td>105</td>
<td>H6</td>
<td>87</td>
</tr>
<tr>
<td>104</td>
<td>M4</td>
<td>65</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UpperBound</th>
<th>LowerBound</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>95</td>
<td>A+</td>
</tr>
<tr>
<td>94</td>
<td>85</td>
<td>A</td>
</tr>
<tr>
<td>84</td>
<td>70</td>
<td>B</td>
</tr>
<tr>
<td>69</td>
<td>65</td>
<td>B-</td>
</tr>
<tr>
<td>64</td>
<td>55</td>
<td>C</td>
</tr>
<tr>
<td>54</td>
<td>45</td>
<td>D</td>
</tr>
<tr>
<td>44</td>
<td>0</td>
<td>E</td>
</tr>
</tbody>
</table>

**Figure 3-10: Third Normal Form**
After normalizing tables to **Third Normal Form (3NF)**, we got rid off all the anomalies and inconsistencies. Now we can add new grade systems, update the existing one and delete the unwanted ones. Hence the **Third Normal Form** is the most optimal normal form and **99%** of the databases which require efficiency in

- **INSERT**
- **UPDATE** and
- **DELETE**

Operations are designed in this normal form.

### 3.4.4. Boyce Codd Normal Form (BCNF)

A relation is said to be in Boyce Codd Normal Form (BCNF) if and only if all the determinants are candidate keys. BCNF relation is a strong 3NF, but not every 3NF relation is BCNF.

Let us understand this concept using slightly different Result table structure.

<table>
<thead>
<tr>
<th>Student#</th>
<th>EmailID</th>
<th>Course#</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td><a href="mailto:Davis@myuni.edu">Davis@myuni.edu</a></td>
<td>M4</td>
<td>82</td>
</tr>
<tr>
<td>102</td>
<td><a href="mailto:Daniel@myuni.edu">Daniel@myuni.edu</a></td>
<td>M4</td>
<td>62</td>
</tr>
<tr>
<td>101</td>
<td><a href="mailto:Davis@myuni.edu">Davis@myuni.edu</a></td>
<td>H6</td>
<td>79</td>
</tr>
<tr>
<td>103</td>
<td><a href="mailto:Sandra@myuni.edu">Sandra@myuni.edu</a></td>
<td>C3</td>
<td>65</td>
</tr>
<tr>
<td>104</td>
<td><a href="mailto:Evelyn@myuni.edu">Evelyn@myuni.edu</a></td>
<td>B3</td>
<td>77</td>
</tr>
<tr>
<td>102</td>
<td><a href="mailto:Daniel@myuni.edu">Daniel@myuni.edu</a></td>
<td>P3</td>
<td>68</td>
</tr>
<tr>
<td>105</td>
<td><a href="mailto:Susan@myuni.edu">Susan@myuni.edu</a></td>
<td>P3</td>
<td>89</td>
</tr>
<tr>
<td>103</td>
<td><a href="mailto:Sandra@myuni.edu">Sandra@myuni.edu</a></td>
<td>B4</td>
<td>54</td>
</tr>
<tr>
<td>105</td>
<td><a href="mailto:Susan@myuni.edu">Susan@myuni.edu</a></td>
<td>H6</td>
<td>87</td>
</tr>
<tr>
<td>104</td>
<td><a href="mailto:Evelyn@myuni.edu">Evelyn@myuni.edu</a></td>
<td>M4</td>
<td>65</td>
</tr>
</tbody>
</table>

**Figure 3-11: RESULT Table**
In the Figure 3-11 RESULT table, we have two candidate keys namely **Student#** and **Course# EmailId**. Course# is overlapping among those candidate keys. Hence these candidate keys are called as "overlapping candidate keys" which is shown in Figure 3-12.

The non-key attribute, Marks is non-transitively and fully functionally dependant on key attributes. Hence this is in 3NF. But this is not in BCNF because there are four determinants in this relation namely:

- **Student#** (Student# decides EmailID)
- **EmailID** (EmailID decides Student#)
- **Student# Course#** (decides rest of the attributes in RESULT table)
- **Course# EmailID** (decides rest of the attributes in RESULT table)

All above determinants are not candidate keys. EmailID decides Student# but EmailID on its own is not a candidate key. Similarly Student# decides EmailID of a student but Student# alone is not a candidate key. Only combination of Student# Course# and Course# EmailID are candidate keys.

To make this table BCNF, we need to split this table into the following structure:

**STUDENT TABLE**

<table>
<thead>
<tr>
<th>Student#</th>
<th>EmailID</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td><a href="mailto:Davis@myuni.edu">Davis@myuni.edu</a></td>
</tr>
<tr>
<td>102</td>
<td><a href="mailto:Daniel@myuni.edu">Daniel@myuni.edu</a></td>
</tr>
<tr>
<td>103</td>
<td><a href="mailto:Sandra@myuni.edu">Sandra@myuni.edu</a></td>
</tr>
<tr>
<td>104</td>
<td><a href="mailto:Evelyn@myuni.edu">Evelyn@myuni.edu</a></td>
</tr>
<tr>
<td>105</td>
<td><a href="mailto:Susan@myuni.edu">Susan@myuni.edu</a></td>
</tr>
</tbody>
</table>

**RESULT TABLE**

<table>
<thead>
<tr>
<th>Student#</th>
<th>Course#</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>M4</td>
<td>82</td>
</tr>
<tr>
<td>102</td>
<td>M4</td>
<td>62</td>
</tr>
<tr>
<td>101</td>
<td>H6</td>
<td>79</td>
</tr>
<tr>
<td>103</td>
<td>C3</td>
<td>65</td>
</tr>
<tr>
<td>104</td>
<td>B3</td>
<td>77</td>
</tr>
<tr>
<td>102</td>
<td>P3</td>
<td>68</td>
</tr>
<tr>
<td>105</td>
<td>P3</td>
<td>89</td>
</tr>
<tr>
<td>103</td>
<td>B4</td>
<td>54</td>
</tr>
<tr>
<td>105</td>
<td>H6</td>
<td>87</td>
</tr>
<tr>
<td>104</td>
<td>M4</td>
<td>65</td>
</tr>
</tbody>
</table>

Figure 3-13: Boyce Codd Normal Form

Now both the tables are not only in 3NF, but also in BCNF because all the determinants are candidate keys. In the first table, Student# decides EmailID and EmailID decides Student# and both are candidate keys.

In second table, Student# Course# is only determinant and candidate key. Hence it qualifies BCNF definition that every determinant must be a candidate key.

**Note:** If the table has only one non-composite candidate key and if it is in 3NF, then the table will also be in BCNF.

Basically 2NF and 3NF takes away the redundancy, anomalies which exist among the key and non-key attributes on other hand BCNF takes away the redundancy, anomalies
which exist among the key attributes. At Infosys, we rarely (around 1% of database design) normalize the databases to BCNF.

3.5. Merits and Demerits of Normalization

The following sections discuss merits and demerits of normalization.

3.5.1. Merits

1) Normalization is based on mathematical foundation.
2) Removes the redundancy to the greater extent. After 3NF, data redundancy is minimized to the extent of foreign keys.
3) Removes the anomalies present in Inserts, Updates and Deletes.

3.5.2. Demerits

1) Data retrieval (Select) operation performance will be severely affected.

Example: Let us assume that the university management wants to have the report of students performance in the following format.

<table>
<thead>
<tr>
<th>Student Name</th>
<th>Course Name</th>
<th>Date Of Exam</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daniel</td>
<td>Applied Mathematics</td>
<td>11-Nov-04</td>
<td>C</td>
</tr>
<tr>
<td>Daniel</td>
<td>Nuclear Physics</td>
<td>12-Nov-04</td>
<td>B</td>
</tr>
<tr>
<td>Davis</td>
<td>Applied Mathematics</td>
<td>11-Nov-04</td>
<td>A</td>
</tr>
<tr>
<td>Davis</td>
<td>American History</td>
<td>22-Nov-04</td>
<td>B</td>
</tr>
<tr>
<td>Evelyn</td>
<td>Botany</td>
<td>26-Nov-04</td>
<td>B</td>
</tr>
<tr>
<td>Evelyn</td>
<td>Applied Mathematics</td>
<td>11-Nov-04</td>
<td>B</td>
</tr>
<tr>
<td>Sandra</td>
<td>Bio Chemistry</td>
<td>16-Nov-04</td>
<td>B</td>
</tr>
<tr>
<td>Sandra</td>
<td>Zoology</td>
<td>27-Nov-04</td>
<td>D</td>
</tr>
<tr>
<td>Susan</td>
<td>Nuclear Physics</td>
<td>12-Nov-04</td>
<td>A</td>
</tr>
<tr>
<td>Susan</td>
<td>American History</td>
<td>22-Nov-04</td>
<td>A</td>
</tr>
</tbody>
</table>

Figure 3-14: Proposed University Report

After applying 3NF normalization technique for database design, a single table will not contain all the information as desired by the college management.

We need to select Student Name from STUDENT table, Course Name from COURSE table, Date of Examination from EXAM_DATE table and Grade from MarksGrade table. In an un-normalized format we would have retrieved all these columns just from one table.

Hence normalization will definitely slow down the Select operations. It is better to restrict normalization process to 2NF, if application has more data retrieval operations than insert or update or delete operations.
If the application is used for querying a database, it is called as “Reporting System”. Let us take the example of a Railway enquiry system. This enquiry system is used to enquire about reservation availability and not used to book the tickets. On the other hand a Railway reservation system is called as “On-line application” because this system is used for booking tickets (inserts), changing travel plans (updates) and canceling tickets (deletes).

Hence one may normalize only up to 2NF for Reporting System and 3NF for Online applications.

2) Normalization may not always represent real world scenarios. It should be borne in mind however that full normalization may not always be desirable and the database designer may take advantage of his/her intimate knowledge of the real world and choose not to normalize in some particular instance.

Example: consider the following relation: CUSTOMER (Name, Street, City, Postcode. Strictly speaking, the attribute Postcode uniquely identifies City, hence transitive dependency exists in the above scenario.

Postcode -> City

Thus CUSTOMER table is not in 3NF. However in practice the attributes City and Postcode are always used together as a unit and decomposing the relation would not be advisable in this case.

Note: Some time to increase the performance of select operations for reporting application, database design is taken back from higher normal form to lower normal form (ex: 3NF to 2NF). This process is called as de-normalization or Second Level Design (SLD).

3.6. Summary

- Normalization is a refinement process. It helps in removing anomalies present in insert, update and delete operations.
- Normalization is also called “Bottom-up approach”, because this technique requires minute details like every participating attribute and how it is dependant on the key attributes is crucial. If you add new attributes after normalization, it may change the normal form itself.
- There are four normal forms that were defined being commonly used.
- 1NF makes sure that all the attributes are atomic in nature.
- 2NF removes the partial dependency.
- 3NF removes the transitive dependency.
- BCNF removes dependency among key attributes.
- Excessive normalization adversely affects select or retrieval operations.
- It is always better to normalize to 3NF for insert, update and delete intensive (online transaction) systems.
- It is always better to restrict to 2NF for select intensive (reporting) systems. While normalizing, use common sense and don’t use the normal forms as absolute measures.

### Points to Remember:

<table>
<thead>
<tr>
<th>Normal Form</th>
<th>Test</th>
<th>Remedy (Normalization)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1NF</td>
<td>Relation should have atomic attributes. The domain of an attribute must include only atomic (simple, indivisible) values.</td>
<td>Form new relations for each non-atomic attribute</td>
</tr>
<tr>
<td>2NF</td>
<td>For relations where primary key contains multiple attributes (composite primary key), non-key attribute should not be functionally dependent on a part of the primary key.</td>
<td>Decompose and form a new relation for each partial key with its dependent attribute(s). Retain the relation with the original primary key and any attributes that are fully functionally dependent on it.</td>
</tr>
<tr>
<td>3NF</td>
<td>Relation should not have a non-key attribute functionally determined by another non-key attribute (or by a set of non-key attributes). In other words there should be no transitive dependency of a non-key attribute on the primary key.</td>
<td>Decompose and form a relation that includes the non-key attribute(s) that functionally determine(s) other non-key attribute(s).</td>
</tr>
</tbody>
</table>
3.7. Case study

Given below is the data in an un-normalized table. Normalize it to 1F. Identify the problems encountered when the table is in 1NF but not in 2NF. Subsequently normalize to 2NF and 3NF, explaining the problems faced and the solution to it.

<table>
<thead>
<tr>
<th>Proj_No</th>
<th>Proj_Name</th>
<th>Emp_No</th>
<th>Emp_Name</th>
<th>Rate_Category</th>
<th>Hourly_Rate_in_dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>1023</td>
<td>Amsterdam travel site</td>
<td>101</td>
<td>Vincent R</td>
<td>A</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>102</td>
<td>Pauline J</td>
<td>B</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>103</td>
<td>Charles C</td>
<td>C</td>
<td>40</td>
</tr>
<tr>
<td>1056</td>
<td>Real Estate Agency</td>
<td>101</td>
<td>Vincent R</td>
<td>A</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>107</td>
<td>David R</td>
<td>B</td>
<td>50</td>
</tr>
</tbody>
</table>

Solution:

Table (1NF)

<table>
<thead>
<tr>
<th>Proj_No</th>
<th>Proj_Name</th>
<th>Emp_No</th>
<th>Emp_Name</th>
<th>Rate_Category</th>
<th>Hourly_Rate_in_dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>1023</td>
<td>Amsterdam travel site</td>
<td>101</td>
<td>Vincent R</td>
<td>A</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>102</td>
<td>Pauline J</td>
<td>B</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>103</td>
<td>Charles C</td>
<td>C</td>
<td>40</td>
</tr>
<tr>
<td>1056</td>
<td>Real Estate Agency</td>
<td>101</td>
<td>Vincent R</td>
<td>A</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>107</td>
<td>David R</td>
<td>B</td>
<td>50</td>
</tr>
</tbody>
</table>

Problems encountered when the table is in 1NF but not in 2NF:

(i) Wastage of space: Information that code 1023 refers to the Amsterdam travel site appears three (3) times.

(ii) Update Anomaly: If the project name has to be changed, it has to be done in all the rows that the project name appears in. If it has not been changed in just one row, this may lead to inconsistency problems.

(iii) Insert Anomaly: The information about a new employee cannot be inserted into the table unless the employee is assigned to a project.
(iv) **Delete Anomaly:** If there is only one employee working on a project, it is not possible to delete information about the employee without losing information about the project. In other words it is not possible to delete a subset of a record.

**Solution: Normalize to 2NF**

(a) Take out the duplication  
(b) Look for partial dependencies i.e. fields that are dependent on a part of a key and not on the entire key.

In the above table, the key is (Proj_No, Emp_No)

The functional dependencies are as follows:

- Proj_No → Proj_Name
- Emp_No → Emp_Name, Rate_Category, Hourly_Rate_in_Dollars
- Rate_Category → Hourly_Rate_in_Dollars

The above table should be decomposed as follows:

**Employee_Project Table**

<table>
<thead>
<tr>
<th>Proj_No</th>
<th>Emp_No</th>
</tr>
</thead>
<tbody>
<tr>
<td>1023</td>
<td>101</td>
</tr>
<tr>
<td>1023</td>
<td>102</td>
</tr>
<tr>
<td>1023</td>
<td>103</td>
</tr>
<tr>
<td>1056</td>
<td>101</td>
</tr>
<tr>
<td>1056</td>
<td>107</td>
</tr>
</tbody>
</table>

**Employee Table**

<table>
<thead>
<tr>
<th>Emp_No</th>
<th>Emp_Name</th>
<th>Rate_Category</th>
<th>Hourly_Rate_in_Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Vincent R</td>
<td>A</td>
<td>60</td>
</tr>
<tr>
<td>102</td>
<td>Pauline J</td>
<td>B</td>
<td>50</td>
</tr>
<tr>
<td>103</td>
<td>Charles C</td>
<td>C</td>
<td>40</td>
</tr>
<tr>
<td>107</td>
<td>David R</td>
<td>B</td>
<td>50</td>
</tr>
</tbody>
</table>

**Project Table**

<table>
<thead>
<tr>
<th>Proj_No</th>
<th>Proj_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1023</td>
<td>Amsterdam Travel site</td>
</tr>
<tr>
<td>1056</td>
<td>Real Estate Agency</td>
</tr>
</tbody>
</table>

**Problems faced with the table in 2NF**

(i) Stores data redundantly: The Hourly_Rate_in_Dollars and Rate_Category are being stored in its entirety for each employee.

(ii) Update Anomaly: If the hourly rate in dollars has to be changed for a particular rate category, it has to be done in all the rows that the rate category appears in. If it has not been changed in just one row, this may lead to inconsistency problems.
(iii) Insert Anomaly: It is not possible to insert information about a new rate category and the corresponding hourly rate in dollars unless there is an employee in that rate category.

(iv) Delete Anomaly: If there is only one employee in a particular rate category, it is not possible to delete information about the employee without losing information about that rate category and the corresponding hourly rate in dollars.

**Solution: Normalize to 3NF**

(i) Remove this excess data into its own table.

(ii) Look for transitive relationships or relationships where a non-key attribute is dependent on another non-key attribute.

In the above table (Employee table), Hourly_Rate_in_Dollars is actually dependent on Rate_Category according to the functional dependency Rate_Category → Hourly_Rate_in_Dollars

The above table (Employee) should be decomposed as follows:

**Employee table**

<table>
<thead>
<tr>
<th>Emp_No</th>
<th>Emp_Name</th>
<th>Rate_Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Vincent R</td>
<td>A</td>
</tr>
<tr>
<td>102</td>
<td>Pauline J</td>
<td>B</td>
</tr>
<tr>
<td>103</td>
<td>Charles C</td>
<td>C</td>
</tr>
<tr>
<td>107</td>
<td>David R</td>
<td>B</td>
</tr>
</tbody>
</table>

**Rate Table**

<table>
<thead>
<tr>
<th>Rate_Category</th>
<th>Hourly_Rate_in_Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>60</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
</tr>
<tr>
<td>C</td>
<td>40</td>
</tr>
</tbody>
</table>
4. **Structured Query Language (SQL)**

SQL is used to interact with a database to manage and retrieve data.

4.1. **The Purpose of SQL**

SQL is used to retrieve data from the database. The DBMS processes the SQL request, retrieves the requested data from the database, and returns it. This process of requesting data from the database and receiving back the results is called a database query and hence the name Structured Query Language. Refer to Figure 4-1.

![Figure 4-1: Using SQL for database access](image)

SQL is used to control all the functions that a DBMS provides for its users, including:

- **Data Definition**: SQL allows a user to define the structure and the organization of the data to be stored and the relationships among the stored data items.
- **Data Retrieval**: SQL allows a user or an application program to retrieve the stored data from the database.
- **Data Manipulation**: SQL lets a user or an application program update the database by allowing to add new data, delete the existing data, and modify the existing data.
- **Access Control**: SQL can be used to restrict a user’s ability to retrieve, add, and modify data, thus protecting the stored data against unauthorized access.
4.2. A Brief History of SQL

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970</td>
<td>The relational model devised by Codd was explored during the 1970s, and commercial relational database products began to emerge in the 1980s, originally for mainframe systems and later for microcomputers. Edgar Codd first wrote about the concept of relational databases in his paper ‘A relational model of data for large shared data banks’ in 1970.</td>
</tr>
<tr>
<td>1979</td>
<td>Oracle Corporation introduced the first commercial RDBMS</td>
</tr>
<tr>
<td>1982</td>
<td>ANSI (American National Standards Institute) formed SQL Standards Committee</td>
</tr>
<tr>
<td>1983</td>
<td>IBM (International Business Machine) announced DB2 (a database)</td>
</tr>
<tr>
<td>1986</td>
<td>ANSI (American National Standards Institute) SQL1 standard is approved</td>
</tr>
<tr>
<td>1987</td>
<td>ISO (International Organization for Standardization) SQL1 standard is approved</td>
</tr>
<tr>
<td>1992</td>
<td>ANSI (American National Standards Institute) SQL2 standard is approved</td>
</tr>
<tr>
<td>2000</td>
<td>Microsoft Corporation introduces SQL Server 2000, aimed at enterprise applications</td>
</tr>
<tr>
<td>2002</td>
<td>Research firm Gartner ranked IBM as #1 database vendor over Oracle</td>
</tr>
<tr>
<td>2004</td>
<td>SQL: 2003 standard is published</td>
</tr>
</tbody>
</table>
### 4.3. Data Types

The data types are used to specify the type of data that will be stored in each column of the table. The following table lists the typical *data types* used in Oracle 8i and Oracle 9i:

<table>
<thead>
<tr>
<th>Data Syntax</th>
<th>Type</th>
<th>Oracle 8i</th>
<th>Oracle 9i</th>
<th>Explanation (if applicable)</th>
</tr>
</thead>
</table>
| dec(p, s)   |      | The maximum precision is 38 digits. | The maximum precision is 38 digits. | Where *p* is the precision and *s* is the scale.  
*Example*: dec(3, 1) is a number that has 2 digits before the decimal and 1 digit after the decimal. |
| decimal(p, s) | | | | |
| numeric(p, s) |      | The maximum precision is 38 digits. | The maximum precision is 38 digits. | Where *p* is the precision and *s* is the scale.  
*Example*: numeric(7, 2) is a number that has 5 digits before the decimal and 2 digits after the decimal. |
| number(p, s) | | | | |
| char (size) | | Up to 2000 bytes in Oracle 8i. | Up to 2000 bytes in Oracle 9i. | Where *size* is the number of characters to store. Fixed-length strings. Space padded.  
*Example*: if the width of a character variable is 10 and the string stored in it is ‘RDBMS’, it will be stored as ‘RDBMS’. |
<p>| varchar2 (size) | | Up to 4000 bytes in Oracle 8i. | Up to 4000 bytes in Oracle 9i. | Where <em>size</em> is the number of characters to store. Variable-length strings. |</p>
<table>
<thead>
<tr>
<th>Data Types</th>
<th>Description</th>
<th>Example: if the width of a character variable is 10 and the string stored in it is 'RDBMS', it will be stored as 'RDBMS'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long</td>
<td>Up to 2 gigabytes.</td>
<td>Variable-length strings. (backward compatible)</td>
</tr>
<tr>
<td>Date</td>
<td>A date between Jan 1, 4712 BC and Dec 31, 9999 AD.</td>
<td>'25-JAN-2005'</td>
</tr>
</tbody>
</table>

---

38 **Data Types**: The description of the kinds of data stored, passed and used.
39 **Backward Compatible**: A design that continues to work with earlier versions of a language, program, etc.
4.4. Statement types

The following table lists the three types of SQL statements:

<table>
<thead>
<tr>
<th>Type of SQL statement</th>
<th>SQL keywords</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Definition Language (DDL)</td>
<td>CREATE, ALTER, DROP</td>
<td>Used to define, change and drop the</td>
</tr>
<tr>
<td></td>
<td>TRUNCATE</td>
<td>structure of a table.</td>
</tr>
<tr>
<td>Data Manipulation Language(DML)</td>
<td>INSERT INTO, UPDATE</td>
<td>Used to enter, modify, delete and</td>
</tr>
<tr>
<td></td>
<td>DELETE FROM, SELECT</td>
<td>retrieve data from a table.</td>
</tr>
<tr>
<td>Data Control Language (DCL)</td>
<td>GRANT, REVOKE</td>
<td>Used to control access to the data in</td>
</tr>
<tr>
<td></td>
<td>COMMIT, ROLLBACK</td>
<td>a database.</td>
</tr>
</tbody>
</table>

Note: All keywords must be entered as described otherwise users get syntax errors.

