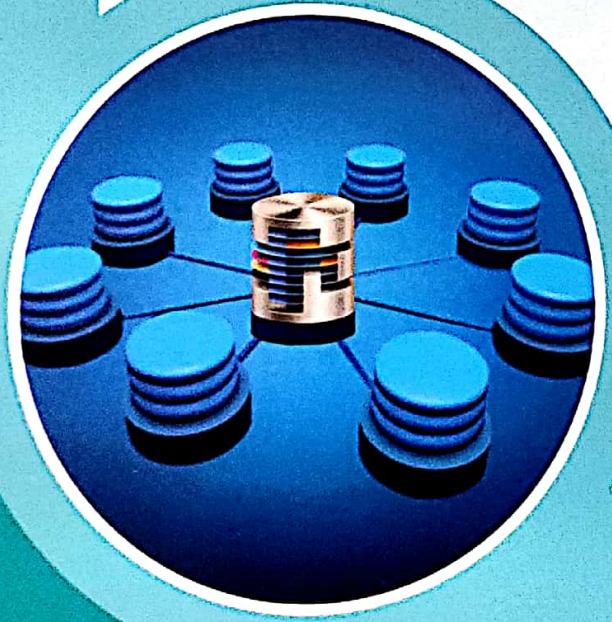
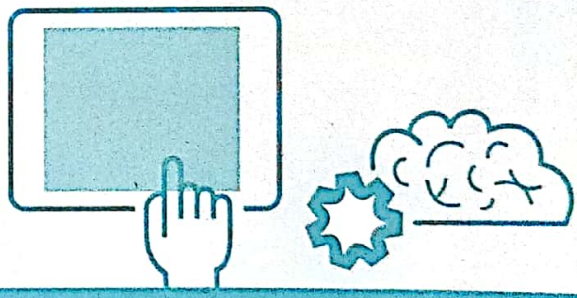
GATE
PSUs

MADE EASY Publications

POSTAL
STUDY PACKAGE

COMPUTER SCIENCE & IT





2020

THEORY BOOK

Database Management System

Well illustrated theory with solved examples

Contents

Database Management System

Chapter 6

Chapter 7

■■■■

Database Management System

Goal of the Subject

The main goal of Data Base management System is to make it possible for users to create, edit and update data in database files. Once created, the DBMS makes it possible to store and retrieve data from those database files.

More specifically, a DBMS provides the following functions:

- Concurrency: concurrent access (meaning 'at the same time') to the same database by multiple users
- Security: security rules to determine access rights of users
- Backup and recovery: processes to back-up the data regularly and recover data if a problem occurs
- Integrity: database structure and rules improve the integrity of the data
- Data descriptions: a data dictionary provides a description of the data

Database Management System

INTRODUCTION

Very often the subject *Data Base management System* is discussed vastly. But in this book we tried to keep it around the GATE syllabus. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into seven chapters as described below.

1. **The Relational Model**: In this chapter we discuss ER model and its constraints, types of relationships and we also the minimization of ER diagrams.
2. **Database design and Normalization: In** this chapter we discuss the types of keys in relational model , integrity constraints , functional dependencies of attributes and their properties, decomposition of relationship into 2NF, 3NF, BCNF etc.
3. **Relational Algebra**: In this chapter we discuss the Operators of relational algebra, operations on the set of records and finally we discuss the Tuple Relational Calculus (TRC).
4. **SQL**: In this chapter we discuss the basic form of SQL Query, operator, types of queries, group by and having clauses and finally we discuss the special properties of NULL value.
5. **Transaction**: In this chapter we discuss the ACID properties of a transaction, problem of concurrent execution, serializability etc.
6. **Concurrency Control Techniques**: In this chapter we discuss the various protocols used to achieve concurrent execution with any of the problem discussed in the previous chapter.
7. **File Organization and Indexing:** in this chapter we discuss concept of file organization, indexing techniques and B, B+ trees which form backbone of indexing techniques.

■■■■

CHAPTER 01

The Relational Model

1.1 Introduction

Entity relationships (ER) model is high level database design allows us to describe the data involved in real-world enterprise in terms of objects and their relationships and is widely used to develop initial database design. In overall design process, the ER model is used in a phase called conceptual database design.

1.2 Database Design and ER Diagrams

The database design process can be divided into six steps. The ER model is most relevant to the first three steps.

Requirements Analysis

The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database. This is usually an informal process that involves discussions with user groups, a study of the current operating environment and how it is expected to change, analysis of any available documentation on existing applications that are expected to be replaced or complemented by the database, and so on.

Conceptual Database Design

The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints known to hold over this data. This step is often carried out using the ER model and is discussed in the rest of this chapter. The ER model is one of several high-level, or semantic, data models used in database design. The goal is to create a simple description of the data that closely matches how users and developers think of the data

Logical Database Design

We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will consider only relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema.

1.3 Entity, Attributes, Entity Set

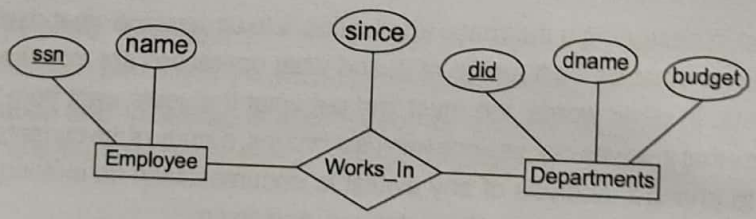
- Entity is an object that exist and is distinguishable from other objects. For example a person with give UID is an entity as he can be uniquely identified as one particular person.
- An entity may be concrete (person) or abstract (job)
- An entity set is a collection of similar entities. (All persons having an account at a bank)
- Entity sets need not be disjoint. For example, the entity set employee (all employees of a bank) and entity set customer (all customers of the bank) may have members (Entity) in common.
- An entity is described using a set of attributes, all entities in a given entity set have same attributes, this is what we mean by similar.
- For each attribute associated with an entity set, we must identify domain of attribute which is the set of permitted values (e.g. if a company rates employees on a scale of 1 to 10 and stores rating in a field called rating, the associated domain consist of integers 1 through 10)
- An analogy can be made with the programming language notion of type definition, concept of entity set corresponds to the programming language type definition.
- A variable of a given type has a particular value at a time, thus a programming language variable corresponds to an entity in ER model.

1.4 Relationship and Relationship Sets

- Relationship is association between two or more entities.
- Relationship set is a set of relationships of same type i.e. relate two or more entity sets.

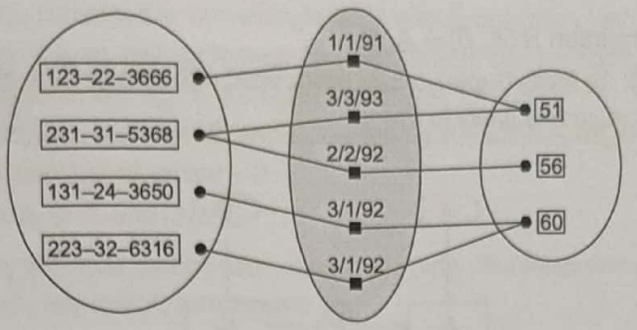
A relationship set can be thought of as a set of n-tuples: $\{(e_1,...,e_n) \mid e_1 \in E_1,..., e_n \in E_n\}$

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



A relationship can also have **descriptive attributes.** Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities; for example, we may wish to record that XYZ works in the pharmacy department as of January 1991. This information is captured in Figure by adding an attribute, *since*, to works_In. A relationship must be uniquely identified by the participating entities, without reference to the descriptive attributes. In the works_In relationship set, for example, each Works_In relationship must be uniquely identified by the combination of employee *ssn* and department did. Thus, for a given Employee-Department pair, we cannot have more than one associated *since* value.

An **instance** of a relationship set is a set of relationships. Intuitively, an instance can be thought of as a 'snapshot' of the relationship set at some instant in time. An instance of the Works_In relationship set is shown in Figure. Each Employees entity is denoted by its *ssn*, and each Departments entity is denoted by its did, for simplicity. The *since* value is shown beside each relationship.



An Instance of the Works_In Relationship Set

1.5 Relationship Constraints

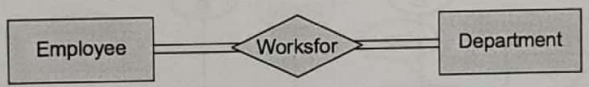
There are two types of relationship constraints

- Participation constraints
- Cardinality ratio.

There are two types of participation constraints:

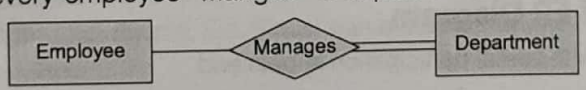
(*i*) **Total participation constraints (existence dependency):** The participation of an entity set *E* in a relationship set *R* is said to be total if every entity in *E* participates in at least one relationship in *R*. This participation is displayed as a double line connecting.

Example: If a company policy states that every employee must work for a department, then an employee entity can exist only if it participates in at least one "worksfor" relationship instance.



(*ii*) **Partial Dependency:** If only some entities in *E* participate in relationship in *R*, the participation of entity set *E* in relationship *R* is said to be partial. This participation is displayed as a single line connecting.

Example: Not every employee "Manges" a department.

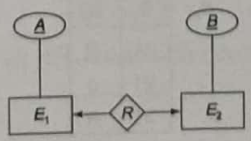


Mapping Constraints: The possible cardinality ratio for binary relationship are:

- One to one (1 : 1)
- One to many (1 : *M*)
- Many to one (*M* : 1)
- Many to many (*M* : *M*)

One to One (1 : 1): An entity (tuple) in E_1 is associated with atmost one Entity (tuple) in E_2, and an entity in E_2 is associated with atmost one entity in *A*

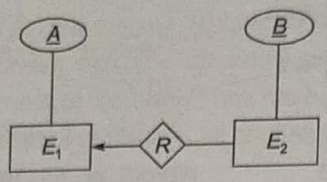
Example:



Candidate keys of relation $R(A, B) = \underline{A}, \underline{B}$

One to Many ($1 : M$): An entity (tuple) in E_1 is associated with zero or more entities in E_2 but an entity in E_2 can be associated with at most one entity in E_1

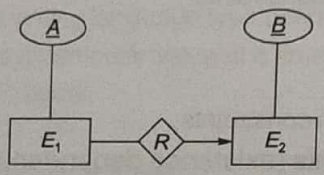
Example :



Candidate key of $R(A, B) = \underline{B}$

Many to one ($M : 1$): An entity (tuple) in E_2 is associated with zero or more entities in E_1 but an entity in E_1 can be associated with at most one entity in E_2.

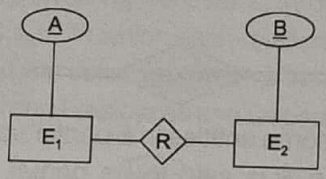
Example :



Candidate key of $R(A, B) = \underline{A}$

Many to many ($M : M$): An entity (tuple) in E_2 is associated with zero or more entities in E_1 and An entity (tuple) in E_1 is associated with zero or more entities in E_2.

