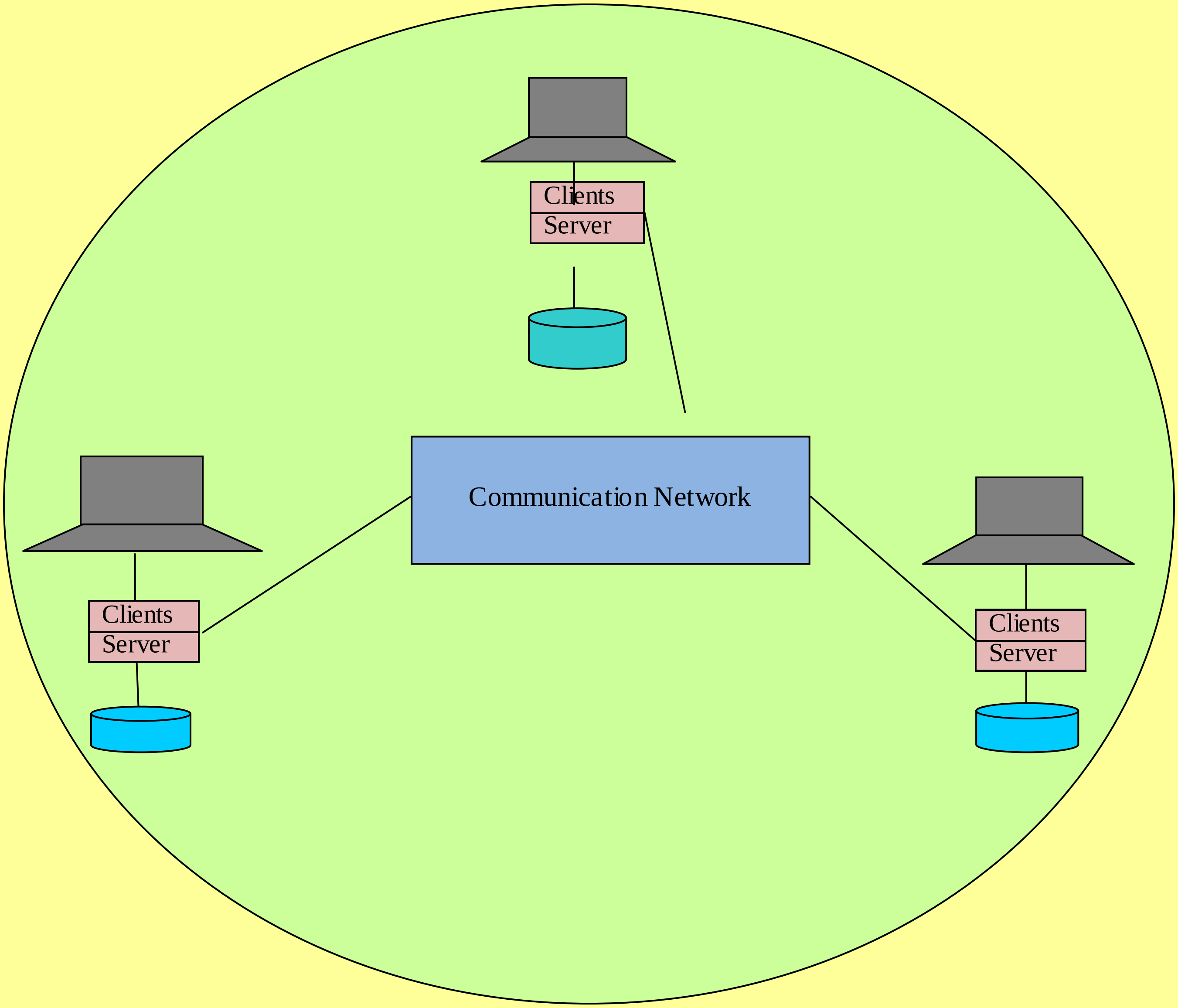
The Dummies' Guide to Database Systems: An Assembly of Information

An Easy to Understand Guide Even for Laypeople



Rosina S Khan

Dedicated to:

You, the Valued Reader

http://www.rosinaskhan.weebly.com

Contents

Preface vi

CHAPTER 1 1

INTRODUCTION 1
1.1 View of Data 3
1.2 Instances and Schemas 6
1.3 Database Languages 6
1.4 Database Users 7
1.5 Database Administrator 8

CHAPTER 2 9

ENTITY RELATIONSHIP MODEL 9
2.1 Entity, Entity set and Attributes 9
2.2 Types of attributes 9
2.3 Relationship Set 11
2.4 Mapping Cardinalities 11
2.5 Super key, Candidate key and Primary key 12
2.6 Entity-Relationship Diagram (ERD) 13
2.7 ER-diagrams with different cardinality ratios 14
2.8 Definition of Participation 19
2.9 ER-diagram Problems 20

CHAPTER 3 22

RELATIONAL MODEL 22
3.1 Converting ER diagrams to relational models 22
3.2 Relational model Problems 29
3.3 Query Languages 30
3.4 Relational Algebra 31
3.5 Outer Join 37
3.6 The Division Operation 39
3.7 Modification of the Database 40

CHAPTER 4 42

THE QUERY LANGUAGE SQL 42
4.1 Introduction to SQL 42
4.2 SQL Expressions using functions 44
4.3 SQL Expressions using Grouping 45
4.4 SQL Expressions using Sorting 46
4.5 Searching for partial strings 46
4.6 SQL Expressions concerning Input of Data 47
4.7 SQL Expressions concerning deletion of tuples 47
4.8 Update of tuples/values 47
4.9 Deletion of tables 47
4.10 Schema updates 48

4.11 Definition of a Domain 48
4.12 Creation and Deletion of Views 49
4.13 The Rename Operation 49
4.14 UNION ALL 50
4.15 INTERSECT & INTERSECT ALL 50
4.16 Inner and Outer Joins 50
CHAPTER 5 54
INTEGRITY CONSTRAINTS IN RELATIONAL SYSTEMS 54
5.1 Required Data 54
5.2 Domain Constraints 54
5.3 Entity Integrity 55
5.4 Referential Integrity 56
5.5 Modification 59
CHAPTER 6 60
FUNDAMENTAL DEPENDENCIES AND NORMALIZATION 60
6.1 Definition of functional dependency 60
6.2 Definition of full functional dependencies 60
6.3 Normalization 61
6.4 Notes on Normalization 67
6.5 Quality Criteria for Relational Design 68
CHAPTER7 69
QUERY PROCESSING 69
7.1 Introduction to Query Processing 69
CHAPTER8 72
FILE ORGANIZATION 72
8.1 Fixed-Length Records 72
8.2 Variable-Length records 73
8.3 Byte-String Representation 74
8.4 Slotted-page structure 74
8.5 Types of Record Organizations 76
8.6 Sequential File Organization 76
8.7 Multitable Clustering File Organization 78
CHAPTER9 80
DATA-DICTIONARY STORAGE 80
CHAPTER10 81
INDEXING 81
10.1 Basic Concepts 81
10.2 Ordered Indices 82
10.3 B^+-Tree Index Files 88
CHAPTER11 91

TRANSACTIONS 91
11.1 Syntax for Transactions 91
CHAPTER 12 93
RECOVERY IN TRANSACTIONS 93
12.1 Problem Sources 93
12.2 Logging and Recovery from Software Failures 93
CHAPTER 13 96
CONCURRENCY CONTROL 96
13.1 Problems Caused by Violation of Isolation 96
13.2 Serializability 97
13.3 Some Synchronization Protocols 99
CHAPTER 14 102
ADVANCED DATABASES 102
14.1 Distributed Databases 102
14.2 Data Warehouses 106
14.3 Multimedia Databases 107
14.4 Data Mining 108
14.5 What is a NoSQL (Not Only SQL) Database? 108
MISCELLANEOUS DATABASE PROJECT 112

Preface

Nowadays there is a growing need to organize, store and retrieve information of any organization real fast and convenient. Lots of paper files need manual work for updates and throwing away paper clutter, retaining the importanrt ones. All of this can be done smoothly and speedily by making the information computerized via databases. That is what this book is about and it covers a fundamental approach to this topic which can be taught and be useful for undergraduate courses in databases or any newbie who wants to grab the concepts behind databases.

While databases are themselves actually a collection/assembly of information, the content of this book is really an assembly of information collected from the following resources. **So the title of the book is intentionally ambiguous.** I must admit many of the chapters in this book are written in such an easy-to-understand matter that even laypeople can grasp the concepts.

[1] Database System Concepts, 5th Edition, Abraham Silberschatz, S. Sudarshan, Mc Graw Hill International Editions, 2006

[2] Professor Dorothee Koch's Lecture notes, Stuttgart University of Applied Sciences, Germany, 2005

[3] An Introduction to Database Systems, C. J Dates, Addison Wesley, 8th Edition, 2003

[4] http://planetcassandra.org/what-is-nosql, 2014

Organization

Let me now explain the organization of this book.

Chapter 1 is an introductory chapter starting with the idea of introducing backend and front end of database systems and covers stuff from [1] about why databases are at all useful.

Chapter 2 contains basics of Entity Relationship model (ERM), the fundamental step in designing databases at the backend and the resources are mainly from [1] and [2].

Chapter 3 covers how to convert ERMs to relational models (table schemas) [2], and introduces relational algebra, a pure and procedural form of query language [1].

Chapter 4 introduces Structured Query Language (SQL), the most widely used language used for querying relational databases and retrieving info. [mainly [2] and partly [1])

Chapter 5 covers examples of integrity constraints (conditions) that can be imposed in relational systems while creating tables in SQL [2].

Chapter 6 consists of functional dependencies of one attribute (field) on another attribute(s) in a table and based on these whether we have to split the tables or not according to violation or not of normal forms based on the concept of normalization. [2]

Chapter 7 mainly covers query processing which is the series of activities involved in extracting data from a database. [1]

Chapter 8 explains why the need to map databases to files may arise and how. [1]

Chapter 9 is a short explanation of a data dictionary and what info about databases it contains. [1]

Chapter 10 includes sophisticated indexing techniques for files. Just as words or phrases in a text book index appear in a sorted order, an index for a file in a database works in a similar way. [1]

Chapter 11 introduces the concept of transactions which are a sequence of data access operations that transfers the database from one consistent state to another consistent state. [2]

Chapter 12 explains recovery in transactions and how to preserve correctness and consistency of data over time, allowing for parallel access (transactions) of multiple users after transaction failures. [2]

Chapter 13 covers concurrency control which rectifies the problems occurring if two or more transactions using the same data items are executed in parallel. [2]

Chapter 14 includes advanced databases covering introductions to distributed databases and their possible architectures, data warehouses, multimedia databases and data mining in brief, as well as introduction to NoSQL environment. [1,2,3,4]

There is also a miscellaneous database project at the very end which students can work on through out the whole semester in parallel with theory lectures. The project is equally useful for even those who are not students and can work on it in their own interests.

Acknowledments

I am deeply indebted to the authors and publisher of Database System Concepts, Professor Dorothee Koch, C.J Dates and publisher, and Planet Cassandra website for collecting their resources, paraphrasing and combining them with mine and hence the creation of this very book.

Last but not the least I am thankful to my mom and Dr Manzur for all their help and support while writing out this book.

--Rosina S Khan

CHAPTER 1

INTRODUCTION

A database system is defined to be a collection of interrelated data and a set of programs to access those data.

Why *interrelated* data? Because some sort of relationship exists among the data. (It will become clearer when we come to the topic *Normalization of tables.)* This actually occurs in the backend. A backend is nothing but a collection of interrelated data in different tables. A table has some fields in a row. These fields are also called attributes. Corresponding to the attributes are some data in rows. These are called records or tuples. The entire table corresponds to an entity. We will cover more on entities and attributes in the next chapter.

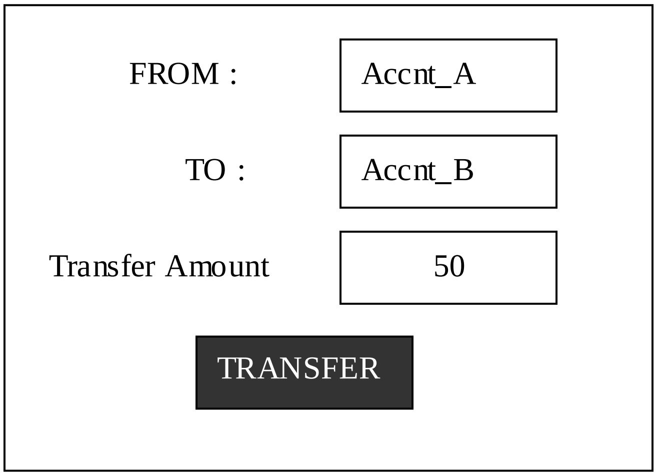
Why *set of programs*? Programs here refer to application programs at the front end or *interfaces* connected to data at the backend. A typical layout for backend and front end is given as an example below:

Table 1.1: Jone's Account Info (*Backend)*

Accnt _A	Accnt_B	} Fields/attributes
150	200	} Tuple/record

Table 1.2: Jone's Updated Account Info after transaction

Accnt _A	Accnt_B
100	250



FROM : Accnt_A

TO : Accnt_B

Transfer Amount 50

Fig 1.1: Jone's Account Interface (*Front end)*

A backend database for example, may consist of Jone’s Account Info table. After a transaction, Jone’s Account Info may be updated as shown in Table 1.2. An interface to the backend database may be depicted as shown in Fig 1.1. Given the values of source and destination accounts as well as the transfer amount in textboxes, hitting the transfer button will enable to make the transaction, which will in turn be updated in Jone’s Account Info table in the backend.

The backend may be developed using software tools such as MS SQL Server, MySQL etc. while the front end may be developed using C#, or PHP, JavaScript and HTML, etc.

Before the advent of databases, organizations stored information using a typical file-processing system. In this system, permanent records were stored in various files and different application programs were written to extract records and add records to the appropriate files.

File-processing systems have a number of disadvantages. These are outlined as follows:

- Data redundancy and inconsistency: Different programmers may write the files and application programs over a passage of time. As a result, files may have different structures and the programs may be written in different several programming languages. Also, the same information may appear in different files. Data inconsistency results when the same information is updated in one place but not in another place in addition to higher storage and access cost.

- Difficulty in accessing data: Data retrieval may be problematic. Suppose a bank officer needs to find out the list of customers who live in a certain postal code. Such an application program does not exist. There is, however, an application program to generate the list of customers. The data processing department based on the demand of the bank officer has either to generate the list of customers and extract the needed info manually. The other alternative is to write a new application program to meet the demand. Both the alternatives are unsatisfactory. After a week or so, the bank officer needs to trim down the list of customers with bank balances more than Tk50,000. Again the data processing department is left with two alternatives both of which are unsatisfactory. The point here is that file processing systems do not allow data to be retrieved in a convenient and efficient manner.

- Data Isolation: As data lie in different files and files may be in different formats, it is difficult to retrieve the appropriate data by writing new application programs each time.

- Integrity Problems: The data values stored in the database must satisfy certain conditions called integrity or consistency constraints. For example, the bank balance of a customer must never fall below Tk200. Developers impose these constraints by writing appropriate code in the various application programs. To enforce a new constraint such as the bank balance of customers should not exceed

10 crore taka, it becomes difficult for the developer to enforce the new constraint and change the programs. The problem worsens when constraints involve several data items from different files.

- Atomicity Problems: Atomic transactions mean they either occur completely or none at all. For example, consider a program to transfer Tk500 from Account A to Account B. If a system failure occurs during the transaction, it is possible Tk500 was debit from Account A but not credited to Account B, resulting in an inconsistent database state. It is important that to maintain database consistency, either both the debit and credit occur, or that neither occurs. It is difficult to ensure atomicity in conventional file processing systems.

- Concurrent-access anomalies: Multiple users may update a system simultaneously. This facilitates faster response and overall performance of the system. Interaction of concurrent updates may result in inconsistent data. Consider bank account A, containing Tk500. If two customers withdraw funds say, Tk50 and Tk100, respectively from Account A at about the same time, the result of concurrent transactions may leave the account in an inconsistent state. If the two transactions occur concurrently, they may both read the value Tk500, and write back Tk450 and Tk400, respectively. Depending on which one writes the value last, the account may contain either Tk450 or Tk400, rather than the correct value of Tk350. To prevent this from happening, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by various different application programs that have not been coordinated previously.

- Security problems: Every user of the database system should not be able to access all the data. For example, in a banking system, payroll personnel (tax officer) needs to see only that part of the database that has information about the various bank employees. They do not need to access information about customer accounts. As another example, bank tellers see only that part of the database that has information on customer accounts. They cannot access information about salaries of bank employees. Enforcing such security constraints on a file-processing system is difficult because application programs are added to the system in an ad hoc manner.

1.1 View of Data

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

1.1.1 Data Abstraction

Since many database-system users are not handy with computers, developers hide certain complexity details through several levels of abstraction in order to make it easier for users' interaction with the system.

- Physical level: The lowest level of abstraction describes how data are actually stored. It describes complex low-level data structures in detail.

- Logical level: The next-higher level of abstraction describes what data are stored in the database, and what relationships exist among those data. It describes the entire database in terms of a small number of relatively simple structures. Database administrators use the logical level of abstraction who must decide what information to store in the database.

- View Level: The highest level of abstraction describes only part of the entire database. In this level users need to access only part of the entire database. The level simplifies the interaction of users with the system. The system may provide many views for the entire database.

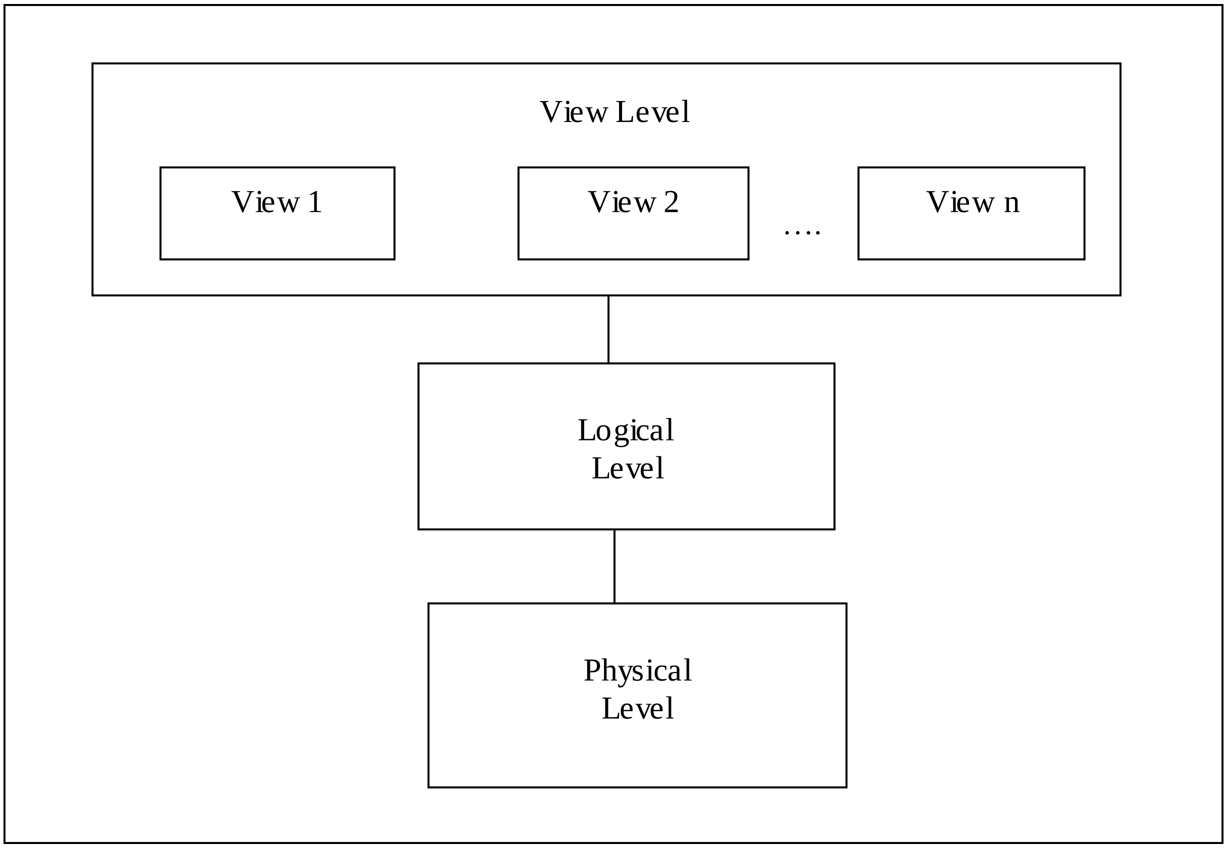


Fig 1.2: The three levels of data abstraction

Distinction among levels of abstraction may be compared to the concept of data types in programming languages. Most high-level programming languages support the notion of a structured type. For example, in a Pascal-level language, we may declare a record as follows:

```
type  customer = record
                    customer_id: string;
                    customer_name: string;
                    customer_street: string;
                    customer_city: string;
                 end;
```

Code 1.1: Pascal-like definition of record structure

Code 1.1 defines a record type called customer with four fields. Each field has a name and a type associated with it. A banking enterprise may have several such record types, including

- account, with fields account_number and balance
- employee, with fields employee_name and salary

At the physical level, a customer, account, or employee record can be described as block of consecutive storage locations or bytes. The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest-level storage details from database programmers. Database administrator, however, may be aware of certain details of the physical organization of the data.

At the logical level, each such record is described by a type definition as shown in Code 1.1, and the level also defines the interrelationships among the different record types. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction too.

Finally at the view level, computer users see a set of application programs or front end interfaces that hide details of the data types. At this level several views are defined and database users see and access these views. The views also provide a kind of security mechanism to allow users only to access certain parts of the database. This has been explained in detail in the subsection **Security problems** on page 3. Some examples of views can be:

- View1: customer_name|account_number|balance
- View2: employee_name|account_number|balance|salary

1.2 Instances and Schemas

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

Consider the following database table:

Employee Table:

employee_name	salary
Jones	10000
Jim	20000
Jack	15000
David	25000
Henry	30000

} Schema

Instance

In the above Employee Table, schema and instance are clearly shown.

Database systems have several schemas, partitioned according to the levels of abstraction. The physical schema describes the database design at the physical level, while the logical schema describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called subschemas, that describe different views of the database.

The logical schema is the most important among all the schemas since programmers construct application programs or front end interfaces by using logical schemas. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting logical level. Hence, application programs do not need to be rewritten if physical schema changes and are said to exhibit physical data independence.

1.3 Database Languages

A database system provides a data-definition language to specify the database schema and a data-manipulation language to express database queries and updates. In practice, the data-definition and data-manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL (Structured Query Language).

a) Data-Manipulation Language

A data-manipulation language (DML) is a language that enables users to access or manipulate data as organized by the appropriate data model. The types of access are:

- Retrieval of information stored in the database
- Insertion of new information into the database
- Deletion of information from the database
- Modification of information stored in the database

b) Data-Definition Language

We specify a database schema by a set of definitions expressed by a special language called a data-definition language (DDL). The DDL is also used to specify additional properties of the data. We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

1.4 Database Users

- **Application programmers** are computer professionals who write application programs or front end interfaces. They can choose tools such as **Rapid application development (RAD)** to develop front end interfaces such as forms and reports with minimal programming effort.

- **Sophisticated users** interact with the system without writing programs. They form their requests to the system using a database query language e.g, SQL.

- **Specialized users** are sophisticated users who write specialized database applications such as computer-aided design (CAD), knowledge-base and expert systems that store complex data such as graphics and audio data and environment-modeling systems.

- **Naive users** are unsophisticated users who interact with the system by invoking one of the application forms that have been written previously. An example of an application program can be that of Jone's Account Interface in Fig. 1.1 where the naïve users fill in the fields of the form and hit the button. They may also simply read reports generated from the database

1.5 Database Administrator

A person who has central control over the whole database system is a database administrator (DBA). His functions may be summarized as below

- **Schema definition**: The DBA creates the original database schema or the overall design of the database.

- **Storage structure and access-method definition:** The DBA is aware of certain details of physical organization of the data i.e. how records occupy storage locations or bytes and the methods to access those data.

- **Schema and physical-organization modification:** The DBA can change the schema and physical organization according to the demands of the organization, or simply to improve performance. This can be done without affecting application programs at logical level.

- **Granting of authorization for data access:** The DBA can grant different types of authorization to different users and hence control which user has the right to access which parts of the database. This enhances to keep the system secure as has been explained earlier.

- **Routine maintenance:** Some examples of DBA's routine maintenance activities are as follows:

  i) Backup the database periodically to tapes or remote servers so that they can be recovered in case of disasters such as database sabotage.
  ii) Ensure enough disk space is available for normal operations and upgrade disk space as required.
  iii) Monitor jobs running in the database and ensure overall performance is good.

CHAPTER 2

ENTITY RELATIONSHIP MODEL

2.1 Entity, Entity set and Attributes

An entity is a thing or object in the real world that is distinguishable from other objects. An entity can be concrete, such as a book or a person or it may be an abstract, such as a loan, holiday or a concept. An entity in a database system actually represents a table.

The properties or parts of an entity are called attributes. For example, a person has the attributes person_id, name, occupation, salary etc. A book has the attributes book_id, author, publisher, category, number of copies etc. Attributes are actually the fields of a database table or entity.

An entity set is a set of entities of the same type that share the same properties or attributes. For example, the set of all students who take a class can represent a student entity set in which each entity is a student sharing similar attributes with other students such as student_id, name, contact_no, address etc.

2.2 Types of attributes

An attribute can be of the following types:

- **Simple and Composite** attributes: Simple attributes are those as the name implies that is they are not divided into subparts. For example, the attribute *student_id* for the entity student has no subparts and therefore, is a simple attribute. On the other hand, an attribute *name* can be structured as a composite attribute consisting of first_name, middle_name and last name. As another example, the attribute *customer_street* for the entity customer can be composite consisting of street_number, street_name and apartment_number. (Fig 2.1)

- **Single-valued and mutivalued** attributes: The loan_number attribute for a specific loan refers to one loan_number. Such attributes are said to be single valued. On the other hand there may be instances where an attribute has a set of values for a specific entity. For example, consider the employee entity set with the attribute phone number. An employee may have zero, one or several phone_numbers and different employees may have different number of phone numbers. This type of attribute is said to be multivalued. As another example, the attribute dependent name of the employee entity set would be multivalued, since any particular employee may have zero, one or more dependent(s).

  Upper and lower bounds may be placed on the values in a multivalued attribute as needed. For instance, a bank may limit storing the number of phone_numbers for a customer to two. Placing bounds in this way means that the phone_number attribute for the customer entity set may have the range 0<=phone_number<=2.

- **Null** attributes: An attribute takes a null value when that value is missing, not applicable or unknown. For example, a person may have no middle name (not applicable). If the name for a particular customer is null we assume that the value is missing since every customer must have a name. A null value for an apartment_number could mean the address does not include an apartment number (not applicable), that an apartment number exists but we do not know what it is (missing), or that we don't know whether an apartment number is part of the customer's address (unknown).

- **Derived** attributes: The value for this type of attribute can be derived from the values of other related attributes or entities. For example, a customer entity has the attribute customer_age. If the customer attribute also has an attribute date_of_birth, we can calculate his age from data_of_birth and the current date. Therfore, customer_age is a derived attribute. As another example, the entity employee can have employment_length as an attribute. If this entity has another attribute start_date of his employment, then we can calculate the employee's employment_length from the start_date and current_date. Hence, in this case employment_length is a derived attribute.