4.5. Data Definition Language (DDL) Statements

DDL statements are used to define and manage table(s).

The DDL statements are used to:
- Define and create a new table
- Remove a table that is no longer needed
- Change the definition of an existing table
- Define a virtual table (view) of data (Covered in section 4.7)
- Build an index\(^{40}\) to access a table faster (Covered in section 4.5.5)

\(^{40}\) Index: Indices are created in an existing table to locate rows more quickly and efficiently. It is possible to create an index on one or more columns of a table, and each index is given a name. The users cannot see the indexes; they are just used to speed up queries. More on index is covered in Section 4.5.5
CONSTRAINTS
Data types limit the kind of data that can be stored in a table. This is however, not enough. For example, a column to store a product price should accept only positive values. But there is no data type that accepts only positive numbers. Another requirement could be to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should only be one row for each product number.
SQL allows the definition of constraints on columns and tables. If a user attempts to store data in a column that would violate a constraint, an error is raised.

Types of Constraints:
- **Column Constraint**: A column constraint is specified as part of a column definition and applies only to that column
- **Table Constraint**: A table constraint is declared independently from a column definition and can apply to more than one column in a table

**Note**: Table constraints must be used when constraint is applied for more than one column of a table.

4.5.1. CREATE TABLE Statement
The CREATE TABLE statement can:
- Create a table
- Define column constraints
- Define table constraints
Refer to Figure 4-2.
CREATE TABLE table-name

Column-Definition:

| column-name | data-type [ DEFAULT value ] |

Table-Constraint-Definition:

CONRAINT constraint-name

- primary-key-constraint
- foreign-key-constraint
- uniqueness-constraint
- check-constraint

Primary-Key-Constraint:

PRIMARY KEY ( column-name )

Foreign-Key-Constraint:

FOREIGN KEY ( column-name ) REFERENCES table-name [ column-name ]

Uniqueness-Constraint:

UNIQUE ( column-name )

Check-Constraint:

CHECK ( search-condition )

Figure 4-2: CREATE TABLE syntax diagram

Note: Anything enclosed between [ ] is optional.
Example:

1. Create a table Customer_Details with the following specifications

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type and Width</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cust_ID</td>
<td>Number(5)</td>
<td>Not Null</td>
</tr>
<tr>
<td>Cust_Last_Name</td>
<td>Varchar2(20)</td>
<td>Not Null</td>
</tr>
<tr>
<td>Cust_Mid_Name</td>
<td>Varchar2(4)</td>
<td></td>
</tr>
<tr>
<td>Cust_First_Name</td>
<td>Varchar2(20)</td>
<td></td>
</tr>
<tr>
<td>Account_No</td>
<td>Number(5)</td>
<td>Primary Key</td>
</tr>
<tr>
<td>Account_Type</td>
<td>Varchar2(10)</td>
<td>Not Null</td>
</tr>
<tr>
<td>Bank_Branch</td>
<td>Varchar2(25)</td>
<td>Not Null</td>
</tr>
<tr>
<td>Cust_Email</td>
<td>Varchar2(30)</td>
<td></td>
</tr>
</tbody>
</table>

Syntax:

CREATE TABLE Customer_Details(
  Cust_ID Number(5) CONSTRAINT Nnull1 NOT NULL,
  Cust_Last_Name VarChar2(20) CONSTRAINT Nnull2 NOT NULL,
  Cust_Mid_Name VarChar2(4),
  Cust_First_Name VarChar2(20),
  Account_No Number(5) CONSTRAINT Pkey1 PRIMARY KEY,
  Account_Type VarChar2(10) CONSTRAINT Nnull3 NOT NULL,
  Bank_Branch VarChar2(25) CONSTRAINT Nnull4 NOT NULL,
  Cust_Email VarChar2(30)
);

2. Create a table Employee_Manager with the following specifications

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type and Width</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee_ID</td>
<td>Number(6)</td>
<td>Primary Key</td>
</tr>
<tr>
<td>Employee_Last_Name</td>
<td>Varchar2(25)</td>
<td></td>
</tr>
<tr>
<td>Employee_Mid_Name</td>
<td>Varchar2(5)</td>
<td></td>
</tr>
<tr>
<td>Employee_First_Name</td>
<td>Varchar2(25)</td>
<td></td>
</tr>
<tr>
<td>Employee_Email</td>
<td>Varchar2(45)</td>
<td></td>
</tr>
<tr>
<td>Department</td>
<td>Varchar2(10)</td>
<td></td>
</tr>
<tr>
<td>Grade</td>
<td>Number(2)</td>
<td></td>
</tr>
<tr>
<td>Manager_ID</td>
<td>Number(6)</td>
<td>Foreign Key Referencing Employee_ID</td>
</tr>
</tbody>
</table>
Syntax:

CREATE TABLE Employee_Manager(
    Employee_ID Number(6) CONSTRAINT Pkey2 PRIMARY KEY,
    Employee_Last_Name VarChar2(25),
    Employee_Mid_Name VarChar2(5),
    Employee_First_Name VarChar2(25),
    Employee_Email VarChar2(45),
    Department VarChar2(10),
    Grade Number(2),
    Manager_ID Number(6) CONSTRAINT Fkey2
       REFERENCES Employee_Manager(Employee_ID)
);

A column level constraint follows a column definition whereas a table level constraint follows a table definition. A table level constraint generally involves two or more columns.

3. A primary key as a column constraint

Syntax:

CREATE TABLE Customer_Details(
    Cust_ID Number(5) CONSTRAINT Nnull1 NOT NULL,
    Cust_Last_Name VarChar2(20) CONSTRAINT Nnull2 NOT NULL,
    Cust_Mid_Name VarChar2(4),
    Cust_First_Name VarChar2(20),
    Account_No  Number(5) CONSTRAINT Pkey1 PRIMARY KEY,
    Account_Type VarChar2(10) CONSTRAINT Nnull3 NOT NULL,
    Bank_Branch VarChar2(25) CONSTRAINT Nnull4 NOT NULL,
    Cust_Email VarChar2(30)
);

The primary key definition in the above example follows the column (Account_No) definition. A column definition includes the name of the column, data type and length of the column.

4. A primary key as a table constraint

Syntax:

CREATE TABLE Customer_Details(
    Cust_ID Number(5) CONSTRAINT Nnull1 NOT NULL,
    Cust_Last_Name VarChar2(20) CONSTRAINT Nnull2 NOT NULL,
    Cust_Mid_Name VarChar2(4),
    Cust_First_Name VarChar2(20),
    Account_No  Number(5),
    Account_Type VarChar2(10) CONSTRAINT Nnull3 NOT NULL,
    Bank_Branch VarChar2(25) CONSTRAINT Nnull4 NOT NULL,
    Cust_Email VarChar2(30), CONSTRAINT pkey1 PRIMARY KEY(Cust_ID,
    Account_No)
);
The primary key definition in the above example follows the table definition i.e. the primary key definition occurs after all the columns in the table have been defined for their data type and width.

5. **To create a table from another table**

**Syntax:**

```
CREATE TABLE Cust_Details AS
    SELECT Cust_ID, Account_No, Account_Type, Bank_Branch, Cust_Email
    FROM Customer_Details;
```

In the example above Cust_Details table is created from Customer_Details table. Cust_details table is created with attributes Cust_ID, Account_No, Account_Type, Bank_Branch and Cust_Email. If it is required that Cust_Details table has exactly the same structure as Customer_Details table the syntax would be as follows:

```
CREATE TABLE Cust_Details as
    SELECT *
    FROM Customer_Details;
```

In the examples above not only is the structure copied but the data is also copied.

To copy only the structure and not the data

```
CREATE TABLE Cust_Details as
    SELECT *
    FROM Customer_Details
    WHERE 1=2;
```

**Note:** When a table is created from another table, only the NOT NULL constraints are copied. All the other constraints are not copied.

6. **Domain integrity constraint (check constraint – column constraint)**

```
CREATE TABLE Customer_Details(
    Cust_ID Number(5) CONSTRAINT Nnull1 NOT NULL
        CONSTRAINT Ccheck1 CHECK( Cust_ID BETWEEN 101 AND 105),
    Cust_Last_Name VarChar2(20) CONSTRAINT Nnull2 NOT NULL,
    Cust_Mid_Name VarChar2(4),
    Cust_First_Name VarChar2(20),
    Account_No  Number(5) CONSTRAINT Pkey1 PRIMARY KEY,
    Account_Type VarChar2(10) CONSTRAINT Nnull3 NOT NULL,
    Bank_Branch VarChar2(25) CONSTRAINT Nnull4 NOT NULL,
    Cust_Email VarChar2(30)
);
```
7. Domain integrity constraint (check constraint – table constraint)

CREATE TABLE Customer_Details(
  Cust_ID Number(5) CONSTRAINT Nnull1 NOT NULL,
  Cust_Last_Name VarChar2(20) CONSTRAINT Nnull2 NOT NULL,
  Cust_Mid_Name VarChar2(4),
  Cust_First_Name VarChar2(20),
  Account_No Number(5) CONSTRAINT Pkey1 PRIMARY KEY,
  Account_Type VarChar2(10) CONSTRAINT Nnull3 NOT NULL,
  Bank_Branch VarChar2(25) CONSTRAINT Nnull4 NOT NULL,
  Cust_Email VarChar2(30),
  CONSTRAINT Ccheck2 CHECK(Cust_ID BETWEEN 101 AND 105 AND
  ACCOUNT_TYPE in (‘Savings’, ‘Checkings’))
);

Note: Although giving a name to a constraint is optional, it is a good programming practice to give every constraint a name (preferably a meaningful name) which is unique and can’t be applied to any other constraint of any table. The name of the constraint is required when the constraint has to be dropped.

A **NOT NULL** constraint on a column(s) means that it is mandatory to provide a value for the column(s).

A **UNIQUE** constraint on a column(s) means that the values in the column(s) should be distinct although it can have NULL values.

In most of DBMS, a **PRIMARY KEY** constraint implicitly imposes a NOT NULL and UNIQUE constraint. If there is a composite primary key, each of the attribute constituting the primary key is NOT NULL. In other words it will not allow a NULL value in any of the attributes constituting the composite primary key. However the combination of attributes constituting the primary key should offer a unique value.

A **Foreign Key** constraint on a set of attribute(s) does not prevent them from having duplicate or NULL values.

Note: Users can use the DESCRIBE <tablename> or DESC <tablename> statement to see the structure of the table.
Example:

```
DESCRIBE Customer_Details;
Or
DESC Customer_Details;
```

### 4.5.2. ALTER TABLE statement

The ALTER TABLE statement can:
- Add a column definition to a table
- Drop a column from a table
- Add or drop a primary key to / from a table
- Add or drop a foreign key to / from a table
- Add or drop a unique constraint to / from a table
- Add or drop a check constraint to / from a table

![Figure 4-3: ALTER TABLE statement syntax diagram](image)

**Note:** The check constraint enforces the domain integrity constraint. It permits only values allowed by the constraint into the column(s). The domain integrity constraint will be covered in detail in chapter 5.
Example:

1. **Adding a column**

   *Add a phone number to the Customer_Details table*

   ```sql
   ALTER TABLE Customer_Details
   ADD Contact_Phone Char(10);
   ```

2. **Modifying a column definition**

   *Modify the size of the Contact_Phone column*

   ```sql
   ALTER TABLE Customer_Details
   MODIFY Contact_Phone Char(12);
   ```

3. **Adding a NOT NULL Constraint**

   *Add the NOT NULL constraint on the Contact_Phone column*

   ```sql
   ALTER TABLE Customer_Details
   MODIFY Contact_Phone Char(12) CONSTRAINT Nnull5 NOT NULL;
   ```

4. **Adding a UNIQUE Constraint**

   *Add the UNIQUE constraint on the Contact_Phone column*

   ```sql
   ALTER TABLE Customer_Details
   ADD CONSTRAINT Uunique1 UNIQUE (Contact_Phone);
   ```

5. **Dropping a constraint**

   *Drop the NOT NULL constraint on Contact_Phone column*

   ```sql
   ALTER TABLE Customer_Details
   DROP CONSTRAINT Nnull5;
   ```

6. **Dropping a column**

   *Drop the Contact_Phone column from the Customer_Details table*

   ```sql
   ALTER TABLE Customer_Details
   DROP (Contact_Phone);
   ```

7. **Adding a simple PRIMARY KEY**

   *Make the Account_No column as the primary key*

   ```sql
   ALTER TABLE Customer_Details
   ADD CONSTRAINT Pkey1 PRIMARY KEY (Account_No);
   ```
8. **Adding a composite PRIMARY KEY (table level constraint)**

Make the Account_No and Cust_ID columns as the primary key

```
ALTER TABLE Customer_Details
    ADD CONSTRAINT Pkey2 PRIMARY KEY (Account_No, Cust_ID);
```

9. **Adding FOREIGN KEY**

Make Account_No column in Customer_Transaction table as the foreign key referencing Account_No column of Customer_Details

```
ALTER TABLE Customer_Transaction
    ADD CONSTRAINT Fkey1 FOREIGN KEY (Account_No)
    REFERENCES Customer_Details (Account_No);
```

10. **Adding a CHECK constraint**

```
ALTER TABLE Customer_Details
    ADD CONSTRAINT Ccheck1 CHECK (Cust_ID BETWEEN 101 AND 105);
```

11. **Dropping a simple or composite PRIMARY KEY constraint**

Drop the primary key constraint

```
ALTER TABLE Customer_Details
    DROP PRIMARY KEY;
```

Or

```
ALTER TABLE Customer_Details
    DROP CONSTRAINT Pkey1;
```

**Note:** The syntax is the same irrespective of whether the primary key is simple or composite.

A table can have only one primary key. A table can have one or more foreign keys. If a table already has a primary key, adding a primary key using the ALTER TABLE statement results in error. The RDBMS will not allow a PRIMARY KEY constraint (using the ALTER TABLE statement) on column(s) if the column(s) has NULL or duplicate values.

**Note:** The ALTER TABLE statement cannot be used to change the name of a column or a table. It can be used to change the data type or length of the column. Columns to be modified should be empty to decrease column length. Columns to be modified should be empty to change the data type.

If the table has only one column, the ALTER TABLE statement
4.5.3. DROP TABLE statement

The DROP TABLE statement is used to drop a table from the database.

```
DROP TABLE table-name
```

Figure 4-4: DROP TABLE statement syntax diagram

When the DROP TABLE statement removes a table from the database, its schema/structure and all of its contents are lost. There is no way to recover the data.

**Note:** Most RDBMS will restrict the dropping of a table if it has attribute(s) being referred to by attribute(s) of another table. This is called the referential integrity constraint.

Example:

```
Discard Customer_Details table
DROP TABLE Customer_Details;
```

4.5.4. TRUNCATE TABLE statement

The TRUNCATE TABLE statement is used to remove/delete all rows from a table.

```
TRUNCATE TABLE table-name
```

Figure 4-5: TRUNCATE TABLE statement syntax diagram

When the TRUNCATE TABLE statement is used, all the contents of the specified table are lost but its definition remains intact. There is no way to recover the data. It releases the memory occupied by the contents of the specified table.

Example:

```
Delete all rows from the Customer_Details table
TRUNCATE TABLE Customer_Details;
```
4.5.5. CREATE INDEX statement

An index is a structure that provides rapid access to the rows of a table based on the values of one or more columns. The index stores the data values and pointers to the rows where those data values occur. In the index, the data values are arranged either in ascending or in descending order, so that the RDBMS can quickly lookup the index to find a particular value. It then follows the pointer to locate the row containing the value.

In Figure 4-6, the index is created on the Account_No which in turn points to the corresponding rows in the table.

Note: The SQL user, who accesses a table, is unaware of the presence or absence of the index on the table.

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Account_No</th>
<th>Account_Type</th>
<th>Bank_Branch</th>
<th>Cust_Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A</td>
<td>Mike</td>
<td>102</td>
<td>Savings</td>
<td>Downtown</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
</tr>
<tr>
<td>102</td>
<td>Smith</td>
<td>S</td>
<td>Graham</td>
<td>2348</td>
<td>Checking</td>
<td>Bridgewater</td>
<td><a href="mailto:Smith_Graham@rediffmail.com">Smith_Graham@rediffmail.com</a></td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>S</td>
<td>Justin</td>
<td>3421</td>
<td>Savings</td>
<td>Plainsboro</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
</tr>
<tr>
<td>104</td>
<td>Quails</td>
<td>D</td>
<td>Jack</td>
<td>2387</td>
<td>Checking</td>
<td>Downtown</td>
<td><a href="mailto:Quails_Jack@yahoo.com">Quails_Jack@yahoo.com</a></td>
</tr>
<tr>
<td>105</td>
<td>Jones</td>
<td>E</td>
<td>Simon</td>
<td>2389</td>
<td>Checking</td>
<td>Brighton</td>
<td><a href="mailto:Jones_Simon@rediffmail.com">Jones_Simon@rediffmail.com</a></td>
</tr>
</tbody>
</table>

Customer_Detail records from Customer_Details file

Figure 4-6: An index on the Customer_Details table on column Account_No

Advantages of having an INDEX:
- It speeds up the execution of SQL statements with search conditions that refer to the indexed column(s)
- It is most appropriate when retrieval of data from tables is more frequent than inserts and updates

Disadvantages of having an INDEX:
- It consumes additional disk space
- The INDEX must be updated every time a row is added to the table and every time the indexed column is updated in an existing row. This imposes additional overhead on INSERT and UPDATE statements for the table
Note:

- Most RDBMS products automatically create an index on the primary key of a table because they anticipate that most frequently access to the table is via the primary key.

- Most RDBMS products also automatically create an index on any column (or column combination) defined with a unique constraint. The RDBMS must check the value of such a column whenever a new row is inserted, or an existing row is updated, to make certain that the value does not duplicate a value already contained in the table. Without the index on the column(s), the RDBMS would have to sequentially search through every row of the table to check the constraint. With an index, the RDBMS can simply use the index to find a row (if it exists) with the value in question, which is a much faster operation than a sequential search.

- When the primary key of the table or the unique constraint on column(s) is dropped, the index which was built on them is also dropped automatically.

CREATE [UNIQUE] INDEX index-name on table-name (column-name)

Figure 4-7: CREATE INDEX statement syntax diagram

DROP INDEX index-name

Figure 4-8: DROP INDEX statement syntax diagram
Example:

1. Create a simple index for the Customer_Details table on Cust_ID
   
   ```sql
   CREATE UNIQUE INDEX Cust_Idx
   ON Customer_Details (Cust_ID);
   ```

2. Create a composite index for the Customer_Details table on Cust_ID and Account_No
   
   ```sql
   CREATE UNIQUE INDEX ID_AccountNo_Idx
   ON Customer_Details (Cust_ID, Account_No);
   ```

3. Drop the index created earlier
   
   ```sql
   DROP INDEX ID_AccountNo_Idx;
   ```

Note: The keyword UNIQUE in the CREATE INDEX statement is optional. If the keyword UNIQUE is omitted, the index table may have duplicates entries.

Points to Remember:

- The CREATE TABLE statement creates a table and defines its columns, PRIMARY KEY, FOREIGN KEY(s) and other constraints like UNIQUE and NOT NULL
- The DROP TABLE statement removes a previously created table from the database
- The ALTER TABLE statement can be used to add a column to an existing table, modify a column definition, add/drop a PRIMARY KEY, FOREIGN KEY and other constraints like UNIQUE and NOT NULL
- The CREATE INDEX statement can be used to define indexes, which speeds up database queries but add overheads to database updates

4.6. Data Manipulation Language Statements

The DML statements are used to:
- Insert data into the table
- Delete data from the table
- Retrieve data from the table
- Modify/update data in the table
4.6.1. **INSERT Statement**

Single-row insert: A single-row INSERT statement adds a single new row of data to the table. Refer to Figure 4-10.

The Single-Row INSERT statement

```
INSERT INTO table-name [ column-name(s) ] VALUES ( constant(s) )
```

**Figure 4-9: Single-row insert statement syntax diagram**

```
INSERT INTO Customer_Details
    ( Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name, Account_No, Account_Type, Bank_Branch, Cust_Email )
VALUES ( 106, 'Costner', 'A.', 'Kevin', 3350, 'Savings', 'Brighton', 'Costner_Kevin@times.com' );
```

**Figure 4-10: Inserting a single row**

**NOTE:** The purpose of the column list in the INSERT statement is to match the data values in the VALUES clause with the columns. The list of values and the column list must both contain the same number of items, and the data type of each value must be compatible with the data type of the corresponding column, or an error will occur.

Data of type Char, Varchar2 and Date are always enclosed within single quotes.

**Example:** 'Costner', '12-Jan-2005'.

Users can use the SELECT * from <tablename> to view the records inserted into the specified table. The SELECT statement is covered in detail in Section 4.6.4.
Example of Invalid INSERT statements:

1. INSERT INTO Customer_Details
   (Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name, 
    Account_No, Account_Type, Bank_Branch)
   VALUES (106, `Costner', 'A.', 'Kevin', 3350, `Savings', 
           `Brighton', `Costner_Kevin@times.com');

   The above INSERT statement is invalid because the number of values in the VALUES clause exceeds the number of columns that are to receive them.

2. INSERT INTO Customer_Details
   (Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name, 
    Account_No, Account_Type, Bank_Branch, Cust_Email)
   VALUES (106, 'Costner', 'A.', 'Kevin', 3350, 'Savings', 
           'Brighton');

   The above INSERT statement is invalid because the number of values in the VALUES clause is less than the columns that are to receive them.

3. Assume Account_No is the Primary Key for the Customer_Details table

   INSERT INTO Customer_Details
   (Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name, 
    Account_Type, Bank_Branch)
   VALUES (106, 'Costner', 'A.', 'Kevin', 'Savings', 'Brighton');

   The above INSERT statement is invalid because the column list specified does not include the attribute, Account_No which happens to be the primary key for the table. Because it has not been included in the column list, SQL automatically assigns a NULL value to it. Being a primary key attribute it cannot have a NULL value.

Inserting NULL values

SQL supports missing, unknown, or inapplicable data explicitly, through the concept of a NULL value. A NULL value is an indicator that tells SQL (and the user) that the data is missing or not applicable. But the NULL value is not a real data value like 0, 473.83 or 'John Clark'. The NULL value occupies space.

Refer to Figure 4-11.
When SQL inserts a new row of data to a table, it automatically assigns a NULL value to any column whose name is missing from the column list in the INSERT statement.

**Example:**

```
INSERT INTO Customer_Details
(Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name, Account_No, Account_Type, Bank_Branch, Cust_Email)
VALUES (106, 'Costner', 'A.', 'Kevin', 3350, 'Savings', 'Brighton', NULL);
```

The assignment of NULL value can be made more explicit by including these columns in the column list and specifying the keyword NULL in the values list.

