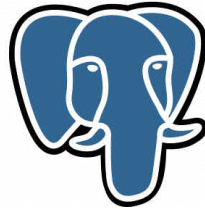


# DBMS: AN INTERACTIVE TUTORIAL

---

PostgreSQL



ORACLE

SYBASE



Organized & Prepared By

**Sharafat Ibn Mollah Mosharraf**

12<sup>th</sup> Batch (05-06)

Dept. of Computer Science & Engineering

University of Dhaka



# Table of Contents

<b>INTRODUCTION TO DATABASE AND DESIGNING RELATIONAL DATABASES.....</b>	<b>1</b>
DATABASE: WHAT IT IS AND WHY IT IS NEEDED .....	1
DESIGNING A DATABASE.....	1
EXAMPLE DATABASE DESIGN.....	5
<b>ENTITY-RELATIONSHIP MODEL.....</b>	<b>7</b>
E-R MODEL: WHAT IT IS.....	7
BASIC CONCEPTS .....	7
<i>Entity Sets and Attributes</i> .....	7
Types of Attributes .....	8
<i>Relationship Sets</i> .....	8
CONSTRAINTS .....	9
<i>Cardinality Constraints / Mapping Cardinalities / Cardinality Ratios</i> .....	10
<i>Participation Constraints</i> .....	11
<i>Cardinality Limits</i> .....	11
WEAK ENTITY SETS.....	11
Discriminator / Partial Key of a Weak Entity Set .....	12
How the primary key of a weak entity set is formed.....	12
Placement of descriptive attributes in a weak entity set.....	12
Participation of weak entity sets in relationships .....	12
Weak entity sets – should they be designed as multivalued attributes? .....	13
SPECIALIZATION AND GENERALIZATION .....	13
<i>Specialization</i> .....	13
<i>Generalization</i> .....	13
REDUCTION OF E-R SCHEMAS TO TABLES .....	14
<i>Strong Entity Sets</i> .....	14
<i>Weak Entity Sets</i> .....	14
<i>Relationship Sets</i> .....	14
One-to-One Relationship .....	14
One-to-Many Relationship.....	14
Many-to-One Relationship.....	14
Many-to-Many Relationship.....	15
<i>Composite Attributes</i> .....	15
<i>Multivalued Attributes</i> .....	15
<i>Generalization</i> .....	15
E-R DIAGRAM SYMBOLS AT A GLANCE.....	16
<b>DML: DATA-MANIPULATION LANGUAGE.....</b>	<b>18</b>
DML (DATA-MANIPULATION LANGUAGE): WHAT IT IS .....	18
QUERY LANGUAGE .....	18
THE SELECT OPERATION / STATEMENT .....	18
RELATIONAL ALGEBRA .....	21
<i>Fundamental Relational Algebra Operations</i> .....	21
The Selection Operation .....	21
The Projection Operation.....	21
Composition of the Relational Operations .....	21
The Union Operation .....	22
The Set-Difference Operation.....	22
The Cartesian-Product Operation.....	23
The Rename Operation .....	24
<i>Additional Relational Algebra Operations</i> .....	25
The Set-intersection Operation .....	26
The Natural Join Operation.....	26
<i>Extended Relational Algebra Operations</i> .....	27
The Outer Join Operation.....	27
Generalized Projection.....	28
Aggregate Functions .....	28
MODIFICATION OF DATABASE .....	31
<i>Insertion</i> .....	31
<i>Deletion</i> .....	31
<i>Update</i> .....	32
<b>INDEXING AND HASHING.....</b>	<b>33</b>
THE PROBLEM .....	33

BASIC CONCEPTS .....	33
<i>Types of Indices</i> .....	33
<i>Index Technique Choosing Factors</i> .....	33
ORDERED INDICES .....	34
Primary / Clustering Index .....	34
Secondary / Non-Clustering Index .....	34
Index-Sequential Files .....	34
Contents of an index record / entry .....	34
<i>Types of ordered indices</i> .....	34
Dense Index.....	35
Dense index for primary indices .....	35
Another implementation of dense indices .....	35
Sparse Index.....	35
Comparative Analysis of Dense and Sparse Index.....	36
A good trade-off.....	36
Why this trade-off is good .....	36
Multi-Level Indices.....	36
The problem with single-level indices .....	36
Solution to this problem.....	36
Secondary Indices .....	37
B <sup>+</sup> TREE INDEX .....	37
<i>The Problem with Indexed-Sequential File Organization</i> .....	37
<i>How B<sup>+</sup> Tree Index Solves the Problem</i> .....	37
<i>Structure of a B<sup>+</sup> Tree</i> .....	37
<i>Operations on a B<sup>+</sup> Tree</i> .....	38
Adding Records to a B <sup>+</sup> Tree .....	38
Deleting records from a B <sup>+</sup> Tree .....	40
B <sup>+</sup> Tree File Organization.....	41
B-TREE INDEX .....	42
Advantages of B-Tree .....	43
Disadvantages of B-Tree .....	43
HASHING .....	43
<i>The Problem with Sequential File Organization and How Hashing Solves It</i> .....	43
Hash File Organization.....	43
Manipulation of Records in Hash Files.....	43
Hash Functions .....	43
Distribution Qualities for Choosing a Hash Function .....	44
Some Examples Illustrating These Qualities.....	44
How Hash Functions Should be Designed .....	44
Handling of Bucket Overflows .....	44
Causes of bucket overflows .....	45
Reducing bucket overflows.....	45
Handling bucket overflows .....	45
Hash Indices.....	46
Static and Dynamic Hashing.....	46
Drawbacks of Hashing.....	47
COMPARISON OF ORDERED INDEXING AND HASHING .....	47
MULTIPLE-KEY INDICES .....	48
Problem with Multiple Single-Key Indices.....	48
Advantages of Using Multiple-Key Indices .....	48
INDEX DEFINITIONS IN SQL.....	49
Creating an Index.....	49
Removing an Index.....	49

## Meanings of Special Formatting used

Formatting	Meaning	Example
<b>Bold text</b>	Technical terms in Database	<b>tuple, one-to-many</b>
<i>Italic text</i>	Attribute names, table names	<i>account, customer-name</i>
Dark red colored text	Topic heading	Designing a database
Dark red colored text and objects within a diagram or table	Newly added item compared to previous diagram or table or text	
Blue colored text	Points to be remembered	What we've learnt from here is that, when ...
Blue colored text under a figure	Figure Caption	Figure 1: ...
Red colored text	Conclusion	To sum-up, we have ...

# Introduction to Database and Designing Relational Databases

## Database: What it is and why it is needed

A *database* is a structured collection of records or data that is stored in a computer system.

Fine. But we can implement a database using files – we can write a program to put some information into a file and when needed, we can retrieve that information. Then what's the point of introducing a course on databases???

Well, that's true... But think about this situation: suppose your employer wants to manage a *student* database. So you write a file containing students' class, roll number and marks in each subject. Now, your employer wants to get the list of students who have failed in a particular course. You write some code to display the list. However, some time later, your employer wants to get the list of students who scored A<sup>+</sup> in a particular course. You again write some code to display the list. Now, once again, your employer wants the total number of students in a particular class. You have to modify your program again. But how many times are you going to take the pain of modifying your code?

If there were such a system which would instantly respond to your any kind of query within a database, then that would make your life much comfortable. Here comes the Database Management System (DBMS). This system provides you with an easy way to store and retrieve data and handle all your queries about those data. There are much more functionalities of a DBMS, of course.

The objective of our course on DBMS is to learn *how to* implement, manage and use a DBMS.

At the preliminary level, for the time being, our main learning objectives are:

1. Designing a database (efficiently).
2. Using queries (efficiently) to store data into and retrieve data from a database.

## Designing a database

Suppose, we have to design a database to manage a banking system.

First, we need to know what data we should need to manage a banking system. Let's start with information on customers. For each customer, we can record his *name*, *street address*, *city*, and most importantly, his *account number* in the bank. These *name*, *street address*, *city* and *account number* are called **attributes** or **fields** of *customer*.

Now that we've got our attributes, let's try to establish a relationship among them. In this case, it's quite simple: a customer with a name '*name*' lives in '*street address*' in city '*city*' and has an account number specified by the attribute '*account number*' in the bank.

In **relational database model**, we depict the above relation by placing the attributes into a box, which is called a **schema** and giving a name to the schema, somewhat like the following:

*customer*

<i>customer-name</i>
<i>customer-street</i>
<i>customer-city</i>
<i>account-number</i>

The *customer* **relation** (also called a **table**) with two sample **records** (also called **rows** or **tuples**<sup>1</sup>) might be as follows:

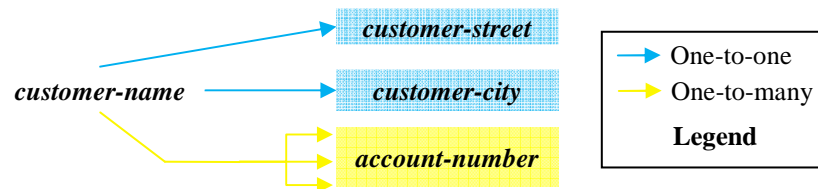
---

<sup>1</sup> **Tuple**: (computing and mathematics) a finite sequence of objects; a structure containing multiple parts.

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>	<i>account-number</i>
Somebody	Mirpur Road	Dhaka	A-101
Anybody	XYZ Road	Khagrachhori	A-104

A question may arise – why would we ever need to define a relationship among the attributes? Well, we’ll find the answer soon...

Let’s take a closer look at the relation. A customer can live at only one street address and in only one city at a time. So, we can say that the relationships of *customer-name* with *customer-street* and *customer-city* are **one-to-one**. But as for *account-number*, we know that a customer might have more than one account at a time; hence the relationship of *customer-name* with *account-number* is **one-to-many**.



**Figure 1:** Relationships among the attributes of *customer* table.

So why does it matter? Suppose, the customer from above table named *Somebody* has three accounts at the bank (A-101, A-102 and A-103). Then we can store that information into the table as follows:

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>	<i>account-number</i>
Somebody	Mirpur Road	Dhaka	A-101, A-102, A-103

Now, when we query for the details of the customer who has an account number of A-103, we have to specifically search through each field for the account number; thus implementing our own search program (as the built-in query *naturally* do not search into a field). So, the objective of DBMS cannot be achieved.<sup>2</sup>

However, we can implement a work-around for the problem by placing each account number in a single record like the following:

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>	<i>account-number</i>
Somebody	Mirpur Road	Dhaka	A-101
Somebody	Mirpur Road	Dhaka	A-102
Somebody	Mirpur Road	Dhaka	A-103

But this makes several copies of the same data and thus wastes storage space:

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>	<i>account-number</i>
Somebody	Mirpur Road	Dhaka	A-101
Somebody	Mirpur Road	Dhaka	A-102
Somebody	Mirpur Road	Dhaka	A-103

Duplicate data

To solve both the problems, let’s split up the table into two tables as below:

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
Somebody	Mirpur Road	Dhaka

*customer*

<i>account-number</i>	<i>customer-name</i>
A-101	Somebody
A-102	Somebody
A-103	Somebody

*account*

Now, when we query for the details of the customer who has an account number of A-103, the *account* table will be queried first. From the account table, it will be found that a customer named *Somebody* holds that account number. Next, the details of the customer will be queried from the *customer* table using that customer name ‘*Somebody*’. Thus, we’ll be able to easily and successfully retrieve all the necessary data using a DBMS.

In the two tables, the field *customer-name* is acting as a link between records.

<sup>2</sup> There are some rules for designing efficient databases, one of which is “each field must contain **atomic** data” – i.e., data which represents *only one* thing, not several things.

What we've learnt from here is that, when a *one-to-many* relationship exists – suppose – from field *A* to field *B*, we'll create two tables: in the first one, we'll keep the field *A*, and in the second one, we'll place the field *B* and also the field *A*.

This is the reason why we need to define relationships among the attributes – so that we can distribute the attributes among different tables when designing and implementing a database.

Well, if you haven't figured out yet, there is a problem with the above solution. We all know that it is possible that any two customers might have the same name. Consider the following situation:

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
Somebody	Mirpur Road	Dhaka
Somebody	Aga Kha Road	Bogra
Anybody	XYZ Road	Khagrachhori

*customer*

<i>account-number</i>	<i>customer-name</i>
A-101	Anybody
A-102	Anybody
A-103	Somebody

*account*

From *account* table, we find that account *A-103* is owned by *Somebody*. Now, from *customer* table, how can we determine whether this *Somebody* is the one living in *Dhaka*, or the one in *Bogra*?

It seems that we need to use such a linking field in *account* table, which can *uniquely* identify a record in the *customer* table. So, which field in *customer* table uniquely identifies a record therein? There might be several customers who have the same name or live at the same street or even in the same city. Therefore, a value may appear more than once in any of the fields.

In such cases where none of the attributes uniquely identifies a record in the table, we have to use a field – usually named *id* – in which unique integer values are assigned for each record. Thus, the above two tables become:

<i>customer-id</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
1	Somebody	Mirpur Road	Dhaka
2	Somebody	Aga Kha Road	Bogra
3	Anybody	XYZ Road	Khagrachhori

*customer*

<i>account-number</i>	<i>customer-id</i>
A-101	3
A-102	3
A-103	1

*account*

Now we can definitely say that the owner of the account *A-103* is a customer named *Somebody* who lives in *Dhaka* (and not in *Bogra*).

A field which uniquely identifies a record in a table is called a **primary key**. In *customer* table, *customer-id* is a primary key. Again, a primary key which is used as a field in another table for linking (i.e., relationship establishing) purposes, is called a **foreign key**. In *account* table, *customer-id* is a foreign key. Primary key and foreign key attributes are usually *underlined* to express their nature.

Primary Key

<u><i>customer-id</i></u>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
1	Somebody	Mirpur Road	Dhaka
2	Somebody	Aga Kha Road	Bogra
3	Anybody	XYZ Road	Khagrachhori

*customer*

Foreign Key

<i>account-number</i>	<u><i>customer-id</i></u>
A-101	3
A-102	3
A-103	1

*account*

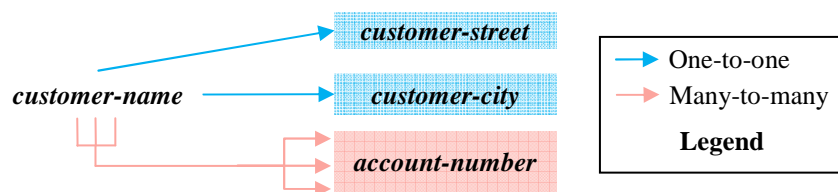
Note that we can also define a primary key for the *account* table, and it would be the *account-number* field.

So what we've learnt from this section is that we need a primary key to uniquely identify a record in a table, and we need to uniquely identify a record in a table to properly establish relationship among attributes; thus design an efficient database. However, primary key is also used to prevent inserting records which contain duplicate primary key field values.

You might have started thinking, “Well, we've solved the problem quite well!” But you're wrong! It's true that a customer might have more than one account at a time; but it's also true that more than one



customer might own a single account (which we call *joint-account*). So, the actual relationship between *customer-name* and *account-number* is **many-to-many**.



**Figure 2:** Relationships among the attributes of *customer* table.

Now, what happens when the account number *A-104* is owned by both the customers named *Somebody* and *Anybody*? In the *account* table, we'll either have to break the rule of *atomic data*, or we'll have to keep duplicate records. Both of the solutions are unacceptable.

So, again, let's split up the table *account* into two tables:

Primary Key				Primary Key		Foreign Key	Foreign Key
<u>customer-id</u>	<u>customer-name</u>	<u>customer-street</u>	<u>customer-city</u>	<u>account-number</u>		<u>account-number</u>	<u>customer-id</u>
1	Somebody	Mirpur Road	Dhaka	A-101		A-101	3
2	Somebody	Aga Kha Road	Bogra	A-102		A-102	3
3	Anybody	XYZ Road	Khagrachhori	A-103		A-103	1
						A-104	1
						A-104	3

*customer*

*account*

*depositor*

What we've done here is, we've created a *link* table named *depositor* containing two foreign keys – each of which refers to the corresponding primary key in the tables *customer* and *account*. Thus, from the *depositor* table, we can say that the account number *A-104* is owned by two customers whose *ids* are *1* and *3*. And then from the *customer* table, we can find the details about those customers. The interesting fact is that, the previous queries we've tried can also be executed easily from these tables with this configuration. Why don't you try those for yourself right now? 😊

[Umm... you might think that the *account* table seems to be lonely. However, when you'll be designing a database in a real situation, you'll definitely find that the *account* table does not consist of only one attribute. It will contain more than one fields, for example, *balance* – the amount of money currently available under that account number.]

What we've learnt from this section is that when a *many-to-many* relationship exists – suppose – from field *A* to field *B*, we'll create three tables: in the first one, we'll keep the field *A* and the fields related to it; in the second one, we'll place the field *B* and the fields related to it; and finally, in the last table, we'll put two fields from the previous two tables which are primary keys of those tables. The last table is just a linking table.

To sum-up, we have to follow the following steps when designing a database:

1. Identify which attributes we need and place them into relevant tables.
2. Identify the primary keys of the tables.
3. Identify the relations among the attributes and modify table design accordingly.

Note that these steps are the *primary* steps for designing a database. There are other issues to consider when designing a database efficiently. We'll discuss those later.



## Example Database Design

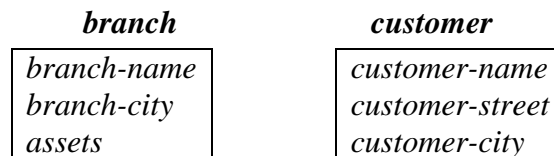
Let's design a database for bank management.

