RDBMS (Oracle)
Copyright ©
This book contains the course content for RDBMS (Oracle).

First Edition 2014

Printed by
Universal Training Solutions Private Limited

Address
05th Floor, I-Space,
Bavdhan, Pune 411021.

All rights reserved. This book or any portion thereof may not, in any form or by any means including electronic or mechanical or photocopying or recording, be reproduced or distributed or transmitted or stored in a retrieval system or be broadcasted or transmitted.
### Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Content</td>
<td>II</td>
</tr>
<tr>
<td>II. List of Figures</td>
<td>V</td>
</tr>
<tr>
<td>III. List of Tables</td>
<td>VI</td>
</tr>
<tr>
<td>IV. Abbreviations</td>
<td>VII</td>
</tr>
<tr>
<td>V. Application</td>
<td>134</td>
</tr>
<tr>
<td>VI. Bibliography</td>
<td>148</td>
</tr>
<tr>
<td>VII. Self Assessment Answers</td>
<td>151</td>
</tr>
<tr>
<td>Book at a Glance</td>
<td></td>
</tr>
</tbody>
</table>
Contents

Chapter I................................................................................................................................................. 1
RDBMS Basic Concept............................................................................................................................. 1
  Aim ......................................................................................................................................................... 1
  Objectives ............................................................................................................................................. 1
  Learning outcome ................................................................................................................................. 1
  1.1 Database System Application ........................................................................................................ 2
  1.2 Purpose of Database System .......................................................................................................... 3
  1.3 Database Languages ....................................................................................................................... 3
    1.3.1 Data Definition Language ...................................................................................................... 4
    1.3.2 Data Manipulation Language ................................................................................................. 5
    1.3.3 Data Control Languages ........................................................................................................ 6
    1.3.4 Transaction Control Languages ............................................................................................. 7
  1.4 Relational Databases ..................................................................................................................... 8
  1.5 Database Design ............................................................................................................................. 10
  1.6 Database Architecture .................................................................................................................... 15
  Summary .............................................................................................................................................. 19
  References .......................................................................................................................................... 19
  Recommended Reading .......................................................................................................................... 20
  Self Assessment ................................................................................................................................... 21

Chapter II .............................................................................................................................................. 23
Relational Model ...................................................................................................................................... 23
  Aim ......................................................................................................................................................... 23
  Objectives ............................................................................................................................................. 23
  Learning outcome ............................................................................................................................... 23
  2.1 Introduction to Relational Model .................................................................................................... 24
  2.2 Structure of Relational Database ................................................................................................... 24
  2.3 Fundamental Relational-Algebra Operation .................................................................................... 25
    2.3.1 The Select Operation ............................................................................................................. 25
    2.3.2 The Cartesian Product Operation ........................................................................................... 25
    2.3.3 The Natural Join Operation .................................................................................................... 29
    2.3.4 The Division Operation .......................................................................................................... 30
    2.3.5 The Assignment Operation ..................................................................................................... 31
  Summary .............................................................................................................................................. 32
  References .......................................................................................................................................... 32
  Recommended Reading .......................................................................................................................... 32
  Self Assessment ................................................................................................................................... 33

Chapter III............................................................................................................................................ 35
SQL ......................................................................................................................................................... 35
  Aim ......................................................................................................................................................... 35
  Objectives ............................................................................................................................................. 35
  Learning outcome ............................................................................................................................... 35
  3.1 Introduction to SQL ......................................................................................................................... 36
  3.2 Data Definition Language ............................................................................................................. 36
  3.3 Basic Structure of SQL Queries ..................................................................................................... 38
  3.4 Aggregate Functions ....................................................................................................................... 40
  3.5 Nested Sub-queries ........................................................................................................................ 43
  3.6 Join Relations ................................................................................................................................ 49
  3.7 Integrity Constraints ......................................................................................................................... 55
  3.8 Authorisation ................................................................................................................................. 57
List of Figures

Fig. 1.1 Database design ................................................................. 10
Fig. 1.2 Entity and entity types .................................................. 12
Fig. 1.3 Weak entity ................................................................. 13
Fig. 1.4 Attributes ................................................................... 13
Fig. 1.5 Primary keys ............................................................... 14
Fig. 1.6 Cardinality ................................................................... 14
Fig. 1.7 E-R diagram .................................................................. 15
Fig. 1.8 Two-tier client-server architecture ............................. 17
Fig. 1.9 Web-based, two-tier client-server architecture .......... 17
Fig. 1.10 N-Tier client .............................................................. 18
Fig. 3.1 SQL for database access ............................................. 36
Fig. 3.2 Classification schemes ............................................... 50
Fig. 4.1 Employees entity set ................................................... 65
Fig. 4.2 Works in relationship set ............................................. 66
Fig. 4.3 Mapping cardinalities (a) One to one (b) One to many .. 67
Fig. 4.4 Mapping cardinalities (a) Many to one (b) Many to many .. 67
Fig. 4.5 A simple, example ERD ............................................. 68
Fig. 4.6 An example rough ERD ................................................ 71
Fig. 4.7 Access-date as attribute of the depositor relationship set .................................................. 73
Fig. 4.8 Specialisation and generalisation .................................. 75
Fig. 4.9 E-R diagram with redundant relationships .................. 77
Fig. 4.10 E-R diagram with aggregation ....................................... 78
Fig. 5.1 ASSIGN FD diagram .................................................. 88
Fig. 5.2 Example 1 .................................................................. 89
Fig. 5.3 Testing dependency preservation .................................. 91
Fig. 5.4 PROJECT relation ....................................................... 91
Fig. 5.5 Multivalued dependencies ........................................... 92
Fig. 6.1 State diagram of a transaction ....................................... 100
Fig. 6.2 Shadow-copy technique for atomicity and durability ... 101
Fig. 7.1 Granularity hierarchy .................................................. 119
Fig. 8.1 Block storage operations ............................................ 127
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1 Relation database</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Table 1.2 Database of name and number of city</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Table 2.1 Database of bank account</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>Table 2.2 Database of cities</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>Table 2.3 Result of $\sigma$branch-name = “Perryridge” (loan)</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Table 2.4 Customers with an account but no loan</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Table 2.5 Result of borrower $\times$ loan</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>Table 2.6 Result of $\sigma$branch-name = “Perryridge” (borrower $\times$ loan)</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>Table 2.7 Result of $\Pi$customer-name</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>Table 2.8 Customers with both an account and a loan at the bank</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>Table 2.9 Result of $\Pi$customer-name, loan-number, amount (borrower $\times$ loan)</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>Table 2.10 Result of $\Pi$branch-name(branch-city = “Brooklyn” (branch)</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>Table 2.11 Result of $\Pi$customer-name, branch-name (depositor $\times$ account)</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Table 4.1 An example relationship matrix</td>
<td></td>
<td>70</td>
</tr>
<tr>
<td>Table 4.2 An example complete relationship matrix</td>
<td></td>
<td>70</td>
</tr>
<tr>
<td>Table 5.1 Relation: Assign</td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>Table 5.2 Drinker tuples 1</td>
<td></td>
<td>92</td>
</tr>
<tr>
<td>Table 5.3 Drinker tuples 2</td>
<td></td>
<td>92</td>
</tr>
<tr>
<td>Table 5.4 Drinker tuples 3</td>
<td></td>
<td>93</td>
</tr>
<tr>
<td>Table 6.1 Schedule 1: a serial schedule in which $T_1$ is followed by $T_2$</td>
<td></td>
<td>103</td>
</tr>
<tr>
<td>Table 6.2 Schedule 3: a concurrent schedule equivalent to schedule 1</td>
<td></td>
<td>103</td>
</tr>
<tr>
<td>Table 6.3 Schedule 4: a concurrent schedule</td>
<td></td>
<td>104</td>
</tr>
<tr>
<td>Table 6.4 Schedule 3: showing only the read and write instructions</td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>Table 6.5 Schedule 7</td>
<td></td>
<td>106</td>
</tr>
<tr>
<td>Table 6.6 Schedule 8</td>
<td></td>
<td>106</td>
</tr>
<tr>
<td>Table 6.7 Schedule 9: a view-serialisable schedule</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>Table 6.8 Schedule 11</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>Table 6.9 Schedule 12</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>Table 7.1 Lock-compatibility matrix</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>Table 7.2 Schedule 4</td>
<td></td>
<td>116</td>
</tr>
<tr>
<td>Table 7.3 Schedule 5: a schedule produced by using validation</td>
<td></td>
<td>118</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>ACID</td>
<td>Atomicity, Consistency, Isolation, Durability</td>
<td></td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
<td></td>
</tr>
<tr>
<td>BCNF</td>
<td>Boyce-Codd Normal Form</td>
<td></td>
</tr>
<tr>
<td>CPUs</td>
<td>Central Processing Units</td>
<td></td>
</tr>
<tr>
<td>DBA</td>
<td>Database Administrator</td>
<td></td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
<td></td>
</tr>
<tr>
<td>DDL</td>
<td>Data Definition Language</td>
<td></td>
</tr>
<tr>
<td>DML</td>
<td>Data Manipulation Language</td>
<td></td>
</tr>
<tr>
<td>DTP</td>
<td>Distributed Transaction Processing</td>
<td></td>
</tr>
<tr>
<td>DQL</td>
<td>Data Query Language</td>
<td></td>
</tr>
<tr>
<td>E-R</td>
<td>Entity-Relationship</td>
<td></td>
</tr>
<tr>
<td>ERM</td>
<td>Entity-Relationship Model</td>
<td></td>
</tr>
<tr>
<td>ERD</td>
<td>Entity-Relationship Diagram</td>
<td></td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
<td></td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
<td></td>
</tr>
<tr>
<td>NF</td>
<td>Normal Forms</td>
<td></td>
</tr>
<tr>
<td>2NF</td>
<td>Second Normal Form</td>
<td></td>
</tr>
<tr>
<td>3NF</td>
<td>Third Normal Form</td>
<td></td>
</tr>
<tr>
<td>4NF</td>
<td>Fourth Normal Form</td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Operating Systems</td>
<td></td>
</tr>
<tr>
<td>PRDBMS</td>
<td>Pseudo Relational Database Management Systems</td>
<td></td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
<td></td>
</tr>
<tr>
<td>RI</td>
<td>Referential Integrity</td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td>Structure Query Language</td>
<td></td>
</tr>
<tr>
<td>TRDBMS</td>
<td>Truly-Relational Database Management Systems</td>
<td></td>
</tr>
</tbody>
</table>
Chapter I
RDBMS Basic Concept

Aim
The aim of this chapter is to:

• explain database system application
• elucidate data storage management
• discuss data transformation and presentation

Objectives
The objectives of this chapter are to:

• explain security management
• discuss the security problem
• describe backup and recovery management

Learning outcome
At the end of this chapter, you will be able to:

• recognise data integrity management
• identify database communication interfaces
• understand database language
1.1 Database System Application

A Database Management System performs the following functions:

- Data Dictionary Management
- Data Storage Management
- Data Transformation and Presentation
- Security Management
- Multi User Access Control
- Backup and Recovery Management
- Data Integrity Management
- Database Access Languages and Application Interface
- Database Communication Interface
- Data Dictionary Management

The data dictionary stores definitions of data elements and their relationships. This information is termed as metadata. The metadata includes definition of data, data types, relationship between data, integrity constraints, etc. Any changes made in a database structure are automatically reflected in the data dictionary. In short DBMS provides data abstraction and removes structural and data dependency from the system.

**Data storage management**: DBMS creates the complex structures required for data storage. The users are freed from defining, programming and implementing the complex physical data characteristics.

**Data transformation and presentation**: DBMS supports data independence. Hence the DBMS translate logical request into commands that physically locate and retrieve the requested data. The DBMS formats the physically retrieved data according to the logical data format specifications.

**Security management**: DBMS creates a security system that enforces user security and data privacy within the database. Security rules determine the access rights of the users. Read/write access is given to the user is specified using access rights.

**Multiuser access control**: DBMS ensures that multiple users can access the database concurrently without compromising the integrity of the database. Hence the database ensures data integrity and data consistency.

**Backup and recovery management**: DBMS provide backup and data recovery procedures to ensure data safety and integrity. DBMS system provides special utilities which allow the DBA to perform routine and special backup and restore procedures. Recovery Management deals with the recovery of the database after a failure.

**Data integrity management**: DBMS promotes and enforce integrity rules to eliminate data integrity problems, thus minimising the data redundancy and maximising data consistency.

**Database access languages and application interface**: DBMS provides data access via query language. A query language is a non-procedural language that is the user only need to specify what must be done without specifying how it is to be done. DBMS’s query language contains two components: a data definition language (DDL) and a data manipulation language (DML). The DBMS also provide data access to programmers via programming languages.

**Database communication interfaces**: Different users may access the database through a network environment. So DBMS provide communication functions to access the database through computer network environment.
1.2 Purpose of Database System

To see why database management systems are necessary, let’s look at a typical “file-processing system” supported by a conventional operating system.

The application is a savings in the bank
- Savings account and customer records are kept in permanent system
- Application programs are written to manipulate files to perform the following tasks:
  - Debit or credit an account.
  - Add a new account.
  - Find an account balance.
  - Generate monthly statements.

Development of the system proceeds as follows:
- New application programs must be written as the need arises.
- New permanent files are created as required, but over a long period of time file may be in different format and application program may be in different language. So we can see there are problems with the straight file-processing approach
- Data redundancy and inconsistency: Same information may be duplicated in several places. All copies may not be updated properly.
- Difficulty in accessing data: May have to write a new application program to satisfy an unusual request for example, find all customers with the same postal code. Could generate this data manually, but a long job.
- Data isolation:
  - Data in different files
  - Data in different formats
  - Difficult to write new application programs
- Security problems: Every user of the system should be able to access only the data they are permitted to see. For example, payroll people only handle employee records, and cannot see customer accounts; tellers only access account data and cannot see payroll data. Difficult to enforce this with application programs.
- Integrity problems: Data may be required to satisfy constraints, e.g., no account balances below $25.00. Again, difficult to enforce or to change constraints with the file-processing approach. These problems and others led to the development of database management systems.

1.3 Database Languages

A database language standard specifies the semantics of various components of a database management system (DBMS).
- In particular, it defines the structures and operations of a data model implemented by the DBMS, as well as other components that support data definition, data access, security, programming language interface, and data administration.
- The SQL standard specifies data definition, data manipulation, and other associated facilities of a DBMS that supports the relational data model.
- A database language standard is appropriate for all database applications where data will be shared with other applications, where the life of the application is longer than the life of current equipment, or where the application is to be understood and maintained by programmers other than the original ones.
1.3.1 Data Definition Language

The Data Definition Language (DDL) is used to create and destroy databases and database objects.

- These commands will mainly be used by database administrators during the setup and removal phases of a database project.
- The Data Definition Language (DDL) is used to create and destroy databases and database objects. These commands will primarily be used by database administrators during the setup and removal phases of a database project.

Let’s take a look at the structure and usage of four basic DDL commands:

CREATE

Installing a database management system (DBMS) on a computer allows you to create and manage many independent databases.

- For example, you may want to maintain a database of customer contacts for your sales department and a personnel database for your HR department.
- The CREATE command can be used to establish each of these databases on your platform. For example, the command:

```
CREATE DATABASE employees
```

Creates an empty database named “employees” on your DBMS. After creating the database, your next step is to create tables that will contain data.

Another variant of the CREATE command can be used for this purpose. The command:

```
CREATE TABLE personal_info (first_name char(20) not null, last_name char(20) not null, employee_id int not null)
```

Establishes a table titled “personal_info” in the current database. In our example, the table contains three attributes: first_name, last_name and employee_id.

- Don’t worry about the other information included in the command -- we’ll cover that in a future article.

USE

The USE command allows you to specify the database you wish to work with within your DBMS.

- For example, if we’re currently working in the sales database and want to issue some commands that will affect the employee’s database; we would preface them with the following SQL command:
- USE employees: It’s important to always be conscious of the database you are working in before issuing SQL commands that manipulate data.

ALTER

Once you have created a table within a database, you may wish to modify the definition of it.

- The ALTER command allows you to make changes to the structure of a table without deleting and recreating it. Take a look at the following command:

```
ALTER TABLE personal_info
    ADD salary money null
```

- This example adds a new attribute to the personal_info table an employee’s salary.
- The “money” argument specifies that an employee’s salary will be stored using dollars and cents format.
- Finally, the “null” keyword tells the database that it’s OK for this field to contain no value for any given employee.
DROP
The final command of the Data Definition Language, DROP, allows us to remove entire database objects from our DBMS.

- For example, if we want to permanently remove the personal_info table that we created, we’d use the following command:
  
  ```sql
  DROP TABLE personal_info
  ```

- Similarly, the command below would be used to remove the entire employees database.

DROP DATABASE employees:

- Creates an empty database named “employees” on your DBMS.
- After creating the database, your next step is to create tables that will contain data.
- Another variant of the CREATE command can be used for this purpose. The command:
  ```sql
  CREATE TABLE personal_info (first_name char(20) not null, last_name char(20) not null, employee_id int not null)
  ```

  Establishes a table titled “personal_info” in the current database. In our example, the table contains three attributes: first_name, last_name and employee_id. Don’t worry about the other information included in the command -- we’ll cover that in a future article.

USE
The USE command allows you to specify the database you wish to work with within your DBMS.

- For example, if we’re currently working in the sales database and want to issue some commands that will affect the employees database, we would preface them with the following SQL command:
  ```sql
  USE employees
  ```

  It’s important to always be conscious of the database you are working in before issuing SQL commands that manipulate data.

ALTER
Once you have created a table within a database, you may wish to modify the definition of it.

- The ALTER command allows you to make changes to the structure of a table without deleting and recreating it.
- Take a look at the following command:
  ```sql
  ALTER TABLE personal_info
  ADD salary money null
  ```

  This example adds a new attribute to the personal_info table an employee’s salary.

  - The “money” argument specifies that an employee’s salary will be stored using a dollars and cents format.
  - Finally, the “null” keyword tells the database that it’s OK for this field to contain no value for any given employee.

DROP
The final command of the Data Definition Language, DROP, allows us to remove entire database objects from our DBMS.

- For example, if we want to permanently remove the personal_info table that we created, we’d use the following command:
  ```sql
  DROP TABLE personal_info
  ```

- Similarly, the command below would be used to remove the entire employees database:
  ```sql
  DROP DATABASE employees
  ```
One has to use this command carefully.

- The DROP command removes entire data structures from your database.
- If you want to remove individual records, use the DELETE command of the Data Manipulation Language.

### 1.3.2 Data Manipulation Language

The Data Manipulation Language (DML) is used to retrieve, insert and modify database information.

- These commands will be used by all database users during the routine operation of the database.
- Let's take a brief look at the basic DML commands:

#### INSERT

The INSERT command in SQL is used to add records to an existing table.

- Returning to the personal_info example from the previous section, let's imagine that our HR department needs to add a new employee to their database.
- They could use a command similar to the one shown below:

```sql
INSERT INTO personal_info
VALUES('bart', 'simpson', 12345, $45000)
```

- Note that there are four values specified for the record. These correspond to the table attributes in the order they were defined: first_name, last_name, employee_id, and salary.

#### SELECT

The SELECT command is the most commonly used command in SQL.

- It allows database users to retrieve the specific information they desire from an operational database.
- Let's take a look at a few examples, again using the personal_info table from our employees database.
- The command shown below retrieves all of the information contained within the personal_info table. Note that the asterisk is used as a wildcard in SQL.

```sql
SELECT *
FROM personal_info
```

- Alternatively, users may want to limit the attributes that are retrieved from the database.
- For example, the Human Resources department may require a list of the last names of all employees in the company.
- The following SQL command would retrieve only that information:

```sql
SELECT last_name
FROM personal_info
```

- Finally, the WHERE clause can be used to limit the records that are retrieved to those that meet specified criteria.
- The CEO might be interested in reviewing the personnel records of all highly paid employees.
- The following command retrieves all of the data contained within personal_info for records that have a salary value greater than $50,000:

```sql
SELECT *
FROM personal_info
WHERE salary > $50000
```
**UPDATE**
The UPDATE command can be used to modify information contained within a table, either in bulk or individually.

- Each year, our company gives all employees a 3% cost-of-living increase in their salary.
- The following SQL command could be used to quickly apply this to all of the employees stored in the database:

  ```sql
  UPDATE personal_info
  SET salary = salary * 1.03
  ```

- On the other hand, our new employee Bart Simpson has demonstrated performance above and beyond the call of duty.
- Management wishes to recognise his stellar accomplishments with a $5,000 raise.
- The WHERE clause could be used to single out Bart for this raise:

  ```sql
  UPDATE personal_info
  SET salary = salary + $5000
  WHERE employee_id = 12345
  ```

**DELETE**
Finally, let’s take a look at the DELETE command. You will find that the syntax of this command is similar to that of the other DML commands.

- The DELETE command with a WHERE clause can be used to remove his record from the personal_info table:

  ```sql
  DELETE FROM personal_info
  WHERE employee_id = 12345
  ```

1.3.3 Data Control Languages
A Data Control Language (DCL) is a part of SQL and is used to control the access to data in a database.

- Data Control Language primarily includes the commands grant and revoke.
- The following privileges (permissions) can be granted to or revoked from a user or role:

  - **SELECT** to return a result set from a table.
  - **INSERT** to insert records.
  - **UPDATE** to modify a record in a table.
  - **DELETE** to delete specific records.
  - **CONNECT** to set a database connection.
  - **EXECUTE** to execute a procedure.
  - **USAGE** to modify the maximum number of records allowed.

1.3.4 Transaction Control Languages
Transaction control statements manage changes made by DML statements.

**What is a transaction?**
A transaction is a set of SQL statements which Oracle treats as a Single Unit i.e., all the statements should execute successfully or none of the statements should execute.

- To control transactions Oracle does not made permanent any DML statements unless you commit it.
- If you don’t commit the transaction and power goes off or system crashes then the transaction is roll backed.
- TCL Statements available in Oracle are

  - COMMIT: Make changes done in transaction permanent.
  - ROLLBACK: Rollbacks the state of database to the last commit point.
  - SAVEPOINT: Use to specify a point in transaction to which later you can rollback.
COMMIT
• To make the changes done in a transaction permanent issues the COMMIT statement.
• The syntax of COMMIT Statement is
  COMMIT [WORK] [COMMENT ‘your comment’];

WORK is optional. COMMENT is also optional, specify this if you want to identify this transaction in data dictionary DBA_2PC_PENDING.

Example
  insert into emp (empno,ename,sal) values (101,’Abid’,2300);
  Commit;

ROLLBACK
To rollback the changes done in a transaction give rollback statement. Rollback restore the state of the database to the last commit point.

Example:
  delete from emp;
  rollback; /* undo the changes */

SAVEPOINT
Specify a point in a transaction to which later you can roll back.

Example
  insert into emp (empno,ename,sal) values (109,’Sami’,3000);
  savepoint a;
  insert into dept values (10,’Sales’,’Hyd’);
  savepoint b;
  insert into salgrade values (‘III’,9000,12000);
  Now if you give
  rollback to a;

Then row from salgrade table and dept will be roll backed. Now you can commit the row inserted into emp table or rollback the transaction.

1.4 Relational Databases
Databases have been a staple of business computing from the very beginning of the digital era.
• In fact, the relational database was born in 1970 when E.F. Codd, a researcher at IBM, wrote a paper outlining the process. Since then, relational databases have grown in popularity to become the standard.
• Originally, databases were flat. This means that the information was stored in one long text file, called a tab delimited file.
• Each entry in the tab delimited file is separated by a special character, such as a vertical bar (|).
• Each entry contains multiple pieces of information (fields) about a particular object or person grouped together as a record.
• The text file makes it difficult to search for specific information or to create reports that include only certain fields from each record. Here’s an example of the file created by a flat database:

| Lname, FName, Age, Salary | Smith, John, 35, $280 | Doe, Jane, 28, $325 | Brown, Scott, 41, $265 | Howard, Shemp, 48, $359 | Taylor, Tom, 22, $250 |
• You can see that you have to search sequentially through the entire file to gather related information, such as age or salary.
• A relational database allows you to easily find specific information. It also allows you to sort based on any field and generate reports that contain only certain fields from each record.
• Relational databases use tables to store information. The standard fields and records are represented as columns (fields) and rows (records) in a table.
• Look at this example:

<table>
<thead>
<tr>
<th>LName</th>
<th>FName</th>
<th>City</th>
<th>Age</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>John</td>
<td>3</td>
<td>35</td>
<td>$280</td>
</tr>
<tr>
<td>Doe</td>
<td>Jane</td>
<td>1</td>
<td>28</td>
<td>$325</td>
</tr>
<tr>
<td>Brown</td>
<td>Scott</td>
<td>3</td>
<td>41</td>
<td>$265</td>
</tr>
<tr>
<td>Howard</td>
<td>Shemp</td>
<td>4</td>
<td>48</td>
<td>$359</td>
</tr>
<tr>
<td>Taylor</td>
<td>Tom</td>
<td>2</td>
<td>22</td>
<td>$250</td>
</tr>
</tbody>
</table>

Table 1.1 Relation database

• In the relational database example, you can quickly compare salaries and age because of the arrangement of data in columns.
• The relational database model takes advantage of this uniformity to build completely new tables out of required information from existing tables.
• In other words, it uses the relationship of similar data to increase the speed and versatility of the database.
• The “relational” part of the name comes into play because of mathematical relations.
• A typical relational database has anywhere from 10 to more than 1,000 tables.
• Each table contains a column or columns that other tables can key on to gather information from that table.
• Look at the table below that matches the number in the city column of the above table with the name of a city.

<table>
<thead>
<tr>
<th>City #</th>
<th>City Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Boston</td>
</tr>
<tr>
<td>2</td>
<td>London</td>
</tr>
<tr>
<td>3</td>
<td>New York</td>
</tr>
<tr>
<td>4</td>
<td>Los Angeles</td>
</tr>
</tbody>
</table>

Table 1.2 Database of name and number of city

• By storing this information in another table, the database can create a single small table with the locations that can then be used for a variety of purposes by other tables in the database.
• A typical large database, like a big Web site, such as Amazon would have, will contain hundreds or thousands of tables like this all used together to quickly find the exact information needed at any given time.
• Relational databases are created using a special computer language, structured query language (SQL) that is the standard for database interoperability.
• SQL is the foundation for all of the popular database applications available today, from Access to Oracle.
1.5 Database Design

The ability to design databases and associated applications is critical to the success of the modern enterprise.

- Database design requires understanding both the operational and business requirements of an organisation as well as the ability to model and realise those requirements using a database.
- Developing database and information systems is performed using a development lifecycle, which consists of a series of steps.