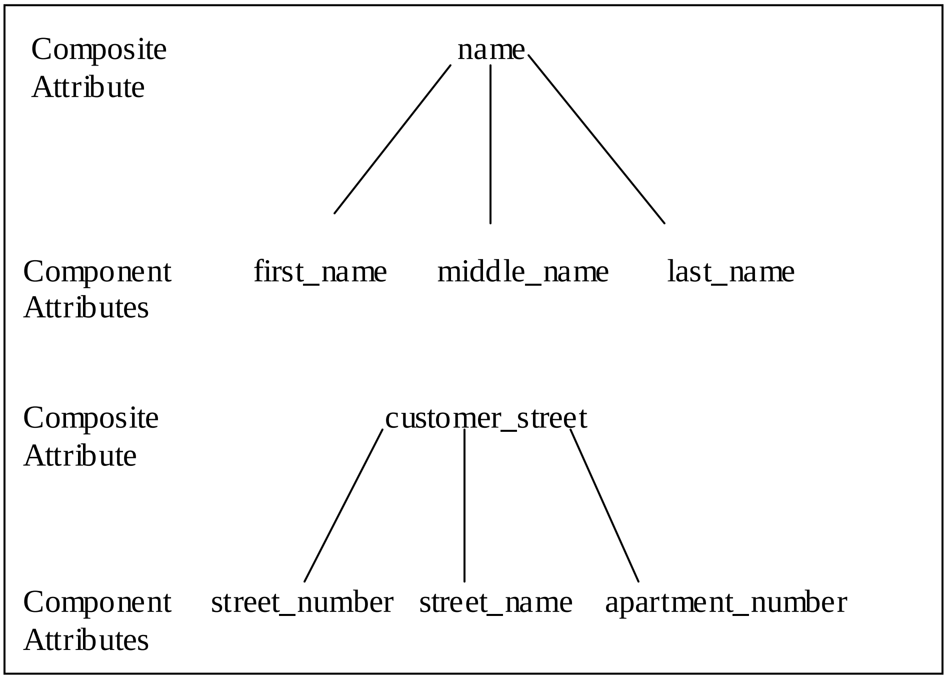


Fig. 2.1: Composite Attributes *name* and *customer_street*

2.3 Relationship Set

A relationship is an association among several entities. For example, we can define a relationship that associates customer James with loan L-20. This relationship specifies that James is a customer with loan number L-20.

A relationship set is a set of relationships of the same type. Consider two entity sets customer and loan. We define the relationship set *borrows* to denote the association between customers and bank loans that the customers have.

2.4 Mapping Cardinalities

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

Mapping cardinalities are useful in describing binary relationships, although they can contribute to the description of relationship sets that involve more than two entity sets. In this section we shall concentrate on only binary relationship sets.

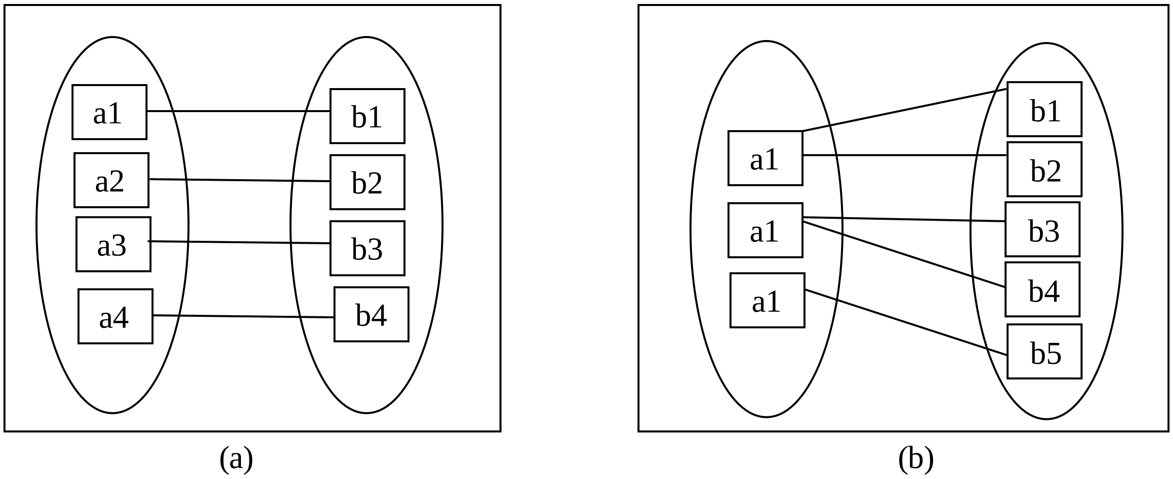
For a binary relationship set R between entities A and B, mapping cardinalities must be one of the following:

- 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.

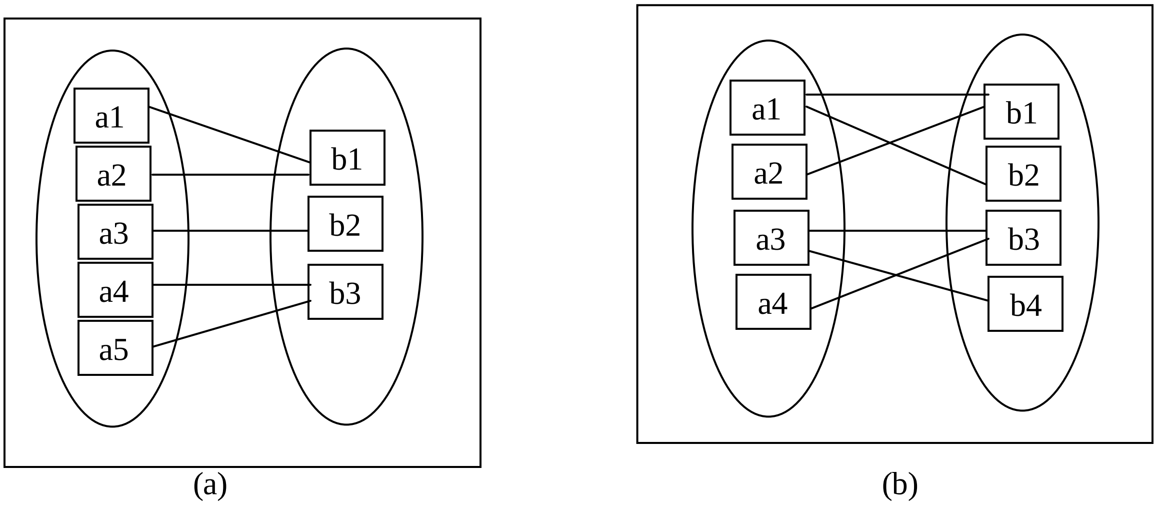
- Many-to-one: An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A.

- 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.



(a) (b)

Fig 2.2: Mapping Cardinalities (a) One-to-one b) One-to-many



(a) (b)

Fig 2.3: Mapping Cardinalities (a) Many-to-one (b) Many-to-many

2.5 Super key, Candidate key and Primary key

A superkey is a set of one or more attributes that taken collectively allow us to identify uniquely an entity in the entity set. For example, the customer_id attribute of the entity set customer is sufficient to distinguish one customer entity from another. Thus, customer_id is a super key. Similarly, the combination of customer_id and customer_name is a superkey for the entity set customer. Thus, a minimal super key with extraneous attributes is also a superkey. However only the customer_name attribute of a customer is not a superkey, because several people might have the same name.

Thus, we now know that if K is a superkey, then so is any superset of K. But we are often interested in minimal superkeys with no extraneous attributes. Such minimal superkeys are called candidate keys.

Several distinct sets of attributes could serve as a candidate key so long they identify uniquely an entity in the entity set. Consider the combination of customer_name and customer_street. This set of attributes can distinguish uniquely among members of the customer entity set. Therefore, both {customer_id} and {customer_name, customer_street} are candidate keys. The latter is, however, not a super key. Additionally, although the set {customer_id, customer_name} can distinguish customer entities, their combination does not form a candidate key, since the attribute customer_id alone is a candidate key. This set is, on the other hand, a super key.

We shall use the term primary key to denote a candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set. Unlike super keys and candidate keys, there can always only be one way of representing a primary key for the entities of the entity set. It is noteworthy that every entity set must have a primary key to distinguish uniquely the entities within the entity set and to distinguish from other entity sets.

2.6 Entity-Relationship Diagram (ERD)

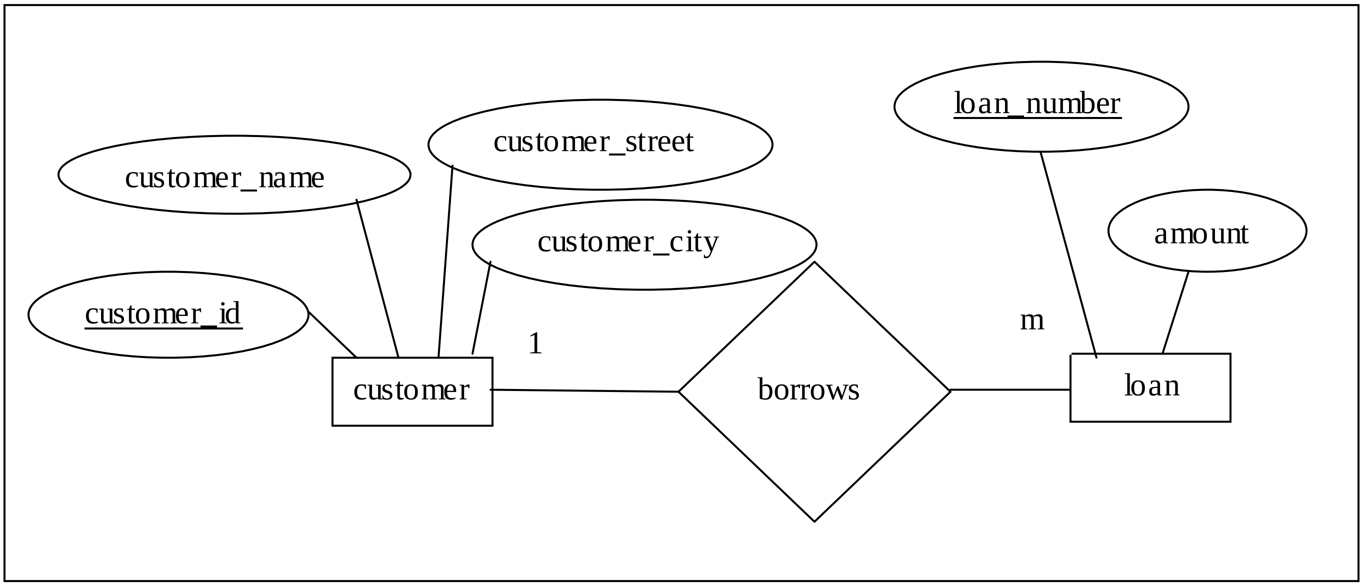


Fig 2.4: E-R diagram corresponding to customers and loans

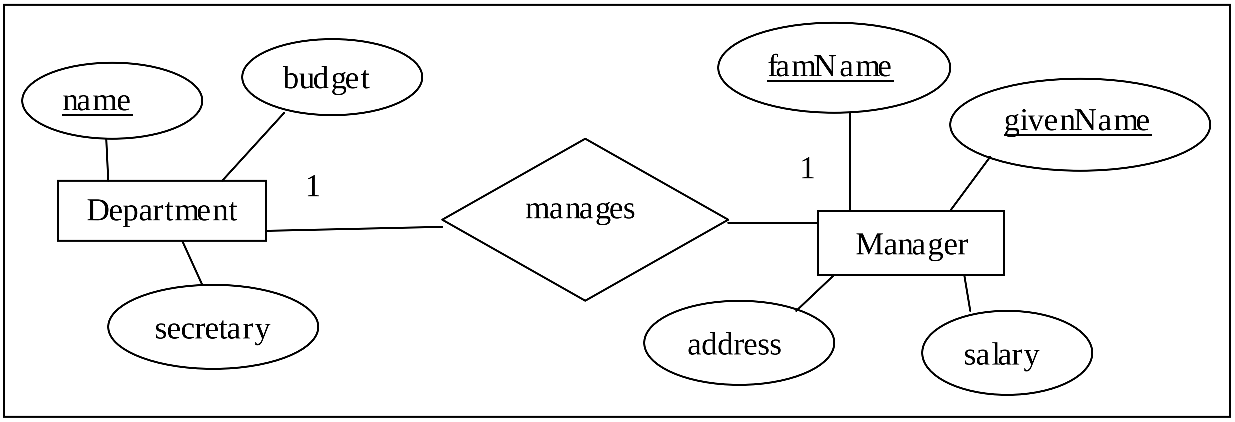
Fig 2.4 shows an example of an entity-relationship diagram. Such a diagram must have the following criteria:

- Entities represented by rectangles (e.g. customer and loan)

- Attributes of an entity represented by ellipses (e.g. loan_number and amount attributes for the entity loan)
- Relationship sets represented by diamonds (e.g. borrows relationship set.)
- Primary key attribute of an entity set underlined (e.g. primary key customer_id underlined for entity customer)
- Links joining diamond relationship set to entities. (e.g. there are links joining the entities to the borrows relationship to complete the E-R diagram.)
- Cardinality Ratio that is as explained in sec 2.4. (e.g. fig 2.4 depicts a binary relationship between the two entities customer and loan. The cardinality ratio from customer to loan is one-to-many because a customer can borrow many loans from the bank but one loan from the bank can belong to one customer assuming the loan is not joint i.e, an individual loan.)

2.7 ER-diagrams with different cardinality ratios

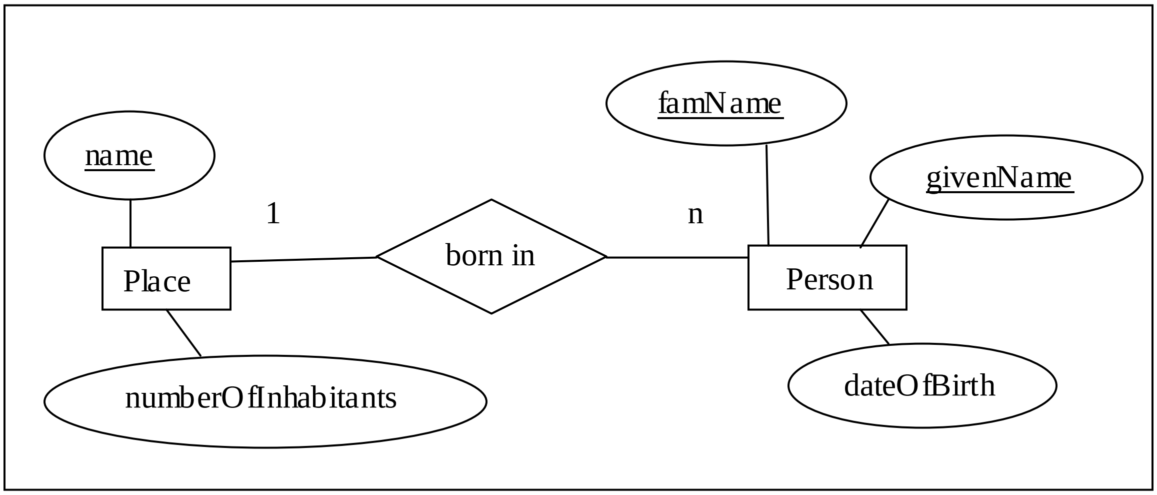
a)1-1 relationship (one-to-one relationship)



Assumption: 1 department has 1 manager.

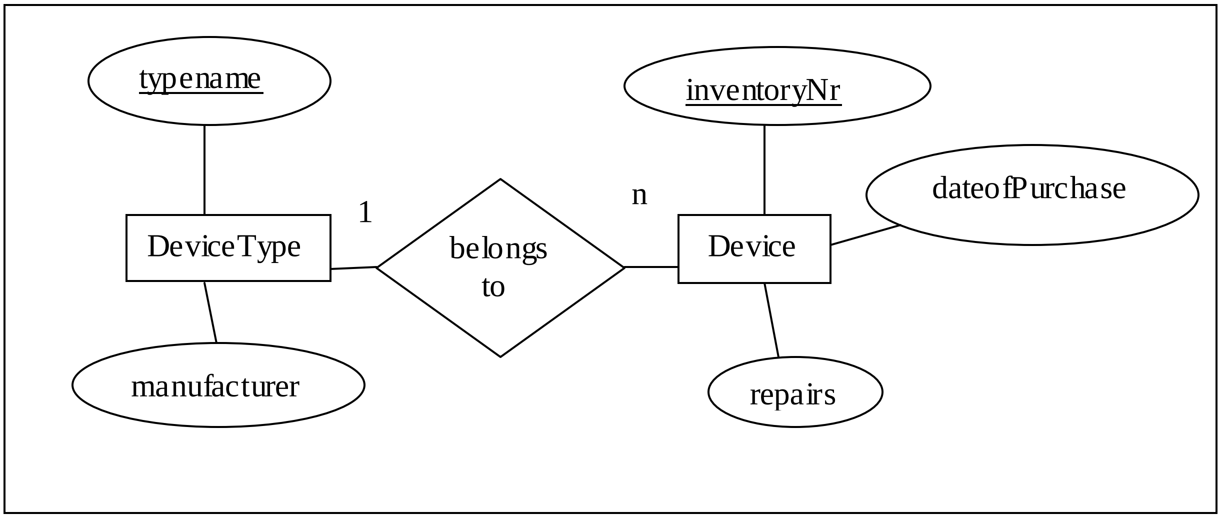
The given ER diagram has cardinality ratio 1:1 because 1 department has 1 manager and 1 manager manages 1 department.

b)1-n relationships (one-to-many relationships)
i)



The above ER diagram has cardinality ratio 1: n because in 1 place many persons are born and 1 person is born in 1 place.

ii)



In the above ER diagram, DeviceType may be categories of tools while a device belongs to one such category.

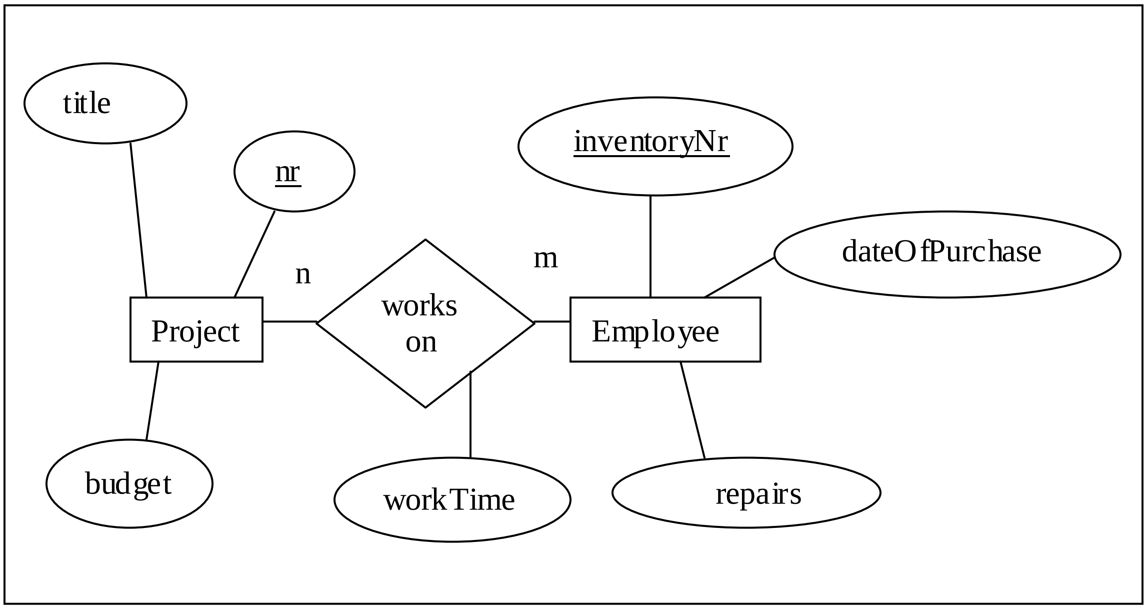
For example:

DeviceType	Device
Electric tools	drill machine, electric knife, iron
Manual tools	scissors, pliers, screwdriver

Considering the above ER diagram and the given example, we can say the ERD has cardinality ratio 1: n because one DeviceType contains many devices but one device can belong to one DeviceType only.

c) n-m relationships (many-to-many relationships)

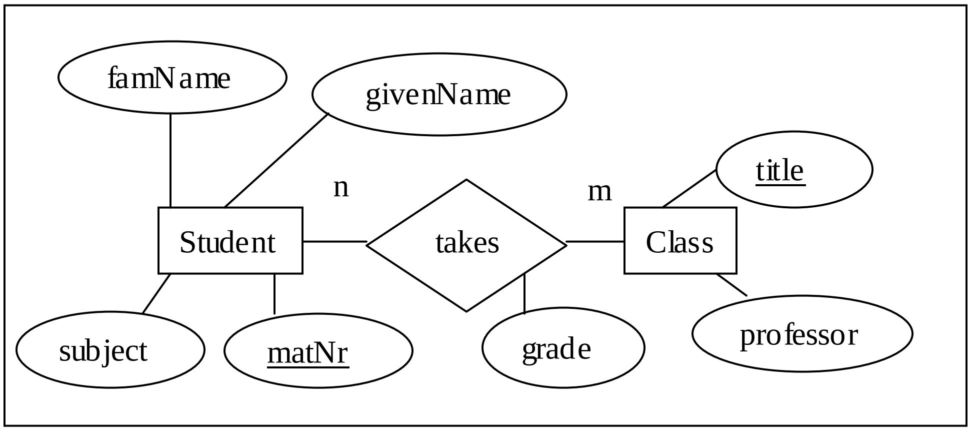
i)



The above ER diagram has cardinality ratio n : m because in 1 project many employees can work and 1 employee can work in many projects.

Note: In this diagram, we see that the relationship *works on* has an attribute workTime. In case of n-m relationships, the diamond relationship can have attributes if such attributes cannot be directly assigned to the entities. For example, if the entity employee is assigned workTime, we cannot find out how much time the employee works for which project. On the other hand, if we assign workTime to Project entity, we cannot find out which employee works for how much time for a particular project. It is further to be noted that in case of binary relationships 1:1 and 1: n, the middle diamond relationship cannot have any attributes.

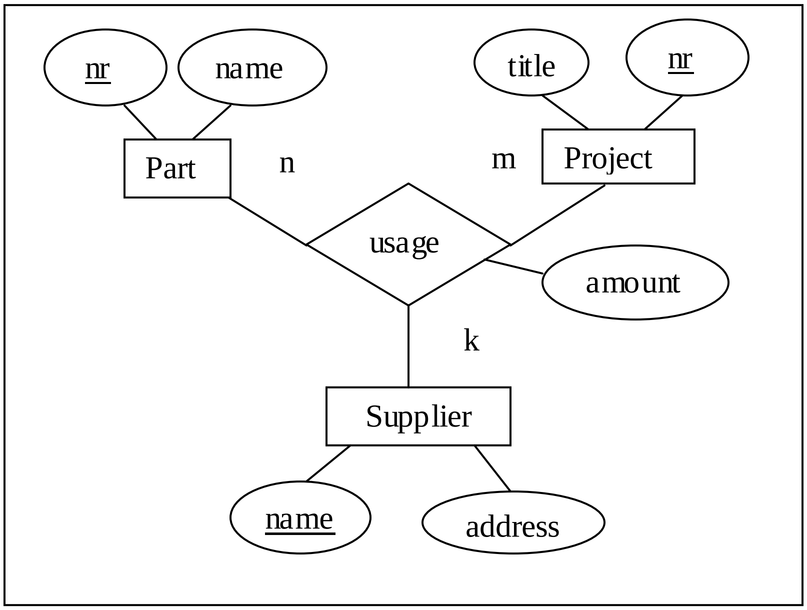
ii)



The above ER diagram has cardinality ratio n : m because one student can take many classes and in 1 class, there can be many students.

Note: For reasons similar to the previous example, the attribute grade is assigned to *takes* relationship. Grade cannot be assigned to class because if we do, we lose the information on which student gets a grade for a particular class and if we assign grade to student, we lose the information on which class a particular student gets a grade.

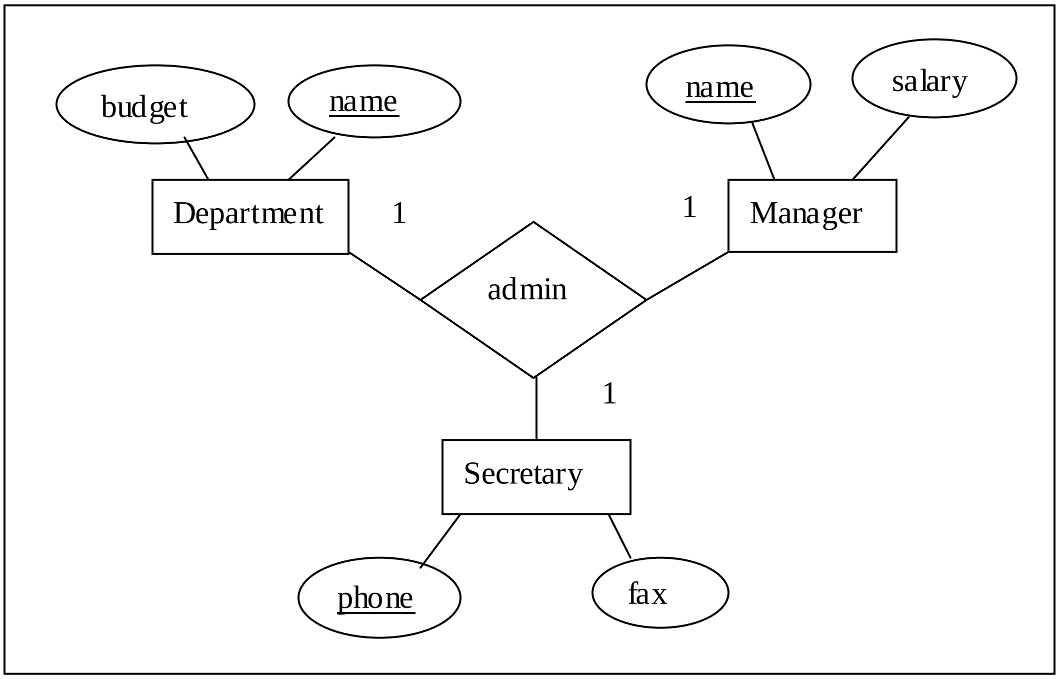
d) n-m-k ternary relationship (many-to-many-to-many ternary relationship)



In the above ER diagram, the cardinality ratio is n:m:k. One supplier can supply for one project many parts. The same part can be supplied by one supplier for many projects. The same part can be supplied for one project by many suppliers.

Note: In all ternary relationships, the middle diamond relationship can have attribute(s) when those attribute(s) cannot be directly assigned to the entities as has been explained in case of n-m binary relationships.