Example :

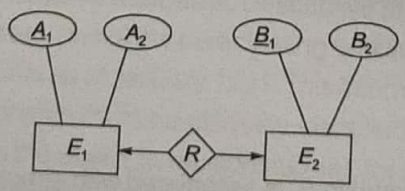


Candidate key of $R(A, B) = \underline{AB}$

1.6 Minimization of ER Diagram

1. **1 : 1 cardinality** with partial participation at both end

Example:



Consider the relation instances of relation E_1, E_2 and R.

E_1:

A_1	A_2
1	P
2	P
3	q

E_2:

B_1	B_2
11	P
21	q
31	R

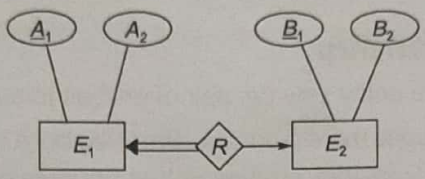
R:

A_1	B_1
1	11
3	21

E_1 and R can be combined to form a single table with A_1 is primary key (Unique and not NULL) and B_1 as alternate key as well as foreign key. Similarly R can be combined with E_2. But can't be combined to both E_1 and E_2 i.e. a single table E_1RE_2

- If R is combined with both E_1 and E_2 i.e. a single relation E_1RE_2 then it has no primary key
- **Minimum number of tables** = 2
  (E_1R) and E_2 or E_1 and (E_2R)

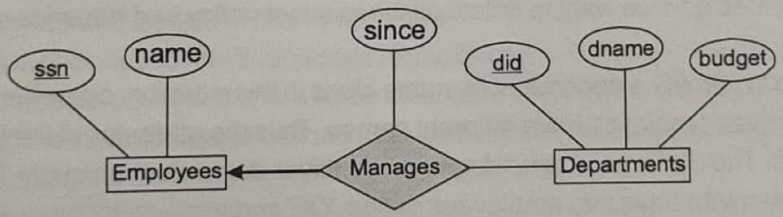
2. **1 : 1 cardinality** with total participation atleast one side. The relationship set E_1RE_2 (A_1, A_2, B_1, B_2) has B_1 as primary key and A_1 as alternate key.



Note that A_1 can be null because there may be an entity in E_2 which is not related to any entity of E_1.

- **Minimum number of tables** = 1 *i.e.* E_1RE_2

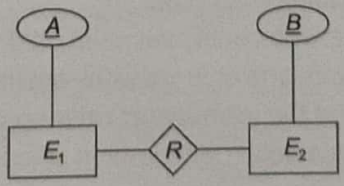
3. **1 : *M* Cardinality:** Consider relationship set called manages between the Employees and Departments entity sets such that each department has at most one manager, although a single employee is allowed to manage more than one department. The restriction that each department entity appears in at most one manager is an example of a key constraint, and it implies that each departments entity appears in at most one manages relationship in any allowable instance of Manages. This restriction is indicated in the ER diagram of Figure.



The entity set Departments and relationship set Manages combined into a single entity set dept_manages (did, dname, Eid, Since) with did as primary key and Eid, is foreign key to the entity set Employees.

- If foreign key attribute Eid is NULL, then partial participation from Employees set.
- Minimum number of tables = 2

4. ***M* : *M* Cardinality:** Consider the following ER model

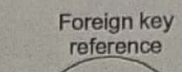


Relationship R has primary key as AB. R can't be combined with E_1 or E_2.
Minimum number of tables = 3

NOTE

When to minimize?

If relationship set *R* is having a key as *A* which is also foreign key referencing to entity set *E*, then *R* and *E* combined to a single entity set e.g.

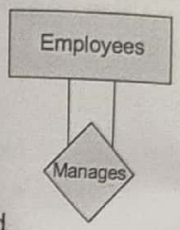


R (*A*, *B*) and *E* (*C*, *D*)

R and *E* combined to a single table = *RE* (*C*, *B*, *D*)

1.7 Self Referential Relationship

Relationship set relates to same entity sets *i.e.* pair of entities relating to each other.



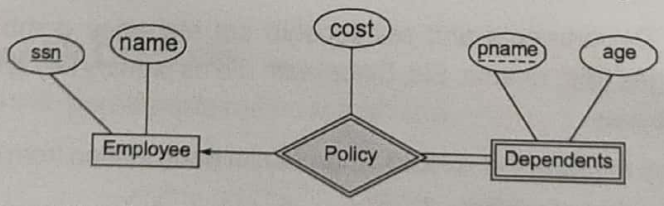
Example: Each employee manages more than one employee but a employee has only one manager *i.e.* 1 : *M* relation exist. Consider employee has attributes Eid, Ename, and Manages have attribute Sup*ID*, Sub*ID*, both Sup*ID* and Sub*ID* are foreign key in the employee referencing Eid, if 1 : *M* relationship exist, Sub*ID* is the key in relationship set manages.

Therefore combined into a single table. If *M* : *M* relationship then two tables are required.

1.8 Weak Entity Set

Thus far, we have assumed that the attributes associated with an entity set include a key. This assumption does not always hold. For example, suppose that employees can purchase insurance policies to cover their dependents. We wish to record information about policies, including who is covered by each policy, but this information is really our only interest in the dependents of an employee. If an employee quits, any policy owned by the employee is terminated and we want to delete all the relevant policy and dependent information from the database.

We might choose to identify a dependent by name alone in this situation, since it is reasonable to expect that the dependents of a given employee have different names. Thus the attributes of the Dependents entity set might be *pname* and *age.* The attribute *pname* does *not* identify a dependent uniquely. Recall that the key for Employees is *ssn;* thus we might have two employees called *XYZ* and each might have a son called *ABC*.



Dependents is an example of a **weak entity set.** A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the **identifying owner**. The following restrictions must hold:

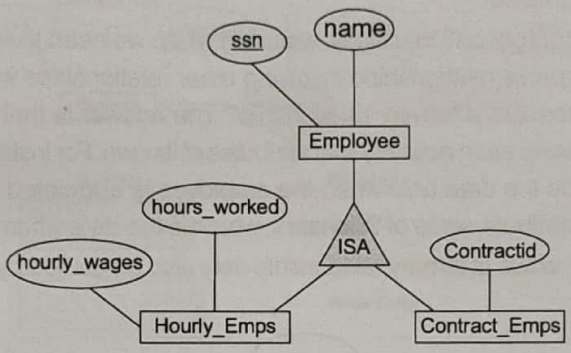
1. The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner). This relationship set is called the **identifying relationship set** of the weak entity set.
2. The weak entity set must have total participation in the identifying relationship set.

For example, a Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity. The set of attributes of a weak entity set that uniquely identify a weak entity for a given owner entity is called a *partial key* of the weak entity set. In our example, *pname* is a **partial key** for Dependents.

The Dependents weak entity set and its relationship to Employees is shown in Figure. The total participation of Dependents in Policy is indicated by linking them with a double line. The arrow from Policy to Dependents indicates that each Dependents entity appears in at most one (indeed, exactly one, because of the participation constraint) Policy relationship. To underscore the fact that Dependents is a weak entity and Policy is its identifying relationship, we draw both with double lines. To indicate that *pname* is a partial key for Dependents, we underline it using a broken line. This means that there may be two dependents with the same *pname* value.

Class Hierarchies

Sometimes it is natural to classify the entities in an entity set into subclasses. For example, we might want to talk about an Hourly_Emps entity set and a Contract_Emps entity set to distinguish the basis on which they are paid. We might have attributes hours_worked and hourly_wage defined for Hourly_Emps and an attribute contractid defined for Contract_Emps.



1.9 Specialization and Generalization

A class hierarchy can be viewed in one of two ways:

1. Employees is specialized into subclasses. Specialization is the process of identifying subsets of an entity set (the superclass) that share some distinguishing characteristic. Typically, the superclass is defined first, the subclasses are defined next, and subclass-specific attributes and relationship sets are then added.
2. Hourly-Emps and Contract_Emps are generalized by Employees. As another example, two entity sets Motorboats and Cars may be generalized into an entity set Motor Vehicles. Generalization consists of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics. Typically, the subclasses are defined first, the superclass is defined next, and any relationship sets that involve the superclass are then defined.

There are two basic reasons for identifying subclasses (by specialization or generalization):

- We might want to add descriptive attributes that make sense only for the entities in a subclass. For example, *hourly_wages* does not make sense for a Contract_Emps entity, whose pay is determined by an individual contract.
- We might want to identify the set of entities that participate in some relationship. For example, we might wish to define the Manages relationship so that the participating entity sets are Senior-Emps and Departments, to ensure that only senior employees can be managers. As another example, Motorboats and Cars may have different descriptive attributes (say, tonnage and number of doors), but as Motor_Vehicles entities, they must be licensed. The licensing information can be captured by a Licensed_To relationship between Motor_Vehicles and an entity set called Owners.

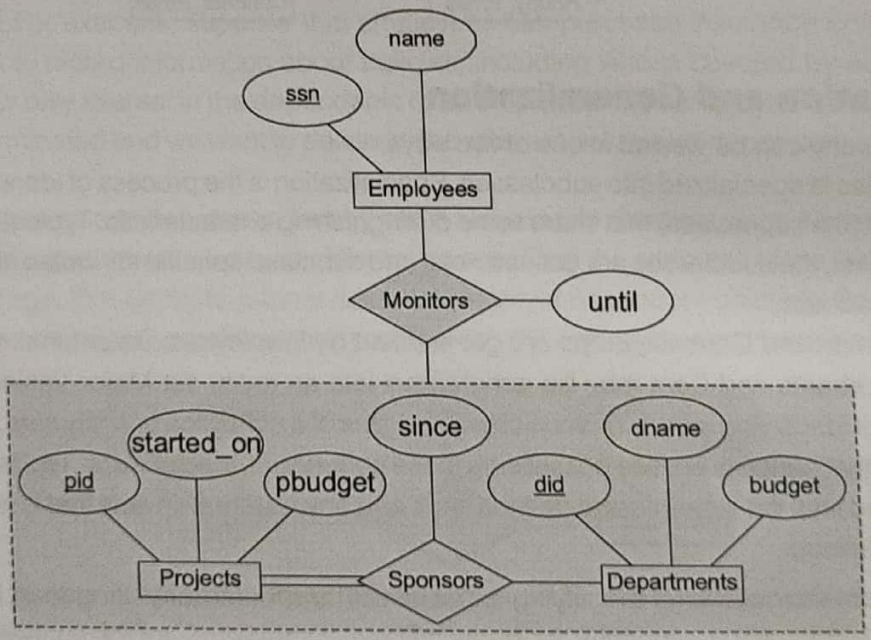
1.10 Aggregation

A relationship set is an association between entity sets. Sometimes, we have to model a relationship between a collection of entities and *relationships*. Suppose that we have an entity set called Projects and that each Projects entity is sponsored by one or more departments. The Sponsors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Monitors should be a relationship set that associates a Sponsors relationship (rather than a Projects or Departments entity) with an Employees entity. However, we have defined relationships to associate two or more *entities*.