**Example:**

```
INSERT INTO Customer_Details
(Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name, Account_No, Account_Type, Bank_Branch, Cust_Email)
VALUES (106, 'Costner', 'A.', 'Kevin', 3350, 'Savings', 'Brighton', NULL);
```

**Inserting all columns:**

SQL permits omitting of the column list from the INSERT statement. When the column list is omitted, SQL automatically generates a column list consisting of all columns of the table, in left to right sequence.
Example:

```
INSERT INTO Customer_Details
VALUES (106, 'Costner', 'A.', 'Kevin', 3350, 'Savings', 'Brighton', NULL);
```

<table>
<thead>
<tr>
<th></th>
<th>Costner</th>
<th>A.</th>
<th>Kevin</th>
<th>3350</th>
<th>Savings</th>
<th>Brighton</th>
<th>NULL</th>
</tr>
</thead>
</table>

Note: When the column list is omitted, the NULL keyword must be used in the values list to explicitly assign NULL values to columns. In addition, the sequence of data values must correspond exactly to the sequence of columns in the table.

Example:

To Insert a row into the Customer_Transaction Table

```
INSERT INTO Customer_Transaction
VALUES (2367, '17-JAN-2005', 'Deposit', 2000.00, 14456);
```

Note: The Date value should be input in the format ‘dd-mmm-yyyy’ or ‘dd-mmm-yy’.

Example of Invalid INSERT into the Customer_Details Table

```
INSERT INTO Customer_Details
VALUES (106, 'Costner');
```

In the above INSERT statement, the column list is omitted. The value for all columns should have been provided but the value for only Cust_ID and Cust_Last_Name is provided.

4.6.2. DELETE Statement

The DELETE statement can delete one or more rows from a table. Refer to Figure 4-12.
**Note:** Even if all the data is deleted from the table, the definition of the table and its column is still stored in the database. The table still exists. To erase the definition of the table from the database, the DROP TABLE statement must be used.

The DELETE statement cannot delete column(s) from a table. It deletes only row(s). To delete a column from a table, the ALTER TABLE statement must be used.

```
DELETE FROM table-name [ where search-condition ]
```

**Example:**

*Deleting all rows of a table - Delete all current customers*

1. `DELETE`  
   `FROM Customer_Details;`

*Deleting some rows of a table - Delete Customer with Cust_ID=102 from the list of customers*

2. `DELETE`  
   `FROM Customer_Details`  
   `WHERE Cust_ID = 102;`

**Examples of invalid DELETE Statements**

3. `DELETE *`  
   `FROM Customer_Details;`

   OR

4. `DELETE Cust_ID`  
   `FROM Customer_Details;`

**Difference between TRUNCATE and DELETE statement**

- TRUNCATE is a DDL statement whereas DELETE is a DML statement
- TRUNCATE deletes all records from the table whereas DELETE can be used to selectively delete records from a table using the WHERE clause
- TRUNCATE releases the memory occupied by the records of the table whereas DELETE does not do so
- Data removed using TRUNCATE cannot be recovered whereas data removed using DELETE can be recovered (using ROLLBACK, a DCL statement which is covered in chapter 5)
### 4.6.3. UPDATE Statement

The UPDATE statement modifies the values of one or more columns in selected rows of a table.

The target table to be updated is named in the statement. The ‘WHERE clause’ selects the rows of the table to be modified. The ‘SET clause’ specifies which columns are to be updated and calculates the new values for them.

```
UPDATE table-name SET column-name1 = expression1, column-name2 = expression2, ----- 
[ WHERE search-condition ]
```

**Example:**

1. **Changing all rows**

   *Until fresh instructions come in, delete Rate_of_Interest values for all customers.*

   ```
   UPDATE Customer_Fixed_Deposit
       SET Rate_of_Interest_in_Percent = NULL;
   ```

2. **Changing some rows**

   *For customers with a fixed deposit > 3000, increase Rate_of_Interest to 7.3%*

   ```
   UPDATE Customer_Fixed_Deposit
       SET Rate_of_Interest_in_Percent = 7.3
       WHERE Amount_in_Dollars > 3000;
   ```

3. **Changing the value for more than one column.**

   *Change the Email_ID and Rate_of_Interest of Customer (Cust_ID = 104)*

   ```
   UPDATE Customer_Fixed_Deposit
       SET Cust_Email = 'Quails_Jack@rediffmail.com',
           Rate_of_Interest_in_Percent = 7.3
       WHERE Cust_ID = 104;
   ```

### 4.6.4. SELECT Statement

The SELECT statement retrieves data from a database and returns it in the form of query results. Refer to Figure 4-14. The result of a SQL query is always a table of data.

Refer to Figure 4-15.
4.6.4.1. Simple SELECT Statement

The `SELECT` statement is used to select either some or all the columns from a table.

The asterisk (*) character is a wildcard that is used to denote all columns. It is a good programming practice to avoid the use of `SELECT *`. It is better to list the column names.

**Example:**

1. **Selecting all columns**

   *List all information about all the customers*

   ```sql
   SELECT Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name, Account_No, Account_Type, Bank.Branch, Cust_Email
   FROM Customer_Details;
   ```

   Or
SELECT *  
FROM Customer_Details;

2. Selecting some columns

List Cust_ID, Account_No of all customers

SELECT Cust_ID, Account_No  
FROM Customer_Details;

4.6.4.2. Avoiding duplicates (DISTINCT)

By default the SELECT statement retrieves all rows that are filtered by the SELECT statement. This may however contain duplicates rows. In order to eliminate duplicate rows from the result set returned by the SELECT statement use the keyword DISTINCT. The default keyword is ALL.

Example:

1. List all customers name

SELECT ALL Cust_Last_Name  
FROM Customer_Details;

This is equivalent to:

SELECT Cust_Last_Name  
FROM Customer_Details;

2. This is likely to return duplicate rows. To avoid this:

SELECT DISTINCT Cust_Last_Name  
FROM Customer_Details;

4.6.4.3. Row Selection (WHERE clause)

The WHERE clause is used to specify a search condition that limits the number of rows retrieved. It is a row wise operation.

Refer to Figure 4-16.

For each row, the search condition can produce one of the three results:
- If the search condition is true, the row is included in the query results
- If the search condition is false, the row is excluded from the query results
- If the column being searched has a NULL value, the row is excluded from the query results
**Problem Statement:** To select rows which have 102 in the Manager column.

**Figure 4-16: Row selection with the WHERE clause**

**Example:**

1. **List all customers with an account balance > $10000**
   
   SELECT Account_No, Total_Available_Balance_in_Dollars  
   FROM Customer_Transaction  
   WHERE Total_Available_Balance_in_Dollars > 10000.00;

2. **List the Cust_ID, Account_No of 'Graham'**.
   
   SELECT Cust_ID, Account_No  
   FROM Customer_Details  
   WHERE Cust_First_Name = 'Graham';

   **Note:** The comparison is case sensitive. The column-names are not case-sensitive; the values of the column(s) are case sensitive.

   For Example: ‘GRAHAM’ is not the same as ‘graham’ or ‘Graham’

The **WHERE** clause can be used with any of the comparison operators (\(=\), \(>\), \(<\), \(>=\), \(<=\), \(<>\)) or the logical operators (AND, OR, NOT).

**Figure 4-17: Comparison test syntax diagram**
When SQL compares the values of the two expressions in the comparison test, three results can occur:
1. The test may yield a TRUE result
2. The test may yield a FALSE result
3. If either of the two expressions produces a NULL value, the comparison yields a NULL result.

Example:

1. List all Account_No where Total_Available_Balance_in_Dollars is at least $10000.00

   ```sql
   SELECT Account_No
   FROM Customer_Transaction
   WHERE Total_Available_Balance_in_Dollars >= 10000.00;
   ```

2. List all Cust_ID, Cust_Last_Name where Account_type is 'Savings' and Bank_Branch is 'Downtown'.

   ```sql
   SELECT Cust_ID, Cust_Last_Name
   FROM Customer_Details
   WHERE Account_Type = 'Savings'
       AND Bank_Branch = 'Downtown';
   ```

3. List all Cust_ID, Cust_Last_Name where neither Account_type is 'Savings' and nor Bank_Branch is 'Downtown'.

   ```sql
   SELECT Cust_ID, Cust_Last_Name
   FROM Customer_Details
   WHERE NOT Account_Type = 'Savings'
       AND NOT Bank_Branch = 'Downtown';
   ```

4. List all Cust_ID, Cust_Last_Name where either Account_type is 'Savings' or Bank_Branch is 'Downtown'.

   ```sql
   SELECT Cust_ID, Cust_Last_Name
   FROM Customer_Details
   WHERE Account_Type = 'Savings'
       OR Bank_Branch = 'Downtown';
   ```

The Multi-Row INSERT statement (the discussion of multi-row insert was deferred because it uses a SELECT statement)

A multi-row INSERT statement extracts rows of data from another part of the database and adds them to a table. Refer to Figure 4-18.
In this form of the INSERT statement, the data values for the new rows are not explicitly specified within the statement text. Instead, the source of new rows is a database query, specified in the statement as shown in Figure 4-19.

```
INSERT INTO table-name [column-name(s)] query
```

**Figure 4-18: Multi-row INSERT statement syntax diagram**

**Example:**

```
INSERT INTO OldCust_details
(Account_No, Transaction_Date, Total_Available_Balance_in_Dollars)
SELECT Account_No, Transaction_Date, Total_Available_Balance_in_Dollars
From Customer_Transaction
WHERE Total_Available_Balance_in_Dollars > 10000.00;
```

<table>
<thead>
<tr>
<th>Account_No</th>
<th>Transaction_Date</th>
<th>Transaction_Amount _in_Dollars</th>
<th>Total_Available_Balance _in_Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>1020</td>
<td>12-Jan-2005</td>
<td>Deposit 5000.00</td>
<td>10000.00</td>
</tr>
<tr>
<td>2348</td>
<td>4-Jan-2005</td>
<td>Withdrawal 2500.00</td>
<td>13500.00</td>
</tr>
<tr>
<td>3241</td>
<td>4-Jan-2005</td>
<td>Deposit 2000.00</td>
<td>27234.00</td>
</tr>
<tr>
<td>2387</td>
<td>6-Jan-2005</td>
<td>Withdrawal 1200.00</td>
<td>12456.00</td>
</tr>
<tr>
<td>1020</td>
<td>17-Jan-2005</td>
<td>Withdrawal 1500.00</td>
<td>8500.00</td>
</tr>
</tbody>
</table>

Customer_Transaction records from Customer_Transaction table

```
SELECT Account_No, Transaction_Date, Total_Available_Balance_in_Dollars
FROM Customer_Transaction
WHERE Total_Available_Balance_in_Dollars > 10000
```

Customer_Transaction table

```
SELECT Account_No, Transaction_Date, Total_Available_Balance_in_Dollars
FROM OldCust_details
```

OldCust_details Table

```
Account_No | Transaction_Date | Total_Available_Balance _in_Dollars |
------------|------------------|-------------------------------------|
2348        | 4-Jan-2005       | 13500.00                            |
3241        | 4-Jan-2005       | 27234.00                            |
2387        | 6-Jan-2005       | 12456.00                            |
```

Query Results

```
Account_No | Transaction_Date | Total_Available_Balance _in_Dollars |
------------|------------------|-------------------------------------|
2348        | 4-Jan-2005       | 13500.00                            |
3241        | 4-Jan-2005       | 27234.00                            |
2387        | 6-Jan-2005       | 12456.00                            |
```

**Figure 4-19: Inserting Multiple Rows**

The logical restrictions on the query that appears within the multi-row INSERT statement:

- The query results must contain the same number of columns as the column list in the INSERT statement and the data types must be compatible, column by column.

**4.6.4.4. BETWEEN, IN, LIKE**

The BETWEEN operator includes both the end values specified.
The IN operator is used to check if a value belongs to a set of values.

Note that BETWEEN and IN can be fully substituted with a combination of AND, OR, NOT.

The LIKE operator is used to check for similarity of strings.

When used with LIKE the use of “_” refers to exactly one unknown character; “%” refers to an unknown number of unknown characters.

test-expression [NOT] BETWEEN low-expression AND high-expression

Figure 4-20: Range test (Between) syntax diagram

test-expression [NOT] IN (constant1, constant2............)

Figure 4-21: Set membership test (IN) syntax diagram

Column-name [NOT] LIKE pattern ESCAPE escape-character

Figure 4-22: Pattern matching test (LIKE) syntax diagram
### Example:

1. **List all Account Nos with balance in the range $10000.00 to $20000.00.**

   ```sql
   SELECT Account_No
   FROM Customer_Transaction
   WHERE Total_Available_Balance_in_Dollars BETWEEN 10000.00 AND 20000.00;
   
   Or
   
   SELECT Account_No
   FROM Customer_Transaction
   WHERE Total_Available_Balance_in_Dollars >= 10000.00
   AND Total_Available_Balance_in_Dollars <= 20000.00;
   ```

2. **List all customers who have account in Downtown or Brighton.**

   ```sql
   SELECT Cust_ID
   FROM Customer_Details
   WHERE Bank_Branch IN ('Downtown', 'Brighton');
   
   Or
   
   SELECT Cust_ID
   FROM Customer_Details
   WHERE Bank_Branch = 'Downtown'
   OR Bank_Branch = 'Brighton';
   ```

3. **List all Accounts where the Bank_Branch begins with a ‘D’ and has ‘o’ as the second character.**

   ```sql
   SELECT Cust_ID, Cust_Last_Name, Account_No
   FROM Customer_Details
   WHERE Bank_Branch LIKE 'Do%';
   ```

4. **List all Accounts where the Bank_Branch column has ‘o’ as the second character.**

   ```sql
   SELECT Cust_ID, Cust_Last_Name, Account_No
   FROM Customer_Details
   WHERE Bank_Branch LIKE '_o%';
   ```

5. **List all Account Nos with balance not in the range $10000.00 to $20000.00.**

   ```sql
   SELECT Account_No
   FROM Customer_Transaction
   WHERE Total_Available_Balance_in_Dollars NOT BETWEEN 10000.00 AND 20000.00;
   ```

### 4.6.4.5. **IS NULL, IS NOT NULL**

The NULL value is used to indicate the absence of a value. It is not a zero or blank character. NULL cannot be compared to any other value. If compared, since the result of the comparison cannot be determined, the result of the comparison is also a NULL.
Figure 4-23: NULL value test (IS NULL) syntax diagram

Note: A NULL value is not equal to another NULL value. The result of comparing two NULL values is NULL. It is neither TRUE nor FALSE.

Example:

List employees who have not been assigned a Manager yet.

```sql
SELECT Employee_ID
  FROM Employee_Manager
  WHERE Manager_ID IS NULL;
```

List employees who have been assigned to some Manager.

```sql
SELECT Employee_ID
  FROM Employee_Manager
  WHERE Manager_ID IS NOT NULL;
```

### 4.6.4.6. Column titles using AS

When the SELECT statement returns a column, the title of the result column set is the name of the column. If the statement includes an evaluated expression, the column title is a default name that the RDBMS gives the expression.

To give meaningful column titles use the keyword AS.

Example:

List those customer accounts whose account balance is greater than $10000.00.

```sql
SELECT Account_No AS “Customer Account No.”,
      Total_Available_Balance_in_Dollars AS “Total Balance”
  FROM Customer_Transaction
  WHERE Total_Available_Balance_in_Dollars > 10000.00;
```
4.6.4.7. Sorting Query Results (ORDER BY clause)

The rows of the query results are not arranged in any particular order. SQL can sort the results of a query by including the ORDER BY clause in the SELECT statement. The ORDER BY is a row-wise operation. By default the ORDER BY clause arranges the rows of the query result in ascending order. To arrange the rows of the query result in descending order, use the keyword DESC.

```
ORDER BY Column name1, Column name2, .......... ASC 
Column-number1, Column number2, .... DESC
```

Figure 4-24: The ORDER BY clause syntax diagram

**Example:**

1. List the customers account numbers and their account balances, in the increasing order of the balance.

```sql
SELECT Account_No, Total_Available_Balance_in_Dollars
FROM Customer_Transaction
ORDER BY Total_Available_Balance_in_Dollars;
```

2. List the customers and their account numbers in the decreasing order of the account numbers.

```sql
SELECT Cust_Last_Name, Cust_First_Name, Account_No
FROM Customer_Details
ORDER BY 3 DESC;
```

Note: ORDER BY clause can be followed by the column name or the position of the column as it appears in the SELECT statement.

3. List the customers and their account numbers in the decreasing order of the Customer Last Name and increasing order of account numbers.

```sql
SELECT Cust_Last_Name, Cust_First_Name, Account_No
FROM Customer_Details
ORDER BY Cust_Last_Name DESC, Account_No;
```

Or

```sql
SELECT Cust_Last_Name, Cust_First_Name, Account_No
FROM Customer_Details
ORDER BY 1 DESC, 3;
```
4.6.4.8. **Aggregate Functions / Column Functions**

SQL allows summarizing data from the database through a set of column functions. A SQL column function takes an entire column of data as its arguments and produces a single data item that summarizes the column.

Following are some of the widely used column functions:

- **SUM()**: computes the total of a column
- **AVG()**: computes the average value in a column
- **MIN()**: finds the smallest value in a column
- **MAX()**: finds the largest value in a column
- **COUNT()**: counts the number of non-NULL values in a column
- **COUNT (*)**: counts rows of query results and does not depend on the presence or absence of NULL values in a column. If there are no rows, it returns a value of zero.

**NOTE:** Rows that have a NULL value in the relevant column are ignored by all the above aggregate function except count (*).

![Figure 4-25: Column functions syntax diagram](image-url)
Example:

1. List the minimum account balance.
   SELECT MIN (Total_Available_Balance_in_Dollars)
   FROM Customer_Transaction;

2. List the maximum account balance.
   SELECT MAX (Total_Available_Balance_in_Dollars)
   FROM Customer_Transaction;

3. List the average account balance of customers.
   SELECT AVG (Total_Available_Balance_in_Dollars)
   FROM Customer_Transaction;

4. List total number of account holders in the 'Downtown' Branch.
   SELECT COUNT (*)
   FROM Customer_Details
   WHERE Bank_Branch = 'Downtown';

5. List total number of Customers.
   SELECT COUNT (*)
   FROM Customer_Details;

   SELECT COUNT (*)
   FROM Customer_Details
   WHERE Account_Type = 'Savings';

7. List the minimum and sum of all account balances.
   SELECT MIN (Total_Available_Balance_in_Dollars),
       SUM (Total_Available_Balance_in_Dollars)
   FROM Customer_Transaction;

8. List total number of unique Customer Last Names.
   SELECT COUNT (DISTINCT Cust_Last_Name)
   FROM Customer_Details;

Difference between COUNT(*) and COUNT(Column-name):

9. List total number of Employees.
   SELECT COUNT (*)
   FROM Employee_Manager;

10. List total number of Employees who have been assigned a Manager.
    SELECT COUNT (Manager_ID)
        FROM Employee_Manager;

Note: COUNT (Column-Name) counts the number of non-NULL values in a column whereas COUNT (*) counts rows of query results and does not depend on the presence or absence of NULL values in a column.
4.6.4.9. GROUP BY

The GROUP BY clause is used in a SELECT statement to collect data across multiple records and group the results by one or more columns.

Sometimes it is required to get information not about each row, but about each group.

**Example:** Consider the Customer_Loan table that has data about all the loans taken by all the customers of the bank. Assume that we want to retrieve the total loan-amount of all loans taken by each customer.

Related rows can be grouped together by the GROUP BY clause by specifying a column as a grouping column.

In the above example, the Cust_ID will be the grouping column.

In the output table all the rows with an identical value in the grouping column will be grouped together. Hence, the number of rows in the output is equal to the number of distinct values of the grouping column.

```
SELECT Cust_ID, SUM(Amount_in_Dollars) FROM Customer_Loan GROUP BY Cust_ID;
```

![Figure 4-26: Example of Group BY Clause](image)

---

© Infosys Technologies Limited

Relational Database Management System
**Figure 4-27: Example of GROUP BY clause**

```
SELECT Department, COUNT (Employee_ID) FROM Employee_Manager GROUP BY Department;
```

<table>
<thead>
<tr>
<th>Employee_ID</th>
<th>Employee_Last_Name</th>
<th>Employee_Mid_Name</th>
<th>Employee_First_Name</th>
<th>Employee_Email</th>
<th>Department</th>
<th>Grade</th>
<th>Manager_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>Atherton</td>
<td>S.</td>
<td>Cindy</td>
<td><a href="mailto:Atherton_Cindy@yahoo.com">Atherton_Cindy@yahoo.com</a></td>
<td>HR</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>3556</td>
<td>George</td>
<td>A.</td>
<td>Henry</td>
<td><a href="mailto:George_Henry@rediffmail.com">George_Henry@rediffmail.com</a></td>
<td>Finance</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>3623</td>
<td>Jackson</td>
<td>S.</td>
<td>Matt</td>
<td><a href="mailto:Jackson_Matt@samsonite.co.in">Jackson_Matt@samsonite.co.in</a></td>
<td>Design</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>2278</td>
<td>Stevenson</td>
<td>S.</td>
<td>Crystal</td>
<td><a href="mailto:Stevenson_Crystal@mag.com">Stevenson_Crystal@mag.com</a></td>
<td>HR</td>
<td>2</td>
<td>2345</td>
</tr>
<tr>
<td>2345</td>
<td>Smith</td>
<td>A.</td>
<td>Luther</td>
<td><a href="mailto:Smith_Luther@yahoo.com">Smith_Luther@yahoo.com</a></td>
<td>Finance</td>
<td>2</td>
<td>3556</td>
</tr>
<tr>
<td>30456</td>
<td>Langer</td>
<td>C.</td>
<td>Christina</td>
<td><a href="mailto:Langer_Christina@rediffmail.com">Langer_Christina@rediffmail.com</a></td>
<td>HR</td>
<td>3</td>
<td>2345</td>
</tr>
<tr>
<td>31234</td>
<td>Frost</td>
<td>J.</td>
<td>Robert</td>
<td><a href="mailto:Frost_Robert@training.com">Frost_Robert@training.com</a></td>
<td>Finance</td>
<td>3</td>
<td>3556</td>
</tr>
<tr>
<td>32345</td>
<td>Austen</td>
<td>L.</td>
<td>Jane</td>
<td><a href="mailto:Austen_Jane@yahoo.com">Austen_Jane@yahoo.com</a></td>
<td>Design</td>
<td>2</td>
<td>3620</td>
</tr>
</tbody>
</table>

**Figure 4-28: Example of GROUP BY clause**

```
SELECT Manager_ID, COUNT (Employee_ID) FROM Employee_Manager GROUP BY Manager_ID;
```

<table>
<thead>
<tr>
<th>Employee_ID</th>
<th>Employee_Last_Name</th>
<th>Employee_Mid_Name</th>
<th>Employee_First_Name</th>
<th>Employee_Email</th>
<th>Department</th>
<th>Grade</th>
<th>Manager_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>Atherton</td>
<td>S.</td>
<td>Cindy</td>
<td><a href="mailto:Atherton_Cindy@yahoo.com">Atherton_Cindy@yahoo.com</a></td>
<td>HR</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>3556</td>
<td>George</td>
<td>A.</td>
<td>Henry</td>
<td><a href="mailto:George_Henry@rediffmail.com">George_Henry@rediffmail.com</a></td>
<td>Finance</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>3623</td>
<td>Jackson</td>
<td>S.</td>
<td>Matt</td>
<td><a href="mailto:Jackson_Matt@samsonite.co.in">Jackson_Matt@samsonite.co.in</a></td>
<td>Design</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>2278</td>
<td>Stevenson</td>
<td>S.</td>
<td>Crystal</td>
<td><a href="mailto:Stevenson_Crystal@mag.com">Stevenson_Crystal@mag.com</a></td>
<td>HR</td>
<td>2</td>
<td>2345</td>
</tr>
<tr>
<td>2345</td>
<td>Smith</td>
<td>A.</td>
<td>Luther</td>
<td><a href="mailto:Smith_Luther@yahoo.com">Smith_Luther@yahoo.com</a></td>
<td>Finance</td>
<td>2</td>
<td>3556</td>
</tr>
<tr>
<td>30456</td>
<td>Langer</td>
<td>C.</td>
<td>Christina</td>
<td><a href="mailto:Langer_Christina@rediffmail.com">Langer_Christina@rediffmail.com</a></td>
<td>HR</td>
<td>3</td>
<td>2345</td>
</tr>
<tr>
<td>31234</td>
<td>Frost</td>
<td>J.</td>
<td>Robert</td>
<td><a href="mailto:Frost_Robert@training.com">Frost_Robert@training.com</a></td>
<td>Finance</td>
<td>3</td>
<td>3556</td>
</tr>
<tr>
<td>32345</td>
<td>Austen</td>
<td>L.</td>
<td>Jane</td>
<td><a href="mailto:Austen_Jane@yahoo.com">Austen_Jane@yahoo.com</a></td>
<td>Design</td>
<td>2</td>
<td>3620</td>
</tr>
</tbody>
</table>

**Query Results**

<table>
<thead>
<tr>
<th>Department</th>
<th>Count(Employee_ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HR</td>
<td>3</td>
</tr>
<tr>
<td>Finance</td>
<td>3</td>
</tr>
<tr>
<td>Design</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Manager_ID</th>
<th>Count(Employee_ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>2</td>
</tr>
<tr>
<td>3556</td>
<td>2</td>
</tr>
<tr>
<td>3620</td>
<td>1</td>
</tr>
</tbody>
</table>

**IF GROUP BY clause has been used in a SELECT statement, all the rows with an identical value in the grouping column will be grouped together.**
Once the GROUP BY clause is used, the aggregate functions in the SELECT statement are calculated after grouping i.e., there is one value of the aggregate column for each value of the grouping column.