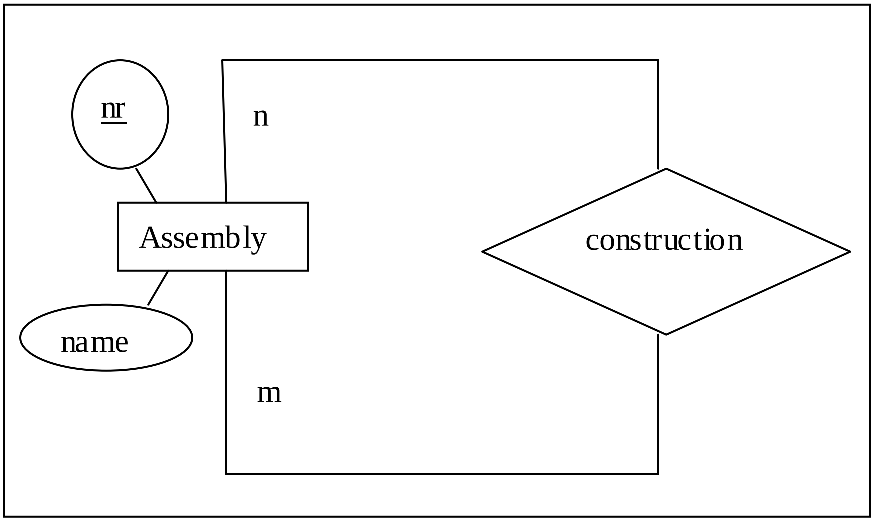
e) 1-1-1 ternary relationship (one-to-one-to-one ternary relationship)



The above ER diagram has cardinality ratio 1:1:1. Let us assume 1 department has 1 manager and 1 secretary. Now 1 department with 1 manager works with 1 secretary. 1 department with 1 secretary works with 1 manager. 1 manager together with 1 secretary works in 1 department.

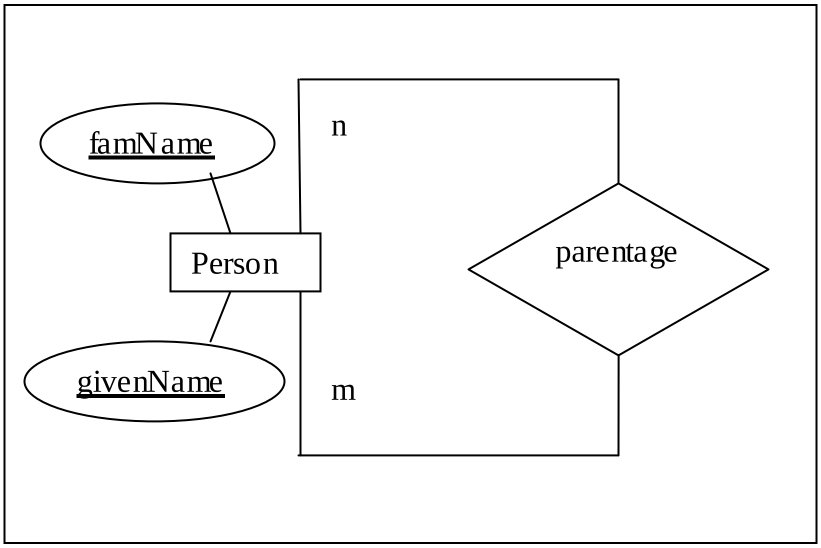
f) Recursive relationship: An entity can be related to itself.

i)



The above ER diagram comprises of products consisting of a hierarchy of assemblies. One assembly contains other assemblies. One assembly is contained in other assemblies.

ii) Another example of Recursive ER diagram

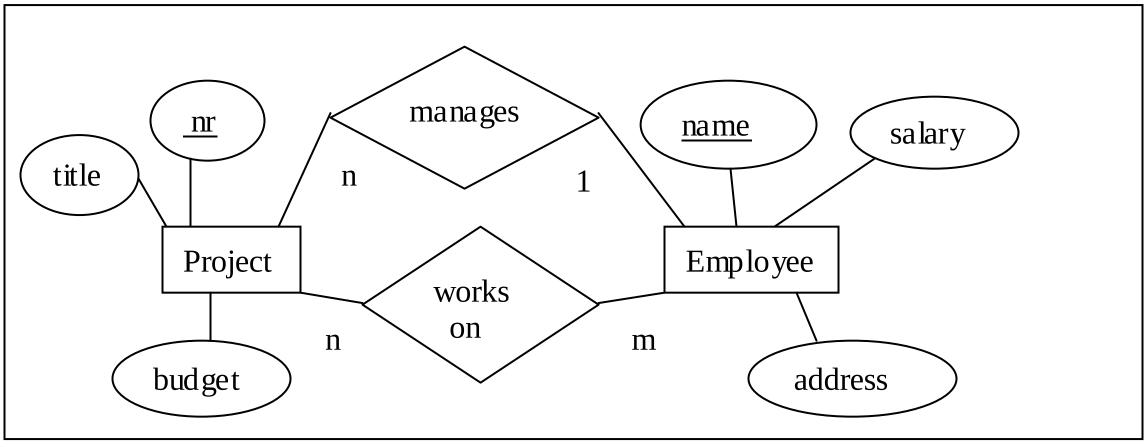


In the above recursive ER diagram, people are parents and children of other people.

2.8 Definition of Participation

If every instance of an entity is related to one or more instances of another relation via a relationship, this is called "total participation". If not, every instance of the entity is related via the relationship, this is called "partial participation"

Example:



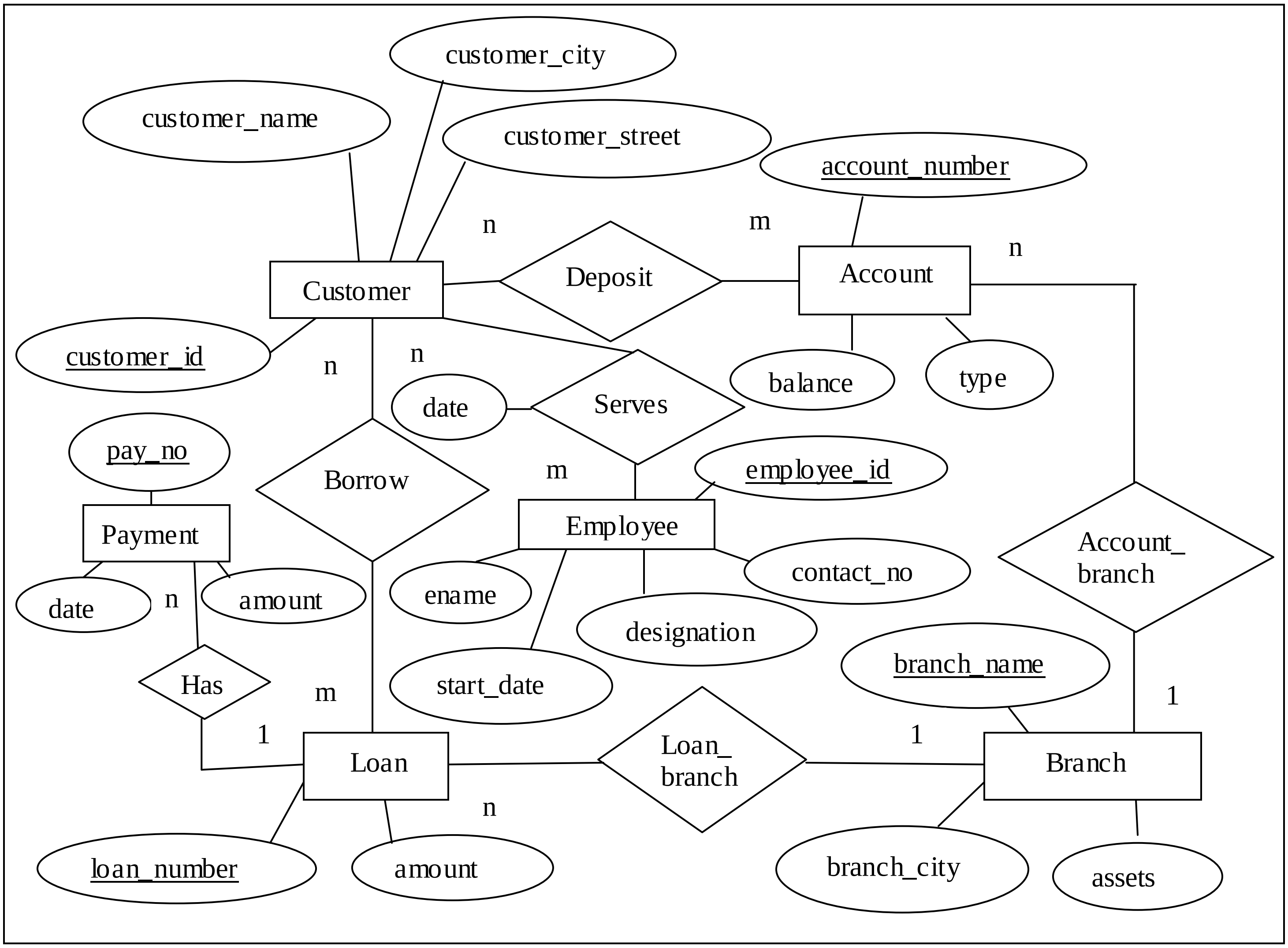
Not every employee manages a project, so Employee participates only partially in the relationship “manages”, while Project may participate totally in “manages” (if projects can only be defined if they have a manager).

2.9 ER-diagram Problems

i)Banking Enterprise Problem

Customers can deposit money in an account either checking or savings account. They can borrow loans from the bank and pay them back in installments. More than one account can be set up in a particular branch. Loans can be borrowed from the same branch. Employees in the bank including a manager serve the customers for their specific interests. Develop an ER diagram for the above application.

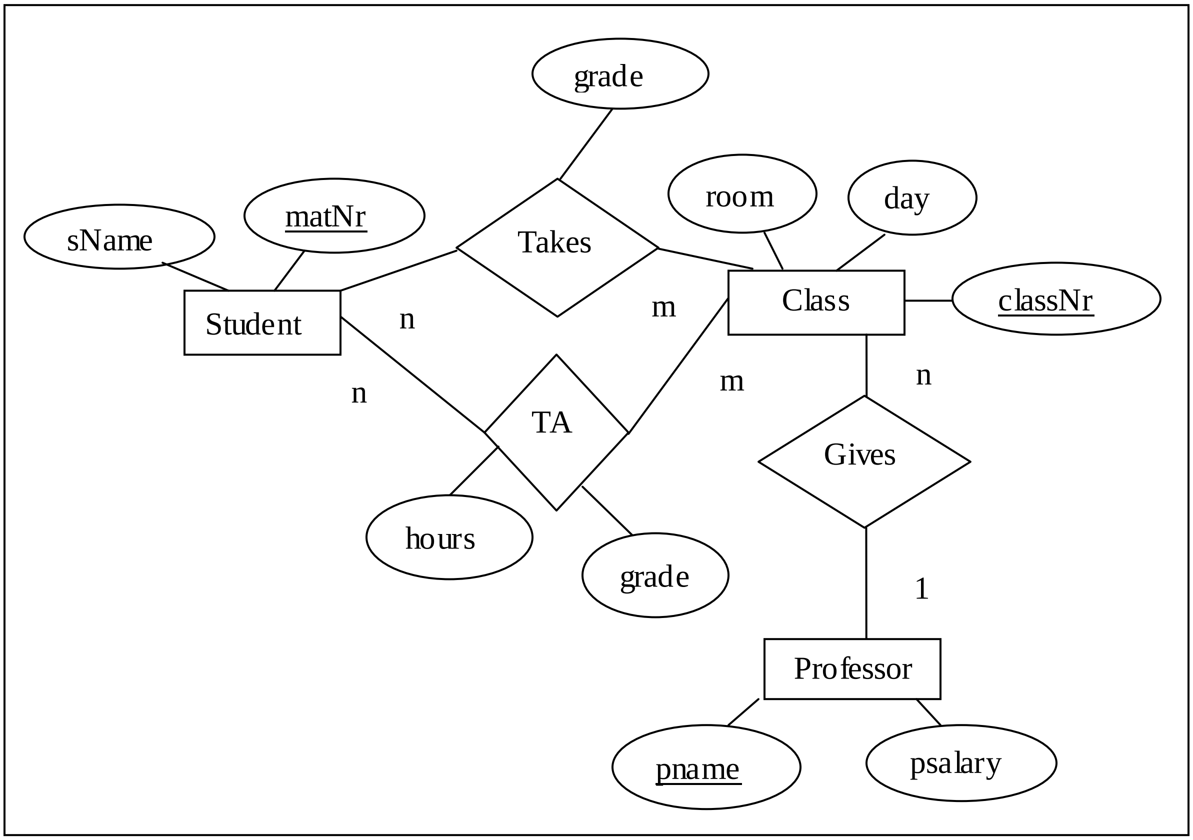
Solution:



ii) University Organization Problem

Consider the following university database application. Every student has a matriculation number (student id) and a name. A student takes a class in a room on a particular day. A teaching assistant who is also a student of the university can take class(es) for a number of hours not exceeding 80 hrs and is given a salary per month not exceeding Tk2000. In addition to TA's, there are professors who can give lectures for more than one class but one class lecture is given by one professor, who have a salary not exceeding Tk3 lacs. A student at the end of the semester gets a grade for each class number (course).
Develop an ER diagram for the above application.

Solution:



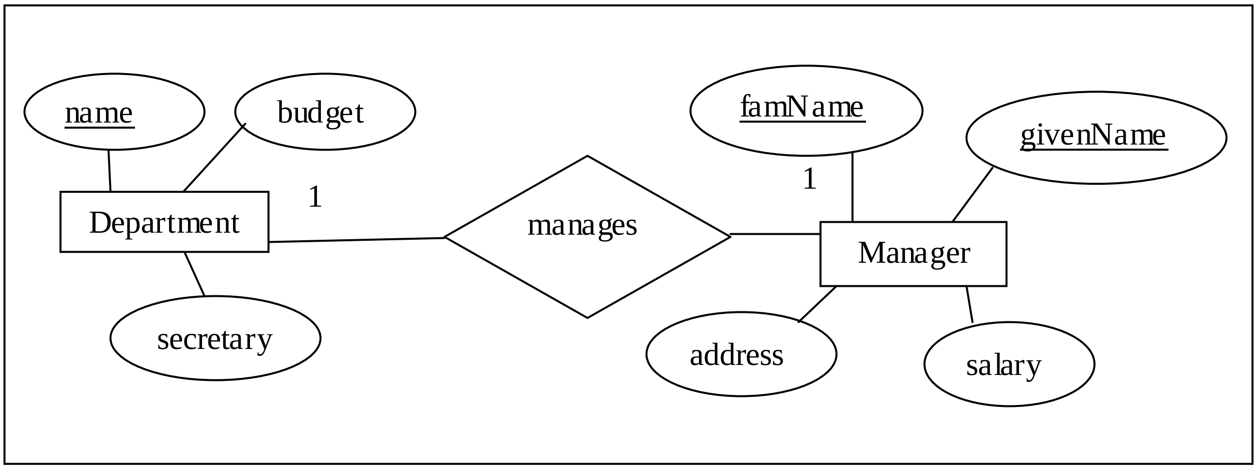
CHAPTER 3

RELATIONAL MODEL

3.1 Converting ER diagrams to relational models

3.1.1 One-one relationship

Model each entity as a separate relation. Add the primary key of one relation as a foreign key (a copy of the primary key from the first relation, which may or may not be a part of the primary key) to the other relation.



The above ER diagram can be modeled as a relational model or table schemas as follows:

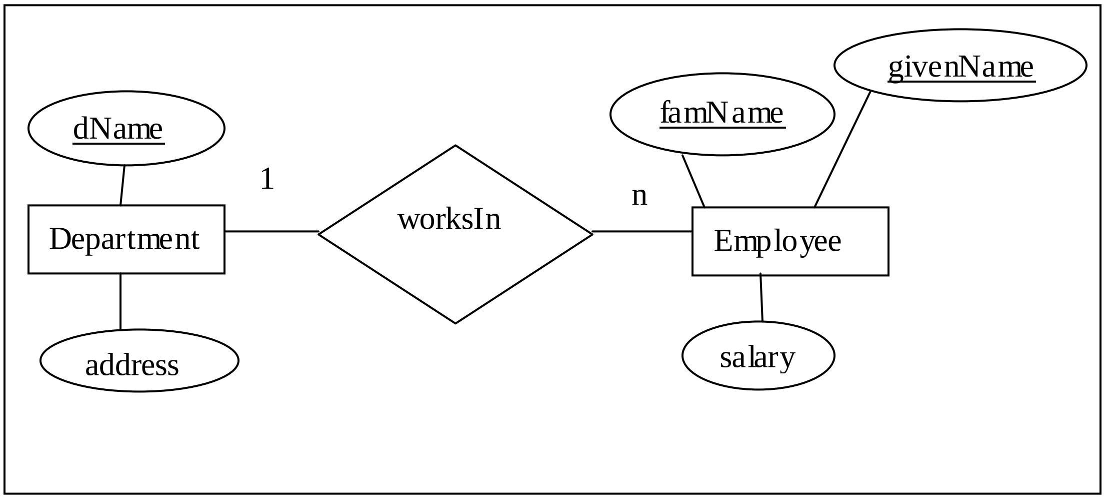
Department:

name | budget | secretary | famName | givenName
pk
(primary key)
fk (foreign key)

Employee:
famName | givenName | address | salary
pk

3.1.2 One-many relationship

Add the primary key of the relation with the “1” as foreign key to the relation with the “n”.



The above 1-n relationship in the ER diagram can be converted to a relational model as follows:

Employee:

$$\underbrace{\underline{\text{famName}} \mid \underline{\text{givenName}}}_{\text{pk}} \mid \text{salary} \mid \underset{\text{fk}}{\text{dName}}$$

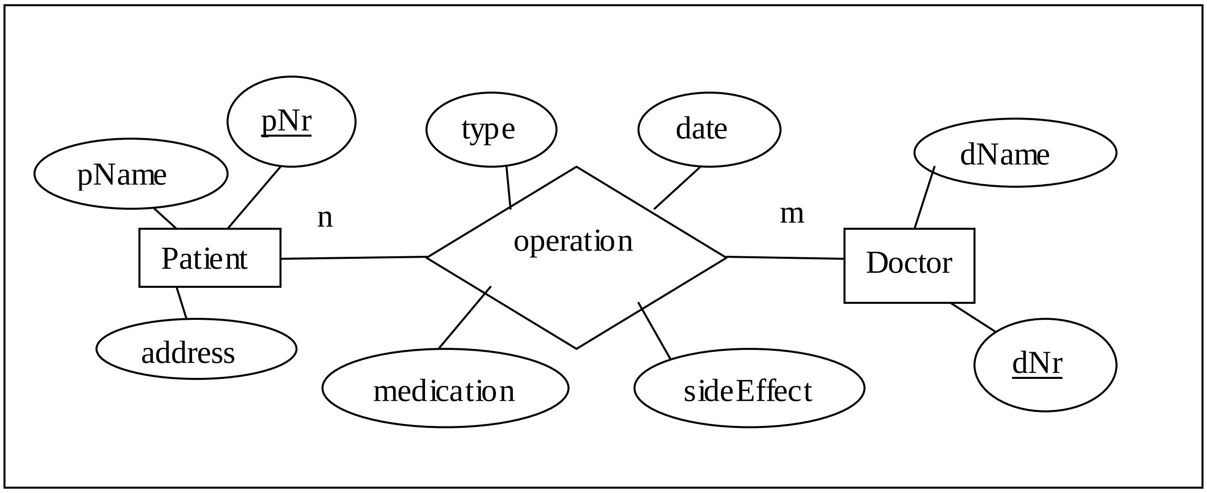
Department:

$$\underset{\text{pk}}{\underline{\text{dName}}} \mid \text{address}$$

3.1.3 Many-many relationship

Create an additional relation for the relationship.
Define in the additional relation the primary keys of the other relations as well as all attributes of the relationship.

Note: The primary key of the relation modeled from the relationship needs to be determined after careful inspection
Example:



The above ER diagram is transformed to the following relational model:

Patient:

<u>pNr </u> | pName | address

Doctor:

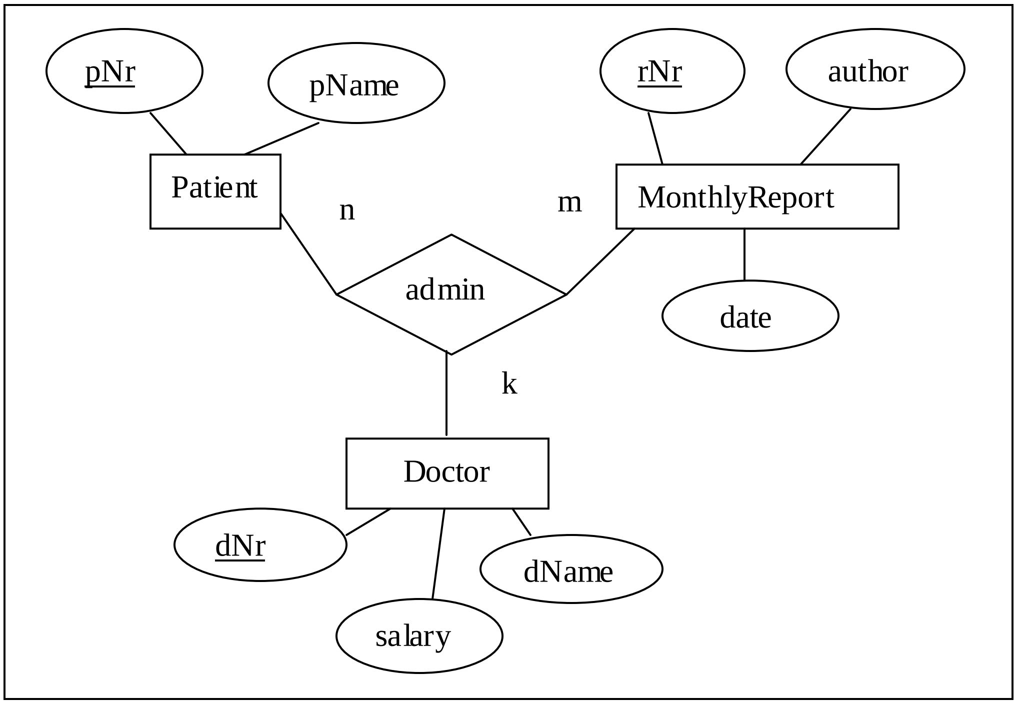
<u>dNr</u> | dName

operation:

<u>pNr</u> | <u>dNr</u> | <u>date</u> | type | medication | sideEffects

In *operation* relationship, date needs to be part of the primary key. The same patient can be operated by the same doctor on two different dates for example. In that case values for pNr and dNr become the same in two different tuples. We know that the value of primary key has to be different in every tuple. So we need to add date as a part of the primary key for the relation *operation* in order to make the key unique.

3.1.4 Many-many-many ternary relationship



The above ER diagram is transformed to the following relational model:

Patient:

pNr | pName

Doctor:

dNr | dName | salary

MonthlyReport:

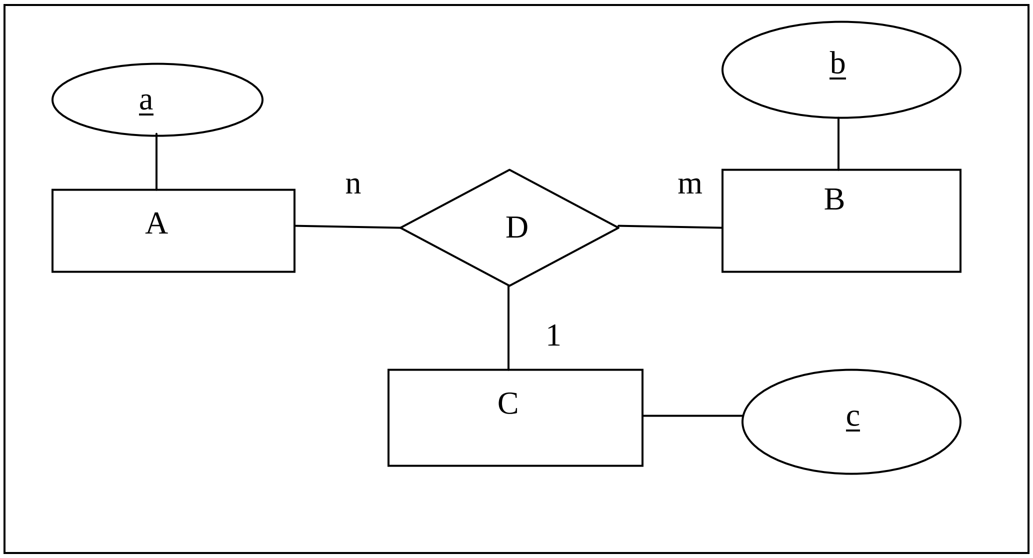
rNr | date | author

admin:

dNr | pNr | rNr

3.1.5 Arbitrary examples of other ternary relationships

i) Many-many-one relationship

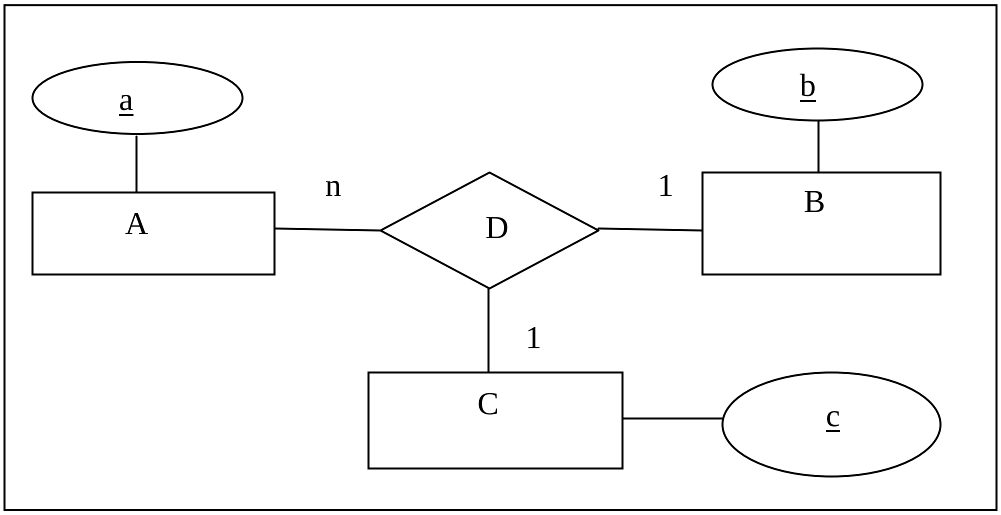


Converting the above ER diagram to a relational model:

A: B: C: D:

a b c a | b | c
fk

ii) Many-one-one relationship

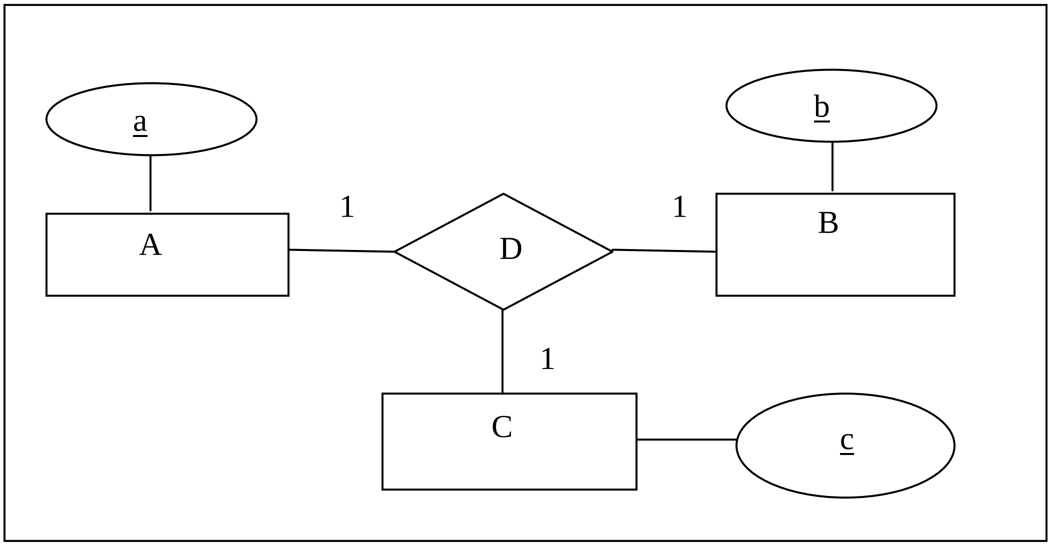


Converting the above ER diagram to a relational model:

A:	B:	C:	D:
a	b	c	a | b | c fk fk

iii)One-one-one relationship

a)

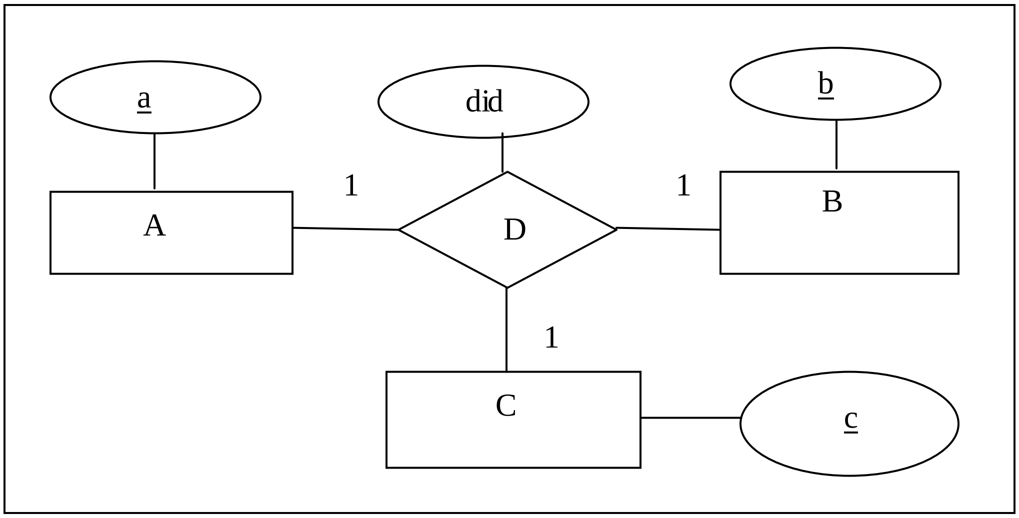


Converting the above ER diagram to a relational model:

A:	B:	C:	D:
a	b	c	a | b | c

For the relationship D to be modeled as a relation, primary keys a, b, c from entities A, B and C respectively go to D as foreign keys, which in turn are reset altogether as a primary key for the relation.

b) Alternative relational model for one-one-one relationship



Converting the above ER diagram to a relational model:

A:	B:	C:	D:
a	b	c	did | a | b | c
			fk fk fk

Alternatively, if the relationship D has already a primary key did, it can be modeled as a relation by taking primary keys a, b, c from entities A, B and C respectively as foreign keys.