Aggregation allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set. This is illustrated in Figure, with a dashed box around Sponsors (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the Monitors relationship set.

When should we use aggregation? Intuitively, we use it when we need to express a relationship among relationships. But can we not express relationships involving other relationships without using aggregation? In our example, why not make Sponsors a ternary relationship? The answer is that there are really two distinct relationships, Sponsors and Monitors, each possibly with attributes of its own. For instance, the Monitors relationship has an attribute until that records the date until when the employee is appointed as the sponsorship monitor. Compare this attribute with the attribute *since* of Sponsors, which is the date when the sponsorship took effect.

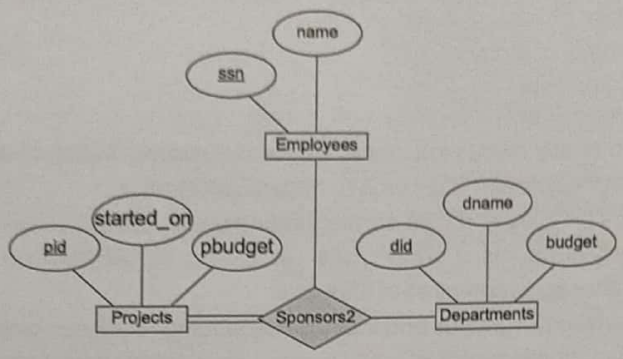
The use of aggregation versus at ternary relationship may also be guided by certain integrity constraints, as explained in next section.



1.11 Aggregation Vs Ternary Relationships

The choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a relationship *set* to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express. For example, consider the ER diagram shown in figure above. According to this diagram, a project can be sponsored by any number of departments, a department can sponsor one or more projects, and each sponsorship is monitored by one or more employees. If we don't need to record the until attribute of Monitors, then we might reasonably use a ternary relationship, say, Sponsors2, as shown in Figure below.

Consider the constraint that each sponsorship (of a project by a department) be monitored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship Sponsors to the relationship Monitors in figure above. Thus, the presence of such a constraint serves as another reason for using aggregation rather than a ternary relationship set.



Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Indentifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
E_1 E E_2	Total Participation of E_2 in R
E_1 1 R N E_2	Cardinality Ratio 1 : N for E_1 : E_2 in R
(min, max) R E	Structural Constraint (min, max) on Participation of E in R

Summary

In this chapter we presented the modeling concepts of a high-level conceptual data model, the Entity-Relationship (ER) model.We started by discussing the role that a high-level data model plays in the database design process, we defined the basic ER model concepts of entities and their attributes. Which can be nested arbitrarily to produce complex attributes:

- Simple or atomic
- Composite
- Multivalued

We also briefly discussed stored versus derived attributes. Then we discussed the ER model concepts at the schema or "intension" level:

- Entity types and their corresponding entity sets
- Key attributes of entity types
- Value sets (domains) of attributes
- Relationship types and their corresponding relationship sets
- Participation roles of entity types in relationship types

We presented two methods for specifying the structural constraints on relationship types. The first method distinguished two types of structural constraints:

- Cardinality ratios (1:1, 1:N, M:N for binary relationships)
- Participation constraints (total, partial)

We discussed weak entity types and the related concepts of owner entity types, identifying relationship types, and partial key attributes.

The ER modeling concepts we have presented thus far—entity types, relationship types, attributes, keys, and structural constraints—can model many database applications. However, more complex applications—such as engineering design, medical information systems, and telecommunications—require additional concepts if we want to model them with greater accuracy.

Student's Assignment

Q.1 Which of the following statement above ER models is/are correct?

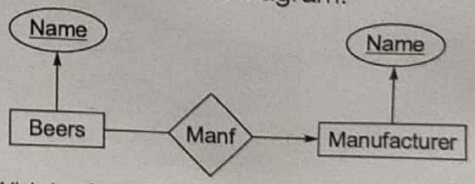
I. Many-many relationships cannot be representation ERD.

II. Relationship sets can have attributes of their own.

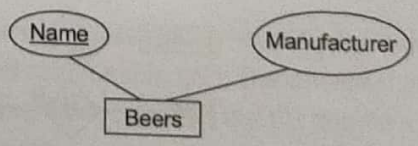
III. All many to one relationships are represented by the relationships between a weak and a non weak entity set.

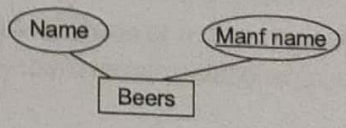
(a) II only (b) III only
(c) II and III only (d) I and II only

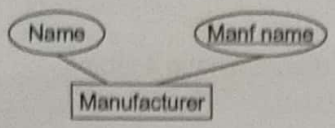
Q.2 Figure shows an ER diagram:

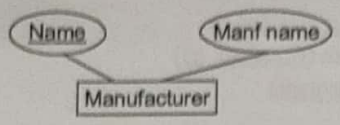


Which of the following best describes the above ER diagram?

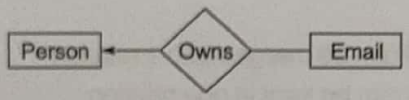
(a)


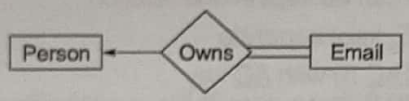
(b)


(c)


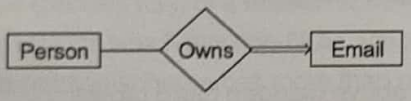
(d)


Q.3 Suppose we have two entity sets person, Email and use a relationship Owns. A person own atmost one email account but an email account can be owned by multiple persons. Which of the following is an ER diagram based on above description.

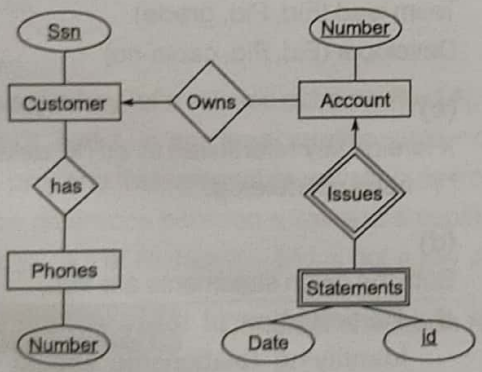
(a)


(b)


(c)


(d)


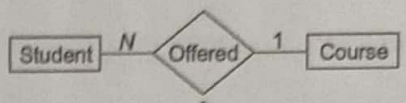
Q.4 Consider the following ER-Model

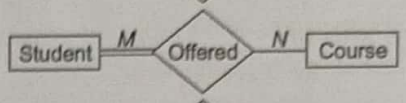


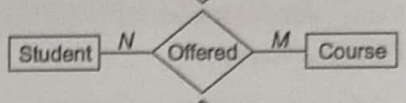
Find number of tables present in minimized ER model

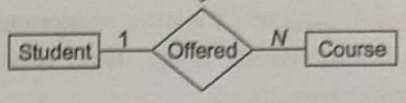
(a) 2 (b) 3
(c) 4 (d) 5

Q.5 A student can take one or more courses and courses can be offered to any number of students. Which of the following represents given scenario in ER-model

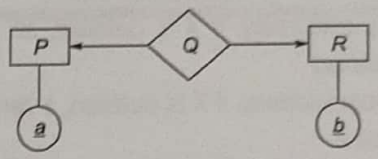
(a)


(b)


(c)


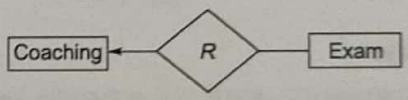
(d)


Q.6 Consider converting the following E-R diagram into a relational schema and that we must have tables for both P and Q



(a) It requires that we make a table for Q
(b) To model Q, we just need one forgien key from P to R
(c) To model S we need a forgien key from A to B and a forgien key from B to A
(d) It's logic can not be captured properly by a relational schema

Q.7 Consider the entities coaching, exam with relationship. R



If we wish to store information about fees for different coaching. This attribute should appear as an attribute of table

(a) Coaching (b) Exam
(c) R (d) None of these

Q.8 R is relationship, with 1 : 1 cardinality 30% participation at E_1 end 70% participation at E_2 end which is the best possible design?

(a) E_1 and E_2 kept separate with foriegn key at E_1 end
(b) E_1 and E_2 kept separate with foriegn key at E_2 end
(c) E_1 and E_2 kept separate with foriegn key at E_1 as well as E_2
(d) E_1 and E_2 merges into a single table with no foriegn key

Q.9 For weak entity set to be meaningful it must be part of
(a) one to one relationship
(b) one to many relationship
(c) many to one relationship
(d) All of the above

Q.10 In an E-R diagram, Y is the dominant entity and X is a subordinate entity. Then which of the following is incorrect:
(a) Operationally, if Y is deleted, so X is also deleted
(b) X existence is dependent on Y
(c) Operationally, if X is deleted, so Y is also deleted
(d) Operationally, if X is deleted, Y remains the same

Q.11 Map the following statements with True (T)/ False (F)
S_1: Participation of the weak entity set in identifying relationship must be total.
S_2: Multivalued attributes in E-R diagram require separate tables when converted into relational model with satisfy 2NF.
(a) FT (b) TF
(c) FF (d) TT

Answers Key:

1. (a)	**2.** (a)	**3.** (c)	**4.** (d)	**5.** (b)
6. (b)	**7.** (a)	**8.** (b)	**9.** (b)	**10.** (c)
11. (d)				

Student's Assignments Explanations

2. (a)
Manufacturer is a name and it is at the 'one' end of any relationship. Hence it should not be an entity set option (a) is correct.

3. (c)
The relationship is many to one and the partial participation from both entity set.

4. (d)
Above ER diagram have 4 tables
Customer (ssn)
Account (number, ssn)
Has (ssn, number)
Statements (number, id)
Phone (number)

5. (b)
M: N mapping and total participation at student end.

7. (a)
Both A and B are correct ER diagram for the above statement description.

8. (b)
E_1 R can be merged into 1 relation
E_1 S can be kept in one relation
E_2, E_3 kept separate
E_{12} (AC B) with AC key
E_{13} (AE B) with AE key
E_2 (CD) with C key
E_3 (E F) with E key

9. (b)
The above diagram has 3 tables
Manager (Eid, did)
Team lead (Eid, Pid, grade)
Developer (Eid, Pid, cabin-no)

10. (c)
X foreign key references to Y if "X" deleted then "Y" need not deleted.

11. (d)
Both the given statements are true.
S_1: Participation of the weak entity set in identifying relationship should be total because primary key of weak entity set gets defined only by relating it to strong entity and its primary key value.
S_2: Multivalued attributes in E-R diagram require separate tables along with key attribute when converted into relational model.