**Example:** Refer to Figure 4-28. In the example, the grouping is based on the Manager_ID column. There are three records with NULL values in the Manager_ID column. All the three records are placed in the same group. It is the group with indeterminate values. This does not imply that NULL values are equal.

**Note:** If the GROUP BY clause has been used in a SELECT statement, only the grouping columns (columns on which grouping has been done) or aggregate functions can appear in the column list specified in the SELECT statement.

**Example:**

*Invalid SQL statement*

```sql
SELECT Department, Manager_ID, COUNT(Employee_ID) FROM Employee_Manager GROUP BY Manager_ID;
```

*The above SQL statement should be written as:*

```sql
SELECT Department, Manager_ID, COUNT(Employee_ID) FROM Employee_Manager GROUP BY Manager_ID, Department;
```

Refer to Figure 4-29.
SELECT Department, Manager_ID, COUNT(Employee_ID) FROM Employee_Manager
GROUP BY Manager_ID, Department;

<table>
<thead>
<tr>
<th>Employee_ID</th>
<th>Employee_Last_Name</th>
<th>Employee_Mid_Name</th>
<th>Employee_First_Name</th>
<th>Employee_Email</th>
<th>Department</th>
<th>Grade</th>
<th>Manager_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>Atherton</td>
<td>S.</td>
<td>Cindy</td>
<td><a href="mailto:Atherton_Cindy@yahoo.com">Atherton_Cindy@yahoo.com</a></td>
<td>HR</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>3556</td>
<td>George</td>
<td>A.</td>
<td>Henry</td>
<td><a href="mailto:George_Henry@rediffmail.com">George_Henry@rediffmail.com</a></td>
<td>Finance</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>3622</td>
<td>Jackson</td>
<td>S.</td>
<td>Matt</td>
<td><a href="mailto:Jackson_Matt@samsonite.co.in">Jackson_Matt@samsonite.co.in</a></td>
<td>Design</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>2278</td>
<td>Stevenson</td>
<td>S.</td>
<td>Crystal</td>
<td><a href="mailto:Stevenson_Crystal@mag.com">Stevenson_Crystal@mag.com</a></td>
<td>HR</td>
<td>1</td>
<td>2345</td>
</tr>
<tr>
<td>2345</td>
<td>Smith</td>
<td>A.</td>
<td>Luther</td>
<td><a href="mailto:Smith_Luther@yahoo.com">Smith_Luther@yahoo.com</a></td>
<td>Finance</td>
<td>2</td>
<td>3556</td>
</tr>
<tr>
<td>3045</td>
<td>Langer</td>
<td>C.</td>
<td>Christiana</td>
<td><a href="mailto:Langer_Christiana@rediffmail.com">Langer_Christiana@rediffmail.com</a></td>
<td>HR</td>
<td>2</td>
<td>2345</td>
</tr>
<tr>
<td>3123</td>
<td>Frost</td>
<td>J.</td>
<td>Robert</td>
<td><a href="mailto:Frost_Robert@training.com">Frost_Robert@training.com</a></td>
<td>Finance</td>
<td>3</td>
<td>3556</td>
</tr>
<tr>
<td>3234</td>
<td>Austen</td>
<td>L.</td>
<td>Jane</td>
<td><a href="mailto:Austen_Jane@yahoo.com">Austen_Jane@yahoo.com</a></td>
<td>Design</td>
<td>2</td>
<td>3622</td>
</tr>
</tbody>
</table>

Figure 4-29: An example of GROUP BY clause

4.6.4.10. HAVING

The HAVING clause is used along with the GROUP BY clause. The HAVING clause can be used to select and reject row groups. The format of the HAVING clause is similar to the WHERE clause, consisting of the keyword HAVING followed by a search condition. The HAVING clause thus specifies a search condition for groups.
SELECT Cust_ID, SUM(Amount_in_Dollars) FROM Customer_Loan GROUP BY Cust_ID
HAVING SUM(Amount_in_Dollars) > 4000.00;

```
<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Loan_No</th>
<th>Amount_in_Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1011</td>
<td>8755.00</td>
</tr>
<tr>
<td>103</td>
<td>2010</td>
<td>2555.00</td>
</tr>
<tr>
<td>104</td>
<td>2056</td>
<td>3050.00</td>
</tr>
<tr>
<td>103</td>
<td>2015</td>
<td>2000.00</td>
</tr>
</tbody>
</table>
```

Figure 4-30: An example of HAVING Clause

SELECT Department, COUNT(Employee_ID) FROM Employee_Manager GROUP BY Department HAVING COUNT(Employee_ID) > 2;

```
<table>
<thead>
<tr>
<th>Employee_ID</th>
<th>Employee_Last_Name</th>
<th>Employee_Fist_Name</th>
<th>Employee_Email</th>
<th>Department</th>
<th>Grade</th>
<th>Manager_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>Atherton</td>
<td>Cindy</td>
<td><a href="mailto:Atherton_Cindy@yahoo.com">Atherton_Cindy@yahoo.com</a></td>
<td>HR</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>3550</td>
<td>George</td>
<td>Henry</td>
<td><a href="mailto:George_Henry@rediffmail.com">George_Henry@rediffmail.com</a></td>
<td>Finance</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>3620</td>
<td>Jackson</td>
<td>Matt</td>
<td><a href="mailto:Jackson_Matt@samsonite.co.in">Jackson_Matt@samsonite.co.in</a></td>
<td>Design</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>22788</td>
<td>Stevenson</td>
<td>Crystal</td>
<td><a href="mailto:Stevenson_Crystal@mag.com">Stevenson_Crystal@mag.com</a></td>
<td>HR</td>
<td>2</td>
<td>2345</td>
</tr>
<tr>
<td>23456</td>
<td>Smith</td>
<td>Luther</td>
<td><a href="mailto:Smith_Luther@yahoo.com">Smith_Luther@yahoo.com</a></td>
<td>Finance</td>
<td>2</td>
<td>3556</td>
</tr>
<tr>
<td>30496</td>
<td>Langer</td>
<td>Christiana</td>
<td><a href="mailto:Langer_Christiana@rediffmail.com">Langer_Christiana@rediffmail.com</a></td>
<td>HR</td>
<td>3</td>
<td>2345</td>
</tr>
<tr>
<td>31234</td>
<td>Frost</td>
<td>Robert</td>
<td><a href="mailto:Frost_Robert@training.com">Frost_Robert@training.com</a></td>
<td>Finance</td>
<td>2</td>
<td>3556</td>
</tr>
<tr>
<td>32343</td>
<td>Austen</td>
<td>Jane</td>
<td><a href="mailto:Austen_Jane@yahoo.com">Austen_Jane@yahoo.com</a></td>
<td>Design</td>
<td>2</td>
<td>3620</td>
</tr>
</tbody>
</table>
```

Records from Employee_Manager Table

Figure 4-31: An example of HAVING Clause
Note: The WHERE clause can be used to select and reject the individual rows that participate in a query. The HAVING clause can be used to select and reject row groups.

### 4.6.4.11. Retrieval using UNION

The UNION operation combines the rows from two sets of query results. By default, the UNION operation eliminates duplicate rows as part of its processing.

**Example:**

```sql
SELECT Cust_ID
FROM Customer_Fixed_Deposit
UNION
SELECT Cust_ID
FROM Customer_Loan;
```

Refer to Figure 4-32.

To retain duplicate rows in a UNION operation, specify the ALL keyword immediately following the word UNION.

**Example:**

```sql
SELECT Cust_ID
FROM Customer_Fixed_Deposit
UNION ALL
SELECT Cust_ID
FROM Customer_Loan;
```

Refer to Figure 4-33.
Figure 4-32: Using UNION to combine query results

Figure 4-33: Using UNION ALL to combine query results

There are some restrictions on the table that can be combined by a UNION operation:

- The SELECT statements combined using UNION or UNION ALL must contain the same number of columns
• The data type of each column in the first table must be the same as the data type of the corresponding column in the second table. The data width and column name can differ
• Neither of the two tables can be sorted with the ORDER BY clause. However, the combined query results can be sorted

NOTE: Eliminating duplicate rows from query results is a time-consuming process, especially if the query results contain a large number of rows. If one is sure that the UNION operation cannot produce duplicate rows, one should specifically use the UNION ALL operation because the query will execute much more quickly.

4.6.4.12. Retrieval using INTERSECT

The INTERSECT operation selects the common row from two sets of query results. Refer to Figure 4-34.

Example:

```sql
SELECT Cust_ID
FROM Customer_Fixed_Deposit
INTERSECT
SELECT Cust_ID
FROM Customer_Loan;
```

![Figure 4-34: Using INTERSECT to combine query results](image)
4.6.5. **Sub-Queries**

A sub-query is a query within a query. The results of the sub-query are used by the DBMS to determine the results of the higher-level query that contains the sub-query. Usually, the sub-query appears within the WHERE or HAVING clause of another SQL statement.

```
SELECT [ALL / DISTINCT] column-name1, column-name2, ------ FROM table-specification
   [ WHERE search-condition ]
   [ GROUP BY grouping column ]
   [ HAVING search-condition ]
   [ ORDER BY sort-specification ]
```

Figure 4-35: Basic sub-query syntax diagram

The sub-query is enclosed in parentheses, but otherwise it has a form similar to that of a SELECT statement, with a FROM clause and optional WHERE, GROUP BY, and HAVING clauses. The form of these clauses in a sub-query is identical to that in a SELECT statement, and they perform their normal functions when used within a sub-query.

4.6.5.1. **Independent Sub-Queries**

- Inner Query is independent of Outer Query
- Inner Query is executed first and the results are stored
- Outer Query then runs on the stored results

**Example:** To list the Cust_ID and Loan_No for all Customers who have taken a loan of amount greater than the loan amount of Customer (Cust_ID = 104). 
**Figure 4-36: How an independent sub-query executes**

The inner query, which retrieves the Amount_in_Dollars of Cust_ID, 104 can be executed independent of the outer query. Hence the name independent sub-query.

The inner query needs to be executed only once, since it returns one constant value irrespective of the outer query.

In the above example, the innermost query is executed first, the result is stored and then the outer query is executed for each row of the Customer_Loan table. The inner query is executed only once, while the outer one is executed as many times as the number of rows in the Customer_Loan table.
Example:

1. List customer names of all customers who have taken a loan > $3000.00.

   SELECT Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
   FROM Customer_Details
   WHERE Cust_ID IN
     ( SELECT Cust_ID
       FROM Customer_Loan
       WHERE Amount_in_Dollars > 3000.00);

2. List customer names of all customers who have the same Account_type as Customer ‘Jones Simon’.

   SELECT Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
   FROM Customer_Details
   WHERE Account_Type =
     ( SELECT Account_Type
       FROM Customer_Details
       WHERE Cust_Last_Name = 'Jones'
       AND Cust_First_Name = 'Simon');

3. List customer names of all customers who do not have a Fixed Deposit.

   SELECT Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
   FROM Customer_Details
   WHERE Cust_ID NOT IN
     ( SELECT Cust_ID
       FROM Customer_Fixed_Deposit);

4. List customer names of all customers who have either a Fixed Deposit or a loan but not both at any of the Bank Branches. It will include names that have no fixed deposit and loan as well.

   SELECT Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
   FROM Customer_Details
   WHERE Cust_ID NOT IN
     ( SELECT Cust_ID
       FROM Customer_Loan
       WHERE Cust_ID IN
         (SELECT Cust_ID
          FROM Customer_Fixed_Deposit));

---

4.6.5.2. Co-Related Sub-Queries

In co-related sub-queries, SQL performs a sub-query, once for each row of the main query. The column(s) from the table of the outer query is always referred in the inner query.
Refer to Figure 4-37.

**Example:** To list all Customers who have a fixed deposit of amount less than the sum of all their loans.

Query:

```sql
SELECT Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
FROM Customer_Fixed_Deposit
WHERE Amount_in_Dollars <
(SELECT SUM(Amount_in_Dollars)
FROM Customer_Loan
WHERE Customer_Loan.Cust_ID = Customer_Fixed_Deposit.Cust_ID);
```

**Figure 4-37: A Correlated Query**

**Explanation of the query:**

The inner query is repeated once for every record of the outer query. The outer query uses the `Customer_Fixed_Deposit` table. Refer to Figure 4-38.

**Figure 4-38: Customer_Fixed_Deposit table**

The inner query uses the `Customer_Loan` table. Refer to Figure 4-39.

**Figure 4-39: Customer_Loan table**

The `Customer_Fixed_Deposit` table has three records. The inner query will be repeated three times. This is similar to the nested FOR loop that has been covered in Programming Fundamentals course.

**For the first record in the Customer_Fixed_Deposit table:**

1. The record with the value of 101 in the Cust_ID column of the `Customer_Fixed_Deposit` table is read.
Step 1

**Customer_Fixed_Deposit**

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Cust_Email</th>
<th>Fixed_Deposit_No</th>
<th>Amount_in_Dollars</th>
<th>Rate_of_Interest in_Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
<td>2011</td>
<td>8055.00</td>
<td>6.5</td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>S.</td>
<td>Justin</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
<td>2015</td>
<td>2050.00</td>
<td>6.5</td>
</tr>
<tr>
<td>104</td>
<td>Quails</td>
<td>D.</td>
<td>Jack</td>
<td><a href="mailto:Quails_Jack@yahoo.com">Quails_Jack@yahoo.com</a></td>
<td>3010</td>
<td>3050.00</td>
<td>6.5</td>
</tr>
</tbody>
</table>

2. All records with a value of 101 in the Cust_ID column of the Customer_Fixed_Deposit table are retrieved and their Amount_in_Dollars values are summed up. In the Example, there is only one record with a value of 101 in the Cust_ID column and the Amount_in_Dollars value is $8755.00.

Step 2

**Customer_Loan**

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Loan_No</th>
<th>Amount_in_Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1011</td>
<td>8755.00</td>
</tr>
<tr>
<td>103</td>
<td>2010</td>
<td>2555.00</td>
</tr>
<tr>
<td>104</td>
<td>2056</td>
<td>3050.00</td>
</tr>
<tr>
<td>103</td>
<td>2015</td>
<td>2000.00</td>
</tr>
</tbody>
</table>

Customer_Loan records from Customer_Loan table

3. This value is compared with $8055.00. Since the target value of $8055.00 is less than the Amount_in_Dollars value of $8755.00, the record with the Cust_ID value of 101 is part of the query result.

Step 3

8055.00 < 8755.00 (True)
The record with Cust_ID = 101 from Customer_Fixed_Deposit will occur in the query results

For the second record in the Customer_Fixed_Deposit table:

1. The record with the value of 103 in the Cust_ID column of the Customer_Fixed_Deposit table is read.

Step 1

**Customer_Fixed_Deposit**

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Cust_Email</th>
<th>Fixed_Deposit_No</th>
<th>Amount_in_Dollars</th>
<th>Rate_of_Interest in_Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
<td>2011</td>
<td>8055.00</td>
<td>6.5</td>
</tr>
<tr>
<td>103</td>
<td>Langer</td>
<td>S.</td>
<td>Justin</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
<td>2015</td>
<td>2050.00</td>
<td>6.5</td>
</tr>
<tr>
<td>104</td>
<td>Quails</td>
<td>D.</td>
<td>Jack</td>
<td><a href="mailto:Quails_Jack@yahoo.com">Quails_Jack@yahoo.com</a></td>
<td>3010</td>
<td>3050.00</td>
<td>6.5</td>
</tr>
</tbody>
</table>

2. All records with a value of 103 in the Cust_ID column of the Customer_Loan table are retrieved and their Amount_in_Dollars values are summed up. In the Example, there are two records with a value of 103 in the Cust_ID column and the sum of their Amount_in_Dollars values is $4555.00.
Step 2

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Loan_No</th>
<th>Amount_in_Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1011</td>
<td>8755.00</td>
</tr>
<tr>
<td>103</td>
<td>2010</td>
<td>2555.00</td>
</tr>
<tr>
<td>104</td>
<td>2056</td>
<td>3050.00</td>
</tr>
<tr>
<td>103</td>
<td>2015</td>
<td>2000.00</td>
</tr>
</tbody>
</table>

Customer_Loan records from Customer_Loan table

3. This value is compared with $2060.00. Since the target value of $2060.00 is less than the sum(Amount_in_Dollars) value of $4555.00, the record with the Cust_ID value of 103 is part of the query result.

Step 3

2060.00 < 4555.00 \( \text{True} \)  
The record with Cust_ID = 103 from Customer_Fixed_Deposit will occur in the query results

For the third record in the Customer_Fixed_Deposit table:

- The record with the value of 104 in the Cust_ID column of the Customer_Fixed_Deposit table is read.

Step 1

Customer_Fixed_Deposit

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
<th>Cust_Email</th>
<th>Fixed_Deposit_No</th>
<th>Amount_in_Dollars</th>
<th>Rate_of_Interest_in_Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
<td><a href="mailto:Smith_Mike@yahoo.com">Smith_Mike@yahoo.com</a></td>
<td>2011</td>
<td>8055.00</td>
<td>6.5</td>
</tr>
<tr>
<td>103</td>
<td>Unser</td>
<td>G.</td>
<td>Justin</td>
<td><a href="mailto:Langer_Justin@yahoo.com">Langer_Justin@yahoo.com</a></td>
<td>2015</td>
<td>2060.00</td>
<td>6.5</td>
</tr>
<tr>
<td>104</td>
<td>Qualis</td>
<td>D.</td>
<td>Jack</td>
<td><a href="mailto:Qualis_Jack@yahoo.com">Qualis_Jack@yahoo.com</a></td>
<td>3010</td>
<td>3050.00</td>
<td>6.5</td>
</tr>
</tbody>
</table>

- All records with a value of 104 in the Cust_ID column of the Customer_Loan table are retrieved and their Amount_in_Dollars values are summed up. In the Example, there is only one record with a value of 104 in the Cust_ID column and the Amount_in_Dollars value is $3050.00.

Step 2

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Loan_No</th>
<th>Amount_in_Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1011</td>
<td>8755.00</td>
</tr>
<tr>
<td>103</td>
<td>2010</td>
<td>2555.00</td>
</tr>
<tr>
<td>104</td>
<td>2056</td>
<td>3050.00</td>
</tr>
<tr>
<td>103</td>
<td>2015</td>
<td>2000.00</td>
</tr>
</tbody>
</table>

Customer_Loan records from Customer_Loan table

- This value is compared with $3050.00. Since the target value of $3050.00 is equal to the Amount_in_Dollars of $3050.00, the record with the Cust_ID value of 104 is not part of the query result.
Step 3

3050.00 < 3050.00 (False)
The record with Cust_ID = 104 from Customer_Fixed_Deposit will not occur in the query results

Output of the co-related query:

Output Table

<table>
<thead>
<tr>
<th>Cust_ID</th>
<th>Cust_Last_Name</th>
<th>Cust_Mid_Name</th>
<th>Cust_First_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Smith</td>
<td>A.</td>
<td>Mike</td>
</tr>
<tr>
<td>103</td>
<td>Anger</td>
<td>G.</td>
<td>Justin</td>
</tr>
</tbody>
</table>

Figure 4-40: Output of co-related query

Note: For each row of the Customer_Fixed_Deposit table to be tested by the WHERE clause of the main query, the Cust_ID column (which appears in the sub-query as an outer reference) has a different value. Thus SQL carries out this sub-query - once for each row in the Customer_Fixed_Deposit table.

A sub-query containing an outer reference is called a correlated sub-query because its results are correlated with each individual row of the main query. For the same reason, an outer reference is sometimes called a correlated reference.

Example:

List customer IDs of all customers who have both a Fixed Deposit and a loan at any of Bank Branches.

SELECT Cust_ID
FROM Customer_Details
WHERE Cust_ID IN
  (SELECT Cust_ID
   FROM Customer_Loan
   WHERE Customer_Loan.Cust_ID = Customer_Details.Cust_ID)
AND Cust_ID IN
  (SELECT Cust_ID
   FROM Customer_Fixed_Deposit
   WHERE Customer_Fixed_Deposit.Cust_ID = Customer_Details.Cust_ID);

4.6.6. Joins

Join operations take two tables and return another table as a result.
**Cartesian Product / Cross Join**

Cross joins return all rows from the first table. Each row from the first table is combined with all rows from the second table. Cross joins are also known as the *Cartesian product*[^1] (or just the product) of two tables. The columns of the product table are all the columns of the first table, followed by all the columns of the second table.

Refer to Figure 4-41.