Note: As with many-many binary relationship, with ternary relationships, the primary key of the relation modeled from the relationship needs to be determined after careful inspection.

3.2 Relational model Problems

3.2.1 The Banking Enterprise Problem

Referring to the ER model for Banking Enterprise Problem in section 2.9(i), it can be converted to a relational model as follows:

Customer:

customer_id | customer_name | customer_street | customer_city

Branch

branch_name | branch_city | assets

Account:

account_number | type | balance | branch_name
fk

Deposit:

customer_id | account_number

Loan:

loan_number | branch_name | amount
fk

Borrow:

customer_id | loan_number

Payment:

pay_no | loan_number | date | amount
fk

Employee:

employee_id | name | designation | contact_no

Serves:

customer_id | employee_id | date

3.2.2 The University Organization Problem

Student:

matNr | sName

Professor:

pname | psalary

Class:

classNr | room | day | pname
fk

Takes:

matNr | classNr | grade

TA:

matNr | classNr | hours | tasalary

3.3 Query Languages

A query language is a language in which a user requests information from the database. These languages can be categorized as either procedural or non-procedural. In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a non-procedural language, the user describes the desired information without giving a specific procedure for that information.

Most commercial relational database systems offer a query language that includes elements of both the procedural and nonprocedural approaches. The most widely used query language SQL (Structured Query Language) is such a query language.

There are a number of pure query languages. The relational algebra is procedural, where as the tuple relational calculus and domain relational calculus are nonprocedural. These query languages are terse and formal, lacking the “syntactic sugar” of commercial languages, but they illustrate the fundamental techniques for extracting data from the database.

3.4 Relational Algebra

3.4.1 The SELECT operation

The select operation selects tuples that satisfy a given condition. The select operation is denoted by the Greek letter sigma (σ) The condition appears as a subscript to σ. The relation is enclosed in parenthesis after the σ.

Considering the Banking Enterprise problem, if we want to select those tuples of the loan relation where the branch is "Kakrail", we write

$\sigma_{\text{branch\_name}=\text{``kakrail''}}$ (Loan)

Another problem to find all tuples of loan relation in which the amount lent is more than Tk 1000 can be solved as :

$\sigma_{\text{amount}>1000}$ (Loan)

In general, we allow comparisons using =, !=, <, <=, >, >= in the selection condition.

To find those tuples of loan relation in which the amount lent is more than Tk1000 made by Kakrail branch, we write

$\sigma_{\text{branch\_name=''kakrail''} \Lambda \text{ amount}>1000}$ (Loan)

Here $\wedge$ stands for AND.

3.4.2 The Project Operation

The project operation helps to filter out some of the attributes of a relation. In other words we can select part of the attributes of a relation using project operation. Projection is denoted by the uppercase Greek letter pi (Π). We list those attributes of a relation we wish to appear in the result as a subscript to Π. The relation appears in parenthesis after the projection.

If we wish to list all loan numbers and amount of the loans for the loan relation, we write:

$\Pi_{\text{loan\_number, amount}}$ (Loan)

3.4.3 Composition of Relational Operations

Consider the more complicated query “Find those customers who live in Dhaka city”. We write:

$$\Pi_{\text{customer\_name}} (\sigma_{\text{customer\_city = “Dhaka”}} (\text{Customer}))$$

This query is a composition of the relational operations of both select and project.

3.4.4 The Union Operation

Consider a query to find the names of all bank customer ids who have either an account or a loan or both. Note that the customer relation does not contain the information but to answer this query, we need to extract information from both the deposit and borrow relations:

We know how to find the names of all customers with a loan in the bank:

$$\Pi_{\text{customer\_id}} (\text{Borrow})$$

We also know how to find the names of all customers with an account in the bank:

$$\Pi_{\text{customer\_id}} (\text{Deposit})$$

To answer the query we need the union of these two sets; that is, we need all customer names that appear in either or both of the two relations. So we write:

$$\Pi_{\text{customer\_id}} (\text{Borrow}) \; \text{U} \; \Pi_{\text{customer\_id}} (\text{Deposit})$$

3.4.5 The Set-Difference Operation

The set-difference operation, denoted by -, allows us to find tuples that are in one relation but are not in another. The expression r-s produces a relation containing those tuples in r but not in s.

We find all customer ids of the bank who have an account but not a loan by writing:

$$\Pi_{\text{customer\_id}} (\text{Deposit}) - \Pi_{\text{customer\_id}} (\text{Borrow})$$

3.4.6 The Set-Intersection Operation

The set-intersection operation is denoted by ∩. Suppose that we wish to find all customers who have both a loan and an account. Using set intersection, we can write

$$\Pi_{\text{customer\_id}} \text{ (Borrow) } \cap \Pi_{\text{customer\_id}} \text{ (Deposit)}$$

3.4.7 The Cartesian-Product Operation

The Cartesian-product operation, denoted by a cross (x), allows us to combine information from any two relations. We write the Cartesian product of relations r1 and r2 as r1 X r2. For example, the relation schema r= borrow x loan is

(borrow.customer_id, borrow.loan_number, loan.loan-number, loan.branch_name, loan.amount)

We can write the relation schema for r as

(customer_id, borrow.loan_number, loan.loan-number, branch_name, amount)

We have dropped relation-name prefixes from those attributes that appear in only one of the two schemas and with the simplified relation schema, we can distinguish borrow.loan_number from loan.loan_number.

Now consider sample values in the two relations borrow and loan.

Borrow:

customer_id	loan_number
01	L-15
02	L-16
03	L-17

Loan:

loan_number	branch_name	amount
L-15	Kakrail	10000
L-17	Motijheel	15000
L-20	Rampura	20000

Now r=borrow x loan would be:

customer_id	borrow.loan_number	loan.loan_number	branch_name	amount
01	L-15	L-15	Kakrail	10000
01	L-15	L-17	Motijheel	15000
01	L-15	L-20	Rampura	20000
02	L-16	L-15	Kakrail	10000
02	L-16	L-17	Motijheel	15000
02	L-16	L-20	Rampura	20000
03	L-17	L-15	Kakrail	10000
03	L-17	L-17	Motijheel	15000
03	L-17	L-20	Rampura	20000

Thus we see that in the Cartesian product of borrow and loan relations, every tuple of borrow relation is combined with every tuple of loan relation.

Suppose we want to find the customer ids with loans at Kakrail Branch.

First we need to find that in Kakrail branch customers have a loan in the bank. We write:

$$\sigma_{\text{branch\_name}=\text{“kakrail”}} (\text{Borrow x Loan})$$

Since the Cartesian-product operation combines every tuple of loan with every tuple of borrow, we know that, if a customer has a loan in the Kakrail branch, then there is some tuple in borrow x loan that contains his id, and borrow.loan_number = loan.loan_number. So if we write,

$$\sigma_{\text{branch\_name}=\text{“kakrail”} \ \Lambda \ \text{borrow.loan\_number=loan.loan\_number}} (\text{Borrow x Loan})$$

we get only those tuples of borrow x loan that pertain to customers who have a loan at the Kakrail branch.

Finally, since we want only customer_id, we do a projection:

$$\Pi_{\text{customer\_id}} (\sigma_{\text{branch\_name}=\text{“kakrail”} \ \Lambda \ \text{borrow.loan\_number=loan.loan\_number}} (\text{Borrow x Loan})) \quad \text{........ [1]}$$

The result of this expression, shown in Table 3.1, is the correct answer to the query.

Table 3.1: The result of expression [1]

customer_id
01

3.4.8 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names by the rename operator, denoted by the lowercase Greek letter rho (ρ); let us do this. Given a relational-algebra expression E, the expression

$\rho_{x}(E)$

returns the result of expression E under the name x.

We can also apply the rename operation to a relation r to get the same relation under a new name.

A second form of the rename operation is as follows. Assume that a relational-algebra expression has arity n. Then, the expression

$\rho_{x\,(A1,A2\ldots\ldots An)}(E)$

returns the result of Expression E under the name x, and with the attributes renamed to A1, A2…….An.

To illustrate renaming a relation, we consider the query "Find the largest account balance in the bank". Our strategy is to (1) compute first a temporary relation consisting of those balances that are not the largest and (2) take the set difference between the relations:

$\Pi_{\text{account.balance}}$ (Account) and the temporary relation just computed, to obtain the result.

We shall use the rename operation to rename one reference to the account relation. Thus, we can reference the relation without ambiguity.

We can now write the temporary relation that consists of balances that are not the largest..

$\Pi_{\text{account.balance}}\ (\sigma_{\text{account.balance} < \text{d.balance}}\ (\text{Account x } \rho_{d}(\text{Account})))$

The query to find the largest account balance in the bank can be written as:

Π_{balance} (account) -

$\Pi_{\text{account.balance}}$ ($\sigma_{\text{account.balance < d.balance}}$ (Account x ρ_{d}(Account)))

3.4.9 The Natural-Join Operation

It is often desirable to simplify certain queries that require a Cartesian product. Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian Product. Consider the query “Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount.”

$\Pi_{\text{customer\_name, loan.loan\_number, amount}}$ ($\sigma_{\text{borrow.loan\_number= loan.loan\_number}}$ (Borrow x 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 ∞. The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both the relation schemas and finally removes duplicate attributes.

We express this query by using the natural join as follows:

$\Pi_{\text{customer\_name, loan\_number, amount}}$ (Borrow ∞ Loan)

- Find the names of all branches with customers who have an account in the bank and who live in Dhaka.

$\Pi_{\text{branch\_name}}$ ($\sigma_{\text{customer\_city = “Dhaka”}}$ (Customer ∞ Account ∞ Deposit))

The natural join operation on the three relations can be executed in any order.

- Find all customers who have both a loan and an account at the bank.

$\Pi_{\text{customer\_name}}$ (Borrow ∞ Deposit)

Note that in section 3.46 we wrote an expression for this query by using set intersection. We repeat this expression here.

$\Pi_{\text{customer\_id}}$ (Borrow) $\cap$ $\Pi_{\text{customer\_id}}$ (Deposit)

3.5 Outer Join

The outer join is an extension of the join operation to deal with missing information. Suppose that we have the relations with the following schemas, which contain data on full-time employees.

employee(employee_name, street, city)
fullt_works(employee_name, branch_name, salary)

Table 3.2 : Table employee

employee_name	street	city
Kamal	Elephant Rd	Dhaka
Jamal	Mirpur Rd	Dhaka
Shumon	New Market Rd	Chittagong
Hena	Dorga Gate Rd	Sylhet

Table 3.3 : Table fullt_works

employee_name	branch_name	Salary
Kamal	Kakrail	10000
Jamal	Motijheel	15000
Swarna	Rampura	20000
Hena	Dhanmondi	25000

Consider the employee and fullt_works relations in Tables 3.2 snd 3.3 respectively. Suppose that we want to generate a single relation with all the information (street, city, branch_name and salary) about full-time employees. A possible approach would be to use the natural join operation as follows:

employee ∞ fullt_works

The result of this expression appears in Table 3.4.

Table 3.4: employee ∞ fullt_works

employee_name	Street	city	branch_name	salary
Kamal	Elephant Rd	Dhaka	Kakrail	10000
Jamal	Mirpur Rd	Dhaka	Motijheel	15000
Hena	Dorga Gate Rd	Sylhet	Dhanmondi	25000

Note that we have lost the street and city information about Shumon, since the tuple describing Shumon is absent from the fullt_works relation; similarly, we have lost the branch_name and salary information about Swarna, since the tuple describing Swarna is absent from the employee relation.

We can use the outer join operation to avoid this loss of information. There are actually three forms of the operation: left outer join, denoted by, (⟕); right outer join, denoted by, (⟖) and full outer join, denoted by, (⟗)

All three forms of outer join compute the join, and add extra tuples to the result of the join. The results of the expressions employee ⟕ fullt_works, employee ⟖ fullt_works and employee ⟗ fullt_works appear in Tables 3.5, 3.6 and 3.7 respectively.

The left outer join takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation, and adds them to the result of the natural join. In fig 3.5 tuple (Shumon, New Market Rd, Chittagong, Null, Null) is such a tuple. All information from the left relation is present in the result of the left outer join.

The right outer join is symmetric with the left outer join. It pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join. In fig 3.6 tuple (Swarna, Null, Null, Rampura, 20000) is such a tuple.

The full outer join does both of these operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join. Fig 3.7 shows the result of a full outer join.

Table 3.5: employee ⟕ fullt_works

employee_name	Street	City	branch_name	salary
Kamal	Elephant Rd	Dhaka	Kakrail	10000
Jamal	Mirpur Rd	Dhaka	Motijheel	15000
Shumon	New Market Rd	Chittagong	Null	Null
Hena	Dorga Gate Rd	Sylhet	Dhanmondi	25000

Table 3.6: employee ⟖ fullt_works

employee_name	Street	City	branch_name	salary
Kamal	Elephant Rd	Dhaka	Kakrail	10000
Jamal	Mirpur Rd	Dhaka	Motijheel	15000
Swarna	Null	Null	Rampura	20000
Hena	Dorga Gate Rd	Sylhet	Dhanmondi	25000

Table 3.7: employee ⟗ fullt_works

employee_name	Street	City	branch_name	salary
Kamal	Elephant Rd	Dhaka	Kakrail	10000
Jamal	Mirpur Rd	Dhaka	Motijheel	15000
Shumon	New Market Rd	Chittagong	Null	Null
Swarna	Null	Null	Rampura	20000
Hena	Dorga Gate Rd	Sylhet	Dhanmondi	25000

3.6 The Division Operation

The division operation in relational algebra, denoted by ÷, is suited to queries that include the phrase “for all”.

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

First, we can obtain all branches in Dhaka by the expression

$r1= \Pi_{\text{branch\_name}} (\sigma_{\text{branch\_city=“Dhaka”}} (\text{branch}))$

We can find all (customer_name, branch_name) for which the customer has an account at a branch by writing

$r2 = \Pi_{\text{customer\_name, branch\_name}}$ (deposit ∞ account)

We need to find customers who appear in r2 with every branch name in r1. The operation that provides exactly those customers is the divide operation that is, the customers who have an account for all branches in Dhaka city.

$r2 \div r1$

3.7 Modification of the Database

3.7.1 Deletion

Here are several examples of relational algebra delete operations:

- Delete all of Smith's account records

  Deposit <- Deposit - $\sigma_{\text{customer\_name="Smith"}}$ (Deposit)

- Delete all loans with amount in the range 0 to 50.

  Loan <- Loan - $\sigma_{\text{amount} >=0 \wedge \text{amount} <= 50}$ (Loan)

- Delete all accounts at branches located in Dhaka.

  r1 <- $\sigma_{\text{branch\_city ="Dhaka"}}$ (account ∞ branch)
  r2 <- $\Pi_{\text{branch\_city, account\_number, balance}}$ (r1)
  account <- account – r2

3.7.2 Insertion

Suppose we wish to insert the fact that Smith has Tk1200 in account A-973 at Kakrail branch. We write:

account <- account U {(A-973, "Kakrail", 1200)}
deposit <- deposit U {("Smith", A-973)}

3.7.3 Updating

Suppose that interest rates are being made and that all branches are to be increased by 5%. For this, we require an update operation.

$$account \leftarrow \Pi_{account\_number,\ branch\_name,\ balance * 1.05} (account)$$

Suppose that accounts with balances over Tk10,000 receive 6% interest, where as all others receive 5%.

$$account \leftarrow \Pi_{account\_number,\ branch\_name,\ balance * 1.06} (\sigma_{balance>10000} (account)) \cup$$

$$\Pi_{account\_number,\ branch\_name,\ balance * 1.05} (\sigma_{balance<=10000} (account))$$

CHAPTER 4

THE QUERY LANGUAGE SQL

4.1 Introduction to SQL

SQL stands for Structured Query Language. SQL is today the most widely used query language for relational databases.

Each column/attribute has a specific data type. Basic data types in relational systems are as follows:

- Smallint: an integer number between -32,768 and 32,767
- Integer: an integer number between -2,147,483,648 and +2,147,483,648
- Numeric(p,q) or Decimal(p,q): a fixed point number with at most p digits, q of them after the decimal point.
- Float, Real, Double Precision: floating point numbers with different precisions
- Char(n): a string up to n characters. 0<n<=255. Each value in a column of type Char(n) requires storage space for n characters
- Varchar(n) or Character Varying(n): a string up to n characters. e.g. 0<n<=255. Each value in a column of such a type requires only enough storage to store the actual characters (which can be fewer than n).
- Date: to store calendar dates with year, month and day fields
- Time: to store time with hour, minute and second fields
- Timestamp: to store combinations of date and time
- Interval: to store time intervals
- Bit(n): a string of bits with fixed length n
- Bit Varying (n): a string of bits with varying length up to n bits

We start with some SQL examples related to the University Organization Problem.

Q1 Find the matriculation number of the student whose name is John.

Query Expression

```
SELECT  matNr FROM Student WHERE sName='John'
```

From the above SQL query expression we note the following:

- The SELECT part is equivalent to a projection, not a selection in relational algebra
- After SELECT, all attributes must be listed onto which a projection is executed.

- After FROM, all tables must be listed that are necessary for finding the query result.
- After WHERE, all constraints (formulas) for joins and selections (conditions) in the query must be listed.

Q2 Find all students from the university.

Query Expression:

SELECT * FROM Student

Q2 Find the students who took the class on CSE 303.

Query Expression:

SELECT sName FROM Student, Takes WHERE Student.matNr=Takes.matNr AND classNr= ‘CSE 303’

Q3 Display the names and salaries of professors whose salary is > than 1 lac.

Query Expression:

SELECT * FROM Professor WHERE psalary > 100000

Q4 Find the matNrs of all students younger than the oldest student named Philips.

Query Expression:

SELECT A.matNr FROM Student A, Student B WHERE B.sName = ‘Philips’ AND A.age < B.age

Q5 Find the names of all students who took CSE303, but who were not yet assigned a grade.

In this case querying for NULL values is possible as shown below.

Query Expression

SELECT sName
FROM Student, Takes
WHERE Student.matNr = Takes.matNr
AND Takes.classNr = ’CSE303’
AND Takes.grade = NULL

Q6 Find all students who took the course CSE 303 or worked as a TA for this class.

Query Expression

```
(SELECT Student.matNr, sName
  FROM Student, Takes
  WHERE Student.matNr=Takes.matNr
   AND Takes.classNr= ‘CSE 303’)
   UNION
   (SELECT student.matNr, sName
     FROM Student, TA
     WHERE Student.matNr=TA.matNr
      AND TA.classNr= ‘CSE 303’)
```

Q7 Find all names of students who took any class not given by Professor James.

Query Expression:

```
SELECT  sName
FROM Student
WHERE matNr IN
          (SELECT matNr
            FROM Takes
            WHERE classNr NOT IN
                        (SELECT classNr
                          FROM class
                          WHERE pname= ‘James’))
```

4.2 SQL Expressions using functions

Functions:

- AVG() (= average)
- MAX()
- MIN()
- COUNT() (= number of tuples)
- SUM()
- DISTINCT() (= list every result value just once; remove duplicates)

Q1 Find the number of tuples in table Professor.

```
SELECT COUNT(*) FROM Professor
```

Q2 List the sum of all professors' salaries, the highest, the smallest, and the average salary in the table.

```
SELECT SUM(psalary), MAX(psalary), MIN(psalary), AVG(psalary) FROM Professor
```

Q3 What is the number of different classes for which teaching assistants work?

```
SELECT COUNT(DISTINCT(classNr)) FROM TA
```

4.3 SQL Expressions using Grouping

Q1 List for every class the classNr, the number of students who took the class, and the average exam grade.

```
SELECT classNr, COUNT(*), AVG(grade)
FROM Takes
GROUP BY classNr
```

What happens using the GROUP BY clause can be explained as follows:

Let us consider the following arbitrary Takes table:

matNr	classNr	grade
01	CSE 303	4
02	CSE 303	3
02	CSE 304	4
03	CSE 304	4

The group by clause is going to group on the classNrs meaning there will be two groups of classNrs : CSE 303 and CSE 304, and for each group the classNr, number of students who took the class and average grade will be listed. So the result will be:

classNr	no of students	average grade
CSE 303	2	3.5
CSE 304	2	4

GROUP BY must always be used whenever the Select part contains attributes with aggregate functions and attributes to which no aggregate functions are applied. In these cases all attributes without aggregate function must be listed in the GROUP BY clause.

Q2 List for every class the classNr, the number of students who took the class, and the average exam grade where the grade average is better than 3.

```
SELECT classNr, COUNT(*), AVG(grade)
FROM Takes
GROUP By classNr
HAVING AVG(grade) > 3
```

A GROUP BY clause can be extended by a HAVING clause which gives a constraint that is applied to the group results.

4.4 SQL Expressions using Sorting

Q List names, grades of students with the grades in descending order.

```
SELECT sName, grade
FROM Student, Takes
WHERE Student.matNr=Takes.matNr
ORDER BY grade DESC
```

Using ORDER BY in the above expression, the grades will be listed in descending order. The default is ASC (ascending order).

4.5 Searching for partial strings

The operators LIKE and NOT LIKE use wild card characters.

- % stands for an arbitrary number (0 or more) of arbitrary characters.
- _ stands for exactly one arbitrary character

Q1 Find all rooms that were used for classes in CSE.

```
SELECT room FROM Class WHERE classNr LIKE ‘CSE%’
```

Q2 Find all students whose names do not end with the letter m.

```
SELECT sName FROM Student WHERE sName NOT LIKE ‘%m’
```

Q3 Find all students whose names contain the letter e.

```
SELECT sName FROM Student WHERE sName LIKE ‘%e%’
```

4.6 SQL Expressions concerning Input of Data

Inserting tuples: (Examples)

INSERT INTO Student VALUES (10, ‘James’)

INSERT INTO Class (pname, classNr) VALUES (‘Jane’, ‘CSE 303’), (‘David’, ‘CSE 304’)

Values that are not specified are automatically either set to NULL or to a default value.

4.7 SQL Expressions concerning deletion of tuples

Examples:

DELETE FROM Professor WHERE pname= ‘David’

DELETE FROM Student

- Deletes all tuples from the table Student

4.8 Update of tuples/values

Examples:

Update Class
SET room=’7B03’, day= ‘Friday’
WHERE classNr=’CSE 304’

Several rows can be updated in one statement.

Q. All TAs get a rise of 10% increase of their salary

UPDATE TA
SET tasalary = 1.1 * tasalary

4.9 Deletion of tables

DROP TABLE tablename {CASCADE|DELETE}

If CASCADE is specified:

All views and references from other tables that refer to this table are also deleted.

If RESTRICT is specified:
Deletion is only executed if the table is not referenced by other tables or views.

4.10 Schema updates

Adding a column:

```
ALTER TABLE tablename
ADD column_name data_type
```

Example:

```
ALTER TABLE Student
Add Birthday Date
```

- NOT NULL cannot be specified here

Dropping a column:

```
ALTER TABLE tablename
DROP columnname CASCADE|RESTRICT
```

DROP requires appending CASCADE or RESTRICT
With CASCADE, all references and views that refer to this column will automatically be deleted with the drop. With RESTRICT, the dropping is executed only if there are no such references to this column.

Example:

```
ALTER TABLE Student
DROP Birthday CASCADE
```

4.11 Definition of a Domain

A domain is essentially a data type with optional constraints (restrictions on the allowed set of values).

```
CREATE DOMAIN domainname AS type
[CHECK (condition)]
```

Examples:

```
CREATE DOMAIN MatNr AS CHAR(6);

CREATE DOMAIN GenderType AS CHAR
CHECK (VALUE IN (‘M’, ‘F’));
```

The second example allows to define restrictions on the values in the desired domain **GenderType**.