CHAPTER 02

Database Design and Normalization

2.1 Definition of Keys and Attributes Participating in Keys

Primary Key

Key (primary key) of a relation schema is the minimal set of attributes that uniquely identifies each tuple (row) in the relation which has non-NULL values.

If a relation schema has more than one key, each is called candidate key. One of the candidate key is designated to be primary key and others are called **secondary keys (Alternate keys)**. Alternate keys allowed NULL values.

Super Key

A superkey of a relation schema $R = \{A_1, A_2, ..., A_n\}$ is set of attributes $S \subseteq R$ with the property that no two tuples, t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A key K is a superkey with the additional property that removal of any attribute from K will cause K not to be superkey any more.

The difference between a key and a superkey is that a key has to be minimal; that is, if we have a key $K = \{A_1, A_2, ..., A_k\}$ of R, then $K - \{A_i\}$ is not a key of R for any A_i, $1 \le i \le k$.

Example-2.1 Consider a relation $R(A_1, A_2, A_3, \ldots A_n)$. Find super key of R.

Solution:

Maximum super keys = $2^n - 1$

If each attribute of relation is candidate key.

Example-2.2 Relation $R(A_1, A_2, \ldots A_n)$ with A_1 as the primary key. Then how many super keys possible?

Solution:

A candidate key remaining $A_2\ A_3 \ldots A_n$ any subset of attribute which combine with A_1 is superkey.

Total keys = 2^{n-1}.

Foreign Key

Foreign key is the set of attribute references to the primary key or alternate key of same table or some other table.

2.2 Integrity Constraints

An Integrity Constrains (IC) is a condition specified on a database schema and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constrains specified on the database schema it is legal instance. A DBMS enforces IC, in that it permits only legal instances to be stored in the database.

Integrity constrains are specified and enforces at different times.

(*i*) When a DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.

(*ii*) When a database application is run, the DBMS checks for violations and disallows changes ICs. It is important to specify when ICs are checked, i.e., when a data is inserted, deleted or updated in the table.

Examples of integrity constraints

- Domain constraints
- Referential Integrity Constraints
- Function dependencies (FDs)
- Assertions
- Triggers

2.2.1 Domain Constraints

A relation schema specifies the domain of each field or column in the relation. These domain constraints condition that we want each instance of the relation to the relation to satisfy. The values that appear in a column must be drawn from the domain associated with that column. The domain of a field is essentially the type of that field, in programming language terms, and restricts the values that can appear in the field. The check clause in SQL permits the schema designer to specify a predicate (condition) that must be satisfied by value assigned to a variable whose type is the domain.

2.2.2 Referential Integrity Constraints (RIC)

Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked and perhaps modified to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks. The most common *IC* involving two relations is a **foreign key constraint** or **RIC**.

Suppose we have two relations

Student (Sid, name, gpa)

Enrolled (studid, cid, grade)

Where studid in Enrolled references to the primary key Sid in the students relation. To ensure that only bonafide students can enroll in courses, any value that appears in the studid field of an instance of the Enrolled relation should also appear in the sid field of some tuple in the Students relation. The studid field of Enrolled is called a **foreign key** and **refers** to Students. The foreign key in the referencing relation (Enrolled, in our example) must match the primary key of the referenced relation (Students); that is, it must have the same number of columns and compatible data types, although the column names can be different.

Every value of the referencing attribute (Studid) must be null or available in the referenced attribute (sid) i.e., Studid is subset of Sid. Finally note that a foreign key could refer to the same relation.

Enforcing RIC

SQL provides several alternative ways to handle foreign key violations. We must consider three basic questions:

1. What should we do if an Enrolled row is inserted, with a studid column value that does not appear in any row of the Students table? In this case, the Insert command is simply rejected.
2. What should we do if a Students row is deleted?
   The options are:
   - Delete all Enrolled rows that refer to the deleted students row.
   - Disallow the deletion of the students row if an enrolled row refers to it.
   - Set the studid column to the sid of some (existing) 'default' student, for every enrolled row that refers to the deleted students row.
   - For every enrolled row that refers to it, set the studid column to null. In our example, this option conflicts with the fact that studid is part of the primary key of enrolled and therefore cannot be set to null. Therefore, we are limited to the first three options in our example, although this fourth option (setting the foreign key to null) is available in general.
3. What should we do if the primary key value of a students row is updated?
   The options here are similar to the previous case.
   SQL allows us to choose any of the four options on DELETE and UPDATE. For example, we can specify that when a students row is deleted, all enrolled rows that refer to it are to be deleted as well, but that when the sid column of a students row is modified, this update is to be rejected if an enrolled row refers to the modified students row:

```
CREATE TABLE Enrolled    ( studid CHAR (20),
                           cid  CHAR (20),
                           grade CHAR (10),
                          PRIMARY KEY (studid, cid),
                          FOREIGN KEY (studid) REFERENCES Students (Sid)
                                ON DELETE CASCADE
                                ON UPDATE NO ACTION)
```

The options are specified as part of the foreign key declaration. The default option is NO ACTION, which means that the action (DELETE or UPDATE) is to be rejected. Thus, the ON UPDATE clause in our example could be omitted, with the same effect. The CASCADE keyword says that, if a Students row is deleted, all enrolled rows that refer to it are to be deleted as well. If the UPDATE clause specified CASCADE, and the sid column of a students row is updated, this update is also carried out in each enrolled row that refers to the updated students row.

If a Students row is deleted, we can switch the enrollment to a 'default' student by using ON DELETE SET DEFAULT. The default student is specified as part of the definition of the sid field in enrolled; for example, sid CHAR (20) DEFAULT '53666'. Although the specification of a default value is appropriate in some situations (*e.g.*, a default parts supplier if a particular supplier goes out of business), it is really not appropriate to switch enrollments to a default student. The correct solution in this example is to also delete all enrollment tuples for the deleted student (that is, CASCADE) or to reject the update.

SQL also allows the use of null as the default value by specifying ON DELETE SET NULL.

2.2.3 Functional Dependency (FD)

A functional dependency (FD) is a kind of IC that generalizes the concept of a key. Let R be a relation schema and let X and Y be nonempty sets of attributes in R. We say that an instance r of R satisfies the FD $X \rightarrow Y$. If the following holds for every pair of tuples t_1 and t_2 in r.

If $t_1.X = t_2.X$, then $t_1.Y = t_2.Y$.

$X \rightarrow Y$ is read as X functionally determines Y or simply X determines Y.

An FD $X \rightarrow Y$ essentially says that if two tuples agree on the values in attributes X, they must also agree on the values in attributes Y.

If a constraints on R states that there cannot be more than one tuple with a given X value in any relation instance $r(R)$, that is X is the key of R, however the definition of an FD does not require that the set X be minimal, the additional minimality condition must be met for X to be a key. If $X \rightarrow Y$ holds, where Y is set of all attribute and there is some subset V of X such that $V \rightarrow Y$ holds then X is a super key.

There are two types of FD.

1. **Trivial FD:** If X and Y are attribute set of R and $X \supseteq Y$ then $X \rightarrow Y$ is trivial FD.
   Example: sid → sid
   Sid Sname → sid
   Sid Sname → Sname
   Every trivial FD implies in relation.
2. **Non-trivial FD:** If X and Y are attribute sets of R and no common attribute between X and Y i.e., $X \cap Y = \phi$ then $X \rightarrow Y$ is non- trivial FD.
   Example: sid → gpa
   Sname → sid gpa
   There may be relation with no non-trivial FD.

Example-2.3 Given an instance of relation R ($ABCD$). Find all non-trivial FDs.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_1	b_2	c_2	d_1
a_2	b_1	c_3	d_1

An instance of relation R

Solution:

$AB \rightarrow C$
$C \rightarrow A$
$C \rightarrow B$

2.3 Closure of set of FDs

Set of all FDs that include given FDs as well as those that can be inferred from the given. FDs is called the closure of FDs. If F is the set of given FDs the F^+ is called closure of F.

The following three rules, called Armstrong's Axioms can be applied repeatedly to infer all FDs implied by a set F of FDs. Let X, Y and Z denotes sets of attributes over a relation schema R.

- **Reflexivity:** If $X \supseteq Y$ then $X \rightarrow Y$
- **Augmentation:** If $X \rightarrow Y$ then $XZ \rightarrow YZ$ for any Z
- **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

Armstrong's Axioms are sound, by sound we mean that given a set of FDs F specified on relational schema R, any dependency that we can infer from F by using Armstrong's Axioms holds in every relation r of R that also complete by complete we mean set of dependencies F^+, which are called closure of F can be determined from F by using Armstrong's Axioms only.

It is convenient to use some additional rule while finding F^+.

- Decomposition or projection rule: If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$
- Union or additive rule: If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$

2.4 Attribute Closure

The algorithm for computing the attribute closure of a set X of attributes is given below:

closure = X;

repeat until there is no change {

if there is an FD $U \rightarrow V$ in F such that $U \subseteq$ closure,

then set closure = closure $\cup$ V

}

This algorithm can be modified to find keys by starting with set X containing a single attribute and stopping as soon as closure contains all attributes in the relation schema. By varying the starting attribute and the order in which the algorithm considers FDs, we can obtain all candidate keys.

Example-2.4 Given a relation $R(A, B, C, D, E, F)$ with FDs

$AB \rightarrow C$

$B \rightarrow D$

$AD \rightarrow E$

Compute $(AB)^+$

Solution:

$(AB)^+ \rightarrow AB$

$A \rightarrow ABC$ $\{AB \rightarrow C\}$

$\rightarrow ABCD$ $\{B \rightarrow D\}$

$\rightarrow ABCDE$ $\{AD \rightarrow E\}$

$(AB)^+ \rightarrow ABCDE$

2.5 Membership Test

If we just want to check whether a given dependency $X \rightarrow Y$ is in the closure of set F of FDs. We can do so efficiently without computing F^+ by using closure of X (finding X^+). If X^+ contains Y then $X \rightarrow Y$ is a member of functional dependency set F i.e. $X \rightarrow Y$ is logically implies in F or $F \Rightarrow X \rightarrow Y$.

Example-2.5 Prove or disprove the following inference rule for functional dependency using (*i*) Armstrong's Axioms (*ii*) Attribute closure

$$\{X \rightarrow Y, XY \rightarrow Z\} \Rightarrow \{X \rightarrow Z\}$$

Solution:

(*i*) Armstrong's Axioms:

$X \rightarrow X$ (trivial) and $X \rightarrow Y$ by union rule $X \rightarrow XY$

$X \rightarrow XY$ and $XY \rightarrow Z$ by transitivity $X \rightarrow Z$

(*ii*) **Attribute closure:** If closure of X determines Z in the given FD set then $X \rightarrow Z$ is logically implies in the given FD. $X^+ \rightarrow XYZ$
Since X^+ contains Z, $X \rightarrow Z$ implies in the given FDs.