![Cartesian Product Diagram](image)

**Figure 4-41: The Cartesian product of two tables**

[^1]: *Cartesian product*: A mathematical term that, when applied to relational databases, refers to the result obtained by joining all the rows of one table with all the rows of another table in every possible combination.
4.6.6.1. **SELF JOIN**

Joining a table with itself is a self-join.

**Example:**

**Problem Statement:** To list all the Employees (Employee_ID, Employee_Last_Name, Employee_First_Name) along with their Managers (Manager_ID, Manager_Last_Name, Manager_First_Name).

**Query:**

```
SELECT Emp.Employee_ID as "Employee ID", Emp.Employee_Last_Name as "Employee Last Name", Emp.Employee_First_Name as "Employee First Name", Emp.Manager_ID as "Manager ID", Manager.Employee_Last_Name as "Manager Last Name", Manager.Employee_First_Name as "Manager First Name"
FROM Employee_Manager Emp, Employee_Manager Manager
WHERE Emp.Manager_ID = Manager.Employee_ID;
```

**Processing of the Query:**

**Step 1:** The table Employee_Manager has two aliases (another name), Emp and Manager.

**Step 2:** Manager_ID attribute of Emp (alias for Employee_Manager) is matched with Employee_ID attribute of Manager (alias for Employee_Manager). The figure below shows the matching of only two records. The other records are matched similarly.
**Step 3:** The columns that appear in the output table are specified in the column list used with the SELECT statement.

Query Results

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Employee Last Name</th>
<th>Employee First Name</th>
<th>Manager ID</th>
<th>Manager Last Name</th>
<th>Manager First Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>22789</td>
<td>Stevenson</td>
<td>Crystal</td>
<td>2345</td>
<td>Atherton</td>
<td>Cindy</td>
</tr>
<tr>
<td>23456</td>
<td>Smith</td>
<td>Luther</td>
<td>3556</td>
<td>George</td>
<td>Henry</td>
</tr>
<tr>
<td>30456</td>
<td>Langer</td>
<td>Christiana</td>
<td>2345</td>
<td>Atherton</td>
<td>Cindy</td>
</tr>
<tr>
<td>31234</td>
<td>Frost</td>
<td>Robert</td>
<td>3556</td>
<td>George</td>
<td>Henry</td>
</tr>
<tr>
<td>32345</td>
<td>Austen</td>
<td>Jane</td>
<td>3620</td>
<td>Jackson</td>
<td>Matt</td>
</tr>
</tbody>
</table>

*Figure 4-42: Output of SELF JOIN*
4.6.6.2. INNER JOINS

An inner join between two (or more) tables is the Cartesian product that satisfies the join condition in the WHERE clause.

Inner joins use a comparison operator like = or <> to match rows from two tables based on the values in common columns from each table.

Inner Joins include Equi-Joins A join in which the joining condition is based on equality between values in the common columns.

Example:

```
SELECT Table1.Emp_ID, Table1.City, Table2.Cust_ID, Table2.City
FROM Table1, Table2
WHERE Table1.City = Table2.City;
```

![Diagram of inner join between two tables and an output table](image_url)

**Figure 4-43: An Example of inner join**
4.6.6.3. OUTER JOINS

An inner join provides only those values that satisfy the WHERE condition. However, it may be worthwhile sometimes, to retrieve all rows that match the WHERE clause and those that have unmatched rows in the column being compared. An outer join is then used to retrieve the rows with an unmatched value in the relevant column.

Refer to Figure 4-44.

Constructing a FULL OUTER JOIN:

- Begin with the INNER JOIN of the two tables, using matching columns
- For each row of the left table that is not matched by any row in the right table, add one row to the query results, using the values of the columns in the left table, and assuming a NULL value for all columns of the right table
- For each row of the right table that is not matched by any row in the left table, add one row to the query results, using the values of the columns in the right table, and assuming a NULL value for all columns of the left table

![Diagram of outer join](image)

**Figure 4-44: an example of OUTER JOIN**

**Note:** Full Outer Join is supported by Oracle 9i and later versions.
### 4.6.6.4. LEFT OUTER JOIN

**Constructing a LEFT OUTER JOIN:**

- Begin with the INNER JOIN of the two tables, using matching columns.
- For each row of the left table that is not matched by any row in the right table, add one row to the query results, using the values of the columns in the left table, and assuming a NULL value for all columns of the right table.

Refer to Figure 4-45.

*Note:* The LEFT OUTER JOIN thus includes NULL-extended copies of the unmatched rows from the first (left) table but does not include any unmatched rows from the second (right) table.

**Example:** The syntax given is Oracle specific.

```sql
SELECT Table1.Emp_ID, Table1.City, Table2.Cust_ID, Table2.City
FROM Table1, Table2
WHERE Table1.City = Table2.City (+);
```

![Diagram showing an example of LEFT OUTER JOIN]

**Figure 4-45:** An example of LEFT OUTER JOIN
4.6.6.5. **RIGHT OUTER JOIN**

**Constructing a RIGHT OUTER JOIN:**

- Begin with the INNER JOIN of the two tables, using matching columns
- For each row of the right table that is not matched by any row in the left table, add one row to the query results, using the values of the columns in the right table, and assuming a NULL value for all columns of the left table

Refer to Figure 4-46.

*Note:* The RIGHT OUTER JOIN thus includes NULL-extended copies of the unmatched rows from the SECOND (right) table but does not include any unmatched rows from the first (left) table.

**Example:** The syntax given is Oracle specific.

```sql
SELECT Table1.Emp_ID, Table1.City, Table2.Cust_ID, Table2.City
FROM Table1, Table2
WHERE Table1.City (+) = Table2.City;
```

![Figure 4-46: an example of RIGHT OUTER JOIN](image)
4.6.7. Queries using EXISTS / NOT EXISTS

4.6.7.1. EXISTS

The EXISTS checks whether a sub-query produces any row(s) of results.

Consider a nested query. If the query following the EXISTS returns at least one row, the EXISTS returns TRUE and stops further execution of the inner SELECT statement. The outer query will be executed only if the EXISTS returns true.

If the inner query produces no rows, the EXISTS returns FALSE and the outer query will not be executed. The EXISTS test cannot produce a NULL value.

Example:

1. List all Customers who have at least one Fixed Deposit more than $3000.00.

   SELECT Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
   FROM Customer_Details CD
   WHERE EXISTS
       (SELECT *
        FROM Customer_Fixed_Deposit CFD
        WHERE CFD.Amount_in_Dollars > 3000.00 AND CFD.Cust_ID = CD.Cust_ID);

   Note: CD is the alias for Customer_Details. CFD is the alias for Customer_Fixed_Deposit.

2. List all Customers who have both a Fixed Deposit and a Loan at the Bank.

   SELECT Cust_ID
   FROM Customer_Fixed_Deposit
   WHERE EXISTS
       (SELECT *
        FROM Customer_Loan
        WHERE Customer_Loan.Cust_ID = Customer_Fixed_Deposit.Cust_ID);

4.6.7.2. NOT EXISTS

The logic of the EXISTS test can be reversed by using the NOT EXISTS form. In this case, the test is TRUE if the sub-query produces no rows, and FALSE otherwise.
Example:

List all Customers who do not have a single Fixed Deposit over $3000.00.

```
SELECT Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
FROM Customer_Details CD
WHERE NOT EXISTS
  (SELECT *
   FROM Customer_Fixed_Deposit CFD
   WHERE CFD.Amount_in_Dollars > 3000.00 AND CFD.Cust_ID = CD.Cust_ID);
```

4.6.8. The Order of Execution of a SELECT statement

If a SELECT Statement contains a WHERE, GROUP BY, HAVING and ORDER BY CLAUSE, the order of execution is as follows:

1. The WHERE clause is applied first, and the rows for which the search condition in the WHERE clause returns a TRUE are retained.

2. Next a GROUP BY clause is applied. It will group the rows selected by the WHERE clause such that all the rows in each group have the same value for the column in the GROUP BY clause.

3. Next the HAVING clause is applied. It will retain row groups for which the search condition in the HAVING clause returns a TRUE value.

4. Lastly the query result is sorted in the order specified in the ORDER BY clause.
4.7. Views

A view is a virtual table in the database defined by a query. A view does not exist in the database as a stored set of data values. The rows and columns of data visible through the view are produced by the query that defines the view.

```
CREATE VIEW view-name column-name1, column-name2, ----------- AS query
```

Figure 4-47: The CREATE VIEW statement syntax diagram

4.7.1. Horizontal View

Horizontal view restricts a user’s access to selected rows of a table.

```
CREATE VIEW view_cust AS

SELECT *

FROM Customer_Details

WHERE Cust_ID in (101, 102, 103);
```

Figure 4-48: Horizontal View

4.7.2. Vertical View

Vertical view restricts a user’s access to select columns of a table.

```
CREATE VIEW view_cust AS

SELECT Cust_ID, Account_No, Account_Type

FROM Customer_Details;
```

Figure 4-49: Vertical View
4.7.3. **DROP VIEW Statement**

The DROP VIEW statement is used to drop a view.

```
DROP VIEW view-name
```

*Figure 4-50: DROP VIEW statement syntax diagram*

4.7.4. **Joined Views**

Joined Views are used to simplify multi-table queries. A joined view draws its data from two or three different tables and presents the query results as a single virtual table. Once the view is defined, one can use a single-table query against the view for requests that would otherwise each require a two-table or three-table join.

```
CREATE VIEW Cust_View As

SELECT Customer_Details.Cust_Last_Name, Customer_Details.Cust_First_Name, Fixed_Deposit_No, Amount_in_Dollars
FROM Customer_Details, Customer_Fixed_Deposit
WHERE Customer_Details.Cust_ID = Customer_Fixed_Deposit.Cust_ID;
```

*Figure 4-51: Joined Views*

A view can be referenced like a real table in a SELECT, INSERT, DELETE, or UPDATE statement. However, more complex views cannot be updated; they are read only views.

4.7.5. **VIEW Updates**

A view can be updated if the query that defines the view meets all of the following restrictions:

- DISTINCT must not be specified; that is, duplicate rows must not be eliminated from the query results
- The FROM clause must specify only one updateable table; the view must have a single underlying source table
- The SELECT list cannot contain expressions, calculated columns, or column functions
- The WHERE clause must not include a sub query; only simple row-by-row search conditions may be used
- The SELECT list must include all the columns specified with the NOT NULL constraint
4.7.6. Checking View Updates (CHECK OPTION)

If a view is defined by a query that includes the WHERE clause, only rows that meet the search criteria are visible in the view. Other rows may be present in the source table(s) from which the view is derived, but they are not visible through the view.

Example:

```sql
CREATE VIEW view_customer AS
    SELECT Cust_ID, Cust_Last_Name, Account_No, Account_Type, Bank_Branch
    FROM Customer_Details
    WHERE Bank_Branch = 'Downtown';
```

Figure 4-52: Creation of a simple view

```sql
INSERT INTO view_customer
VALUES (115, 'Costner', 107, 'Savings', 'Bridgewater');
```

Figure 4-53: Insertion in a simple view

**Note:** This is a perfectly valid SQL statement, and the RDBMS inserts a new row with the specified column values into the Customer_Details table. However, the newly inserted row does not meet the search condition for the view. As a result, if one runs this query immediately after the INSERT statement the newly added row does not show up in the view.

```sql
SELECT Cust_ID, Cust_Last_Name, Bank_Branch
FROM view_customer;
```

SQL can allow DBMS to detect and prevent this type of INSERT or UPDATE from taking place through the view by creating the view with the CHECK OPTION. The CHECK OPTION is specified in the CREATE VIEW statement, as shown below:
CREATE VIEW view_customer AS

    SELECT Cust_ID, Cust_Last_Name, Account_No, Account_Type, Bank_Branch
    FROM Customer_Details
    WHERE Bank_Branch = 'Downtown'

    WITH CHECK OPTION;

Figure 4-54: Create view with CHECK OPTION

4.7.7. Advantages of Views

- **Security**: A user can be permitted to access the database, only through a small set of views that contain the specific data the user is authorized to see.

- **Query simplicity**: A view can draw data from several different tables and present it as a single table, thus effectively turning multi-table queries into single-table queries. Internally, RDBMS uses multi-table queries.

- **Structural simplicity**: Views can give a user, a personalized view of the database structure, presenting the database as a set of virtual tables that make sense to the user.

- **Data integrity**: If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets specified integrity constraints.

4.7.8. Disadvantages of Views

- **Performance**: The DBMS translates the queries against the view into queries against the underlying source tables. If a view is defined by a multi-table query, then even a simple query against a view becomes a complicated join, and it may take a long time to complete. This is in reference to insert, delete, and update operations.

- **Update restrictions**: When a user tries to update rows of a view, the DBMS must translate the request into an update on rows of the underlying source tables. This is possible for simple views, but more complicated views cannot be updated.
4.8. Data Control Language (DCL)

DCL statements are used to control access to the database and the data in it. It is used to enforce data security.

4.8.1. Granting Privileges

The GRANT statement is used to grant security privileges on database objects to specific users. Normally, the GRANT statement is used by the owner of the table or view to give other users access to the data.

**Example:**

```sql
GRANT SELECT, INSERT
    ON Customer_Details
    TO Edwin ;

GRANT ALL PRIVILEGES
    ON Customer_Loan
    TO JACK ;

GRANT ALL
    ON Customer_Loan
    TO PUBLIC ;
```

Figure 4-55: The GRANT statement syntax diagram
4.8.1.1. Passing Privileges (GRANT OPTION)

A GRANT statement with the WITH GRANT OPTION clause conveys, along with the specified privileges, the right to grant those privileges to other users.

```
1. WITH GRANT OPTION

   EDWIN

2. GRANT

   JACK

   BORIS
```

Figure 4-56: Using the GRANT OPTION

4.8.2. Revoking Privileges (REVOKE)

The REVOKE statement is used to REVOKE privileges previously granted with the GRANT statement.

```
REVOKE SELECT/INSERT/DELETE/UPDATE/ALL PRIVILEGES ON table-name
FROM user-name/PUBLIC
```

Figure 4-57: The REVOKE statement syntax diagram
Example:

REVOKE SELECT, INSERT
     ON Customer_Details
     FROM Edwin ;

REVOKE ALL PRIVILEGES
     ON Customer_Loan
     FROM JACK ;

REVOKE ALL
     ON Customer_Loan
     FROM PUBLIC ;

Figure 4-58: REVOKE with CASCADE
4.9. Embedded SQL

4.9.1. Purpose

To blend SQL language statements directly into a program written in a host programming languages, such as C, Pascal, COBOL, FORTRAN and PL/I, use embedded SQL statements.

The following techniques are used to embed the SQL statements:

- SQL statements are intermixed with statements of the host language in the source program. This embedded SQL source program is submitted to a SQL pre-complier, which processes the SQL statements.
- Variables of the host programming language can be referenced in the embedded SQL statements, allowing values calculated by the program to be used by the SQL statements.
- Program language variables are also used by the embedded SQL statements to receive the results of SQL queries, allowing the program to use and process the retrieved values.
- Special program variables are used to assign NULL values to database columns and to support the retrieval of NULL values from the database.

4.9.2. Why Embedded SQL?

SQL has the following limitations:

- No provision to declare variables
- No unconditional branching/jump statement
- No IF statement to test conditions
- No FOR, DO or WHILE statements to construct loops
- No block structure
In order to understand the embedded SQL program, one has to be familiar with the following terminologies:

**EXEC SQL:** Every embedded SQL statement begins with an introducer that flags it as a SQL statement. The IBM SQL products use the introducer exec sql for most host languages.

**SQLCA:** The sqlca (SQL Communication Area) is a data structure that contains error variables and status indicators. By examining the SQLCA, the application program can determine the success or failure of its embedded SQL statements.

exec sql include sqlca;

This statement tells the SQL pre-complier to include a SQL Communications Area in this program.

As RDBMS executes each embedded SQL statement, it sets the value of the variable sqlcode in the SQLCA to indicate the completion status of the statement.

- A sqlcode of **zero** indicates successful completion of the statement
- A **negative** sqlcode indicates a serious error that prevented the statement from executing correctly
- A **positive** sqlcode indicates a warning condition. The most common warning with a value of 100, is the out of data warning returned when a program tries to retrieve the next row of query results and no more rows are left to retrieve.

**Host variables:** A host variable is a program variable. It is declared using the data types of the programming language such as “C” and manipulated by programming language statements. A host variable is also used in embedded SQL statements to store/retrieve data to/from the database. To identify the host variable, the variable is prefixed by a colon (:) when it appears in an embedded SQL statement. A host variable can appear in an embedded SQL statement wherever a constant can appear.

The two embedded SQL statements **begin declare section** and **end declare section** bracket the host variable declarations and are non-executable.

**Use of host variables to store data into the database:**
- The input provided by the user using the standard input device is stored in the host variables
- Values of the host variable is then written to the database using the INSERT SQL statement

**Use of host variables to retrieve data from the database:**
- The data values retrieved from the database using the SELECT SQL statement are held in the host variables
• The contents of the host variables are then displayed on the standard output device using functions such as printf() in “C”

**Indicator variables:** To store NULL values in the database or retrieve NULL values from the database, embedded SQL allows each host variable to have a companion host indicator variable. In an embedded SQL statement, the host variable and the indicator variable together specify a single SQL-style value, as follows:

• An indicator value of **zero** indicates that the host variable contains a valid value
• A **negative** indicator value indicates that the host variable should be assumed to have a NULL value; the actual value of the host variable is irrelevant and should be disregarded
• A **positive** indicator value indicates that the host variable contains a valid value, which may have been rounded off or truncated

A host variable is immediately followed by the name of the corresponding indicator variable. Both variable names are preceded by a colon.

**Example:** A simple embedded SQL program written in C.

**Problem statement:** This program asks the customer for his Cust_ID, retrieves his record from the Customer_Details table and displays it on the standard output device.

```c
int main(int argc, char* argv)
{
    /* inclusion of the SQL Communication Area in the program */
    exec sql include sqlca;

    /* declaration of the HOST VARIABLES */
    exec sql begin declare section
        char Mem_Cust_ID[5];
        char Mem_Cust_Last_Name[25];
        char Mem_Account_No[5];
        char Mem_Bank_Branch[25];
        char Mem_Cust_Email[30];
        short iBank_Branch;
    exec sql end declare section;

    /* Prompt the user for Customer ID */
    printf(“Enter Customer ID:”);
    scanf(“%s”, Mem_Cust_ID);
```
/* execute the SQL query */
/* HOST VARIABLES are preceded by a colon (:) e.g.: Mem_Cust_ID */
/* HOST Variable followed by a companion host indicator variable e.g. */
/* Mem_Bank_Branch :iBank_Branch */

exec sql
SELECT Cust_ID, Cust_Last_Name, Account_No, Bank_Branch, Cust_Email
FROM Customer_Details
WHERE Cust_ID =:Mem_Cust_ID
INTO :Mem_Cust_ID, :Mem_Cust_Last_Name,
     :Mem_Account_No, :Mem_Bank_Branch :iBank_Branch, :Mem_Cust_Email;

/* Display the retrieved data */
/* sqlca.sqlcode contains status information of the embedded SQL statement executed */
if (sqlca.sqlcode = = 0)  {
    printf("Customer ID: %s\n", Mem_Cust_ID);
    printf("Customer Name: %s\n", Mem_Cust_Last_Name);
    printf("Account No.: %s\n", Mem_Account_No);

    /* checking the value of the INDICATOR VARIABLE */
    if (iBank_Branch < 0)
        printf("Bank Branch is NULL\n");
    else
        printf("Bank Branch: %s\n", Mem_Bank_Branch);

    printf("Customer Email: %s\n", Mem_Cust_Email);
}
else if (sqlca.sqlcode = = 100)
    printf("No customer with that Customer ID.\n");
else
    printf("SQL error: %d\n", sqlca.sqlcode);

/* returns success code to the operating system */
return 0;

4.10. Best Practices

1. Do not use `SELECT *`. This is time-consuming and reduces performance. Instead, list out each field that is required.

2. It is potentially dangerous to use `SELECT *` in embedded SQL i.e. SQL embedded in an application program because the meaning of the asterisk (*) might change. *Example*: if a column is added to or dropped from some table.

3. While Evaluating NULL in a `WHERE` clause of a query, use `IS NULL` as opposed to `= NULL`.

4. If one is sure that the UNION operation cannot produce duplicate rows, use the `UNION ALL` as opposed to `UNION` because the query will execute much more quickly.

5. If the `GROUP BY` clause has been used in a `SELECT` statement, then use only the grouping columns (columns on which grouping has been done) or aggregate functions in the column list of the `SELECT` statement.

6. Rows that have a NULL value in the relevant column are ignored by all the aggregate function except `count (*)`.

7. `INDEX` is most appropriate when queries against a table are more frequent than `INSERT` and `UPDATE` operations.

8. `EXISTS` is beneficial when the most selective filter is in the parent query. This allows the selective predicates in the parent query to be applied before filtering the rows against the `EXISTS` criteria.

9. `IN` is most beneficial when the most selective filter appears in the sub-query and there are indexes on the join columns.

Tips to write a good query:

1. `SELECT account_no, trans_date, amount`
   `FROM transaction`
   `WHERE amount + 3000 < 5000;`

   *Replace the above query with the following*

   ```sql
   SELECT account_no, trans_date, amount
   FROM transaction
   WHERE amount < 2000;
   ```

   *Reason:* Avoid unnecessary computational overhead in queries.
2. SELECT quantity, AVG(actual_price)
   FROM item
   GROUP BY quantity
   HAVING quantity > 40;

*Replace the above query with the following:*

```sql
SELECT quantity, AVG(actual_price)
FROM item
WHERE quantity > 40
GROUP BY quantity;
```

*Reason:* The `WHERE` clause filters the rows from the table according to the search condition. Then the `GROUP BY` clause is applied only on the filtered rows. It saves time. If as opposed to this, if the rows are grouped first then the row groups are filtered using the `HAVING` clause, it leads to an increased overhead in terms of time required for execution of the query.

3. **Problem Statement:** To retrieve the average salary for 'presidents' and 'managers'.

   ```sql
   SELECT job, avg(sal)
   FROM emp
   GROUP BY job
   HAVING job = 'president' OR job = 'manager';
   ```

*Replace the above query with the following:*

```sql
SELECT job, avg(sal)
FROM emp
WHERE job = 'president' OR job = 'manager'
GROUP BY job;
```

*Reason:* The `WHERE` clause filters the rows from the table according to the search condition. Then the `GROUP BY` clause is applied only on the filtered rows. It saves time. If as opposed to this, if the rows are grouped first then the row groups are filtered using the `HAVING` clause, it leads to an increased overhead in terms of time required for execution of the query.

4. **Problem Statement:** To select records from debit_transactions table, credit_transactions table where tran_date is '31-DEC-99'

   ```sql
   SELECT acct_num, balance_amt
   FROM debit_transactions
   WHERE tran_date = '31-DEC-99'
   ```

   ```sql
   UNION
   ```

   ```sql
   SELECT acct_num, balance_amt
   FROM credit_transactions
   WHERE tran_date = '31-DEC-99';
   ```

*Replace the above query with the following:*
SELECT acct_num, balance_amt
FROM debit_transactions
  WHERE tran_date = '31-DEC-99'
UNION ALL
SELECT acct_num, balance_amt
FROM credit_transactions
  WHERE tran_date = '31-DEC-99';

**Reason:** Eliminating duplicate rows from query results is a very time-consuming process, especially if the query results contain a large number of rows. If one is sure that the UNION operation cannot produce duplicate rows, one should specifically use the UNION ALL operation because the query will execute much more quickly.