First, we have to determine which information we need to manage the bank. Suppose we need the following information:

1. Branch details – which might include branch name, branch city and total assets of that branch.
2. Customer details – customer name, his street address and the city he lives in.
3. Account details – the account numbers, which branches the accounts are from, their owners and their balances.
4. Loan details – the loan numbers, which branches the loans are from, their borrowers and their amounts.

Next, let's try to figure out the relationships among these attributes so that we can determine which tables we should need and which attributes should go to which table.

First, we've to figure out which attributes are *not* related to each-other. The bank details are not in any way related to customer details. So, we can safely create two tables named – for example – *branch* and *customer*. We can fill those tables with necessary attributes – *branch-name*, *branch-city* and *assets* in *branch* table; and *customer-name*, *customer-street* and *customer-city* in *customer* table. Our database schema up to this point should look like the following:



Now, the account details seems to contain attributes which relate to attributes in both branch details and customer details. So, we can create a table named accounts and put the necessary attributes (*account-number*, *branch-name* and *balance*) in it.



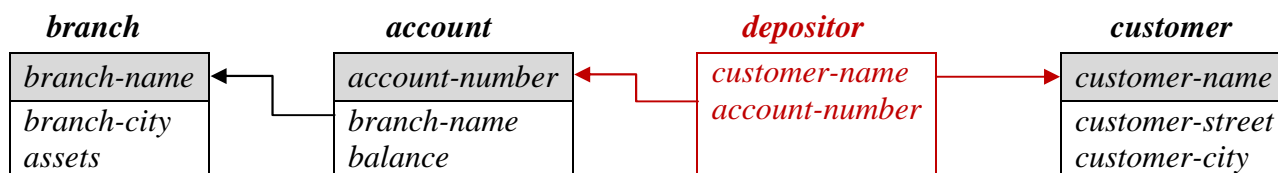
Note that the *branch-name* attribute in *account* table refers to the *branch-name* attribute in *branch* table. So, *branch-name* should uniquely identify each record in *branch* table. In other words, *branch-name* should be a primary key. Let's assume *branch-name* is unique for the bank, i.e., all the branches of the bank have different names. So, we can say that *branch-name* is a primary key in *branch* table, and hence, a foreign key in *account* table.

Again, the *customer-name* attribute in *account* table refers to the *customer-name* attribute in *customer* table. Let's assume that customer names are unique (although in real situation it is not, let's just assume it for ease of our design), and thus *customer-name* is a primary key in *customer* table and a foreign key in *account* table.

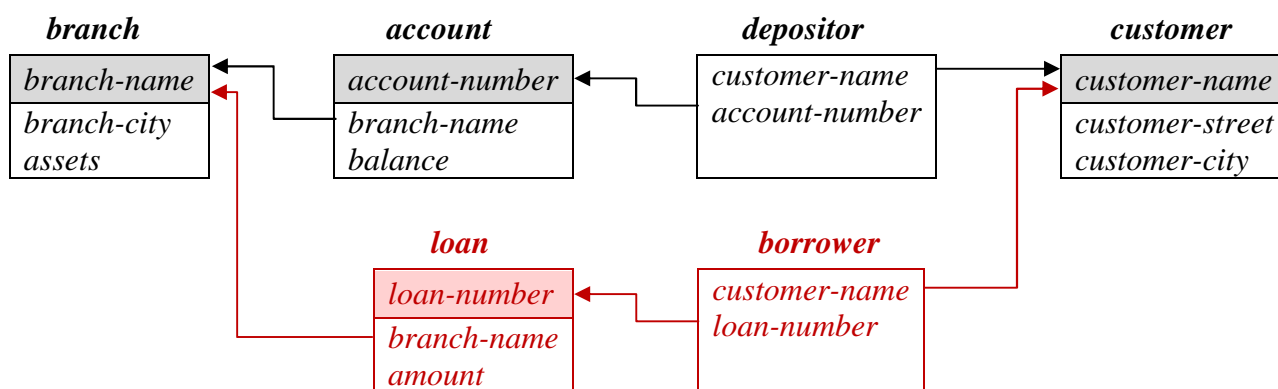
Note that when drawing a schema diagram for a database, the following rules are maintained:

- Each relation appears as a box, with the attributes listed inside it and the relation name *above* it.
- If there are primary key attributes, a horizontal line crosses the box, with primary key attributes listed above the line.
- It is customary to list the primary key attributes of a relation schema before the other attributes.
- Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation (or table – both denotes the same thing) to the primary key of the referenced relation.

Okay, let's get back to our business. If we analyze more carefully, we can confirm that a customer may have multiple accounts and a single account can be shared by multiple customers. So, the relationship with *customer-name* and *account-number* is *many-to-many*. So, according to what we learned previously, let's split-up the account table and design a link table named – for example, *depositor* – like the following:



Now remains the *loan details*. It's much like the *account details* in relationship's point of view. We need a foreign key *branch-name* and we know that the relationship between *loan-number* and *customer-name* is *many-to-many* (as a customer may take multiple loans and a single loan can be jointly taken by multiple customers). So, we can easily design a table named *loan* and a link table named – for example, *borrower*:



So that's our final database schema diagram for the bank management system.

We can represent a schema diagram in textual form, known as **relation schema**. The relation schema of the above diagram is as follows:

Branch\_schema = (branch-name, branch-city, assets)  
 Customer\_schema = (customer-name, customer-street, customer-city)  
 Account\_schema = (account-number, branch-name, balance)  
 Loan\_schema = (loan-number, branch-name, amount)  
 Borrower\_schema = (customer-name, loan-number)  
 Depositor\_schema = (customer-name, account-number)

Note that in relation schema, no foreign relation is depicted. Further note that the schema names start with uppercase letters.

So, the general form of a relation schema is:

$S = (A_1, A_2, \dots, A_n)$ , where  $A$  stands for *Attribute*.

A table filled with some records is called a *relation instance*. A relation instance  $r$  on a schema  $S$  can be written as:

$r(S)$  or  $r(A_1, A_2, \dots, A_n)$

For example, to denote that *account* is a relation on *Account\_schema*, we can write:

*account*(*Account\_schema*) or *account*(*account-number*, *branch-name*, *balance*)

Note that relation instance names start with lowercase letters.

# Entity-Relationship Model

## E-R Model: What it is

The entity-relationship (E-R) data model perceives the real world as consisting of basic objects, called entities, and relationships among these objects. It was developed to facilitate database design by allowing specification of an enterprise schema, which represents the overall logical structure of a database. The E-R data model is one of several semantic data models; the semantic aspect of the model lies in its representation of the meaning of the data. The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema.

## Basic Concepts

The E-R data model employs three basic notions: *entity sets*, *relationship sets*, and *attributes*.

### Entity Sets and Attributes

An **entity** is a “thing” or “object” in the real world that is distinguishable from all other objects. For example, each person in an enterprise is an entity. An entity has a set of properties. For example, a person has a name, address etc.

An **entity set** is a set of entities of the same type that share the same properties, or attributes. The set of all persons who are customers at a given bank, for example, can be defined as the entity set *customer*.

An entity is represented by a set of **attributes**. Attributes are descriptive properties possessed by each member of an entity set. The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the *customer* entity set are *customer-id*, *customer-name*, *customer-street* and *customer-city*.

Each entity has a **value** for each of its attributes. For instance, a particular *customer* entity may have the value *321-12-3123* for *customer-id*, the value *Jones* for *customer-name*, the value *Main* for *customer-street*, and the value *Harrison* for *customer-city*.

For each attribute, there is a set of permitted values, called the **domain** or **value set** of that attribute. The domain of attribute *customer-name* might be the set of all text strings of a certain length.

A database thus includes a collection of entity sets, each of which contains any number of entities of the same type. The following figure shows part of a bank database that consists of two entity sets: *customer* and *loan*:

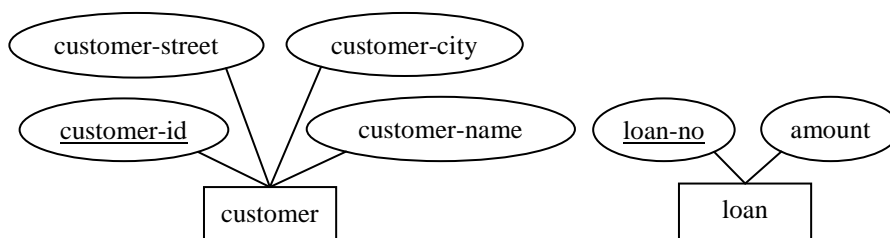


Figure (a): E-R diagram notations for entity sets and attributes.

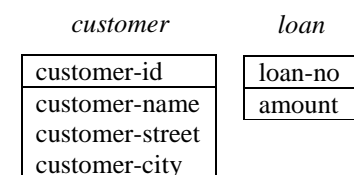


Figure (b): Alternative E-R diagram notations.

<u>customer-id</u>	customer-name	customer-street	customer-city
321-12-3123	Jones	Main	Harrison
321-12-3124	Smith	North	Rye
119-15-4569	Hayes	Dupont	Harrison
123-45-6789	Adams	Spring	Princeton

customer

<u>loan-no</u>	amount
L-17	1000
L-23	500
L-46	9000
L-12	30000

loan

Figure (c): Relation instances of *customer* and *loan*.

## Types of Attributes

An attribute, as used in the E-R model, can be characterized by the following attribute types:

### 1. Simple and Composite Attributes

Attributes that *cannot* be divided into subparts are called **simple** attributes.

For example, *telephone-no*, *salary* etc.

Attributes that *can* be divided into subparts are called **composite** attributes.

For example, the composite attribute *address* can be divided into attributes *street-number*, *street-name* and *apartment-number*.

### 2. Single-valued and multivalued attributes

Attributes that have a single value for a particular entity are called **single-valued** attributes.

For example, *customer-name*, *salary* etc.

Attributes that have multiple values for a particular entity are called **multivalued** attributes.

For example, an *employee* may have multiple telephone numbers. So, the attribute *telephone-no* is a multivalued attribute.

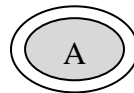


Figure: Symbol for multivalued attribute in E-R diagram.

### 3. Derived Attributes

If the value of an attribute can be derived from the values of other related attributes or entities, then that attribute is called a **derived** attribute.

The attribute from which another attribute is derived is called the **base** or **stored** attribute.

The value of a derived attribute is not stored, but is computed when required.

For example, if an entity set *employee* has two attributes *date-of-birth* and *age*, then the attribute *age* is a derived attribute and the attribute *date-of-birth* is the base or stored attribute.



Figure: Symbol for derived attribute in E-R diagram.

## Relationship Sets

A **relationship** is an association among several entities.

For example, we can define a relationship that associates customer Hayes with loan L-15. This relationship specifies that Hayes is a customer with loan number L-15.

A **relationship set** is a set of relationships of the same type.

For example, consider the two entity sets *customer* and *loan*. We define the relationship set *borrower* to denote the association between customers and the bank loans that the customers have. The following figure depicts this association:

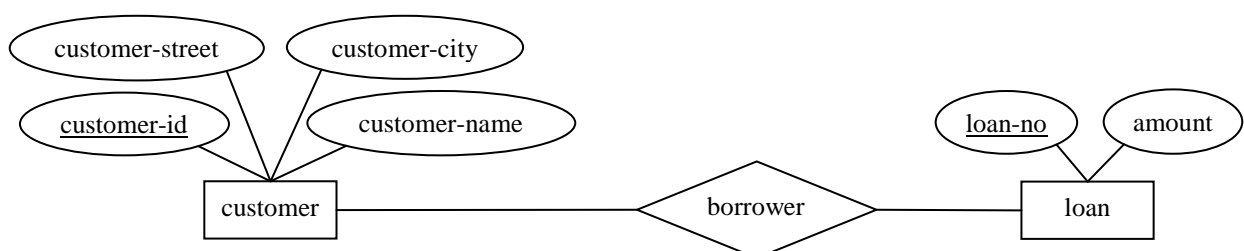


Figure: E-R diagram for *customer* and *loan* entity sets and the relationship set *borrower*.

The association between entity sets is referred to as **participation**; that is, the entity sets  $E_1, E_2, \dots, E_n$  **participate** in relationship set  $R$ .

A **relationship instance** in an E-R schema represents an association between the named entities in the real-world enterprise that is being modeled.

As an illustration, the individual customer entity Hayes, who has customer identifier 677-89-9011, and the loan entity L-15 participate in a relationship instance of *borrower*. This relationship instance represents that, in the real-world enterprise, the person called Hayes who holds customer-id 677-89-9011 has taken the loan that is numbered L-15.

A relationship may also have attributes called **descriptive attributes**. For example, consider the entity sets *customer* and *loan* and the relationship set *borrower*. We could associate the attribute *date-issued* to that relationship to specify the date when the loan was issued:

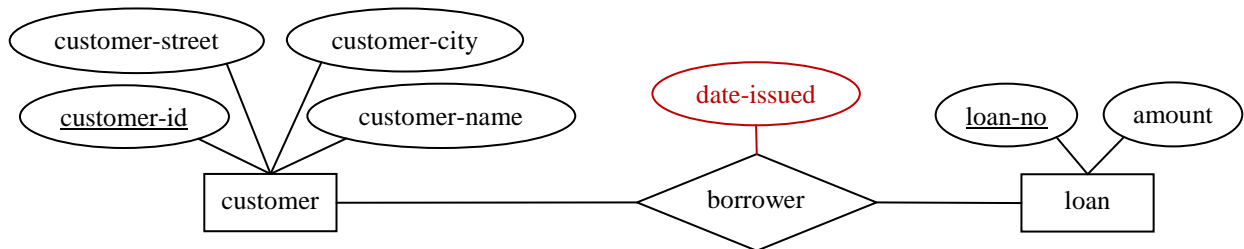


Figure: Descriptive attribute *date-issued*.

Most of the relationship sets in a database system are **binary** – that is, they involve two entity sets. Occasionally, however, relationship sets involve more than two entity sets.

As an example, consider the entity sets *employee*, *branch*, and *job*. Examples of job entities could include manager, teller, auditor, and so on. Job entities may have the attributes *title* and *level*. The relationship set *works-on* among *employee*, *branch*, and *job* is an example of a **ternary** relationship. A ternary relationship among Jones, Perryridge, and manager indicates that Jones acts as a manager at the Perryridge branch. Jones could also act as auditor at the Downtown branch, which would be represented by another relationship. Yet another relationship could be between Smith, Downtown, and teller, indicating Smith acts as a teller at the Downtown branch.

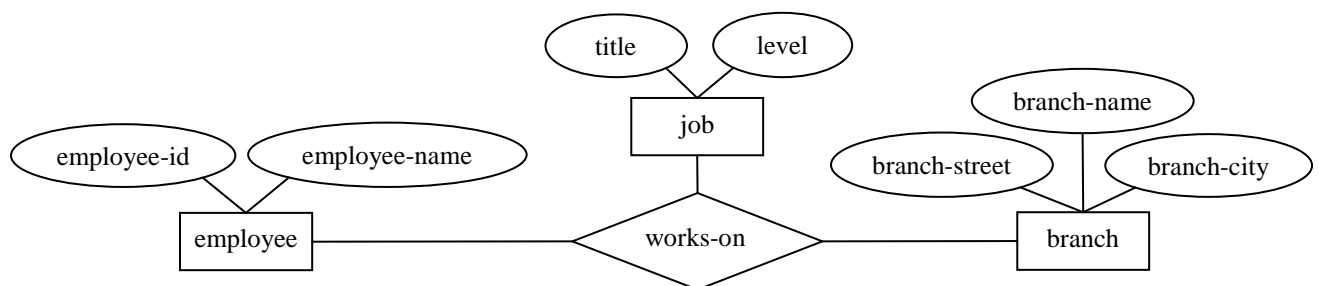


Figure: Ternary relationship.

The number of entity sets that participate in a relationship set is called the **degree** of the relationship set. A binary relationship set is of degree 2; a ternary relationship set is of degree 3.

## Constraints

An E-R enterprise schema may define certain constraints to which the contents of a database must conform. Different types of constraints can be depicted in E-R model:

1. Cardinality Constraints – *one-to-one*, *one-to-many*, *many-to-one*, *many-to-many*
2. Participation Constraints – *partial*, *total*
3. Key Constraints – *superkey*, *candidate key*, *primary key*

## Cardinality Constraints / Mapping Cardinalities / Cardinality Ratios

**Cardinality constraints** express the number of entities to which another entity can be associated via a relationship set.

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

- **One to one.** An entity in  $A$  is associated with at most one entity in  $B$ , and an entity in  $B$  is associated with at most one entity in  $A$ .

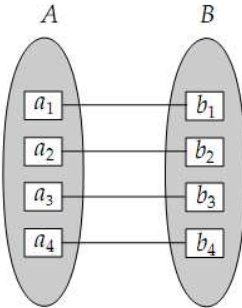


Figure (a): One-to-one mapping cardinality.

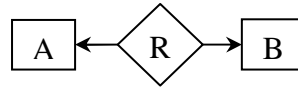


Figure (b): E-R diagram notation for one-to-one relationship

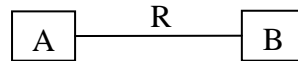


Figure (c): Alternative notation.

- **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  $A$ .

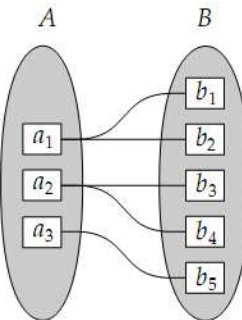


Figure (a): One-to-many mapping cardinality.

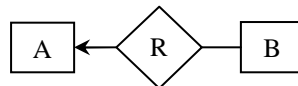


Figure (b): E-R diagram notation for one-to-many relationship

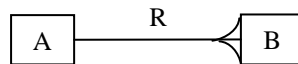


Figure (c): Alternative notation.