![Database Design Diagram]

**Fig. 1.1 Database design**

**Specification requirements gathering**

The most critical aspect of specification is the gathering and compilation of system and user requirements. This process is normally done in conjunction with managers and users. The major goal in requirements gathering is to:

- Collect the data used by the organisation, identify relationships in the data, identify future data needs and determine how the data is used and generated.
- The starting place for data collection is gathering existing forms and reviewing policies and systems.
- Then, ask users what the data means, and determine their daily processes. These things are especially critical.
- Identification of unique fields (keys), data dependencies, relationships, and constraints (high-level), the data sizes and their growth rates
- Fact-finding is using interviews and questionnaires to collect facts about systems, requirements, and preferences.
Five fact-finding techniques:
- Examining documentation
- Interviewing
- Observing the enterprise in operation
- Research
- Questionnaires

Database design
The requirements gathering and specification provides you with a high-level understanding of the organisation, its data, and the processes that you must model in the database. Database design involves constructing a suitable model of this information. Since the design process is complicated, especially for large databases, database design is divided into three phases:
- Conceptual database design
- Logical database design
- Physical database design

Conceptual database design
Conceptual database design involves modelling the collected information at a high-level of abstraction without using a particular data model or DBMS. Reasons for conceptual modeling are:
- Independent of DBMS
- Allows for easy communication between end-users and developers.
- Has a clear method to convert from high-level model to relational model.
- Conceptual schema is a permanent description of the database requirements.

Entity-relationship model
- Most popular conceptual model for database design
- Basis for many other models
- Describes the data in a system and how that data is related
- Describes data as entities, attributes and relationships

Database requirements
- We must convert the written database requirements into an E-R diagram
- Need to determine the entities, attributes and relationships.
  - nouns = entities
  - adjectives = attributes
  - verbs = relationships
**Teaching database**
Design an E-R schema for a database to store info about professors, courses and course sections indicating the following:

- The name and employee ID number, salary and email address(es) of each professor
- How long each professor has been at the university
- The course sections each professor teaches
- The name, number and topic for each course offered
- The section and room number for each course section
- Each course section must have only one professor
- Each course can have multiple sections

**Entities**
Entity is an object of E-R model represents a “thing” with an independent existence. Entity type is used to define a set of entities with the same properties.

![Entity and entity types](image)

**Weak entity**
- Weak entities do not have key attributes of their own.
- Weak entities cannot exist without a relationship to another entity.
- A partial key is the portion of the key that comes from the weak entity. The rest of the key comes from the other entity in the relationship.
- Weak entities always have total participation as they cannot exist without the identifying relationship.
Attributes
Each entity has a set of associated properties that describes the entity. These properties are known as attributes. Attributes can be as follows:
- Simple or composite
- Single or multivalued
- Stored or derived
- NULL

Fig. 1.3 Weak entity

Fig. 1.4 Attributes
**Keys**
- Candidate key: An attribute or set of attributes that uniquely identifies individual occurrences of an entity type.
- Primary key: An entity type may have one or more possible candidate keys, one of which is selected to be a primary key.

**Fig. 1.5 Primary keys**
- Employee ID is the primary key
- Primary keys must be unique for the entity in question

**Relationships:**

Defines a set of associations between various entities.

**Cardinality**
The number of relationships that an entity may participate in: 1:1, 1:N, N:M, M:1
Today’s database professionals face several options when considering architectures in which to employ to address various needs of their employers and/or clients. The text given below will provide an overview of three main categories of database architectures and their sub-categories, as well as offer some insight into the benefits of each.

**Application logic**
Database architectures can be distinguished by examining the way application logic is distributed throughout the system.

- **Application logic consists of three components:** Presentation Logic, Processing Logic, and Storage Logic.
- The presentation logic component is responsible for formatting and presenting data on the user’s screen.
- The processing logic component handles data processing logic, business rules logic, and data management logic.
- Finally, the storage logic component is responsible for the storage and retrieval from actual devices such as a hard drive or RAM.
- By determining which tier(s) these components are processed on we can get a good idea of what type of architecture and subtype we are dealing with.

**One tier architectures**
Imagine a person on a desktop computer who uses Microsoft Access to load up a list of personal addresses and phone numbers that he or she has saved in MS Windows’ “My Documents” folder. This is an example of one-tier database architecture.

- The program (Microsoft Access) runs on the user’s local machine, and refers a file that is stored on that machine’s hard drive, thus using a single physical resource to access and process information.
- Another example of one-tier architecture is file server architecture.
In this scenario, a workgroup database is stored in a shared location on a single machine.

Workgroup members use a software package such as Microsoft Access to load the data and then process it on their local machine.

In this case, the data may be shared among different users, but all of the processing occurs on the local machine.

Essentially, the file-server is just an extra hard drive from which to retrieve files. Yet another one-tier architectures have appeared is in that of mainframe computing.

In this outdated system, large machines provide directly connected unintelligent terminals with the means necessary to access, view and manipulate data.

Even though this is considered as client-server system, since all of the processing power (for both data and applications) occurs on a single machine, we have one-tier architecture.

One-tier architectures can be beneficial when we are dealing with data that is relevant to a single user (or small number of users) and we have a relatively small amount of data.

They are somewhat inexpensive to deploy and maintain.

Two-tier client/server architectures

Two-tier architecture is one that is familiar to many of today’s computer users.

A common implementation of this type of system is that of a Microsoft Windows based client program that accesses a server database such as Oracle or SQL Server.

Users interact through a GUI (Graphical User Interface) to communicate with the database server across a network via SQL (Structured Query Language).

In two-tier architectures it is important to note that there exists two configurations.

A thin-client (fat-server) configuration exists when most of the processing occurs on the server tier.

Conversely, a fat-client (thin-server) configuration exists when most of the processing occurs on the client machine.

Another example of two-tier architecture can be seen in web-based database applications.

In this case, users interact with the database through applications that are hosted on a web-server and displayed through a web-browser such as Internet Explorer.

The web server processes the web application, which can be written in a language such as PHP or ASP.

The web application connects to a database server to pass along SQL statements which in turn are used to access, view, and modify data.

The DB server then passes back the requested data which is then formatted by the web server for the user.

Although this appears to be a three-tier system because of the number of machines required to complete the process, it is not.

The web-server does not normally house any of the business rules and therefore should be considered part of the client tier in partnership with the web-browser.

Two-tier architectures can prove to be beneficial when we have a relatively small number of users on the system (100-150) and we desire an increased level of scalability.
N-tier client/server architectures

Most n-tier database architectures exist in a three-tier configuration.

- In this architecture the client/server model expands to include a middle tier (business tier), which is an application server that houses the business logic. This middle tier relieves the client application(s) and database server of some of their processing duties by translating client calls into database queries and translating data from the database into client data in return.
- Consequently, the client and server never talk directly to one-another.
- A variation of the n-tier architecture is the web-based n-tier application. These systems combine the scalability benefits of n-tier client/server systems with the rich user interface of web-based systems.
- Because the middle tier in three-tier architecture contains the business logic, there is an increased scalability and isolation of the business logic, as well as added flexibility in the choice of database vendors.
Fig. 1.10 N-Tier client
Summary

• The data dictionary stores definitions of data elements and their relationships.
• The metadata includes definition of data, data types, relationship between data, integrity constraints etc.
• DBMS creates complex structures required for data storage. The users are freed from defining, programming and implementing the complex physical data characteristics.
• DBMS supports data independence. Hence the DBMS translate logical request into commands that physically locate and retrieve the requested data.
• DBMS creates a security system that enforces user security and data privacy within the database.
• DBMS ensures that multiple users can access the database concurrently without compromising the integrity of the database.
• DBMS’s query language contains two components: a data definition language (DDL) and a data manipulation language (DML).
• Data redundancy and inconsistency is same information that may be duplicated in several places. All copies may not be updated properly.
• The Data Definition Language (DDL) is used to create and destroy databases and database objects.
• Installing a database management system (DBMS) on a computer allows you to create and manage many independent databases.
• The CREATE command can be used to establish databases on your platform.
• The Data Manipulation Language (DML) is used to retrieve, insert and modify database information.
• A Data Control Language (DCL) is a part of SQL and is used to control the access to data in a database. Data Control Language primarily includes the commands grant and revoke.
• Transaction control statements manage changes made by DML statements.
• A relational database allows you to easily find specific information. It also allows you to sort based on any field and generate reports that contain only certain fields from each record.
• Database design requires understanding both the operational and business requirements of an organisation as well as the ability to model and realise those requirements using a database.
• The most critical aspect of specification is the gathering and compilation of system and user requirements. This process is normally done in conjunction with managers and users.
• Database design involves constructing a suitable model of this information.
• Candidate key is an attribute or set of attributes that uniquely identifies individual occurrences of an entity type.
• Primary key is an entity type and may have one or more possible candidate keys, one of which is selected to be a primary key.

References

RDBMS (Oracle)


**Recommended Reading**

• Date, J. C., 2006. *An Introduction to Database Systems*, Pearson Education India.
Self Assessment

1. The data dictionary stores ____________ of data elements and their relationships.
   a. definitions
   b. copy
   c. values
   d. function

2. Which of the following statements is true?
   a. DBMS creates complex structures required for user.
   b. DBMS creates complex structures required for computer.
   c. DBMS creates complex structures required for data storage.
   d. DBMS creates complex structures required for programmer.

3. The DBMS translates logical request into __________ which physically locate and retrieve the requested data.
   a. commands
   b. code
   c. memory
   d. query

4. DBMS creates a ___________ that enforces user security and data privacy within the database.
   a. system design
   b. security system
   c. synchronisation
   d. code design

5. DBMS ensures that ___________ can access the database concurrently without compromising the integrity of the database.
   a. computers
   b. peripherals
   c. multiple users
   d. system design

6. Which of the following describes the query language?
   a. Non-procedural language
   b. Function language
   c. Object oriented language
   d. Procedural language

7. The ___________ is used to create and destroy databases and database objects.
   a. data definition language
   b. data control language
   c. data manipulation language
   d. data definition language
8. __________ allows us to remove entire database objects from our DBMS.
   a. INSERT
   b. DROP
   c. REMOVE
   d. SELECT

9. The __________ command in SQL is used to add records to an existing table.
   a. INSERT
   b. DROP
   c. REMOVE
   d. SELECT

10. The __________ command is the most commonly used command in SQL.
    a. INSERT
    b. DROP
    c. REMOVE
    d. SELECT
Chapter II
Relational Model

Aim
The aim of this chapter is to:

- explain the concept of relational model
- describe the fundamental relation-algebra operation
- discuss select operation

Objectives
The objectives of this chapter are to:

- explain Cartesian product operation
- recognise natural join operation
- get an overview of division operation

Learning outcome
At the end of this chapter, you will be able to:

- understand assignment operation
- recognise relational modelling
- describe relational model basic
2.1 Introduction to Relational Model

Relational model is the most widely used data model for commercial data-processing because it is simple and easy to maintain. The model is based on a collection of tables. Users of the database can create tables, insert new tables or modify existing tables. There are several languages for database programming. Relational modeling was introduced by Ted Codd in 1970 who was an IBM Researcher. He laid the foundation for database theory. Many database concepts and products are based on his model.

Relational model basic

The relational model gives us a single way to represent data: as a two-dimensional table called a relation.

- Attributes
- Schemas
- Tuples
- Domains
- Equivalent Representations of a Relation

2.2 Structure of Relational Database

A relational database consists of a collection of tables, each having a unique name.

- A row in a table represents a relationship among a set of values. Thus, a table represents a collection of relationships.
- There is a direct correspondence between the concept of a table and the mathematical concept of a relation. A substantial theory has been developed for relational databases.

Basic structure

Table shown below is the deposit and customer table for a banking example. It has four attributes and for each attribute there is a permitted set of values, called the domain of that attribute.

<table>
<thead>
<tr>
<th>bname</th>
<th>Account #</th>
<th>cname</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>101</td>
<td>Johnson</td>
<td>500</td>
</tr>
<tr>
<td>Lougheed</td>
<td>215</td>
<td>Smith</td>
<td>700</td>
</tr>
<tr>
<td>SFU</td>
<td>102</td>
<td>Hayes</td>
<td>400</td>
</tr>
<tr>
<td>SFU</td>
<td>304</td>
<td>Adams</td>
<td>1300</td>
</tr>
</tbody>
</table>

Table 2.1 Database of bank account

<table>
<thead>
<tr>
<th>cname</th>
<th>street</th>
<th>ccity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson</td>
<td>Pender</td>
<td>Vancouver</td>
</tr>
<tr>
<td>Smith</td>
<td>North</td>
<td>Burnaby</td>
</tr>
<tr>
<td>Hayes</td>
<td>Curtis</td>
<td>Burnaby</td>
</tr>
<tr>
<td>Adams</td>
<td>No.3 Road</td>
<td>Richmond</td>
</tr>
<tr>
<td>Jones</td>
<td>Oak</td>
<td>Vancouver</td>
</tr>
</tbody>
</table>

Table 2.2 Database of cities
2.3 Fundamental Relational-Algebra Operation

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result.

2.3.1 The Select Operation

The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection.

- The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. Thus, to select those tuples of the loan relation where the branch is “Perryridge,” we write
  \[ \sigma_{\text{branch-name} = \text{“Perryridge”}} (\text{loan}) \]
- If the loan relation is as shown in the table below, then the relation that results from the preceding query is as shown in Table 2.3.
- We can find all tuples in which the amount lent is more than $1200 by writing
  \[ \sigma_{\text{amount} > 1200} (\text{loan}) \]
- In general, we allow comparisons using =, ≠, <, ≤, >, ≥ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives and (\(\land\)), or (\(\lor\)), and not (\(\neg\)).
- Thus, to find those tuples pertaining to loans of more than $1200 made by the Perryridge branch, we write
  \[ \sigma_{\text{branch-name} = \text{“Perryridge”}} \land \text{amount} > 1200 (\text{loan}) \]

<table>
<thead>
<tr>
<th>Loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
</tbody>
</table>

Table 2.3 Result of \(\sigma_{\text{branch-name} = \text{“Perryridge”}} (\text{loan})\)

- The selection predicate may include comparisons between two attributes.
- To illustrate, consider the relation loan-officer that consists of three attributes: customer-name, banker-name, and loan-number, which specifies that a particular banker is the loan officer for a loan that belongs to some customer.
- To find all customers who have the same name as their loan officer, we can write
  \[ \sigma_{\text{customer-name} = \text{banker-name}} (\text{loan-officer}) \]

2.3.2 The Cartesian Product Operation

The Cartesian-product operation, denoted by a cross (\(\times\)), allows us to combine information from any two relations. We write the Cartesian product of relations \(r_1\) and \(r_2\) as \(r_1 \times r_2\).

<table>
<thead>
<tr>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson</td>
</tr>
<tr>
<td>Lindsay</td>
</tr>
<tr>
<td>Tuner</td>
</tr>
</tbody>
</table>

Table 2.4 Customers with an account but no loan
Recall that a relation is by definition a subset of a Cartesian product of a set of domains.

From that definition, we should already have an intuition about the definition of the Cartesian-product operation. However, since the same attribute name may appear in both $r_1$ and $r_2$, we need to devise a naming schema to distinguish between these attributes.

We do so here by attaching to an attribute the name of the relation from which the attribute originally came. For example, the relation schema for $r = \text{borrower} \times \text{loan}$ is

(borrower.customer-name, borrower.loan-number, loan.loan-number, loan.branch-name, loan.amount)

With this schema, we can distinguish borrower.loan-number from loan.loan-number. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for $r$ as (customer-name, borrower.loan-number, loan.loan-number, branch-name, amount)

This naming convention requires that the relations that are the arguments of the Cartesian-product operation have distinct names.

Now that we know the relation schema for $r = \text{borrower} \times \text{loan}$, what tuples appear in $r$. As you may suspect, we construct a tuple of $r$ out of each possible pair of tuples: one from the borrower relation and one from the loan relation. Thus, $r$ is a large relation, as you can see from table 2.4, which includes only a portion of the tuples that make up $r$.

Assume that we have $n_1$ tuples in borrower and $n_2$ tuples in loan.

Then, there are $n_1 \times n_2$ ways of choosing a pair of tuples one tuple from each relation; so there are $n_1 \times n_2$ tuples in $r$.

In particular, note that for some tuples $t$ in $r$, it may be that $t[\text{borrower.loan-number}] \neq t[\text{loan.loan-number}]$.

In general, if we have relations $r_1$ (R1) and $r_2$ (R2), then $r_1 \times r_2$ is a relation whose schema is the concatenation of R1 and R2. Relation R contains all tuples $t$ for which there is a tuple $t_1$ in $r_1$ and a tuple $t_2$ in $r_2$ for which $t[R1] = t_1[R1]$ and $t[R2] = t_2[R2]$.

Suppose that we want to find the names of all customers who have a loan at the Perryridge branch. We need the information in both the loan relation and the borrower relation to do so.

If we write

$$\sigma\text{branch-name ="Perryridge"}(\text{borrower} \times \text{loan})$$

then the result is the relation in table 2.4.

We get a relation that pertains to only the Perryridge branch. However, the customer-name column may contain customers.
<table>
<thead>
<tr>
<th>Customer-name</th>
<th>borrower loan-number</th>
<th>loan. Loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>L-11</td>
<td>Round Hill</td>
<td>900</td>
</tr>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>L-14</td>
<td>Downtown</td>
<td>1500</td>
</tr>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>L-17</td>
<td>Downtown</td>
<td>1000</td>
</tr>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>L-23</td>
<td>Redwood</td>
<td>2000</td>
</tr>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>L-93</td>
<td>Mianus</td>
<td>500</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>L-11</td>
<td>Round Hill</td>
<td>900</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>L-14</td>
<td>Downtown</td>
<td>1500</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>L-17</td>
<td>Downtown</td>
<td>1000</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>L-23</td>
<td>Redwood</td>
<td>2000</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>L-93</td>
<td>Mianus</td>
<td>500</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>L-11</td>
<td>Mianus</td>
<td>900</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>L-14</td>
<td>1500</td>
<td></td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>L-15</td>
<td>1500</td>
<td></td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>L-16</td>
<td>1300</td>
<td></td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>L-17</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>L-23</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>L-93</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>L-11</td>
<td>Round Hill</td>
<td>900</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>L-14</td>
<td>Downtown</td>
<td>1500</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>L-17</td>
<td>Downtown</td>
<td>1000</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>L-23</td>
<td>Redwood</td>
<td>2000</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>L-93</td>
<td>Mianus</td>
<td>500</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>L-11</td>
<td>Round Hill</td>
<td>900</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>L-14</td>
<td>Downtown</td>
<td>1500</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>L-17</td>
<td>Downtown</td>
<td>1000</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>L-23</td>
<td>Redwood</td>
<td>2000</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>L-93</td>
<td>Mianus</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 2.5 Result of borrower × loan
### Table 2.6 Result of $\sigma_{\text{branch-name} = \text{Perryridge}}$ (borrower $\times$ loan)

<table>
<thead>
<tr>
<th>Customer-name</th>
<th>borrower loan-number</th>
<th>Loan. Loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-14</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-14</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Jones</td>
<td>L-17</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Jones</td>
<td>L-17</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Smith</td>
<td>L-11</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Smith</td>
<td>L-11</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>L-15</td>
<td>Perryridge</td>
<td>1500</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>L-16</td>
<td>Perryridge</td>
<td>1300</td>
</tr>
</tbody>
</table>

• Since the Cartesian-product operation associates every tuple of loan with every tuple of borrower, we know that, if a customer has a loan in the Perryridge branch, then there is some tuple in borrower$\times$loan that contains his name, and borrower.loan-number = loan.loan-number.

• So, if we write $\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}}(\sigma_{\text{branch-name} = \text{Perryridge}}(\text{borrower} \times \text{loan}))$ we get only those tuples of borrower $\times$ loan that pertain to customers who have a loan at the Perryridge branch.

• Finally, since we want only customer-name, we do a projection:

$$
\Pi_{\text{customer-name}} \sigma_{\text{borrower.loan-number} = \text{loan.loan-number}}(\sigma_{\text{branch-name} = \text{Perryridge}}(\text{borrower} \times \text{loan}))
$$

• The result of this expression, shown in table given below, is the correct answer to our query.

<table>
<thead>
<tr>
<th>Customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
</tr>
<tr>
<td>Hayes</td>
</tr>
</tbody>
</table>

Table 2.7 Result of $\Pi_{\text{customer-name}}$
2.3.3 The Natural Join Operation

It is often desirable to simplify certain queries that require a Cartesian product.

- We first form the Cartesian product of the borrower and loan relations.
- Then, we select those tuples that pertain to only the same loan-number, followed by the projection of the resulting customer-name, loan-number, and amount:
  \[ \Pi_{\text{customer-name}, \text{loan-number}, \text{amount}} \sigma_{\text{borrower loan-number} = \text{loan loan-number}} (\text{borrower} \times \text{loan}) \]
- The natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation.
- It is denoted by the “join” symbol \( \times \). The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.
- Although the definition of natural join is complicated, the operation is easy to apply. As an illustration, consider again the example “Find the names of all customers who have a loan at the bank, and find the amount of the loan.”
- We express this query

<table>
<thead>
<tr>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayes</td>
</tr>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>Smith</td>
</tr>
</tbody>
</table>

Table 2.8 Customers with both an account and a loan at the bank

<table>
<thead>
<tr>
<th>customer-name</th>
<th>Loan-number</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>L-16</td>
<td>1300</td>
</tr>
<tr>
<td>Curry</td>
<td>L-93</td>
<td>500</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-15</td>
<td>1500</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-14</td>
<td>1500</td>
</tr>
<tr>
<td>Jones</td>
<td>L-17</td>
<td>1000</td>
</tr>
<tr>
<td>Smith</td>
<td>L-23</td>
<td>2000</td>
</tr>
<tr>
<td>Smith</td>
<td>L-11</td>
<td>900</td>
</tr>
<tr>
<td>Williams</td>
<td>L-17</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 2.9 Result of \( \Pi_{\text{customer-name}, \text{loan-number}, \text{amount}} (\text{borrower} \times \text{loan}) \)
by using the natural join as follows:

\[ \Pi_{\text{customer-name, loan-number, amount}} (\text{borrower} \times \text{loan}) \]

- Since the schemas for borrower and loan that is, Borrower-schema and Loan-schema have the attribute loan-number in common, the natural-join operation considers only pairs of tuples that have the same value on loan-number.
- It combines each such pair of tuples into a single tuple on the union of the two schemas that is, customer-name, branch-name, loan-number, and amount. After performing the projection, we obtain the relation in table shown above.
- Consider two relation schemas R and S which are, of course, lists of attribute names. If we consider the schemas to be sets, rather than lists, we can denote those attribute names that appear in both R and S by \( R \cap S \), and denote those attribute names that appear in R, in S, or in both by \( R \cup S \).
- Similarly, those attribute names that appear in R but not S are denoted by \( R - S \), whereas \( S - R \) denotes those attribute names that appear in S but not in R.
- Note that the union, intersection, and difference operations here are on sets of attributes, rather than on relations.
- We are now ready for a formal definition of the natural join. Consider two relations r(R) and s(S).
- The natural join of r and s, denoted by \( r \times s \), is a relation on schema \( R \cup S \) formally defined as follows:

\[ r \times s = \Pi_{R \cup S} (\sigma_{r.A_1 = s.A_1} \land r.A_2 = s.A_2 \land \ldots \land r.A_n = s.A_n \ r \times s) \] where \( R \cap S = \{A_1, A_2, \ldots, A_n\} \).

2.3.4 The Division Operation

The division operation, denoted by \( \div \), is suited to queries that include the phrase “for all.”
- Suppose that we wish to find all customers who have an account at all the branches located in Brooklyn.
- We can obtain all branches in Brooklyn by the expression

\[ r_1 = \Pi_{\text{branch-name}} (\sigma_{\text{branch-city} = \text{“Brooklyn”}} (\text{branch})) \]

- The result relation for this expression appears in table 2.10.

<table>
<thead>
<tr>
<th>branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
</tr>
<tr>
<td>Downtown</td>
</tr>
</tbody>
</table>

Table 2.10 Result of \( \Pi_{\text{branch-name}}(\sigma_{\text{branch-city} = \text{“Brooklyn”}} (\text{branch})) \)

- Now, we need to find customers who appear in \( r_2 \) with every branch name in \( r_1 \). The operation that provides exactly those customers is the divide operation. We formulate the query by writing

\[ \Pi_{\text{customer-name, branch-name}} (\text{depositor} \times \text{account}) \div \Pi_{\text{branch-name}} (\sigma_{\text{branch-city} = \text{“Brooklyn”}} (\text{branch})) \]

- The result of this expression is a relation that has the schema (customer-name) and that contains the tuple (Johnson).
- Formally, let r(R) and s(S) be relations, and let \( S \subseteq R \); that is, every attribute of schema S is also in schema R.
- The relation \( r \div s \) is a relation on schema \( R - S \) that is, on the schema containing all attributes of schema R that are not in schema S.
- A tuple t is in \( r \div s \) if and only if both of two conditions hold:
  - t is in \( PI_{R-S}(r) \)
  - For every tuple ts in s, there is a tuple tr in r satisfying both of the following:
    - \( tr[S] = ts[S] \)
    - \( tr[R - S] = t \)
It may surprise you to discover that, given a division operation and the schemas of the relations, we can, in fact, define the division operation in terms of the fundamental operations. Let \( r(R) \) and \( s(S) \) be given, with \( S \subseteq R \):

\[
r ÷ s = ΠR\setminusS (r) − ΠR\setminusS ((ΠR\setminusS (r) × s) − ΠR\setminusS, S(r))
\]

<table>
<thead>
<tr>
<th>customer-name</th>
<th>branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayes</td>
<td>Perryridge</td>
</tr>
<tr>
<td>Johnson</td>
<td>Downtown</td>
</tr>
<tr>
<td>Johnson</td>
<td>Brighton</td>
</tr>
<tr>
<td>Jones</td>
<td>Brighton</td>
</tr>
<tr>
<td>Lindsay</td>
<td>Redwood</td>
</tr>
<tr>
<td>Smith</td>
<td>Mianus</td>
</tr>
<tr>
<td>Turner</td>
<td>Round Hill</td>
</tr>
</tbody>
</table>

Table 2.11 Result of \( Π\text{customer-name, branch-name} \) (depositor × account)

- To see that this expression is true, we observe that \( ΠR\setminusS (r) \) gives us all tuples \( t \) that satisfy the first condition of the definition of division.
- The expression on the right side of the set difference operator
  \[
  ΠR\setminusS ((ΠR\setminusS (r) × s) − ΠR\setminusS, S(r))
  \]
  Serves to eliminate those tuples that fail to satisfy the second condition of the definition of division.
- Let us see how it does so. Consider \( ΠR\setminusS (r) × s \). This relation is on schema \( R \), and pairs every tuple in \( ΠR\setminusS (r) \) with every tuple in \( s \).
- The expression \( ΠR\setminusS,S(r) \) merely reorders the attributes of \( r \). Thus, \( (ΠR\setminusS (r) × s) − ΠR\setminusS, S(r) \) gives us those pairs of tuples from \( ΠR\setminusS (r) \) and \( s \) that do not appear in \( r \). If a tuple \( tj \) is in \( ΠR\setminusS ((ΠR\setminusS (r) × s) − ΠR\setminusS, S(r)) \), then there is some tuple \( ts \) in \( s \) that does not combine with tuple \( tj \) to form a tuple in \( r \). Thus, \( tj \) holds a value for attributes \( R \setminus S \) that does not appear in \( r ÷ s \). It is these values that we eliminate from \( ΠR\setminusS (r) \).