4.12 Creation and Deletion of Views

```
CREATE VIEW view_name { (column name, ……)} AS
SELECT_statement

DROP VIEW view_name;
```

Example:

```
CREATE VIEW CSE303_Students (name, matNo) AS
SELECT sName, matNr
FROM Student, Takes
WHERE Student.matNr = Takes.matNr
And ClassNr = ‘CSE303’;
```

Normally, views are not stored like base tables but are generated just in time when a query accesses the view. Some systems allow storing views for performance reasons.

Referring to the Bank Enterprise, here are some sample SQL queries and corresponding SQL expressions:

4.13 The Rename Operation

Q For all customers who have a loan from the bank, find their names, loan numbers and loan amount.

While we can write the SQL expression for the above as

SELECT customer_name, loan_number, amount FROM borrow, loan WHERE borrow.loan_number=loan.loan_number

We can write the above expression using a rename operation as:

```
Select customer_name, T.loan_number, S.amount
From borrow AS T, loan as S
WHERE T.loan_number =  S.loan_number
```

4.14 UNION ALL

Q Find all bank customers having a loan, an account or both at the bank.

We can write the expressiom for the above query using a normal UNION operation as:

```
(SELECT customer_name FROM deposit
UNION
(SELECT customer_name FROM borrow)
```

If we want to retain all duplicates, we must write **UNION ALL** in place of **UNION** as:

```
(SELECT customer_name FROM deposit)
        UNION ALL
(SELECT customer_name FROM borrow)
```

4.15 INTERSECT & INTERSECT ALL

Q Find all customers who have both a loan and an account at the bank.

```
(SELECT DISTINCT customer_name FROM deposit)
                INTERSECT
(SELECT DISTINCT customer_name FROM borrow)
```

If we want to retain duplicates, we write:

```
(SELECT DISTINCT customer_name FROM deposit)
                INTERSECT ALL
(SELECT DISTINCT customer_name FROM borrow)
```

4.16 Inner and Outer Joins

Let us consider the following relations on loan and borrow from the banking example:

Table 4.1: Loan relation

loan_number	branch_name	Amount
L-170	Kakrail	3000
L-230	Motijheel	4000
L-260	Dhanmondi	1700

Table 4.2: Borrow relation

customer_name	loan_number
Kamal	L-170
Tinku	L-230
Jones	L-155

We start with a simple example of inner joins:

loan inner join borrow on loan.loan_number = borrow.loan_number

The expression computes the normal join of loan and borrow relations with the join condition being loan.loan_number = borrow.loan_number

The result of the above join is shown below:

Table 4.3: The result of loan inner join on loan.loan_number = borrow.loan_number

loan_number	branch_name	amount	customer_name	loan_number
L-170	Kakrail	3000	Kamal	L-170
L-230	Motijheel	4000	Tinku	L-230

We may rename the result relation of a join and the attributes of the result relation by using an **'as'** clause as shown below:

loan inner join borrow on loan.loan_number = borrow.loan_number as lb(loan_number, branch, amount, cust, cust_loan_num)

We rename the second occurrence of loan_number to cust_loan_num. The ordering of the attributes in the result of the join is important for the renaming.

Now we consider an example of left outer-join operation:

Loan left outer join borrow on loan.loan_number = borrow.loan_number

The following table shows the result of the above left outer join.

Table 4.4: The result of loan left outer join borrow on loan.loan_number = borrow.loan_number

loan_number	branch_name	amount	customer_name	loan_number
L-170	Kakrail	3000	Kamal	L-170
L-230	Motijheel	4000	Tinku	L-230
L-260	Dhanmondi	1700	NULL	NULL

In the resultant relation, the tuples (L-170, Kakrail, 3000) and (L-230, Motijheel, 4000) from loan join with tuples from borrow and appear in the result of the inner join and hence in the result of the left outer join. On the other hand the tuple(L-260, Dhanmondi, 1700, NULL, NULL) is present in the result of the left outer join.

Finally we consider an example of natural-join operation:

loan natural inner join borrow

This expression computes the natural join of the two relations. The only attribute name common to loan and borrow is loan_number and it appears once in the result unlike the result of the join with the **'on'** condition.

The right outer join is symmetric to the left outer join. Tuples from the right-hand-side relation that do not match any tuples in the left-hand-side relation are padded with nulls and are added to the result of the right outer join.

Here is an example of combining the natural-join condition with the right outer join type:

loan natural right outer join borrow

The result of the above erxpression is shown below:

Table 4.5: The result of loan natural right outer join borrow

loan_number	branch_name	Amount	customer_name
L-170	Kakrail	3000	Kamal
L-230	Motijheel	4000	Tinku
L-155	NULL	NULL	Jones

Now we consider a full outer join operation. For example, the following table shows the result of full outer join expression.

loan full outer join borrow using (loan_number)

Table 4.6: The result of loan full outer join borrow using (loan_number)

loan_number	branch_name	amount	customer_name
L-170	Kakrail	3000	Kamal
L-230	Motijheel	4000	Tinku
L-260	Dhanmodi	1700	NULL
L-155	NULL	NULL	Jones

As another example, we can write the query, “Find all customers who have either an account or loan but not both at the bank” with natural full outer join as:

```
SELECT customer_name FROM (deposit natural full outer join borrow)
WHERE account_number is NULL OR loan_number is NULL
```

CHAPTER 5

INTEGRITY CONSTRAINTS IN RELATIONAL SYSTEMS

In relational systems, usually the following types of integrity constraints are considered:

- Required data (NOT NULL)
- Domain constraints
- Entity integrity
- Referential integrity

5.1 Required Data

Some attributes must contain a valid value. For example, we could assume that every student must have a name in the database. There cannot be student tuples where the name is not given.

This constraint can be defined in the CREATE TABLE statement by adding NOT NULL after the data type of the attribute.

Example:

```
CREATE TABLE Student
(sName Varchar (20) NOT NULL,
 matNr Integer NOT NULL);
```

⇨ The database then will not accept the input of tuples where a name is not given.

5.2 Domain Constraints

By defining a domain, a set of legal values corresponding to a column of a specific data type can be defined. The database then checks at insertion time whether the inserted value is allowed or not. If it is not allowed, the insertion is rejected.

Example:

```
CREATE DOMAIN genderType AS CHAR(1)
CHECK (VALUE IN (‘M’, ‘F’));
```

```
CREATE TABLE Student
( sName Varchar(20) NOT NULL,
  matNr Integer NOT NULL,
  gender genderType);
```

5.3 Entity Integrity

The primary key of a table must contain a unique, non-null value for each row. This can be ensured by adding the phrase PRIMARY KEY(A1,……An) to a CREATE TABLE statement. It can only be used only once for each table.

Example:

```
CREATE TABLE TA
(matNr Integer NOT NULL,
classNr CHAR(10) NOT NULL,
hours Integer,
tasalary Double)
PRIMARY KEY(matNr, classNr)
```

It is also possible to enforce uniqueness and presence of a defined value (other than NULL) for other attributes, for instance alternate keys, by using the expressions UNIQUE and NOT NULL.

Example:

```
CREATE TABLE TA
( matNr  Integer NOT NULL,
 classNr  Char(10) NOT NULL,
 hours Integer,
 tasalary Double)
 PRIMARY KEY(matNr, classNr)
 UNIQUE(classNr)
```

This would mean that the database accepts only one TA for each class.

5.4 Referential Integrity

Relationships are modeled in the relational model by using foreign keys. Referential integrity means that, if the foreign key contains a value, the value must refer to an existing, valid row in the table it refers to.

The definition of foreign keys is supported with the FOREIGN KEY clause in the CREATE TABLE statement.

Example:

```
CREATE TABLE TA
(matNr Integer NOT NULL,
 classNr Char(10) NOT NULL,
 hours Integer,
 tasalary Double)
PRIMARY KEY (matNr, classNr)
FOREIGN KEY (matNr) REFERENCES Student (matNr)
FOREIGN KEY (classNr) REFERENCES Class (classNr)
```

The database then rejects any INSERT or UPDATE operation that attempts to create a foreign key value without a matching value in the table to which it refers.

The action that the DBMS takes for any UPDATE or DELETE operation that concerns a value that matches foreign key values in another table is dependent on the "referential action" specified using the ON UPDATE and ON DELETE subclauses of the FOREIGN KEY clause, discussed as follows.

DELETION

When a tuple with a primary key value is deleted, the following actions can be performed on the tuples with the corresponding foreign key values:

a. CASCADED DELETE

All tuples whose foreign key values equal the deleted primary key values are also deleted.
(Example: when a tuple in student is deleted, matching foreign key tuples in Takes should also be deleted)

In SQL: ON DELETE CASCADE

Example:

```
CREATE TABLE Takes
(matNr Integer NOT NULL
classNr Varchar(10) NOT NULL
FOREIGN KEY (matNr) REFERENCES Student (matNr)
ON DELETE CASCADE
```

b. RESTRICTED DELETE

As long as tuples exist with foreign key values that match the primary key value which is to be deleted, the deletion is not accepted.

(Example: a department record may not be deleted as long as there are tuples with employees belonging to this department). This is the default setting unless no action on delete is specified.

IN SQL: ON DELETE RESTRICT

c. No Action

With the clause ON DELETE NO ACTION it is specified that a deletion of a tuple with the primary key does not invoke any action on the tuples(s) with the concerned foreign key value.

Example:

```
CREATE TABLE Takes
(matNr Integer NOT NULL,
classNr VARCHAR(10) NOT NULL)
FOREIGN KEY (classNr) REFERENCES Class (classNr)
ON DELETE NO ACTION
```

d. Nullification

All foreign key values that match the deleted primary key value are set to NULL. (Example: when a professor is deleted, their name is set to NULL in the Class tuples)

IN SQL: ON DELETE SET NULL

Example:

```
Create TABLE Class
(classNr Char(10) NOT NULL,
 room Integer,
day Date,
pname VarChar(30))
PRIMARY KEY (classNr)
FOREIGN KEY (pname) REFERENCES Professor (pname)
ON DELETE SET NULL
```

e. Default

Only the foreign key values are not set to NULL but to a default value.

IN SQL: ON DELETE SET DEFAULT

For example,

```
CREATE TABLE DegreePrograms
(programId Integer
……
studentCounsellor Integer NOT NULL DEFAULT 123,
……)
PRIMARY KEY(programId)
FOREIGN KEY (studentCounsellor) references Staff (persId)
ON DELETE SET DEFAULT
```

Suppose, there are the following two tables:

Table 5.1: Tables Staff and DegreePrograms

Staff

persId	persname	Designation
001	John	Head
002	Mary	Head
003	Henry	Head
123	David	Dean

DegreePrograms

programId	Program	studentCounsellor (fk)
01	CSE	001
02	EEE	002
03	CE	003

In above tables studentCounsellor in DegreePrograms table is a foreign key to the primary key persId in Staff table. The default studentCounsellor is Dean of Engineering. If studentCounsellor Henry with persId 003 is deleted from Staff, then the default studentCounsellor would be the Dean with persId as 123. The result of this action is shown in the following tables:

Table 5.2: Resulting Tables Staff and DegreePrograms

Staff

persId	persname	Designatiom
001	John	Head
002	Mary	Head
~~003~~	~~Henry~~	~~Head~~
123	David	Dean

DegreePrograms

programId	Program	studentCounsellor
01	CSE	001
02	EEE	002
03	CE	~~003~~ 123

5.5 Modification

When a primary key value is updated, the following actions can be performed on the tuples with the corresponding foreign key values. All cases are analogous to the case of deletion.

a. Cascaded Update: ON UPDATE CASCADE
b. Restricted Update: ON UPDATE RESTRICT
c. Nullification: ON UPDATE SET NULL
d. Default: ON UPDATE SET DEFAULT
e. No Action: ON UPDATE NO ACTION

Note:
It is possible to combine for the same attribute different types of actions for delete and update, for example adding to a foreign key definition the clause ON UPDATE SET DEFAULT ON DELETE CASCADE.

CHAPTER 6

FUNDAMENTAL DEPENDENCIES AND NORMALIZATION

6.1 Definition of functional dependency

Y is said to be functionally dependent on X if for any pair of tuples, two different values of Y do NOT correspond to the same values of X.

Notation: X -> Y

Example:

From our University database example:

Student(matNr, sName)

{matNr} -> {sName}

Two different sNames do NOT correspond to the same matNr. So sName is functionally dependent on matNr.

Other examples of functional dependencies from Banking Enterprise example:

{account_number} -> {balance}
{branch_name} -> {assets}
{loan_number }-> {amount}
{pay_no } -> {amount}

6.2 Definition of full functional dependencies

Y is said to be **fully** functionally dependent on X if there is no proper subset X’ C X where X’ -> Y.

Notation X=> Y

Example:

Our familiar university example:

Class(classNr, room, day, pname)

{classNr, room} -> {pname}
{classNr} => {pname}

The set {classNr, room} on the left can be further reduced to the set {classNr}. Two different professors do NOT correspond to the same classNr. room can be ignored. So pname is fully functionally dependent on classNr.

Similarly:

{classNr, day , pname} -> {room}
{classNr} => {room}

6.3 Normalization

Codd introduced a number of “normal forms”. They are principles that can hold for a relation or not. Relations can be transformed in order to normalize them. We will be talking of first normal form (1NF), second normal form (2NF), third normal form (3NF) and Boyce Codd Normal Form (BCNF).

6.3.1 First Normal form

Definition:

A relation is in first normal form if it contains only simple, atomic values for attributes, no sets. In other words, attributes should not have subattributes.

According to Codd, all relations in the relational model must be in 1NF.

Example:

A relation that is not in 1NF

Relation Person:

Name	offspring		Place
	child	age	
James	Christa	12	Sweden
	Peter	10	
	Iris	9	
Schmidt	Martin	17	Germany
	Rainer	18	

Ways to make the above relation in 1NF:

First attempt:

Person(name, place, child1, child2, child3)

⇨ Not good. Reason: either not enough available columns for some data records. (How many children can a person have?) or, if all columns are at all needed? Wastage of space (many null values).

Second Attempt:

Person:

pName	Place
James	Sweden
Scmidt	Germany

Child:

pName	chName	Age
James	Christa	12
James	Peter	10
James	Iris	9
Schmidt	Martin	17
Schmidt	Rainer	18

Advantage:

This requires just the right amount of space that is needed and it is in 1NF.

Disadvantage:

It requires an additional table. pName is redundantly stored.

6.3.2 Second Normal Form

Definition:
A relation is in 2NF if it is in 1NF and every non-primary key attribute is fully functionally dependent on the primary key of the relation.

Example:
Our University Database:
TA(matNr, classNr, sName, hours, tasalary)

Full functional dependencies:

{matNr, classNr}=>{hours}
{matNr, classNr} => {tasalary}
{matNr} => {sName}

TA is not in 2NF because sName is not fully functionally dependent on the primary key (matNr, classNr) but is fully functionally dependent on part of the primary key (matNr).

Solution:
We have to split the original relation to make it 2NF i.e. Move the dependency {matNr} => {sName} to a separate relation=> relation "Student"

After splitting we have:

TA(matNr, classNr, hours, tasalary}
Student(matNr, sName)

Now the relations are in 2NF.

6.3.3 Third Normal Form (3NF)

Definition of transitive dependency:

Z is transitively dependent on X if
a) A chain exists: X => Y => Z
b) Y is not a super key
c) Z is not part of primary key

Definition of third normal form:
A relation is in 3NF if it is in 2NF and no non-primary key attribute is transitively dependent on any primary key.

In other words, there should not be dependencies between non-key attributes.

Example:

Our university database again:

TA(matNr, classNr, hours, tasalary)

Functional dependencies:
{matNr, classNr} => {hours}
{matNr, classNr} => {tasalary}

Assumption:
{hours} => {tasalary}

There is the following transitive dependency:

{matNr, classNr} => {hours} => {tasalary}
Since tasalary is not a part of primary key and hours is not a superkey, a transitive dependency exists, and TA is not in 3NF.

Solution:

Split the TA relation: Move the dependency {hours}=>{tasalary} to a separate relation
i.e. TANew(matNr, classnr, hours) and TAsalary(hours, tasalary)

6.3.4 Boyce-Codd Normal Form (BCNF)

Definition:

A relation is in BCNF if part of a primary key is not fully functionally dependent on any non-primary key attribute.

Example:

Relation:- Speedlimits(town, streetSegment, postcode, speed)

Full functional dependencies:

{town, streetSegment} => {postcode}
{town, streetSegment} => {speed}
{postcode} => {town}
{postcode, streetSegment} => {speed}

Here the relation is not in BCNF because of the dependency: {postcode} => {town}

Part of primary key {town} is fully functionally dependent on nonprimary key {postcode}

To make the relation in BCNF, we split

Speedlimit(<u>town</u>, <u>streetSegment</u>, speed)
Codes(<u>postcode</u>, town)

Now the relation is in BCNF.
There are problems with this decomposition:

- The dependency {town, streetSegment} => {postcode} is no longer recognizable.
- The decomposition is lossy (having redundant information)!! After joining, each streetSegment has all postcodes in a town as shown below:

Suppose, we consider the original relation as:

<u>Town</u>	<u>streetSegment</u>	postcode	speed
Dhaka	A-str	1217	30
Dhaka	B-str	1217	30
Dhaka	C-str	1217	50
Dhaka	D-str	1000	70

Now as a first attempt if we split as

Spedlimit(<u>town</u>, <u>streetSegment</u>, speed)
Codes(<u>postcode</u>, town)

Speedlimit:

<u>Town</u>	<u>streetSegment</u>	speed
Dhaka	A-str	30
Dhaka	B-str	30
Dhaka	C-str	50
Dhaka	D-str	70

Codes

<u>Postcode</u>	town
1217	Dhaka
1000	Dhaka

Now if we join the split tables, each streetSegment has all the postcodes in a town. So the decomposition is lossy as shown:

Town	streetSegment	speed	postcode
Dhaka	A-Str	30	1217
Dhaka	B-Str	30	1217
Dhaka	C-Str	50	1217
Dhaka	D-Str	70	1217
Dhaka	A-str	30	1000
Dhaka	B-str	30	1000
Dhaka	C-str	50	1000
Dhaka	D-str	70	1000

As a second attempt, if we split as:

Speedlimit(town, streetSegment, speed)
Postcodes(streetSegment, postcode)

The relations are in BCNF.
The problems with this decomposition are:

- The dependency {town, streetsegment} => {postcode} is again not recognizable.
- There is repetition of values as shown below in the following tables.

Speedlimit

Town	streetSegment	speed
Dhaka	A-str	30
Dhaka	B-str	30
Dhaka	C-str	50
Dhaka	D-str	70

Postcodes

streetSegment	Postcode
A-str	1217
B-str	1217
C-str	1217
D-str	1000

Third attempt:

Speedlimit(postcode, streetSegment, speed)
Codes(postcode, town)

Here both the relations are in BCNF. The decomposition is lossless and there is less repetition of values.

Speedlimit

postcode	streetSegment	speed
1217	A-str	30
1217	B-str	30
1217	C-str	50
1000	D-str	70

Codes

postcode	town
1217	Dhaka
1000	Dhaka

It is possible to show:

- A relation that is not in BCNF can always be losslessly decomposed towards BCNF.
- A lossless decomposition into BCNF that preserves all dependencies does not always exist.

6.4 Notes on Normalization

Advantages of normalization:

- Many unnecessary redundancies are avoided.
- Anomalies with input, deletion and updates can be avoided
- Fully normalized relations tend to need less space than if not normalized

Disadvantages of normalization:

- Normalization splits entities and relationships into many relations, thus making them harder to understand.
- Queries become more complex because they have to involve more relations.
- Response times are longer because of a higher number of joins in the queries.

6.5 Quality Criteria for Relational Design

1. A relational design should be simple to understand
2. A relational schema should not contain input, update or
3. deletion anomalies.
4. Decompose relations only in a lossless way.
5. Avoid attributes in a relation that would contain too many null values.

CHAPTER 7

QUERY PROCESSING

7.1 Introduction to Query Processing

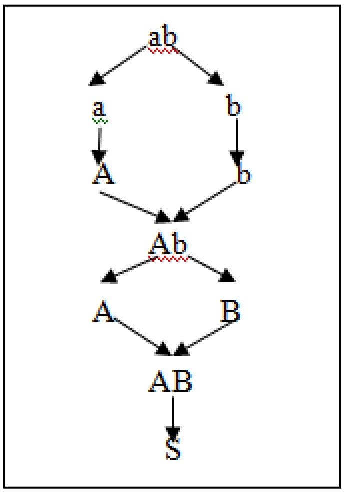
Query Processing refers to the range of activities involved in extracting data from a database.

The steps involved in processing a query are:

1) Parsing and translation
2) Optimizing
3) Evaluation

The first step in query processing is that the system must translate a given query (SQL format) in its internal form, that is, convert it to a relational algebra expression. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of relations in the database and so on. A parse-tree representation of the query is constructed, which is then translated into a relational algebra expression.

As an example let S= relational algebra expression; a, b: parts of SQL command
If the language is
S -> AB
A -> a
B -> b



Based on the above language, bottom up parsing can be as an example as above.
As shown in the example, parts of SQL command (a, b) are converted to a relational algebra expression S via bottom up parsing.

A relational-query language is either declarative or algebraic. Declarative languages permit users to specify what a query should generate without saying how the system should do the generating (e.g, SQL). Algebraic languages allow for algebraic transformations of user's queries (e.g. Relational algebra).

The algebraic basis provided by a relational model helps in query optimization. Query optimization is the process of selecting the most efficient query evaluation plan for a query, which means a method for evaluating the efficient performance cost of a query i.e, a method showing the minimum time it takes to execute the query. An SQL query can be translated into a relational algebra expression in one of several ways:

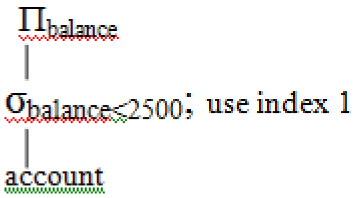
Select balance from account where balance < 2500

This SQL expression can be translated to a relational algebra expression in one of two ways:

- $\sigma_{balance< 2500}$ ($\Pi_{balance}$(account))
- $\Pi_{balance}$ ($\sigma_{balance<2500}$(account))

To implement the 2nd way, we can search every tuple in account to find tuples with balance less than 2500. If an index (sorting the values of an attribute value in a database table) is available on the attribute balance, we can use the index instead.

Query optimization involves the selection of instructions for processing and evaluating each operation of a query such as choosing an algorithm to use for executing an operation, choosing the specific indices to use and so on. A relational algebra operation annotated with instructions on how to evaluate it is called an evaluation primitive. Several primitives may be grouped together into a pipeline in which several operations are performed in parallel. A sequence of primitive evaluations that can be used to evaluate such a query is a query-evaluation plan or a query execution plan.



The above figure illustrates an evaluation plan for our example query in which a particular index 1 (on the attribute balance) is specified for the selection operation. The different evaluation plans for a given query can have different costs. It is the responsibility of the system to evaluate a query evaluation plan that minimizes the cost of query evaluation. The most relevant performance measure is minimizing the number of disk accesses. Optimizers make use of statistical information about the relations such as relation sizes and index depths to make a good estimate of the cost of a plan. In the above figure, the evaluation plan in which the selection is done using the index is likely to have the lowest cost and thus, to be chosen. The query evaluation engine takes a query evaluation plan, evaluates the query with that plan and executes it and returns the answers to the query.

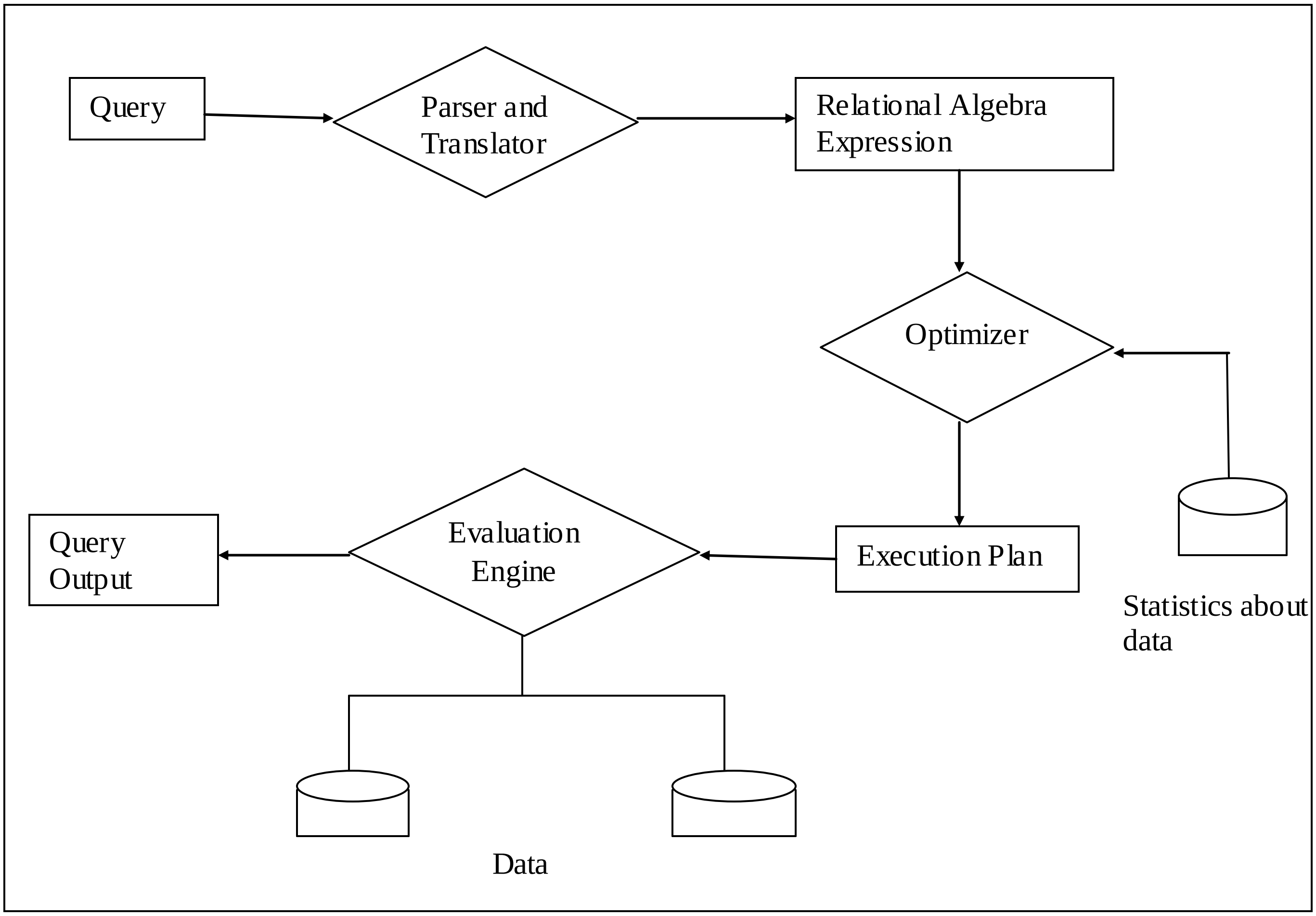


Fig 7: Steps in Query Processing

CHAPTER 8

FILE ORGANIZATION

As operating systems are efficient in managing file systems, there may arise the need to map a database to a file organization. A file is organized logically as a sequence of records. These records are mapped onto disk blocks.

Although blocks are of a fixed size determined by the physical properties of the disk and by the operating system, record sizes vary. In a relational system, tuples of distinct relations are generally of different sizes.