- **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  $A$ .

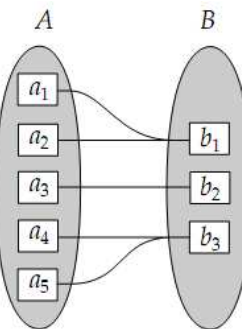


Figure (a): many-to-one mapping cardinality.

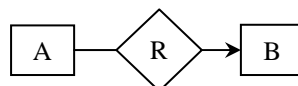


Figure (b): E-R diagram notation for many-to-one relationship

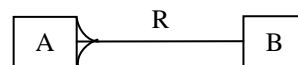


Figure (c): Alternative notation.

- **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  $A$ .

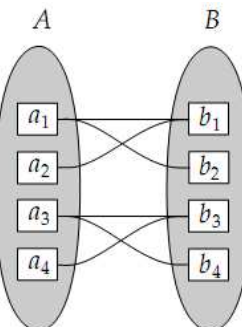


Figure (a): many-to-many mapping cardinality.

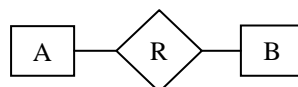


Figure (b): E-R diagram notation for many-to-many relationship

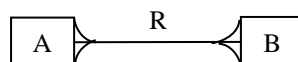


Figure (c): Alternative notation.



## Participation Constraints

The participation constraints used in E-R model are:

1. Total
2. Partial

The participation of an entity set  $E$  in a relationship set  $R$  is said to be **total** if every entity in  $E$  participates in at least one relationship in  $R$ . If only some entities in  $E$  participate in relationships in  $R$ , the participation of entity set  $E$  in relationship  $R$  is said to be **partial**.

For example, we expect every *loan* entity to be related to at least one *customer* through the *borrower* relationship. Therefore the participation of *loan* in the relationship set *borrower* is total.

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

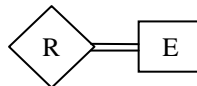


Figure: E-R notation for total participation of entity set in relationship.

## Cardinality Limits

E-R diagrams also provide away to indicate more complex constraints on the number of times each entity participates in relationships in a relationship set.

An edge between an entity set and a binary relationship set can have an associated *minimum* and *maximum* cardinality, shown in the form  $l..h$ , where  $l$  is the minimum and  $h$  the maximum cardinality.

A minimum value of 1 indicates total participation of the entity set in the relationship set.

A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value  $*$  indicates no limit.

Note that a label  $1..h$  on an edge is equivalent to a double line.

For example, consider the following figure. The edge between *loan* and *borrower* has a cardinality constraint of  $1..1$ , meaning the minimum and the maximum cardinality are both 1. That is, each loan must have exactly one associated customer. The limit  $0..*$  on the edge from *customer* to *borrower* indicates that a customer can have zero or more loans. Thus, the relationship *borrower* is one to many from *customer* to *loan*, and further the participation of *loan* in *borrower* is total.

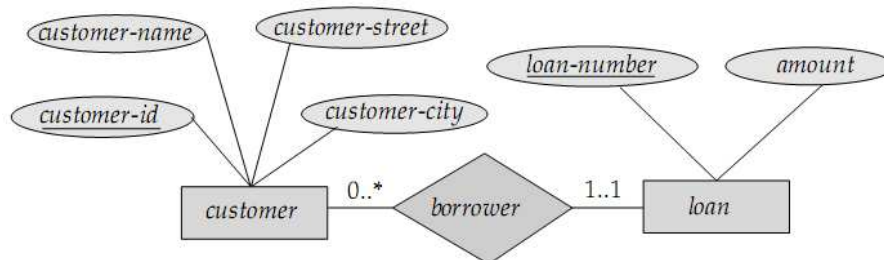


Figure: Cardinality limits on relationship sets.

## Weak Entity Sets

An entity set that do not have sufficient attributes to form a primary key is called a **weak entity set**.

An entity set that has a primary key is termed a **strong entity set**.

As an illustration, consider the entity set *player*, which has two attributes – *name* and *number*. Now, *name* cannot be a primary key as there might be more than one player with the same name. However, *number* is unique for a particular team, but different teams have the same player *number*. For example, there



might be player A with number 1 of team X, and player A with number 1 of team Y. So, the entity set *player* cannot be complete without being somehow associated with the entity set *team*.

As another illustration, consider the entity set *payment*, which has the three attributes: *payment-number*, *payment-date*, and *payment-amount*. Payment numbers are typically sequential numbers, starting from 1, generated separately for each loan payment. Thus, although each payment entity is distinct, payments for different loans may share the same payment number. Thus, this entity set does not have a primary key; it is a weak entity set.

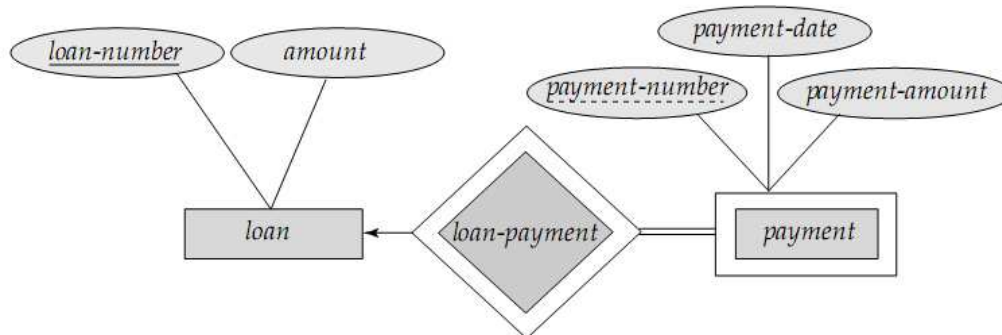


Figure: E-R diagram with a weak entity set.

- For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying** or **owner entity set**.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set. The identifying entity set is said to **own** the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- The identifying relationship is *many to one* from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is *total*.
- In our example, the identifying entity set for *payment* is *loan*, and a relationship *loan-payment* that associates payment entities with their corresponding loan entities is the identifying relationship.

#### Discriminator / Partial Key of a Weak Entity Set

Although a weak entity set does not have a primary key, we nevertheless need a means of distinguishing among all those entities in the weak entity set that depend on one particular strong entity. The **discriminator** or **partial key** of a weak entity set is a set of attributes that allows this distinction to be made.

For example, the discriminator of the weak entity set *payment* is the attribute *payment-number*, since, for each loan, a payment number uniquely identifies one single payment for that loan.

#### How the primary key of a weak entity set is formed

The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator.

In the case of the entity set *payment*, its primary key is {*loan-number*, *payment-number*}, where *loan-number* is the primary key of the identifying entity set, namely *loan*, and *payment-number* distinguishes payment entities within the same loan.

#### Placement of descriptive attributes in a weak entity set

The identifying relationship set should have no descriptive attributes, since any required attributes can be associated with the weak entity set.

#### Participation of weak entity sets in relationships

- A weak entity set can participate in relationships other than the identifying relationship.

For instance, the *payment* entity could participate in a relationship with the *account* entity set, identifying the account from which the payment was made.

- A weak entity set may participate as owner in an identifying relationship with another weak entity set.
- It is also possible to have a weak entity set with more than one identifying entity set.

A particular weak entity would then be identified by a combination of entities, one from each identifying entity set.

The primary key of the weak entity set would consist of the union of the primary keys of the identifying entity sets, plus the discriminator of the weak entity set.

### Weak entity sets – should they be designed as multivalued attributes?

In some cases, the database designer may choose to express a weak entity set as a multivalued composite attribute of the owner entity set.

In our example, this alternative would require that the entity set *loan* have a multivalued, composite attribute *payment*, consisting of *payment-number*, *payment-date*, and *payment-amount*.

A weak entity set may be more appropriately modeled as an attribute if it participates in only the identifying relationship, and if it has few attributes.

Conversely, a weak-entity set representation will more aptly model a situation where the set participates in relationships other than the identifying relationship, and where the weak entity set has several attributes.

## Specialization and Generalization

### Specialization

The process of designating subgroupings within an entity set is called **specialization**.

For example, consider an entity set *person*, with attributes *name*, *street* and *city*. A person may be further classified as a *customer* or an *employee*. Each of these person types is described by a set of attributes that includes all the attributes of entity set *person* plus possibly additional attributes. For example, *customer* entities may be described further by the attribute *customer-id*, whereas *employee* entities may be described further by the attributes *employee-id* and *salary*. The specialization of *person* allows us to distinguish among persons according to whether they are *employees* or *customers*.

### Generalization

There are similarities between the *customer* entity set and the *employee* entity set in the sense that they have several attributes in common. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets. In our example, *person* is the higher-level entity set and *customer* and *employee* are lower-level entity sets. Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The *person* entity set is the superclass of the *customer* and *employee* subclasses.

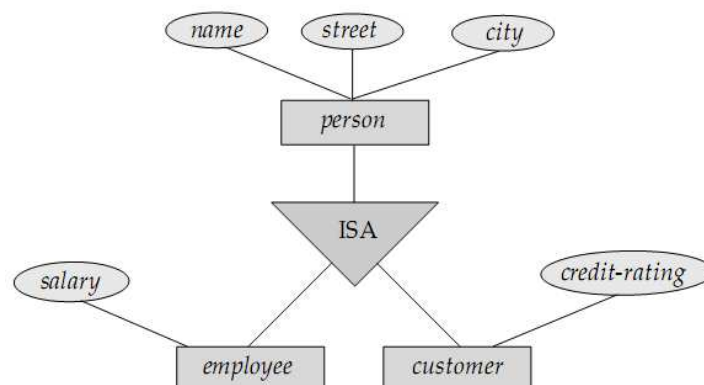
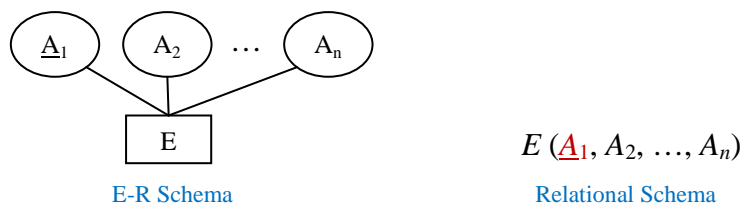


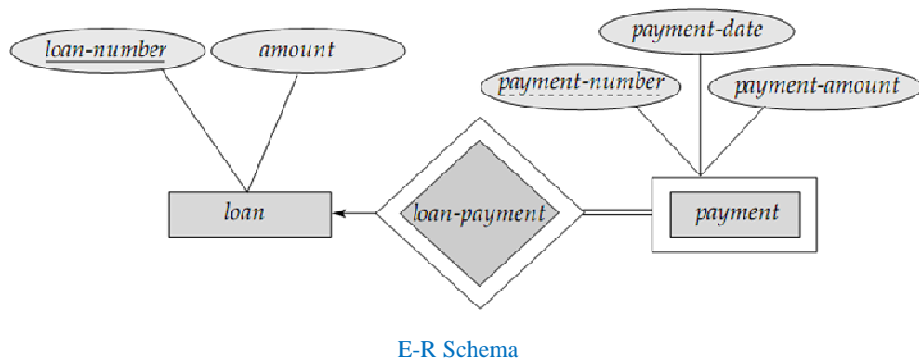
Figure: Specialization and Generalization. [Note: ISA means “is a”]

## Reduction of E-R Schemas to Tables

### Strong Entity Sets

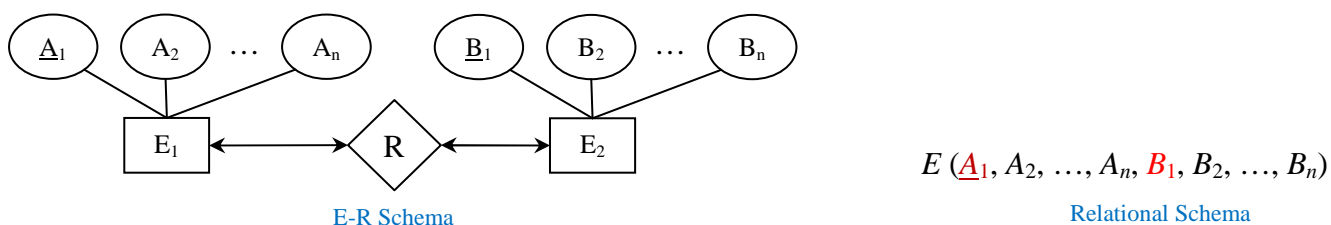


### Weak Entity Sets

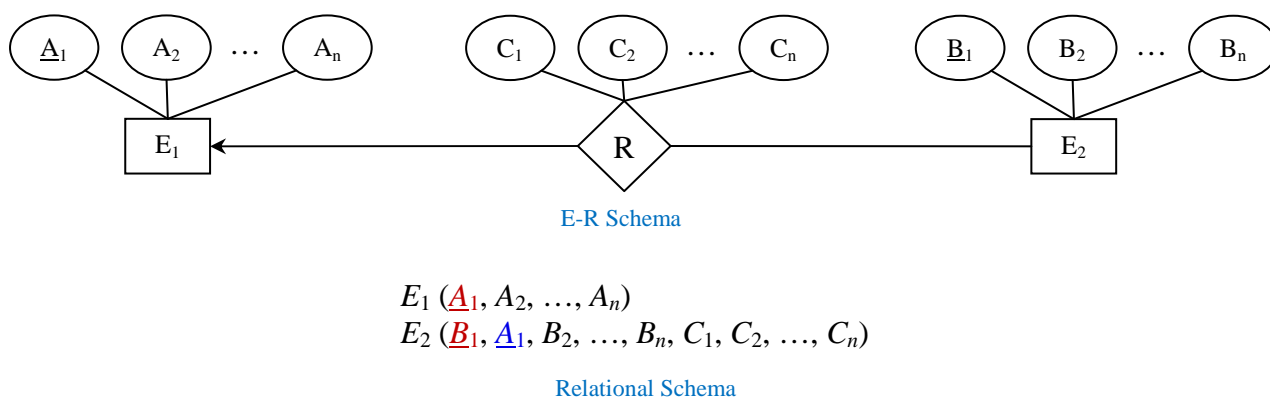


### Relationship Sets

#### One-to-One Relationship



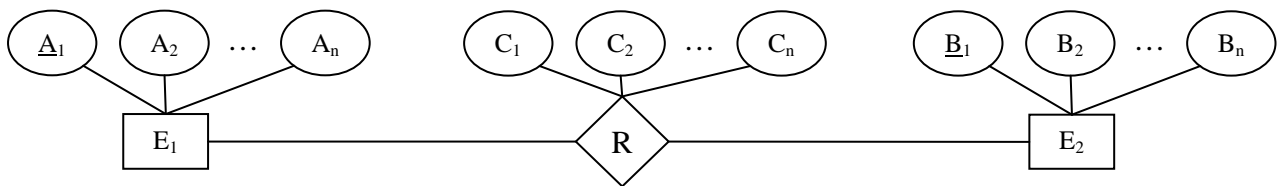
#### One-to-Many Relationship



#### Many-to-One Relationship

Just the reverse of one-to-many.

## Many-to-Many Relationship

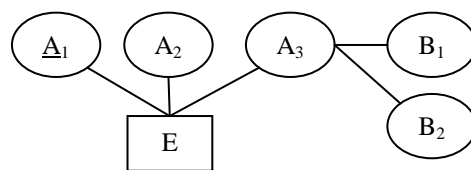


E-R Schema

$E_1 (\underline{A_1}, A_2, \dots, A_n)$   
 $E_2 (\underline{B_1}, B_2, \dots, B_n)$   
 $R (\underline{A_1}, \underline{B_1}, C_1, C_2, \dots, C_n)$

Relational Schema

## Composite Attributes

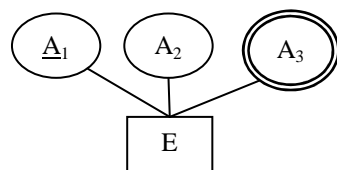


E-R Schema

$E (\underline{A_1}, A_2, B_1, B_2)$

Relational Schema

## Multivalued Attributes

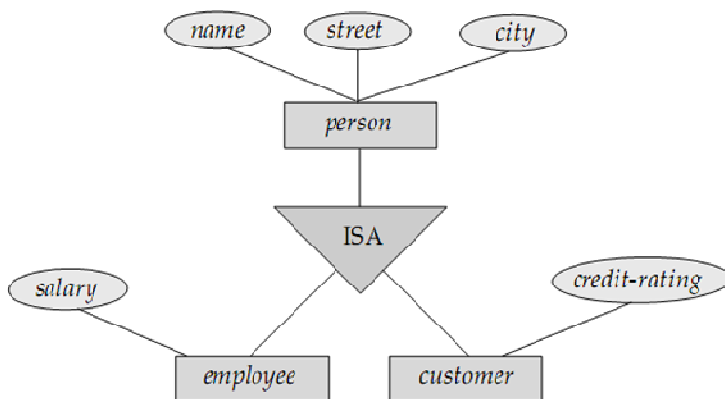


E-R Schema

$E (\underline{A_1}, A_2)$   
 $E\_A (A_3, \underline{A_1})$

Relational Schema

## Generalization



E-R Schema

$person (\underline{person-id}, name, street, city)$   
 $employee (\underline{person-id}, salary)$   
 $customer (\underline{person-id}, credit-rating)$

Relational Schema

## E-R Diagram Symbols at a Glance

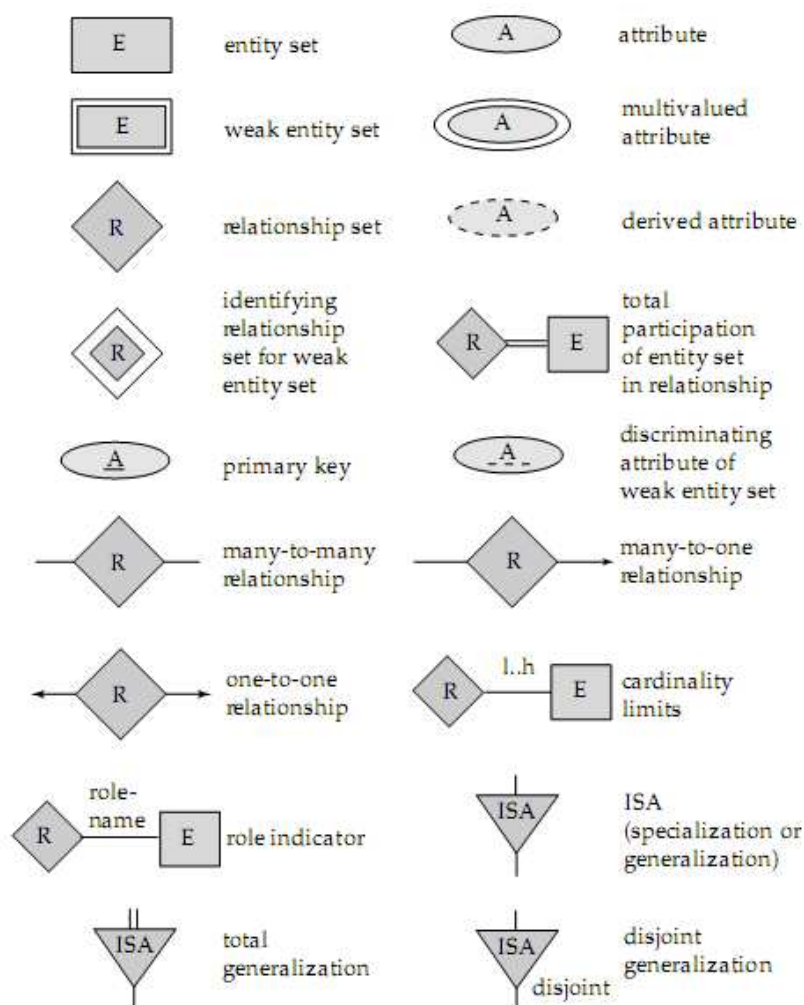


Figure: Symbols used in E-R diagram.