### 2.3.5 The Assignment Operation

The assignment operation, denoted by \( ← \), works like assignment in a programming language.

- To illustrate this operation, consider the definition of division.
  - We could write \( r ÷ s \) as \( \text{temp1} ← ΠR\setminusS (r) \)
    \[
    \text{temp2} ← ΠR\setminusS ((\text{temp1} × s) − ΠR\setminusS, S(r))
    \]
    \[
    \text{result} = \text{temp1} − \text{temp2}
    \]
- The evaluation of an assignment does not result in any relation being displayed to the user.
- Rather, the result of the expression to the right of the \( ← \) is assigned to the relation variable on the left of the \( ← \). This relation variable may be used in subsequent expressions.
- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.
- For relational-algebra queries, assignment must always be made to a temporary relation variable. Assignments to permanent relations constitute a database modification.
Summary

- Users of the database can create tables, insert new tables or modify existing tables. There are several languages for database programming.
- The relational model gives us a single way to represent data: as a two-dimensional table called a relation.
- A relational database consists of a collection of tables, each having a unique name.
- A row in a table represents a relationship among a set of values.
- The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result.
- The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection.
- The Cartesian-product operation, denoted by a cross (×), allows us to combine information from any two relations. We write the Cartesian product of relations $r_1$ and $r_2$ as $r_1 \times r_2$.
- The natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation.
- The division operation, denoted by $\div$, is suited to queries that include the phrase “for all.”
- The assignment operation, denoted by $\leftarrow$, works like assignment in a programming language.

References

- Books Llc, 2010. Relational Model: Relational Algebra, Relational Database Management System, Object-Relational Impedance Mismatch, Synonym, Codd’s Theorem, General Books LLC.

Recommended Reading

- Date, J. C., 2001. The database relational model: a retrospective review and analysis: a historical account and assessment of E.F. Codd’s contribution to the field of database technology, Addison-Wesley.
Self Assessment

1. Relational model is most widely used data model for commercial _________.
   a. data-structure
   b. relation
   c. data-processing
   d. unique name

2. The relational model gives us a single way to represent data as a two-dimensional table called a__________.
   a. relation
   b. data-structure
   c. data-processing
   d. unique name

3. A relational database consists of a collection of tables each having a__________.
   a. data-structure
   b. relation
   c. data-processing
   d. unique name

4. Which of the following statements is true?
   a. The relational algebra is a procedural coding language.
   b. The relational algebra is a procedural programming language.
   c. The relational algebra is a procedural query language.
   d. The relational algebra is a procedural SQL.

5. Which of the following is included in selection predicate?
   a. Comparisons between two attributes.
   b. Comparisons between three attributes.
   c. Comparisons between four attributes.
   d. Comparisons between five attributes.

6. The Cartesian-product operation, denoted by a cross ___________ allows to combine information from any two relations.
   a. ×
   b. +
   c. =
   d. ?

7. The natural join is a __________operation that allows us to combine certain selections and a Cartesian product into one operation.
   a. boolean
   b. binary
   c. mathematical
   d. logical
8. The ____________ operation, denoted by ÷, is suited to queries that include the phrase “for all.”
   a. division
   b. multiplication
   c. assignment
   d. addition

9. The ____________ operation, denoted by ←, works like assignment in a programming language.
   a. division
   b. multiplication
   c. assignment
   d. addition

10. The natural-join operation forms a Cartesian product of its ____________ arguments.
    a. two
    b. three
    c. four
    d. multiple
Chapter III

SQL

Aim

The aim of this chapter is to:

• explain the concept of SQL
• elucidate data definition language
• discuss concept of check (P)

Objectives

The objectives of this chapter are to:

• explain select clause
• discuss where clause
• describe from clause

Learning outcome

At the end of this chapter, you will be able to:

• recognise aggregate functions
• understand COUNT functions
• describe the MIN functions
3.1 Introduction to SQL

SQL is a tool for organising, managing, and retrieving data stored by a computer database. The name “SQL” is an abbreviation for Structured Query Language.

- As the name implies, SQL is a computer language that you use to interact with a database.
- In fact, SQL works with one specific type of database, called a relational database. Figure below shows how SQL works.
- The computer system in the figure has a database that stores important information. If the computer system is in a business, the database might store inventory, production, sales, or payroll data.
- On a personal computer, the database might store data about the checks you have written, lists of people and their phone numbers, or data extracted from a larger computer system.
- The computer program that controls the database is called a database management system, or DBMS.

![Fig. 3.1 SQL for database access](image)

- When you need to retrieve data from a database, you use the SQL language to make the request.
- DBMS processes the SQL request, retrieves the requested data, and returns it to you. This process of requesting data from a database and receiving back the results is called a database query hence the name Structured Query Language.
- SQL is thus a comprehensive language for controlling and interacting with a database management system.

3.2 Data Definition Language

We define an SQL relation by using the create table command:

```sql
create table r(A1D1,A2D2, ..., AnDn,
<integrity-constraint1>,
..., 
<integrity-constraintk>)
```

where r is the name of the relation, each Ai is the name of an attribute in the schema of relation r, and Di is the domain type of values in the domain of attribute Ai. The allowed integrity constraints include
Primary key (Aj1,Aj2, . . . , Ajm):

- The primary key specification says that attributes Aj1,Aj2, . . . , Ajm form the primary key for the relation.
- The primary key attributes are required to be non-null and unique; that is, no tuple can have a null value for a primary key attribute, and no two tuples in the relation can be equal on all the primary-key attributes.
- Although the primary key specification is optional, it is generally a good idea to specify a primary key for each relation.

Check (P)

The check clause specifies a predicate P that must be satisfied by every tuple in the relation.

- Code shown below presents a partial SQL DDL definition of our bank database.
- Note that, as in earlier chapters, we do not attempt to model precisely the real world in the bank-database example. In the real world, multiple people may have the same name, so customer-name would not be a primary key customer; a customer-id would more likely be used as a primary key.
- We use customer-name as a primary key to keep our database schema simple and short.
- If a newly inserted or modified tuple in a relation has null values for any primary key attribute or if the tuple has the same value on the primary-key attributes as does another tuple in the relation, SQL flags an error and prevents the update.
- Similarly, it flags an error and prevents the update if the check condition on the tuple fails.
- By default null is a legal value for every attribute in SQL, unless the attribute is specifically stated to be not null.
- An attribute can be declared to be not null in the following way:

```
account-number char(10) not null
create table customer
(customer-name char(20),
customer-street char(30),
customer-city char(30),
primary key (customer-name))
create table branch
(branch-name char(15),
branch-city char(30),
assets integer,
primary key (branch-name),
check (assets >= 0))
create table account
(account-number char(10),
branch-name char(15),
balance integer,
primary key (account-number),
check (balance >= 0))
create table depositor
(customer-name char(20),
account-number char(10),
primary key (customer-name, account-number))
```

- SQL also supports an integrity constraint unique (Aj1,Aj2, . . . , Ajm)
- The unique specification says that attributes Aj1,Aj2, . . . , Ajm form a candidate key; that is, no two tuples in the relation can be equal on all the primary-key attributes.
A common use of the check clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system. For instance, the check clause in the create table command for relation branch checks that the value of assets is nonnegative. As another example, consider the following:

```
create table student
(name char(15) not null,
 student-id char(10),
 degree-level char(15),
 primary key (student-id),
 check (degree-level in ('Bachelors', 'Masters', 'Doctorate')))```

Here, we use the check clause to simulate an enumerated type, by specifying that degree-level must be one of 'Bachelors', 'Masters', or 'Doctorate'.

We consider more general forms of check conditions, as well as a class of constraints called referential integrity constraints.

A newly created relation is empty initially. We can use the insert command to load data into the relation. Many relational-database products have special bulk loader utilities to load an initial set of tuples into a relation.

To remove a relation from an SQL database, we use the drop table command. The drop table command deletes all information about the dropped relation from the database. The command

```
drop table r```

is a more drastic action than

```
delete from r```

The latter retains relation r, but deletes all tuples in r. The former deletes not only all tuples of r, but also the schema for r.

After r is dropped, no tuples can be inserted into r unless it is re-created with the create table command.

We use the alter table command to add attributes to an existing relation.

All tuples in the relation are assigned null as the value for the new attribute.

The form of the alter table command is

```
alter table r add AD```

Where r is the name of an existing relation, A is the name of the attribute to be added, and D is the domain of the added attribute.

We can drop attributes from a relation by the command

```
alter table r drop A```

Where r is the name of an existing relation, and A is the name of an attribute of the relation.

Many database systems do not support dropping of attributes, although they will allow an entire table to be dropped.

### 3.3 Basic Structure of SQL Queries

Basic structure of an SQL expression consists of select, from and where clauses, select clause lists attributes to be copied - corresponds to relational algebra project from clause corresponds to Cartesian product - lists relations to be used, where clause corresponds to selection predicate in relational algebra.
**Select clause**
An example: Find the names of all branches in the account relation.

```sql
select bname
from account
```

- Distinct vs. all: elimination or not elimination of duplicates. Find the names of all branches in the account relation.

```sql
select distinct bname
from account
```

- By default, duplicates are not removed. We can state it explicitly using all.

```sql
select all bname
from account
```

- `select *` means select all the attributes. Arithmetic operations can also be in the selection list.

**Where clause**
- The predicates can be more complicated, and can involve
  - Logical connectives and, or and not.
  - Arithmetic expressions on constant or tuple values.
  - The between operator for ranges of values.
- Example: Find account number of accounts with balances between $90,000 and $100,000.

```sql
select account#
from account
where balance between 90000 and 100000
```

**From clause**
The from clause by itself defines a Cartesian product of the relations in the clause.
- SQL does not have a natural join equivalent. However, natural join can be expressed in terms of a Cartesian product, selection, and projection.
- For the relational algebra expression we can write in SQL,

```sql
select distinct cname, borrower.loan#
from borrower, loan
where borrower.loan# = loan.loan#
```

More selections with join:
“Find the names and loan numbers of all customers who have a loan at the SFU branch,” we can write in SQL,

```sql
select distinct cname, borrower.loan#
from borrower, loan
where borrower.loan# = loan.loan#
and bname='"SFU"
```
3.4 Aggregate Functions

SQL Aggregate Functions List describe you the Aggregate Function List Queries. The Aggregate Function includes the average, count, min, max, sum etc queries.

Understand with Example

The Tutorial illustrates an example from SQL Aggregate Functions List. To understand the example we create a table ‘stu’ with required fieldnames and datatypes.

Create Table Stu:
create table Stu(Id varchar(2), Name varchar(15), Class varchar(10), sub_id varchar(2), marks varchar(3));

Insert data into Stu:
The insert into is used to add the records or rows to the table ‘Stu’.
insert into Stu values(1,'Komal',10,1,45);
insert into Stu values(2,'Ajay',10,1,56);
insert into Stu values(3,'Rakesh',10,1,67);
insert into Stu values(4,'Santosh',10,1,67);
insert into Stu values(1,'Komal',10,2,47);
insert into Stu values(2,'Ajay',10,2,53);
insert into Stu values(3,'Rakesh',10,2,57);
insert into Stu values(4,'Santosh',10,2,67);
insert into Stu values(5,'Bhanu',10,2,67);
insert into Stu values(1,'Komal',10,3,45);
insert into Stu values(2,'Ajay',10,3,56);
insert into Stu values(3,'Rakesh',10,3,67);
insert into Stu values(4,'Santosh',10,3,67);
insert into Stu values(5,'Bhanu',10,3,67);
insert into Stu values(1,'Komal',10,4,65);
insert into Stu values(2,'Ajay',10,4,56);
insert into Stu values(3,'Rakesh',10,4,37);
insert into Stu values(4,'Santosh',10,4,67);
insert into Stu values(5,'Bhanu',10,2,67);
insert into Stu values(1,'Komal',10,5,65);
insert into Stu values(2,'Ajay',10,5,46);
insert into Stu values(3,'Rakesh',10,5,63);
insert into Stu values(4,'Santosh',10,5,67);
insert into Stu values(5,'Bhanu',10,2,67);
insert into Stu values(1,'Komal',10,1,45);
insert into Stu values(2,'Ajay',10,1,56);
insert into Stu values(3,'Rakesh',10,1,67);
insert into Stu values(4,'Santosh',10,1,67);
insert into Stu values(1,'Komal',10,2,47);
insert into Stu values(2,'Ajay',10,2,53);
insert into Stu values(3,'Rakesh',10,2,57);
insert into Stu values(4,'Santosh',10,2,67);
insert into Stu values(5,'Bhanu',10,2,67);
insert into Stu values(1,'Komal',10,3,45);
insert into Stu values(2,'Ajay',10,3,56);
insert into Stu values(3,'Rakesh',10,3,67);
insert into Stu values(4,'Santosh',10,3,67);
insert into Stu values(5,'Bhanu',10,2,67);
insert into Stu values(1,'Komal',10,4,65);
insert into Stu values(2,'Ajay',10,4,56);
insert into Stu values(3,'Rakesh',10,4,37);
insert into Stu values(4,'Santosh',10,4,67);
insert into Stu values(5,'Bhanu',10,2,67);
insert into Stu values(1,'Komal',10,5,65);
insert into Stu values(2,'Ajay',10,5,46);
insert into Stu values(3,'Rakesh',10,5,63);
insert into Stu values(4,'Santosh',10,5,67);

**AVG functions:**
The AVG functions return the average sum of record marks as fieldname marks, id and name from table ‘stu’. The group by clause returns you the unique records from table ‘stu’.
mysql> select id, name, avg(marks) as 'avg marks' from stu group by id;

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>avg marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Komal</td>
<td>53.4</td>
</tr>
<tr>
<td>2</td>
<td>Ajay</td>
<td>53.4</td>
</tr>
<tr>
<td>3</td>
<td>Rakesh</td>
<td>58.2</td>
</tr>
<tr>
<td>4</td>
<td>Santosh</td>
<td>67</td>
</tr>
<tr>
<td>5</td>
<td>Bhanu</td>
<td>67</td>
</tr>
</tbody>
</table>

5 rows in set (0.00 sec)

COUNT functions:
The count (id) function is a aggregate function that return the sum number of records matched in the table stu that meet the specified criteria.

mysql> select id, name, count(id) as 'paper' from stu group by id;

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Komal</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Ajay</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Rakesh</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Santosh</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Bhanu</td>
<td>3</td>
</tr>
</tbody>
</table>

5 rows in set (0.00 sec)

SQL Aggregate Functions List
SQL Aggregate Functions List describe you the Aggregate Function List Queries. The Aggregate Function includes the average, count, min, max, sum etc queries.

MIN functions:
The MIN functions return the minimum value of the records mark from table stu.

mysql> select id, name, min(marks) as 'Min marks in sub' from stu group by id;

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>Min marks in sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Komal</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>Ajay</td>
<td>46</td>
</tr>
<tr>
<td>3</td>
<td>Rakesh</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>Santosh</td>
<td>67</td>
</tr>
<tr>
<td>5</td>
<td>Bhanu</td>
<td>67</td>
</tr>
</tbody>
</table>

5 rows in set (0.00 sec)
**MAX functions**
The MAX Functions is used to return the maximum records value of marks from table stu. The Group by Id return you the unique id records.

```sql
mysql> select id, name, max(marks) as 'Max marks in sub' from stu  group by id;
```

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>Max marks in sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Komal</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>Ajay</td>
<td>46</td>
</tr>
<tr>
<td>3</td>
<td>Rakesh</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>Santosh</td>
<td>67</td>
</tr>
<tr>
<td>5</td>
<td>Bhanu</td>
<td>67</td>
</tr>
</tbody>
</table>

5 rows in set (0.00 sec)

**SUM functions**
The SUM functions is used to return the sum of total marks from table ‘stu’.

```sql
mysql> select id, name, sum(marks) as 'total marks' from stu  group by id;
```

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>Total Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Komal</td>
<td>267</td>
</tr>
<tr>
<td>2</td>
<td>Ajay</td>
<td>267</td>
</tr>
<tr>
<td>3</td>
<td>Rakesh</td>
<td>291</td>
</tr>
<tr>
<td>4</td>
<td>Santosh</td>
<td>268</td>
</tr>
<tr>
<td>5</td>
<td>Bhanu</td>
<td>201</td>
</tr>
</tbody>
</table>

5 rows in set (0.00 sec)

### 3.5 Nested Sub-queries
A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery. A subquery can return a set of rows or just one row to its parent query. A scalar subquery is a query that returns exactly one value: a single row, with a single column. Scalar subqueries can be used in most places in a SQL statement where you could use an expression or a literal value.

The places in a query where a subquery may be used are as follows:
- In the SELECT list used for column projection
- In the FROM clause
- In the WHERE clause
- In the HAVING clause
A subquery is often referred to as an inner query, and the statement within which it occurs is then called the outer query. There is nothing wrong with this terminology, except that it may imply that you can only have two levels, inner and outer. In fact, the Oracle implementation of subqueries does not impose any practical limits on the level of nesting: the depth of nesting permitted in the FROM clause of a statement is unlimited and that in the WHERE clause is up to 255.

A subquery can have any of the usual clauses for selection and projection. Following are the required clauses:

- A SELECT list
- A FROM clause

Following are the optional clauses:

- WHERE
- GROUP BY
- HAVING

The subquery (or subqueries) within a statement must be executed before the parent query that calls it, so that the results of the subquery can be passed to the parent.

**Types of subqueries**

In this exercise, you will write code that demonstrates the places where subqueries can be used. Use either SQL*Plus or SQL Developer. All the queries should be run when connected to the HR schema.

- Log on to your database as user HR.
- Write a query that uses subqueries in the column projection list. The query will report on the current numbers of departments and staff:

```
select sysdate Today,
(select count(*) from departments) Dept_count,
(select count(*) from employees) Emp_count
from dual;
```

- Write a query to identify all the employees who are managers. This will require using a subquery in the WHERE clause to select all the employees whose EMPLOYEE_ID appears as a MANAGER_ID:

```
select last_name from employees where
(employee_id in (select manager_id from employees));
```

- Write a query to identify the highest salary paid in each country. This will require using a subquery in the FROM clause:

```
select max(salary),country_id from (select salary,department_id,location_id,country_id from employees natural join departments natural join locations) group by country_id;
```
Using subqueries in SQL

There are many situations where you will need the result of one query as the input for another.

Use of a subquery result set for comparison purposes

- Which employees have a salary that is less than the average salary?
- This could be answered by two statements, or by a single statement with a subquery.
- The following example uses two statements:

```sql
select avg(salary) from employees;
select last_name from employees where salary < result_of_previous_query ;
```

- Alternatively, this example uses one statement with a subquery:

```sql
select last_name from employees where salary < (select avg(salary) from employees);
```

- In this example, the subquery is used to substitute a value into the WHERE clause of the parent query: it is returning a single value, used for comparison with the rows retrieved by the parent query.
- The subquery could return a set of rows. For example, you could use the following to find all departments that do actually have one or more employees assigned to them:

```sql
select department_name from departments where department_id in (select distinct(department_id) from employees);
```

- In the preceding example, the subquery is used as an alternative to a join. The same result could have been achieved with the following:

```sql
select department_name from departments inner join employees on employees.department_id = departments.department_id group by department_name;
```

- If the subquery is going to return more than one row, then the comparison operator must be able to accept multiple values. These operators are IN, NOT IN, ANY, and ALL.
- If the comparison operator is EQUAL, GREATER THAN, or LESS THAN (which each can only accept one value), the parent query will fail.

Star transformation

- An extension of the use of subqueries as an alternative to a join is to enable the star transformation often needed in data warehouse applications.
- Consider a large table recording sales. Each sale is marked as being of a particular product to a particular buyer through a particular channel.
- These attributes are identified by codes, used as foreign keys to dimension tables with rows that describe each product, buyer, and channel.
- To identify all sales of books to buyers in Germany through Internet orders, one could run a query like this:
This query uses the WHERE clause to join the tables and then to filter the results. The following is an alternative query that will yield the same result:

```
select … from sales
where prod_code in (select prod_code from products where product='Books')
and buy_code in (select buy_code from buyers where country='Germany') and chan_code in (select chan_code from channels where channel='Internet');
```

The rewrite of the first statement to the second is the star transformation. Apart from being an inherently more elegant structure most SQL developers with any sense of aesthetics will agree with that, there are technical reasons why the database may be able to execute it more efficiently than the original query. Also, star queries are easier to maintain; it is very simple to add more dimensions to the query or to replace the single literals ‘Books,’ ‘Germany,’ and ‘Internet’ with lists of values.

**Generate a table from which to SELECT**

Subqueries can also be used in the FROM clause, where they are sometimes referred to as inline views.

Consider another problem based on the HR schema: employees are assigned to a department, and departments have a location.

Each location is in a country. How can you find the average salary of staff in a country, even though they work for different departments?

```
select avg(salary),country_id from (select salary,department_id,location_id,country_id from employees natural join departments natural join locations) group by country_id;
```

The subquery constructs a table with every employee’s salary and the country in which his department is based.

The parent query then addresses this table, averaging the SALARY and grouping by COUNTRY_ID.

**Generate values for projection**

The third place a subquery can go is in the SELECT list of a query.

How can you identify the highest salary and the highest commission rate and thus what the maximum commission paid would be if the highest salaried employee also had the highest commission rate?

Like this, with two subqueries:

```
select (select max(salary) from employees) * (select max(commission_pct) from employees) / 100 from dual;
```
• In this usage, the SELECT list used to project columns is being populated with the results of the subqueries.
• A subquery used in this manner must be scalar, or the parent query will fail with an error.

**Generate rows to be passed to a DML statement**

- Consider the following examples:

```sql
insert into sales_hist select * from sales where date > sysdate-1;
update employees set salary = (select avg(salary) from employees);
delete from departments
where department_id not in (select department_id from employees);
```

- The first example uses a subquery to identify a set of rows in one table that will be inserted into another.
- The second example uses a subquery to calculate the average salary of all employees and passes this value a scalar quantity to an update statement.
- The third example uses a subquery to retrieve all DEPARTMENT_IDs that are in use and passes the list to a DELETE command, which will remove all departments that are not in use.
- Note that it is not legal to use a subquery in the VALUES clause of an insert statement; this is fine:

```sql
insert into dates select sysdate from dual;
But this is not:
insert into dates (date_col) values (select sysdate from dual);
```

**List the types of SQL subqueries**

There are three broad divisions of subquery:

- Single-row subqueries
- Multiple-row subqueries
- Correlated subqueries

**Single- and multiple-row subqueries**

- The single-row subquery returns one row. A special case is the scalar subquery, which returns a single row with one column. Scalar subqueries are acceptable and often very useful in virtually any situation where you could use a literal value, a constant, or an expression.
- Multiple-row subqueries return sets of rows. These queries are commonly used to generate result sets that will be passed to a DML or SELECT statement for further processing.
- Both single-row and multiple-row subqueries will be evaluated once, before the parent query is run.
- Single- and multiple-row subqueries can be used in the WHERE and HAVING clauses of the parent query, but there are restrictions on the legal comparison operators.
- If the comparison operator is any of the ones in the following table, the subquery must be a single-row subquery:
If any of the operators in the preceding table are used with a subquery that returns more than one row, the query will fail.

The operators in the following table can use multiple-row subqueries:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>equal to any member in a list</td>
</tr>
<tr>
<td>NOT IN</td>
<td>not equal to any member in a list</td>
</tr>
<tr>
<td>ANY</td>
<td>returns rows that match any value on a list</td>
</tr>
<tr>
<td>ALL</td>
<td>returns rows that match all the values in a list</td>
</tr>
</tbody>
</table>

Correlated subqueries

- A correlated subquery has a more complex method of execution than single and multiple-row subqueries and is potentially much more powerful.
- If a subquery references columns in the parent query, then its result will be dependent on the parent query. This makes it impossible to evaluate the subquery before evaluating the parent query.

Consider this statement, which lists all employees who earn less than the average salary:

```sql
select last_name from employees where salary < (select avg(salary) from employees);
```

The single-row subquery need to be executed only once, and its result is substituted into the parent query. But now consider a query that will list all employees whose salary is less than the average salary of their department. In this case, the subquery must be run for each employee to determine the average salary for her department; it is necessary to pass the employee’s department code to the subquery. This can be done as follows:

```sql
select p.last_name, p.department_id from employees p where p.salary < (select avg(s.salary) from employees s where s.department_id=p.department_id);
```

- In this example, the subquery references a column, p.department_id, from the select list of the parent query. This is the signal that, rather than evaluating the subquery once, it must be evaluated for every row in the parent query.
- To execute the query, Oracle will look at every row in EMPLOYEES and, as it does so, run the subquery using the DEPARTMENT_ID of the current employee row.
The flow of execution is as follows:
Step 1: Start at the first row of the EMPLOYEES table.
Step 2: Read the DEPARTMENT_ID and SALARY of the current row.
Step 3: Run the subquery using the DEPARTMENT_ID from step 2.
Step 4: Compare the result of step 3 with the SALARY from step 2.
Step 5: Return the row if the SALARY is less than the result.
Step 6: Advance to the next row in the EMPLOYEES table.
Step 7: Repeat from step 2.

- A single-row or multiple-row subquery is evaluated once, before evaluating the outer query. A correlated subquery must be evaluated once for every row in the outer query.
- A correlated subquery can be single or multiple-row, if the comparison operator is appropriate.

### 3.6 Join Relations

Joins are one of the basic constructions of SQL and Databases as such they combine records from two or more database tables into one row source, one set of rows with the same columns. These columns can originate from either of the joined tables as well as be formed using expressions and built-in or user-defined functions.

- Depending on join type and join restrictions, returned row count can be from 0 till all possible combinations of involved tables. So if one has two tables containing 10 000 rows each then, the maximum number of resultant rows can be $10 000 \times 10 000 = 100 000 000$ rows.
- Databases are built to make joins as efficient as possible. It means always joining data in database is more efficient than doing that somewhere else.
- It also means that one has to know the power and possibilities of joins to fully exploit their strength. And it doesn’t matter whether one is using Oracle, SQL Server, MySQL, IBM DB2, PostgreSQL or whatever else DBMS.
- It is worth to mention that throughout this entire chapter it is supposed two tables are joined together.
- It is used both for theoretical parts of it as well as examples. However, one has to remember that every two row sources can be joined together, and both or one of them can be table, view, subquery, materialised view or other construction returning rows.
- All examples are created for Oracle database and written according to Oracle syntax. However, it doesn’t matter what database management system is used, many of them with very little modifications or even exactly the same can be used for every other DBMS supporting joins.
- Exactly why they work or why not are described for Oracle, SQL Server and MySQL.

**Classification schemes**

- There are different classification schemes and different criteria according to what joins are classified.
- As a result there is a bit mess in the process of understanding them. Here it has not found a nice scheme or even textual description how each one of various classifications schemes and join types relates to other.
- In the following Meta model there is one possible variant of that, however theoreticians would break lances around other possible classification schemes.
Depending on condition existence
Depending on whether we add any join condition or not there are following join types:

CROSS JOIN (synonyms also CARTESIAN JOIN, CARTESIAN PRODUCT):
- There is not any join condition or it is always true.
- All other join types degrade to CROSS JOINS as soon as join condition (-s) is (are) always true.

JOIN WITH RESTRICTION
There is applied join condition to joined tables. One can write join with restriction using different syntactic notations. As already said above every join with restriction may degrade to cross join.

NATURAL JOIN
Syntactic notation joining source tables on all columns having the same name. This can be quite dangerous as explained below in the chapter for Natural join. Natural joins always are Equi joins.

QUALIFIED JOIN
User has possibility to define which columns are used in join condition.
**NAMED COLUMNS JOIN**
Syntactic notation joining source tables on user defined columns having the same name. This is less dangerous than Natural join and just short form of writing Equi joins on some common columns joined together. Named columns joins always are Equi joins.

**CONDITIONAL JOIN**
Fully controllable syntax by user. This is the most widespread and most useful syntactic convention. Depending on used predicates in join condition it may be Equi join as well as Non-equi join.

**Depending on row selection**
Depending on whether only rows satisfying join condition are selected or all rows are selected in one or both involved tables, joins are divided:

**INNER JOIN**
Only rows satisfying selection criteria from both joined tables are selected.

**LEFT OUTER JOIN**
Rows satisfying selection criteria from both joined tables are selected as well as all remaining rows from left joined table are being kept along with Nulls instead of actual right joined table values.

**RIGHT OUTER JOIN**
Rows satisfying selection criteria from both joined tables are selected as well as all remaining rows from right joined table are being kept along with Nulls instead of actual left joined table values.

**FULL OUTER JOIN**
Rows satisfying selection criteria from both joined tables are selected as well as all remaining rows both from left joined table and right joined table are being kept along with Nulls instead of values from other table.

**Depending on comparison operator**
Depending on used comparison operator in join condition there are following join types:

**PREDICATE OPERATOR TYPE**
Based on predicate operator type (i.e. equality and everything other) joins are divided into two parts Equi joins and Nonequi joins.

**EQUI JOIN**
Join condition uses only equality predicate “=”. It can be both explicit for Conditional join and implicit for Natural join and Named columns join.

**THETA (NONEQUI) JOIN** - everything other than equality predicate “=”, for example “>=”, between. This can be used only by Conditional joins.

Depending on used tables there is one special case for classification based on what tables are involved in join orange in model.

**SELF JOIN**
Table is joined to itself.
**Classification schemes summary**

There are some interesting facts which we observe from the diagram above:

- Every join either is Cartesian join or can degrade to Cartesian join in case the join criteria always are true.
- Each and every join type, be it inner, outer, self, natural, equi, non-equi, named column or join with explicitly declared join condition can degrade to Cartesian join.
- Classification scheme Depending on row existence i.e., inner and outer joins are absolutely independent of other classification schemes, e.g. natural joins can be both inner or outer, as well as self join can be inner or outer.
- Every other join type can be self join if both joined tables are the same. Equi join - although most used one - is not the only one.
- It is the only one if natural join or named columns join syntax is used, but for explicitly specified join condition other comparison operators besides equality ("=") can be used.

**Used data model for examples**

Throughout this entire article we will use following table names according to my favourite naming conventions and table data:

```sql
CREATE TABLE addresses (  
    adr_id INTEGER NOT NULL PRIMARY KEY,  
    adr_city VARCHAR(15),  
    adr_country VARCHAR(15) NOT NULL);
CREATE TABLE persons (  
    prs_id INTEGER NOT NULL PRIMARY KEY,  
    prs_father_id INTEGER,  
    prs_mother_id INTEGER,  
    prs_adr_id INTEGER,  
    prs_first_name VARCHAR(15),  
    prs_surname VARCHAR(15),  
    CONSTRAINT prs_prs_father_fk FOREIGN KEY (prs_father_id)  
      REFERENCES persons(prs_id),  
    CONSTRAINT prs_prs_mother_fk FOREIGN KEY (prs_mother_id)  
      REFERENCES persons(prs_id),  
    CONSTRAINT prs_adr_fk FOREIGN KEY (prs_adr_id)  
      REFERENCES addresses(adr_id));
INSERT INTO addresses VALUES (1, 'RIGA', 'LATVIA');  
INSERT INTO addresses VALUES (2, 'BERLIN', 'GERMANY');  
INSERT INTO addresses VALUES (3, 'NEW YORK', 'USA');  
INSERT INTO persons VALUES (1, NULL, NULL, NULL, 'JANIS', 'BERZINS');  
INSERT INTO persons VALUES (2, 1, NULL, 2, 'PETER', 'BERZINS');  
INSERT INTO persons VALUES (3, NULL, NULL, 2, 'ANN', 'SMYTH');  
INSERT INTO persons VALUES (4, 2, 3, 2, 'CHARLES', 'BERZINS');
COMMIT;
```

Although data model is very simple, it is very important for you to understand the data for examples, because all the examples are as important as theory. So the data model is as follows:
And data looks as follows:

```
SQL> SELECT * FROM persons;
PRS_ID  PRS_FATHER_ID  PRS_MOTHER_ID  PRS_ADR_ID  PRS_FIRST_NAME  PRS_SURNAME
---------  -------------  -------------  ----------  --------------  -----------
1             1                    2 PETER       BERZINS
2             1                    2 ANN         SMYTH
3             2             3          2 CHARLES     BERZINS

SQL> SELECT * FROM addresses;
ADR_ID  ADR_CITY       ADR_COUNTRY
---------  ---------------  -----------
1     RIGA           LATVIA
2    BERLIN         GERMANY
3    NEW YORK       USA
```

**Cross joins**
Cross join consists of all possible combinations of two table’s rows. Imagine we have two tables Table A and Table B with 4 and 3 rows respectively as follows:
Cross join of both tables is as follows:

It is obvious that cross join of rather small tables may lead to quite disastrous resultant record count, for example, two tables of just 100 thousand rows each results in $100,000 \times 100,000 = 10,000,000,000$ (10 billion rows).
Syntax
There are two syntax types how to normally get cross join. One explicitly declares cross join:

```sql
SELECT <column list>
FROM <left joined table>
CROSS JOIN <right joined table>
```

Second syntax type is functionally the same; just both tables are written in FROM clause:

```sql
SELECT <column list>
FROM <left joined table>, <right joined table>
```

- As one can see, second query hasn’t WHERE clause, it means as soon as one omits it, and uses more than one table in FROM clause, it results in cross join and possibly very many rows.

Facts to remember
Usually cross joins are used quite rarely; some of the scenarios could be as follows:
- Possibility to generate high amount of rows. As we can see from relatively small tables there is possibility to get quite monstrous numbers.
- Find out all possible row combinations of some tables. Mostly this is useful for reports where one needs to generate all combinations for example all nationalities x genders for persons.
- To join a table with just one row. Most often used to get some configuration parameters.

3.7 Integrity Constraints
- Integrity constraints are the rules that restrict the values for one or more columns in a table.
- Constraint clauses can appear in either CREATE TABLE or ALTER TABLE statements, and identify the column or columns affected by the constraint and identify the conditions of the constraint.

Integrity Constraint States
One can specify that a constraint is enabled (ENABLE) or disabled (DISABLE).
- If a constraint is enabled, data is checked as it is entered or updated in the database, and data that does not conform to the constraint is prevented from being entered.
- If a constraint is disabled, then data that does not conform can be allowed to enter the database.
- Additionally, you can specify that existing data in the table must conform to the constraint (VALIDATE).
- Conversely, if you specify NOVALIDATE, you are not ensured that existing data conforms.

An integrity constraint defined on a table can be in one of the following states:
- ENABLE, VALIDATE
- ENABLE, NOVALIDATE
- DISABLE, VALIDATE
- DISABLE, NOVALIDATE

For details about the meaning of these states and an understanding of their consequences, see the Oracle Database SQL Language Reference. Some of these consequences are discussed here.
Disabling constraints
To enforce the rules defined by integrity constraints, the constraints should always be enabled.

- It is consider temporarily disabling the integrity constraints of a table for the following performance reasons:
  - When loading large amounts of data into a table
  - When performing batch operations that make massive changes to a table for example, changing every employee’s number by adding 1000 to the existing number.
  - When importing or exporting one table at a time
- In all three cases, temporarily disabling integrity constraints can improve the performance of the operation, especially in data warehouse configurations.
- It is possible to enter data that violates a constraint while that constraint is disabled. Thus, you should always enable the constraint after completing any of the operations listed in the preceding bullet list.

Enabling constraints
- Constraint is enabled; no row violating the constraint can be inserted into the table. However, while the constraint is disabled such a row can be inserted. This row is known as an exception to the constraint.
- If the constraint is in the enable novalidated state, violations resulting from data entered while the constraint was disabled remain.
- The rows that violate the constraint must be either updated or deleted in order for the constraint to be put in the validated state.
- One can identify exceptions to a specific integrity constraint while attempting to enable the constraint.
- All rows violating constraints are noted in an EXCEPTIONS table, which one can examine.

Enable novalidate constraint state
When a constraint is in the enable novalidate state, all subsequent statements are checked for conformity to the constraint. However, any existing data in the table is not checked.

- A table with enable novalidated constraints can contain invalid data, but it is not possible to add new invalid data to it.
- Enabling constraints in the novalidated state is most useful in data warehouse configurations that are uploading valid OLTP data.
- Enabling a constraint does not require validation.
- Enabling a constraint novalidate is much faster than enabling and validating a constraint. Also, validating a constraint that is already enabled does not require any DML locks during validation unlike validating a previously disabled constraint.
- Enforcement guarantees that no violations are introduced during the validation. Hence, enabling without validating enables you to reduce the downtime typically associated with enabling a constraint.

Efficient use of integrity constraints: a procedure
Using integrity constraint stated in the following order can ensure the best benefits:

- Disable state
- Perform the operation (load, export, import)
- Enable no validate state
- Enable state
Some benefits of using constraints in this order are:

- No locks are held
- All constraints can go to enable state concurrently
- Constraint enabling is done in parallel
- Concurrent activity on table is permitted
- Setting Integrity Constraints Upon Definition

When an integrity constraint is defined in a CREATE TABLE or ALTER TABLE statement, it can be enabled, disabled, or validated as determined by your specification of the ENABLE/DISABLE clause. If the ENABLE/DISABLE clause is not specified in a constraint definition, the database automatically enables and validates the constraint.

**Disabling constraints upon definition**
The following CREATE TABLE and ALTER TABLE statements both define and disable integrity constraints:

```sql
CREATE TABLE emp (  
  empno NUMBER(5) PRIMARY KEY DISABLE, . . . ; 
ALTER TABLE emp 
ADD PRIMARY KEY (empno) DISABLE;
```

An ALTER TABLE statement that defines and disables an integrity constraint never fails because of rows in the table that violate the integrity constraint. The definition of the constraint is allowed because its rule is not enforced.

**Enabling constraints upon definition**
The following CREATE TABLE and ALTER TABLE statements both define and enable integrity constraints:

```sql
CREATE TABLE emp  
( empno NUMBER(5) CONSTRAINT emp.pk PRIMARY KEY , . . . ; 
ALTER TABLE emp  
ADD CONSTRAINT emp.pk PRIMARY KEY (empno);
```

- An ALTER TABLE statement which defines and attempts to enable an integrity constraint can fail because rows of the table violate the integrity constraint.
- If this case, the statement is rolled back and the constraint definition is not stored and not enabled.
- When you enable a UNIQUE or PRIMARY KEY constraint an associated index is created.

### 3.8 Authorisation

Use the derby. database.sqlAuthorisation property to enable SQL standard authorisation.

- The derby. database.sqlAuthorisation property controls the ability for object owners to grant and revoke permission for users to perform actions on database objects. It also controls the ability for users to create, set, and drop roles.
- The valid settings for the derby.database.sqlAuthorisation property are:
  - TRUE
  - FALSE
- The default setting for the derby.database.sqlAuthorisation property is FALSE.
- The derby.database.sqlAuthorisation property is usable only if the property derby.connection.requireAuthentication is also set to true, since SQL authorisation is of no value unless authentication is also enabled.
After you set the `derby.database.sqlAuthorisation` property to TRUE, you cannot set the property back to FALSE.

You can set the `derby.database.sqlAuthorisation` property as a system property or as a database property. If you set this property as a system property before you create the databases, all new databases will automatically have SQL authorisation enabled. If the databases already exists, you can set this property only as a database property.

To enable SQL standard authorisation for the entire system, set the `derby.database.sqlAuthorisation` property as a system property:

```
derby.database.sqlAuthorisation=true
```

To enable SQL standard authorisation for a specific database, set the `derby.database.sqlAuthorisation` property as a database property:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY( 'derby.database.sqlAuthorisation', 'true')
```
Summary

- The entity-relationship (ER) data model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design.
- An entity is an object in the real world that is distinguishable from other objects. It is often useful to identify a collection of similar entities. Such a collection is called an entity set.
- A relationship is an association among two or more entities. For example, we may have the relationship that Attishoo works in the pharmacy department.
- An E-R enterprise schema may define certain constraints to which the contents of a database must conform.
- Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.
- The participation of an entity set E in a relationship set R is said to be total if every entity in E participates in at least one relationship in R. If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be partial.
- Entity Relationship Diagrams are a major data modelling tool and will help organise the data in your project into entities and define the relationships between the entities.
- A data entity is anything real or abstract about which we want to store data. Entity types fall into five classes: roles, events, locations, tangible things or concepts.
- A data attribute is a characteristic common to all or most instances of a particular entity.
- As with many data modelling tools there are a number of different styles used to create. The style used in this course will be the one labelled “Information Engineering”.
- Relationship is a data relationship is a natural association that exists between one or more entities.

References


Recommended Reading

Self Assessment

1. Which of the following statements is true?
   a. The single-row subquery returns one row.
   b. The multiple-row subquery returns one row.
   c. The single-row subquery returns void row.
   d. The single-row subquery returns two row.

2. Which of the following returns a row?
   a. Single-row
   b. Multiple-row
   c. Mix-row
   d. Mega-row

3. Every join with restriction may degrade to_________.
   a. last-join
   b. end-join
   c. cross-join
   d. plus-join

4. ________ constraints are the rules that restrict the values for one or more columns in a table.
   a. Integrity
   b. Novalidate
   c. Validate
   d. Efficient

5. ________ clauses can appear in either CREATE TABLE or ALTER TABLE statements.
   a. Integrity
   b. Where
   c. When
   d. Constraint

6. Enabling a constraint does not require_________.
   a. query
   b. validation
   c. value
   d. rules

7. If the __________ clause is not specified in a constraint definition, the database automatically enables and validates the constraint.
   a. ENABLE/DISABLE
   b. WHER
   c. SELECT
   d. REMOVE
8. automatically enables and validates the constraint.
   a. Database
   b. Integrity
   c. Efficiency
   d. Select

9. Use the derby.database.sqlAuthorisation property to enable _______ standard authorisation.
   a. DLL
   b. DLM
   c. SQL
   d. C

10. _______ a constraint novalidate is much faster than enabling and validating a constraint.
    a. Enabling
    b. Disabling
    c. Validating
    d. Selecting
Chapter IV
Database Design and ER-Model

Aim
The aim of this chapter is to:

- explain the design process
- elucidate mission statement
- discuss mission objectives

Objectives
The objectives of this chapter are to:

- analyse current database
- explain creation of data structure
- determine business rule

Learning outcome
At the end of this chapter, you will be able to:

- understand data integrity
- classify entity relationship model
- describe constraint
4.1 Overview of Design Process

There are seven phases of database design and they are as follows:

Phase 1: Define a mission statement and mission objectives

The mission statement

Conduct an interview with the owner or manager or the organisation. Establish the purpose of the database. Start the process by defining the end result. A well written mission statement is free of any phrases or sentences that explicitly describe specific tasks.

The purpose of the ABC company database is to maintain the data we use and provide the information we need to run the day-to-day affairs of our business.

- How would you describe the purpose of your organisation to a new client?
- What would you say is the purpose of your organisation?
- What is the major function of your organisation?
- How would you describe what your organisation does?
- Will you define the single most important reason for the existence of your organisation?
- What is the main focus of your organisation?

The mission objectives

Prepare statements that represent the general tasks to be performed against the data collected in the database. This determines the subjects that each of the tables represent. The purpose of a mission objective is to help define various structures within the database and to help guide the overall direction of the database’s development. A mission objective is a declarative sentence that clearly defines a general task and is free from unnecessary details. If a mission objective represents more than one general task, it should be broken into two or more mission objectives.

Examples of mission objectives:

- We need to maintain complete patient address information
- We need to keep track of all customer sales
- We need to make sure an account representative is responsible for no more than twenty accounts at any given time.
- We need to keep track of vehicle maintenance
- We need to produce employee phone directories

Phase 1: Types of questions to ask

- What kind of work do you perform on a daily basis?
- How would you define your job description?
- What kind of data do you work with?
- What types of reports do you generate?
- What types of things do you keep track of?
- What types of services does your organisation provide?
- How would you describe the type of work that you do?

Phase 2: Analyse the current database

- Review the way data is currently collected and presented.
- Conduct interviews with users and management to identify how they interact with the database on a daily basis.
- Prepare a list of fields and calculations using the information from the analysis and the interviews.
Phase 3: Create the data structures
- Define tables by identifying the subjects that will be tracked by the database.
- Assign each table, fields that best characterise its subject.
- Establish primary key
- Define field specifications for every field (conduct interviews with users and management to help identify any specific field characteristics that may be important to them.
- Modify multi-valued fields so that each field stores only a single value

Phase 4: Determine and establish table relationships
- Conduct interviews with users and management to identify relationships, relationship characteristics and establish relationship level integrity.
- Once relationships are identified, establish the logical connection for each relationship. Depending on the type of relationship, use Primary Keys and Foreign Keys or Linking Tables to connect the tables.
- Determine the type of participation for each relationship.
- Determine the degree of participation for each relationship.
- The type and degree could be obvious or based on specific business rules.

Phase 5: Determine and define business rules
- Interview users and management to determine specific limitations and requirements that will be imposed on the data, data structures, or relationships.
- These specifications or constraints will become the business rules that serve to establish various levels of data integrity.

Phase 6: Determine and establish views
- Interview users and management to identify different ways they look at the data in the database and develop them as views.
- The views are the queries performed on the data, whether it’s from one or more tables and fields.

Phase 7: Review data integrity
- Review each table and its fields to ensure that it meets the criteria of a properly designed table and field.
- Review and check field specifications for each field and check field level integrity.
- Review the validity of each relationship, confirming the type of relationship as well as the type or participation and degree of participation for each table within the relationship.
- Go over the Business Rules to confirm that the limitations and/or requirements have been placed on the database.

4.2 The Entity Relationship Model

The entity-relationship (ER) data model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design.

The ER model is important primarily for its role in database design. It provides useful concepts that allow us to move from an informal description of what users want from their database to a more detailed and precise, description that can be implemented in a DBMS. Within the larger context of the overall design process, the ER model is used in a phase called conceptual database design.
**Entities, attributes, and entity set**

An entity is an object in the real world that is distinguishable from other objects.

- It is often useful to identify a collection of similar entities. Such a collection is called an entity set. Note that entity sets need not be disjoint.
- An entity is described using a set of attributes. All entities in a given entity set have the same attributes; this essentially is what we mean by similar.
- Our choice of attributes reflects the level of detail at which we wish to represent information about entities.
- For each attribute associated with an entity set, we must identify a domain of possible values.
- Further, for each set, we choose a key. A key is a minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one candidate key; if so, we designate one them as the primary key.

**Relationships and relationship sets**

A relationship is an association among two or more entities. For example, we may have the relationship that Attishoo works in the pharmacy department.

- As with entities, we may wish to collect a set of similar relationships into a relationship set.

![Fig. 4.1 Employees entity set](image)

- A relationship set can be thought of as a set of n-tuples:

  \{ (e_1; \ldots; e_n) \mid e_1 \in E_1; \ldots; e_n \in E_n \}

- Each n-tuple denotes a relationship involving n entities e_1 through e_n, where entity e_i is in entity set E_i.
- In figure shown below, we show the relationship set Works_In.
- In each relationship it indicates a department in which an employee works.
- Note that several relationship sets might involve the same entity sets. For example, we could also have a Manages relationship set involving Employees and Departments.
A relationship can also have descriptive attributes.

Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities; for example, we may wish to record that Attishoo works in the pharmacy department as of January 1991. This information is captured in above figure by adding an attribute, since, to Works In.

A relationship must be uniquely identified by the participating entities, without reference to the descriptive attributes. The since value is shown beside each relationship.

### 4.3 Constraints

An E-R enterprise schema may define certain constraints to which the contents of a database must conform. In this section, we examine mapping cardinalities and participation constraints, which are two of the most important types of constraints.

#### Mapping cardinalities

Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.

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

- One to one: An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.
- One to many: An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with at most one entity in A.
- Many to one: An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A.
- Many to many: An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.
- The appropriate mapping cardinality for a particular relationship set obviously depends on the real-world situation that the relationship set is modeling. As an illustration, consider the borrower relationship set.
- If, in a particular bank, a loan can belong to only one customer, and a customer can have several loans, then the relationship set from customer to loan is one to many. If a loan can belong to several customers, the relationship set is many to many.
Participation constraints

- The participation of an entity set E in a relationship set R is said to be total if every entity in E participates in at least one relationship in R.
- If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be partial.
- For example, we expect every loan entity to be related to at least one customer through the borrower relationship. Therefore the participation of loan in

Fig. 4.4 Mapping cardinalities (a) Many to one (b) Many to many

- The relationship set borrower is total. In contrast, an individual can be a bank customer whether or not she has a loan with the bank.
- Hence, it is possible that only some of the customer entities are related to the loan entity set through the borrower relationship, and the participation of customer in the borrower relationship set is therefore partial.

4.4 Entity Relationship Diagram

Entity Relationship Diagrams are a major data modelling tool and will help organise the data in your project into entities and define the relationships between the entities. This process has proved to enable the analyst to produce a good database structure so that the data can be stored and retrieved in a most efficient manner.

By using a graphical format it may help communication about the design between the designer and the user and the designer and the people who will implement it.
Components of an ERD
An ERD typically consists of four different graphical components:

Entity:
A data entity is anything real or abstract about which we want to store data. Entity types fall into five classes: roles, events, locations, tangible things or concepts. E.g. employee, payment, campus, book. Specific examples of an entity are called instances. For example, the employee John Jones, Mary Smith’s payment, etc.

Relationship:
A data relationship is a natural association that exists between one or more entities. E.g. Employees process payments.

Cardinality:
Defines the number of occurrences of one entity for a single occurrence of the related entity. For example, an employee may process many payments but might not process any payments depending on the nature of her job.

Attribute:
A data attribute is a characteristic common to all or most instances of a particular entity. Synonyms include property, data element, and field. E.g. Name, address, Employee Number, pay rate are all attributes of the entity employee. An attribute or combination of attributes that uniquely identifies one and only one instance of an entity is called a primary key or identifier. For example, Employee Number is a primary key for Employee.

Figure shown below is a very simple, example ERD with each of the four components labelled.

![Figure 4.5 A simple, example ERD](image)

Different ERD styles
As with many data modelling tools there are a number of different styles used to create. The style used in this course will be the one labelled “Information Engineering”. One Methodology for Developing an ERD.

Typically, one will start with a case study or perhaps a logical model of the system to be developed. This document will demonstrate how to use the following process to convert that information into an ERD.
The process has ten steps:

<table>
<thead>
<tr>
<th>Step 1: Identify Entities</th>
<th>Identify the roles, events, locations, tangible things or concepts about which the end-users want to store data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2: Find Relationships</td>
<td>Find the natural associations between pairs of entities using a relationship matrix.</td>
</tr>
<tr>
<td>Step 3: Draw Rough ERD</td>
<td>Put entities in rectangles and relationships on line segments connecting the entities.</td>
</tr>
<tr>
<td>Step 4: Fill in Cardinality</td>
<td>Determine the number of occurrences of one entity for a single occurrence of the related entity.</td>
</tr>
<tr>
<td>Step 5: Define Primary Keys</td>
<td>Identify the data attribute(s) that uniquely identify one and only one occurrence of each entity.</td>
</tr>
<tr>
<td>Step 6: Draw Key-Based ERD</td>
<td>Eliminate Many-to-Many relationships and include primary and foreign keys in each entity.</td>
</tr>
<tr>
<td>Step 7: Identify Attributes</td>
<td>Name the information details (fields) which are essential to the system under development.</td>
</tr>
<tr>
<td>Step 8: Map Attributes</td>
<td>For each attribute, match it with exactly one entity that it describes.</td>
</tr>
<tr>
<td>Step 9: Draw fully attributed ERD</td>
<td>Adjust the ERD from step 6 to account for entities or relationships discovered in step 8.</td>
</tr>
<tr>
<td>Step 10: Check Results</td>
<td>Does the final Entity Relationship Diagram accurately depict the system data?</td>
</tr>
</tbody>
</table>

A simple example
The above process will be illustrated by working through the following example. A company has several departments.

- Each department has a supervisor and at least one employee.
- Employees must be assigned to at least one, but possibly more departments.
- At least one employee is assigned to a project, but an employee may be on vacation and not assigned to any projects.
- The important data fields are the names of the departments, projects, supervisors and employees, as well as the supervisor and employee number and a unique project number.
- Each of the following sections corresponds to one of the stages above.

Identify entities

- In this stage, you look through the information about the system and seek to identify the roles, events, locations, concepts and other tangible things that you wish to store data about.
- One approach to this is to work through the information and highlight those words which you think correspond to entities.
- A company has several departments. Each department has a supervisor and at least one employee. Employees must be assigned to at least one, but possibly more departments.
- At least one employee is assigned to a project, but an employee may be on vacation and not assigned to any projects.
The important data fields are the names of the departments, projects, supervisors and employees, as well as the supervisor and employee number and a unique project number.

This example is quite simple in that the last couple of lines actually tell you what data is being stored and that makes it somewhat easy to identify the entities. You may notice that “company” has been highlighted.