One approach to mapping the database to files is to use several files and to store records of only one fixed length in any given file. An alternative is to structure our files such that we can accommodate multiple lengths for records.

8.1 Fixed-Length Records

Let us consider a file of account records for our bank database. Each record of this file is defined as follows:

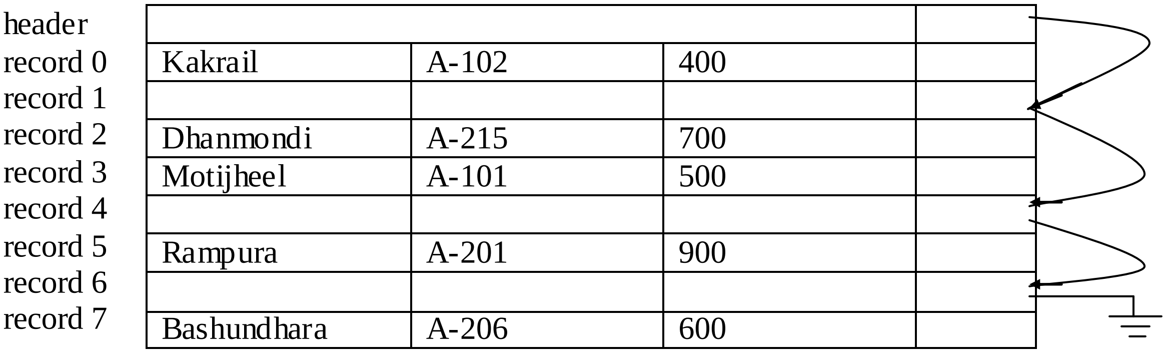
```
Type deposit = record
                    branch_name: char(22);
                    account_number: char(10);
                    balance: real;
                 end
```

If we assume that each character occupies 1 byte and that a real occupies 8 bytes, our account record is 40 bytes long. A simple approach is to use the first 40 bytes for the first record, the next 40 bytes for the second record and so on. There are two problems with this approach:

1) The space occupied by a record to be deleted must be filled with some other record of the file otherwise sequence of records will be lost and wastage of space would result.
2) Part of a record may be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record and so on, until every record following the deleted record has been moved ahead. Such an approach requires moving a large number of records.

A good approach for the above problem is to allocate a certain number of bytes as a file header at the beginning of the file. The header will contain a variety of information about the file. In this header we can store the address of the first record whose contents are deleted. The header thus points to the first available record and the first available record will point to the second available record and so on. The deleted records thus form a linked list which is often referred to as a free list. On insertion of a new record, we use the record pointed to by the header. We change the header to point to the next available record. If no free space is available, we add the new record to the end of the file.



header				
record 0	Kakrail	A-102	400	
record 1				
record 2	Dhanmondi	A-215	700	
record 3	Motijheel	A-101	500	
record 4				
record 5	Rampura	A-201	900	
record 6				
record 7	Bashundhara	A-206	600	

Fig 8.1: Fixed Length Records

8.2 Variable-Length records

An example of variable-length records is shown below:

```
Type account_list = record
                        branch_name: char(22)
                        account_info: array[1..∞] of record;
                                            account_number: char(10);
                                            balance: real;
                                        end
                        end
```

Here account_info is an array with the information account_number and balance stored in indices 1 and 2. This information may be repeated with other values respectively in indices 3 and 4 and so on for each branch_name. So there is no limit on how large a record of this array can grow up to, of course, the size of the disk!

8.3 Byte-String Representation

A simple method for implementing variable-length records is to attach a special end-of-record symbol to the end of each record. We can then store each record as a string of consecutive bytes.

The above approach has disadvantages:

1) It is not easy to reuse space formerly occupied by a deleted record. This space may be too small or too large for the new record that is inserted.
2) There is no space for the records to grow larger. If a variable-length record becomes larger, it must be moved, and movement is costly if the record contains garbage and is pointed to by another record. Such a record is said to be pinned.

Thus, the basic-byte representation technique is not used for implementing variable-length records. However a modified form of the byte string representation called the slotted page structure is commonly used for implementing variable-length records.

0	Kakrail	A-102	400	A-201	900	A-218	700	⊥
1	Bashundhara	A-305	350	⊥				
2	Rampura	A-215	700	⊥				
3	Motijheel	A-101	500	A-110	600	⊥		
4	Dhanmondi	A-222	700	⊥				
5	Shantinagar	A-217	750	⊥				

Fig 8.2: Variable Length Records

8.4 Slotted-page structure

There is a header at the beginning of each block, containing the following information:

1. The number of record entries in the header
2. The end of free space in the block
3. An array whose entries contain the location and size of each record

The actual records are allocated contiguously in the block, starting from the end of the block. The free space in the block is contiguous between the final entry in the header array and the first record.

If a record is inserted, space is allocated for it at the end of free space and an entry containing its size and location is added to the header.

If a record is deleted, the space that it occupies is freed and its entry is set to deleted (its size is set to -1, for example). Further, the records in the block before the deleted record are moved so that the free space created by the deletion is occupied and all free space is again between the final entry in the header array and the first record.

The end-of-free space pointer in the header is appropriately updated as well. Records can be grown or shrunk using similar techniques as long there is space in the block

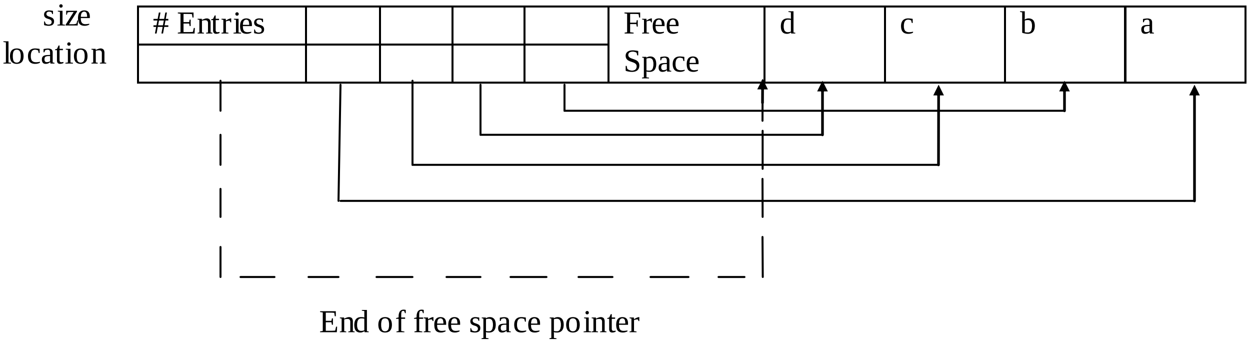


Fig 8.3 Slotted Page Structure

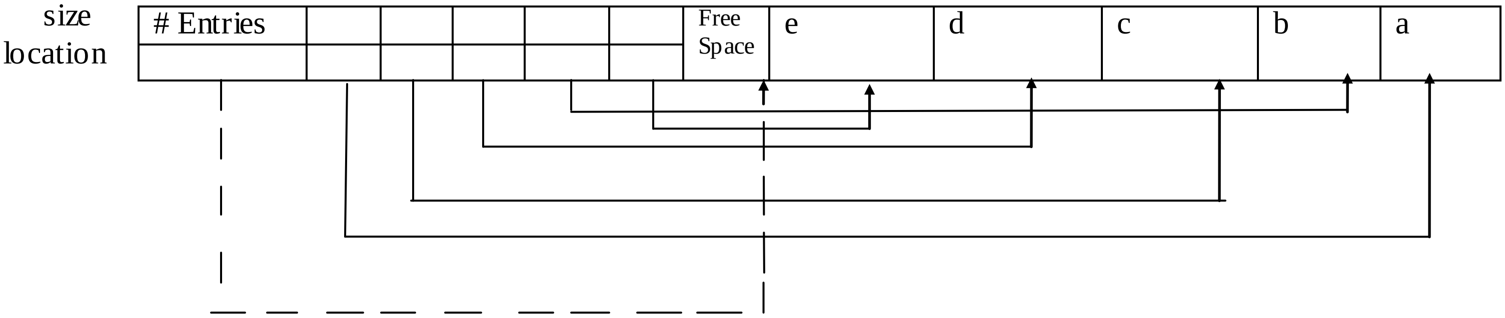


Fig 8.4: Insertion of record e in slotted page structure

In Fig 8.3, deletion of record c will result in the following figure:

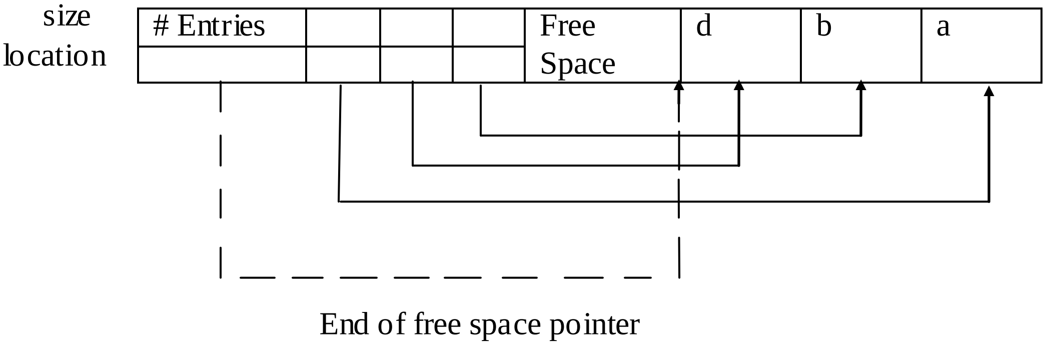


Fig 8.5: Deletion of record c in slotted page structure of Fig 8.3

8.5 Types of Record Organizations

So far we have discussed record length types. Now we will talk about how records are organized within a file. There may be one relation or groups of relations within a file. We will discuss this shortly.

i) Heap File Organization:

In this organization there is one relation per file and the order of records within that file does not matter.

ii) Sequential File Organization

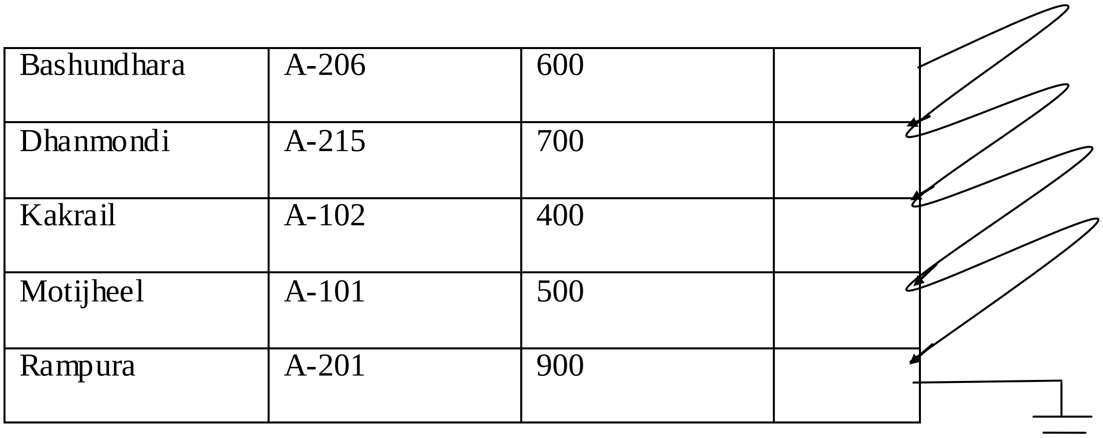
In the above file organization, there is still one relation per file but the order of records within the file does matter and this is maintained in a sorted order based on the value of a search key. We will discuss around this topic only for the rest of the chapter.

iii) Hash File Organization

The hash function in this type of organization is based on an attribute or a set of attributes. The result of hash function orders the records in an indexed method and specifies in which block of the file they should be placed.

8.6 Sequential File Organization

In sequential file organization, records follow order based on the value of a search key which maybe the value of an attribute or a set of attributes. The search key order of the records is maintained using pointers. An example of this is shown below from the banking enterprise organization.



Bashundhara	A-206	600	
Dhanmondi	A-215	700	
Kakrail	A-102	400	
Motijheel	A-101	500	
Rampura	A-201	900	

Fig 8.6: Sequential file for account records

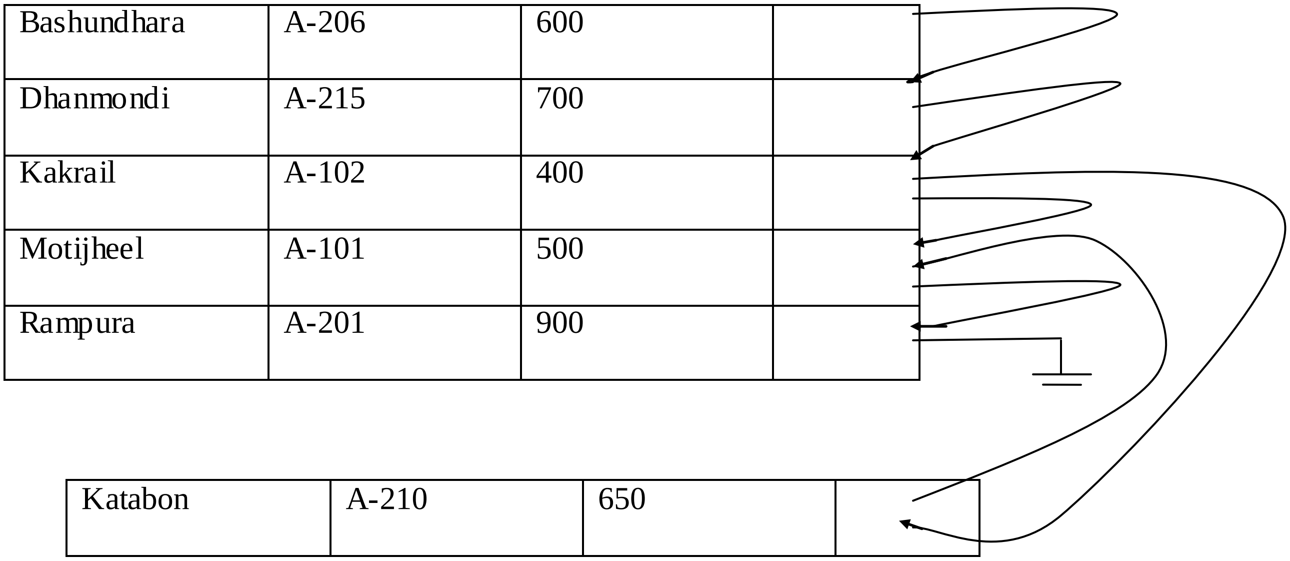
When insertions and deltions are done on the relation, reorganizations of the records based on the search key beome necessary which may be costly. A record can be inserted in the following way:

i) We need to find the record before in the file that comes before the record to be inserted in search key order.

ii) If the record is free, we insert the new record there. Otherwise we insert the new record in an overflow block.

iii) In either case, the records need to be chained together by pointers in search key order.

We show an example of insertion in the above way which is shown below:



Bashundhara	A-206	600	
Dhanmondi	A-215	700	
Kakrail	A-102	400	
Motijheel	A-101	500	
Rampura	A-201	900	

Katabon	A-210	650	

Fig 8.7: Sequential file after an insertion

If relatively a few records need to be inserted in overflow blocks, this approach works well otherwise the method of sequential approach may not be efficient.

As we mentioned earlier, reorganizations of file are costly and so must be carried out when the system load is low. In the extreme cases, when insertions rarely occur, reorganizations are minimal and maintaining sequential ordering of records is easy and what more, the pointer field may be eliminated.

8.7 Multitable Clustering File Organization

In sequential file organization, one file is stored per relation as we have seen. This is suitable for low-cost database systems and can take full advantage of the file system that the operating system provides.

However many large-scale database systems do not rely directly on the underlying operating system for file management. Instead one large operating-system is allocated to the database system. The database system stores all relations in this file and manages the file itself.

Consider the following query for the bank database:

SELECT account_number, customer_name, customer_street, customer_city FROM deposit, customer WHERE deposit.customer_name = customer.customer_name

This query computes a join of deposit and customer relations. In the worst case, each record will reside on a different block, forcing us to do one block read required by the query. Deposit and Customer relations as well as deposit ∞ customer join relation are shown below:

Table 8.1: The deposit relation

customer_name	account_number
Kamal	A-105
Kamal	A-220
Kamal	A-300
Shammi	A-301

Table 8.2: The Customer relation

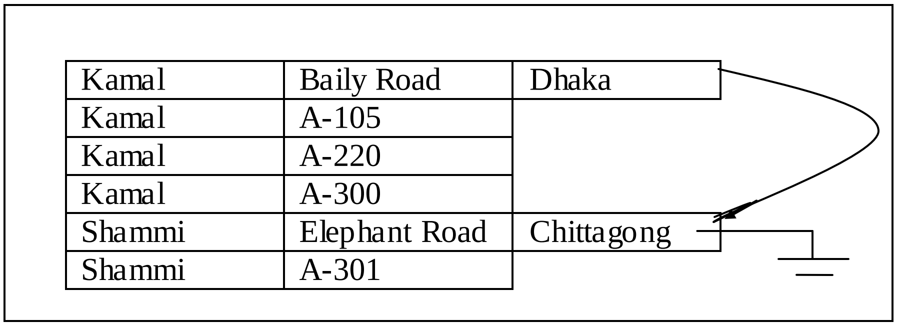
customer_name	customer_street	customer_city
Kamal	Baily Road	Dhaka
Shammi	Elephant Road	Chittagong

Table 8.3: Multitable clustering file structure

Kamal	Baily Road	Dhaka
Kamal	A-105	
Kamal	A-220	
Kamal	A-300	
Shammi	Elephant Road	Chittagong
Shammi	A-301	

The file structure for the join mixes together tuples of two relations but allows for efficient processing of the join. When a tuple of the customer relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding deposit tuples are stored on the disk near the customer tuple, the block containing the customer tuple contains tuples of the deposit relation needed for efficient processing of the query. Thus a multitable clustering file organization stores records of two or more relations in each block allowing us to read records that would satisfy the join by using one block read and process the query efficiently.

Under the above scheme, processing of some queries may become slow. For example, if we were to find all customer records, each record would be located in a distinct block. So in order to locate all tuples of the customer relation, some additional structure such as chaining all the records of the relation using pointers may be used as shown below. Careful use of multitable clustering produces significant performance gain in query processing.



Kamal	Baily Road	Dhaka
Kamal	A-105	
Kamal	A-220	
Kamal	A-300	
Shammi	Elephant Road	Chittagong
Shammi	A-301	

Fig 8.8 : Multitable clustering file structure with pointer chains

CHAPTER 9

DATA-DICTIONARY STORAGE

A relational database system needs to maintain data about the relations, such as the schema of the relations. This information is called the data dictionary, or system catalog. Among the types of information the system must store are these:

- Names of the relations
- Names of the attributes of each relation
- Domains and lengths of attributes
- Names of views defined on the database and definition of those views
- Integrity constraints

In addition, many systems keep the following data on users of the system:

- Names of authorized users
- Accounting Information about users

Further, statistical and descriptive data about relations may be kept.

- Number of tuples in each relation
- Method of storage used for each relation

There is also a need to store information about each index on each of the relations:

- Name of the index
- Name of the relation being indexed
- Attributes on which the index is defined
- Type of index formed

CHAPTER 10

INDEXING

10.1 Basic Concepts

Just as words or phrases in a text book index appear in a sorted order, an index for a file in a database works in a similar way. Because the words in the book index are sorted, it is easy to find the word we are looking for. Also, the index is smaller than the book, making it easier to find the words we are looking for.

Database system indices work in a similar way as book indices in libraries. As an example, if we want to retrieve an account record corresponding to an account number, the datatabase system would look up an index to find on which disk block the corresponding record is stored and then fetch the disk block to get the record.

However a sorted list of account numbers would not work well on very large databases with millions of records as the index would become very big. This gives rise to the need for more sophisticated indexing techniques which we discuss below.

There are two kinds of indices:

i) Ordered indices: Based on a sorted ordering of the values.
ii) Hash indices: Based on a uniform distribution of values in buckets. The bucket to which a value is assigned is distributed by a function, called a hash function.

In this chapter our focus will only be on ordered indices.

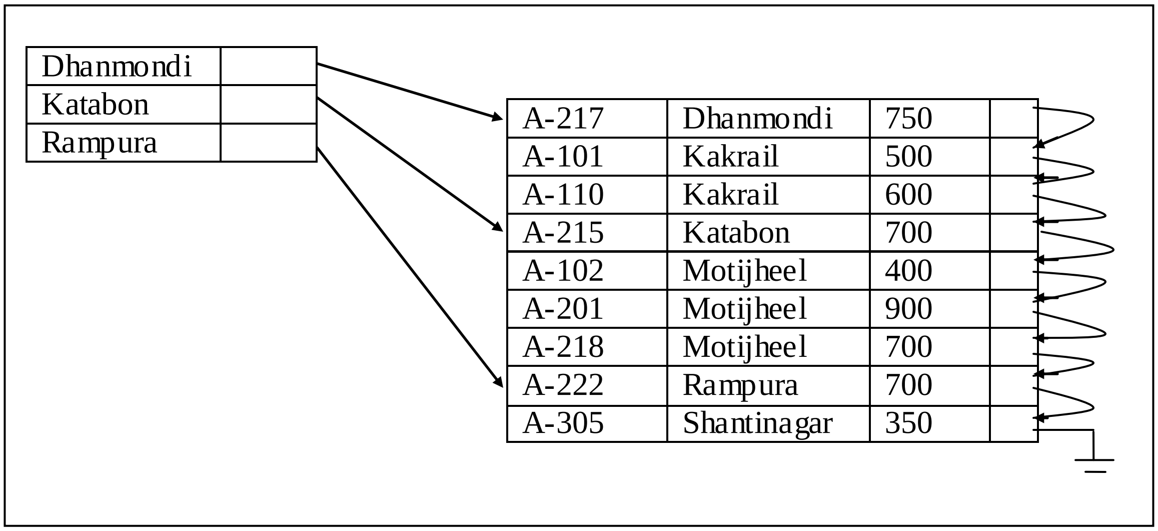
No one technique for ordered indices is the best for database applications. These techniques must be evaluated on the basis of the following factors:

i) Access types: The access types must be efficient. These include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
ii) Access time: The time it takes to find a particular data item or set of items.
iii) Insertion time: The time it takes to insert a new data item including the time it takes to update the index structure.
iv) Deletion time: The time it takes to delete a data item including time to update the index structure.
v) Space overhead: The additional space occupied by an index structure which is preferred to be moderate for performance reasons.

10.2 Ordered Indices

We can use an index structure in order to gain fast retrieval of records in a file. Each index structure is associated with a particular search key. The records in the indexed file are sorted. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a clustering index is an index whose search key also defines the sequential order of the file. Clustering indices are also called primary indices. Indices whose search key defines an order different from the sequential order of the file are called nonclustering indices or secondary indices.

The following figure shows a sequential file of account records taken from our banking example. In this figure, the records are stored in search key order, with branch name used as the search key.



Dhanmondi	
Katabon	
Rampura	

A-217	Dhanmondi	750	
A-101	Kakrail	500	
A-110	Kakrail	600	
A-215	Katabon	700	
A-102	Motijheel	400	
A-201	Motijheel	900	
A-218	Motijheel	700	
A-222	Rampura	700	
A-305	Shantinagar	350	

Fig 10.1: Sequential file for account records

10.2.1 Dense and Sparse Indices

An index record or index entry consists of a search-key value and pointers to one or more records with that value as their search key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify within the block.

There are two types of ordered indices:

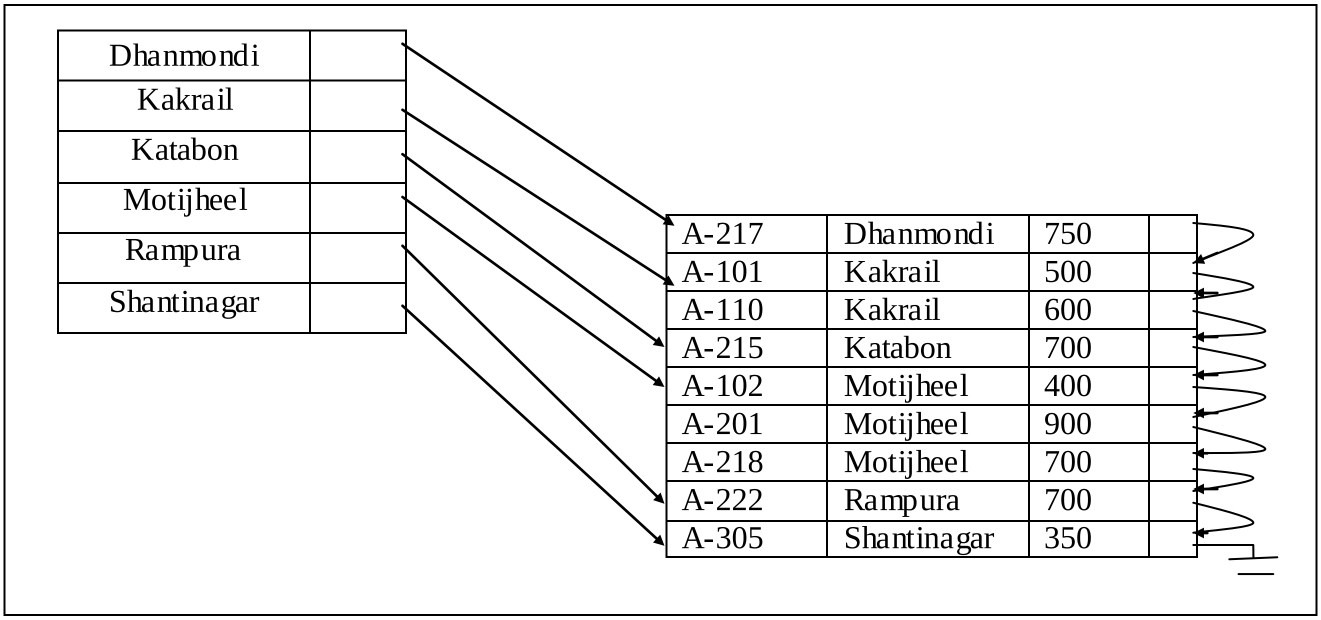
i) Dense Index: An index record appears for every search key value in the file. In a dense clustering record, the index record contains the search key value and a pointer to the first data record with that search key value.

ii) Sparse Index: An index record appears for only some of the search key values. Like dense indices, each sparse index record contains a search key value and a pointer to the first data record with that search key value.

The following figure shows dense index for the account file. Suppose we are looking records for Motijheel branch.

Using the dense index, we follow the pointer in that record to locate the next record in search key (branch_name) order. We continue processing records until we encounter a record for a branch other than Motijheel.

In the next figure showing sparse index, we do not find an index entry for Motijheel. Since the last entry in alphabetic order before ‘Motijheel’ is ‘Katabon’, we follow the pointer and then read the account file in sequential order until we find the first Motijheel record and begin processing at that point.