2.6 Equivalence of Sets of Functional Dependencies

Two sets of functional dependencies E and F are equivalent if E covers F ($E \supseteq F$) and F covers E ($F \supseteq E$). Therefore equivalence means that every FD in E can be inferred from F and every FD in F can be inferred from E.

E covers F	F covers E	Result
Yes	Yes	$E \equiv F$
Yes	No	$E \supset F$
No	Yes	$F \supset E$
No	No	E and F not comparable

Example-2.6 Given below two sets of FDs for a relation R (A, B, C, D, E). Are they equivalent?

$F1$: $\{A \rightarrow B, AB \rightarrow C, D \rightarrow AC, D \rightarrow E\}$
$F2$: $\{A \rightarrow BC, D \rightarrow AE\}$

Solution:
If $F1$ covers $F2$ then every FD in $F2$ logically implies in $F1$.
FDs in $F2$ $A \rightarrow BC$, $D \rightarrow AE$
Check for $A \rightarrow BC$
$(A)^+ \rightarrow ABC$ $\{A \rightarrow B, AB \rightarrow C$ in $F1\}$
Check for $D \rightarrow AE$
$(D)^+ \rightarrow DACE$ $\{D \rightarrow AC, D \rightarrow E$ in $F1\}$
Hence $F1$ covers $F2$ (i.e., $F1 \supseteq F2$)
If $F2$ covers $F1$ then every FD in F1 logically implies in F2.
$F1$ has 4 FDs $\{A \rightarrow B, AB \rightarrow C, D \rightarrow AC, D \rightarrow E\}$
Check for $A \rightarrow B$
$(A)^+ \rightarrow AB$ $\{A \rightarrow BC$ in $F2\}$
Check for $AB \rightarrow C$
$(AB)^+ \rightarrow ABC$ $\{A \rightarrow BC$ in $F2\}$
Check for $D \rightarrow AC$
$(D)^+ \rightarrow DAEBC$ $\{D \rightarrow AE, A \rightarrow BC$ in $F2\}$
Check for $D \rightarrow E$
$(D)^+ \rightarrow DAEBC$ $\{D \rightarrow AE$ in $F2\}$
Hence $F2$ covers $F1$ i.e., $F2 \supseteq F1$
So $F1 \equiv F2$

2.7 Minimal Cover

A minimal cover (canonical cover) of a set of functional dependencies F is a minimum set of dependencies that is equivalent to F. There exist at least one minimal cover E for set of FDs F.

We can think of minimal set of dependencies as being a set of dependencies with no redundancies. If several sets of FDs qualify as minimal covers of F then we can choose the minimal set with smallest number of dependencies.

Minimal Cover Procedure

1. Split FD's of FD set F such that right hand side of FD is a single attribute.
2. Identify extraneous attribute of left hand side of any FD and remove extraneous attribute.
3. Eliminate redundant FD's: Let $X \to Y$ is the FD of FD set F if $(F - \{X \to Y\}) \equiv F$ then $X \to Y$ is redundant FD. Which can be removed from F.

Extraneous Attribute

An attribute of FD is said to be extraneous if we can remove it without changing the closure of set of FDs. If a FD $XY \to Z$ and $X^+ = Y$ or Z in FD set F then Y is extraneous in FD $XY \to Z$.

Example: $F = \{AB \to CD, A \to E, E \to C\}$

$$AB \to C$$
$$AB \to D$$
$$A \to E$$
$$E \to C$$

Since A^+ contains C, B is extraneous in $AB \to C$.

Example-2.7 Consider the following set F of functional dependencies on schema $R(A,B,C)$:

$$A \to BC$$
$$B \to C$$
$$A \to B$$
$$A \to C$$

Find the minimal cover of functional dependency set F.

Solution:

Step-1:

$$A \to BC$$
$$B \to C$$
$$A \to B$$
$$A \to C$$

Step-2: $A \to B, B \to C, A \to C$

$\Rightarrow \quad A \to B, B \to C \qquad \{A \to C \text{ is redundant}\}$

A canonical cover may not be unique.

Example: $F = \{A \to BC, B \to AC \text{ and } C \to AB\}$

$A \to B, A \to C, B \to A, B \to C, C \to A, C \to B$

Canonical cover 1 = $A \to B, B \to C, C \to A, C \to B$

Canonical cover 2 = $A \to B, B \to C, B \to A, C \to B$

Canonical cover 3 = $A \to B, A \to C, B \to A, C \to A$

2.8 Problem caused by redundancy

Storing the same information **redundantly,** that is, in more than one place within a database, can lead to several problems:

- **Redundant Storage:** Some information is stored repeatedly.
- **Update Anomalies:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.

- **Insertion Anomalies:** It may not be possible to store certain information unless some other, unrelated, information is stored as well.
- **Deletion Anomalies:** It may not be possible to delete certain information without losing some other, unrelated, information as well.

2.9 Normalization of Relations

Normalization is the process of decomposition of relation into several relations to achieve the desirable property of:

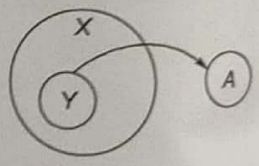
- Minimizing redundancy
- Minimizing the insertion, deletion and update anomalies discussed above.

The normal form of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

The normal forms based of FDs are first normal form (1NF), second normal form (2NF), third normal form (3NF), and Boyce-Codd normal form (BCNF). These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3*NF*, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is in first normal form if every field contains only atomic values, i.e., no lists or sets. This requirement is implicit in our definition of the relational model.

Second normal form (2NF)

Second normal form (2NF) is based on the concept of full functional dependency. A functional dependency $X \rightarrow Y$ is a full function dependency if removal of any attribute A from X means that the dependency does not hold any more; i.e., for any attribute $A \in X$, $(X - \{A\})$ does not functionally determine Y. A functional dependency $X \rightarrow Y$ is a partial dependency if some attribute $A \in X$ can be removed from X and the dependency still holds; i.e., for some $A \in X$, $(X - \{A\}) \rightarrow Y$.



A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R. A relation schema R is in 2NF if every non prime attribute A in R is not partially dependent on any key of R i.e. fully functionally dependent on the primary key of R.

The test for 2NF involves testing for functional dependency whose left hand side attributes are part of the primary key and right hand side is non prime attribute. If the primary key contains a single attribute the test need not to be applied at all

X : is any candidate key
Y : Proper subset of key
A : None prime attribute
$Y \rightarrow A$ is partial dependency.

Example-2.8 Consider a relation $R(A, B, C, D)$ holds following FDs.

$$AB \rightarrow C,\ AB \rightarrow D,\ C \rightarrow A,\ B \rightarrow D$$

Find the normal form of R.

Solution:

Find key of R:

$(AB)^+ \rightarrow ABCD$ $(BC)^+ \rightarrow BCAD$

Prime attribute = $\{A, B, C\}$

Non prime attribute = $\{D\}$

There exist a partial dependency

$$B \rightarrow D$$
$$\downarrow \quad \downarrow$$

Proper subset of key — Non prime attribute

Relation R is in 1NF.

Third Normal Form

The relation schema R is **third normal form (3NF)** if, whenever a nontrivial functional dependency $X \rightarrow A$ holds in R, either (*a*) X is a superkey of R, or (*b*) A is a prime attribute of R.

Third normal form (3NF) is based on the concept of transitive dependency. A function dependency $X \rightarrow Y$ in a relation scheme R is a transitive dependency if there is a set of attributes Z that is neither a candidate key nor a subset of any key of R, and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

A relation schema R is in 3NF if it satisfies 2NF and no nonprime attributed of R is transitively dependent on the primary key.

Example-2.9 Consider a relation $R(ABCD)$ holds following dependencies $AB \rightarrow CD$, $D \rightarrow A$. Find the normal form of R.

Solution:

Key = AB, DB

In $AB \rightarrow CD$,

AB is key hence 3NF

$D \rightarrow A$,

A is prime attribute hence 3NF

Boyce-Codd Normal Form (BCNF)

Every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF. A relation schema R is in BCNF if whenever a nontrivial functional dependency $X \rightarrow A$ holds in R, then X is superkey of R.

Example-2.10 Consider a relation $R(ABC)$ with following FDs $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$. Find the normal form of R.

Solution:

Key = $\{A, B, C\}$ $\Rightarrow$ Every attribute is key. Hence BCNF.

2.10 Properties of Decomposition

Decomposition is a tool that allows us to eliminate redundancy we should check whether a decomposition allows us to recover the original relation and wheather it allows us to check integrity constraints efficiently.

2.10.1 Lossless Join Decomposition

Let R be a relation and F be a set of FDs that holds over R. The decomposition of R into relations with attribute sets $R1$ and $R2$ is lossless.

(*i*) $R_1 \cup R_2 = R$

(*ii*) F^+ contains either the FD $R_1 \cap R_2 \rightarrow R_1$ or the FD $R_1 \cap R_2 \rightarrow R_2$ i.e., common attribute is key for either of the relation.

Example: Consider relation $R(A, B, C, D, E)$ with FD set

$F = \{AB \rightarrow C, C \rightarrow D, B \rightarrow E\}$

(*i*) $R_1(ABC)$ AND $R_2(CD)$

$R_1 \cup R_2 \neq R$

lossy join decomposition.

(*ii*) $R_1(ABC)$ AND $R_2(D,E)$

$R_1 \cup R_2 = R$

$R_1 \cap R_2 = \phi$

lossy join decomposition.

(*iii*) $R_1(ABCD)$ AND $R_2(BE)$

$R_1 \cup R_2 = R$

$R_1 \cap R_2 = B$

$B^+ = BE$

B is key for $R2$ Hence lossless join decomposition.

(*iv*) $R_1(ABC)$ AND $R_2(BDE)$

$R_1 \cup R_2 = R$

$R \cap R_2 = B$

$B^+ \rightarrow BE$

Common attribute is not super key for any relation. Hence lossy join decomposition.

(*v*) $R_1(ABC)$ AND $R_2(ABDE)$

$R_1 \cup R_2 = R$

$R_1 \cap R_2 = AB$

$(AB)^+ = ABCDE$

Loss less decomposition.

2.10.2 Dependency Preserving Decomposition

The decomposition of relation schema R with FDs F into schema with attribute sets X and Y is dependency-preserving if $(F_X \cup F_Y)^+ = F^+$. Let R be the relational schema with FD set F decomposed into $R_1, R_2 \ldots R_n$. With FD sets. $F_1, F_2 \ldots F_n$ respectively.

In General $\{F_1 \cup F_2 \ldots F_n\} \subseteq F$. If $\{F_1 \cup F_2 \ldots F_n\} \supseteq F$ then dependency preserving decomposition.