It is not an example of an entity. A single company will use the system we are designing to keep track of its departments, projects, supervisors and employees.

A true entity should have more than one instance.

Our system will probably contain information about multiple employees, supervisors, projects and departments. But it will only contain one instance of a company.

Find relationships

In this step the aim is to identify the associations, the connections between pairs of entities.

A simple approach to do this is using a relationship matrix. This is a fancy name for a table that has rows and columns for each of the identified entities.

Table below is an example relationship matrix for the above example. With four entities, there are four rows and four columns.

Each cell is used to indicate whether that combination of entities has an association.

<table>
<thead>
<tr>
<th></th>
<th>Department</th>
<th>Employee</th>
<th>Supervisor</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Employee</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supervisor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1 An example relationship matrix

Having created your relationship matrix you should now go through each cell and decide whether there is an association. For example, the first cell on the second row is used to indicate if there is a relationship between the entity “Employee” and the entity “Department”. Table 2 is an example relationship matrix that has been completed for the current example.

<table>
<thead>
<tr>
<th></th>
<th>Department</th>
<th>Employee</th>
<th>Supervisor</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Employee</td>
<td>Is assigned</td>
<td>Run by</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supervisor</td>
<td>Run</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>Uses</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 An example complete relationship matrix

The names placed in the cells are meant to capture/describe the relationships. So you can use them like this

- A department is assigned an employee
- A department is run by a supervisor
- An employee belongs to a department
- An employee works on a project
- A supervisor runs a department
- A project uses an employee
**Draw rough ERD**

Completed relationship matrix now contains a list of all the entities and all the relationships between those entities. This is enough information to create a rough ERD.

Draw a diagram and follow procedure given below:
- Place all the entities in rectangles
- Use diamonds and lines to represent the relationships between entities.

Obviously, you should lay out the entities so there is no overlap of the relationships. Figure given below is an example rough ERD that represents the content of table 4.2.

![Fig. 4.6 An example rough ERD](image-url)

### 4.5 Entity Relationship Design Issues

The notions of an entity set and a relationship set are not precise, and it is possible to define a set of entities and the relationships among them in a number of different ways. In this section, we examine basic issues in the design of an E-R database schema.

**Use of entity sets versus attributes**

Consider the entity set employee with attributes employee-name and telephone-number. It can easily be argued that a telephone is an entity in its own right with attributes telephone-number and location (the office where the telephone is located). If we take this point of view, we must redefine the employee entity set as:
- The employee entity set with attribute employee-name
- The telephone entity set with attributes telephone-number and location
- The relationship set emp-telephone, which denotes the association between employees and the telephones that they have.
Use of entity sets versus relationship sets

- It is not always clear whether an object is best expressed by an entity set or a relationship set. We assumed that a bank loan is modelled as an entity.
- An alternative is to model a loan not as an entity, but rather as a relationship between customers and branches, with loan-number and amount as descriptive attributes. Each loan is represented by a relationship between a customer and a branch.

Two problems arise because of the replication:

- The data are stored multiple times, wasting storage space.
- Updates potentially leave the data in an inconsistent state, where the values differ in two relationships for attributes that are supposed to have the same value. The issue of how to avoid such replication is treated formally by normalisation theory.

Binary versus n-ary relationship sets

Relationships in databases are often binary. Some relationships that appear to be non-binary could actually be better represented by several binary relationships. For instance, one could create a ternary relationship parent, relating a child to his/her mother and father. However, such a relationship could also be represented by two binary relationships, mother and father, relating a child to his/her mother and father separately. Using the two relationships mother and father allows us record a child's mother, even if we are not aware of the father’s identity; a null value would be required if the ternary relationship parent is used. Using binary relationship sets is preferable in this case.

In fact, it is always possible to replace a nonbinary (n-ary, for n > 2) relationship set by a number of distinct binary relationship sets. For simplicity, consider the abstract ternary (n = 3) relationship set R, relating entity sets A, B, and C. We replace the relationship set R by an entity set E, and create three relationship sets:

- RA, relating E and A
- RB, relating E and B
- RC, relating E and C

We can generalise this process in a straightforward manner to n-ary relationship sets. Thus, conceptually, we can restrict the E-R model to include only binary relationship sets. However, this restriction is not always desirable.

- An identifying attribute may have to be created for the entity set created to represent the relationship set. This attribute, along with the extra relationship sets required, increases the complexity of the design and overall storage requirements.
- An n-ary relationship set shows more clearly that several entities participate in a single relationship.
- There may not be a way to translate constraints on the ternary relationship into constraints on the binary relationships. For example, consider a constraint that says that R is many-to-one from A, B to C; that is, each pair of entities from A and B is associated with at most one C entity. This constraint cannot be expressed by using cardinality constraints on the relationship sets RA, RB, and RC.

Placement of relationship attributes

The cardinality ratio of a relationship can affect the placement of relationship attributes. Thus, attributes of one-to-one or one-to-many relationship sets can be associated with one of the participating entity sets, rather than with the relationship set.

- For instance, let us specify that depositor is a one-to-many relationship set such that one customer may have several accounts, but each account is held by only one customer.
- In this case, the attribute access-date, which specifies when the customer last accessed that account, could be associated with the account entity set, as in figure shown below depicts; to keep the figure simple, only some of the attributes of the two entity sets are shown.
Since each account entity participates in a relationship with at most one instance of customer, making this attribute designation would have the same meaning as would placing access-date with the depositor relationship set.

Attributes of a one-to-many relationship set can be repositioned to only the entity set on the “many” side of the relationship.

For one-to-one relationship sets, on the other hand, the relationship attribute can be associated with either one of the participating entities.

The design decision of where to place descriptive attributes in such cases as a relationship or entity attribute should reflect the characteristics of the enterprise being modelled.

4.6 Extended E-R Features

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. In this section, we discuss the extended E-R features of specialisation, generalisation, higher- and lower-level entity sets, attribute inheritance, and aggregation.

Specialisation

An entity set may include sub groupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set. The E-R model provides a means for representing these distinctive entity groupings. Consider an entity set person, with attributes name, street, and city. A person may be further classified as one of the following:

- customer
- employee

Each of these person types is described by a set of attributes that includes all the attributes of entity set person plus possibly additional attributes. For example, customer entities may be described further by the attribute customer-id, whereas employee entities may be described further by the attributes employee-id and salary. The process of designating sub groupings within an entity set is called specialisation. The specialisation of person allows us to distinguish among persons according to whether they are employees or customers.

As another example, suppose the bank wishes to divide accounts into two categories, checking account and savings account. Savings accounts need a minimum balance, but the bank may set interest rates differently for different customers, offering better rates to favoured customers. Checking accounts have a fixed interest rate, but offer an overdraft facility; the overdraft amount on a checking account must be recorded.
The bank could then create two specialisations of account, namely savings-account and checking-account. As we saw earlier, account entities are described by the attributes account-number and balance. The entity set savings-account would have all the attributes of account and an additional attribute interest-rate. The entity set checking account would have all the attributes of account, and an additional attribute overdraft amount.

We can apply specialisation repeatedly to refine a design scheme. For instance, bank employees may be further classified as one of the following:

- officer
- teller
- secretary

Each of these employee types is described by a set of attributes that includes all the attributes of entity set employee plus additional attributes. For example, officer entities may be described further by the attribute office-number, teller entities by the attributes station-number and hours-per-week, and secretary entities by the attribute hours-per week. Further, secretary entities may participate in a relationship secretary-for, which identifies which employees are assisted by a secretary.

In terms of an E-R diagram, specialisation is depicted by a triangle component labeled ISA, as in figure shown below. The label ISA stands for “is a” and represents, for example, that a customer “is a” person. The ISA relationship may also be referred to as a superclass-subclass relationship. Higher- and lower-level entity sets are depicted as regular.

**Generalisation**

The refinement from an initial entity set into successive levels of entity subgroupings represents a top-down design process in which distinctions are made explicit.

- The design process may also proceed in a bottom-up manner, in which multiple entity sets are synthesised into a higher-level entity set on the basis of common features.
- There are similarities between the customer entity set and the employee entity set in the sense that they have several attributes in common. This commonality can be expressed by generalisation, which is a containment relationship that exists between a higher-level entity set and one or more lower-level entity sets.
- In our example, person is the higher-level entity set and customer and employee are lower-level entity sets.
- For all practical purposes, generalisation is a simple inversion of specialisation.
• Specialisation stems from a single entity set; it emphasises differences among entities within the set by creating distinct lower-level entity sets.
• Generalisation proceeds from the recognition that a number of entities sets share some common features.
• On the basis of their commonalities, generalisation synthesises these entity sets into a single, higher-level entity set.
• Generalisation is used to emphasise the similarities among lower-level entity sets and to hide the differences; it also permits an economy of representation in that shared attributes are not repeated.

Attribute inheritance
A crucial property of the higher and lower-level entities created by specialisation and generalisation is attribute inheritance.
• The attributes of the higher-level entity sets are said to be inherited by the lower-level entity sets.
• A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher-level entity (or super class) participates.
• Whether a given portion of an E-R model was arrived at by specialisation or generalisation, the outcome is basically the same:
  • A higher-level entity set with attributes and relationships that apply to all of its lower-level entity sets
  • Lower-level entity sets with distinctive features that apply only within a particular lower-level entity set.
Constraints on generalisations

- To model an enterprise more accurately, the database designer may choose to place certain constraints on a particular generalisation.
- One type of constraint involves determining which entities can be members of a given lower-level entity set. Such membership may be one of the following:

Condition-defined

In condition-defined lower-level entity sets, membership is evaluated on the basis of whether or not an entity satisfies an explicit condition or predicate.

User-defined

- User-defined lower-level entity sets are not constrained by a membership condition; rather, the database user assigns entities to a given entity set.
- A second type of constraint relates to whether or not entities may belong to more than one lower-level entity set within a single generalisation. The lower-level entity sets may be one of the following:
  - Disjoint
    - A disjointness constraint requires that an entity belong to no more than one lower-level entity set.
  - Overlapping
    - In overlapping generalisations, the same entity may belong to more than one lower-level entity set within a single generalisation.
    - Lower-level entity overlap is the default case; a disjointness constraint must be placed explicitly on a generalisation (or specialisation).
    - We can note a disjointedness constraint in an E-R diagram by adding the word disjoint next to the triangle symbol.
    - A final constraint, the completeness constraint on a generalisation or specialisation, specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalisation/specialisation.
  - Aggregation
    - One limitation of the E-R model is that it cannot express relationships among relationships.
    - To illustrate the need for such a construct, consider the ternary relationship works-on, which we saw earlier, between an employee, branch, and job.
    - Now, suppose we want to record managers for tasks performed by an employee at a branch; that is, we want to record managers for (employee, branch, job) combinations.
    - Let us assume that there is an entity set manager. One alternative for representing this relationship is to create a quaternary relationship manages between employee, branch, job, and manager.
    - Using the basic E-R modeling constructs, we obtain the E-R diagram of in figure shown below.
    - We have omitted the attributes of the entity sets, for simplicity.
    - It appears that the relationship sets works-on and manages can be combined into one single relationship set.
There is redundant information in the resultant figure, however, since every employee, branch, job combination in manages is also in works-on.

The best way to model a situation such as the one just described is to use aggregation. Aggregation is an abstraction through which relationships are treated as higher level entities.

Alternative E-R Notations: Fig. 4.11 summarises the set of symbols we have used in E-R diagrams.

There is no universal standard for E-R diagram notation, and different books and E-R diagram software use different notations; fig. 4.11 indicates some of the alternative notations that are widely used.

An entity set may be represented as a box with the name outside, and the attributes listed one below the other within the box.

The primary key attributes are indicated by listing them at the top, with a line separating them from the other attributes. Cardinality constraints can be indicated in several different ways, as fig. 4.11 shows.

The labels * and 1 on the edges out of the relationship are sometimes used for depicting many-to-many, one-to-one, and many-to-one relationships, as the figure shows.

The case of one-to-many is symmetric to many-to-one, and is not shown. In another alternative notation in the figure, relationship sets are represented by lines between entity sets, without diamonds; only binary relationships can be modelled thus, cardinality constraints in such a notation are shown by “crow’s foot” notation, as in the figure.
Fig. 4.10 E-R diagram with aggregation
Summary

- Conduct an interview with the owner or manager or the organisation. Establish the purpose of database. Start the process by defining the end result.
- A well written mission statement is free of any phrases or sentences that explicitly describe specific tasks.
- The purpose of a mission objective is to help define various structures within the database and to help guide the overall direction of the database’s development.
- The entity-relationship (ER) data model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design.
- An entity is an object in the real world that is distinguishable from other objects. It is often useful to identify a collection of similar entities. Such a collection is called an entity set.
- A relationship is an association among two or more entities. For example, we may have the relationship that Attishoo works in the pharmacy department.
- An E-R enterprise schema may define certain constraints to which the contents of a database must conform.
- Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.
- The participation of an entity set E in a relationship set R is said to be total if every entity in E participates in at least one relationship in R. If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be partial.
- Entity Relationship Diagrams are a major data modelling tool and will help organise the data in your project into entities and define the relationships between the entities.
- A data entity is anything real or abstract about which we want to store data. Entity types fall into five classes: roles, events, locations, tangible things or concepts.
- Defines the number of occurrences of one entity for a single occurrence of the related entity.
- A data attribute is a characteristic common to all or most instances of a particular entity.
- As with many data modelling tools there are a number of different styles used to create. The style used in this course will be the one labelled “Information Engineering”.
- Entity is a data entity is anything real or abstract about which we want to store data. Entity types fall into five classes: roles, events, locations, tangible things or concepts.
- Relationship is a data relationship is a natural association that exists between one or more entities.

References

Recommended Reading

1. The ER model is important primarily for its role in ________ design.
   a. database
   b. program
   c. code
   d. system

2. The ________ data model allows describing the data involved in a real-world enterprise.
   a. SQL
   b. Secure
   c. entity-relationship
   d. entity

3. Which of the following is true?
   a. An entity is described using a set of attributes.
   b. An entity is described using a set of tables.
   c. An entity is described using a set of value.
   d. An entity is described using a set of entity.

4. Descriptive __________ are used to record information about the relationship.
   a. attributes
   b. entity
   c. values
   d. rules

5. Which of the following are most important types of constraints?
   a. Attribute and ER
   b. DML and DLL
   c. SQL and query
   d. mapping cardinalities and participation

6. Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a __________.
   a. relationship set
   b. database
   c. data model
   d. data type

7. A ________ is anything real or abstract about which we want to store data.
   a. database
   b. data model
   c. data type
   d. data entity
8. A data ________ is a characteristic common to all or most instances of a particular entity.
   a. entity
   b. attribute
   c. values
   d. rules

9. Relationships in databases are often ________.
   a. float
   b. binary
   c. octane
   d. decimal

10. An identifying attribute have to be created for the ________ created to represent the relationship set.
    a. entity set
    b. relationship set
    c. database
    d. data model
Chapter V
Relational Database Design

Aim
The aim of this chapter is to:

• explain dependency preserving decomposition
• elucidate decomposition to BCNF
• discuss decomposition to 3NF

Objectives
The objectives of this chapter are to:

• enlist features of good relational design
• enlighten atomic domain and 1NF
• describe redundant data can cause problems in a database

Learning outcome
At the end of this chapter, you will be able to:

• recognise redundancy may lead to anomalies of data
• identify decomposition using normal dependencies
• understand lossless join decompositions
### 5.1 Features of Good Relational Design

Designing of relation schema is based on the conceptual design. The relation schema can be generated directly using data model such as ER or enhanced ER (EER) or some other conceptual data model. The goodness of the resulting set schemas depends on the designing of the data model. All the relations that are obtained from data model have basic properties, that is, relations having no duplicate tuples, tuples having no particular ordering associated with them and each element in the relation being atomic. However, to qualify as a good relational database design, it should also have following features.

- The goal of a relational database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy and to retrieve information easily.
- This is accomplished by designing schemas that are in an appropriate normal form.
  - Normalisation ensures that the data model minimises duplication of data or redundancy.
  - Normalisation is a technique to reduce redundancy
  - It is a decomposition process to split tables.
- Redundant data can cause huge problems in a database:
  - Someone has to enter the same data repeatedly.
  - If a change is made in one piece of data, the change has to be made in many places.
- Redundancy may lead to anomalies of data:
  - A deletion anomaly results when the deletion of information about one entity leads to the deletion of information about another entity.
  - An insertion anomaly occurs when the information about an entity cannot be inserted unless the information about another entity is known.

### 5.2 Atomic Domains and First Normal Forms

Quick rehash relations are sets of attributes, and each attribute is supposed to belong to a domain.

- Domains are supposed to be atomic that is, their members may not be broken down into subcomponents.
- This is the first distinction between E-R and UML designs and a relational design non-atomic attributes. While E-R and UML both allow multivalued attributes (for example, list of phone numbers) and composite attributes (for example, address consisting of street, city, state, and zip), the relational model does not.
- So, the first step in relational design is to remove all multivalued and composite attributes. Or, make all attributes atomic.
- A relational schema R is in first normal form (1NF) if the domains of all attributes in R are atomic.
- In other words, a relational schema R is in first normal form if it conforms to the formal definition of a relational schema. Seems like an obvious no-brainer, but it is an essential first building block.

### 5.3 Decomposition using Normal Dependencies

Suppose that relation R contains attributes A1 ... An. A decomposition of R consists of replacing R by two or more relations such that:

- Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
- Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- For example, Can decompose SNLRWH into SNLRH and RW.
- Decompositions should be used only when needed.
SNLRWH has FDs $S \rightarrow SNLRWH$ and $R \rightarrow W$

Second FD causes violation of 3NF; W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations, and to remove W from the main schema i.e., we decompose SNLRWH into SNLRH and RW.

The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of.

**Problems with decompositions**

- There are three potential problems to consider:
  - Some queries become more expensive, for example, How much did sailor Joe earn? (salary = W*H)
  - Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation! Fortunately, not in the SNLRWH example.
  - Checking some dependencies may require joining the instances of the decomposed relations. Fortunately, not in the SNLRWH example.
  - Tradeoff: Must consider these issues vs. redundancy.

**Lossless Join decompositions**

- Decomposition of R into X and Y is lossless-join w.r.t. a set of FDs F if, for every instance r that satisfies F:
  - $\pi_X(\gamma) \bowtie \pi_X(\gamma) = \gamma$
  - It is always true that $r \subseteq \pi_X(\gamma) \bowtie \pi_X(\gamma)$
  - In general, the other direction does not hold. If it does, the decomposition is lossless-join.
  - Definition extended to decomposition into 3 or more relations in a straightforward way.
  - It is essential that all decompositions used to deal with redundancy be lossless.

**Dependency Preserving decomposition**

- Consider CSJDPQV, C is key, $JP \rightarrow C$ and $SD \rightarrow P$.
  - BCNF decomposition: CSJDQV and SDP
  - Problem: Checking $JP \rightarrow C$ requires a join!

- Dependency preserving decomposition (Intuitive):
  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold.

- Projection of set of FDs $F$: If R is decomposed into X, ... projection of F onto X (denoted $F_X$) is the set of FDs $U \rightarrow V$ in $F^+$ (closure of F) such that U, V are in X.

- Decomposition of R into X and Y is dependency preserving if $(F_X \cup F_Y)^+ = F^+$ i.e., if we consider only dependencies in the closure $F^+$ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in $F^+$.

- Important to consider $F^+$, not F, in this definition:
  - $ABC, \ A \rightarrow B, \ B \rightarrow C, \ C \rightarrow A, \ decomposed \ into \ AB \ and \ BC$.
  - Is this dependency preserving? Is $C \rightarrow A$ preserved

- Dependency preserving does not imply lossless join:
  - $ABC, \ A \rightarrow B, \ decomposed \ into \ AB \ and \ BC$. 

---

85
**Decomposition to BCNF**

- To decompose a relational schema into BCNF, we first have to know when that schema is not in BCNF. Of course the general test is to check the relation directly against the definition of BCNF, but that can be very expensive because the definition involves $F^+$.
- The text provides a number of simpler tests that take less work to catch a non-BCNF relation. These function as a subroutine in the actual decomposition algorithm, used as the condition for performing decomposition:

```plaintext
result := {R};
done := false;
compute Fplus;
while (not done) {
    if (there is a schema R[i] in result that is not in BCNF) {
        let alpha -> beta be a nontrivial functional dependency such that
        alpha -> beta holds on R[i],
        alpha -> R[i] is not in Fplus, // alpha is not a superkey
        alpha and beta are disjoint;
        result.remove(R[i]);
        result.add(R[i] - beta); // yank out the dependent attributes
        result.add({alpha, beta}); // represent alpha -> beta directly
    } else {
        done := true;
    }
}
```

The core notion of this algorithm is that one end up with a set of relations with superkeys that correspond to the left sides of some functional dependency in $F^+$. The resulting decomposition is lossless, but it is not necessarily dependency-preserving.

**Decomposition to 3NF**

The decomposition algorithm to 3NF takes a different tack than the BCNF version; instead of taking a single relation then breaking it up, this algorithm starts from scratch, and actually builds the relations from a canonical cover of $F$:

```plaintext
Fc := compute a canonical cover for F;
result := {};
for (alpha -> beta: Fc) {
    if no schema in result contains {alpha, beta} {
        result.add({alpha, beta});
    }
    if no schema in result contains a candidate key for R {
        result.add(any candidate key for R)
    }
return result;
```
Because it builds a schema from nothing, the algorithm is also known as the 3NF synthesis algorithm.

- Note also that, because $F_c$ is not unique for a given set $F$, then the algorithm can produce more than one valid 3NF decomposition, depending on which canonical cover is chosen and the order in which the functional dependencies are listed.
- This algorithm guarantees both a lossless and dependency-preserving decomposition. The trade off is that, since 3NF is not as “strict” as BCNF, there may be some redundancy of data and use of nulls.

**Decomposition bottom line**

Through all these algorithms and definitions, this is what we want ideally from a relational database schema:

- **BCNF**
- lossless decomposition
- dependency-preserving decomposition

However, sometimes we can’t have all three, specifically BCNF and dependency preservation. If we hit that situation, then we choose between BCNF without dependency preservation or 3NF with dependency preservation.

The practical upshot of all of this is that most relational database implementations base their integrity checks mainly on primary and foreign keys. Furthermore, an implementation feature called a materialised view allows a database to automatically perform joins on certain relations, with tests performed on the view. This frequently results in an acceptable workaround for dependency preservation while maintaining BCNF. Thus, in practice, the scale tends to tip toward choosing BCNF over dependency preservation.

### 5.4 Functional Dependency Theory

Functional Dependency (FD) is a term derived from the mathematical theory that underpins relational database theory. It concerns the dependence of the values of one set of attributes on those of another set of attributes.

Consider the relation:

$$\text{STUDENTS} = \{\text{Student\_ID, First\_Name, Last\_Name}\}$$

We may state that the set of attributes \{First\_Name, Last\_Name\} is functionally dependent on the attribute set \{Student\_ID\}. This means that, given a value for Student\_ID, we can always uniquely determine the value of First\_Name and the value of Last\_Name. Note that, for this relation, the opposite would not be true. For example, if there are three students with the same first and last name we will get a list of three student IDs. That is we cannot uniquely determine a value for Student\_ID given values of the attributes \{First\_Name, Last\_Name\}.

$$\text{Student\_ID determines \{First\_Name, Last\_Name\}}$$

but \{First\_Name, Last\_Name\} does not determine Student\_ID.

We define functional dependency more formally:

A set of attributes $Y$ is functionally dependent on a set of attributes $X$ if a given set of values for each attribute in $X$ determines unique values for the set of attributes in $Y$.

We use the notation:

$$X \rightarrow Y$$

To denote that $Y$ is functionally dependent on $X$. The set of attributes $X$ is known as the determinant of the FD $X \rightarrow Y$. 


Table 5.1 Relation: Assign

The attribute Project-Budget is functionally dependent on the attribute Project, because each project has one given project budget. Once a project name is known, a unique value of project-Budget is also known. We can alternatively express this FD using the notation: Project → Project-Budget:

Another FD in ASSIGN is:

\( \{\text{Person-ID, Project}\} \rightarrow \text{Time-spent-by-person-on-project} \)

We can express these FDs of the relation ASSIGN graphically using an FD diagram (as below) which represents attributes and sets of attributes as rectangles, and FDs as arrows directed from the determinant to the attribute(s) it determines.

Exercise

Write out each of the FDs depicted in the diagram below and describe what each one means. If all the attributes were part of the same relation, what would the relation key be?
Exercise
Again referring to the database schema from Exercise 3, write down the functional dependencies between the attributes in each relation.

STUDENT (Student_ID, First_Name, Last_Name)
REGISTER (Student_ID, Module_ID, Semester-Start-Date)
LECTURER (Lecturer_ID, First_Name, Last_Name)
MODULE (Module_ID, Module_Name, Lecturer_ID)

Full Functional dependencies
Consider the following FD from our previous example relation:

\{Person-ID, Project, Project-Budget\} \rightarrow \{Time-spent-by-person-on-project\}

This FD is true, as it states that for each value of Person-ID, Project and Project-Budget, there is a unique value for the time spent. However, the attribute Project-Budget is not required in the determinant.

The term full functional dependency is used to describe the minimum set of attributes in the determinant of an FD. The rules for full functional dependency are that if the set of attributes Y are to be fully dependent on the set of attributes X, the following must hold: Y is functionally dependent on X and Y is not functionally dependent on any subset of X.

According to these rules, the above FD is not fully dependent, whereas, by removing the attribute Project-Budget, we can acquire an FD which is fully functionally dependent. We say that the attribute Project-Budget is redundant. It may be removed without losing information.

The converse of full FD: A functional dependency \(X \rightarrow Y\) is called a partial functional dependency if some attribute \(A \in X\) can be removed from \(X\) and the dependency will still hold.

Question: How does the concept of a relation key relate to the concept of full functional dependency?
Trivial and Non-Trivial Functional dependencies
It is often the case when examining the FDs associated with a relation that some FDs are of less interest because they are self-evident. For example consider the relation:

```
STUDENT (Student_ID, First_Name, Last_Name)
```

The FD \{First_Name, Last_Name\}→\{First_Name\} is true but of little informational interest.

We say that an FD is a trivial FD if and only if the right-hand side is a subset (not necessarily a proper subset) of the left-hand side. Non-trivial FDs are, of course, those which are not by definition trivial.

5.5 Decomposition using Functional Dependencies
In this section, we introduce some new mathematical concepts relating to functional dependency and, along the way, show their practical use in relational design. Our goal is to have a toolbox to algorithmically generate relations, which meet our criteria for good relational design:

- Eliminate/reduce redundancy and the possibilities for update anomalies by decomposing relations without losing any information from the original relations (i.e., all of the original attributes must be present in the set of resulting relations; decomposition should preserve functional dependencies and not create spurious data when relations are joined).