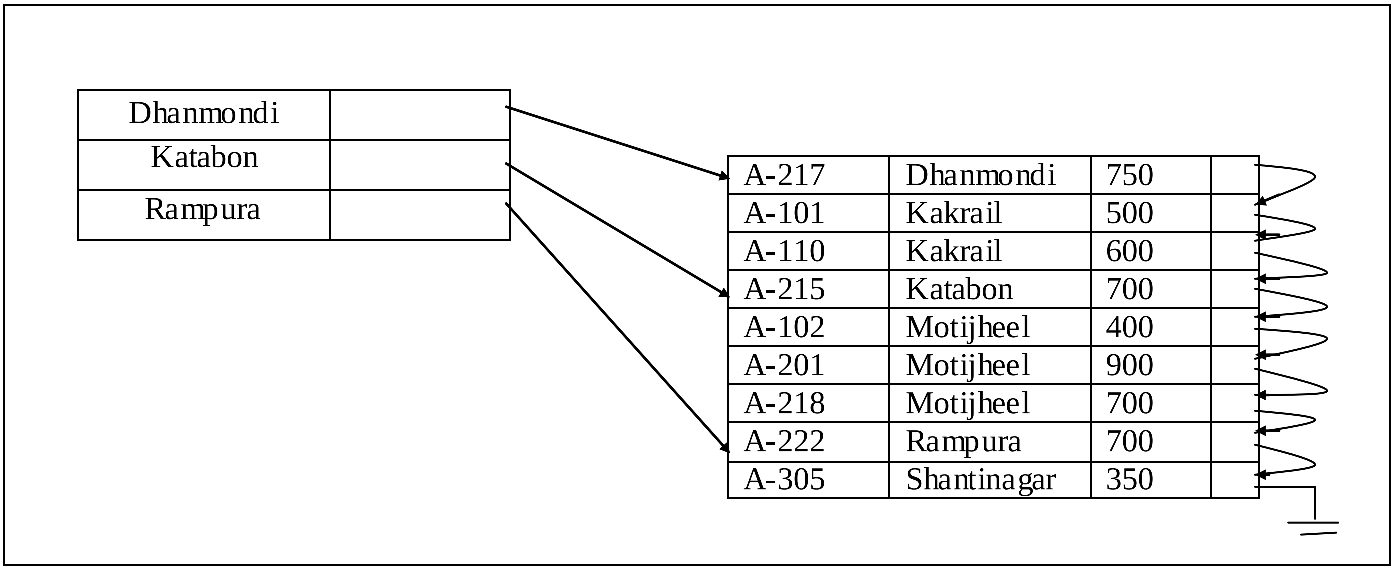


Dhanmondi	
Kakrail	
Katabon	
Motijheel	
Rampura	
Shantinagar	

A-217	Dhanmondi	750	
A-101	Kakrail	500	
A-110	Kakrail	600	
A-215	Katabon	700	
A-102	Motijheel	400	
A-201	Motijheel	900	
A-218	Motijheel	700	
A-222	Rampura	700	
A-305	Shantinagar	350	

Fig 10.2: Dense index

As we have seen, it is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.



Dhanmondi	
Katabon	
Rampura	

A-217	Dhanmondi	750	
A-101	Kakrail	500	
A-110	Kakrail	600	
A-215	Katabon	700	
A-102	Motijheel	400	
A-201	Motijheel	900	
A-218	Motijheel	700	
A-222	Rampura	700	
A-305	Shantinagar	350	

Fig 10.3: Sparse Index

10.2.2 Multilevel Indices

Even if we use a sparse index, it may become too large, and the process of searching a large index may be inefficient and costly. To deal with this problem, we construct a sparse index on the clustering index as shown below. To locate a record, we first use binary search on the outer index to find the record for the largest search key value less than or equal to the one we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has largest search key value less than or equal to the one we desire. The pointer in this record points to the block of the file that contains the desired record.

Using the two levels of indexing, we have read only one index block, rather than the seven we read with binary search, provided that the outer index is already in main memory.

If our file is very large, we can use multilevel indices i.e, indices with two or more levels. Searching for records with a multilevel index requires significantly fewer I/O (input/output) operations than does searching by binary search.

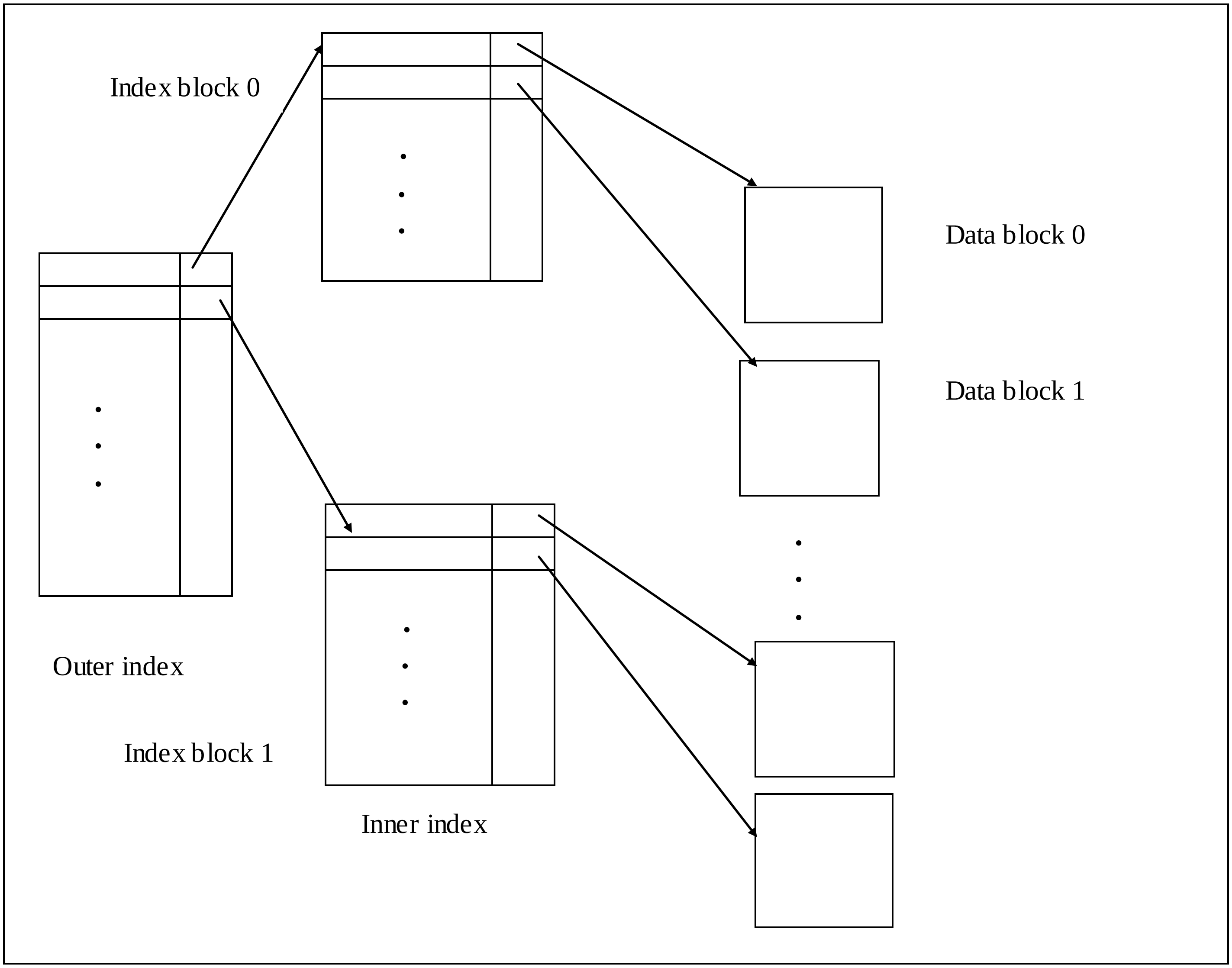


Fig 10.4: Two-level sparse index

10.2.3 Index Updates

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file. We now describe processes for updating single level indices.

- Insertion: First, the system performs a lookup using the search key value that appears in the record to be inserted. The action the system takes next depends on whether the index is dense or sparse.

- Dense indices:

  1) If the search key value does not appear in the index, the system inserts an index record with the search key value in the index at the appropriate position.

  2) Otherwise, the following actions are taken
     a) If the index stores pointers to all records with the same search key value, the system adds a pointer to the new record to the index record.
     b) Otherwise, the index record stores a pointer to only the first record with the search key value. The system then places the record to be inserted after the other records with same search key values.

- Sparse Indices: We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search key value (in search key order) appearing in the new block into the index. On the other hand, if the new record has the least search key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

- Deletion: To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse.

  - Dense indices:

    1) If the deleted record was the only record with its particular search key value, then the system deletes the corresponding index record from the index.
    2) Otherwise the following actions are taken:
       a) If the index record stores pointers to all records with the same search key value, the system deletes the pointer to the deleted record from the index record.
       b) Otherwise, the index record stores a pointer to only the first record with the search key value. In this case, if the deleted record was the first record with the search key value, the system

updates the index record with the search key value to point to the next record.

- Sparse indices:

1) If the index does not contain an index record with the search key value of the deleted record, nothing needs to be done to the index.
2) Otherwise the system takes the following actions:

a) If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search key value in search key order. If the next search key value already has an index entry, the entry is deleted instead of being replaced.

b) Otherwise, if the index record for the search key value points to the record being deleted, the system updates the index record to point to the next record with the same search key value.

10.2.4 Secondary Indices

A secondary index on a candidate key looks just like a dense clustering index except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from clustering indices. If the search key of a clustering index is not a candidate key, it is still ok if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.

In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search key value. The remaining records with the same search key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file. The following figure shows the structure of a secondary index that uses an extra level of indirection on the ‘account’ file, on the search key balance.

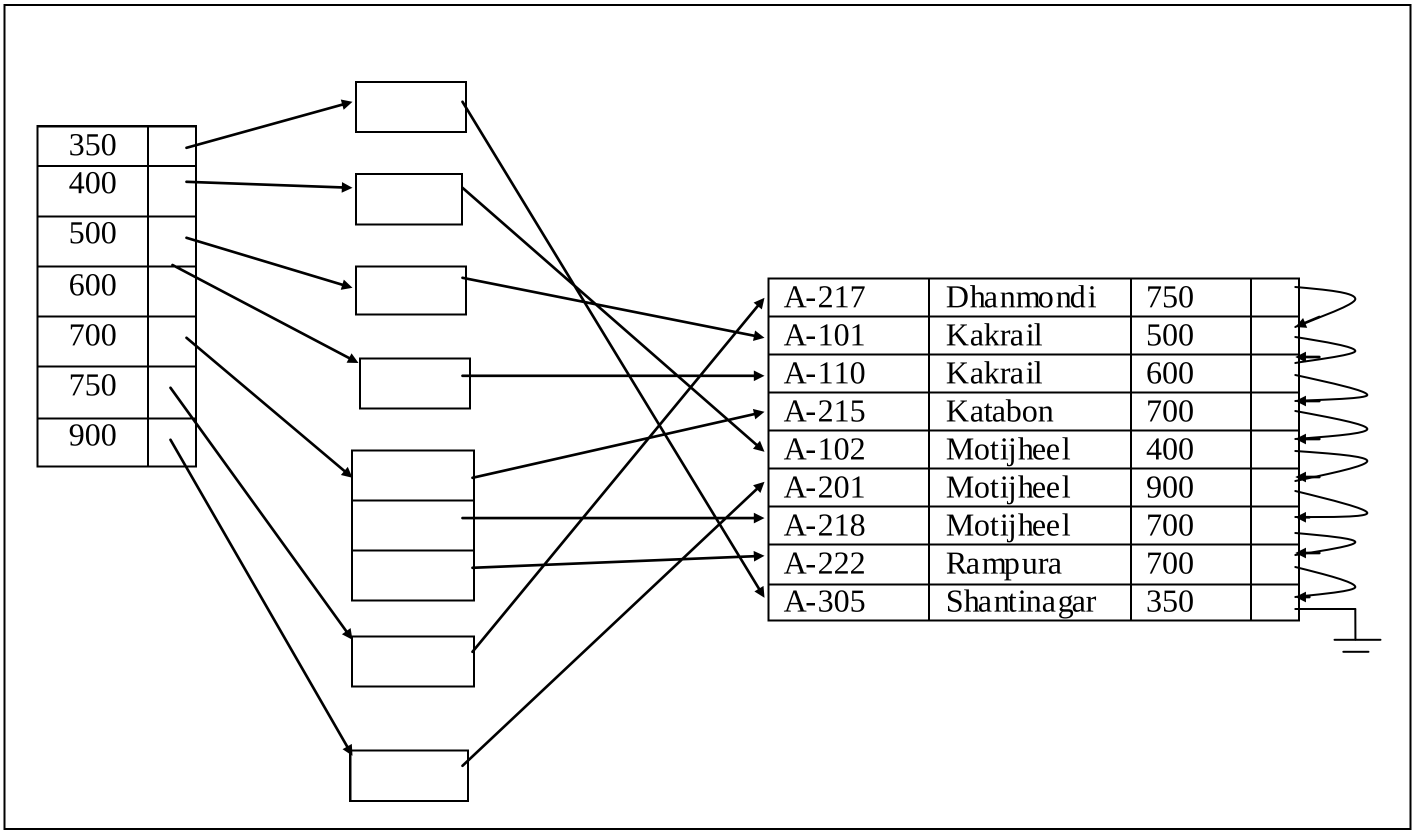


Fig 10.5: Secondary index on account file on noncandidate key balance

10.3 B$^+$-Tree Index Files

The main disadvantage of the index-sequential file organization is that performance is poor as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by disorganization of the file, frequent reorganizations are not well off.

The **B$^+$-** tree index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B$^+$- tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length. The B in B$^+$-tree stands for 'balanced'. Each nonleaf node in the tree has between $\lceil n/2 \rceil$ (ceiling of n/2) and n children, where n is fixed for a particular tree.

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

Fig 10.6: Typical node of a B$^+$-tree

10.3.1 Structure of a B^+-Tree

A B^+-Tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. The following figure shows a typical node of a B^+-Tree**.** It contains up to n-1 search key values K1, K2,……..Kn-1 and n pointers P1, P2, ……Pn. The search-key values within a node are kept in sorted order; thus, if i<j, Ki<Kj.

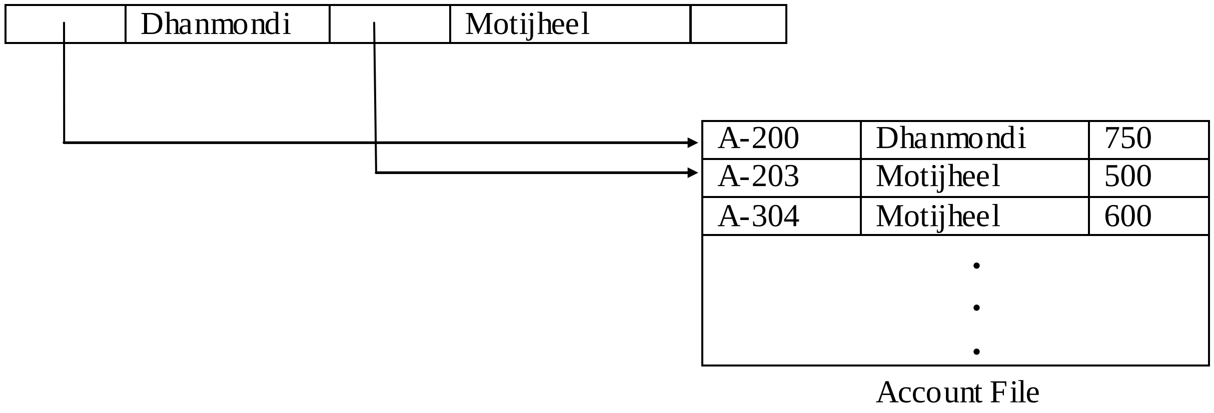


Fig 10.7: A leaf node for account B^+-Tree index (n=3)

We consider first the structure of the leaf nodes. For i=1, 2, ………., n-1, pointer Pi points to either a file record with search key value Ki or to a bucket of pointers, each of which points to a file record with search key value Ki. The bucket structure is used only if the search key does not form a candidate key, and if the file is not sorted in the search-key value order.

The above figure shows one leaf node of a B^+-Tree for the account file, in which we have chosen n to be 3, and the search key is branch_name. Note that, since the account file is ordered by branch_name, the pointers in the leaf node point directly to the file.

Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to n-1 values. We allow leaf nodes to contain at least $\lceil (n-1)/2 \rceil$ values. The ranges of values in each leaf do not overlap. Thus if Li and Lj are leaf nodes and i < j, then every search key value in Li is less than every search key value in Lj. If the B^+-Tree index is to be a dense index, every search-key value must appear in some leaf node.

Now we can explain the use of the pointer Pn. Since there is a linear order on the leaves based on the search key values that they contain, we use Pn to chain together the leaf nodes in search key order. This ordering allows for efficient sequential processing of the file.

The nonleaf nodes of the B^+-Tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers, and must hold at least n/ 2 pointers. The number of pointers in a node is called the fanout of the node.

Let us consider a node containing m pointers. For i=2,3……..m-1, pointer Pi points to the subtree that contains search key values less than Ki and greater than or equal to Ki-1. Pointer Pm points to the part of the subtree that contains those key values greater than or equal to Km-1, and pointer P1 points to the part of subtree that contains those search key values less than K1.

Unlike other nonleaf nodes, the root node can hold fewer than $\lceil n/2 \rceil$ pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible construct a B$^+$-Tree, for any n, that satisfies the preceding requirements. The following figure shows a complete B$^+$-Tree for the account file (n=3). For simplicity, we have omitted both the pointers to the file itself and the null pointers.

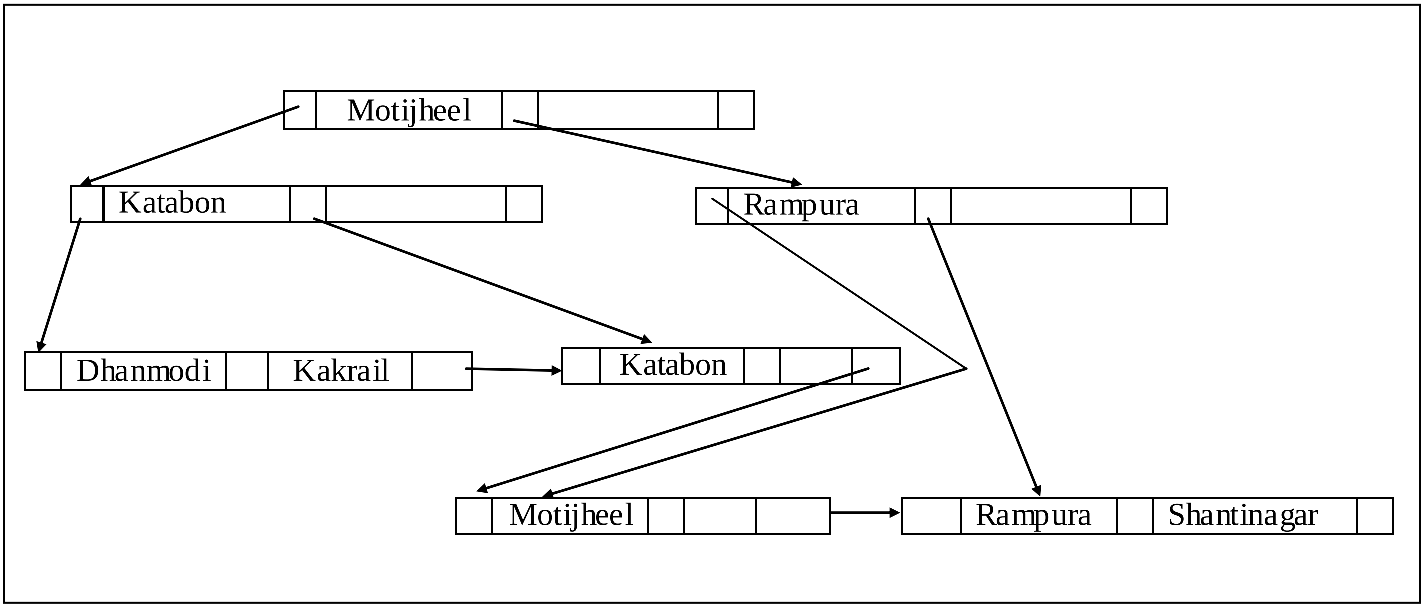


Fig 10.8: B+ tree for account file (n=3)

It may be mentioned that the root of above B+ tree structure corresponds to the branch name with the highest balance. The branch names within a leaf or tree node follow sorted (alphabetical) order.

CHAPTER 11

TRANSACTIONS

A transaction is a sequence of data access operations that transfers the database from one consistent state to another consistent state (where the new state is not necessarily different from the old).

Examples of transactions:

- Compile a list of all students that take the class on CSE 303.
- Set the grade 3.0 in the class on CSE 304 for the student with matNr 003
- Give all TAs a 10% pay rise.

A transaction can be an entire program, a part of a program or a single command.

Transaction processing systems are systems with large databases and a large number of users who execute transactions in parallel. Examples include travel reservation systems (hostels, flights), credit card processing, stock markets and supermarket checkout.

A transaction can terminate successfully or reach some state where it is discovered that an error has occurred. In the first case, the transaction is said to be committed; in the second case it is aborted. An aborted transaction must be undone (any changes it has performed must be reversed) to go back to the previous consistent state. The undo is also known as a "roll back".

11.1 Syntax for Transactions

```
BEGIN_TRANSACTION
    Instructions
    ...
if everything ok then COMMIT
else ABORT
END_OF_TRANSACTION
```

Some systems do not provide a begin_transaction statement. Instead, the first SQL command that is executed after the end of a transaction automatically starts a new transaction.

Important ACID features of transactions that must be provided by a DBMS are:

- Atomicity: Either all operations of a transaction are executed or none at all => If the transaction is interrupted by some error, all intermediate changes must be reversed.
- Consistency: A transaction must transform the database from a consistent state to another consistent state.
- Isolation: Only after successful transaction, partial results of a transaction may be released for usage by other transactions => Transactions act independently of each other.
- Durability: After the successful termination of a transaction, its results are persistent i.e., they can only be changed or undone by a new transaction, even if errors or failures occur.

Reason for Isolation:
Assumption:
Transactions A and B are executed concurrently. A uses a partial result of B and then terminates successfully. After this, B is aborted => the partial result must be changed back.

⇨ A would be need to be changed as well (-> cascading aborts)
⇨ Contradiction to Durability

Note: If it is discovered after a commit that the effects of a transaction must be undone, a compensating transaction is necessary to do this.

Implementation of ACID properties:

- Atomicity: Recovery Manager rolls back in case of error.
- Consistency: Transaction developers must program correctly; integrity constraints may be automatically checked by the DBMS; recovery manager treats execution interruptions.
- Isolation: ensured by concurrency control mechanisms. (more about this in chapter 13)
- Durability: Recovery manager ensures durability after errors (redo if necessary).

CHAPTER 12

RECOVERY IN TRANSACTIONS

Goal:
Preserve correctness and consistency of data over time, allowing for parallel access (transactions) of multiple users and occurring errors.

12.1 Problem Sources

Potential reasons for transaction failure

Software error:

- in an application program
- in the operating system
- in the database system

Hardware error:

- disk error or damage
- CPU error
- bus error
- power failure or fluctuation

Operator error (e.g. mounted wrong tape, wrong file system)

Database Sabotage

Recovery is supposed to restore a state of the database that is known to be correct, after an occurred or presumed error.

12.2 Logging and Recovery from Software Failures

Reading and writing data

Reading and writing data items from and to a database are not immediate, atomic operations but actions consisting of several steps:

Reading a data item from the database requires the following steps:

1. Find the address of the disk block that contains the data item.
2. Copy this disk block into a buffer in memory (if it is not there

already).

3. Copy the value of the data item into a variable in the application program.

Writing a data item to the database requires the following steps:

1. Find the address of the disk block that contains the old version of the data item or the address of the disk block where the data item is newly entered.
2. Copy this disk block into a buffer in memory (if it is not there already)
3. Copy the value of the data item from an application variable into the correct location in the block in the buffer.
4. Copy the updated block from the buffer back to disk. This may be done immediately or later.

Log Files

Changes to the data are written to a log file to record what has been done by transactions. This will be used during recovery.

The log file contains transaction records consisting of:

- transaction identifiers
- type of log record (begin_transaction, insert, update, delete, commit, abort)
- identifier of the data item affected by the database access (insert, delete, update)
- before-image of the data item (value before the change, in case of delete or update; in case of an insert, there is no before-image)
- after-image of the data item (value after the change, in case of insert or update; in case of a delete, there is no after-image)
- log management information, such as pointer to previous and next log records for that transaction.

Recovery Approaches

Typical approach for recovery:

1. Periodically, the entire database is copied (backedup), typically to tape or optical disks. Backups also can be made incrementally i.e. only the changes since the last backup are stored.

   A log file is used to record the changes in the database at each update, typically the old and new values of the updated attribute(s).

2. When an error occurs:

a) If the database itself is damaged (media failure, e.g. by a head crash) =>

-The last backup is loaded

-The updates of all committed transactions that were executed since the time of last backup are shown by the log ('redo'). Those transactions that had not committed yet, need to be restarted.

b) If the database is itself is not damaged, but the correctness of the contents cannot be guaranteed (e.g. because a program with updates crashed)

=>

-The log is used for repeating committed changes (redo) and reversing non-committed changes (undo). Then, the non-committed transactions can be restarted.

CHAPTER 13

CONCURRENCY CONTROL

13.1 Problems Caused by Violation of Isolation

If two or more transactions using the same data items are executed in parallel, problems may occur:

Example 1:
Two people own two ATM cards for the same bank account. Both people withdraw money from the account at the same time at two different teller machines.

In short, two transactions A and B run concurrently:

Start transaction A	Start transaction B
A reads a copy of the account balance: copy1 = Tk1000	B reads a copy of the account balance: copy2 = Tk1000
A withdraws Tk200 copy1 := 1000 - 200 = 800	B withdraws Tk400 copy2 := 1000 - 400 = 600
A writes Tk800 to the database as the new balance	B writes Tk600 to the database as the new balance

Evaluation

- It is not predictable whether the new balance will be 800 or 600.
- Both alternatives are false! The correct value should be 400.

This is called the "lost update problem".
The consistency property of transactions is violated here.

Example 2

Start transaction A	
A reads a copy of the account balance: copy1 = Tk1000	
A withdraws Tk200 copy1 := 1000 - 200 = Tk800	
A writes Tk800 to the database as the new balance	Start transaction B
	B reads a copy of the account balance: copy2 = Tk800
	B withdraws Tk600 copy2 := 800 - 600 = Tk200
A aborts and rolls back.	B writes Tk200 to the database as the new balance
	B commits.

This is also called a "dirty read", a "temporary update" or an "uncommitted dependency problem".

The problems in the examples can occur if the isolation property of transactions is violated.

13.2 Serializability

If transactions are programmed correctly, they start on a consistent state and leave the database in a consistent state.

Definitions:

A **schedule** S of transactions T1, ..., Tn is an ordering of the operations of the transactions under the constraint that for each transaction Ti that participates in S, the operations of Ti must appear in the same order in S as in Ti. This means, the operations of each transaction appear in their correct order, but they may be interleaved with the operations of other transactions.

A **serial schedule** is a schedule in which the operations of the participating transactions are not interleaved, i.e. the transactions are executed one after the other.

A schedule is called **serializable** if its effects are the same as the effect of some serial schedule.

The above could also be phrased as: Two transactions are called serializable if *every* parallel execution delivers the same result as *some* serial execution.