Example: Suppose $R(ABCD)$ is decomposed into $R_1(AB)$ and $R_2(ACD)$ with FDs F_1 and F_2

$F = A \rightarrow B, B \rightarrow C, C \rightarrow D$

$F_1 = A \rightarrow B$

$F_2 = A \rightarrow CD, C \rightarrow D$

$\{B \rightarrow C\}$ does not implies in $F_1 \cup F_2$

So decomposition is not dependency preserving.

2.11 Multivalued Dependencies

Let R be a relation schema and let X and Y be subsets of the attributes of R. Intuitively, the multivalued dependency $X \rightarrow\rightarrow Y$ is said to hold over R if, in every legal instance r of R, each X value is associated with a set of Y values and this set is independent of the values in the other attributes.

Formally, if the MVD $X \rightarrow\rightarrow Y$ holds over R and $Z = R - (X \cup Y)$ the following must be true for every legal instance r of R:

t_1 and t_2 tuple such that $t_1 \cdot X = t_2 \cdot X$ and alsot t_3 and t_4 tuple such that

X	Y	Z	
a	b_1	c_1	–tuple t_1
a	b_1	c_2	–tuple t_2
a	b_2	c_1	–tuple t_3
a	b_2	c_2	–tuple t_4

1. $t_1 \cdot X = t_2 \cdot X = t_3 \cdot X = t_4 \cdot X$ and
2. $t_1 \cdot Y = t_2 \cdot Y$ and $t_3 \cdot Y = t_4 \cdot Y$ and
3. $t_1 \cdot Z = t_3 \cdot Z$ and $t_2 \cdot Z = t_4 \cdot Z$

This table suggests another way to think about MVDs: If $X \twoheadrightarrow Y$ holds over R, then $\pi_{YZ}(\sigma_{X=x}(R)) = \pi_Y(\sigma_{X=x}(R)) \times \pi_Z(\sigma_{X=x}(R))$ in every legal instance of R, for any value x that appears in the X column of R. In other words, consider groups of tuples in R with the same X-value. In each such group consider the projection onto the attributes YZ. This projection must be equal to the cross-product of the projections onto Y and Z. i.e., for a given X-value, the Y-values and Z-values are independent. (From this definition it is easy to see that $X \twoheadrightarrow Y$ must hold whenever $X \rightarrow Y$ holds. If the FD $X \rightarrow Y$ holds, there is exactly one Y-value for a given X-value, and the conditions in the MVD definition hold trivially. The converse does not hold.

Given a set of FDs and MVDs, in general, we can infer that several additional FDs and MVDs hold. A sound and complete set of inference rules consists of the three Armstrong Axioms plus five additional rules. Three of the additional rules involve only MVDs:

- MVD Complementation: If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow R - (X \cup Y)$.
- MVD Augmentation: If $X \twoheadrightarrow Y$ and $W \supseteq Z$, then $WX \twoheadrightarrow YZ$.
- MVD Transitivity: If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow (Z - Y)$.

As an example of the use of these rules, since we have $C \twoheadrightarrow T$ over GTB, MVD complementation allows us to infer that $C \twoheadrightarrow$ OTB – CT as well, that is, $C \twoheadrightarrow B$. The remaining two rules relate FDs and MVDs:

- **Replication:** If $X \rightarrow$Y, then $X \twoheadrightarrow Y$.
- **Coalescence:** If $X \twoheadrightarrow Y$ and there is a W such that $W \cap Y$ is empty, $W \rightarrow Z$, and $Y \supseteq Z$, then $X \rightarrow Z$.

Observe that replication states that every FD is also an MVT).

2.12 Fourth Normal Form

Fourth Normal form is a direct generalization of BCNF. Let R be a relation schema, X and Y be non empty subsets of the attributes of R, and F be a set of dependencies that includes both FDs and MVDs. R is said to be in fourth normal form (4NF), if, for every MVD $X \twoheadrightarrow Y$ that holds over R, one of the following statements is true:

- $Y \subseteq X$ or $X \cup Y = R$, or
- X is a superkey.

In reading this definition, it is important to understand that the definition of a KEY has not changed the key must uniquely determine all attributes through FDs alone. $X \twoheadrightarrow$ Y is a trivial MVD if $Y \subseteq X \subseteq R$ or $X \cup Y = R$; such MVDs always hold.

Example-2.11 Consider the relation $R(ABCD)$ with FDs set F. $F = \{AB \rightarrow CD, D \rightarrow A\}$. Find the key for R.

Solution:

Find $(AB)^+$ $\quad AB^+ \rightarrow ABCD$

Prime attribute = $\{A, B\}$

Now look at these FD where right hand side is prime attribute.

$D \rightarrow A$

Replace A with D in key $\quad AB \Rightarrow DB$

Key = $\{AB, DB\}$

Example-2.12 Find the best normal form of R in Q. 2.11.

Solution:

Key = $\{AB, DB\}$

Prime attribute = $\{A, B, D\}$

Non prime attribute = $\{C\}$

$AB \rightarrow C$ (BCNF) AB is key

$AB \rightarrow D$ (BCNF) AB is key

$D \rightarrow A$ (3NF) A is prime attribute

Hence R is in 3NF

Example-2.13 Consider the relation $R(ABCDE)$ with following dependencies: $A \rightarrow B$, $BC \rightarrow E$, and $ED \rightarrow A$. (a) List all keys of R (b) Find the normal form of R.

Solution:

(a) $(A)^+ \rightarrow AB$

$(AC)^+ \rightarrow ABCE$

$(ACD)^+ \rightarrow ACDBE$

Look at the FDs where right hand side is A or C or D

$ED \rightarrow A$

Replace A with ED in key ACD

Another Key = ECD

Look at the FDs where right hand side is E or C or D

$BC \rightarrow E$

Replace E with BC in the key ECD

another key = BCD

Keys = $\{ACD, ECD, BCD\}$

(b) Prime attribute = $\{A, B, C, D, E\}$.

When all the attributes are prime attribute the relation is always in 3NF

Check for BCNF: For a relation to be in BCNF left hand side should be key in every FD.

R is not in BCNF it is in 3NF.

Example-2.14 What is trivial functional dependencies?

Solution:

A functional dependency where right hand side is subset of left hand side e.g. if $X \rightarrow Y$ is trivial FD then $X \supseteq Y$.

Example-2.15 Consider the following FDs

Proper subset of key → Non prime attribute ... (a)
Non prime attribute → Non prime attribute ... (b)
Proper subset of key → Proper subset of key ... (c)

Which of the above FDs are allowed/not allowed in 1NF, 2NF, 3NF, BCNF.

Solution:

	a	b	c
1NF	✓	✓	✓
2NF	x	✓	✓
3NF	x	x	✓
BCNF	x	x	x

✓ Allowed x Not allowed

BCNF allowed only FD where left side is key.

Example-2.16 Consider the attribute set $R = ABCDEFG$ and the FD set $F = \{AB \rightarrow C, AC \rightarrow B, AD \rightarrow E, B \rightarrow D, BC \rightarrow A, E \rightarrow F\}$. Compute set of functional dependencies for the relations and find the Normal form for each relation.

(a) ABC
(b) $ABCD$
(c) $ABCEF$

Solution:

(a) $R_1(ABC)$ has FD :

$$AB \rightarrow C$$
$$AC \rightarrow B$$
$$BC \rightarrow A$$
$$\text{Key} \rightarrow \{AB, AC, BC\}$$

Relation R_1 is in BCNF

(b) $R_2(ABCD)$

$$AB \rightarrow C$$
$$AC \rightarrow B$$
$$BC \rightarrow A$$
$$B \rightarrow D$$
$$\text{Key} \rightarrow AB, BC, AC$$

Relation $R_2(ABCD)$ is in 1NF because partial dependency $B \rightarrow D$ exist

(c) $R_3(ABCEF)$ has FDs:

$$AB \rightarrow C$$
$$AC \rightarrow B$$
$$BC \rightarrow A$$
$$E \rightarrow F$$
$$\text{KEY} = ABE, ACE, BCE$$

R_3 is in 1NF because partial dependency exist

$$E \rightarrow F$$

Example-2.17 Suppose the relational schema $R(A,B,C,D,E)$ holds following FDs. $A \rightarrow BC$, $CD \rightarrow E$, $B \rightarrow D$, $E \rightarrow A$. R is decomposed into R_1 (A, B, C) and R_2 (A, D, E) then this decomposition is:

(a) Lossless and dependency preserving
(b) Lossy and dependency preserving
(c) Lossless and not dependency preserving
(d) Lossy and not dependency preserving

Solution:

R_1 (A, B, C) holds $A \rightarrow BC$

R_2 (A, D, E) holds $A \rightarrow DE$

$E \rightarrow AD$

- Common attribute A which is key for R_1 and R_2 so lossless.
- Not dependency preserving

Summary

- We defined the concept of functional dependency and discussed some of its properties. Functional dependencies specify semantic constraints among the attribute of a relation schema. We showed how from a given set of functional dependencies, additional dependencies can be inferred using a set of inference rules. We defined the concepts of closure and cover related to functional dependencies. Then we defined minimal cover of a set of dependencies and provided an algorithm to compute a minimal cover. We also showed how to check whether two sets of functional dependencies are equivalent.
- Next the described the normalization process for achieving good designs by testing relations for undesirable types of problematic functional dependencies. We provided relation, and then relaxed this requirement and provided more general definitions of second normal form (2NF) and third form (3NF) that take all candidate keys of a relations into account. We presented examples to illustrate how by using the general definition of 3NF a given relation may be analyzed and decomposed to eventually yield a set of relations in 3NF.
- Finally, the presented Boyce-Codd normal form (BCNF) and discussed how it is a stronger form of 3NF. We also illustrated how the decomposition of a non-BCNF relation must be done by considering the nonadditive decomposition requirement.
- We first discussed two important properties of decompositions: the nonadditive join property, and the dependency-preserving property. An algorithm to test for nonadditive decomposition and a simpler test for checking the losslessness of binary decompositions, were described.
- We saw that it is possible to synthesize 3NF relation schemas that meet both of the above properties; however, in the case of BCNF, it is possible to aim only for the nonadditiveness of joins-dependency preservation cannot be necessarily guaranteed. If one has to aim for one of these two, the nonadditive join condition is an absolute must.
- Then we defined additional types of dependencies and some additional normal forms. Multivalued dependencies, which arise from an improper combination of two or more independent multivalued attributes in the same relation, that result in a combinational expansion of the tuples are used to define fourth normal form (4NF).

Student's Assignment

Q.1 Consider the following in the given table

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

What is the number of functional dependencies in the canonical cover of this instance?

Q.2 Consider relation r (P, Q, R, S) with functional dependencies

$PQ \rightarrow R$
$PQ \rightarrow S$
$R \rightarrow P$
$S \rightarrow Q$

Find the number of candidate keys in relation.

Q.3 Given a relation R with four attributes A, B, C, D. The following FDs holds for R

$AB \rightarrow C$
$AB \rightarrow D$
$C \rightarrow A$
$D \rightarrow B$

Identify the best normal form that R satisfies?