Closure of a set of FDs:
Given a set of FDs we can usually infer additional FDs. For example, consider the following relation and set of FDs holding on the relation:

```
TEACH = \{Module_ID, Lecturer_ID, Lect_First_Name, Lect_Last_Name\}
F = \{ \{Module_ID\}→\{Lecturer_ID\},
\{Lecturer_ID\} → \{Lect_First_Name, Lect_Last_Name\}\}
```

Given these FDs we can infer that, for example:
```
\{Module_ID\}→\{Lect_First_Name, Lect_Last_Name\}
```

In addition, given F, it is also true that, for example:
```
\{Lecturer_ID\} → \{Lect_First_Name\}
```

We say that these FDs are logically implied by F. We can continue to find other FDs that are logically implied by F. The set of all such FDs (and including those in F) are called the closure of F (denoted \(F^+\)).

Stating these concepts formally:

```
An FD \(f\) is logically implied by a set of FDs \(F\) if \(f\) holds whenever all FDs in \(F\) hold. The closure of \(F\) is the set of all FDs that are implied by \(F\). The closure of \(F\) is denoted as \(F^+\).
```

In order to fully define logical implication, so that we can compute the closure of a set of FDs, we must have a set of rules (axioms) which govern how an FD can be logically implied by some set of FDs.

The Inference Axioms (Armstrong’s Axioms)
- If \(Y \subseteq X\), then \(X→Y\) (reflexivity)
- If \(Z \subseteq W\) and \(X→Y\), then \(XW→YZ\) (augmentation)
- If \(X→Y\) and \(Y→Z\) then \(X→Z\) (transitivity)
Other rules which may be derived from the axioms include:
If \( X \rightarrow Y \) and \( YW \rightarrow Z \) then \( XW \rightarrow Z \)
If \( X \rightarrow Z \) and \( X \rightarrow Y \) then \( X \rightarrow YZ \)
If \( X \rightarrow YZ \) then \( X \rightarrow Y \) and \( X \rightarrow Z \)

Using these axioms we can find all FDs logically implied by some set of FDs \( F \) and hence find its closure \( F^+ \).

**Exercise**
Using the axioms, find the closure \( F^+ \) of \( F = \{AB \rightarrow CD, C \rightarrow E\} \).

**Application of closure to testing dependency preservation**
Looking again at the PROJECTS schema from our previous 3NF example, we had the following functional dependencies:

![Fig. 5.3 Testing dependency preservation](image)

We decomposed the PROJECTS relation into two relations on which two corresponding sets of FDs held:

![Fig. 5.4 PROJECT relation](image)

If we look at all FDs in the new schema taken together:
\{Project→Project-Budget, Project→Department, Department→Dept-Address\}
It appears that we have lost one of the original FDs: Project→Dept-Address.

However, using the axioms, we can infer that Project→Dept-Address is logically implied by Project→Department and Department→Dept-Address so that the FD has not actually been lost.
One of our goals during decomposition is to preserve dependencies. This is important as dependencies represent constraints on the data which must be enforced to maintain the semantics of the relations. Given our definition of closure we can now define precisely how to check if a decomposition is dependency preserving:

A decomposition of a relation \( R \), on which the dependencies \( F \) hold, into relations \( R_1 \) and \( R_2 \) is dependency preserving if:

\[
(F_{R_1} \cup F_{R_2})^+ = F^+
\]

Where \( F_{R_1} \) is the set of dependencies that apply between attributes only in \( R_1 \) and similarly \( F_{R_2} \) is the set of dependencies that apply between attributes only in \( R_2 \).

Although the set of dependencies \( (F_{R_1} \cup F_{R_2}) \) might be different to the original set of dependencies \( F \), we only require that the two sets are equivalent (that is, their closures are the same) for the decomposition to be dependency preserving. For example, the closure of the PROJECTS relation in the previous example is:

\[
\{\text{Project} \rightarrow \text{Project-Budget}, \text{Project} \rightarrow \text{Department}, \text{Department} \rightarrow \text{Dept-Address}, \text{Project} \rightarrow \text{Dept-Address}\}
\]

The closure of the combined FDs in the decomposed tables is the same, so the decomposition is dependency preserving.

### 5.6 Decomposition using Multivalued Dependencies

The multivalued dependency \( X \rightarrow Y \) holds in a relation \( R \) if whenever we have two tuples of \( R \) that agree in all the attributes of \( X \), then we can swap their \( Y \) components and get two new tuples that are also in \( R \).

![Fig. 5.5 Multivalued dependencies](image)

**Example**

Drinkers (name, addr, phones, beersLiked) with MVD name \( \rightarrow \) phones. If Drinkers has the two tuples:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Phones</th>
<th>BeersLiked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>A</td>
<td>p1</td>
<td>b1</td>
</tr>
<tr>
<td>Sue</td>
<td>A</td>
<td>p2</td>
<td>b2</td>
</tr>
</tbody>
</table>

**Table 5.2 Drinker tuples 1**

It must also have the same tuples with phones components swapped:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Phones</th>
<th>BeersLiked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>A</td>
<td>p1</td>
<td>b1</td>
</tr>
<tr>
<td>Sue</td>
<td>A</td>
<td>p2</td>
<td>b1</td>
</tr>
</tbody>
</table>

**Table 5.3 Drinker tuples 2**
Note: we must check this condition for all pairs of tuples that agree on name, not just one pair.

**MVD Rules**
- Every FD is an MVD.
- Because if X ! Y, then swapping Y’s between tuples that agree on X doesn’t create new tuples. Example, in Drinkers: name → addr.
- Complementation: if X→ Y, then X→ Z, where Z is all attributes not in X or Y. Example: since name → phones holds in Drinkers, so does name → addr beersLiked.

**Splitting Doesn’t Hold**
Sometimes you need to have several attributes on the right of an MVD. For example:

Drinkers(name, areaCode, phones, beersLiked, beerManf)

<table>
<thead>
<tr>
<th>Name</th>
<th>areaCode</th>
<th>Phones</th>
<th>BeersLiked</th>
<th>beermanf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>650</td>
<td>555-1111</td>
<td>Bud</td>
<td>A.B.</td>
</tr>
<tr>
<td>Sue</td>
<td>650</td>
<td>555-1111</td>
<td>WickedAle</td>
<td>Pete’s</td>
</tr>
<tr>
<td>Sue</td>
<td>415</td>
<td>555-9999</td>
<td>Bud</td>
<td>A.B.</td>
</tr>
<tr>
<td>Sue</td>
<td>415</td>
<td>555-9999</td>
<td>WickedAle</td>
<td>Pete’s</td>
</tr>
</tbody>
</table>

Table 5.4 Drinker tuples 3

name→areaCode phones holds, but neither
name→areaCode nor name→phones do
Summary

- Designing of relation schema is based on the conceptual design. The relation schema can be generated directly using data model such as ER or enhanced ER (EER) or some other conceptual data model.
- The goodness of the resulting set schemas depends on the designing of the data model.
- The goal of a relational database design is to generate a set of relation schemas that allows to store information without unnecessary redundancy and to retrieve information easily.
- Quick rehash relations are sets of attributes, and each attribute is supposed to belong to a domain.
- Domains are supposed to be atomic that is, their members which is not to be broken down into subcomponents.
- To decompose a relational schema into BCNF, we first have to know when that schema is not in BCNF.
- The text provides a number of simpler tests that take less work to catch a non-BCNF relation.
- The core notion of this algorithm is that one end up with a set of relations with superkeys that correspond to the left sides of some functional dependency in $F^+$. The resulting decomposition is lossless, but it is not necessarily dependency-preserving.
- Functional Dependency (FD) is a term derived from the mathematical theory that underpins relational database theory. It concerns the dependence of the values of one set of attributes on those of another set of attributes.
- The attribute Project-Budget is functionally dependent on the attribute Project, because each project has one given project budget. Once a project name is known, a unique value of project-Budget is also known.

References


Recommended Reading

Self Assessment

1. Which of the following statements is true?
   a. Designing of relation schema is based on the conceptual design.
   b. Designing of relation schema is based on the practical design.
   c. Designing of relation schema is based on the illusion.
   d. Designing of relation schema is based on the programming.

2. The __________ can be generated directly using data model such as ER or enhanced ER (EER).
   a. normalisation
   b. relation schema
   c. entity schema
   d. quick rehash

3. ____________ ensures that the data model minimises duplication of data or redundancy.
   a. Normalisation
   b. Relation schema
   c. Entity schema
   d. Quick rehash

4. Which of the following is a process of reducing redundancy?
   a. Normalisation
   b. Relation schema
   c. Entity schema
   d. Quick rehash

5. A/ An________ anomaly occurs when the information about an entity cannot be inserted unless the information about another entity is known.
   a. normalisation
   b. relation schema
   c. insertion
   d. quick rehash

6. __________ relations are sets of attributes, and each attribute is supposed to belong to a domain.
   a. Normalisation
   b. Relation schema
   c. Entity schema
   d. Quick rehash

7. ____________ is a term derived from the mathematical theory that underpins relational database theory.
   a. Functional dependency
   b. Budget dependency
   c. Schedule dependency
   d. Method dependency
8. One of the goals during decomposition is to preserve ________.
   a. strategy
   b. rules
   c. dependencies
   d. methods

9. Full functional dependency is used to describe the ________ set of attributes in the determinant of an FD.
   a. minimum
   b. maximum
   c. average
   d. double

10. A __________ X→Y is called a partial functional dependency.
    a. normalisation
    b. relation schema
    c. entity schema
    d. functional dependency
Chapter VI
Transaction Management

Aim
The aim of this chapter is to:

• explain the transaction concept
• elucidate transaction state
• discuss implementation of atomicity and durability

Objectives
The objectives of this chapter are to:

• explain concurrent execution
• discuss resource utilisation
• describe reduce waiting time

Learning outcome
At the end of this chapter, you will be able to:

• understand the concept of serialisability
• identify recoverability
• talk about cascade less schedules
6.1 Transaction Concept

A transaction is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language or programming language (for example, SQL, COBOL, C, C++, or Java), where it is delimited by statements (or function calls) of the form begin transaction and end transaction. The transaction consists of all operations executed between the begin transaction and end transaction. To ensure integrity of the data, we require that the database system maintains the following properties of transactions:

- **Atomicity**: Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency**: Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation**: Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions \( T_i \) and \( T_j \), it appears to \( T_i \) that either \( T_j \) finished execution before \( T_i \) started, or \( T_j \) started execution after \( T_i \) finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
- **Durability**: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures. These properties are often called the ACID properties; the acronym is derived from the first letter of each of the four properties.

**Transactions access data using two operations**

- **read(X)**, which transfers the data item \( X \) from the database to a local buffer belonging to the transaction that executed the read operation.
- **write(X)**, which transfers the data item \( X \) from the local buffer of the transaction that executed the write operation back to the database.

Let \( T_i \) be a transaction that transfers $50 from account A to account B. This transaction can be defined as

\[
T_i: \text{read}(A); \\
A := A - 50; \\
\text{write}(A); \\
\text{read}(B); \\
B := B + 50; \\
\text{write}(B).
\]

We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either

- the updates carried out by the transaction have been written to disk before the transaction completes.
- information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.
- ensuring durability is the responsibility of a component of the database system called the recovery-management component.

Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially that is, one after the other. However, concurrent execution of transactions provides significant performance benefits.
6.2 Transaction State

In the absence of failures, all the transactions are completed successfully. However, as we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed aborted. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been rolled back. It is part of the responsibility of the recovery scheme to manage transaction aborts. A transaction that completes its execution successfully is said to be committed. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure. Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a compensating transaction.

We need to be more precise about what we mean by successful completion of a transaction. We therefore establish a simple abstract transaction model. A transaction must be in one of the following states:

- Active, the initial state; the transaction stays in this state while it is executing
- Partially committed, after the final statement has been executed
- Failed, after the discovery that normal execution can no longer proceed
- Aborted, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction
- Committed, after successful completion

The state diagram corresponding to a transaction appears in figure below. We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have terminated if has either committed or aborted. A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

- It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.
It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

For certain applications, it may be desirable to allow active transactions to display data to users, particularly for long-duration transactions that run for minutes or hours. Unfortunately, we cannot allow such output of observable data unless we are willing to compromise transaction atomicity. Most current transaction systems ensure atomicity and, therefore, forbid this form of interaction with users.

### 6.3 Implementation of Atomicity and Durability

The recovery-management component of a database system can support atomicity and durability by a variety of schemes. We first consider a simple, but extremely inefficient, scheme called the shadow copy scheme. This scheme, which is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.

If the transaction completes, it is committed as follows. First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.) After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted. Shown in figure below depicts the scheme, showing the database state before and after the update.
Fig. 6.2 Shadow-copy technique for atomicity and durability

The transaction is said to have been committed at the point where the updated db-pointer is written to disk.

The implementation actually depends on the write to db-pointer being atomic; that is, either all its bytes are written or none of its bytes are written. If some of the bytes of the pointer were updated by the write, but others were not, the pointer is meaningless, and neither old nor new versions of the database may be found when the system restarts.

Thus, the atomicity and durability properties of transactions are ensured by the shadow-copy implementation of the recovery-management component.

6.4 Concurrent Execution

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run serially that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

Improved throughput and resource utilisation
A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk utilisation also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

Reduced waiting time
There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.
The motivation for using concurrent execution in a database is essentially the same as the motivation for using multiprogramming in an operating system.

When several transactions run concurrently, database consistency can be destroyed despite the correctness of each individual transaction. In this section, we present the concept of schedules to help identify those executions that are guaranteed to ensure consistency.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called concurrency-control schemes. Consider again the simplified banking system, which has several accounts, and a set of transactions that access and update those accounts. Let T1 and T2 be two transactions that transfer funds from one account to another. Transaction T1 transfers $50 from account A to account B. It is defined as

\[
\begin{align*}
T1: & \text{ read}(A); \\
& A := A - 50; \\
& \text{ write}(A); \\
& \text{ read}(B); \\
& B := B + 50; \\
& \text{ write}(B).
\end{align*}
\]

Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as

\[
\begin{align*}
T2: & \text{ read}(A); \\
& \text{ temp } := A * 0.1; \\
& A := A - \text{ temp}; \\
& \text{ write}(A); \\
& \text{ read}(B); \\
& B := B + \text{ temp}; \\
& \text{ write}(B)
\end{align*}
\]

Suppose the current values of accounts A and B are $1000 and $2000, respectively. Suppose also that the two transactions are executed one at a time in the order T1 followed by T2. This execution sequence appears in shown in figure below. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T1 appearing in the left column and instructions of T2 appearing in the right column. The final values of accounts A and B, after the execution in Shown in figure below takes place, are $855 and $2145, respectively. Thus, the total amount of money in accounts A and B—that is, the sum $A + B$—is preserved after the execution of both transactions.

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is that of Shown in figure below. Again, as expected, the sum $A + B$ is preserved, and the final values of accounts A and B are $850 and $2150, respectively.

The execution sequences just described are called schedules. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction.
When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions. Several execution sequences are possible, since various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction. Thus, the number of possible schedules for a set of n transactions is much larger than n!.

Table 6.1 Schedule 1: a serial schedule in which T1 is followed by T2

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A := A − 50</td>
<td>temp := A * 0.1</td>
</tr>
<tr>
<td>write (A)</td>
<td>A := A − temp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>B := B + 50</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>B := B + temp</td>
</tr>
</tbody>
</table>

Table 6.2 Schedule 3: a concurrent schedule equivalent to schedule 1

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A := A − 50</td>
<td>temp := A * 0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A := A − temp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>B := B + 50</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>B := B + temp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A := A − 50</td>
<td>temp := A * 0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A := A − temp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>B := B + 50</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>B := B + temp</td>
</tr>
</tbody>
</table>
Not all concurrent executions result in a correct state. If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that gets executed will leave the database in a consistent state. The concurrency-control component of the database system carries out this task.

We can ensure consistency of database under concurrent execution by making sure that any schedule that was executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule.

### 6.5 Serialisability

The database system must control concurrent execution of transactions to ensure that the database state remains consistent. Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not interpret the type of operations that a transaction can perform on a data item.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A= A−50</td>
<td>temp= A * 0.1</td>
</tr>
<tr>
<td></td>
<td>A=A−temp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>B= B + 50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>B= B + temp</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Table 6.3 Schedule 4: a concurrent schedule

Instead, we consider only two operations: read and write. We thus assume that, between a read(Q) instruction and a write(Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. Thus, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. We shall therefore usually show only read and write instructions in schedules, as we do in schedule 3 as shown in the table below. In this section, we discuss different forms of schedule equivalence; they lead to the notions of conflict serialisability and view serialisability.
Table 6.4 Schedule 3: showing only the read and write instructions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>Read(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Conflict serialisability
Let us consider a schedule $S$ in which there are two consecutive instructions $I_i$ and $I_j$, of transactions $T_i$ and $T_j$, respectively ($i = j$). If $I_i$ and $I_j$ refer to different data items, then we can swap $I_i$ and $I_j$ without affecting the results of any instruction in the schedule. However, if $I_i$ and $I_j$ refer to the same data item $Q$, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

- $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$. The order of $I_i$ and $I_j$ does not matter, since the same value of $Q$ is read by $T_i$ and $T_j$, regardless of the order.

- $I_i = \text{read}(Q)$, $I_j = \text{write}(Q)$. If $I_i$ comes before $I_j$, then $T_i$ does not read the value of $Q$ that is written by $T_j$ in instruction $I_j$. If $I_j$ comes before $I_i$, then $T_i$ reads the value of $Q$ that is written by $T_j$. Thus, the order of $I_i$ and $I_j$ matters.

- $I_i = \text{write}(Q)$, $I_j = \text{read}(Q)$. The order of $I_i$ and $I_j$ matters for reasons similar to those of the previous case.

- $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$. Since both instructions are write operations, the order of these instructions does not affect either $T_i$ or $T_j$. However, the value obtained by the next read($Q$) instruction of $S$ is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write($Q$) instruction after $I_i$ and $I_j$ in $S$, then the order of $I_i$ and $I_j$ directly affects the final value of $Q$ in the database state that results from schedule $S$.

Thus, only in the case where both $I_i$ and $I_j$ are read instructions then they perform execution. We say that $I_i$ and $I_j$ conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

To illustrate the concept of conflicting instructions, we consider schedule 3, as shown in figure below. The write($A$) instruction of $T_1$ conflicts with the read($A$) instruction of $T_2$. However, the write($A$) instruction of $T_2$ does not conflict with the read($B$) instruction of $T_1$, because the two instructions access different data items. Let $I_i$ and $I_j$ be consecutive instructions of a schedule $S$. If $I_i$ and $I_j$ are instructions of different transactions and $I_i$ and $I_j$ do not conflict, then we can swap the order of $I_i$ and $I_j$ to produce a new schedule $S'$.

We continue to swap nonconflicting instructions:

- Swap the read($B$) instruction of $T_1$ with the read($A$) instruction of $T_2$.
- Swap the write($B$) instruction of $T_1$ with the write($A$) instruction of $T_2$.
- Swap the write($B$) instruction of $T_1$ with the read($A$) instruction of $T_2$. 
If a schedule $S$ can be transformed into a schedule $S'$ by a series of swaps of nonconflicting instructions, we say that $S$ and $S'$ are conflict equivalent.

The concept of conflict equivalence leads to the concept of conflict serialisability. We say that a schedule $S$ is conflict serialisable if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serialisable, since it is conflict equivalent to the serial schedule 1. Finally, consider schedule 7 of table; it consists of only the significant operations (that is, the read and write) of transactions T3 and T4. This schedule is not conflict serialisable, since it is not equivalent to either the serial schedule $<T3, T4>$ or the serial schedule $<T4, T3>$.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (A)</td>
<td>Read (A)</td>
</tr>
<tr>
<td>Write (A)</td>
<td>Write (A)</td>
</tr>
<tr>
<td>Read (B)</td>
<td>Read (B)</td>
</tr>
<tr>
<td>Write (B)</td>
<td>Write (B)</td>
</tr>
<tr>
<td>Read (Q)</td>
<td>Write (Q)</td>
</tr>
<tr>
<td>Write(Q)</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.5 Schedule 7**

**View serialisability**

In this section, we consider a form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based only on the read and write operations of transactions.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(B)</td>
</tr>
<tr>
<td>A = A − 50</td>
<td>B = B − 10</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(B)</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>A = A + 10</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
</tbody>
</table>

**Table 6.6 Schedule 8**
Consider two schedules S and Q, where the same set of transactions participates in both schedules. The schedules S and S are said to be view equivalent if three conditions are met:

- For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S, also read the initial value of Q.
- For each data item Q, if transaction Ti executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction Tj, then the read(Q) operation of transaction Ti must, in schedule S’, also read the value of Q that was produced by the same write(Q) operation of transaction Tj.
- For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S’.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state. In our previous examples, schedule 1 is not viewed as equivalent to schedule 2, since, in schedule 1, the value of account A read by transaction T2 was produced by T1, whereas this case does not hold in schedule 2. However, schedule 1 is view equivalent to schedule 3, because the values of account A and B read by transaction T2 were produced by T1 in both schedules.

The concept of view equivalence leads to the concept of view serialisability. We say that a schedule S is view serialisable if it is view equivalent to a serial schedule.

Every conflict-serialisable schedule is also view serialisable, but there are view serialisable schedules that are not conflict serialisable.

In schedule 9, transactions T4 and T6 perform write(Q) operations without having performed a read(Q) operation. Writes of this sort are called blind writes. Blind writes appear in any view-serialisable schedule that is not conflict serialisable.

<table>
<thead>
<tr>
<th>T3</th>
<th>T4</th>
<th>T6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (Q)</td>
<td></td>
<td>Write (Q)</td>
</tr>
<tr>
<td>Write (Q)</td>
<td></td>
<td>Write (Q)</td>
</tr>
</tbody>
</table>

Table 6.7 Schedule 9: a view-serialisable schedule

6.6 Recoverability

If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction Tj that is dependent on Ti (that is, Tj has read data written by Ti) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

Recoverable schedules

Consider schedule 11 shown in figure below, in which T9 is a transaction that performs only one instruction: read(A). Suppose that the system allows T9 to commit immediately after executing the read(A) instruction. Thus, T9 commits before T8 does. Now suppose that T8 fails before it commits. Since T9 has read the value of data item A written by T8, we must abort T9 to ensure transaction atomicity. However, T9 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T8. Schedule 11, with the commit happening immediately after the read(A) instruction, is an example of a nonrecoverable schedule, which should not be allowed. Most database systems require that all schedules be recoverable. A recoverable schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the commit operation of Tj.
Cascadeless schedules
Even if a schedule is recoverable, to recover correctly from the failure of a transaction $T_i$, we may have to roll back several transactions. Such situations occur if transactions have read data written by $T_i$. As an illustration, consider the partial schedule of Shown in figure below.

<table>
<thead>
<tr>
<th>T8</th>
<th>T9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (A)</td>
<td></td>
</tr>
<tr>
<td>Write (A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A)</td>
</tr>
<tr>
<td>Read (B)</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8 Schedule 11

<table>
<thead>
<tr>
<th>T10</th>
<th>T11</th>
<th>T12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read (B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write (A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read (A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write (A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read (A)</td>
</tr>
</tbody>
</table>

Table 6.9 Schedule 12

Transaction $T_{10}$ writes a value of $A$ that is read by transaction $T_{11}$. Transaction $T_{11}$ writes a value of $A$ that is read by transaction $T_{12}$. Suppose that, at this point, $T_{10}$ fails. $T_{10}$ must be rolled back. Since $T_{11}$ is dependent on $T_{10}$, $T_{11}$ must be rolled back. Since $T_{12}$ is dependent on $T_{11}$, $T_{12}$ must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.
Summary

• A transaction is a unit of program execution that accesses and possibly updates various data items.

• Usually, a transaction is initiated by a user program written in a high-level data-manipulation language or programming language.

• Atomicity is either all operations in the transaction are reflected properly in the database, or none are.

• Consistency: Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

• Isolation is even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj, it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

• Durability is after a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

• Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

• Ensuring durability is the responsibility of a component of the database system called the recovery-management component.

• The recovery-management component of a database system can support atomicity and durability by a variety of schemes.

• The transaction is said to have been committed at the point where the updated dbpointer is written to disk.

• The implementation actually depends on the write to db-pointer being atomic; that is, either all its bytes are written or none of its bytes is written.

• Transaction-processing systems usually allow multiple transactions to update data concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier.

• A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel.

References


Recommended Reading


Self Assessment

1. A _________ is a unit of program execution that accesses and possibly updates various data items.
   a. transaction
   b. programming
   c. coding
   d. system

2. The _________ consists of all operations executed between the begin transaction and end transaction.
   a. transaction
   b. programming
   c. coding
   d. system

3. _________ is all operations of the transaction are reflected properly either in the database or none is.
   a. Consistency
   b. Isolation
   c. Atomicity
   d. Transaction

4. _________ is execution of a transaction in isolation that is, with no other transaction executing concurrently
   preserves the consistency of the database.
   a. Consistency
   b. Isolation
   c. Atomicity
   d. Transaction

5. In which of the following does multiple transactions execute concurrently?
   a. Consistency
   b. Isolation
   c. Atomicity
   d. Transaction

6. Which of the following statements is true?
   a. A transaction enters the failed state after the system determines that the transaction executing normally.
   b. A transaction enters the failed state after the system determines that the transaction is done.
   c. A transaction enters the failed state after the system determines that the transaction can no longer proceed
      with its normal execution.
   d. A transaction enters the failed state after the system determines that the transaction fix.

7. The _________ component of a database system can support atomicity and durability by a variety of
   schemes.
   a. recovery-management
   b. system
   c. computer
   d. device
8. Transaction-processing systems usually allow multiple transactions to run ____________.
   a. at different time
   b. concurrently
   c. one after another
   d. equally

9. The database system must control concurrent execution of ____________, to ensure that the database state remains consistent.
   a. consistency
   b. isolation
   c. atomicity
   d. transaction

10. The motivation for using concurrent execution in a database is essentially the same as the motivation for using multiprogramming in an/ a ____________.
    a. operating system
    b. system
    c. computer
    d. device
Chapter VII
Concurrency Control

**Aim**

The aim of this chapter is to:

- explain locked based protocol
- describe locks
- discuss granting locks

**Objectives**

The objectives of this chapter are to:

- explain two phase locking protocol
- elucidate timestamp based protocols
- describe Thomas write rule

**Learning outcome**

At the end of this chapter, you will be able to:

- recognise validation based protocol
- identify multiple granularity
- understand validation based protocol
7.1 Lock Based Protocols

One way to ensure serialisability is that the data items must be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

7.1.1 Locks

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

- **Shared**: If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but cannot write, Q.
- **Exclusive**: If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read and write Q.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>X</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 7.1 Lock-compatibility matrix \( \text{comp} \)

- We require that every transaction requests a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q. The transaction makes request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction.