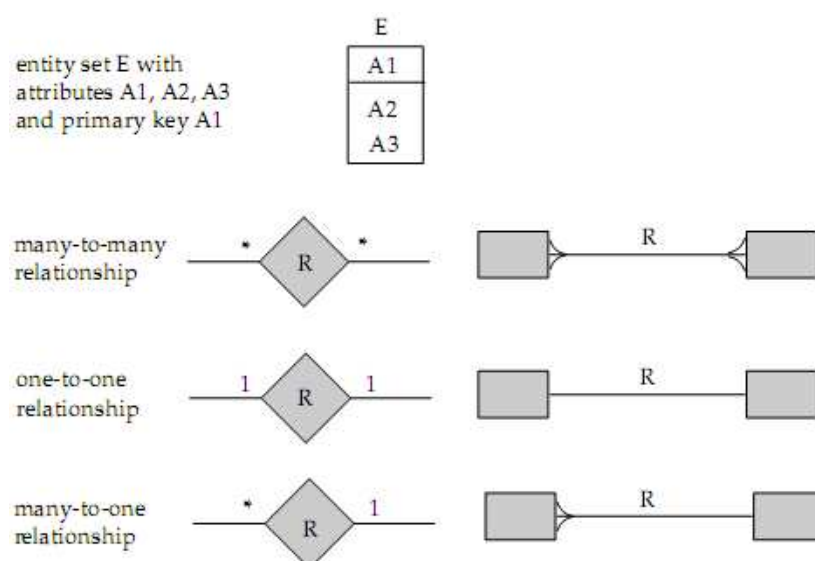


Figure: Alternative E-R notations.

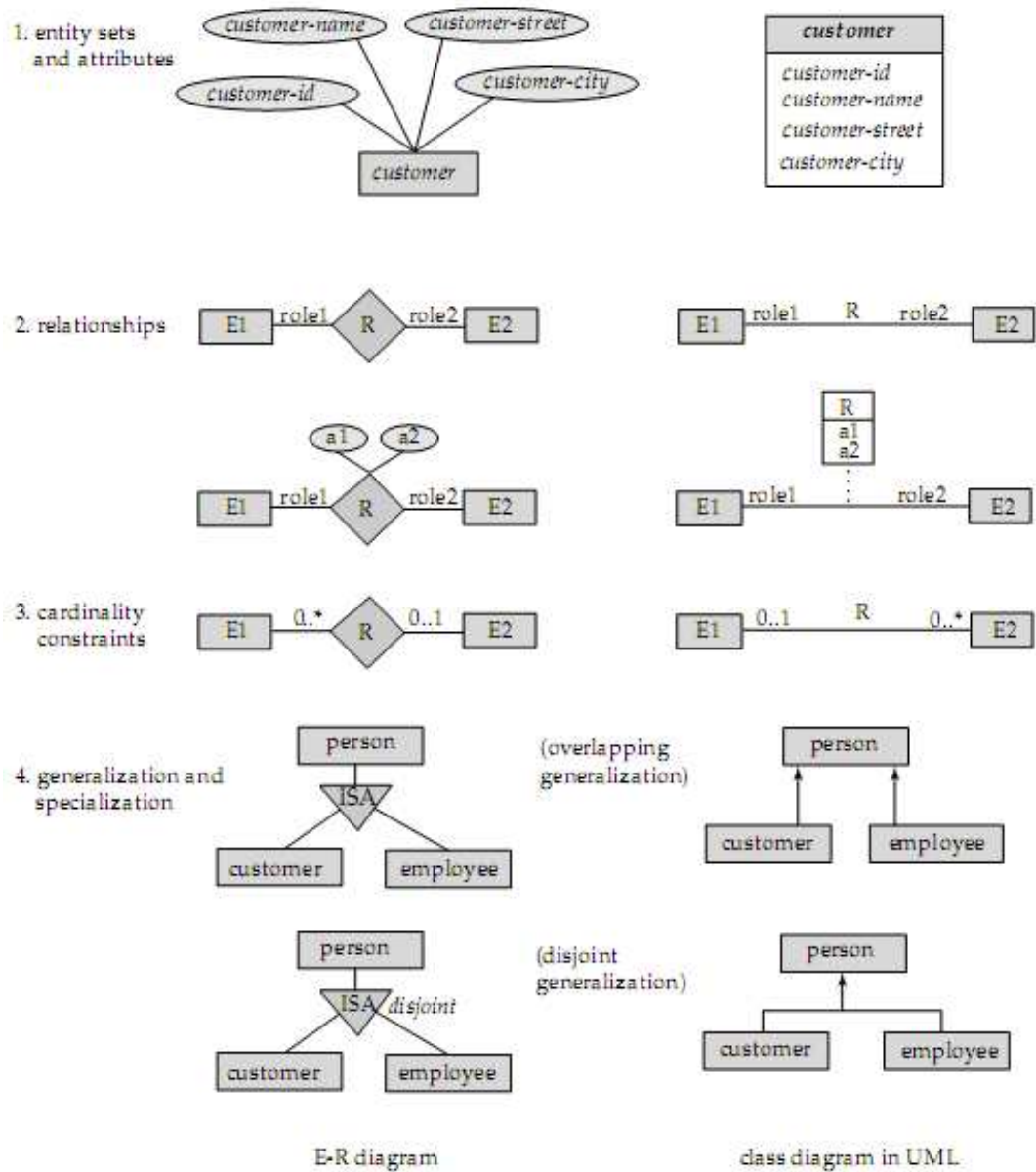


Figure: Symbols used in the UML class diagram notation.

# DML: Data-Manipulation Language

So far we've seen how to use table manipulation commands [*sorry, we couldn't include the commands due to lack of time... ☹*]. Now we'll observe how to put data into tables, retrieve data from tables and manipulate these data.

## DML (Data-Manipulation Language): What it is

A *data-manipulation language* is a language that enables users to access or manipulate data as organized by the appropriate data model.

Data-manipulation is

- The retrieval of information stored in the database
- The insertion of new information into the database
- The deletion of information from the database
- The modification of information stored in the database

## Query Language

A *query* is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a *query language*. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

## The SELECT Operation / Statement

The SELECT statement is used to retrieve information from the database according to provided conditions.

To demonstrate the use of SELECT statement, let's assume the following two relation instances.

customer_name	customer_street	customer_city
Somebody	Mirpur Road	Dhaka
Somebody	Aga Kha Road	Bogra
Anybody	XYZ Road	Khagrachhori
Nobody	Mirpur Road	Dhaka

*customer*

account_number	customer_name	balance
A-101	Anybody	1000
A-102	Anybody	1500
A-103	Somebody	2000
A-104	Nobody	2500

*account*

**Q1. From the *customer* table, find out the information of all the customers who live in Dhaka.**

```
select * from customer where customer_city = 'Dhaka';
```

**Output:**

customer_name	customer_street	customer_city
Somebody	Mirpur Road	Dhaka
Nobody	Mirpur Road	Dhaka

**Q2. From the *customer* table, find out the information of all the customers who live in either Dhaka or Bogra.**

```
select * from customer where customer_city = 'Dhaka' or customer_city = 'Bogra';
```

**Output:**

customer_name	customer_street	customer_city
Somebody	Mirpur Road	Dhaka
Somebody	Aga Kha Road	Bogra
Nobody	Mirpur Road	Dhaka



**Q3. Find out the customers along with their account numbers who have account balances of at least 2000 taka.**

```
select customer_name, account_number from account where balance >= 2000;
```

**Output:**

customer_name	account_number
Somebody	A-103
Nobody	A-104

From the above three queries, we can observe the general structure for the `select` statement:

```
SELECT attribute_1, attribute_2, ... , attribute_N
FROM table
WHERE expression
```

- The *attributes* after the **SELECT** phrase are those attributes which we want to display at the output result.  
If an asterisk (\*) is placed after the select phrase instead of attribute names, it would mean that we want all the attributes at the output.
- The *table name* after the **FROM** clause refers to the relation from which we want to perform the query.
- The *expressions* after the **WHERE** clause contains the actual query. The expressions may contain the following operators:

**Comparison operators:** <, <=, >, >=, =, <>

**Logical connectives:** AND, OR, NOT

**Simplified operators:** *this* BETWEEN *some\_value* AND *some\_other\_value*,  
*this* NOT BETWEEN *some\_value* AND *some\_other\_value*

[These are the substitutes for:

*this* <= *some\_value* AND *this* >= *some\_other\_value*]

**Q4. From the following table named *result*, find out the students (along with their marks) who have scored more than 80 marks, and display them in the descending order of their marks.**

student_name	marks
A	55
B	90
C	40
D	80
E	85
F	95
G	82

```
select student_name, marks from result where marks > 80 order by marks desc;
```

Or, as the table contains only these two fields, we can write:

```
select * from result where marks > 80 order by marks desc;
```

**Output:**

student_name	marks
F	95
B	90
E	85
G	82

**Q5. From the above table in Q4, we want to find out the top-most 3 students (to give them prizes for obtaining the 1<sup>st</sup>, 2<sup>nd</sup> and 3<sup>rd</sup> places). How would you find them?**

```
select student_name from result order by marks desc limit 3;
```

**Output:**

student_name
F
B
E

**Q6. From the table in Q4, we want to find out the student who stood 2<sup>nd</sup>. How would you find him?**

```
select student_name from result order by marks desc limit 1, 1;
```

**Output:**

student_name
B

**Q7. From the table in Q4, we want to find out *how many* students scored  $\geq 80$ . How would you find it?**

```
select count(*) from result where marks >= 80;
```

**Output:**

count(*)
5

**Q8. We want to know how the results of the students (in Q4) would look like if we were to give away 5 marks as grace. How would you find it?**

```
select student_name, marks + 5 from result;
```

**Output:**

student_name	marks + 5
A	60
B	95
C	45
D	85
E	90
F	100
G	87

From the queries Q4 to Q8, we can observe the general structure for the `select` statement:

```
SELECT attribute(s) and/or function(s)
FROM table
WHERE expression
ORDER BY attribute(s)
LIMIT number_of_rows / starting_row_index, number_of_rows
```

## Relational Algebra

We've seen various query statements for retrieving information according to our needs. But did you ever wonder how these queries are executed by the computer? Put in other words, how does the computer interpret these queries? To get the answer, we have to go back into 1970; we need to know the history of how the DBMS was born.

*History omitted for lack of time... ☹*

### Fundamental Relational Algebra Operations

These operations are called fundamental as they are sufficient for expressing any relational algebra query, although some common queries become lengthy to express.

1. Selection (unary)
2. Projection (unary)
3. Rename (unary)
4. Union (unary)
5. Set-difference (binary)
6. Cartesian product (binary)

#### The Selection Operation

**Q9. From the *customer* table in Q1, find out the information of all the customers who live in Dhaka.**

**SQL:** `select * from customer where customer_city = 'Dhaka';`

**RA:**  $\sigma_{\text{customer\_city} = \text{"Dhaka"}}(\text{customer})$

So, the general structure for the select operation of relational algebra is:

$$\sigma_{\text{expression}}(\text{relation})$$

#### The Projection Operation

**Q10. From the *customer* table in Q1, find out the names and street addresses of all the customers.**

**SQL:** `select customer_name, customer_street from customer`

**RA:**  $\Pi_{\text{customer\_name}, \text{customer\_city}}(\text{customer})$

So, the general structure for the projection operation of relational algebra is:

$$\Pi_{\text{attributes}}(\text{relation})$$

#### Composition of the Relational Operations

**Q11. From the *customer* table in Q1, find out the names of all the customers who live in either Dhaka or Bogra.**

**SQL:** `select customer_name  
from customer  
where customer_city = 'Dhaka' or customer_city = 'Bogra';`

**RA:**  $\Pi_{\text{customer\_name}}(\sigma_{\text{customer\_city} = \text{"Dhaka"} \vee \text{customer\_city} = \text{"Bogra"}}(\text{customer}))$

### The Union Operation

**Q12. From the tables below, find the names of all the customers who have either an account or a loan or both.**

customer_name	loan_number
Somebody	L-101
Somebody	L-102
Anybody	L-103

*borrower*

customer_name	account_number
Anybody	A-102
Somebody	A-103
Nobody	A-104

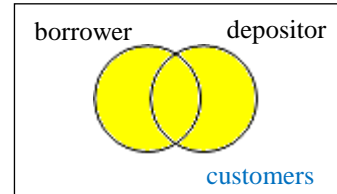
*depositor*

**SQL:** `select customer_name from borrower  
union  
select customer_name from depositor;`

**RA:**  $\Pi_{\text{customer\_name}}(\text{borrower}) \cup \Pi_{\text{customer\_name}}(\text{depositor})$

**Output:**

customer_name
Somebody
Anybody
Nobody



### The Set-Difference Operation

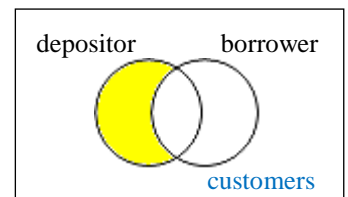
**Q13. From the tables in Q12, find all the customers who have accounts but no loans.**

**SQL:** `select customer_name from depositor  
minus (or except)  
select customer_name from borrower;`

**RA:**  $\Pi_{\text{customer\_name}}(\text{depositor}) - \Pi_{\text{customer\_name}}(\text{borrower})$

**Output:**

customer_name
Nobody



### Notes on the union and set-difference operations:

- Unlike the **SELECT** clause, the **UNION**, **MINUS** and **EXCEPT** clauses will eliminate duplicate values.
- For a union operation  $r_1 \cup r_2$  or a set-difference operation  $r_1 - r_2$  to be valid, two conditions must hold:
  1. The relation  $r_1$  and  $r_2$  must be of the same arity; i.e., they must have the same number of attributes.
  2. For all  $i$ , the domains of  $i$ -th attribute of  $r_1$  and the domains of  $i$ -th attribute of  $r_2$  must be the same.

Note that  $r_1$  and  $r_2$  can be, in general, temporary relations that are the result of relational algebra expressions.

- MySQL does not support the **MINUS** or **EXCEPT** clauses. Rather, set-difference is performed via outer join operations. We'll learn about outer join operation later.

### The Cartesian-Product Operation

**Q14. From the tables below, find all the customers who have a loan at Dhaka branch.**

loan_number	branch_name	amount
L-101	Dhaka	1000
L-103	Khulna	2000

*loan*

customer_name	loan_number
Somebody	L-101
Nobody	L-103
Anybody	L-103

*borrower*

The problem is, *customer\_name* is in one table, whereas *branch\_name* is in another table. Then how would we perform a *select* operation? Well, the good news is, we have the *loan\_number* in common. So, we can solve the problem by the following algorithm: for each *loan\_number* in *loan* table where *branch\_name* is Dhaka, find out the *customer\_name* corresponding to that *loan\_number*.

**Step 1:** Find the *loan\_numbers* from *loan* table where *branch\_name* is Dhaka.

**SQL:** `select loan_number from loan where branch_name = 'Dhaka';`

**RA:**  $\Pi_{\text{loan\_number}} (\sigma_{\text{branch\_name} = \text{"Dhaka"}} (\text{loan}))$

**Step 2:** Find the *customer\_names* from *borrower* table where *loan\_numbers* are equal to the *loan\_numbers* derived from step 1.

**SQL:** `select customer_name from borrower  
where loan_number = any_loan_number_in_the_relation_found_from_step_1;`

**RA:**  $\Pi_{\text{customer\_name}} (\sigma_{\text{loan\_number} = \text{any\_loan\_number\_in\_the\_relation\_found\_from\_step\_1}} (\text{borrower}))$

But the problem is, how can we express *any\_loan\_number\_in\_the\_relation\_found\_from\_step\_1*? The problem arises as that expression is *a set of values*, not a *single* value that we can compare with another value.