For a set of transactions, there may be several serializable schedules that have different outcomes, depending on the order of the operations. This is okay, since we consider here transactions that are executed in parallel, meaning that we do not care which one comes first in the end.

Example 1:
Two transactions reserving a room in the same hotel run in parallel. Both transactions read the list of available rooms. If the system ensures serializable schedules, data consistency is guaranteed; however, it is unknown which transaction will get which room. The first to access the list of rooms will probably reserve the next room on the list. In this example, it probably does not matter which client gets which room; the system decides who comes first.

Example 2:
Two transactions both access a bank account, one makes a deposit, the other withdraws some cash from it. If the bank does not care how high or low the account balance is, this can be run concurrently. However, if the customer is at the lowest borderline of the allowed negative balance, the cash withdrawal may not be granted by the bank before the deposit is made. Therefore, these transactions should not be run in parallel, but the withdrawal transaction should be started after the deposit transaction has completed.

Background of concurrency control mechanisms:

Since serial executions are correct, so are serializable ones.

The algorithms for concurrency control provide ways of recognizing serializable schedules and disallowing non-serializable schedules. However, none of them recognize all situations where serializability is given because it is generally not easy to determine serializability of two transactions. Thus, generally more transactions are aborted for concurrency control reasons than would really be necessary.

For serializability, the order of read and write operations is relevant:
• If two transactions only read the same data item, they do not conflict and the order of interleaved operations is not relevant.
• If two transactions either read or write completely separate data items, they do not conflict and the order of interleaved operations is not relevant.
• If one transaction writes a data item, and another either reads or writes the same data item, the order of execution is relevant for correctness.

13.3 Some Synchronization Protocols

The conflicting operations in a schedule can be arranged in a serializable way based on the principles of synchronization protocols. Here we discuss some of them:

13.3.1 No parallel access

Only serial transactions are executed in the system. This is usually a waste of capacity and user time. However, serial executions of transactions is always correct, since every transaction leaves the database in a consistent state.

13.3.2 Optimistic Protocols

If we assume conflicts occur rarely it is more efficient to restart a transaction in case of a conflict rather than to prevent from happening conflicts in the first place. A transaction is always executed, but only on copies of data items. In the end, a check is performed on the original data item to determine if a conflict has occurred.
If so, the transaction is started anew otherwise the results of the transactrion are written to the disk. This concept can be summarized in the following way:

Begin_Transaction -> read and update phase -> validation phase -> write phase -> End of Transaction

The above protocol is good for situations with a few conflicts. If there are too many conflicts, the overhead is immense. Many transactions have to be restarted. The more transactions are restarted, the higher the number of conflicts; hence even more transactions are restarted and hence, even more conflicts happen and the cycle continues.

13.3.3 Synchronization with Timestamps

Every transaction receives a timestamp when it is started. Every data item receives a read and write timestamps that mark most recent read and write accesses respectively. A transaction may execute an operation on a data item if the timestamps indicate serializability. Otherwise the transacition is aborted and restarted. An upside to this process is that no central data structure is necessary and distributed databases are good enough. A downside is not all serializable transactions are recognized and there may be redundant restarts.

13.3.4 Synchronization by Locking

This is the most popular synchronization protocol. Locking is normally hidden from application programmers and performed by a resource manager in the system. One of its important roles is setting and releasing locks

The DBMS manager manages a central table recording every data item currently in use by a transaction. It stores locking modes and a list of lock requests corresponding to data items.

Locking modes:
• exclusive lock (write lock, X lock) for changing an item
• shared lock (read lock, S lock) for reading an item

Example of a lock table:

Data Item	**Locks being held**	**Waiting lock requests**
X	T1:S, T2:S	T3:X
Y	T2:X	T4:S, T1:S
Z	T1:S	

Locking tables are held in main memory. This is because in a system with many transactions, lock and unlock operations are frequent and therefore must operate speedily.

Access to data items is granted to transactions according to a compatibility table as shown:

		Locking mode of the data item		
		none	**S**	**X**
Lock request of the transaction	**S**	+	+	-
	X	+	-	-

+ : lock request granted

-: lock request denied

The operation of setting a lock must be implemented atomically, i.e. the setting of a lock may not be interrupted by another process also setting a lock. Otherwise two parallel processes might both detect that a data item is unlocked and acquire a write lock on it. A possible solution to retain this atomicity might be that the lock manager executes each lock or unlock operation completely before starting a new one. There is only one lock manager (1 process) so that interleaving setting and releasing locks with other processes cannot happen.

Phase Locking Protocol (2PL)

Locking single items only makes the access to this one item consistent. Generally, transactions consist of more than one data access. In order to ensure the serializability of schedules, the two-phase locking protocol is used. This method is based on the "fundamental locking theorem" in databases. It consists of five conditions in combination necessary for the serializability of the schedules.

1. Every data item that a transaction needs to access is locked before the access.
2. A transaction does not request a lock that it owns already.
3. A transaction must respect the locks of other transactions (according to a compatibility table).
4. Every transaction goes through two phases:
- a growth phase in which it requests locks but must not release any locks, and
- a shrink phase in which it releases its acquired locks, but may not request any
new locks.
5. A transaction must release all of its locks before End_of_Transaction.

C H A P T E R 14

ADVANCED DATABASES

As part of advanced databases, I include here distributed databases, data warehouses and multimedia databases, and introduce data mining and NoSQL.

14.1 Distributed Databases

A distributed database is a database with one common schema whose parts are physically distributed via a network. For a user, a distributed database appears like a central database i.e. it is invisible to users where each data item is actually located. However, the DBMS must periodically synchronize the scattered databases to make sure that they have all consistent data.

Advantages:

i) Reflects organizational structure: database fragments are located in the departments they relate to.

ii) Local autonomy: a department can control the data about them (As they are the ones familiar with it)

iii) Improved availability: a fault in one database system will affect one fragment instead of the entire database.

iv) Improved performance: data is located near the site of greatest demand, and the database systems themselves are parallelized, allowing load on the databases to be balanced among servers. (A high load on one module of the database wont affect other modules of the database in a distributed database)

v) Ergonomics: It costs less to create a network of smaller computers with the power of a single large computer.

vi) Modularity: Systems can be modified, added and removed from the distributed database without affecting other modules (systems).

14.1.1 Client-Server Architecture

Distributed database applications are being developed in the context of the client-server architecture. This section discusses client-server in the context of distributed processing. The term distributed processing means that distinct machines can be connected into communications network such that a single data processing task can span several machines in the network.

i) One simple case involves running the DBMS backend (the server) on one machine and the application front ends (the clients) on another. (Fig 14.1)

ii) Several different client machines access the same server machine. Thus, a single database might be shared among several distinct clients. (Fig 14.2)

The client machines might have stored data of their own, and the server might have applications of its own. Therefore, each machine will act as a server for some users and a client for others. A single client machine might be able to access several different machines. Such access can basically be achieved in the following two different ways:

iii) A given client might be able to access any number of servers, but only one at a time (i.e., each individual database request must be directed to just one server). In such a system it is not possible, within a single request, to combine data from two or different servers. Furthermore, the user in such a system has to know which particular machine holds which pieces of data.(Fig 14.3)

iv) The client might be able to access many servers simultaneously (i.e., a single database request might be able to combine data from several servers).

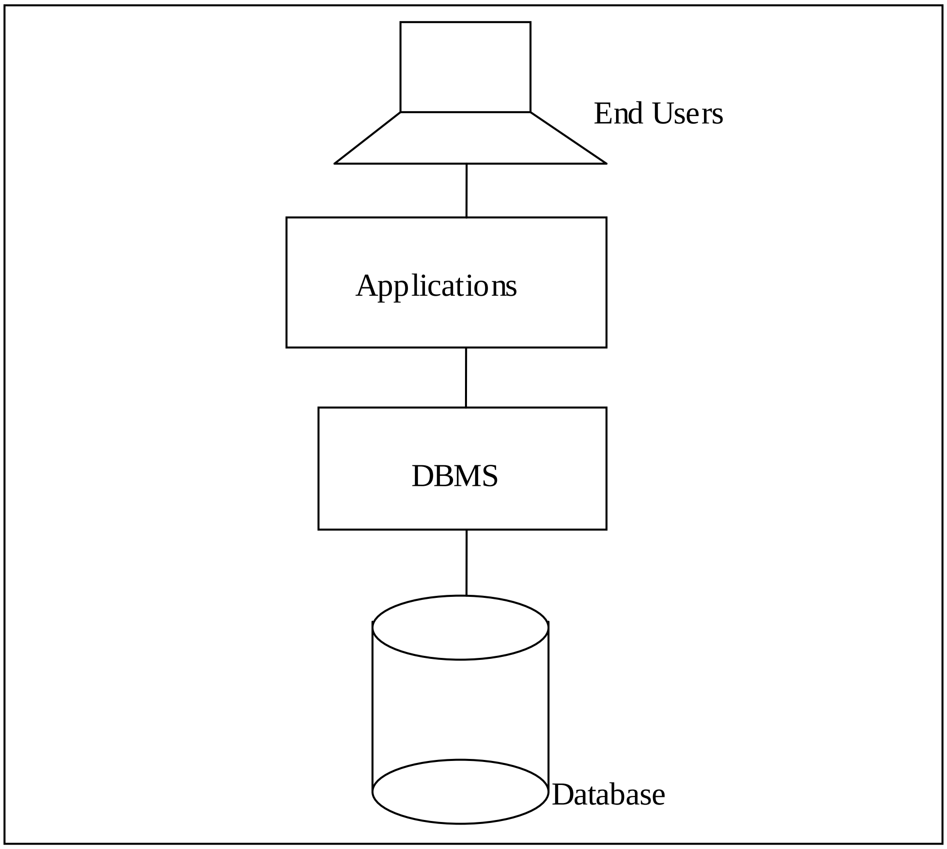


Fig 14.1: Basic Client-Server Architecture

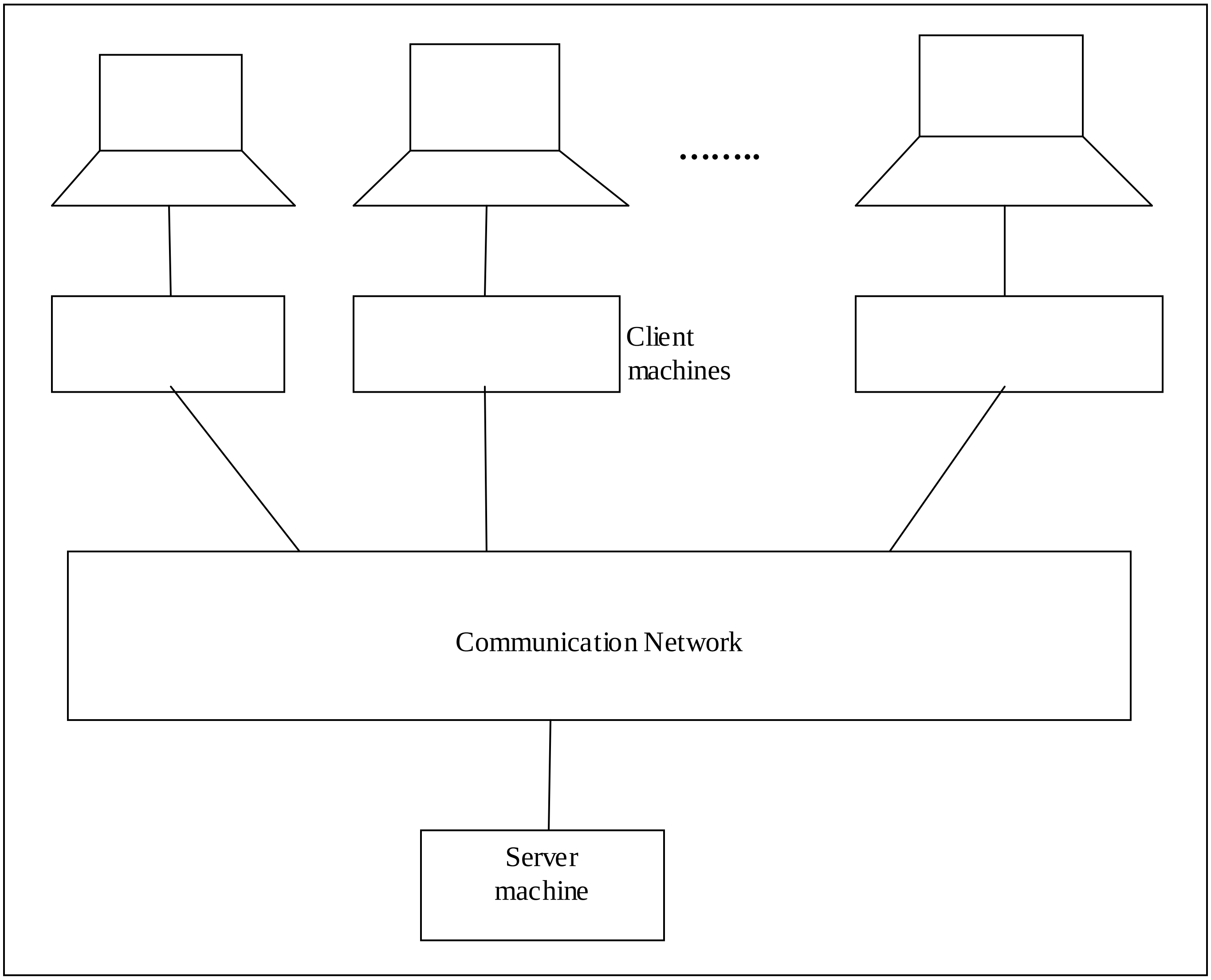


Fig 14.2: One server, many client machines

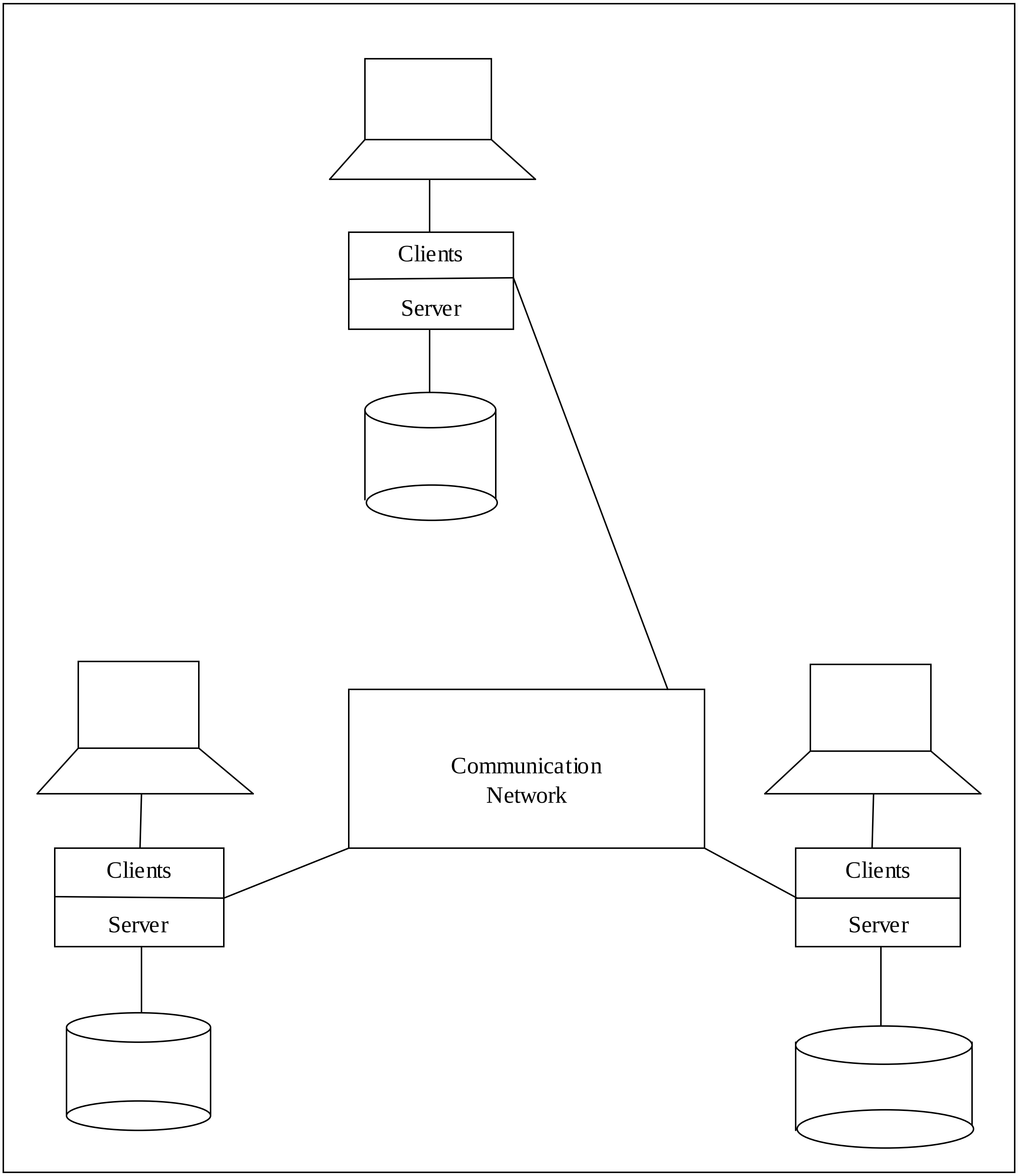


Fig 14.3: Each machine runs both client(s) and server

14.2 Data Warehouses

A data warehouse (DW) is a subject-oriented, integrated, non-volatile and time-variant collection of data in support of management's decisions. (Inmon's definition)

Explanation:

- Subject-oriented: The system focus is not on the applications required by the different departments of a company (e.g. econometrics and finance, medical research and biotechnology, data mining, engineering etc) but on subject areas those that relate to all departments like customers, products, profits etc. Traditional database systems are developed for the different applications and data warehouses for the subject areas.

- Integration: Data from various sources is represented in the data warehouse. Different sources often use different conventions in which their data is represented. It must be unified to be represented in a single format in the data warehouse. E.g., Application A uses "m" and "f" to denote gender. Application B uses "1" and "0" and application C uses "female" and "male". One of the conventions can be used for the data warehouse; others can be converted.

- Non-volatility: Data that have been migrated into the DW are not changed or deleted.

- Time-variance: Processing of DW data is stored in a way to allow comparisons of data loaded at different times (e.g. a company's profits of last year versus the profits of the year before that). DW is like a series of snapshots of the data of its different sources, taken at different times, over a long period of time (typically 5-10 years)

  The purpose of most databases is to present current, not historical data. Data in traditional databases is not always associated with a time whereas data in a DW always is.

Advantages:

i) Because DW is subject-oriented, it deals with subject areas like customers, products relating to all departments of a company but not to different applications relating to different departments.

ii) It converts non-homogeneous data to homogeneous data.

iii) Data do not require to be updated or deleted. It can store redundantly.

iv) It can present historical data over a period of 5-10 years. So it can be used for the purpose of analysis of data

14.2.1 Application Areas of Data Warehouses

Most application areas are in the business field.

- **Business**: anything to support business success by providing the necessary information about an enterprise.
- **E-business**: DWs can be used for information purposes in E-Commerce via the Internet, intranets and extranets. (Internet is accessible by anyone, intranet only by employees of a company, extranet includes also customers and suppliers by password.)
- **Science**: Scientific and empiric studies often use large amounts of data (instance measurements) Examples: climate and weather (meteorological data in TB sizes everyday).
- **Technical Applications**: Examples: water quality for environment control; production factory with a materials database to document ingredients or materials.

14.2.2 Specific Application Examples

- **WallMart (a large retailer):** Data Warehouse sized about 25 TB; 20,000 queries everyday, used for market basket analysis and customer classification.
- **EOS (Earth Observing System):** a project concerned with climate and environment research; it receives about 1.9 TB meteorological data everyday, analysing with data mining tools.

14.3 Multimedia Databases

Multimedia databases store multimedia such as images, audio and video. The database functionality becomes important when the number of multimedia objects stored is large.

Several issues have to be addressed if multimedia data are to be stored in a database.

1. The database must support large objects since multimedia data such as videos can occupy up to a few gigabytes of storage.
2. Similarity-based retrieval is needed in many multimedia database applications. For example, in a database that stores fingerprint images, a query fingerprint is provided and fingerprints in the database that are similar to the query finger print must be retrieved.
3. The retrieval of some types of data such as audio and video has the requirement that data delivery must proceed at a guaranteed steady rate. For example, if audio data are not supplied in time, there will be gaps in the sound. If data are supplied too fast, system buffers may overflow resulting in loss of data.

14.4 Data Mining

Definition:

Extraction of implicit, previously unknown and unexpected, potentially extremely useful information/patterns from data in large databases or data warehouses.

Data Mining Applications:

1) Science and Medicine
   i) Telescope-star galaxy classification
   ii) Medical image analysis
   iii) Identical successful medical therapies

2) Bio Science
   Changes in gene expression

3) Business and Ecommerce
   i) Predict Customer Spending
   ii) Sales Forecasting
   iii) Fraud Detection

4) Marketing
   i) Customer Buying Patterns
   ii) Market Basket Analysis

14.5 What is a NoSQL (Not Only SQL) Database?

A NoSQL database environment is a non-relational and largely distributed system that enables in the foundation of a rapid ad-hoc organization. It involves analysis of extremely high-volume, disparate data types. NoSQL databases are sometimes referred to as cloud databases, non-relational databases, Big Data databases or other noteworthy terms. They are large volumes of data being generated, stored and analyzed by modern users and their applications.

The key deciding factors why NoSQL databases have become the first alternative to relational databases are scalability, availability and fault tolerance which I will go into details later on. They are in fact schema-less data models with horizontal scalability, distributed architecture and usage of languages and interfaces.

Considering a “NoSQL” or “BigData” environment has been shown to provide a clear competitive advantage in numerous industries. In the age of large volumes of data, the importance of data is summed up as “if your data is not growing, then neither is your business”.

14.5.1 Types of NoSQL Databases

There are four general types of NoSQL databases, each with their own specific attributes:

- **Key-Value store**: In this type of database, we use some of the least complex NoSQL options. These databases are designed for storing data in a schema-less way. In a key-value store, all of the data consists of an indexed key and a value.
- **Column store**: (aka as wide-column stores) These databases are designed for storing data tables as sections of column of data, rather than as rows of data. Wide-column stores offer a very high performance and high scalable structure.
- **Document Database**: extends the idea of key-value stores where documents contain more complex data and each document is assigned a unique key, which is used to retrieve the document. Summing up, these databases store, retrieve and manage document-oriented data, also known as semi structured data.
- **Graph Database**: Based on graph theory, these databases are designed with data whose relations are well represented as a graph with interconnecting elements in them.

14.5.2 Advantages of NoSQL Databases over Relational Databases

The Growth of Big Data

Big Data act as force multipliers for data growth for simple online actions to point of sale system to GPS tools to smart phones and tablets to sophisticated sensors and many more.

The simple reason that NoSQL becomes a necessity is because you have a Big Data Project to handle. A Big Data project is characterized by:

1. High data velocity:-lots of data coming in rapidly from various locations.
2. Data Variety:- storage of structured, semi-structured and unstructured data.
3. Data volume:- many terabytes and petrabytes of data involved.
4. Data complexity:- data stores and centers in different locations storing and managing data.

Continuous Data Availability
Hardware failures are apt to occur in today's business. Fortunately, NoSQL databases have distributed architectures so that if one or more database servers or nodes go down, the other nodes of the system will continue to operate, exhibiting fault tolerance. However, when deployed in an appropriate way, NoSQL databases can provide high performance at a massive scale without going down and hence without losing real dollars.

Real Location Independence
The term "location independence" means reading and writing to a database regardless of location, where that I/O operation occurs physically and is available to users and machines at other sites.

Modern Transactional Capabilities
ACID transctions are not maintained in NoSQL databases. The "Consistency" that concerns NoSQL databases is found in a theorem that signifies the immediate or eventual consistency of data in a distributed environment.

Flexible Data Models
A NoSQL data model aka a schema-less one can support huge volumes of data that do not fit well into a RDMS. A NoSQL database can store all types of data: structured, semi-structured, and unstructured data unlike a relational database. Such a NoSQL data model easily delivers fast performance for both read and write operations.

Better Architecture
A NoSQL database strives to provide a more suitable architecture for a particular application in many cases. Modern NoSQL databases not only provide storage and management of application data that cater for an instant understanding of complex data and facilitate flexible data mining, analysis and decision-making.

14.5.3 Choosing the Right NoSQL Database

Key cosiderations when choosing your NoSQL platform include:

- **Workload diversity**: Big data comes in all shapes, colors, and sizes which requires a flexible design unlike rigid schemas. This preferred design will perform transactions real fast, run analytics just as fast and identify any data from volumes of data and much more.

- **Scalability**: With big data comes the need for scaling very rapidly and elastically across multiple data centers and also even across clouds.

- **Performance**: In a NoSQL database, Big Data must move at extremely high velocities no matter how much you scale or what workloads your database must perform because even nanosecond delays can cost dollars.

- **Continuous Availability**: This consideration of Big Data is not always high enough, considering high performance on the other hand, because it may be difficult to maintain all the time that data can never go down and hence, there can be no single point of failure in a NoSQL environment.
- **Manageability**: For this consideration, we have to make sure that administration and development both maintain and maximize the benefits of moving to a NoSQL environment.
- **Cost**: Deploying NoSQL properly allows for all of the benefits above including most of all, lowering operational costs.
- **Strong community**: Maintaining and choosing NoSQL environment not only requires strong support and resources availability but also a solid and capable community around the technology, including individuals and teams.

MISCELLANEOUS DATABASE PROJECT

i) Find an exciting and interesting database project to work on. It can be a hospital system, a bank system, musical band system, hobbies system, ticket airlines system, fast food restaurent system, hotel system or anything that passionates and inspires you.

ii) Once you have located the system, think about and find out what can stand for entities and attributes for those entities.

iii) Finally draw the entity relatioship diagram (ERD), linking up the entities. Find out whether binary or ternary relationships suit among them as appropriate among the entities.

iv) Once the ERD is complete, convert it to a relational model (table schemas).

v) Once the relational model is done, together with the help of ERD, find out full functional dependencies among the attributes of the relations, and then normalize the tables of the relational model and see if they fulfill 1NF, 2NF, 3NF and BCNF. In case of violation of one or more of these, split tables as described within the content of this book as required step by step.

vi) Next utilize the above information in mysql or sql server taking care to include split tables, if any, from (v).

vii) Run simple to more complex SQL expressions corresponding to queries starting with small data in tables and going on to bigger data sets. Check back to see if your expressions are giving the correct results otherwise modify the SQLexpressions to get correct results.

viii) **BONUS :** If you want to take a bigger challenge, create an interface or front end for your system using PHP/Javascript, C Sharp or ASP.net by connecting with mysql or sql server at the backend.

About the Author

Rosina S Khan, the author of this book is a former senior faculty member of a private university in Dhaka city. She has taught the undergraduate course Databases and guided group and individual experiments in Database labs for quite a many semesters for years in the university. Furthermore, she has managed and supervised serveral 4th year undergraduate theses and projects in the field of Databases. Her own master's thesis done as a part of MSc degree in Software Technology in Germany was also based on Databases.

WHOEVER
WHENEVER
WHEREVER
YOU ARE