5. **Problem statement:** To determine if transaction(s) was made on '25-JAN-2005'

```
SELECT COUNT(*)
FROM Customer_Transaction
  WHERE Transaction_Date = '25-JAN-2005';
```

*Replace the above query with the following:*

```
SELECT Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
FROM Customer_Details
  WHERE EXISTS
    (SELECT Cust_ID
     FROM Customer_Transaction
     WHERE Transaction_Date = '25-JAN-2005');
```

**Reason:** When `COUNT (*)` is used, it scans the entire table which is a time consuming operation. If `EXISTS` is used, it checks whether a sub-query produces any rows of query results. If the sub-query following the `EXISTS` returns at least one row, the `EXISTS` test returns `TRUE` and stops further execution of the inner `SELECT` statement. It thus minimizes overhead.
4.11. Summary

- The CREATE TABLE statement creates a table and defines its columns, PRIMARY KEY, FOREIGN KEY(s) and other constraints like UNIQUE and NOT NULL.

- The DROP TABLE statement removes a previously created table from the database.

- The ALTER TABLE statement can be used to add a column to an existing table, modify a column definition, add/drop a PRIMARY KEY, FOREIGN KEY and other constraints like UNIQUE and NOT NULL.

- The CREATE INDEX statement can be used to define indexes, which speeds up database queries but add overheads to database updates.

- If a SELECT Statement contains a WHERE, GROUP BY, HAVING and ORDER BY CLAUSE, the order of execution is as follows:
  
  - The WHERE clause is applied first, and the rows for which the search condition in the WHERE clause returns a TRUE are retained.

  - Next a GROUP BY clause is applied. It will group the rows selected by the WHERE clause such that all the rows in each group have the same value for the column in the GROUP BY clause.

  - Next the HAVING clause is applied. It will retain row groups for which the search condition in the HAVING clause returns a TRUE value.

  - Lastly the query result is sorted in the order specified in the ORDER BY clause.

- DCL statements are used to control access to the database and the data in it. It is used to enforce data security.
5. **On-Line Transaction Processing (OLTP)**

“In our database we have 270 million records for one million customers!”  
- Anonymous

5.1. **Purpose**

The biggest responsibility of the modern day information system is
- To *simulate*\(^\text{42}\) the manual system
- To record every transaction that the organization undertakes
- Capture the day-to-day activities in the life cycle of an enterprise
- Help the organization to make quick and correct decisions based on the data
- Protect the data from unauthorized access
- Recovering the data in case of failures

Every organization needs some on-line systems to manage their day to day activities. These systems help in recording the transactions, the organization goes through with their employees, customers and vendors. It is impossible to imagine an enterprise without an on-line transaction system.

To build an efficient on-line transaction system, it is necessary to know how these systems are built, the difficulties that may encounter and how to overcome them.

In this cyber-age, we need to know how to protect data from un-authorized usage and how to recover the data in case of failures.

5.2. **Transaction**

A **transaction** is nothing but an interaction between different users, or different systems or user and a system.

A transaction is a **logical unit of work** which takes the database from one consistent state to another consistent state. While moving from one consistent state to another consistent state, the database may pass through multiple discrete steps.

At the end of the transaction, the database may move back to its original state (in the case of failure) or to the next logical step (in the case of success).

Consider the following examples:

---

\(^{42}\) *Simulate*: To make a model.
**Example1:** Drawing money from a bank account is one transaction. This transaction has multiple steps.

- Insert the ATM card into the ATM machine
- Enter the PIN number
- Machine validates the PIN number
- Choose the appropriate menu for money withdrawal
- The ATM machine checks for the account balance to ensure that all banking business rules are strictly followed.
- After doing all the checks, the ATM machine correctly dispenses out the exact amount and updates the records accordingly

For any reason, if any of the above steps fail, then the transaction itself fails and the records are not modified. This means the system is goes back to original state and there will not be any changes to the system.

If ALL the steps have been successfully carried out, then the records are updated accordingly and the system is ready for the next transaction.

**Example2:** A person is interested in transferring money from account A to account B. This transaction has following steps:

- Insert the ATM card into the ATM machine
- Enter the PIN number
- Machine validates the PIN number
- Choose the appropriate menu for money transfer
- Enter information of the beneficiary account
- The ATM machine checks for the account balance to ensure that all banking business rules are strictly followed
- After verifying the balance, amount will be debited from account A and the records are updated accordingly
- The debited amount will be credited to Account B and the records are updated accordingly

From the above steps, it is evident that the transaction may have ‘n’ number of physical steps. transaction is successful only if ALL the steps are carried out successfully. A transaction cannot be divided into smaller tasks.

The successful completion of the transaction is called as the **commit** state. After this state, changes are permanent and irreversible.

If ANY ONE step fails, the complete transaction fails and the system is taken back to the original state that existed before the start of the transaction. This process of going back to original state is called as **rollback**.

If the transaction rolls back, then the transaction reaches the **abort** state.
The Transaction state diagram is as shown in Figure 5-1:

![Transaction State Diagram](image_url)

Figure 5-1: Transaction State Diagram

### 5.3. Transaction Systems

The transaction processing (TP) systems which mimic the real life system like Salary processing, library, banking, airline, defence missile systems are basically divided into three categories.

#### 5.3.1. Batch Transaction Processing System

In the batch transaction processing system, a set of application programs will be working on a set of input data to produce the desired output. In this process there will be absolutely NO human interaction.

The best example for batch processing is the salary slip generation application. The salary slip generation program may read the data like employee name, grade, basic salary, date of joining, overtime for the week, loss of pay, loans, recoveries, etc., from the database and generates the salary slips. Mail will be sent to employees automatically, without any intervention by the users.
### 5.3.2. On-line Transaction Processing System (OLTP)

In OLTP system, the user will be continuously interacting with the system through a computer or a terminal on a regular basis. Some examples for the online systems are the Air-line reservation, Railway reservation system, the Banking ATM machine, the Library application, etc.

In these kind of systems, the user needs to enter pre-defined inputs like flight number, train number, date of journey, amount to withdraw, book access number, return date, etc. Based on these pre-defined inputs, the system produces pre-defined outputs like the confirmed tickets, or non availability of ticket, the issuing of library book for a certain period, etc.

We shall study in-depth about the OLTP system in this chapter.

### 5.3.3. Real time Transaction Processing System

This system is the most complicated among all the transaction systems. It is capable of handling unexpected inputs to unexpected outputs. *Examples: Air traffic control system or Missile defence system.*

These systems are capable of handling a sudden change in the air pressure, the temperature, the wind direction, the target speed and the direction and can change their output based on these inputs.

These real time applications are similar to on-line systems except for the reason that input and output to the system is not all the time pre-defined.

### 5.4. Transaction Properties

Every transaction system must possess the following characteristics:

**Atomicity**: Transactions should either completely succeed or completely fail. For any reasons, if the system crashes before the transaction completes, the database state should not change. The data in the database, which was involved with the transaction, should be restored to the previous consistent state. The transaction is indivisible which means it cannot be divided into sub tasks.

**Consistency**: Transactions must preserve database consistency. A transaction transforms the database from one consistent state to another consistent state.

**Isolation**: A transaction's operations like SELECT, INSERT, UPDATE and DELETE do not interfere with other transactions, or other users of the database. Until a transaction completely succeeds, the database system conceals the individual changes from other transactions.
Durability: Once a transaction completes (commits), the changes made to database permanent and are available to all the transactions that follow it.

These properties are called as ACID (derived from the first letter of the above characteristics) properties.

5.5. Requirements for an OLTP System

In addition to ACID properties, OLTP systems have additional requirements to meet. In the following sections, these requirements are discussed.

5.5.1. Integrity

All the data entering into the system must be validated for its correctness and adherence to the organization’s business rules. This is implemented in RDBMS through three types of integrity checks.

Domain Integrity involves the implementation of business domain specific rules.

Example: If an organization decides not to hire an employee who is less than 18 years and above 60 years. This can be implemented using the CHECK constraint.

```sql
CREATE TABLE Employee(
    Emp_number Number(5) PRIMARY KEY,
    Emp_Name  Varchar2(25) NOT NULL,
    Dept_number Number(5) REFERENCES DEPARTMENT(Dept_number),
    Date_of_Birth Date NOT NULL,
    Date_of_joining Date Default sysdate,
    CHECK ((Date_of_joining - Date_of_Birth) >= 18 AND
    (Date_of_joining - Date_of_Birth) <= 60 ))
```

Entity integrity is implemented using the primary key constraint. Basically entity integrity refers to the fact that a particular attribute uniquely identifies the physical entity.

Example: For each employee, the employee number uniquely identifies an employee. This means employee number 3882 always represent details of employee Hanumesh. Hence entity integrity enforces that primary keys cannot have null values and duplicate values.

Referential Integrity is implemented using the relationships between primary keys and foreign keys of a tables within a database. This ensures data consistency. Referential integrity requires that the value of every foreign key in every table be matched by the value of a primary key in another table. This relationship is called as parent-child relationship.
For example, every employee of the organization must belong to a valid department. Hence department number column of the employee table refers to the department number column of department table.

One can not insert into the department number column of employee table unless the value is present in the department number column of department table (except for NULL values). If foreign key column is NULL, it represents the 'unknown state' and is not a violation of referential integrity. In above example, department table is parent and employee table is child.

After enforcing referential integrity, the parent table primary key value might be deleted. This will violate the referential integrity of the child table. This is because the child table might still contain records containing the original parent table primary key value.

For example, employee table contains department number as the foreign key referring to the primary key of the department table. If department number is deleted from the parent table (department table), if the employee table contain the corresponding department number value, it leads to violation of referential integrity. To avoid such situations, the following restrictions can be put on the foreign key columns of child table at the time of creation.

- **ON DELETE RESTRICT** – Do not allow to delete the parent table data if it is referred in child table. For example if department number 10 is referred in employee table, then do not allow to delete the department number 10 in department table. This is default clause for Oracle.

- **ON DELETE SET NULL** – On delete of the parent data, set NULL value in child table wherever the deleted data is referred. For example if department number 10 is referred in employee table, and it is deleted in department table, set NULL values in corresponding department number columns of employee table (wherever department number 10 was referred).

- **ON DELETE SET DEFAULT** – Set the default values to child records on deletion of parent records. For example if department number 10 is referred in employee table, and it is deleted in department table, set default values (say 0000) in employee table wherever department number 10 was referred.

- **ON DELETE SET CASCADE** – Delete all the child table records from child table on deletion of parent record in parent table. For example if department number 10 is referred in employee table, and it is deleted in department table delete all the records in employee table wherever department number 10 was referred.
5.5.2. Concurrency

Concurrency means allowing different transactions to execute simultaneously. The biggest challenge of having a concurrent system is maintenance of consistency in the system, in spite of the multiple transactions executing simultaneously.

Consider a simple example to know how concurrency affects consistency.

Assume there are only three seats available between Bangalore and Singapore on a particular day and around ten people are trying to book the ticket on the same flight, the same day.

The system allows transactions to occur simultaneously and these ten people can book those three seats from different locations. These ten people get a ticket each when there were just three seats available! This is a BIG violation of consistency or integrity of the system.

In next section listed all the possible consistency problems with transactions, occurring simultaneously.

5.5.2.1. Lost Update

Let us understand lost update concept using a banking application. Assume Person named Hilary holds an account in the Capital Bank, San Jose branch with USD 1500 account balance.

One fine day she deposits USD 2000 by cash to her account.

While the branch clerk is updating her account at San Jose, her husband Kevin deposits USD 1800 to her account almost at the same time in San Francisco.

Clerk in San Francisco is not aware of the other transaction and adds this amount to her balance. Her balance is now updated to USD 3300 thereby ignoring her last deposit of USD 2000 at San Jose. Basically we lost one update which happened on her account!

This problem has occurred because two transactions are working on the same resource without knowing each other’s activity.

The Figure 5-2 gives the snapshot of main memory in which these two transactions operating on Hilary’s account.

Note that:

43 Main Memory: Please recall CHSSC concepts - All the read and write operations happens in main memory before they are written into hard disks.
This diagram is not RDBMS table. It is a snapshot of main memory at that point of time.

Balance column indicates the current available balance of Hilary when two transactions are concurrently running.

<table>
<thead>
<tr>
<th>Time</th>
<th>Hilary’s Deposit</th>
<th>Balance</th>
<th>Kevin’s Deposit</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:22</td>
<td>Read Balance (1500)</td>
<td>1500</td>
<td></td>
</tr>
<tr>
<td>10:23</td>
<td>Balance=1500+2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:24</td>
<td></td>
<td></td>
<td>Read Balance (1500)</td>
</tr>
<tr>
<td>10:25</td>
<td>Write new Balance (3500)</td>
<td>3500</td>
<td></td>
</tr>
<tr>
<td>10:26</td>
<td>Commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:27</td>
<td></td>
<td></td>
<td>Balance=1500+1800</td>
</tr>
<tr>
<td>10:28</td>
<td></td>
<td>3300</td>
<td>Write new Balance (3300)</td>
</tr>
<tr>
<td>10:29</td>
<td></td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Figure 5-2: Lost Update

5.5.2.2. Dirty Read

Let us re-visit the same example discussed for lost update case with slightly different scenario as shown in Figure 5-3.

This time, Hilary again tries to deposit USD 2000 to her account but due to some technical reasons the transaction will not be successful. We know that a transaction can be either in the prior state or a new state after the completion of the transaction. So, her deposit transaction is aborted and her balance is rolled back to the original value USD 1500.

But unfortunately Kevin’s transaction read the balance value as USD 3500 in main memory before the roll back of previous transaction. Due to this problem, Kevin’s transaction occurring in San Francisco read the dirty data. This kind of problem is known as dirty read.

<table>
<thead>
<tr>
<th>Time</th>
<th>Hilary’s Deposit</th>
<th>Balance</th>
<th>Kevin’s Deposit</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:22</td>
<td>Read Balance (1500)</td>
<td>1500</td>
<td></td>
</tr>
<tr>
<td>11:23</td>
<td>Balance=1500+2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:24</td>
<td>Write new Balance (3500)</td>
<td>3500</td>
<td></td>
</tr>
<tr>
<td>11:25</td>
<td></td>
<td></td>
<td>Read Balance (3500)</td>
</tr>
<tr>
<td>11:26</td>
<td>Rollback</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:27</td>
<td></td>
<td></td>
<td>Balance=3500+1800</td>
</tr>
<tr>
<td>11:28</td>
<td></td>
<td>5300</td>
<td>Write new Balance (5300)</td>
</tr>
<tr>
<td>11:29</td>
<td></td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Figure 5-3: Dirty Read
5.5.2.3. Incorrect Summary

Let us consider another scenario where Hilary wants to transfer amount of USD 500 to her sister Evelyn’s account in the same branch. After deducting USD 500, Hilary’s balance will be USD 1000.

Evelyn’s account balance was USD 1500 before and now will become USD 2000 with the addition of USD 500.

Almost at the same time, the bank branch manager starts another transaction to calculate the total sum available in bank through customer deposits.

This program calculates the sum by reading Hilary’s balance amount as USD 1500 (before deduction of transfer amount) and Evelyn’s balance as USD 2000 (after addition of transfer amount). This program concludes that the sum is USD 3500 (sum of Hilary’s balance amount and Evelyn’s balance amount) but actually it is only USD 3000. This problem is known as incorrect summary. Snapshot of main memory for these transactions are shown in Figure 5-4.

<table>
<thead>
<tr>
<th>Time</th>
<th>Hilary’s Transfer</th>
<th>Balance</th>
<th>Summary Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:22</td>
<td>Read Hilary’s Balance (1500)</td>
<td>1500</td>
<td>Sum = 0</td>
</tr>
<tr>
<td>12:23</td>
<td>Balance=1500-500</td>
<td></td>
<td>Read Hilary’s Balance (1500)</td>
</tr>
<tr>
<td>12:24</td>
<td>Write new Balance (1000)</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>12:25</td>
<td></td>
<td></td>
<td>Sum=Sum + Balance (1500)</td>
</tr>
<tr>
<td>12:26</td>
<td>Read Evelyn’s Balance (1500)</td>
<td>1500</td>
<td></td>
</tr>
<tr>
<td>12:27</td>
<td>Balance=1500 + 500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12:29</td>
<td>Commit</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>12:30</td>
<td></td>
<td></td>
<td>Read Evelyn’s Balance (2000)</td>
</tr>
<tr>
<td>12:31</td>
<td></td>
<td></td>
<td>Sum=Sum + Balance (3500)</td>
</tr>
<tr>
<td>12:32</td>
<td></td>
<td>3500</td>
<td>Write Sum (3500)</td>
</tr>
<tr>
<td>12:33</td>
<td></td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Figure 5-4: Incorrect Summary

5.5.2.4. Phantom Record

Let us consider the snapshot of two different transactions which are running simultaneously almost at the same time as shown in Figure 5-5.

One transaction is counting the number of accounts held by the bank. Another transaction is creating new accounts. Though two accounts are created and committed before completion of “Total Accounts” transaction, these accounts (Simon’s account and Mike’s account) are missed by “Total accounts” transaction for counting.
These newly inserted rows appear as phantom to the “Total Accounts” transaction, inconsistently appearing and disappearing. This is called Phantom record because Phantom is considered as an invisible ghost as in the case of newly inserted rows.

<table>
<thead>
<tr>
<th>Time</th>
<th>Create account</th>
<th>Total Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>13:22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:25</td>
<td>Create account for Simon with a deposit of 500</td>
<td>Read the total number of accounts in the bank as total</td>
</tr>
<tr>
<td>13:26</td>
<td>Create account for Mike with a deposit of 1000</td>
<td></td>
</tr>
<tr>
<td>13:27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:28</td>
<td>Commit</td>
<td></td>
</tr>
<tr>
<td>13:29</td>
<td></td>
<td>Write Total</td>
</tr>
<tr>
<td>13:30</td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

**Figure 5-5: Phantom Record**

If we observe these problems closely, all these problems are because of interleaving of the transactions. The solution to overcome these problems would be to make every transaction follow each other. This is called as serialization. Serialization of transactions can be achieved by setting following rules on transactions.

1. If any row is being modified, then do not allow any other transaction either to read or update/delete that row until the first transaction completes.
2. If a transaction is reading a particular row, prevent other transactions from making any changes to that row until the first transaction completes.
3. If a transaction is reading some data, do not allow any other transaction to insert new rows into the same table until the first transaction completes. This will avoid problems like Phantom records.

**Serialization** can be achieved using Locking or Time Stamping techniques.
5.6. Locks

Locking is a technique to have a controlled access to the resources like a database, *tablespace*\(^{44}\), table, rows and columns. While these resources are put under lock by some transaction, other transactions have very restricted or no access to these resources, depending on the locking mode.

Locking is one of the most widely used techniques in commercial RDBMS products to achieve consistency while supporting concurrency of transactions.

Basically resources can be locked either in Shared (S) mode for Read purpose or in Exclusive (X) mode for Update, Delete or Insert purpose.

5.6.1. Shared Lock (S)

This locking technique allows a higher transaction concurrency. When a particular resource like a table or a row is locked in the *shared* mode by one transaction, all other transactions can perform the *read* operation on the locked resource, but no updates or modifications are possible by other transactions. Usually SELECT operation takes the S lock on resources.

5.6.2. Exclusive Lock (X)

This is the most restrictive lock. Once a transaction puts the X lock on a particular resource, no other transaction can put any kind of lock on this resource. This resource is exclusively reserved for the first transaction and no other transaction can use it for read or write operation. Hence this X lock allows the least concurrency.

Usually INSERT/ UPDATE/ DELETE operations put the X lock on resources before writing/ modifying/ deleting operations.

The Figure 5-6 explains the compatibility of these locks. This figure can be interpreted as:

If transaction T1 locks the resource A (database/tablespace/table/row/field) in shared(S) mode, another transaction T2 can also lock the same resource A, in shared(S) mode.

If transaction T1 locks the resource A (database/tablespace/table/row/field) in shared (S) mode, another transaction T2 can not lock the same resource A, in exclusive (X) mode until T1 releases its S lock on the resource A.

\(^{44}\) *Tablespace*: The logical part of the database which represents collection of the structures like tables, etc created by various users.
If transaction T1 locks resource A in X mode no other transaction can lock resource A in any other (S or X) mode until T1 releases its X lock on the resource A.

**Figure 5-6: Share - Exclusive Lock Matrix**

In Figure 5-6 symbol ✗ represents incompatibility of the lock and ✓ symbol represents compatibility of the lock.

### 5.7. Granularity of Locking

Granularity of locking refers to the granular level at which a resource can be locked. Consider for example a database. A database is made up of multiple tablespaces. Each tablespace hosts multiple tables. Within a table there are multiple rows and fields as shown in Figure 5-7. It is possible to lock a

- Database
- Tablespace
- Table
- Row
- Field

If RDBMS application is capable of locking a field of a table explicitly, then the granularity of locking is at field level. If it can lock only up to the row level, the locking granularity of that RDBMS product is row level. Thus, the higher the granularity of locking, the higher will be the concurrency.

In above case database, tablespace, table and row are all the ancestors of the field. Similarly database, tablespace are ancestors of the table.

Tablespace, table, row and fields are descendants of database. In the same way rows and fields are descendants of table.
S and X locks alone cannot achieve complete concurrency. This is illustrated below.

Let us consider the following scenario and analyze the concurrency that can be possible.

Assume in a banking application transaction called BalanceUpdate has locked Row R7 of table ACC_DETAILS in X mode for updating the account balance. Because of this X mode lock, no other transactions can acquire either S or X lock on row R7 or any of its fields.
Let us assume following scenarios:

- A transaction called BalanceEnquiry requires a lock on first row of table ACCDETAILS in S mode
- A transaction called SummaryReport requires a lock on complete table ACCDETAILS in S mode

In ideal condition:

- System should allow to lock first row of ACCDETAILS table for transaction BalanceEnquiry
- System should prevent transaction SummaryReport from acquiring a lock on table ACCDETAILS

Transaction SummaryReport should be prevented from acquiring a lock on table ACCDETAILS because row R7 of table ACCDETAILS is already locked by transaction BalanceUpdate in X mode.

If transaction SummaryReport is allowed to acquire S lock on table ACCDETAILS, we may encounter the dirty read problem. On same lines no S or X locks are allowed by any other transaction on the tablespace TS_CUSTDETAILS or database DB_BANKDETAILS because row R7 is part of TS_CUSTDETAILS and database DB_BANKDETAILS.

In other words, although row R7 of table ACCDETAILS was locked explicitly in X mode, the table ACCDETAILS, the tablespace TS_CUSTDETAILS and the entire database DB_BANKDETAILS, was locked implicitly in the X mode to avoid any parents of R7 being locked by some other transactions.