The solution comes with the *Cartesian-product* (also known as *cross-product*) operation. From set theory, we know that,

if  $A = \{1, 2, x\}$

and  $B = \{x, b\}$

then  $A \times B = \{(1, x), (1, b), (2, x), (2, b), (x, x), (a, b)\}$

If our requirement is to get the common value (in this case, *x*), then we can cross-product the sets and find our required value by checking where both parts of a pair are the same (in this case, *(x, x)*).

Similarly, if we cross-product the above two relations, we will get the following relation:

loan_number	branch_name	amount	customer_name	loan_number
L-101	Dhaka	1000	Somebody	L-101
L-101	Dhaka	1000	Nobody	L-103
L-101	Dhaka	1000	Anybody	L-103
L-103	Khulna	2000	Somebody	L-101
L-103	Khulna	2000	Nobody	L-103
L-103	Khulna	2000	Anybody	L-103

The above relation can be obtained by the following query or relational algebra expression:

**SQL:** `select * from loan, borrower;`

**RA:**  $\sigma (\text{loan} \times \text{borrower})$

Now, let's filter-out the rows where both the *loan\_numbers* are the same. Then we'll get the following relation:

loan_number	branch_name	amount	customer_name	loan_number
L-101	Dhaka	1000	Somebody	L-101
L-103	Khulna	2000	Anybody	L-103

The above filtration (along with the cross-product) can be done by the following query or RA expression:

**SQL:** `select * from loan, borrower where loan_number = loan_number;`

**RA:**  $\sigma_{\text{loan\_number} = \text{loan\_number}} (\text{loan} \times \text{borrower})$

But the DBMS system will complain if we try to execute the above query. How is the program supposed to know that these two *loan\_numbers* are to be checked for equality from both the sides and not from only the left side or only the right side?

That's why we need to specify the attribute names along with their table names:

**SQL:** `select * from loan, borrower  
where loan.loan_number = borrower.loan_number;`

**RA:**  $\sigma_{\text{loan.loan\_number} = \text{borrower.loan\_number}} (\text{loan} \times \text{borrower})$

Fine. Now, to get our desired result, we need to filter-out the records which contain 'Dhaka' as the *branch\_name*. Then we'll get the following relation:

loan_number	branch_name	amount	customer_name	loan_number
L-101	Dhaka	1000	Somebody	L-101

The above filtration (along with the cross-product and the previous filtration) can be done by the following query or RA expression:

**SQL:** `select * from loan, borrower  
where loan.loan_number = borrower.loan_number and branch_name = 'Dhaka';`

**RA:**  $\sigma_{\text{branch\_name} = \text{'Dhaka'}} (\sigma_{\text{loan.loan\_number} = \text{borrower.loan\_number}} (\text{loan} \times \text{borrower}))$

Finally, we only need the *customer\_names*. So, the final query for performing all these steps would be:

**SQL:** `select customer_name from loan, borrower  
where loan.loan_number = borrower.loan_number and branch_name = 'Dhaka';`

**RA:**  $\Pi_{\text{customer\_name}} (\sigma_{\text{branch\_name} = \text{'Dhaka'}} (\sigma_{\text{loan.loan\_number} = \text{borrower.loan\_number}} (\text{loan} \times \text{borrower})))$

And the final output would be:

customer_name
Somebody

### The Rename Operation

Let's go through an example.

**Q15. From the table named *result* below, find the highest marks.**

student_name	marks
A	60
B	70
C	80

**SQL:** `select max(marks) from result;`

But how would you express this in relational algebra? Solution:

**Step 1:** Compute the Cartesian-product relation:

student_name	marks	student_name	marks
A	60	A	60
A	60	B	70
A	60	C	80
B	70	A	60

B	70	B	70
B	70	C	80
C	80	A	60
C	80	B	70
C	80	C	80

**RA:**  $\sigma$  (result  $\times$  result)

**Step 2:** Filter out the tuples in which the marks on the left side are *lesser* than the marks on the right side (or vice-versa):

student_name	marks	student_name	marks
A	60	B	70
A	60	C	80
B	70	C	80

**RA:**  $\sigma_{\text{marks} < \text{marks}}$  (result  $\times$  result)

Now the problem is obvious. How can we distinguish the two *marks*? Using the relation name (i.e., result.marks = result.marks) can't remove the ambiguity. In this case, we have to rename *one* or *both* the relations. It can be done as below (renaming only one of the relations into *res*):

**RA:**  $\sigma_{\text{result.marks} < \text{res.marks}}$  (result  $\times \rho_{\text{res}}$  (result))

**SQL:** `select * from result, result as res where result.marks < res.marks;`

The general structure of the rename operation is as follows:

$\rho_{\text{new-name}}(\text{old-name})$

**Step 3:** Take only the *marks* field:

marks
60
70

**RA:**  $\Pi_{\text{result.marks}} (\sigma_{\text{result.marks} < \text{res.marks}} (\text{result} \times \rho_{\text{res}} (\text{result})))$

**SQL:** `select distinct result.marks from result, result as res where result.marks < res.marks;`

Note that we're using the keyword **DISTINCT** to get unique values. RA automatically provides distinct values, whereas SQL `select` query doesn't.

**Step 4:** Take the set-difference between the marks attribute in result relation and this temporary relation:

**RA:**  $\Pi_{\text{marks}} (\text{result}) - \Pi_{\text{marks}} (\sigma_{\text{result.marks} < \text{res.marks}} (\text{result} \times \rho_{\text{res}} (\text{result})))$

**SQL:** `select marks from result minus (select distinct result.marks from result, result as res where result.marks < res.marks);`

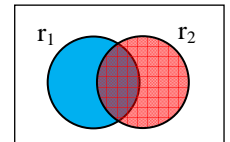
## Additional Relational Algebra Operations

These operations do not add any power to the algebra, but they simplify common queries.

1. Set-intersection (binary)
2. Natural join (binary)
3. Division (binary)
4. Assignment (unary)



### The Set-intersection Operation



**Note:**  $r_1 \cap r_2 = r_1 - (r_1 - r_2) = r_2 - (r_2 - r_1)$

**Q16. From the table in Q12, find all the customers who have both an account and a loan.**

**RA:**  $\Pi_{\text{customer\_name}}(\text{borrower}) \cap \Pi_{\text{customer\_name}}(\text{depositor})$

**SQL:** `select customer_name from borrower  
intersect  
select customer_name from depositor;`

Note that MySQL doesn't support the **INTERSECT** operation. So, the following query is used instead:

**SQL:** `select distinct borrower.customer_name from borrower, depositor  
where borrower.customer_name = depositor.customer_name;`

**OR,** `select distinct customer_name from borrower  
where customer_name in (select customer_name from depositor);`

**OR,** `select distinct borrower.customer_name from borrower  
where exists  
(select * from depositor  
where borrower.customer_name = depositor.customer_name);`

**RA:**  $\Pi_{\text{borrower.customer\_name}}(\sigma_{\text{borrower.customer\_name} = \text{depositor.customer\_name}}(\text{borrower} \times \text{depositor}))$

**Output:**

customer_name
Somebody
Anybody

### The Natural Join Operation

This operation simplifies the Cartesian-product operation in that it automatically applies the conditions that common fields are to be matched. Also, the common fields are displayed as a single field at output.

Thus, the operation

$\Pi_{\text{borrower.customer\_name}}(\sigma_{\text{borrower.customer\_name} = \text{depositor.customer\_name}}(\text{borrower} \times \text{depositor}))$

becomes

$\Pi_{\text{borrower.customer\_name}}(\text{borrower} \bowtie \text{depositor})$

**Q17. From the table in Q12, find the customers (along with all their information) who have both an account and a loan.**

**SQL:** `select * from borrower natural join depositor;`

**RA:**  $\sigma(\text{borrower} \bowtie \text{depositor})$

**Output:**

customer_name	loan_number	account_number
Somebody	L-101	A-103
Somebody	L-102	A-103
Anybody	L-103	A-102

## Extended Relational Algebra Operations

These operations simply extend the basic operations in several ways.

1. **Outer join** – allows relational algebra expressions to deal with null values by extending join operation
2. **Generalized Projection** – allows arithmetic operations as part of projection
3. **Aggregate Functions** – sum, avg, count, min, max

### The Outer Join Operation

Consider the following two tables:

id	name
1	A
2	B

*x*

id	name
1	A
2	C

*y*

Both the tables have a common *name* – **A**. However, table *x* has a *name* (**B**) which table *y* doesn't have. On the other hand, table *y* has a *name* (**C**) that table *x* doesn't have.

If we cross-join the tables, we'll get:

id	name	id	name
1	A	1	A
1	A	2	C
2	B	1	A
2	B	2	C

Now, if we use natural join, we'll come up with only the first row. But what if we want to keep all the records from the left table or right table or both anyway? For example, we want to know which records in table *x* doesn't match with records in table *y*?

Outer join answers to this query. There are three types of outer joins:

#### Left Outer Join

Keeps all the records from the left table and assigns null values to those records which do not match on the right table.

id	name	id	name
1	A	1	A
2	B	null	null

#### Right Outer Join

Keeps all the records from the right table and assigns null values to those records which do not match on the left table.

id	name	id	name
1	A	1	A
null	null	2	C

#### Full Outer Join

Keeps all the records from the both the tables and assigns null values to those records which do not match.

id	name	id	name
1	A	1	A
2	B	null	null
null	null	2	C

**Q18. From the tables in Q12, find all the customers who have accounts but no loans.**

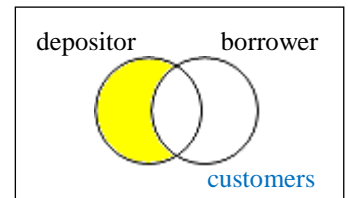
**SQL:** `select customer_name  
from depositor left outer join borrower  
on depositor.customer_name = borrower.customer_name  
where loan_number is null`

**OR,** `select customer_name  
from depositor left outer join borrower  
using (customer_name)  
where loan_number is null`

**RA:**  $\Pi_{customer\_name} (\sigma_{loan\_number = null} (depositor \bowtie borrower))$

**Output:**

customer_name
Nobody



**Q19. From the table in Q12, find the names of all the customers who have either an account or a loan but not both.**

**SQL:** `select customer_name  
from depositor full outer join borrower  
on depositor.customer_name = borrower.customer_name  
where loan_number is null or account_number is null`

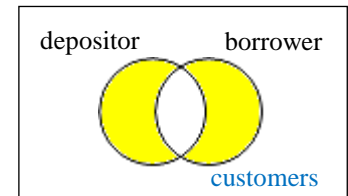
**OR,** `select customer_name  
from depositor full outer join borrower  
using (customer_name)  
where loan_number is null or account_number is null`

**OR,** `select customer_name  
from depositor natural full outer join borrower  
where loan_number is null or account_number is null`

**RA:**  $\Pi_{customer\_name} (\sigma_{loan\_number = null \vee account\_number = null} (depositor \bowtie borrower))$

**Output:**

customer_name
Somebody
Anybody
Nobody



### Generalized Projection

This operation extends the projection operation by allowing arithmetic functions to be used in the projection list.

As an example, let's revisit query Q8: we want to know how the results of the students (in Q4) would look like if we were to give away 5 marks as grace. How would we find it?

**SQL:** `select student_name, marks + 5 from result;`

**RA:**  $\Pi_{student\_name, marks + 5} (result)$

### Aggregate Functions

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

For example, consider the following *employee* table:

employee_name	branch_name	branch_city	salary
A	DU	Dhaka	1000
B	DU	Dhaka	2000
C	BUET	Dhaka	3000
D	KUET	Khulna	4000
E	KU	Khulna	5000
F	RU	Rajshahi	6000

**Q20. Find the number of branches appearing in the *employee* relation.**

**SQL:** `select count(distinct branch_name) from employee;`

**RA:** `⌞ count-distinct(branch_name) (employee)`

**Output:**

count(distinct branch_name)
5

**Q21. Find the total salary of all employees at *each* branch of the bank.**

**SQL:** `select branch_name, sum(salary) from employee group by branch_name;`

**RA:** `branch_name ⌞ sum(salary) (employee)`

**Output:**

branch_name	sum(salary)
BUET	3000
DU	3000
KU	5000
KUET	4000
RU	6000

Note that the column header for the second column is not very meaningful. To give it a meaningful name, e.g., *total\_salary*, we can rewrite the above query as below:

**SQL:** `select branch_name, sum(salary) as total_salary from employee group by branch_name;`

**RA:** `branch_name ⌞ sum(salary) as total_salary (employee)`

**Output:**

branch_name	total_salary
BUET	3000
DU	3000
KU	5000
KUET	4000
RU	6000

**Q22. Find branch city, branch name wise total salary, average salary and also number of employees.**

**SQL:** `select branch_city, branch_name, sum(salary), avg(salary), count(employee_name) from employee group by branch_city, branch_name;`

**RA:** `branch_city, branch_name ⌞ sum(salary), avg(salary), count(salary) (employee)`

**Output:**

branch_city	branch_name	sum(salary)	avg(salary)	count(employee_name)
Dhaka	BUET	3000	3000.0000	1
Dhaka	DU	3000	1500.0000	2
Khulna	KU	5000	5000.0000	1

Khulna	KUET	4000	4000.0000	1
Rajshahi	RU	6000	6000.0000	1

**Q23. From the following table, find the average balance for each customer who lives in Dhaka and has at least two accounts.**

customer_name	customer_city
A	Dhaka
B	Dhaka
C	Khulna
D	Dhaka

*customer*

customer_name	account_no	balance
A	A-101	1000
A	A-102	2000
B	A-103	3000
C	A-104	4000
D	A-105	5000
D	A-106	6000

*account*

Here, we cannot insert both the conditions into the **where** clause, because counting how many accounts a customer has can only be done using an aggregate function (**count**). As aggregate functions operate on groups whereas the **where** clause operates on tuples, these two conditions cannot be put together. For solving this problem, SQL introduces another clause **having** where the conditions to be operated on groups are to be placed.

So how would we write this query?

**Step 1:** Find the customers (with all the other information) who live in Dhaka.

**SQL:** `select * from customer natural join account where customer_city = 'Dhaka';`

This would yield the following relation:

customer_name	customer_city	account_no	balance
A	Dhaka	A-101	1000
A	Dhaka	A-102	2000
B	Dhaka	A-103	3000
D	Dhaka	A-105	5000
D	Dhaka	A-106	6000

**Step 2:** Because we need to find average balance for *each* customer, we need to group the records by customer\_name; but remember that we have to take only those customers who have at least two accounts.

**SQL:** `select * from customer natural join account  
where customer_city = 'Dhaka'  
group by customer_name  
having count(distinct account_no) >= 2;`

This would yield the following relation:

customer_name	customer_city	account_no	balance
A	Dhaka	A-101	1000
D	Dhaka	A-105	5000

**Step 3:** However, we need to find the *average balance* for each customer.

**SQL:** `select customer_name, avg(balance)  
from customer natural join account  
where customer_city = 'Dhaka'  
group by customer_name  
having count(distinct account_no) >= 2;`

Thus, the final output will be:

customer_name	avg(balance)
A	1500.0000
D	5500.0000

## Modification of Database

It addresses how to add, remove or change information in the database. Database modification is expressed by assignment operation,  $\leftarrow$ .

### Insertion

Inserting records in a database is quite easy. We either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Consider the following two tables:

customer	loan_no	branch	amount
A	101	Dhaka	1000
B	102	Rajshahi	2000
C	103	Dhaka	3000

*loan*

customer	account_no	branch	balance

*account*

Now, let's insert a record in the *loan* table:

**SQL:** `insert into loan values ('D', 104, 'Bogra', 4000);`

**RA:**  $\text{loan} \leftarrow \text{loan} \cup \{("D", 104, "Bogra", 4000)\}$

Easy, isn't it? Let's try another example (a bit complex, however). We want to provide a new savings account of Tk. 200 for all the loan holders of Dhaka branch. The query would be as follows:

**SQL:** `insert into account  
select customer, loan_no, branch, 200 from loan where branch = 'Dhaka';`

**RA:**  $\text{account} \leftarrow \text{account} \cup (\Pi_{\text{customer, loan\_no, branch, 200}} (\sigma_{\text{branch} = "Dhaka"}(\text{loan})))$

Note that we're directly projecting 200 to forcefully set the value of balance as 200.

Therefore, we've seen that to insert data to a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. However, two conditions must be met:

1. The attribute values for inserted tuples must be members of the attribute's domain.
2. Tuples inserted must be of the same arity.

### Deletion

Deletion removes the selected tuple/tuples from the database. Only the whole tuples can be deleted, not values of particular attributes.

The SQL syntax for deletion is fairly simple:

`delete from table where expression`

So how does the delete operation work? Let's discover through an example. Consider the following table named *employee*:

employee_name	branch_name	branch_city	salary
A	DU	Dhaka	1000
B	RU	Rajshahi	2000
C	BUET	Dhaka	3000

Suppose we want to delete the employee named *B*. What the `delete` statement does is, it first selects the rows which we want to delete:

$\sigma_{\text{employee\_name} = "B"}(\text{employee})$

So, the *second* row from the table would be selected.

Now, we'll try to get a relation where this selected row doesn't exist. How? Very simple. Just use set difference:

$\text{employee} - \sigma_{\text{employee\_name} = "B"}(\text{employee})$

The above set-difference operation will produce the following relation:

employee_name	branch_name	branch_city	salary
A	DU	Dhaka	1000
C	BUET	Dhaka	3000

Now, we can just replace the original employee table with this new table using the assignment operation:

$$\text{employee} \leftarrow \text{employee} - \sigma_{\text{employee\_name} = \text{"B"}}(\text{employee})$$

Voilà, it's done!