- Given a set of lock modes, we can define a compatibility function on them as follows. Let A and B represent arbitrary lock modes. Suppose that a transaction Ti requests a lock of mode A on item Q on which transaction Tj \( (Ti \neq Tj) \) currently holds a lock of mode B. If transaction Ti can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is compatible with mode B. Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking discussed in this section appears in the matrix \( \text{comp} \) of table 7.1. An element \( \text{comp}(A, B) \) of the matrix has the value true if and only if mode A is compatible with mode B.

- To access a data item, transaction \( Ti \) must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, \( Ti \) is made to wait until all incompatible locks held by other transactions have been released.

\[
\begin{align*}
\text{T1:} & \text{ lock-X(B); } \\
& \text{ read(B); } \\
& \text{ B := B - 50; } \\
& \text{ write(B); } \\
& \text{ unlock(B); } \\
& \text{ lock-X(A); } \\
& \text{ read(A); } \\
& \text{ A := A + 50; } \\
& \text{ write(A); } \\
& \text{ unlock(A). }
\end{align*}
\]

- Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of figure given below for T3 and T4. Since T3 is holding an exclusive-mode lock on B and T4 is requesting a shared-mode lock on B, T4 is waiting for T3 to unlock B.
• Similarly, since T4 is holding a shared-mode lock on A and T3 is requesting an exclusive-mode lock on A, T3 is
waiting for T4 to unlock A. Thus, we have arrived at a state where neither of these transactions can ever proceed
with its normal execution. This situation is called deadlock.
• When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled
back, the data items that were locked by that transaction are unlocked. These data items are then available to
the other transaction, which can continue with its execution.
• We shall require that each transaction in the system follow a set of rules, called a locking protocol, indicating
when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible
schedules.
• The set of all such schedules is a proper subset of all possible serialisable schedules. We shall present several
locking protocols that allow only conflict-serialisable schedules.
• Before doing so, we need a few definitions. Let \{T0, T1, . . . , Tn\} be a set of transactions participating in a
schedule S.
• We say that Ti precedes Tj in S, written Ti → Tj, if there exists a data item Q such that Ti has held lock mode A
on Q, and Tj has held lock mode B on Q later, and comp(A,B) = false. If Ti → Tj , then that precedence implies
that in any equivalent serial schedule, Ti must appear before Tj .
• Conflicts between instructions correspond to non-compatibility of lock modes.
• We say that a schedule S is legal under a given locking protocol if S is a possible schedule for a set of transactions
that follow the rules of the locking protocol.

7.1.2 Granting of Locks
When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the
same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following
scenario.
• Suppose a transaction T2 has a shared-mode lock on a data item, and another transaction T1 requests an
exclusive-mode lock on the data item.
• Clearly, T1 has to wait for T2 to release the shared-mode lock. Meanwhile, a transaction T3 may request a
shared-mode lock on the same data item.
• The lock request is compatible with the lock granted to T2, so T3 may be granted the shared-mode lock.
• At this point T2 may release the lock, but still T1 has to wait for T3 to finish. But again, there may be a new
transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T3 releases
it.
• In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data
item, and each transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-
mode lock on the data item.
• The transaction T1 may never make progress, and is said to be starved.
• We can avoid starvation of transactions by granting locks in the following manner: When a transaction Ti requests a
lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that
• There is no other transaction holding a lock on Q in a mode that conflicts with M.
• There is no other transaction that is waiting for a lock on Q, and that made its lock request before Ti.

7.1.3 Two Phase Locking Protocol
One protocol that ensures serialisability is the two-phase locking protocol. This protocol requires that each transaction
issue lock and unlock requests in two phases:
• Growing phase: A transaction may obtain locks, but may not release any lock.
• Shrinking phase: A transaction may release locks, but may not obtain any new locks.
Cascading rollbacks can be avoided by a modification of two-phase locking called the strict two-phase locking protocol.

Another variant of two-phase locking is the rigorous two-phase locking protocol, which requires that all locks be held until the transaction commits. We can easily verify that, with rigorous two-phase locking, transactions can be serialised in the order in which they commit.

### 7.2 Timestamp Based Protocols

Another method for determining the serialisability order is to select an ordering among transactions in advance. The most common method for doing so is to use a timestamp-ordering scheme.

#### 7.2.1 Timestamps

With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS(Ti). This timestamp is assigned by the database system before the transaction Ti starts execution. If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then TS(Ti) < TS(Tj). There are two simple methods for implementing this scheme:

- Use the value of the system clock as the timestamp; that is, a transaction’s timestamp is equal to the value of the clock when the transaction enters the system.
- Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction’s timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serialisability order. Thus, if TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj. To implement this scheme, we associate with each data item Q two timestamp values:

- W-timestamp(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully.
- R-timestamp(Q) denotes the largest timestamp of any transaction that executed read(Q) successfully.

#### 7.2.2 Timestamp Ordering Protocol

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

- Suppose that transaction Ti issues read(Q).
  - If TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.
  - If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and Rtimestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

- Suppose that transaction Ti issues write(Q).
  - If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls Ti back.
  - If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back.
  - Otherwise, the system executes the write operation and sets W-timestamp (Q) to TS(Ti).

- If a transaction Ti is rolled back by the concurrency-control scheme as result of issuance of either a read or writes operation, the system assigns it a new timestamp and restarts it.

- To illustrate this protocol, we consider transactions T14 and T15. Transaction T14 displays the contents of accounts A and B:

  ```
  T14: read(B);
  read(A);
  display(A + B)
  ```
Transaction T15 transfers $50 from account A to account B, and then displays the contents of both:

T15: read(B);
B := B − 50;
write(B);
read(A);
A := A + 50;
write(A);
display(A + B).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3 of As shown in figure below, TS(T14) < TS(T15), and the schedule is possible under the timestamp protocol.

The timestamp-ordering protocol ensures conflict serialisability. This is because conflicting operations are processed in timestamp order. The protocol ensures freedom from deadlock, since no transaction ever waits.

### 7.2.3 Thomas’ Write Rule

We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol.

- Let us consider schedule 4 of figure given below, and apply the timestamp-ordering protocol. Since T16 starts before T17, we shall assume that TS(T16) < TS(T17).
- The read(Q) operation of T16 succeeds, as does the write(Q) operation of T17. When T16 attempts its write(Q) operation, we find that TS(T16) < W-timestamp(Q), since W-timestamp(Q) = TS(T17). Thus, the write (Q) by T16 is rejected and transaction T16 must be rolled back.
- Although the rollback of T16 is required by the timestamp-ordering protocol, it is unnecessary. Since T17 has already written Q, the value that T16 is attempting to write is one that will never need to be read.
- Any transaction Ti with TS(Ti) < TS(T17) that attempts a read(Q) will be rolled back, since TS(Ti) < W-timestamp(Q). Any transaction Tj with TS(Tj ) > TS(T17) must read the value of Q written by T17, rather than the value written by T16.
- The modification to the timestamp-ordering protocol, called Thomas’ write rule, is this: Suppose that transaction Ti issues write(Q).
  - If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls Ti back.
  - If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
  - Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).
- Thomas’ write rule makes use of view serialisability by, in effect, deleting obsolete write operations from the transactions that issue them. This modification of transactions makes it possible to generate serialisable schedules that would not be possible under the other protocols.

<table>
<thead>
<tr>
<th>T16</th>
<th>T17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(Q)</td>
<td></td>
</tr>
<tr>
<td>Write(Q)</td>
<td>Write(Q)</td>
</tr>
</tbody>
</table>

**Table 7.2 Schedule 4**
Validation-based protocols
In case where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state.

- A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead.
- A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict.
- To gain that knowledge, we need a scheme for monitoring the system.
- We assume that each transaction Ti executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,
  - Read phase: During this phase, the system executes transaction Ti. It reads the values of the various data items and stores them in variables local to Ti. It performs all write operations on temporary local variables, without updates of the actual database.
  - Validation phase: Transaction Ti performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serialisability.
  - Write phase: If transaction Ti succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back Ti.
- To perform the validation test, we need to know when the various phases of transactions Ti took place. We shall, therefore, associate three different timestamps with transaction Ti:
  - Start (Ti), the time when Ti started its execution.
  - Validation (Ti), the time when Ti finished its read phase and started its validation phase.
  - Finish (Ti), the time when Ti finished its write phase.
- We determine the serialisability order by the timestamp-ordering technique, using the value of the timestamp Validation (Ti). Thus, the value TS(Ti) = Validation(Ti) and, if TS(Tj) < TS(Tk), then any produced schedule must be equivalent to a serial schedule in which transaction Tj appears before transaction Tk.
- We determine the serialisability order by the timestamp-ordering technique, using the value of the timestamp Validation (Ti). Thus, the value TS(Ti) = Validation(Ti) and, if TS(Tj) < TS(Tk), then any produced schedule must be equivalent to a serial schedule in which transaction Tj appears before transaction Tk.
- The validation test for transaction Tj requires that, for all transactions Ti with TS(Ti) < TS(Tj), one of the following two conditions must hold:
  - Finish (Ti) < Start (Tj). Since Ti completes its execution before Tj started, the serialisability order is indeed maintained.
  - The set of data items written by Ti does not intersect with the set of data items read by Tj, and Ti completes its write phase before Tj starts its validation phase (Start(Tj) < Finish(Ti) < Validation(Tj)). This condition ensures that the writes of Ti and Tj do not overlap. Since the writes of Ti do not affect the read of Tj, and since Tj cannot affect the read of Ti, the serialisability order is indeed maintained.
This validation scheme is called the optimistic concurrency control scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end.

In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serialisable.

### 7.3 Multiple Granularity

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronisation unit.

- For example, if a transaction Ti needs to access the entire database, and a locking protocol is used, then Ti must lock each item in the database.
- Clearly, executing these locks is time consuming. It would be better if Ti could issue a single lock request to lock the entire database.
- On the other hand, if transaction Tj needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.
- What is needed is a mechanism to allow the system to define multiple levels of granularity.
- We can make one by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree.
- Note that the tree that we describe here is significantly different from that used by the tree protocol.
- A nonleaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.
- As an illustration, consider the tree of as shown in figure below, which consists of four levels of nodes.
- The highest level represents the entire database. Below it is nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children.
- Each area contains exactly those files that are its child nodes. No file is in more than one area.
- Finally, each file has nodes of type record. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file. Each node in the tree can be locked individually.
- As we did in the two-phase locking protocol, we shall use shared and exclusive lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode.
• Suppose that transaction \( T_j \) wishes to lock record \( rb_6 \) of file \( F_b \). Since \( T_i \) has locked \( F_b \) explicitly, it follows that \( rb_6 \) is also locked (implicitly). But, when \( T_j \) issues a lock request for \( rb_6 \), \( rb_6 \) is not explicitly locked. If any node in that path is locked in an incompatible mode, then \( T_j \) must be delayed.

There is an intention mode associated with shared mode, and there is one with exclusive mode.
• If a node is locked in intention-shared (IS) mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks.
• Similarly, if a node is locked in intention-exclusive (IX) mode, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks.
• Finally, if a node is locked in shared and intention-exclusive (SIX) mode, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks.

7.4 Validation Based Protocols
In case where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state.
• A concurrency-control scheme imposes overhead of code execution and possible delay of transactions.
• It may be better to use an alternative scheme that imposes less overhead.
• A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict.
• To gain that knowledge, we need a scheme for monitoring the system.
• We assume that each transaction \( T_i \) executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,
  • Read phase: During this phase, the system executes transaction \( T_i \). It reads the values of the various data items and stores them in variables local to \( T_i \). It performs all write operations on temporary local variables, without updates of the actual database.
  • Validation phase: Transaction \( T_i \) performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serialisability.
  • Write phase: If transaction \( T_i \) succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back \( T_i \).
• Each transaction must go through the three phases in the order shown. However, all three phases of concurrently executing transactions can be interleaved.
To perform the validation test, we need to know when the various phases of transactions Ti took place. We shall, therefore, associate three different timestamps with transaction Ti:

- Start(Ti), the time when Ti started its execution.
- Validation(Ti), the time when Ti finished its read phase and started its validation phase.
- Finish(Ti), the time when Ti finished its write phase.

We determine the serialisability order by the timestamp-ordering technique, using the value of the timestamp Validation(Ti). Thus, the value TS(Ti) = Validation(Ti) and, if TS(Tj) < TS(Tk), then any produced schedule must be equivalent to a serial schedule in which transaction Tj appears before transaction Tk.

The validation test for transaction Tj requires that, for all transactions Ti with TS(Ti) < TS(Tj), one of the following two conditions must hold:

- Finish(Ti) < Start(Tj). Since Ti completes its execution before Tj started, the serialisability order is indeed maintained.
- The set of data items written by Ti does not intersect with the set of data items read by Tj, and Ti completes its write phase before Tj starts its validation phase (Start(Tj) < Finish(Ti) < Validation(Tj)). This condition ensures that the writes of Ti and Tj do not overlap. Since the writes of Ti do not affect the read of Tj, and since Tj cannot affect the read of Ti, the serialisability order is indeed maintained. This validation scheme is called the optimistic concurrency control scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end.

In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serialisable.
Summary

- One way to ensure serialisability is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item.
- When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.
- In a growing phase, a transaction may obtain locks, but may not release any lock.
- In a shrinking phase, a transaction may release locks, but may not obtain any new locks.
- In case where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low.
- A concurrency-control scheme imposes overhead of code execution and possible delay of transactions.
- During read phase, the system executes transaction Ti. It reads the values of various data items and stores them in variables local to Ti. It performs all write operations on temporary local variables, without updates of the actual database.
- Validation phase: Transaction Ti performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serialisability.
- Write phase: If transaction Ti succeeds in validation (step 2), then the system applies the actual updates to the database.

Reference


Recommended Reading

Self Assessment

1. One way to ensure _______________ is that data items be accessed in a mutually exclusive manner.
   a. serialisability
   b. transaction
   c. concurrency control
   d. lock

2. Which of the following makes request to the concurrency control manager?
   a. Serialisability
   b. Transaction
   c. Concurrency control
   d. Lock

3. The transaction can proceed with the operation only after the _______________ manager grants the lock to the transaction.
   a. serialisability
   b. transaction
   c. concurrency control
   d. lock

4. In which of the following phase transaction may obtain locks, but may not release any lock?
   a. Growing phase
   b. Shrinking phase
   c. Read phase
   d. Write phase

5. In ____________ a transaction may release locks, but may not obtain any new locks.
   a. growing phase
   b. shrinking phase
   c. read phase
   d. write phase

6. A ______________ scheme imposes overhead of code execution and possible delay of transactions.
   a. serialisability
   b. transaction
   c. concurrency control
   d. lock

7. The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in__________.
   a. sequential order
   b. timestamp order
   c. linear order
   d. random order
8. _________ rollback can be avoided by a modification of two-phase locking called the strict two-phase locking protocol.
   a. Cascading
   b. Sequential
   c. Timestamp
   d. Linear

9. The most common method for doing serialisability is to use a ____________ scheme.
   a. sequential ordering
   b. timestamp ordering
   c. linear ordering
   d. random ordering

10. The modification to timestamp-ordering protocol, is called _________________.
    a. Thomas’ write rule
    b. Simon’s rule
    c. Newton’s rule
    d. Max rule
Chapter VIII
Recovery System

Aim
The aim of this chapter is to:

• explain failure classification
• elucidate transaction failure
• discuss system crash

Objectives
The objectives of this chapter are to:

• explain disk failure
• describe storage structure
• enlist the storage types

Learning outcome
At the end of this chapter, you will be able to:

• elaborate stable storage implementation
• understand data access
• describe recovery and atomicity
8.1 Failure Classification

A computer system, like any other mechanical or electrical system is subject to failure. There are a variety of causes, including disk crash, power failure, software errors, a fire in the machine room, or even sabotage. Whatever the cause, information may be lost. The database must take actions in advance to ensure that the atomicity and durability properties of transactions are preserved. An integral part of a database system is a recovery scheme that is responsible for the restoration of the database to a consistent stage that existed prior to the occurrence of the failure.

The major types of failures involving data integrity (as opposed to data security) are:

**Transaction failure**
- Logical error
  - The transaction cannot continue with its normal execution because of such things as bad input, data not found, or resource limit exceeded.
- System error
  - The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.

**System crash**
There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of the non-volatile storage remains intact, and is not corrupted.

**Disk failure**
A disk block loses its contents as a result of either a head crash or failure during a data transfer. Copies of data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure.

8.2 Storage Structure

The various data items in the database may be stored and accessed in a number of different storage media. To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of these storage media and their access methods.

**Storage types**
In the storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or non-volatile storage. We review these terms, and introduce another class of storage, called stable storage.

**Volatile storage**
Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.

**Non-volatile storage**
Information residing in non-volatile storage survives system crashes. At the current state of technology, non-volatile storage is slower than volatile storage by several orders of magnitude. This is because disk and tape devices are electromechanical, rather than based entirely on chips, as is volatile storage. In database systems, disks are used for most non-volatile storage.

**Stable storage**
Information residing in stable storage is never lost. Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely.
Stable-storage implementation
To implement stable storage, we need to replicate the needed information in several non-volatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

- RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding.
- Many systems store archival backups of tapes off-site to guard against such disasters. However, since tapes cannot be carried off-site continually, updates since the most recent time that tapes were carried off-site could be lost in such a disaster.
- More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system.
- Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the output is not lost, even in the event of a disaster such as a fire or flood.

Block transfer between memory and disk storage can result in:

- Successful completion: The transferred information arrived safely at its destination.
- Partial failure: A failure occurred in the midst of transfer, and the destination block has incorrect information.
- Total failure: The failure occurred sufficiently early during the transfer that the destination block remains intact.

An output operation is executed as follows:

- Write the information onto the first physical block.
- When the first write completes successfully, write the same information onto the second physical block.
- The output is completed only after the second write completes successfully.

Data access
The database system resides permanently on non-volatile storage (usually disks), and is partitioned into fixed-length storage units called blocks.

- Blocks are the units of data transfer to and from disk, and may contain several data items.
- We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as our banking example.
- Transactions input information from the disk to main memory, and then output the information back onto the disk.
- The input and output operations are done in block units.
- The blocks residing on the disk are referred to as physical blocks; the blocks residing temporarily in main memory are referred to as buffer blocks.
- The area of memory where blocks reside temporarily is called the disk buffer.
- Block movements between disk and main memory are initiated through the following two operations:
  - input(B) transfers the physical block B to main memory.
  - output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.
8.3 Recovery and Atomicity

Consider a simplified banking system and a transaction Ti that transfers $50 from account A to account B, with initial values of A and B being $1000 and $2000, respectively.

- Suppose that a system crash has occurred during the execution of Ti, after output(BA) has taken place, but before output(BB) was executed, where BA and BB denote the buffer blocks on which A and B reside.
- Since the memory contents were lost, we do not know the fate of the transaction; thus, we could invoke one of two possible recovery procedures:
  - Reexecute Ti: This procedure will result in the value of A becoming $900, rather than $950. Thus, the system enters an inconsistent state.
  - Do not reexecute Ti: The current system state has values of $950 and $2000 for A and B, respectively. Thus, the system enters an inconsistent state.

In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work.

- The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit.
- Our goal is to perform either all or no database modifications made by Ti. However, if Ti performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.
- To achieve our goal of atomicity, we must first output information describing the modifications to stable storage, without modifying the database itself.
- We shall assume transactions are executed serially; in other words, only a single transaction is active at a time.

8.4 Log Based Recovery

The most widely used structure for recording database modifications is the log. The log is a sequence of log records, recording all the update activities in the database. There are several types of log records. An update log record describes a single database write. It has these fields:

- Transaction identifier is the unique identifier of the transaction that performed the write operation.
- Data-item identifier is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- Old value is the value of the data item prior to the write.
- New value is the value that the data item will have after the write.
Other special log records exist to record significant events during transaction processing, such as the start of a
transaction and the commit or abort of a transaction. We denote the various types of log records as:

- **<Ti start>**. Transaction Ti has started.
- **<Ti, Xj, V1, V2>**. Transaction Ti has performed a write on data item Xj. Xj had value V1 before the write, and
  will have value V2 after the write.
- **<Ti commit>**. Transaction Ti has committed.
- **<Ti abort>**. Transaction Ti has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the
database is modified. Once a log record exists, we can output the modification to the database if that is desirable.
Also, we have the ability to undo a modification that has already been output to the database. We undo it by using
the old-value field in log records. For log records to be useful for recovery from system and disk failures, the log
must reside in stable storage. For now, we assume that every log record is written to the end of the log on stable
storage as soon as it is created.

**Deferred database modification**
The deferred-modification technique ensures transaction atomicity by recording all database modifications in the
log, but deferring the execution of all write operations of a transaction until the transaction partially commits.

- Recall that a transaction is said to be partially committed once the final action of the transaction has been
  executed.
- The version of the deferred-modification technique that we describe in this section assumes that transactions
  are executed serially.
- The execution of transaction Ti proceeds as follows. Before Ti starts its execution, a record **<Ti start>** is written
to the log.
- A write(X) operation by Ti results in the writing of a new record to the log. Finally, when Ti partially commits,
a record **<Ti commit>** is written to the log.
- When transaction Ti partially commits, the records associated with it in the log are used in executing the deferred
  writes. Since a failure may occur while this updating is taking place, we must ensure that, before the start of
  these updates, all the log records are written out to stable storage.
- To illustrate, reconsider our simplified banking system. Let T0 be a transaction that transfers $50 from account
  A to account B:
  T0: read(A);
  A := A − 50;
  write(A);
  read(B);
  B := B + 50;
  write(B).

Let T1 be a transaction that withdraws $100 from account C:

- T1: read(C);
- C := C − 100;
- write(C)

**Immediate database modification**
The immediate-modification technique allows database modifications to be output to the database while the transaction
is still in the active state.

- Data modifications written by active transactions are called uncommitted modifications.
- Before a transaction Ti starts its execution, the system writes the record **<Ti start>** to the log.
- During its execution, any write(X) operation by Ti is preceded by the writing of the appropriate new update
  record to the log. When Ti partially commits, the system writes the record **<Ti commit>** to the log.
Checkpoints
When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

- The search process is time consuming.
- Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database.

Although redoing them will cause no harm, it will nevertheless cause recovery to take longer. To reduce these types of overhead, we introduce checkpoints. In addition, the system periodically performs checkpoints, which require the following sequence of actions to take place:

- Output onto stable storage all log records currently residing in main memory.
- Output to the disk all modified buffer blocks.
- Output onto stable storage a log record <checkpoint>.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress. The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure.

8.5 Recovery with Concurrent Transaction
We considered recovery in an environment where only a single transaction at a time is executing. We now discuss how we can modify and extend the log-based recovery scheme to deal with multiple concurrent transactions. Regardless of the number of concurrent transactions, the system has a single disk buffer and a single log. All transactions share the buffer blocks. We allow immediate modification, and permit a buffer block to have data items updated by one or more transactions.

Interaction with concurrency control:
The recovery scheme depends greatly on the concurrency-control scheme that is used. To roll back a failed transaction, we must undo the updates performed by the transaction.

Transaction rollback
We roll back a failed transaction, Ti, by using the log. The system scans the log backward; for every log record of the form <Ti, Xj, V1, V2> found in the log, the system restores the data item Xj to its old value V1. Scanning of the log terminates when the log record <Ti, start> is found.

Scanning the log backward is important, since a transaction may have updated a data item more than once. As an illustration, consider the pair of log records

<Ti, A, 10, 20>
<Ti, A, 20, 30>

The log records represent a modification of data item A by Ti, followed by another modification of A by Ti. Scanning the log backward sets A correctly to 10. If the log were scanned in the forward direction, A would be set to 20, which is incorrect. If strict two-phase locking is used for concurrency control, locks held by a transaction T may be released only after the transaction has been rolled back.
Checkpoints
We used checkpoints to reduce the number of log records that the system must scan when it recovers from a crash. Since we assumed no concurrency, it was necessary to consider only the following transactions during recovery:
- Those transactions that started after the most recent checkpoint
- The one transaction, if any, that was active at the time of the most recent checkpoint. The situation is more complex when transactions can execute concurrently, since several transactions may have been active at the time of the most recent checkpoint.

In a concurrent transaction-processing system, we require that the checkpoint log record be of the form <checkpoint L>, where L is a list of transactions active at the time of the checkpoint. Again, we assume that transactions do not perform updates either on the buffer blocks or on the log while the checkpoint is in progress. A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.

Restart recovery
When the system recovers from a crash, it constructs two lists: The undo-list consists of transactions to be undone, and the redo-list consists of transactions to be redone. The system constructs the two lists as follows: Initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint> record:
- For each record found of the form <Ti commit>, it adds Ti to redo-list.
- For each record found of the form <Ti start>, if Ti is not in redo-list, then it adds Ti to undo-list.

When the system has examined all the appropriate log records, it checks the list L in the checkpoint record. For each transaction Ti in L, if Ti is not in redo-list then it adds Ti to the undo-list.

Once the redo-list and undo-list have been constructed, the recovery proceeds as follows:

Step 1:
The system rescans the log from the most recent record backward, and performs an undo for each log record that belongs to the transaction Ti on the undo-list. Log records of transactions on the redo-list are ignored in this phase. The scan stops when the <Ti start> records have been found for every transaction Ti in the undo-list.

Step 2:
The system locates the most recent <checkpoint L> record on the log. Notice that this step may involve scanning the log forward, if the checkpoint record was passed in step 1.

Step 3:
The system scans the log forward from the most recent <checkpoint L> record, and performs redo for each log record that belongs to a transaction Ti that is on the redo-list. It ignores log records of transactions on the undo-list in this phase.

The recovery process has completed transaction processing resumes. It is important to undo the transaction in the undo-list before redoing transactions in the redo-list, using the algorithm in steps 1 to 3; otherwise, a problem may occur.
A computer system, like any other mechanical or electrical system is subject to failure.

- A disk block loses its contents as a result of either a head crash or failure during a data transfer.
- Various data items in the database may be stored and accessed in a number of different storage media.
- Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory.
- Information residing in non-volatile storage survives system crashes. At the current state of technology, non-volatile storage is slower than volatile storage by several orders of magnitude.
- Information residing in stable storage is never lost.
- RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding.
- The database system resides permanently on non-volatile storage (usually disks), and is partitioned into fixed-length storage units called blocks.
- Transactions input information from the disk to main memory, and then output the information back onto the disk.
- The input and output operations are done in block units.
- Transaction identifier is the unique identifier of the transaction that performed the write operation.
- Data-item identifier is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- The deferred-modification technique ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits.
- The immediate-modification technique allows database modifications to be output to the database while the transaction is still in the active state.