This implicit locking of complete database now avoids lock on row R1 of Table ACCDETAILS by transaction BalanceEnquiry. This is serious threat to concurrency of the transactions.

Let us look at the solution for this problem in next section.

### 5.8. Intent Locking

In Intent locking only the intention of locking is expressed at the ancestor node of the required resource and the resource at the lower level is locked explicitly only when required.

Consider the example discussed in section 5.7. In the earlier case, it was required to lock row R7 of table ACCDETAILS in X mode explicitly but all its ancestors were implicitly locked in the same mode (Refer Figure 5-7). This has reduced the concurrency considerably.
To overcome this concurrency issue it is necessary for a transaction to express only the intension of locking the database DB_BANK_DETAILS, the tablespace TS_CUST_DETAILS and the table ACC_DETAILS in the X mode and in turn lock the row R7 explicitly in X mode. This concept is called as intent locking.

Some other transactions still can express their intension of exclusive or shared locking on database DB_BANK_DETAILS or tablespace TS_CUST_DETAILS or table ACC_DETAILS and explicitly lock any other row other than Row R7, either in X or S mode.

This intent locking mechanism not only increases concurrency but also stops the implicit locking of ancestral resources.

Hence intent locking is called as Parent-Child locking. You express your intension of locking at parent level and lock child resource in explicit mode.

Intent locking is classified as Intent Shared (IS) locking and Intent Exclusive (IX) locking.

5.8.1. Intent Share (IS)

This lock has the intention to share the requested node. This also allows the requester to explicitly lock the descendants of this node in S or IS mode.

Example: Transaction SummaryReport explained in section 5.7 can lock entire database DB_BANK_DETAILS and tablespace TS_CUST_DETAILS in IS mode and ACC_DETAILS table explicitly in S mode.

5.8.2. Intent Exclusive (IX)

This lock has the intention to have exclusive access to the requested node and allows the requester to explicitly lock the descendants in IX or X modes.

Example: Transaction BalanceUpdate explained in section 5.7 can lock database DB_BANK_DETAILS, the tablespace TS_CUST_DETAILS and the table ACC_DETAILS in IX mode and lock the row R7 explicitly in X mode.

5.8.3. Shared Intent Exclusive (SIX)

The Combination of Shared and Intent exclusive lock is referred to as Shared Intent Exclusive Lock or SIX Lock.

A share and intent exclusive lock (or SIX lock, pronounced as the separate letters S I X rather than like the number six) indicates an S lock at the current level plus an intention to insert, delete, update data at a lower level of granularity. Think of a SIX lock as an S lock plus an IX lock as shown in
Figure 5-8. Only one transaction can be granted a SIX lock on a table at a time.

A SIX lock on a table indicates an intention to read all of the rows in the table and to delete/update/insert to a few. The S lock in the SIX lock at the table level covers all of the rows. Rows that are updated will obtain X locks, but only after IX intention locks have been obtained on the pages\footnote{Page: It is part of a table. Usually in one page multiple rows are stored} that contain them.

A SIX lock is stronger than a S lock or an IX lock. When a transaction obtains a SIX lock on a table, only that transaction will be able to modify data in the table. In this respect, a SIX lock slightly resembles an X lock. With a SIX lock, however, other transactions that want to read some of the data (read data at the row or page level and obtain an IS lock on the table) are allowed to proceed, so concurrency is better than with an X lock. Lock mode compatibility will be described in greater detail later in this module.

If other transactions obtain S lock on row of a table or S lock on page of a table the SIX transactions wants to modify, the SIX transaction must wait until the S locks are released before it can modify the data.

Other transactions that want to read all of the data (obtain an S lock on the table) or that want to write to any portion of the data are not allowed to proceed until the SIX lock is released.
A SIX lock is also called a share sub-exclusive lock.

A complete compatibility matrix of these locks is shown in Figure 5-9. Note that in Figure 5-9 symbol ✗ represents incompatibility of the lock and ✓ symbol represents compatibility of the lock.

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>A</th>
<th>X</th>
<th>S</th>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>S</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>IS</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IX</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>SIX</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

Figure 5-9: Complete Lock Matrix

This lock matrix can be interpreted as:

If resource A (tablespace/table/row) is locked in X mode by transaction T1, no other transactions can lock resource A in any mode.

If resource A is locked in S mode by transaction T1, another transaction say T2 can lock the same resource A in S or IS mode but it cannot lock in IX or SIX or X mode.

If resource A is locked in IS mode by transaction T1, another transaction say T2 can lock the same resource A in S or IS or IX or SIX mode but it cannot lock in X mode.

If resource A is locked in IX mode by transaction T1, another transaction say T2 can lock the same resource A in IS or IX mode but it cannot lock in S or X or IX or SIX mode.

If resource A is locked in SIX mode by transaction T1, another transaction say T2 can lock the same resource A in IS mode but it cannot lock in S or X or IX or SIX mode. SIX lock is the combination of S and IX. Hence SIX lock is compatible with that lock which has the common compatibility with S and IX locks. Since S and IX are together compatible with IS lock, SIX lock is compatible with IS lock only.
The biggest problem with locking technique is that it may lead to **Deadlock**.

### 5.8.4. Case study for Intent Locks

**Objective:** To study about the compatibility of locks.

**Assumptions:**

A database **db** has two tables (files) **f1** and **f2**.

File **f1** has pages **p11**, **p12**, and **p13**.

File **f2** has pages **p21**, **p22** and **p23**.

Page **p11** has 2 records, **r111** and **r112**.

Page **p12** has 2 records, **r121** and **r122**.

Page **p13** has 2 records, **r131** and **r132**.

Page **p21** has 2 records, **r211** and **r212**.

Page **p22** has 2 records, **r221** and **r222**.

Page **p23** has 2 records, **r231** and **r232**.

**Consider the following situation:**

Transaction **T1** wants to update record **r111** and record **r211**.

Transaction **T2** wants to update all records on page **p12**.

Transaction **T3** wants to read record **r112** and the entire file **f2**.

Assume that transaction **T4** and **T5** starts only after all the other transactions have committed.

Transaction **T4** wants to modify **r111** and transaction **T5** wants to read record **r112**.

**Problem statement:** Specify

- The locks which will be acquired by the transactions
- The order in which the locks will be acquired by the transactions
- The order in which the locks will be released by the transactions

**Solution:**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>IX(db)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IX(f1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IX(db)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IX(f1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IS(db)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IS(f1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IS(p11)</td>
</tr>
<tr>
<td>IX(p11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(r111)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IX(f1)</td>
</tr>
<tr>
<td>T4</td>
<td>T5</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>SX(db)</td>
<td>IS(db)</td>
<td></td>
</tr>
<tr>
<td>SX(f1)</td>
<td>IS(f1)</td>
<td></td>
</tr>
<tr>
<td>IX(p11)</td>
<td>IS(p11)</td>
<td></td>
</tr>
<tr>
<td>X(r111)</td>
<td>IS(r112)</td>
<td></td>
</tr>
<tr>
<td>Unlock(r111)</td>
<td>Unlock(r112)</td>
<td></td>
</tr>
<tr>
<td>Unlock(p11)</td>
<td>Unlock(p11)</td>
<td></td>
</tr>
<tr>
<td>Unlock(f1)</td>
<td>Unlock(f1)</td>
<td></td>
</tr>
<tr>
<td>Unlock(db)</td>
<td>Unlock(db)</td>
<td></td>
</tr>
</tbody>
</table>

Assumption: Transaction T4 and transaction T5 start after the transactions T1, T2 and T3 have committed and released the locks.

Learning's from the above case study:
- Locks are acquired from the root to the place (node) one wants to lock. (top to bottom)
- S or X mode locks are applied only at very fine granularity (only on the specific node that the user wishes to read or update)
- Locks are released in bottom to top fashion
- Check for the compatibility of the locks in cases where a transaction already holds a lock on the node and another transaction wants to acquire a lock on the same node
5.9. Deadlock

Deadlock is a situation where one transaction is waiting for another transaction to release the resource it needs, and vice versa. Each transaction will be waiting forever for the other to release the resource. This is shown in the Figure 5-10:

<table>
<thead>
<tr>
<th>Time</th>
<th>Transaction</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:22</td>
<td>Lock ACC_DETAILS</td>
<td>Lock LOAN_DETAILS</td>
</tr>
<tr>
<td>10:23</td>
<td>Update ACC_DETAILS</td>
<td>Update LOAN_DETAILS</td>
</tr>
<tr>
<td>10:24</td>
<td>Try for lock on LOAN DETAILS</td>
<td>Try for lock on ACC DETAILS</td>
</tr>
<tr>
<td>10:25</td>
<td>Wait for lock</td>
<td>Wait for lock</td>
</tr>
<tr>
<td>10:26</td>
<td>Wait for lock</td>
<td>Wait for lock</td>
</tr>
<tr>
<td>10:27</td>
<td>Wait for lock</td>
<td>Wait for lock</td>
</tr>
<tr>
<td>10:28</td>
<td>Wait for lock</td>
<td>Wait for lock</td>
</tr>
</tbody>
</table>

Figure 5-10: Deadlock

In the above diagram transaction BalanceUpdate locked the table ACC_DETAILS in X mode at time 10:22 and waiting to acquire lock on table LOAN_DETAILS for some updation. But transaction LoanUpdate already has the X lock on table LOAN_DETAILS and waiting for table ACC_DETAILS which is locked by transaction BalanceUpdate. These two transactions will be waiting infinitely for each other to release the locked resources. This is known as Deadlock.

If a Deadlock occurs, one of the participating transactions must be rolled back to allow the other to proceed. There are various methods to choose which transaction to roll back when a deadlock is detected. Usually rollback action is decided on:

- How long the transactions have been running
- Data already updated by the transaction
- Data that remains to be updated by the transaction

There are schemes available for preventing deadlock. Most of the RDBMS products allow deadlocks to occur and resolve them, when they are detected.

5.10. Timestamping

Another concurrency management technique is Timestamping. Every resource in database will be associated with last successful read and last successful write
timestamp (time of occurrence up to milliseconds Ex: 12th December 2004 11:22:33.345).

Let us consider:

- RDBMS author by name Hanu is modifying this course material as one transaction
- Trainees reading this course material as another transaction

If other RDBMS author, say Seema, is also modifying the same course material at the same time, it leads to Lost update and Phantom record conditions.

If Hanu starts modifying while trainees are studying this material, it leads to dirty read or incorrect summary problems.

To avoid these problems we can follow these two rules:

Hanu can start the course modification transaction only if:
- Course material is successfully modified before starting this transaction
- No Trainees are currently reading this course material

Similarly Trainees can start reading course material transaction only if:
- It was successfully updated before they start reading it

Let us consider an example of database DB_BANK_DETAILS (Refer Figure 5-7). In table ACCDETAILS, a particular row is read successfully by transaction BalanceEnquiry at 12:11:45.345 of 12th December 1945. This will be the last read timestamp of this row. If any other transaction reads this row after this time, that particular time will be the last read timestamp of the row.

Similarly every row will have last updated timestamp. If transaction BalanceUpdate updates the row R1 at 13:32:22.345 of 12th December 1945, this will be recorded as last updated timestamp of the row R1.

A transaction can read only rows or columns that have been updated by an older transaction if not, transaction is rolled back.

Let us assume that Row R7 of the table ACCDETAILS was successfully updated at 09:24:22.46 Hrs of 15th August 1947 by some transaction.

Any transactions started after 09:24:22.46 Hrs of 15th August 1947 can read this row.

Transactions started before 09:24:22.46 Hrs of 15th August 1947 need to be rolled back and start afresh to read this data.
In general for read, the condition can be defined as TS > TU where TS is the start time of transaction and TU is the last successful update timestamp of the resource.

A transaction can update only rows or columns that have been read and updated by an older transaction else this transaction is rolled back.

Similarly any transaction can update row R7 only if it is started after the last successful update and the last successful read.

Assume a transaction started at 10:24:23.49 Hrs of 15\textsuperscript{th} August 1947 and wishes to update row R7 at 10:29:11.34 Hrs of 15\textsuperscript{th} August 1947.

It is possible to update this row only if the row R7 was successfully updated and read before 10:24:23.49 Hrs of 15\textsuperscript{th} August 1947.

Any transaction started after 10:24:23.49 Hrs of 15\textsuperscript{th} August 1947 can not change the value of this row.

**Generic rule for updating data is TS > TU and TS > TR.** Where TS is transaction start timestamp, TU is the last successful updated timestamp and TR is the last successful read time.

The biggest advantage of timestamping is it leads to no dead lock condition as no resources are locked.

Timestamping technique leads to large number of rollbacks. Due to this reason timestamping technique is not implemented as the concurrency control mechanism in most of the commercial RDBMS applications.

> **Note:** Almost all the commercial RDBMS packages use a locking technique as the concurrency controlling mechanism while maintaining the consistency in the system.

### 5.11. Security

Security is one of the best implemented strategies in RDBMS.

Security is implemented in RDBMS packages using:

1. **USERID** and **PASSWORD** to restrict the users from acquiring an un-authorized access
2. Grant and Revoke statements (Data Control Language) to provide restricted access control to resources like Tables
3. Database views to restrict access to sensitive data
4. *Encryption* of data to avoid un-authorized access

## 5.12. Recovery

A database might be left in an inconsistent state by:

- An Application error
- Power failure
- O/S or Database failure
- Network failure
- Hardware or Media failures

If the database is in an inconsistent state, it is necessary to restore it to a consistent state. Recovery process can be achieved either using log files or backups of the database.

The simplest backup technique is ‘Dumping’. The entire content of the database are backed up on to secondary devices like tapes on a regular basis. This backing up operation must be performed when the state of the database is consistent. Therefore no transactions which modify the database can be running during this backup process.

This dumping process can take a long time to perform and one may not be able to stop transactions for a longer time in the production environment. Hence it cannot be performed as often as one would like to. This type of back-up is called cold back up and is usually done on a periodic basis like once a week or once a month at night when transactions in system are at their minimal threshold. These tapes can then be used in case of complete hard disk failures. This is showed in the Figure 5-11.

---

46 **Encryption**: The process of manipulation of data to prevent accurate interpretation by all but those for whom the data is intended.
If the database back up is done while transactions are running, it is called as Hot backups. Usually hot backups are incremental in nature. This means only modified data since the last backups are captured. Usually it takes less time and is done on a daily basis.

The hot and cold backups are useful only in the case of media or hard disk failure. This back up cannot be used for

- Un-planned power shutdown
- Sudden breakdown in O/S or database
- Memory failure

These kinds of failures are called as instance failures. Instance failures can be handled by making use of transactional log or redo log files. These are further explained in the following sections.

### 5.13. Transaction Log

Transaction log or the journal log or redo-log is a physical file. Usually the Transaction ID, the time stamp of the transaction, the old value and the new values of the data are stored in transaction logs file. Therefore the RDBMS is aware of the state of the database i.e. before and after image of data after each transaction. Every database is returned to a consistent state and the log may be truncated to remove committed transactions.

Normally there are two techniques used to maintain the log files.

#### 5.13.1. Deferred update

Deferred update, or NO UNDO/REDO, is an algorithm to support transaction failures owing to O/S, application, power, memory and machine failures.

While a transaction runs, not updates/alterations made by that transaction are recorded in the database but captured only in the log files.

On commit, data changes are applied to the database using the log files. This process is called as "Re-doing".

On rollback, data changes which are captured in the log files are discarded and no changes are made to the database.

On system restart, due to any of the above mentioned reasons if transaction fails and it is not committed, contents of the log files are discarded and the transaction will be restarted. If it is committed before crashing then after restart, the log file contents are applied to the database.

Sequences of deferred update are explained in Figure 5-12 and Figure 5-13.
From Figure 5-12 it is evident that when transaction updates the field F1 to 23 and field F2 to 45 in main memory log file will have old value and new value of the field. Database disk file still holds the old values. Contents of database are modified using log file only after transaction commits. The process or re-doing the transaction from the log is sometimes referred as ‘Rollforward’. Disadvantage of deferred update technique is increased time of recovery in case of system failure.
5.13.2. **Immediate Update**

Immediate update, or UNDO/REDO, is another algorithm to support transaction failure owing to OS, application, power, memory or machine failure.

While a transaction runs, updates/alterations made by that transaction can be written to the database directly. However, the original and the new data being written must both be stored in the log **BEFORE** writing it to the database.

On commit, all the changes to the database are made permanent and log contents are discarded.

On rollback, using the log entries, old values are restored. All the changes which that transaction has made to the database disk are discarded. This process is called as “Un-doing”.

---

**Figure 5-13: Sequences of Deferred Update**

- **START**
- Update Record in Memory
- Update in Logs
- **Has System crashed?**
  - NO
  - **Is transaction committed?**
    - YES
      - **Do you find commit in log?**
        - YES
          - **Make changes permanent in database using log**
        - **NO**
          - Discard Log data
    - **NO**
      - **Restart System**
  - **NO**
    - Discard Log data
- **STOP**
Database changes are made permanent once the system restarts, after the crash for committed transactions. The original values are restored using the log files for uncommitted transactions.

Transaction snapshot is shown in Figure 5-14 and sequences of immediate update process are shown in Figure 5-15.

<table>
<thead>
<tr>
<th>Time</th>
<th>Transaction</th>
<th>Disk Before</th>
<th>Disk After</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:22</td>
<td>Start</td>
<td>6</td>
<td>6</td>
<td>Start Timestamp</td>
</tr>
<tr>
<td>10:23</td>
<td>Read column F1</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>10:24</td>
<td>Update F1 to 23</td>
<td>6</td>
<td>23</td>
<td>(6,23)</td>
</tr>
<tr>
<td>10:25</td>
<td>Read Column F2</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>10:25</td>
<td>Update F2 to 45</td>
<td>12</td>
<td>45</td>
<td>(12,45)</td>
</tr>
<tr>
<td>10:25</td>
<td>Commit</td>
<td>F1=23, F2=45</td>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-14: Immediate Update

From Figure 5-14 is evident that when transaction updates the field F1 to 23 and field F2 to 45 in main memory log file will have old value and new value of the field. Simultaneously database disk file also modified to reflect the new values even before transaction commits. For any reason if transaction fails to commit, contents of database disk files values are restored to old values using log file. The process of undoing changes using the log files is frequently referred to as **rollback**. Disadvantage of immediate update technique is frequent I/O operations while the transaction is active.
Figure 5-15: Sequences in Immediate Update
5.13.3. **Check-Points**

Usually in commercial RDBMS applications neither the deferred updates nor the immediate updates are used because of their disadvantages. In these commercial RDBMS applications, databases are updated at fixed intervals of time; say every 2 minutes, irrespective of the transaction commit/uncommit state. Updating the database at fixed intervals of time is called as check-pointing.

At the check point time, the contents of the log files are applied to the database. Transactions may be committed or non-committed at the check point. Later if the transaction rolls back, the database is restored to the original state using the log files. As already explained, this process is called as “Un-doing”. If the transaction commits, changes are made permanent, again using the log files. This process is called as “Re-doing”. Hence check point based updates use both the Rollforward and the Rollback mechanism. To some extent this technique speeds up the recovery mechanism during instance failures.

For example consider the snapshot of the database shown in Figure 5-16.

![Checkpoint Scenario](image)
Let us analyze the situation on a system restart, after an unfortunate crash.

1. Transaction T1 committed before check point and also wrote to the database hence no changes are required in the database.
2. Transaction T2 committed before system failure but partially wrote to the database at the check point. After restart, other parts of T2 should be written to the database using the log files.
3. Transaction T3 began after the checkpoint hence contents were not written to the database but successfully completed before crash. Complete transaction needs to be written to the database using the log file.
4. Transaction T4 began before the checkpoint hence part of T4 was written to the database. The unfortunate crash happened before T4 committed. Hence it is required to undo the changes to the database using the log files and restore it to the old values.
5. The contents of the transaction T5 in the log files needs to be discarded and the transaction needs to be re-started as this transaction started after checkpoint and hence no traces of this transaction exist in the database.

Recovery scenario is explained in the Figure 5-17:
Examples:

1. Recovery using deferred update in a single-user environment

Consider the read and write operations of two transactions T1 and T2 given below:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_item(A)</td>
<td>read_item(B)</td>
</tr>
<tr>
<td>read_item(D)</td>
<td>write_item(B)</td>
</tr>
<tr>
<td>write_item(D)</td>
<td>read_item(D)</td>
</tr>
<tr>
<td>write_item(D)</td>
<td>write_item(D)</td>
</tr>
</tbody>
</table>

The system log at the time of crash is as given below:

<start T1>
<write_item, T1, D, 20>
<commit T1>
<start T2>
<write_item, T2, B, 10>
<write_item, T2, D, 25>  -----------------System crash

Solution:  Transaction T1 commits before the system crash. The operations of transaction T1 are therefore redone (redone means contents of the log files are applied to the data file). The entries in the log corresponding to transaction T2 are ignored by the system because T2 is not committed.

2. Recovery using check points (concurrent transactions considered)

Consider the read and write operations of transactions T1, T2, T3 and T4 given below:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_item(A)</td>
<td>read_item(C)</td>
<td>read_item(A)</td>
<td>read_item(C)</td>
</tr>
<tr>
<td>read_item(B)</td>
<td>write_item(C)</td>
<td>write_item(A)</td>
<td>write_item(C)</td>
</tr>
<tr>
<td>write_item(B)</td>
<td>read_item(B)</td>
<td>read_item(E)</td>
<td>read_item(A)</td>
</tr>
<tr>
<td></td>
<td>write_item(B)</td>
<td>write_item(E)</td>
<td>write_item(A)</td>
</tr>
</tbody>
</table>

The system log at the time of crash is as given below:

<start T1>
Solution:

Transaction T1 committed before the checkpoint. Therefore no operation is performed on account of it.

Transaction T4 is redone because its commit point is after the last system checkpoint.

Transaction T2 and T3 are ignored because they did not reach their commit points.

5.14. Summary

- All transactions should be:
  - Atomic
  - Consistent
  - Isolated
  - Durable

- OLTP applications should ensure:
  - Integrity
  - Concurrency
  - Security
  - Recovery

- Integrity of the RDBMS application can be maintained using:
  - Entity Integrity
  - Referential Integrity
  - Domain Integrity

- While allowing Concurrency one may face problems in implementing consistency. Following are the four major problems encountered:
  - Lost Updates
  - Dirty Read
  - Incorrect Summary
- Phantom records

- Consistency can be implemented using serialization techniques like:
  - Locking
  - Time-stamping

- Locking technique leads to the dead lock problem

- Time stamping technique leads to many rollback problem

- Security is implemented in RDBMS using:
  - User ID / Password
  - Grant and Revoke commands
  - Views

- Two types of recovery mechanism can be implemented in the RDBMS application:
  - Media failures using back-up strategy
  - Instance recovery using transaction log files

- Two types of backups are possible:
  - Cold backup
  - Hot backup

- Three types of updates to the database are possible, using the transaction log files:
  - Immediate update
  - Deferred update
  - Check-point based updates
6. On Line Analytical Processing (OLAP)

Data is the key asset of any organization or enterprise. Operational activities of an organization include day-to-day business processes necessary to run it. Systems that support such processes are called the On Line Transaction Processing (OLTP) systems. Operational data are highly structured data that is continuously generated and stored in what is typically called as operational or transactional or OLTP databases.