## Update

Update is used to change or modify values of one or more attributes of a tuple/tuples. The SQL syntax for updating is:

```
update table
set attribute_1 = value_1, ..., attribute_n = value_n
where expression
```

So how does the update operation work? Consider the above table again. Suppose we want to change employee **B**'s city from *Rajshahi* to *Comilla*. To do this, we'll first use projection to project the attributes of that row. But instead of projecting *branch\_city*, we'll directly project "*Comilla*". This will forcibly place "*Comilla*" as the attribute's value. The projection is as follows:

$$\Pi_{\text{employee\_name}, \text{branch\_name}, \text{"Comilla"}, \text{salary}}(\text{employee})$$

We'll get a relation containing the following tuple:

employee_name	branch_name	branch_city	salary
B	RU	Comilla	2000

Now, we'll delete the row to be updated (i.e., the row containing *employee\_name* **B**) from the original table:

$$\text{employee} \leftarrow \text{employee} - \sigma_{\text{employee\_name} = \text{"B"}}(\text{employee})$$

The original table should look like the following:

employee_name	branch_name	branch_city	salary
A	DU	Dhaka	1000
C	BUET	Dhaka	3000

Now, let's add our previously projected record to this new table using union operation:

$$\text{employee} \leftarrow \text{employee} \cup (\Pi_{\text{employee\_name}, \text{branch\_name}, \text{"Comilla"}, \text{salary}}(\text{employee}))$$

Thus, in the final *employee* table, the attribute will be updated. Therefore, the steps for updating the above attribute can be expressed as follows:

```
t ← Π_employee_name, branch_name, "Comilla", salary (employee)
employee ← employee - σ_employee_name = "B" (employee)
employee ← employee ∪ (t)
```



# Indexing and Hashing

## The Problem

Many queries reference only a small fraction of records in a file. For example, “find all accounts at Mirpur branch” only returns records from *account* file where *branch\_name* = “Mirpur”. It is inefficient for the system to read every record and to check the *branch\_name* field for the name “Mirpur”.

We should be able to locate these records directly. To allow these forms of access, we design additional structures associated with files. These additional structures are called indices.

## Basic Concepts

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information we are looking for. The words in the index are in sorted order, making it easy to find the word we are looking for.

Card catalogs in libraries works in a similar manner. To find a book by a particular author, we would search in the author catalog, and a card in the catalog tells us where to find the book. To assist us in searching the catalog, the library would keep the cards in alphabetic order by authors, with one card for each author of each book.

Database system indices play the same role as book indices or card catalogs in libraries. For example, to retrieve an account record given the account number, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the account record.

Keeping a sorted list of account numbers would not work well on very large databases with millions of accounts, since the index would itself be very big. So, more sophisticated indexing techniques may be used.

## Types of Indices

There are two basic kinds of indices.

1. **Ordered indices:** indices are based on a sorted ordering of the values.
2. **Hash indices:** indices are based on the values being distributed uniformly across a range of buckets. The bucket to which a value is assigned is determined by a function, called a *hash function*.

## Index Technique Choosing Factors

There are several techniques for both ordered indexing and hashing. No one technique is the best. Each technique is best suited to particular database applications. Each technique must be evaluated on the basis of these factors:

1. **Access types** – Finding records with a *specified value* or a *range of values*.
  2. **Access time** – Time to find a *particular data item* or a *set of items*.
  3. **Insertion time** – Time to find the correct place to insert the item + time to update the index.
  4. **Deletion time** – Time to find the item to be deleted + time to update the index.
  5. **Space overhead** – The additional space occupied by an index structure.
- We may have more than one index or hash function for a file. (The library may have card catalogues by author, subject or title)
  - An attribute or set of attributes used to look up records in a file is called the **search key** (This definition of key differs from that used in super key, candidate key or primary key).

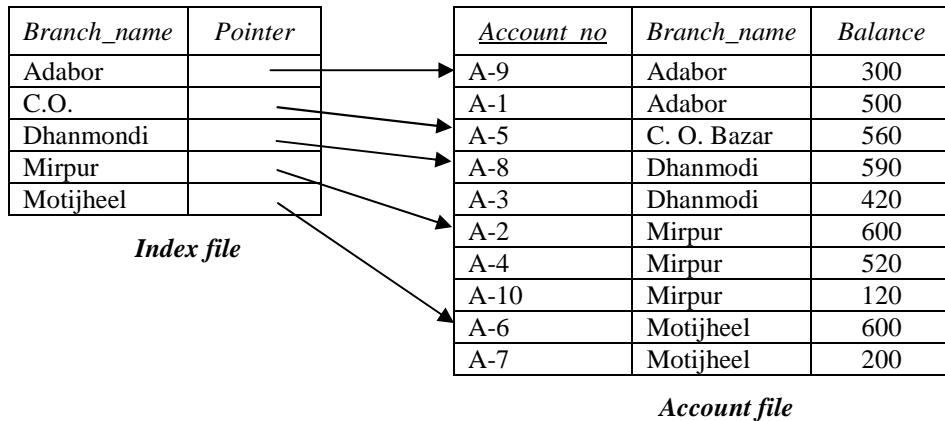
## Ordered Indices

An ordered index stores the values of the search keys in *sorted* order and associates with each search key the records that contain it.

### Primary / Clustering Index

If the file containing the records is sequentially ordered, a **primary index** is an index whose search key also defines the sequential order of the file.

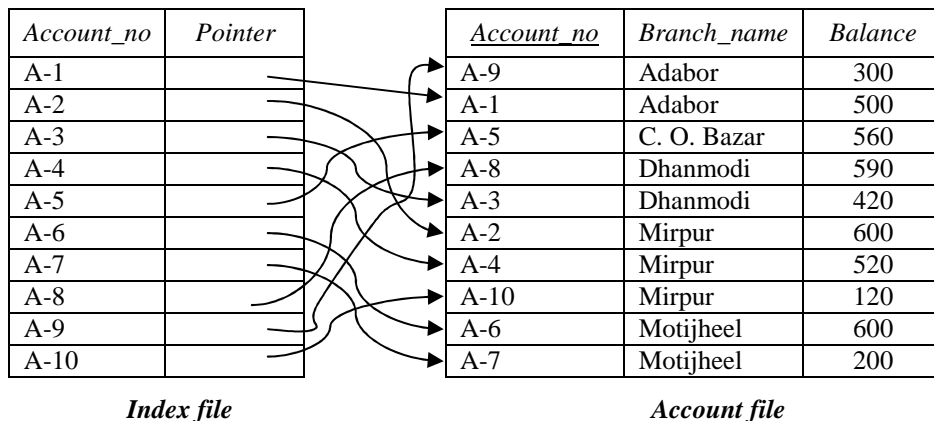
*The search key of a primary index is usually the primary key, but it is not necessarily so.*



**Figure:** A primary index. The Account file is ordered according to *Branch\_name* (which is *not* a primary key). The index file is also ordered according to *Branch\_name*.

### Secondary / Non-Clustering Index

Indices whose search key specifies an order different from the sequential order of the file are called **secondary indices**.



**Figure:** A secondary index. The Account file is ordered according to *Branch\_name*. But the index file is ordered according to *Account\_no*.

### Index-Sequential Files

Files that are ordered sequentially on some search key and have a primary index on that search key are called **index-sequential files**.

### Contents of an index record / entry

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 the record within the block.

### Types of ordered indices

There are two types of ordered indices:

1. Dense Index
2. Sparse Index

### Dense Index

**Dense index** is the index where an index record appears for *every* search-key value in the file.

#### Dense index for primary indices

In a dense primary index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search key-value would be stored sequentially after the first record, since, because the index is a primary one, records are sorted on the same search key.

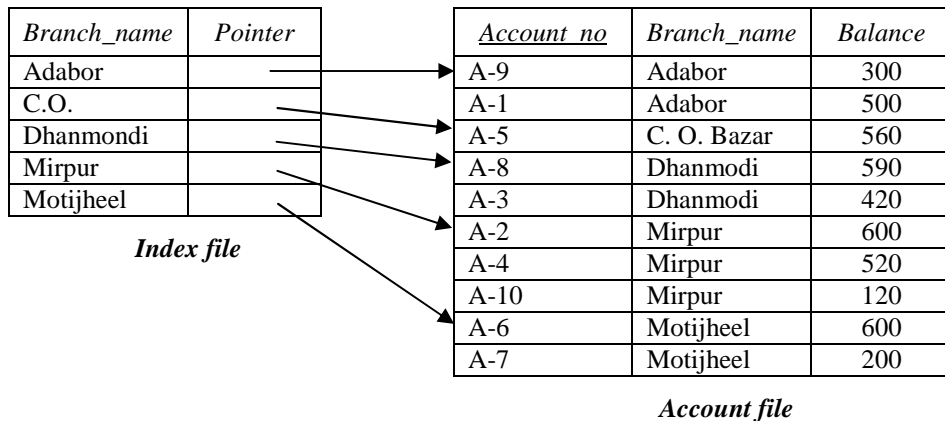


Figure: Dense index for a primary index.

#### Another implementation of dense indices

A dense index can also be implemented by storing a list of pointers to all records with the same search-key value. *Doing so is not essential for primary indices.*

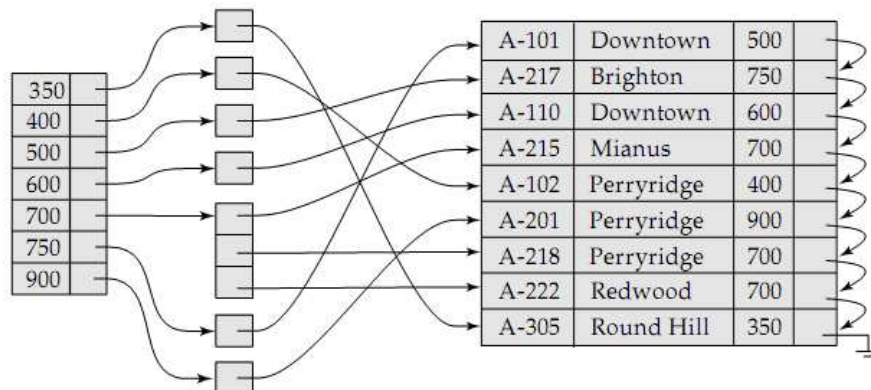


Figure: Another implementation of dense index.

### Sparse Index

**Sparse index** is the index where an index record appears for only *some* of the search-key values in the file.

As is true in dense indices, each index record contains a search-key value and a pointer to the first data record with that search-key value.

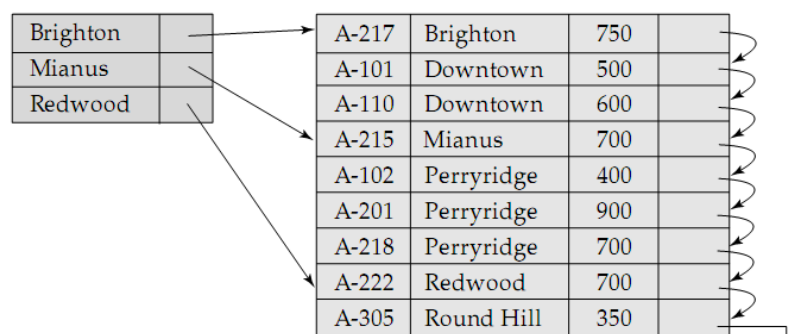


Figure: Sparse index.

To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

## Comparative Analysis of Dense and Sparse Index

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.

### A good trade-off

There is a trade-off that the system designer must make between *access time* and *space overhead*. Although the decision regarding this trade-off depends on the specific application, *a good compromise is to have a sparse index with one index entry per block*.

### Why this trade-off is good

The dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible.

Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block, we minimize block accesses while keeping the size of the index (and thus, our space overhead) as small as possible.

## Multi-Level Indices

### The problem with single-level indices

Even if we use a sparse index, the index itself may become too large for efficient processing. It is not unreasonable, in practice, to have a file with 100,000 records, with 10 records stored in each block. If we have one index record per block, the index has 10,000 records. Index records are smaller than data records, so let us assume that 100 index records fit on a block. Thus, our index occupies 100 blocks. *So, how can we minimize disk access?*

- If there are no overflow blocks in the index, we can use binary search. If there are  $B$  blocks, this will read as many as  $\lceil \log_2(B) \rceil$  blocks (as many as 7 for our 100 blocks).
- If index has overflow blocks, then sequential search is typically used, reading all  $B$  index blocks.

Thus, the process of searching a large index may be costly.

### Solution to this problem

We treat the index just as we would treat any other sequential file, and construct a sparse index on the primary index, as in the figure beside.

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 that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.

Using two levels of indexing, it is required to read only one block rather than seven with binary search, if we assume that the outer index is already in main memory.

For very large files, additional levels of indexing may be required.

Indices must be updated at all levels when insertions or deletion operations are performed.

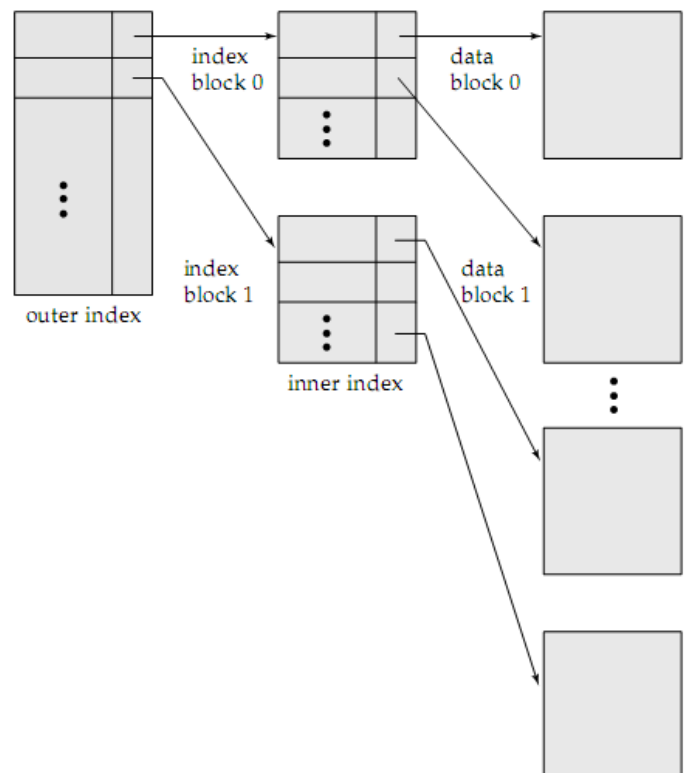


Figure: Two level sparse index.

## Secondary Indices

A secondary index on a candidate key looks just like a dense primary index, except that the records pointed to by successive values in the index are not sorted sequentially.

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 because the remaining records with the same search-key value could be anywhere in the file. 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. A pointer does not point directly to the file but to a bucket that contains pointers to the file.

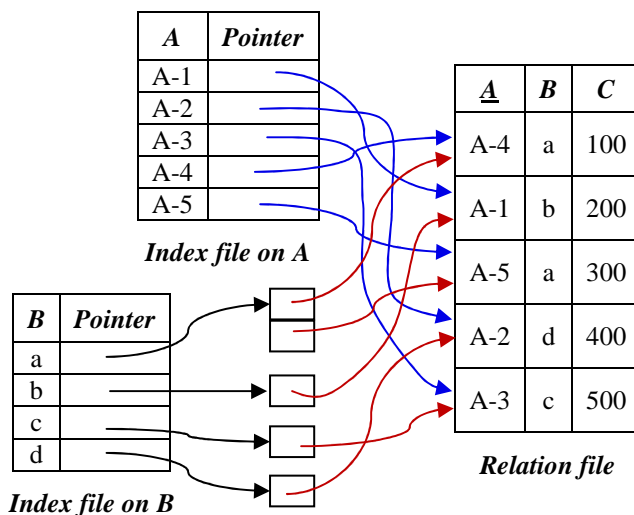


Figure: Secondary indices on candidate key A and non-candidate key B.

*Secondary indices must be dense*, with an index entry for every search-key value, and a pointer to every record in the file.

Secondary indices improve the performance of queries on that use keys other than the search key of the primary index. However, they impose a significant overhead on database modification.

The designer of the database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications.

## B<sup>+</sup> Tree Index

### The Problem with Indexed-Sequential File Organization

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.

### How B<sup>+</sup> Tree Index Solves the Problem

B<sup>+</sup> tree index structure maintains its efficiency despite frequent insertions and deletions. It automatically reorganizes itself with small local changes, in the face of insertions and deletions. It imposes performance overhead on insertion and deletion. Again, since nodes may be as much as half empty (if they have the minimum number of children), it adds space overhead. These overheads are acceptable even for frequently modified files, since the cost of reorganization is avoided.

### Structure of a B<sup>+</sup> Tree

A B<sup>+</sup> tree index is a multilevel index but is structured differently from that of multi-level indexed-sequential files. It is a balanced tree in which every path from the root to a leaf is of the same length.

A typical node contains up to  $n - 1$  search key values  $K_1, K_2, \dots, K_{n-1}$ , and  $n$  pointers  $P_1, P_2, \dots, P_n$ . Search key values in a node are kept in sorted order, thus, if  $i < j$ , then  $K_i < K_j$ , i.e.,  $K_1 < K_2 < \dots < K_{n-1}$ .

$P_1$	$K_1$	$P_2$	...	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	-----	-----------	-----------	-------

Figure: Typical node of a B<sup>+</sup> tree.