References


Recommended Reading

1. A disk block loses its contents as a result of either a head crash or failure during a data _____________.
   a. transfer
   b. storage
   c. copying
   d. removing

2. Information residing in ____________ storage does not usually survive system crashes.
   a. non-volatile
   b. temporary
   c. permanent
   d. volatile

3. Which of the following statements is true?
   a. Information residing in non-volatile storage survives system crashes.
   b. Information residing in volatile storage survives system crashes.
   c. Information residing in ROM storage survives system crashes.
   d. Information residing in RAM storage survives system crashes.

4. Non-volatile storage is ________ than volatile storage by several orders of magnitude.
   a. faster
   b. slower
   c. active
   d. inactive

5. Information residing in ____________ is never lost.
   a. non-volatile
   b. temporary
   c. stable storage
   d. volatile

6. ____________ are the units of data transfer to and from disk, and may contain several data items.
   a. Log
   b. Blocks
   c. Memory
   d. Disk buffer

7. Which of the following is the area of memory where blocks reside temporarily?
   a. Log
   b. Blocks
   c. Memory
   d. Disk buffer
8. The _________ is a sequence of log records, recording all the update activities in the database.
   a. Log
   b. Blocks
   c. Memory
   d. Disk buffer

9. Transaction identifier is the unique identifier of the transaction that performed the _________ operation.
   a. read
   b. copy
   c. save
   d. write

10. Data modifications written by active___________ are called uncommitted modifications.
    a. transactions
    b. memory
    c. data
    d. method
Entity Relationship Diagram for Electronic Resource Management

The following diagram is the complete Entity Relationship Diagram for Electronic Resource Management. It presents an abstract, theoretical view of the major entities and relationships needed for management of electronic resources:
Additional Inheritance Constraints
Unlike other modelling systems, entity-relationship diagrams do not explicitly depict situations in which entity instances are expected to inherit attributes or relationships of related entity instances.

This data model assumes the following relationship inheritance constraints:

• An e-resource may inherit its interface’s relationship with an acquisition, administrative information, or access information. Alternatively, the e-resource may have a separate acquisition administrative information, and/or access information from its interface.

• An e-resource must inherit its interface’s relationship or relationships with licenses (and with the terms defined by the interface’s licenses—although different terms from a different license may prevail for the e-resource).

• An e-resource may inherit its parent e-resource’s relationship with an acquisition, administrative information, or access information. Alternatively, the child e-resource may have a separate acquisition administrative information, and/or access information from its parent.

• An e-resource must inherit its parent e-resource’s relationship or relationships with licenses (and with the terms defined by the parent’s licenses—although different terms from a different license may prevail for the child e-resource).

Explanation of complex relationships or concepts
Although the overall diagram is large and complex, most of the entities and relationships depicted in the ERD are relatively straightforward, and can be fully understood in consultation with the definitions provided in Appendix E (Data Element Dictionary) and Appendix F (Data Structure).

Electronic product and its subtypes
The electronic product entity represents electronic things that may be acquired, licensed, or managed. Electronic product encompasses interfaces, e-resource packages, packages of packages, and individual e-resource titles.

Many attributes and relationships are shared by all electronic products, regardless of whether the e-product is an interface, package, or individual title. For example, all types of e-products may be licensed, may be acquired, may undergo a product trial, etc. On the other hand, interfaces and e-resources do have some unique characteristics and in certain cases need to be handled in distinct ways. The electronic product entity and its subtypes accommodate the depiction of these relationships.

E-Resource recursive relationship
The e-resource entity represents e-resource packages, packages of packages, and individual e-resource titles. Any given e-resource title may be a standalone product, or may be a part of an e-resource package. An e-resource package could in turn be part of a larger package. These possible relationships are accommodated with the optional recursive relationship for e-resources:
The organisation entity
The organisation entity represents any business, vendor, provider, publisher, licensor, etc. with which a library does business related to electronic products. Generally, one records the same attributes of an organisation, such as name and address, regardless of the role the organisation plays. Furthermore, a single organisation frequently fills several roles in the e-resource management environment, functioning as vendor, licensor, publisher, and/or provider.

Rather than create separate entities for each role, which may contain duplicate instances of one another, the model contains the single organisation entity, with the role or roles of the organisation dependent on the relationship or relationships in which the organisation participates:

For definitions of the roles indicated by these relationships, see Appendix E (Data Element Dictionary) and Appendix F (Data Structure).

Terms defined and prevailing terms
Each license and each acquisition defines (or may define) a set of associated rights and restrictions on the usage and management of the e-products covered by the license or acquisition. In some cases, certain terms are defined in licenses, while in other agreements those same term concepts might be defined in the business agreement. To accommodate the uncertainty of where certain rights, responsibilities, and other terms might be defined in any given situation, we have modelled a separate “terms defined” entity:
The terms defined entity is able to record all possible rights, responsibilities, and other terms, including terms that are typically defined in licenses (such as “interlibrary loan” or “scholarly sharing” permissions), as well as terms that are typically defined in acquisitions or business terms (such as “number of concurrent users”).

Each acquisition may define at most one terms defined instance. Each license may define at most one terms defined instance. Each terms defined instance is defined either by one acquisition or by one license; this relationship is modelled using the exclusive-or relationship.

Use, rights, and restrictions for any given electronic resource may be governed by multiple agreements, both legal license terms and negotiated acquisition terms. Each acquisition defines one set of terms, and each license defines one set of terms. Through each e-product’s relationship with an acquisition and its relationship with one or more licenses, one can find all of the “terms defined” instances that apply to each e-resource.

It may occur that among several terms defined instances that apply to a given e-product, some of the individual terms are in conflict; this is especially likely in situations where multiple license agreements cover a single e-product. In such cases it is necessary to determine which of the conflicting terms prevail for the e-product in question. This is modelled with the prevailing terms entity, and its relationships with the terms defined entity:

The prevailing terms entity represents all possible terms, just as the terms defined entity does, but for each term merely points to the terms defined entity that prevails. The three relationships lines depicted in the drawing actually represent many relationships: one for each defined term. So while each electronic product may be governed by many different acquisitions and licenses that define terms, each electronic product is governed by at most one complete set of prevailing terms.
Conclusion
This entity-relationship diagram depicts the major concepts and relationships needed for managing electronic resources. It is neither a complete data model depicting every necessary relational database table, nor is it meant to be a prescriptive design for implementations of electronic resource management systems. Alternate models may capture the necessary attributes and relationships. Hopefully this document, along with the other components of the Report of the DLF Electronic Resource Management Initiative will assist developers with envisioning the complexity of the environment that an ERM system must address, and ensure that crucial relationships and features will be included in ERM products.


Questions

1. What are the inheritance constraints in ER Diagram?

   Answer
   Unlike other modelling systems, entity-relationship diagrams do not explicitly depict situations in which entity instances are expected to inherit attributes or relationships of related entity instances.

   This data model assumes the following relationship inheritance constraints:
   • An e-resource may inherit its interface’s relationship with an acquisition, administrative information, or access information. Alternatively, the e-resource may have a separate acquisition administrative information, and/or access information from its interface.
   • An e-resource must inherit its interface’s relationship or relationships with licenses (and with the terms defined by the interface’s licenses—although different terms from a different license may prevail for the e-resource).
   • An e-resource may inherit its parent e-resource’s relationship with an acquisition, administrative information, or access information. Alternatively, the child e-resource may have a separate acquisition administrative information, and/or access information from its parent.
   • An e-resource must inherit its parent e-resource’s relationship or relationships with licenses (and with the terms defined by the parent’s licenses—although different terms from a different license may prevail for the child e-resource).

2. What is an Organisation Entity?

   Answer
   The organisation entity represents any business, vendor, provider, publisher, licensor, etc. with which a library does business related to electronic products. Generally, one records the same attributes of an organisation, such as name and address, regardless of the role the organisation plays. Furthermore, a single organisation frequently fills several roles in the e-resource management environment, functioning as vendor, licensor, publisher, and/or provider.

3. Which are the Electronic product & their subtypes defined in this ER Diagram?

   Answer
   The electronic product entity represents electronic things that may be acquired, licensed, or managed. Electronic product encompasses interfaces, e-resource packages, packages of packages, and individual e-resource titles.
Many attributes and relationships are shared by all electronic products, regardless of whether the e-product is an interface, package, or individual title. For example, all types of e-products may be licensed, may be acquired, may undergo a product trial, etc. On the other hand, interfaces and e-resources do have some unique characteristics and in certain cases need to be handled in distinct ways. The electronic product entity and its subtypes accommodate the depiction of these relationships.
Application II

Normalisation Performance Evaluation Scores

Abstract
Rating tendencies of Managers in an organisation vary from a ‘very lenient’ rate to a ‘very harsh’ rate. The employees reporting to them experience the impact of these variations. This impact becomes crucial in an environment where employees are given performance based remuneration/ incentives. Normalisation of scores is intended to introduce greater objectivity in the Employees Performance Management (EPM) System of an organisation.

Rating patterns of managers

![Rating Patterns of Managers](image)

It may be observed from the figure above that Manager ‘A’, compared to Manager ‘B’, has the tendency to rate most of his subordinates at 7 to 8 points on a Performance rating scale of 1 to 10. Manager ‘C’, on the other hand, is highly conservative and awards given to the best of his subordinates are in the range of 5- 6 points. Thus an average performer with Manager ‘B’ gets equated with the Best employee reporting to Manager ‘C’ and with a low average subordinate of Manager ‘A’. The training of Managers A, B & C on the rating norms may improve this trend marginally during the subsequent years. But what happens to the evaluations already done by them? How has the company management looked into this problem which has an impact on promotions, compensations and career management of all employees? Some outstanding performers (placed under a harsh rate like Manager ‘C’) may quit the organisation and some low caliber people (placed under a lenient rate like Manager ‘A’) may themselves be the Managers of tomorrow. This vicious cycle tends to boost the average performers who cling to their jobs and promote mediocrity in the organisation. A performance driven company can ill afford the luxury of losing their best talent. They have to normalise the rating trends of their Managers and weigh employees’ performance in the correct perspective.

What does normalisation mean?
Assume there are ten Managers in an organisation who are reporting on 10 different executives each and these Managers, in turn, report to three different Senior Managers, in their respective departments. In this scenario of EPM, there are 13 different ‘Appraisers’ who are reporting on 110 employees in the organisation. Amongst these employees, 100 are at the same level (i.e., Executives) and 10 are at the level of Managers. Each of the thirteen Appraisers has a rating style which is different from the others. So the employees reporting to them have a high degree of variability in their performance appraisal scores. The process of balancing this variability is called ‘Normalisation’.
Normalisation process
The process comprises of the following steps:

- Statistical Mean of organisational rating pattern of all the Managers (ie, Appraisers) at the same level, across various departments, is computed. Let this Mean be ‘M’.
- Statistical Mean for each of the Appraisers at the same level (i.e., for all the 10 Managers in the example given above) is computed. Each Manager should have done the Appraisal for 40 to 50 employees (may be over the last 5 years). Let this Mean be ‘Mi’ (i = 1 to 10).
- A correction Factor (CF) for each of these Managers (Appraisers) is then computed = Mi/M. Its value, for example, will be 1.0 if the rating pattern of a Manager is the same as the statistical Mean for all the Managers.
- Performance Score of each individual employee is divided by CF for his/her Manager to compute its normalised value. This normalised score is utilised for all management decisions.

Connected issues
1. To implement the normalisation process, there may be a need to extract information from the Appraisals completed during the last few years so that the statistical Mean for each Manager could be computed based on the 40 - 50 appraisal reports written by him/her. This requires a data base approach.
2. When a Manager has not completed 40 to 50 appraisals, how to find out his/her Correction Factor (CF)? The approach, normally adopted, is to compute the mean Mi & CF for the Manager based on the number of Appraisals so far completed, but not to apply the CF to the Performance scores of the employees without discreet approval by the Management. (Note: Some organisations prefer to use in such cases, the normalised score awarded by the Reviewer).
3. Since the Appraisers tend to refine their rating tendencies automatically as the years roll by, some organisations prefer to use the Moving Mean concept (MMi) for each Manager rather than the Mi. The moving mean for a Manager is an indicator of the improvement in his/her rating trend. CF = MMi/M is the computing algorithm used in this case.
4. In mid to large organisations, with 4 to 5 different levels of Appraisers, a number of departments/functions and employees varying in strength from 200 to 5000 (+), it is not possible to normalise the Appraisal scores without automation of the entire process.

Conclusion
In some organisations, normalisation of performance scores is managed by a high level committee and the process is not made transparent to the employees. If such committees keep in view the rating patterns of various managers such as MMi (discussed above), their decisions would be data based & hence objective.

‘EmpXtrack Employee Appraisal’ caters, in its enterprise level solution, all the normalisation features discussed above. Whether to utilise the auto-normalisation process or not is an option available to the client.


Questions
1. How can Normalisation help in performance evaluation?
2. What is the process of Normalisation carried out by EmpXtrack?
3. What are the rating patterns of Managers?
Application III

Mike’s Bookshop

Let’s say you were looking to start an online bookstore. You would need to track certain information about the books available to your site viewers, such as:

- Title
- Author
- Author Biography
- ISBN
- Price
- Subject
- Number of Pages
- Publisher
- Description
- Review
- Reviewer Name

Let’s start by adding a couple of books written by Luke Welling and Laura Thomson. Because this book has two authors, we are going to need to accommodate both in our table. Let’s take a look at a typical approach I encounter:

<table>
<thead>
<tr>
<th>Title</th>
<th>Author1</th>
<th>Author2</th>
<th>ISBN</th>
<th>Subject</th>
<th>Pages</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP and MySQL Web Development</td>
<td>Luke</td>
<td>Laura</td>
<td>0672317842</td>
<td>PHP, MySQL</td>
<td>867</td>
<td>Sams</td>
</tr>
<tr>
<td>MySQL Tutorial</td>
<td>Welling</td>
<td>Thomson</td>
<td>0672325845</td>
<td>MySQL</td>
<td>300</td>
<td>Sams</td>
</tr>
</tbody>
</table>

**Let’s take a look at some issues involved in this table design:**

First, this table is not very efficient with storage. Let’s imagine for a second that Luke and Laura were extremely busy writers and managed to produce 500 books for our database. The combination of their two names is 25 characters long, and since we will repeat their two names in 500 rows we are wasting $25 \times 500 = 12,500$ bytes of storage space unnecessarily.

Second, this design does not protect data integrity. Let’s once again imagine that Luke and Laura have written 500 books. Someone has had to type their names into the database 500 times, and it is very likely that one of their names will be misspelled at least once (i.e., Thompson instead of Thomson). Our data is now corrupt, and anyone searching for book by author name will find some of the results missing. The same thing could happen with publisher name. Sams publishes hundreds of titles and if the publisher’s name were misspelled even once the list of books by publisher would be missing titles.

Third, this table does not scale well. First of all, we have limited ourselves to only two authors, yet some books are written by over a dozen people. This kind of limitation is often exhibited in personal info tables where the typical design includes Phone1, Phone2, and Phone3 columns. In both cases we are limiting future growth of our data. Another limitation to scalability is that fact that with all our data in one table our one table file will grow faster and we will have more trouble with file size limitations of our underlying operating system.

**First Normal Form**

The normalisation process involves getting our data to conform to three progressive normal forms, and a higher level of normalisation cannot be achieved until the previous levels have been achieved (there are actually five normal forms, but the last two are mainly academic and will not be discussed). The First Normal Form (or 1NF) involves removal of redundant data from horizontal rows. We want to ensure that there is no duplication of data in a given
row, and that every column stores the least amount of information possible (making the field atomic).

In our table above we have two violations of First Normal Form: First, we have more than one author field, and our subject field contains more than one piece of information. With more than one value in a single field, it would be very difficult to search for all books on a given subject. In addition, with two author fields we have two fields to search in order to look for a book by a specific author.

We could get around these problems by modifying our table to have only one author field, with two entries for a book with two authors, as in the following example.

<table>
<thead>
<tr>
<th>Title</th>
<th>Author</th>
<th>ISBN</th>
<th>Subject</th>
<th>Pages</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP and MySQL Web Development</td>
<td>Luke Welling</td>
<td>0672317842</td>
<td>MySQL</td>
<td>867</td>
<td>Sams</td>
</tr>
<tr>
<td>PHP and MySQL Web Development</td>
<td>Laura Thomson</td>
<td>0672317842</td>
<td>PHP</td>
<td>867</td>
<td>Sams</td>
</tr>
<tr>
<td>MySQL Tutorial</td>
<td>Laura Thomson</td>
<td>0672325845</td>
<td>MySQL</td>
<td>300</td>
<td>Sams</td>
</tr>
<tr>
<td>MySQL Tutorial</td>
<td>Luke Welling</td>
<td>0672325845</td>
<td>MySQL</td>
<td>300</td>
<td>Sams</td>
</tr>
</tbody>
</table>

While this approach has no redundant columns and the subject column has only one piece of information, we do have a problem that we now have two rows for a single book. Also, to ensure that we can do a search of author and subject (i.e. books on PHP by Luke Welling), we would need four rows to ensure that we had each combination of author and subject. Additionally, we would be violating the Second Normal Form, which will be described below.

A better solution to our problem would be to separate the redundant data into separate tables, with the tables related to each other. In this case we will create an Author table and a Subject table to store our information, removing that information from the Book table:

**Author Table**

<table>
<thead>
<tr>
<th>Author_ID</th>
<th>Last Name</th>
<th>First Name</th>
<th>Bio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Welling</td>
<td>Luke</td>
<td>Author of books on PHP and MySQL</td>
</tr>
<tr>
<td>2</td>
<td>Thomson</td>
<td>Laura</td>
<td>Another author of books on PHP and MySQL</td>
</tr>
</tbody>
</table>

**Subject Table**

<table>
<thead>
<tr>
<th>Subject_ID</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PHP</td>
</tr>
<tr>
<td>2</td>
<td>MySQL</td>
</tr>
</tbody>
</table>
While creating the Author table, I also split author name down into separate first name and last name columns, in order to follow the requirement for storing as little information as possible in a given column. Each table has a primary key value which can be used for joining tables together when querying the data. A primary key value must be unique with in the table (no two books can have the same ISBN number), and a primary key is also an INDEX, which speeds up data retrieval based on the primary key.

**Defining Relationships**
As you can see, while our data is now split up, relationships between the table have not been defined. There are various types of relationships that can exist between two tables:

- One to (Zero or) One
- One to (Zero or) Many
- Many to Many

The relationship between the Book table and the Author table is a many-to-many relationship: A book can have more than one author, and an author can write more than one book. To represent a many-to-many relationship in a relational database we need a third table to serve as a link between the two:

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Author_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0672317842</td>
<td>1</td>
</tr>
<tr>
<td>0672317842</td>
<td>2</td>
</tr>
<tr>
<td>0672325845</td>
<td>1</td>
</tr>
<tr>
<td>0672325845</td>
<td>2</td>
</tr>
</tbody>
</table>

Similarly, the Subject table also has a many-to-many relationship with the Book table, as a book can cover multiple subjects, and a subject can be explained by multiple books:

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Subject_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0672317842</td>
<td>1</td>
</tr>
<tr>
<td>0672317842</td>
<td>2</td>
</tr>
<tr>
<td>0672325845</td>
<td>2</td>
</tr>
</tbody>
</table>

The one-to-many relationship will be covered in the next section. As you can see, we now have established the relationships between the Book, Author, and Subject tables. A book can have an unlimited number of authors, and can refer to an unlimited number of subjects. We can also easily search for books by a given author or referring to a given subject.
In the linking tables above, the values stored refer to primary key values from the Book, Author, and Subject tables. Columns in a table that refer to primary keys from another table are known as foreign keys, and serve the purpose of defining data relationships. In database systems which support referential integrity, such as the InnoDB storage engine for MySQL, defining a column as a foreign key will also allow database software to enforce the relationships you define. For example, with foreign keys defined, the InnoDB storage engine will not allow you to insert a row into the Book_SUBJECT table unless the book and subject in question already exist in the book and subject tables. Such systems will also prevent the deletion of books from the book table that have ‘child’ entries in the book_subject or book_author tables.

Second Normal Form
Where the First Normal Form deals with redundancy of data across a horizontal row, Second Normal Form (or 2NF) deals with redundancy of data in vertical columns. As stated earlier, the normal forms are progressive, so to achieve Second Normal Form, your tables must already be in First Normal Form. Let’s take another look at our new Book table:

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Pages</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>0672317842</td>
<td>PHP and MySQL Web Development</td>
<td>867</td>
<td>Sams</td>
</tr>
<tr>
<td>0672325845</td>
<td>MySQL Tutorial</td>
<td>300</td>
<td>Sams</td>
</tr>
</tbody>
</table>

This table is in First Normal Form; we do not repeat the same data in any two columns, and each column holds only the minimum amount of data. However, in the Publisher column, we have the same publisher repeating in each row. This is a violation of Second Normal Form. As I stated above, this leaves the chance for spelling errors to occur. The user needs to type in the publisher name each time and such an approach is also inefficient in regards to storage usage, and the more data a table has, the more IO requests are needed to scan the table, resulting in slower queries.

To normalise this table to Second Normal Form, we will once again break data off to a separate table. In this case we will move publisher information to a separate table as follows:

<table>
<thead>
<tr>
<th>Publisher_ID</th>
<th>Publisher Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sams</td>
</tr>
</tbody>
</table>

In this case we have a one-to-many relationship between the book table and the publisher. A given book has only one publisher (for our purposes), and a publisher will publish many books. When we have a one-to-many relationship, we place a foreign key in the many, pointing to the primary key of the one. Here is the new Book table:

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Pages</th>
<th>Publisher_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0672317842</td>
<td>PHP and MySQL Web Development</td>
<td>867</td>
<td>1</td>
</tr>
<tr>
<td>0672325845</td>
<td>MySQL Tutorial</td>
<td>300</td>
<td>1</td>
</tr>
</tbody>
</table>
RDBMS (Oracle)

Since the Book table is the ‘many’ portion of our one-to-many relationship, we have placed the primary key value of the ‘one’ portion in the Publisher_ID column as a foreign key.

The other requirement for Second Normal Form is that you cannot have any data in a table with a composite key that does not relate to all portions of the composite key. A composite key is a primary key that incorporates more than one column, as with our Book_Author table:

Let’s say we wanted to add in tracking of reviews and started with the following table

**Review Table**

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Reviewer_ID</th>
<th>Review_Text</th>
<th>Reviewer_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0672325845</td>
<td>1</td>
<td>What a great book! I learned a lot!</td>
<td>Mike Hillyer</td>
</tr>
</tbody>
</table>

In this table the combination of Reviewer_ID and ISBN form the primary key (a composite key), ensuring that no reviewer writes more than one review for the same book. In this case the reviewer name is only related to the Reviewer_ID, and not the ISBN, and is therefore in violation of Second Normal Form.

In this case we solve the problem by having a separate table for the data that relates to only one part of the key. If the portion of the primary key that the field relates to is a foreign key to another table, move the data there (as in this situation where the Reviewer_ID will be a separate table)

**Reviewer Table**

<table>
<thead>
<tr>
<th>Reviewer_ID</th>
<th>Reviewer_Name</th>
<th>Reviewer_House</th>
<th>Reviewer_Street</th>
<th>Reviewer_City</th>
<th>Reviewer_Province</th>
<th>Reviewer_PostalCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mike Hillyer</td>
<td>123</td>
<td>Main Street</td>
<td>Calgary</td>
<td>Alberta</td>
<td>T1S-2N2</td>
</tr>
</tbody>
</table>

**Review Table**

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Reviewer_ID</th>
<th>Review_Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>0672325845</td>
<td>1</td>
<td>What a great book! I learned a lot!</td>
</tr>
</tbody>
</table>

**Third Normal Form**

I do not often use Third Normal Form. In Third Normal Form we are looking for data in our tables that is not fully dependant on the primary key, but dependant on another value in the table. In the reviewer table above the Reviewer_Street and Reviewer_City fields are really dependant on the Reviewer_PostalCode and not the Reviewer_ID. To bring this table into compliance with Third Normal Form, we would need a table based on Postal Code.

**PostalCode Table**

<table>
<thead>
<tr>
<th>Postal_Code</th>
<th>Street</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S-2N2</td>
<td>Main Street</td>
<td>Calgary</td>
</tr>
</tbody>
</table>
The point of the last example is that Normalisation is a trade-off. In fact, there are two additional normal forms that are generally recognised, but are mainly academic. Additional normalisation results in more complex JOIN queries and this can cause a decrease in performance. In some cases, performance can even be increased by de-normalising table data, but de-normalisation is beyond the scope of this article.

**Conclusion**

It is generally a good idea to make sure that your data at least conforms to Second Normal Form. Our goal is to avoid data redundancy to prevent corruption and make the best possible use of storage. Make sure that the same value is not stored in more than one place. With data in multiple locations we have to perform multiple updates when data needs to be changed, and corruption can begin to creep into the database. Third Normal Form removed even more data redundancy, but at the cost of simplicity and performance. In our example above, do we really expect the City and Street names to change very regularly? In this situation Third Normal Form still prevents misspelling of City and Street names, but it is up to you to find a balance between Normalisation and Speed/Simplicity. In an upcoming article we will look at how to form queries that join our normalised data together.


**Questions**

1. What is the requirement for a good Database Design?
2. What are the anomalies of Database Design?
3. What are “not very efficient” tables known as in Database Design?
Bibliography

References


• Shortt, P. and Huntanar, M., 2011. *Entity Relationship Modelling* [Video Online] Available at: <http://www.youtube.com/watch?v=mQ4D0drMrYI> [Accessed 02 December 2011].


• *The Relational Data Model: Formalisms on Design* [pdf] Available at: <http://myweb.lmu.edu/dondi/share/db/relational-design1.pdf> [Accessed 02 December 2011].


Recommended Reading

- Date, J. C., 2006. *An Introduction to Database Systems*, Pearson Education India.
Self Assessment Answers

Chapter I
1. a
2. c
3. a
4. b
5. c
6. a
7. d
8. b
9. a
10. d

Chapter II
1. c
2. a
3. d
4. c
5. a
6. a
7. b
8. c
9. c
10. a

Chapter III
1. a
2. b
3. c
4. a
5. d
6. b
7. a
8. a
9. c
10. a

Chapter IV
1. a
2. c
3. a
4. a
5. d
6. a
7. d
8. b
9. b
10. a
Chapter V
1. a
2. b
3. b
4. a
5. c
6. d
7. a
8. c
9. a
10. d

Chapter VI
1. a
2. a
3. c
4. a
5. b
6. c
7. a
8. b
9. d
10. a

Chapter VII
1. a
2. b
3. c
4. a
5. b
6. c
7. b
8. a
9. b
10. a

Chapter VIII
1. a
2. d
3. a
4. b
5. c
6. b
7. d
8. a
9. d
10. a