An organization’s success also depends on its ability to analyze data and to make intelligent decisions that would potentially affect its future. Systems that facilitate such analysis are called On Line Analytical Processing (OLAP) systems.

An OLTP application rarely requires historical data. An OLAP application requires historical data because an analysis is generally based on a substantial amount of historical data to enable trend analysis and future predictions.

An OLTP transaction is characterized by several users creating, updating or retrieving individual records whereas OLAP application is characterized by higher level views of the data.

Thus the focus of OLTP and OLAP are fundamentally different. The following section gives the difference between OLTP and OLAP.

6.1. Difference between OLTP and OLAP

<table>
<thead>
<tr>
<th></th>
<th>OLTP</th>
<th>OLAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>On Line Transaction Processing</td>
<td>On Line Analytical Processing</td>
</tr>
<tr>
<td>Data</td>
<td>Dynamic (day to day transaction / operational data)</td>
<td>Static (historical data)</td>
</tr>
<tr>
<td>Data Atomicity</td>
<td>Data is stored at microscopic level</td>
<td>Data is aggregated or summarized and stored at the higher level</td>
</tr>
<tr>
<td>Normalization</td>
<td>Normalized Databases to facilitate insertion, deletion and updation</td>
<td>De-normalized Databases to facilitate queries and analysis</td>
</tr>
<tr>
<td>History</td>
<td>Old data is purged or archived</td>
<td>Historical data stored to enable trend analysis and future predictions</td>
</tr>
<tr>
<td>Queries</td>
<td>Simple queries and updates Queries use small amounts of data (one record or a few)</td>
<td>Complex queries Queries use large amounts of data</td>
</tr>
</tbody>
</table>
A practical solution to enable analytical processes is to implement a data warehouse.

### 6.2. Data Warehouse

A data warehouse is a repository which stores integrated information for efficient querying and analysis. Data warehouse has data collected from multiple, disparate sources of an organization. It is the basis for decision support and data analysis systems.

#### 6.2.1. Why data warehouse is needed?

Analysis requires millions of records of data which are historical in nature. Data is collected from heterogeneous sources (e.g. RDBMS, flat files, etc.) Need to make quick and effective strategic decisions.

In essence, it is a copy of the organization’s operational data adequately modified to support the needs of analytical processes and stored outside the operational database.

#### 6.2.2. Characteristics of Data Warehouse:

According to Bill Inmon, known as the father of Data Warehousing, a data warehouse is a subject oriented, integrated, time-variant, nonvolatile collection of data in support of management decisions.
- **Subject-oriented**: means that all data pertinent to a subject/business area are collected and stored as a single unit.

- **Integrated**: means that data from multiple disparate sources are transformed and stored in a globally accepted fashion.

- **Static/non-volatile**: means data once entered into the warehouse does not change frequently. It is periodically updated if required.

- **Time variant**: Data warehouse maintains historical data which are used to analyse the business or market trends and facilitate future predictions.

**Figure 6-1: Data warehouse architecture**

<table>
<thead>
<tr>
<th>Data Warehouse Server (Tier 1)</th>
<th>OLAP Servers (Tier 2)</th>
<th>Clients (Tier 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>serve</code></td>
<td><code>serve</code></td>
<td><code>serve</code></td>
</tr>
<tr>
<td><code>serve</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.2.3. Data Warehousing Terminology

**Data sources**: An organization has many functional units with their own data. Data from all such sources have to be consolidated and put into a consistent form that would reflect the business of the organization as a whole. These sources of data for a data warehouse are known as data sources or operational data sources.

**Metadata**: Metadata is the information about the data. This is the layer of the data warehouse which stores the information like the source data, transformed data, date and time of data extraction, target databases, data and time of data loading, etc.

**Measure attributes**: A numerical value that can be summarized or can be aggregated upon. **Example**: Consider an inventory application. Assume that the inventory store
sells twenty products in one day, each for 5 dollars. Thus it generates 100 dollars in total sales for the day. Therefore, sales dollars is one measure. The store owner might want to know how many customers they had that day. Did 5 customers buy 4 products each, or did one customer buy twenty products. Thus, customer count is another measure.

**Dimension attributes:** Dimensions can be defined as the perspectives used for looking at the data. Dimensions are the answer to the question “How do you want to see your data?” Some examples of dimensions are:

- Product
- Time
- Location
- Customer Age
- Customer Income

There is almost always a time dimension on anything which is being analyzed. Considering the example given for measure attributes, sales can be analyzed by day, or by quarter, or by year. Sales can also be analyzed by category or by product. There is almost always a time dimension and product and geographic dimensions are very common as well.

Data that can be modeled as dimension attributes and measure attributes are called multidimensional data.

### 6.2.4. Data Collection for Data Warehouse Applications

**Extraction, transformation and loading (ETL):** ETL is the most important step in Data Warehousing.

**Definition of ETL:** Extract, transform and load process is described as the process of selecting, migrating, transforming, cleansing and converting mapped data from the operational environment to data warehouse environment.

Data needs to be taken from various disparate sources to the data preparation area. This process is known as data extraction. This data preparation area, also known as data staging area consists of relational tables. Data from various heterogeneous sources are transformed into a common format and put into relational tables of data preparation area, so that it can be loaded into the data warehouse database. This process is known as loading. The data is loaded into the fact table and dimension tables in the data warehouse database. Refer to Figure 6-2.
6.2.5. Storing of data in Data warehouse

**Dimensional Modeling:** The dimensional modeling is also known as star schema because in dimensional modeling there is a large central fact table with many dimension tables surrounding it.

**Fact Tables:** Each data warehouse or data mart includes one or more fact tables. A fact table is central to a star or snowflake schema, and captures the data that measures the organization’s business operations. Fact tables usually contain large numbers of rows.

A key characteristic of a fact table is that it contains numerical data (facts) that can be summarized to provide information about the history of the operation of the organization. Each fact table also includes a multipart index that contains as foreign keys, the primary keys of related dimension tables, which contain the attributes of the fact records. Fact tables should not contain descriptive information or any data other than the numerical measurement fields and the index fields that relate the facts to corresponding entries in the dimension tables.

**Dimension Tables:** Dimension tables contain attributes that describe fact records in the fact table. Some of these attributes provide descriptive information; others are used to specify how fact table data should be summarized to provide useful information to the analyst. Dimension tables contain hierarchies of attributes that aid in summarization. For example, a dimension containing product information would often contain a hierarchy that separates products into categories, with each of these categories further subdivided into manufacturer.

**Cube:** The OLAP tools allows you to turn data stored in relational databases into meaningful, easy to navigate business information by creating data cube. The dimensions of a cube represent distinct categories for analyzing business data.
Categories such as time, geography or product line breakdowns are typical cube dimensions.

**Dimension hierarchies:** Refer to Figure 6-3. The product dimension contains individual products. Products are classified into categories, and further classified according to manufacturer. The hierarchy for the dimension is stored in the dimension table.

![Figure 6-3: Dimension Hierarchies](image)

**Available Schemas for dimensional modeling:**

- Star schema
- Snowflake Schema

**Star Schema:** It is the simplest data warehouse schema. It resembles a star. The center of the star consists of one or more fact tables and the points radiating from the center are the dimension tables. Refer to Figure 6-4.

![Star Schema](image)

**Snowflake Schema:** It is a complex data warehouse schema. The snowflake schema has a single, central fact table surrounded by normalized dimension hierarchies. Each dimension level is represented in a table. Refer to Figure 6-5.
Disadvantages of Snowflake Schema:
- It increases the number of dimension tables
- It requires more foreign key joins

6.2.6. Reporting of a Data warehouse application

A data mart is a focused subset of a data warehouse that deals with a single area of data and is organized for quick analysis. It can be a small data warehouse itself.

6.2.6.1. Advantages of Data Marts:
- It focuses on presentation rather than the organization of data
- It facilitates data reporting
- It provides meaningful reports to the users pertaining to their business area thereby allowing them to view and concentrate only on the data that is related to their business area
  Example: providing sales data to the sales department, providing financial data to the financial department
- It makes the data design simpler and easier. It breaks the whole design into several smaller sub units which is beneficial to the customers and the development team. It is also easier to maintain
- Reporting of data becomes faster and more efficient because reporting is generally done at the sub unit level and data marts assist in faster retrieval compared to querying the entire data warehouse
- It helps in incrementally building up the enterprise data warehouse
- It helps to ensure security
Several data marts can be built, each for a particular business area provided they all conform to the data warehouse architecture from where they get the data for reporting. Data marts can be used in conjunction with each other. Refer to Figure 6-6.

<table>
<thead>
<tr>
<th><strong>6.2.7. Difference between Data Warehouse and Data Mart</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Warehouse</strong></td>
</tr>
<tr>
<td>A data warehouse is a repository which stores integrated information from multiple disparate sources for efficient querying and analysis</td>
</tr>
<tr>
<td>It is concerned with the organization of data and offers little concern regarding the presentation of data to the customers</td>
</tr>
<tr>
<td>There is usually a central data warehouse system</td>
</tr>
<tr>
<td>Data Warehouse is used on an enterprise level</td>
</tr>
<tr>
<td>Data Warehouse contains data from heterogeneous sources for analysis</td>
</tr>
</tbody>
</table>
6.2.8. Popular tools available for data warehousing

**Reporting / Analysis Tools:**
- Micro Strategy: DSS Agent / Server
- Cognos: Improptu
- Brio Technology: Brio Query
- Seagate Software: Crystal Reports

**ETL:**
- Data Junction
- Microsoft DTS (Available with SQL Server 7.0 and above)
- Oracle Warehouse Builder
- Informatica: PowerMart
- IBM: DataPropagator
- Acta: ActaWorks

**Databases:**
- MDDB
  - Oracle Express
  - SAS

### 6.3. Summary

- An OLAP application requires historical data because an analysis is generally based on a substantial amount of historical data to enable trend analysis and future predictions.
- A data warehouse is a repository which stores integrated information for efficient querying and analysis.
- Extract, transform ad load process (ETL) is described as the process of selecting, migrating, transforming, cleansing and converting mapped data from the operational environment to data warehouse environment.
- A data mart is a focused subset of a data warehouse that deals with a single area of data and is organized for quick analysis.
- Star schema is the simplest data warehouse schema. It resembles a star. The center of the star consists of one or more fact tables and the points radiating from the center are the dimension tables.
- Snowflake schema has a single, central fact table surrounded by normalized dimension hierarchies.
7. Glossary

Abstract: Conceptual/ theoretical object.

Abstraction: A simplified representation of something that is potentially quite complex. It is often not necessary to know the exact details of how something works, is represented or is implemented, because it can still be used in its simplified form.

Ambiguity: Uncertainty.

Anomalies: Irregularities.

Anomaly: A departure from the expected; an abnormality.

Atomic: The smallest levels to which data may be broken down and remain meaningful.

Attribute: The literal meaning is quality; characteristic; trait or feature. Entities are described in a database by a set of attributes. For example, in the bank system, Cust_ID, Cust_Email etc. describe Customer-Detail entity set.

Backup: A second copy of a file or set of files to be used in case the primary file(s) are destroyed or corrupted. Backups are essential for all but the most trivial work. For critical work, two backup sets are advisable.

Business rules: The rules/policies which govern the functioning of the application.

Business users: The users who owns the application.

Cardinality of a relation: The number of records/ tuples in a table.

Centralized: Systems where decision making, flow of data or the beginning of activities are initiated at the same central point and disseminated to remote points in the organization.

Conceptual: To generalize abstract ideas from specific instances.

Concurrent Access: Performing two (or more) operations on the same data at the same time.

Constraints: restriction, limitation.

Data manipulation: Data manipulation refers to the addition of new data, modification of existing data, etc.

Data Redundancy: Having the same data stored in more than one place in a database.

Decomposable: Further split or reduce.

Degree of a relation: The number of columns/ attributes in a table.

Distinct: Not identical.
**Distributed:** A computer system is distributed when different components and objects comprising an application can be located on different computers connected to a network.

**Encryption:** The process of manipulation of data to prevent accurate interpretation by all but those for whom the data is intended.

**End User:** The person for whom a system is being developed. Example: a bank teller or a bank manager is an end user of a bank system.

**Entity:** An entity is a “thing” or “object” in the real world that is distinguishable from other objects. Example: each person is an entity, and bank accounts can be considered to be entities.

**Flat files:** A flat file is a file containing records that has no structured interrelationship. Files used in programming fundamentals (PF) projects were essentially flat files.

**Fourth Generation Language (4GL):** A 4GL is typically non-procedural and designed so that end users can specify what they want without having to know how the computer will process their requirement.

**Grant Privilege:** To assign a privilege to a user or group.

**Heterogeneous:** varied, mixed, diverse.

**Heterogeneous Network:** A network that consists of workstations, servers, network interface cards, operating systems, and applications from many vendors, all working together as a single unit. The network may also use different media and different protocols over different network links.

**Homogeneous:** All the same, uniform, harmonized.

**Homogeneous Network:** A network composed of systems of similar architecture and runs a single network layer protocol.

**Inconsistency:** lacking uniformity or agreement.

**Instance:** Occurrence.

**Integrated:** United into a larger unit. Brought together to form a satisfactory and working whole

**Integrity Constraints:** A set of rules to ensure the correctness and accuracy of data.

**Interrelated:** interconnected

**Intuitive:** Natural.

**Iterative:** Process of repeating the same task.

**Jargons:** The specialized or technical language of a trade, profession, or similar group.

**Main Memory:** Please recall CHSSC concepts - All the read and write operations happen in main memory before they are written into hard disks.

**Minimal:** Minimum, the least possible.

**Model:** A representation or a scaled down structure of an object.
Page: It is part of a table. Usually in one page multiple rows are stored.

Participating entities: The entities which are joined by the relation.

Queries: A query is essentially a request that a user makes on the database.

Recovery: Restoration, return to an original state.

Refer: relate to.

Requirement specification: A document which contains requirement for a specific application.

Revoke Privilege: Cancel, withdraw.

Schema: A description of a database. It specifies (among other things) the relations, their attributes, and the domains of the attributes.

Semantic: Meaning.

Shared: A type of database access that allows multiple users to be logged on simultaneously to the same database

Simulate: To make a model.

Site: Geographical location.

Software application designer: The person who designs software applications.

SQL: (Structured Query Language). A language used by relational databases to query, update and manage data.

Static: Something which does not change. (Example: the typical web page is static in that it does not change until the webmaster physically alters the document.)

Superset: Given two sets, A and B, A is a superset of B if all elements of B are also elements of A. Every set is a superset of itself, and every set is a superset of the empty set.

Table: A table has a specified number of columns but can have any number of rows. Rows stored in a table are structurally equivalent to records from flat files.

Tablespace: The logical part of the database which represents collection of the structures like tables, etc created by various users.

Tangible: Physical object.

Transaction: A group of processing steps that are treated as a single activity to achieve a desired result. In DBMS, collections of operations that form a single logical unit of work are called transactions. A database system ensures proper execution of transactions despite failures—either the entire transaction executes, or none of it does.

Transient: Temporary, transitory, momentary.

Transitive: In-direct.
**Tuple**: This is a mathematical term for a finite sequence of n terms. For example, the set {1, 2, 3, 4} is a four-tuple. A tuple is equivalent of a record. In RDBMS, a table has n tuples.

**Unauthorized**: Not permitted, illegal, unlawful.

**View**: A view is a virtual table in the database defined by a query.
## 8. Index

**A**
- ABORT .............................................. 150
- ACID................................................. 153
- Aggregate Functions.............................. 108
- ALTER TABLE ....................................... 86
- Application Programmer.......................... 21
- Atomicity........................................... 152

**B**
- Batch Transaction Processing System ......... 151
- BETWEEN operator ............................... 103
- Bottom-Up Approach, ......................... 57
- Boyce Codd Normal Form ........................ 67

**C**
- Candidate Key...................................... 27
- Cardinality of relationship ....................... 38
- Centralized ......................................... 15
- CHECK OPTION .................................... 136
- Check-Points....................................... 177
- COMMIT ............................................. 150
- Conceptual / Logical level ....................... 18
- Concurrent access................................... 4
- Concurrent Access Anomalies ..................... 13
- Consistency.......................................... 152
- Co-Related Sub-Queries ........................... 120
- CREATE INDEX...................................... 90
- CREATE TABLE...................................... 80
- Cube................................................. 186

**D**
- Data Isolation....................................... 12
- Data Mart ........................................... 188
- Data Model ......................................... 23
- Data Redundancy ................................... 11
- Data Security ...................................... 11
- Data Warehouse .................................. 183
database .......................................... 2, 15
- Database Administrator.......................... 21
- Database Management System ................. 1
- DBMS Interface ..................................... 8
- DCL ................................................... 138
- DDL................................................... 79
- Deadlock ........................................... 168
- Deferred update ................................... 172
- DELETE Statement ................................ 96
- Determinant ........................................ 59
- Dimension Attributes............................. 185
- Dimension Hierarchies............................ 187
- Dimension Tables .................................. 186
- Dimensional Modeling............................ 186
- Dirty Read........................................... 156
- Distributed.......................................... 15
- DML................................................... 92
- Domain Integrity ................................... 153
- DROP TABLE ......................................... 89
- Durability........................................... 153

**E**
- Embedded SQL .................................... 141
- End User ............................................. 21
- Entity.................................................. 37
- Entity Integrity Constraint ....................... 29
- Entity Relationship Diagram .................... 37
- Entity-Relationship Modeling ................... 37
- ETL ................................................... 185
- Exclusive Lock ...................................... 159
- EXISTS............................................... 132
- External / View level ................................ 18

**F**
- Fact Tables ........................................ 186
- File System ......................................... 6
- First Normal Form .................................. 62
- Flat Files ............................................ 6
- Foreign Key ......................................... 30
- Full Functional Dependency....................... 60
- Functional Dependency............................ 59

**G**
- GRANT .............................................. 138
- Granularity of Locking............................ 160
- GROUP BY .......................................... 110

**H**
- HAVING ............................................. 113
- Heterogeneous...................................... 17
- Hierarchical Data Model ......................... 24
- Homogeneous....................................... 17

**I**
- Immediate Update ................................... 174
- IN operator......................................... 104
- Incorrect Summary .................................. 157
- Independent Sub-Queries ......................... 118
- INSERT .............................................. 93
- Integrated............................................ 3
- Intent Exclusive ................................... 163
- Intent Locking ..................................... 162
- Intent Share........................................ 163
- Internal / Physical level ......................... 19
<table>
<thead>
<tr>
<th>Interrelated</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERSECT</td>
<td>117</td>
</tr>
<tr>
<td>Isolation</td>
<td>152</td>
</tr>
<tr>
<td><strong>K</strong></td>
<td></td>
</tr>
<tr>
<td>Key Attributes</td>
<td>61</td>
</tr>
<tr>
<td><strong>L</strong></td>
<td></td>
</tr>
<tr>
<td>Lack of Flexibility</td>
<td>13</td>
</tr>
<tr>
<td>Lost Update</td>
<td>155</td>
</tr>
<tr>
<td><strong>M</strong></td>
<td></td>
</tr>
<tr>
<td>Many to Many Relationships</td>
<td>41</td>
</tr>
<tr>
<td>Master File</td>
<td>8</td>
</tr>
<tr>
<td>Measure Attributes</td>
<td>184</td>
</tr>
<tr>
<td>Metadata</td>
<td>184</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td></td>
</tr>
<tr>
<td>Network Data Model</td>
<td>24</td>
</tr>
<tr>
<td>Non-Key Attributes</td>
<td>32</td>
</tr>
<tr>
<td>Normalization</td>
<td>57</td>
</tr>
<tr>
<td>NOT EXISTS</td>
<td>132</td>
</tr>
<tr>
<td>NULL value</td>
<td>105</td>
</tr>
<tr>
<td><strong>O</strong></td>
<td></td>
</tr>
<tr>
<td>Object Based Logical Model</td>
<td>23</td>
</tr>
<tr>
<td>On Line Analytical Processing</td>
<td>182</td>
</tr>
<tr>
<td>One to One Relationship</td>
<td>38</td>
</tr>
<tr>
<td>One-to-Many</td>
<td>39</td>
</tr>
<tr>
<td>On-line Transaction Processing System</td>
<td>152</td>
</tr>
<tr>
<td><strong>P</strong></td>
<td></td>
</tr>
<tr>
<td>Partial Dependency</td>
<td>61</td>
</tr>
<tr>
<td>Phantom Record</td>
<td>157</td>
</tr>
<tr>
<td>Primary Key</td>
<td>28</td>
</tr>
<tr>
<td>Program/ Data Dependence</td>
<td>12</td>
</tr>
<tr>
<td><strong>R</strong></td>
<td></td>
</tr>
<tr>
<td>RDBMS</td>
<td>26</td>
</tr>
<tr>
<td>Real time Transaction Processing System</td>
<td>152</td>
</tr>
<tr>
<td>Record Based Logical Model</td>
<td>23</td>
</tr>
<tr>
<td>Referential Constraint</td>
<td>31</td>
</tr>
<tr>
<td>Relational Data Model</td>
<td>25</td>
</tr>
<tr>
<td>Relational Database</td>
<td>26</td>
</tr>
<tr>
<td>Relationship</td>
<td>37</td>
</tr>
<tr>
<td>REVOKE</td>
<td>139</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>150</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td></td>
</tr>
<tr>
<td>Second Normal Form</td>
<td>63</td>
</tr>
<tr>
<td>Security</td>
<td>170</td>
</tr>
<tr>
<td>SELECT</td>
<td>98</td>
</tr>
<tr>
<td>Self Referencing</td>
<td>31</td>
</tr>
<tr>
<td>Shared</td>
<td>3</td>
</tr>
<tr>
<td>Shared Intent Exclusive</td>
<td>163</td>
</tr>
<tr>
<td>Shared Lock</td>
<td>159</td>
</tr>
<tr>
<td>Snowflake Schema</td>
<td>187</td>
</tr>
<tr>
<td>Star Schema</td>
<td>187</td>
</tr>
<tr>
<td>Super Key</td>
<td>32</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td></td>
</tr>
<tr>
<td>tables</td>
<td>128</td>
</tr>
<tr>
<td>Third Normal Form</td>
<td>66</td>
</tr>
<tr>
<td>Top-Down Approach</td>
<td>43</td>
</tr>
<tr>
<td>Transaction File</td>
<td>8</td>
</tr>
<tr>
<td>Transactions</td>
<td>1</td>
</tr>
<tr>
<td>Transitive Dependency</td>
<td>61</td>
</tr>
<tr>
<td>TRUNCATE TABLE</td>
<td>89</td>
</tr>
<tr>
<td><strong>U</strong></td>
<td></td>
</tr>
<tr>
<td>UNION</td>
<td>115</td>
</tr>
<tr>
<td>UPDATE Statement</td>
<td>98</td>
</tr>
<tr>
<td><strong>V</strong></td>
<td></td>
</tr>
<tr>
<td>view</td>
<td>134</td>
</tr>
</tbody>
</table>