### Leaf nodes

- All the pointers  $P_i$  ( $i = 1$  to  $n - 1$ ) in the leaf node points to either a file record with search key value  $K_i$ , or a bucket of pointers, each of which points to a file record with that search key value.  
Bucket structure is used only if search key is not a primary key, and file is not sorted in search key order.
- Pointer  $P_n$  ( $n^{\text{th}}$  pointer in the leaf node) is used to chain leaf nodes together in linear order (search key order). This allows efficient sequential processing of the file.
- Each leaf has between  $\lceil (n - 1) / 2 \rceil$  and  $n - 1$  values. The ranges of values in each leaf do not overlap.
- If the B<sup>+</sup>-tree index is to be a dense index, every search key value must appear in some leaf node.

### Non-leaf nodes

- Non-leaf nodes form a multilevel sparse index on leaf nodes. The structure of the non-leaf nodes is same as that of leaf nodes, except that all pointers are pointing to tree nodes.
- Each non-leaf node in the tree has between  $\lceil n / 2 \rceil$  and  $n$  pointers, where  $n$  is fixed for a particular tree. The number of pointers in a node is called the **fan-out** of the node.

### Root node

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.

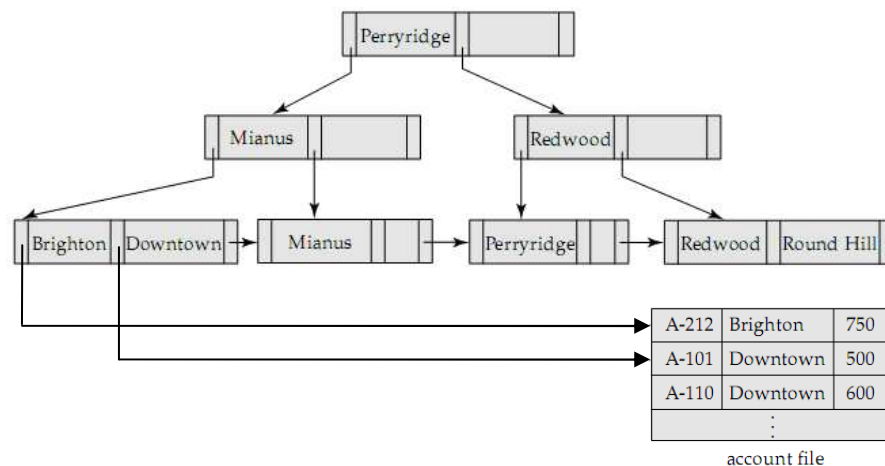
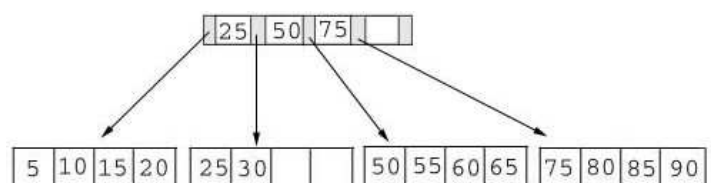


Figure: B+ tree structure with  $n = 3$ . Note that no bucket structure is used as the search-key is a primary key and the file is sorted in the search-key order.

### Operations on a B<sup>+</sup> Tree

The figure beside shows a B<sup>+</sup> tree. As the example illustrates, this tree does not have a *full* index page (we have room for one more key and pointer in the root page). In addition, one of the data pages contains empty slots.



The key-value determines a record's placement in a B<sup>+</sup> tree. The leaf pages are maintained in sequential order and a linked list (not shown) connects each leaf page with its sibling page(s). This linked list speeds data movement as the pages grow and contract.

### Adding Records to a B<sup>+</sup> Tree

We must consider three scenarios when we add a record to a B+ tree. Each scenario causes a different action in the insert algorithm. The scenarios are:

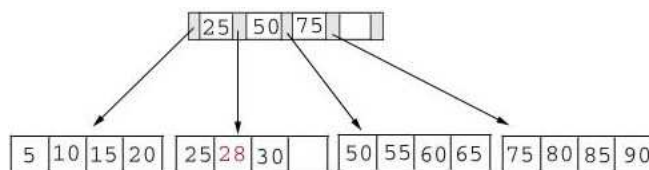


Leaf Page Full	Index Page Full	Action
NO	YES / NO	Place the record in sorted position in the appropriate leaf page.
YES	NO	<ol style="list-style-type: none"> <li>1. Split the leaf page.</li> <li>2. Place Middle Key in the index page in sorted order.</li> <li>3. Left leaf page contains records with keys below the middle key.</li> <li>4. Right leaf page contains records with keys equal to or greater than the middle key.</li> </ol>
YES	YES	<ol style="list-style-type: none"> <li>1. Split the leaf page.</li> <li>2. Records with keys &lt; middle key go to the left leaf page.</li> <li>3. Records with keys &gt;= middle key go to the right leaf page.</li> <li>4. Split the index page.</li> <li>5. Keys &lt; middle key go to the left index page.</li> <li>6. Keys &gt; middle key go to the right index page.</li> <li>7. The middle key goes to the next (higher level) index.</li> </ol> <p>If the next level index page is full, continue splitting the index pages.</p>

## Illustrations of the insert algorithm

### *Adding a record when the leaf page is not full*

We're going to insert a record with a key value of 28 into the B<sup>+</sup> tree. The following figure shows the result of this addition.



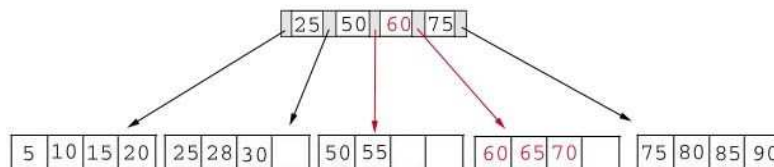
### *Adding a record when the leaf page is full but the index page is not*

Now, we're going to insert another record with a key value of 70 into our B<sup>+</sup> tree. This record should go in the leaf page containing 50, 55, 60, and 65. Unfortunately, this page is full. This means that we must split the page as follows:

Left Leaf Page	Right Leaf Page
50 55	60 65 70

The middle key of 60 is placed in the index page between 50 and 75.

The following table shows the B<sup>+</sup> tree after the addition of 70.



### *Adding a record when both the leaf page and the index page are full*

As our last example, we're going to add another record containing a key value of 95 to our B<sup>+</sup> tree. This record belongs in the page containing 75, 80, 85, and 90. Since this page is full we split it into two pages:

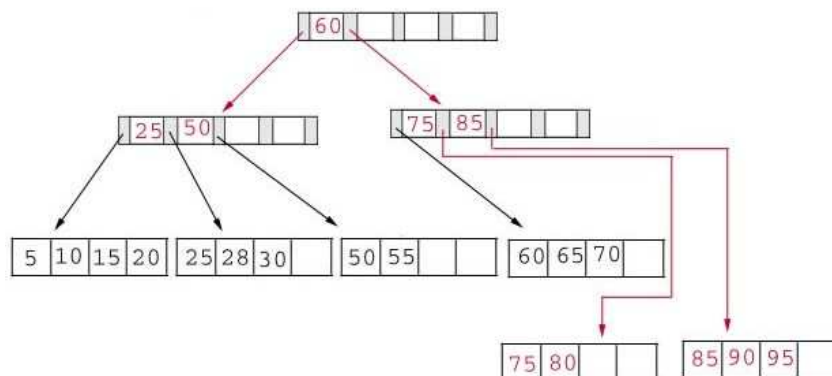
Left Leaf Page	Right Leaf Page
75 80	85 90 95

The middle key, 85, rises to the index page. Unfortunately, the index page is also full, so we split the index page:

Left Index Page	Right Index Page	New Index Page
25 50	75 85	60



The following figure illustrates the addition of the record containing 95 to the B+ tree.



### Deleting records from a B<sup>+</sup> Tree

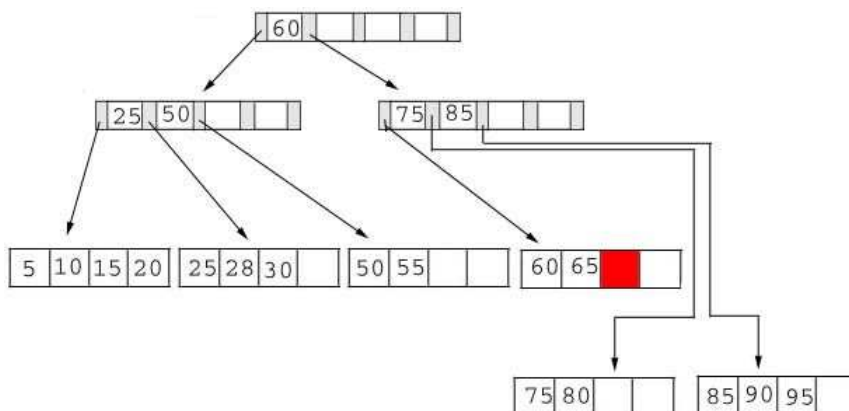
We must consider three scenarios when we delete a record from a B<sup>+</sup> tree. Each scenario causes a different action in the delete algorithm. The scenarios are:

Leaf Page Below Fill Factor	Index Page Below Fill Factor	Action
NO	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
YES	NO	Combine the leaf page and its sibling. Change the index page to reflect the change.
YES	YES	<ol style="list-style-type: none"> <li>Combine the leaf page and its sibling.</li> <li>Adjust the index page to reflect the change.</li> <li>Combine the index page with its sibling.</li> </ol> <p>Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.</p>

### Illustrations of the delete algorithm

#### Deleting a record that leaves the leaf page below fill factor, but not the index page

We begin by deleting the record with key 70 from the B<sup>+</sup> tree. This record is in a leaf page containing 60, 65 and 70. This page will contain 2 records after the deletion. Since our fill factor<sup>3</sup> is 50% or (2 records) we simply delete 70 from the leaf node. The following figure shows the B<sup>+</sup> tree after the deletion:

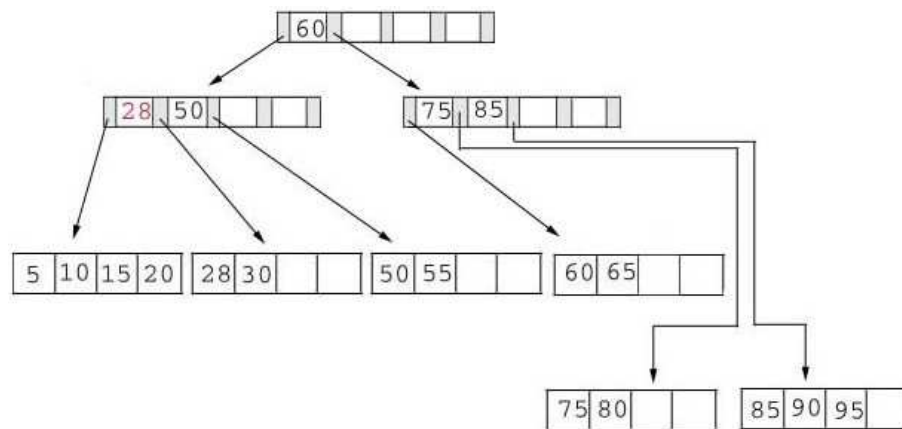


<sup>3</sup> Fill factor: The minimum number of keys that can exist in a node / page. Its value is  $\lceil n / 2 \rceil$ .

### ***Deleting a record that leaves none of the pages below fill factor***

Next, we delete the record containing 25 from the B<sup>+</sup> tree. This record is found in the leaf node containing 25, 28, and 30. The fill factor will be 50% after the deletion; however, 25 appears in the index page. Thus, when we delete 25 we must replace it with 28 in the index page.

The following figure shows the B<sup>+</sup> tree after this deletion:

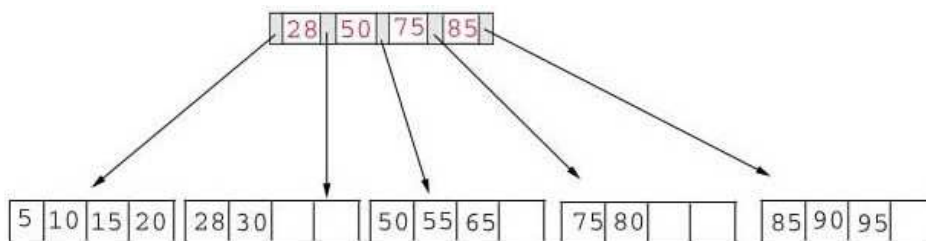


### ***Deleting a record that leaves both of the pages below fill factor***

As our last example, we're going to delete 60 from the B<sup>+</sup> tree. This deletion is interesting for several reasons:

1. The leaf page containing 60 (60 65) will be below the fill factor after the deletion. Thus, we must combine leaf pages.
2. With recombined pages, the index page will be reduced by one key. Hence, it will also fall below the fill factor. Thus, we must combine index pages.
3. 60 appears as the only key in the root index page. Obviously, it will be removed with the deletion.

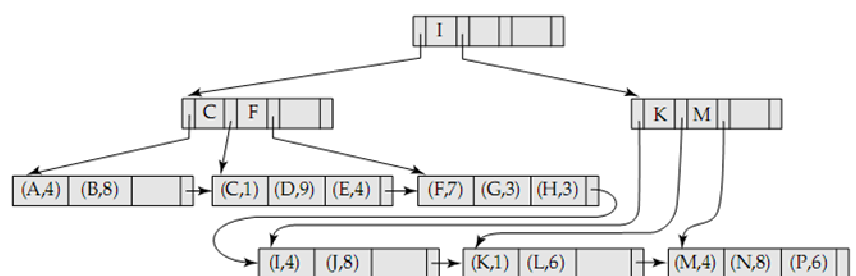
The following figure shows the B<sup>+</sup> tree after the deletion of 60. Notice that the tree contains a single index page.



### **B<sup>+</sup> Tree File Organization**

The performance of index-sequential file organization degrades as the file grows. With growth, an increasing percentage of index records and actual records become out of order and are stored in overflow blocks. We solve the degradation of index lookup by using B<sup>+</sup> tree indices on the file. We solve the degradation problem of storing the actual records by using the leaf level of B<sup>+</sup> tree to organize the blocks containing the actual records.

- In a B<sup>+</sup> tree file organization, the leaf nodes of the tree store records instead of storing pointers to records.
- Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is less than the maximum number of

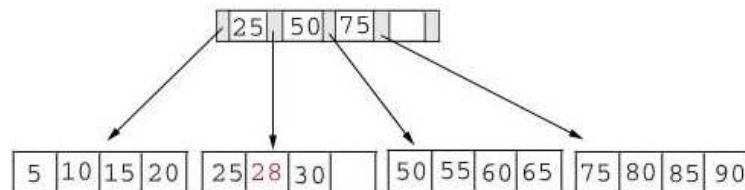


**Figure:** B<sup>+</sup> tree file organization.

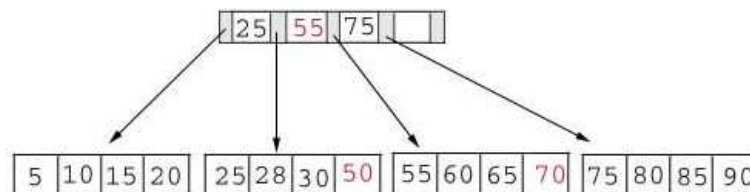
pointers in a non-leaf node.

- However, the leaf nodes are still required to be at least half full.
- Insertion and deletion from a B<sup>+</sup> tree file organization are handled in the same way as that in a B<sup>+</sup> tree index.
- When a B<sup>+</sup> tree is used for file organization, space utilization is particularly important. We can improve the space utilization by involving more sibling nodes in redistribution during splits and merges. This technique is called **rotation**.

As an example, consider the B<sup>+</sup> tree before the addition of the record containing a key of 70. As previously stated, this record belongs in the leaf node containing 50 55 60 65. Notice that this node is full, but its left sibling is not:



Using rotation we shift the record with the lowest key to its sibling. Since this key appeared in the index page we also modify the index page. The new B<sup>+</sup> tree appears in the following figure:



## B-Tree Index

- B-tree indices are almost similar to B<sup>+</sup> tree indices, but B-tree eliminates the redundant storage of search key values. In B<sup>+</sup> tree, some search key values appear twice. A corresponding B-tree allows search key values to appear only once. Thus we can store the index in less space.
- Search keys in non-leaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a non-leaf node must be included.
- For non-leaf node, pointers  $B_i$  are the bucket or file record pointers.

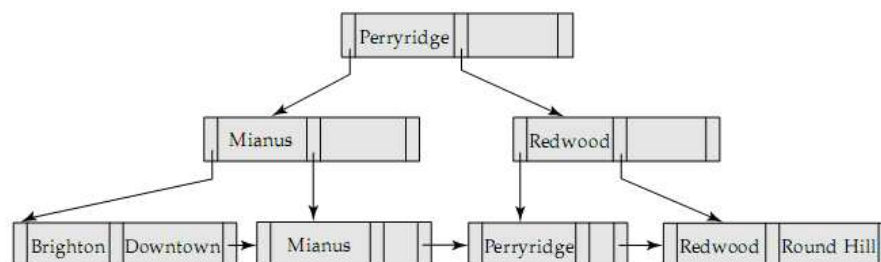
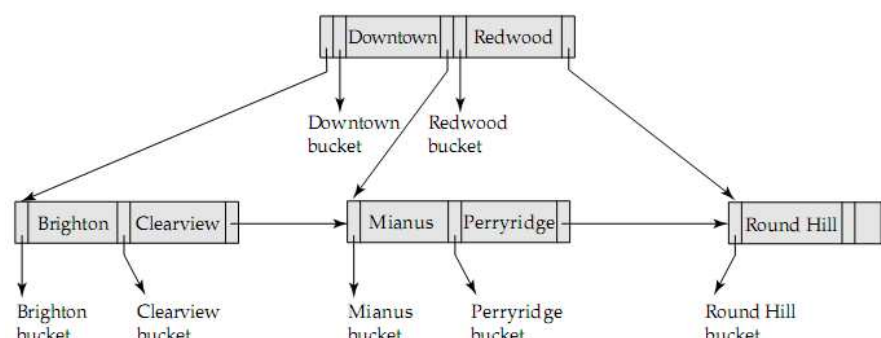


Figure: B<sup>+</sup> tree (above) and the corresponding B-tree (below).