(a) 1 NF (b) 2 NF
(c) 3 NF (d) BCNF

Q.4 Find the highest normal form of the relation $R(A, B, C, D)$ that holds following FDs.

$A \rightarrow B$,
$B \rightarrow D$
$A \rightarrow C$
$BC \rightarrow A$

(a) 1 NF (b) 2 NF
(c) 3 NF (d) BCNF

Q.5 F and G are two FDs sets

F:	G:
$P \rightarrow Q$	$P \rightarrow R$
$R \rightarrow P$	$R \rightarrow Q$
$PQ \rightarrow R$	$QR \rightarrow P$

Which of the following is correct?

(a) F covers G but G does not cover F
(b) G cover F but F does not cover G
(c) F covers G and G covers F
(d) None of these

Q.6 A relation $R(A, B, C, D, E, F)$ holds following FDs.

$AB \rightarrow C$
$C \rightarrow D$
$D \rightarrow EA$
$E \rightarrow F$
$F \rightarrow B$

Find the number of candidate keys of R.

Q.7 Given R(A, B, C, D) with FDs F = {AB → CD, C → A, B → D} is decomposed into R_1(A, B, C) and R_2(B, C, D) then which of the following statement is true about decomposition of R?

(a) loss less and dependency preserve decomposition
(b) loss less and not dependency preserve decomposition
(c) lossy and dependency preserve decomposition
(d) lossy and not dependency preserve decomposition

Q.8 A relation $R(A, B, C, D, E, F, G)$ holds following FDs.

$B \rightarrow ACD$
$BD \rightarrow E$
$EFG \rightarrow H$
$F \rightarrow GH$

Which of the following FD can be removed without altering the key of the relation R?

(a) $B \rightarrow ACD$ (b) $BD \rightarrow E$
(c) $EFG \rightarrow H$ (d) $F \rightarrow GH$

Q.9 Consider the following relational schema R(P, Q, R, S, T) with FD set {P → QR, RS → T, Q → S, T → P} if the relation decomposed into R_1(P, Q, R) R_2(P, S, T). Which of the following true for given decomposition?

(a) Lossless join decomposition and dependency preserving decomposition.
(b) Lossless join decomposition but not dependency preserving decomposition.

(c) Dependency preserving decomposition but not lossless join.
(d) Not dependency preserving and not lossless join decomposition.

Q.10 Consider the following relational schema:
R(ABCDE) with FD set {A → B, C → D, BD → E, E → C}
How many of given FD's violate 3NF?

Q.11 Consider a relation $R(A, B, C, D)$ with FDs $A \rightarrow B$ and $C \rightarrow D$ then the decomposition of R into $R_1(A, B)$ and $R_2(C, D)$ is
(a) Dependency preserving and lossless join
(b) lossless but *NOT* dependency preserving
(c) Dependency preserving but not lossless join
(d) Not dependency preserving and not lossless join

Q.12 Given a relation $R(A, B, C)$ with FDs set $\{A \rightarrow B, B \rightarrow C, C \rightarrow B\}$
(*i*) Lossless decomposition is always possible for R
(*ii*) Dependency preserving decomposition is always possible for R.
Assume decomposition includes all the attributes of R.
(a) Both (*i*) and (*ii*) true
(b) (*i*) is true and (*ii*) is false
(c) (*i*) is false and (*ii*) is true
(d) Both (*i*) and (*ii*) are false

Q.13 Consider the following statement
P : Canonical cover may not be unique.
Q : $F = \{AB \rightarrow C, A \rightarrow B, B \rightarrow A\}$. Canonical cover of F is unique
Which of the above statement is true?
(a) Both P and Q true
(b) P is true and Q is false
(c) P is false and Q is true
(d) Both P and Q are false

Q.14 Consider a relation $R(A, B, C, D, E)$ holds FDs. $F = \{AB \rightarrow C, C \rightarrow D, B \rightarrow E\}$ is decomposed into $R_1(A, B, C)$ and $R_2(C, D)$ then this decomposition
(a) Lossless and dependency preserving
(b) Dependency preserving and not lossless
(c) Lossless and not dependency preserving
(d) Not dependency preserving and not lossless

Q.15 Consider the following FD set on $R(A, B, C)$
$F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C\}$
The canonical cover of this set is
(a) $A \rightarrow B$ and $B \rightarrow C$
(b) $A \rightarrow BC$ and $B \rightarrow C$
(c) $A \rightarrow BC$ and $A \rightarrow B$
(d) $B \rightarrow BC$ and $AB \rightarrow C$

Q.16 Consider a relation $R(A, B, C, D, E)$ holds FDs $AC \rightarrow B$, $BD \rightarrow C$, $CE \rightarrow D$, $DA \rightarrow E$, $EB \rightarrow A$. Find number of keys that contain attribute A.

Q.17 Consider the following statements
1. Prime attribute transitively determined by super key is allowed by 3NF.
2. Non prime attribute transitively determined by super key is allowed by 3NF.
3. Every partial dependency is transitive dependency.
4. Candidate key is only determined by functional dependencies not by MVD's
Which of the above statements are true.
(a) 1, 2 and 4 only (b) 1, 3 and 4 only
(c) 1, 2 and 3 only (d) All of these

Q.18 How many superkeys in the following Relation R (A B C D E) ($AB \rightarrow C$, $BC \rightarrow D$, $CD \rightarrow A$, $AD \rightarrow B$)
(a) 11 (b) 8
(c) 10 (d) 9

Q.19 Choose the correct statement:
(a) If table R has a forgien key constraint referencing table S then each tuple in R is necessarily related to some tuple in S via forgien key
(b) The SQL statement DELETE FROM R might cause tuples in the table other than just R to be deleted
(c) NULL values can be used to opt a tuple out of enforcement of a forgien key.
(d) Anything that can be expressed in an *ER* diagram via ternary relationship can be expressed in some other logically equivalent way without the use of ternary relationship.

Q.20 Consider the following schema:

$R(\underline{C}, D), Q(\underline{B}, C), P(\underline{A}, B)$

C is forgien key in Q referencing $R(C)$ on delete cascade B is forgien key in P referencing $Q(B)$ on delete set null suppose current content of P, Q, R as follows:

P	A	B
	a	a
	b	b

Q	B	C
	a	a
	b	a

R	C	D
	a	a
	b	a

After executing **delete from** R. What tuples P will contain?

(a) (a, NULL) and (b, b)
(b) (a, NULL) and (b, NULL)
(c) (b, b) only
(d) P will not be changed

Q.21 In most general case, if table R has forgien key constraint referencing table S then

(a) Each tuple in R is related to one or more tuples in S.
(b) Each tuple in R is related to exactly one tuple in S
(c) Each tuple in R is related to zero or one tuple in S
(d) Each tuple in R is related to zero or more tuples in S.

Q.22 Consider the following statements.

(*i*) An entity integrity constraint states that no primary key value can be null
(*ii*) A referential integrity constraint is specified between two relations.
(*iii*) A forgien key can't be used to refer to it's own relation

Identify which of the above statements are correct?

(a) Only (*i*) and (*iii*)
(b) Only (*ii*) and (*iii*)
(c) Only (*i*) and (*ii*)
(d) All of these

Q.23 If both the functional dependencies $X \rightarrow Y$ and $Y \rightarrow X$ hold for two attributes X and Y then the relationship between X and Y is

(a) 1 : 1
(b) M : 1
(c) 1 : M
(d) None of these

Q.24 R(A, B, C, D) is a relation. Which of the following does not have a lossless join dependency preserving BCNF decomposition.

(a) $A \rightarrow B, B \rightarrow CD$
(b) $A \rightarrow B, B \rightarrow C, C \rightarrow D$
(c) $AB \rightarrow C, C \rightarrow AD$
(d) $A \rightarrow BCD$

Q.25 A functional dependency of the form $x \rightarrow y$ is trivial if

(a) $y \subseteq x$
(b) $y \subset x$
(c) $x \subseteq y$
(d) $x \subset y$

Q.26 A relation R(ABC) has no non-trivial functional dependencies, and then what should be the set of candidate keys for the relation?

(a) {ABC}
(b) {A, B, C}
(c) {AB, BC, CA}
(d) None of these

Q.27 If a relation is in BCNF, then which of the following statement is always true?

(a) The relation does not have any type of data redundancy
(b) The relation does not have data redundancy (which is due to functional dependency)
(c) The relation may have data redundancy (which is due to functional dependency)
(d) None of these

Answer Key:

1. (2)	**2.** (4)	**3.** (c)	**4.** (a)	**5.** (c)
6. (3)	**7.** (a)	**8.** (c)	**9.** (b)	**10.** (1)
11. (c)	**12.** (b)	**13.** (b)	**14.** (d)	**15.** (a)
16. (3)	**17.** (b)	**18.** (d)	**19.** (d)	**20.** (b)
21. (c)	**22.** (c)	**23.** (a)	**24.** (c)	**25.** (a)
26. (a)	**27.** (b)			

Student's Assignments Explanations

1. (2)
The above relation have following FDs
$A \to B$
$C \to B$
$AC \to B$
Minimal cover of above relation is $A \to B$ and $C \to B$.
Hence number of FDs is 2.

2. (4)
$(PQ)^+ \to PQRS$
Look the FDs where right hand side is P or Q.
$R \to P \Rightarrow RQ$ is the key (replace P).
$S \to Q \Rightarrow RS$ is the key (replace both P and Q) and PS is the key (replace Q).

3. (c)
Candidate key's AB, BC, CD, AD
Prime attributes: A, B, C, D
Relation is in 3NF but not in $BCNF$.

4. (a)
Keys = A, BC
Non prime attributes: D
$B \to D$
Partial dependency exist so the relation is not in 2NF. It is in 1 NF.

5. (c)
Suppose F covers G then every FD in G logically implies in F
Take $P \to R$
$P^+ \to PQR$
Take $R \to Q$
$R^+ \to RPQ$
Take $QR \to P$
$QR^+ \to QRP$
So F covers G similarly we can prove G covers F.

6. (3)
$(AB)^+ \to ABCDEF$
AB is superkey
$A^+ \to A$
$B^+ \to B$
AB is candidate key take those FDs where right hand side is A or B
$F \to B$, AF is candidate key
other candidate keys are AE, C, D

7. (a)
$R_1(ABC)$:
$C \to A$
$AB \to C$
$BC \to A$
$R_2(BCD)$
$B \to D$
$BC \to D$
$AB^+ = ABCD$

8. (c)
Key for relation R is
$(BF)^+ \to BACDFEGH$
$EFG \to H$
can be removed without altering the key of the relation R.