## Advantages of B-Tree

1. May use less tree nodes than a corresponding  $B^+$  tree.
2. Sometimes it is possible to find the desired value before reaching a leaf node.

## Disdvantages of B-Tree

1. Only a small fraction of desired values are found before reaching a leaf node.
2. Fewer search-keys appear in non-leaf nodes; hence, fan-out is reduced. Thus, B-trees typically have greater depth than a corresponding  $B^+$  tree.
3. Insertion and deletion are more complicated than in  $B^+$  trees.
4. Implementation is harder than  $B^+$  trees, since leaf and non-leaf nodes are of different sizes.

Typically, advantages of B-trees do not outweigh its disadvantages.

## Hashing

### The Problem with Sequential File Organization and How Hashing Solves It

One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary search, and that results in more I/O operations. File organizations based on the technique of hashing allow us to avoid accessing an index structure. Hashing also provides a way of constructing indices.

### Hash File Organization

In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by computing a *hash function* on the search-key value of the record.

A **hash function**  $h$  is a function from the set of all search-key values  $K$  to the set of all bucket addresses  $B$ :  $B_i = h(K_i)$ .

The term **bucket** is used to denote a unit of storage that can store one or more records. A bucket is typically a disk block but may be smaller or larger than a disk block.

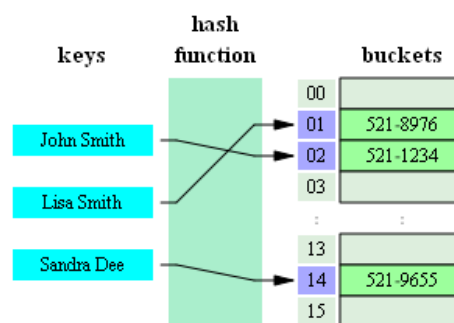


Figure: A typical phone book as a hash table.

### Manipulation of Records in Hash Files

1. **Insertion:** To insert a record with search key  $K_i$ , compute  $h(K_i)$ , which gives the address of the bucket of the record – assuming there is space for the record – and the record is inserted.
2. **Lookup:** To perform a lookup on a search key value  $K_i$ , we compute  $h(K_i)$ , and search the bucket with that address. If two search keys  $i$  and  $j$  map to the same address, because  $h(K_i) = h(K_j)$ , then the bucket at the address obtained will contain records with both search key values. In this case we will have to check the search key value of every record in the bucket to get the ones we want.
3. **Deletion:** If the search key value of the record to be deleted is  $K_i$ , compute  $h(K_i)$ , then search the corresponding bucket for the record and delete the record from the bucket.

### Hash Functions

- The worst possible hash function maps all search-key values to the same bucket. This is undesirable. A lookup has to examine every such record to find the desired one.
- An ideal hash function distributes the stored keys uniformly across all the buckets so that every bucket has the same number of records.

## Distribution Qualities for Choosing a Hash Function

Since it cannot be known at design time precisely which search-key values will be stored in the file, such hash function should be chosen that assigns search-key values to buckets in such a way that the distribution has these qualities:

**1. Uniform:** The distribution is uniform. The hash function assigns each bucket the same number of search-key values from the set of all possible search-key values.

**2. Random:** The distribution is random. In the average case, each bucket will have nearly the same number of values assigned to it regardless of the actual distribution of the search-key values. More precisely, the hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function will appear to be random.

## Some Examples Illustrating These Qualities

### Example 1:

Suppose we have 26 buckets, and map branch-names of *account* file beginning with  $i^{\text{th}}$  letter of the alphabet to the  $i^{\text{th}}$  bucket.

### Problem:

This does not give uniform distribution. Many more names will be mapped to “B” and “R” than to “Q” and “X”.

### Example 2:

Suppose we have 10 buckets, and a hash function to map search-key *balance* of *account* file. Supposing min and max values to be 1 and 100,000, we can use a hash function that divides the values into 10 ranges: 1 – 10,000, 10,001 – 20,000, ..., 90,001 – 100,000.

### Problem:

The distribution is not random. It's also not uniform as balances between 1 and 10,000 are far more common than are records with balances between 90,001 and 100,000.

## How Hash Functions Should be Designed

Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well-designed function gives an average-case lookup time that is a small constant independent of the number of search-keys in the file.

Typical hash functions perform some operation on the internal binary machine representations of characters in the search-key. For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo number of buckets could be returned. The figure beside shows the application of such a scheme, with 10 buckets, to the *account* file, under the assumption that the  $i^{\text{th}}$  letter in the alphabet is represented by the integer  $i$ .

## Handling of Bucket Overflows

So far, we have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a bucket overflow is said to occur.

bucket 0			
bucket 1			
bucket 2			
bucket 3	A-217	Brighton	750
	A-305	Round Hill	350
bucket 4	A-222	Redwood	700
bucket 5	A-102	Perryridge	400
	A-201	Perryridge	900
	A-218	Perryridge	700
bucket 6			
bucket 7	A-215	Mianus	700
bucket 8	A-101	Downtown	500
	A-110	Downtown	600
bucket 9			

Figure: Typical application of a hash function scheme.

### Causes of bucket overflows

1. **Insufficient buckets.** Our estimate of the number of records that the relation will have was too low, and hence the number of buckets allotted was not sufficient.
2. **Skew.** Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called *bucket skew*.

#### Causes of Skew

1. Multiple records may have the same search key.
2. The chosen hash function may result in non-uniform distribution of search keys.

### Reducing bucket overflows

To reduce the occurrence of overflows, we can:

1. Choose the hash function more carefully, and make better estimates of the relation size.
2. If the estimated size of the relation is  $n_r$  and number of records per block is  $f_r$ , allocate  $(n_r / f_r) \times (1 + d)$  buckets instead of  $(n_r / f_r)$  buckets. Here  $d$  is a *fudge factor*<sup>4</sup>, typically around 0.2. Some space is wasted: about 20% of the space in the buckets will be empty. But the benefit is that some of the skew is handled and the probability of overflow is reduced.

### Handling bucket overflows

Bucket overflows can be handled using two techniques:

1. **Closed Hashing.** If records must be inserted into a bucket and the bucket is already full, they are inserted into overflow buckets which are chained together in a linked list. Overflow handling using such a linked list is called **overflow chaining**. The form of hash structure that we have just described is sometimes referred to as *closed hashing*.

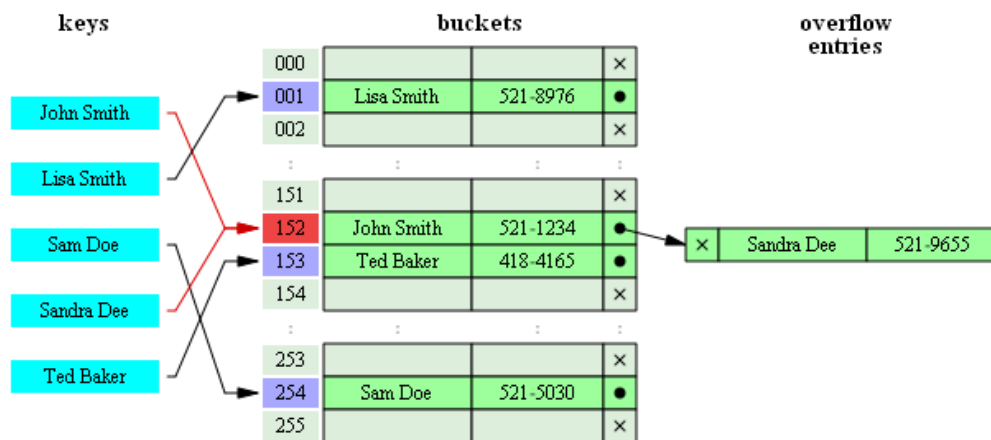
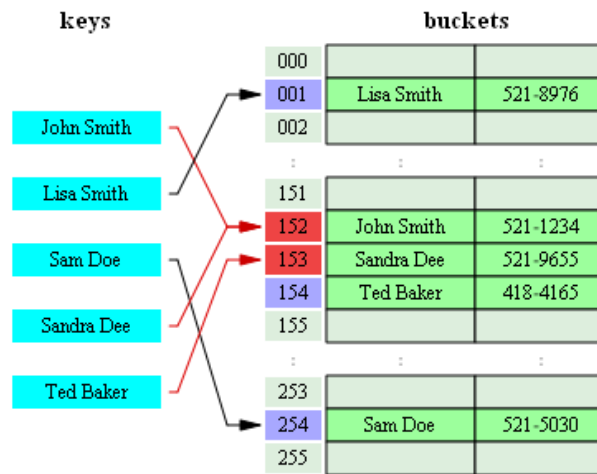


Figure: Bucket overflow handling using overflow chaining.

2. **Open Hashing.** In this technique, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. The probe sequence can be any of the following:
  - a. **Linear probing.** The interval between probes is fixed (usually 1).
  - b. **Quadratic probing.** The interval between probes increases by some constant (usually 1) after each probe.
  - c. **Double hashing.** The interval between probes is computed by another hash function.

<sup>4</sup> Fudge factor: A quantity that is added or subtracted in order to increase the accuracy of a scientific measure.





**Figure:** Hash collision (bucket overflow) resolved by open hashing with linear probing (interval=1).

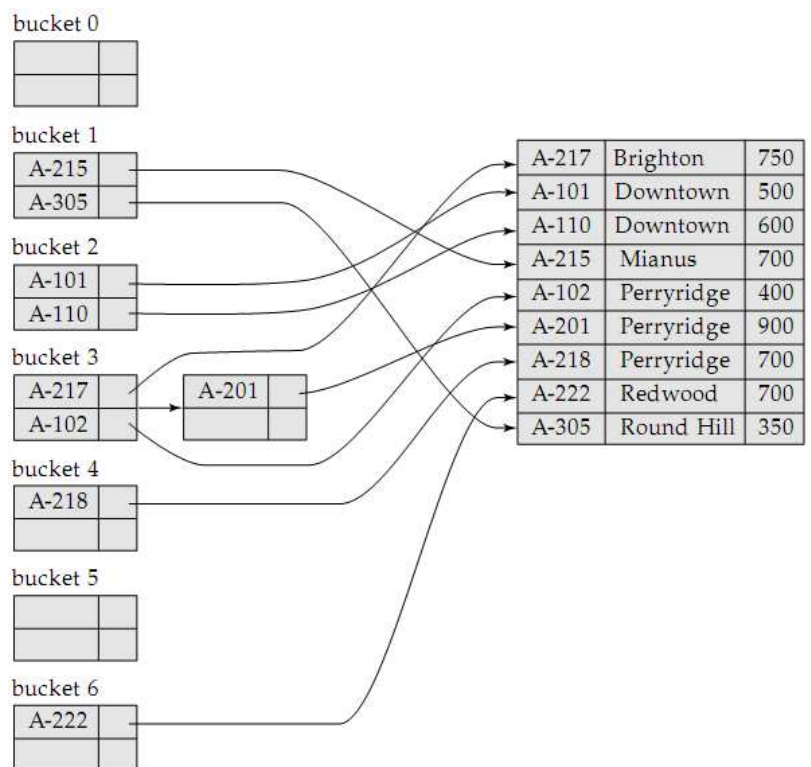
Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee" which had previously collided with "John Smith".

### Comparative analysis of closed and open hashing

Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems. The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations in their symbol tables. However, in a database system insertion-deletion occurs frequently. Thus, open hashing is of only minor importance in database implementation.

### Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search-keys, with their associated record pointers, into a hash file structure.
- We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).
- Hash indices are always secondary indices — if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary. However, we use the term *hash index* to refer to both *secondary index structures* and *hash organized files*.



**Figure:** Hash index on search key account-number of account file.

Here,  $B_i = h(\text{sum of digits of the account number modulo } 7)$

### Static and Dynamic Hashing

In static hashing, we need to fix the set  $B$  of bucket addresses. Since most databases grow over time, if we are to use static hashing for such a database, we have three classes of options:

1. **Choose a hash function based on the current size of the database.** This option will result in performance degradation as the database grows.



2. **Choose a hash function based on the anticipated size of the file at some point of time in future.** Although performance degradation is avoided, a significant amount of space may be wasted.
3. **Periodically reorganize the hash structure in response to the file growth.** Such reorganization involves choosing a new hash function, re-computing the hash function on every record in the file and generating new bucket assignments. This reorganization is massive and time-consuming operation. Furthermore, it is necessary to forbid access to the file during reorganization.

Dynamic hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database while *not* degrading performance and *without* any space overhead. There are several dynamic hashing techniques, for example:

1. Extendable Hashing
2. Linear Hashing

### Drawbacks of Hashing

1. Hash function must be chosen when the system is implemented and it cannot be changed easily thereafter if the file being indexed grows or shrinks.
2. Since the hash function maps search-key values to a fixed set of bucket addresses, space is wasted if the set of buckets is made large to handle further growth of the file.
3. If the set of buckets is too small, they will contain records of many different search-key values and bucket overflow can occur. As the file grows, performance suffers.

## Comparison of Ordered Indexing and Hashing

To make a wise choice of file organization and indexing techniques for a relation, a database designer must consider the following issues:

1. Is the cost of periodic re-organization of index or hash structure acceptable?
2. What is the relative frequency of insertion and deletion?
3. Is it desirable to optimize average access time at the expense of increasing worst-case access time?
4. What types of queries are users likely to pose?

This issue is critical to the choice between indexing and hashing.

### Queries that specify a single value

**Form:** `select  $A_1, A_2, \dots, A_n$  from  $r$  where  $A_i = c$`

**Average Case:** Favors hashing.

#### Analysis:

1. *Index lookup* takes time proportional to *log* of number of values in  $r$  for  $A_i$ .
2. *Hash structure* provides average lookup time that is a small constant (independent of database size).

**Worst Case:** Favors Indexing.

#### Analysis:

1. In *hash structure*, worst-case time is proportional to the number of values in  $r$  for  $A_i$ .
2. In *index structure*, the worst-case time is still *log* of number of values in  $r$ .

**Conclusion:** The worst-case lookup time is unlikely to occur with hashing; hence, **for these sorts of queries (having a specified value for the key), a hashing scheme is preferable.**

### Queries that specify a range of values

**Form:** `select  $A_1, A_2, \dots, A_n$  from  $r$  where  $A_i \leq c$  and  $A_i \geq c$`

## Preferred Scheme: Indexing.

### Analysis:

1. Using an index structure, we can find the bucket for value  $c_1$ , and then follow the pointer chain to read the next buckets in alphabetic (or numeric) order until we find  $c_2$ .
2. If we have a hash structure instead of an index, we can find a bucket for  $c_1$  easily, but it is not easy to find the next bucket *in sorted order*. Because:
  - a. A good hash function assigns values randomly to buckets.
  - b. Each bucket may be assigned many search key values, so we cannot chain them together.

**Conclusion:** Index methods are preferable where a range of values is specified in the query.

## Multiple-Key Indices

### Problem with Multiple Single-Key Indices

If there are two indices on *account* file, one on *branch-name* and one on *balance*, then suppose we have a query like:

```
select loan-no from account where branch-name = 'Perryridge' and balance = 1000
```

There are 3 possible strategies to process this query:

1. Use the index on *branch-name* to find all records pertaining to Perryridge branch. Examine them to see if balance = 1000.
2. Use the index on *balance* to find all records pertaining to accounts with balances of 1000. Examine them to see if branch-name = 'Perryridge'.
3. Use the index on *branch-name* to find pointers to records pertaining to 'Perryridge' branch. Also, use the index on *balance* to find pointers to records pertaining to 1000. Take the intersection of these two sets of pointers.

The third strategy takes advantage of the existence of multiple indices. This may still not work well if *all* the following conditions hold:

- a. There are a large number of Perryridge records AND
- b. There are a large number of 1000 records AND
- c. Only a small number of records pertain to both Perryridge and 1000.

To speed up the intersection operation, special structures such as *bitmap indices* can be used.

### Advantages of Using Multiple-Key Indices

Suppose we have an index on combined search-key (*branch-name, balance*).

With the *where* clause `where branch-name = 'Perryridge' and balance = 1000`, the index on the combined search-key will fetch only records that satisfy both conditions. Using separate indices is less efficient – we may fetch many records (or pointers) that satisfy only one of the conditions.

It can also efficiently handle `where branch-name = 'Perryridge' and balance < 1000`.

However, it cannot efficiently handle `where branch-name < 'Perryridge' and balance = 1000`. For each value of branch-name that is less than "Perryridge" in alphabetic order, the system locates records with a balance value of 1000. However, each record is likely to be in a different disk block, because of the ordering of records in the file, leading to many I/O operations.

To speed up the processing of general multiple search-key queries (involving one or more comparison operations), we can use special structures like *grid file*, *R-tree* etc.

## Index Definitions in SQL

### Creating an Index

```
create [unique] index index-name on relation-name (attribute-list)
```

The *attribute-list* is the list of attributes in relation *r* that form the search key for the index.

If the search key is a candidate key, we add the word `unique` to the definition. In that case,

- If the search-key is not a candidate key, an error message will appear.
- If the index creation succeeds, any attempt to insert a tuple violating this requirement will fail.
- The *unique* keyword is redundant if primary keys have been defined with integrity constraints already.

For example, to define an index on *branch-name* for the *branch* relation:

```
create index b-index on branch(branch-name)
```

### Removing an Index

```
drop index index-name
```