9. (b)
R(P, Q, R, S, T) {P → QR, RS → T, Q → S, T → P}
LLJ Test: $R_1 \cap R_2 = P^+ = PQRST$
$R_1 \cap R_2$ key for both relations
Given decomposition lossless join
Dependency preserve test:

$R_1(PQR)$	$R_2(PST)$
{P → QR	P → ST
	T → PS}

Because of decomposition RS → T, Q → S, lost.
Not dependency preserving decomposition.

10. (1)
Candidate keys: {AC, AE, AD}
$(AC)^+ = ABCDE$
$(AE)^+ = ABECD$
$(AD)^+ = ABCDE$
Only A → B failed 3NF definition.

11. (c)
$R_1(A, B)$ and $R_2(C, D)$
No common attribute so not lossless

12. (b)
$R_1(A, C)$ and $R_2(A, B)$. Not dependency preserving.
$B \to C$, $C \to B$ is lost

13. (b)

$F = AB \rightarrow C$

$A \rightarrow B,$

$B \rightarrow A$

have 2 canonical cover

$F_{min1} = A \rightarrow C$

$A \rightarrow B$

$B \rightarrow A$

$F_{min2} = B \rightarrow C$

$A \rightarrow B$

$B \rightarrow A$

14. (d)

$R_1 \cup R_2 \neq R$ lossy join decomposition

Attribute E is not present in $R_1 \cup R_2$

$B \rightarrow E$ is lost so not dependency preserving.

15. (a)

Check extraneous attribute in *LHS*

$A \rightarrow BCA \rightarrow BC$

$B \rightarrow C \Rightarrow B \rightarrow C$

$A \rightarrow B$ and $A \rightarrow C$

C is extraneous in *RHS*

$A \rightarrow B$

$B \rightarrow C$

16. (3)

$AB^+ \rightarrow AB$

$AC^+ \rightarrow ABC$

$AD^+ \rightarrow ADE$

$AE^+ \rightarrow AE$

$ABD^+ \rightarrow ABCDE$ ✓

$ACD^+ \rightarrow ABCDE$ ✓

$ABE^+ \rightarrow ABE$

$ACE^+ \rightarrow ABCDE$ ✓

$ADE^+ \rightarrow ADE$

Keys *ABD, ACD, ACE*

18. (d)

Key for above relation are (*ABE, BCE, CDE, ADE*)

$ABE \rightarrow ABCE, ABDE, ABCDE$

$BCE \rightarrow ABCE, BCDE$

$CDE \rightarrow ACDE, BCDE$

$ADE \rightarrow ABDE, ACDE$

Number of super keys = 9

19. (d)

Only statement (d) is false.

20. (b)

After executing above common table Q will be empty and table P contains (a, NULL) and (*b*, NULL)

21. (c)

Since *FK* is primary key in other table and it's value is subset of primary key values so the table containing *FK* have atmost one tuple in other table where it is related.

22. (c)

Only (*i*) and (*ii*) are correct.

CHAPTER 03

Relational Algebra

3.1 Introduction

Relational Algebra (RA) is a procedural query language i.e. internal implementation is required to retrieve the data. Queries in RA are composed using a collection of operators. Every operator in RA accepts (one or two) relation instances as arguments and returns a relation instance as the result. This makes it easy to compose operator to form a complex query called a **relation algebra expression**, defined to be a relation, a **uniary** operator is applied to a single relation, or a binary operator applied to 2 expressions.

There are 6 basic operators of RA

Selection, projection, union, cross-product, set difference and rename.

3.2 Selection and Projection

RA includes operators to select rows (σ) from a relation and to project columns (π). These operation allow us to manipulate data in a single relation i.e. they are uniary operators

The select operation

In general SELECT operation is denoted by

$\sigma_{\theta}(R)$

σ (Sigma): SELECT operator

θ : Selection condition

R: Relation or relational algebraic expression.

In general, the selection condition is a Boolean combination (*i.e.* an expression using logical connective $\wedge$ and $\vee$) of terms that have the form attribute *OP* constant or attribute1 *OP* attribute2 where *OP* is the comparison operators $<, <=, =, \neq, >=, >$.

- SELECT Operator is Unary
- In the result table from SELECT operation number of attribute is same as R.
- Number of tuples in the resulting relation is always less than or equal to number of tupples in R *i.e.* $|\theta_c(R)| \leq |R|$ for any condition c.

- SELECT operation is commutative

$$\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$$

We can combine a **cascade** of SELECT operations into a single SELECT operation with conjuction (*AND*) condition

$$\sigma_{c1}(\sigma_{c2}(\sigma_{c3}\cdots)) = \sigma_{c1 \wedge c2 \wedge c3\cdots}$$

The project operation

The project operation selects certain columns (attribute) from the table (Relation)

The general form of project operation

$\pi_{<\text{attribute list}>}(R)$

If attribute list includes only nonkey attributes of *R*, duplicate tuples are likely to occur. So the result of project operation is a set of tuples and hence a valid relation. If duplicates are not eliminated, the result would be a multiset or bag of tuples rather than a set. Duplicates are allowed in practice but not in formal relational model.

- Commutativity does not hold on project.
- The number of tuples in a relation from a PROJECT operation is always less than or equal to the number of tuples in *R*.
- If the projection list is a superkey of *R* the resulting relation has same number of tuples as *R*.
- $\pi_{<\text{list 1}>}(\pi_{<\text{list 2}>}(R)) = \pi_{<\text{list 1}>}(R)$ iff list 2 $\supseteq$ list 1

3.3 Set Operations

The set operations union ($\cup$), intersection ($\cap$), set difference (–) and cross product (X) are also available in relational algebra.

Union

R U S returns a relation instance containing all tuples that occur in *either* relation instance *R* or relation instance *S* (or both). *R* and *S* must be *union-compatible*, and the schema of the result is defined to be identical to the schema of *R*.

Two relation instances are said to be **union-compatible** if the following conditions hold:

- They have the same number of the fields, and
- Corresponding fields, taken in order from left to right, have the same *domains*.

Note that field names are not used in defining union-compatibility. for convenience, we will assume that the fields of *R* U S inherit names from *R*, if the fields of *R* have names.

Intersection

R $\cap$ S returns a relation instance containing all tuples that occur in *both R* and *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.

Set-difference

R-S returns a relation instance containing all tuples that occur in *R* but not in *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.

Cross-product

R × S returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result of R × S contains One tuple (*r*, *s*) (the concatenation of tuples *r* and *s*) for each pair of tuples $r \in R$, $s \in S$. The cross-product opertion is sometimes called Cartesian product.

3.4 The Rename Operation

Unlike relations in the database, the results of relational-algebra expression do not have a name that we can use to refer to them. It is useful to be able to give them names the rename operator, denoted by the lower case Greek letter rho (ρ) allow us to do this. The general rename operation can be expressed by any of the following:

1. $\rho_s(R)$: Renaming R to S.
2. $\rho_s(B_1, B_2, B_3, \ldots B_n)$: is a renamed relation S based on R, with column names $B_1, B_2, \ldots B_n$.

3.5 Joins

The join operation is one of the most useful operations in relational algebra and the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-product. There are following join operation.

- Conditional join
- Equi join
- Natural join

} Inner joins (Result is only matching tuples)

- Outerjoin

Condition Joins

The most general version of the join operation accepts a **join condition** c and a pair of relation instances as arguments and returns a relation instance. The **join condition** is identical to a **selection condition** in form. The operation is defined as follows:

$$R \bowtie_c S = \sigma_c(R \times S)$$

Thus $\bowtie$ is defined to be a cross-product followed by a selection. Note that the condition c can (and typically does) refer to attributes of both R and S. The reference to an attribute of a relation, say, R, can be by position (of the form $R.i$) or by name (of the form R.name).

Equi Join

A common special case of the join operation $R \bowtie S$ is when the join condition consists solely of equalities (connected by $\wedge$) of the form R.name1 = S.name2, that is, equalities between two fields in R and S. In this case, obviously, there is some redundancy in retaining both attributes in the result. For join conditions that contain only such equalities, the join operation is refined by doing an additional projection in which S.name2 is dropped. The join operation with this refinement is called equijoin.

The schema of the result of an equijoin contains the fields of R (with the same names and domains as in R) followed by the fields of S that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from R and S, they are unnamed in the result relation.

Natural Join

A further special case of the join operation $R \bowtie S$ is an equijoin in which equalities are specified on all fields having the same name in R and S. In this case, we can simply omit the joint condition; the default is that the join condition is a collection of equalities on all common fields. We call this special case a natural join, and it has the nice property that the result is guaranteed not to have two fields with the same name.

If the two relations have no attributes in common, $S \bowtie R$ is simply the cross-product.

Outer Joins

Left outer join : ($\bowtie$)

Natural joins ($R \bowtie S$) and tuples from R those are failed in join condition

if $|R| = n$

$|S| = m$

$n \le |R \bowtie S| \le n \times m$

Right outer join : ($R \bowtie S$)

Natural joins ($R \bowtie S$) and tuples from S those are failed in join condition

if $|R| = n$

$|S| = m$

$m \le |R \bowtie| \le n \times m$

Full outer join : ($R \bowtie S$)

$(R \bowtie S) \cup (R \bowtie S)$

$\max(n, m) \le |R \bowtie S| \le n \times m$

3.6 Division

The division operation is useful for expressing certain kinds of queries for example "Find the names of students who have enrolled in **all** courses".

Consider two relation instances A and B in which A has (exactly) two fields x and y and B has just one field y, with the same domain as in A. We define the division operation A/B as the set of all x values (in the form of unary tuples) such that for every y value in (a tuple of) B, there is a tuple (x, y) in A.

Another way to understand division is as follows. For each x value in (the first column of) A, consider the set of y values that appear in (the second field of) tuples of A with that x value. If this set contains (all y values in) B, the x values is in the result of A/B.

An analogy with integer division may also help to understand division. For integers A and B, A/B is the largest integer Q such that $Q * B \le A$. for relation instances A and B, A/B is the largest relation instance Q such that $Q \times B \subseteq A$.

Expressing A/B *in* terms of the basic algebra operators is an interesting exercise, and the reader should try to do this before reading further. The basic idea is to compute all : r values in A that are not *disqualified.* An x value is *disqualified* if by attaching a y value from B, we obtain a tuple (x, y) that is not in A. We can compute disqualified tuples using the algebra expression.

$\pi_x((\pi_x(A) \times B) - A)$

Thus, we can define A/B as,

$\pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$

Operation	Purpose	Notation
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{<\text{selection condition}>}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{<\text{attribute list}>}(R)$
CONDITIONAL JOIN (THETA JOIN)	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{<\text{join condition}>} R_2$

EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\text{<join condition>}} R_2$ OR $R_1 \bowtie_{(\text{<join attributes>}), (\text{<join attributes 2>})} R_2$
NATURAL JOIN	Same as *EQUIJOIN* except that the join attributes of R_2 are not included in the resulting relation; if the joint attributes have the same names, they do not have to be specified at all.	$R_1 *_{\text{<join condition>}} R_2$, OR $R_1 *_{(\text{<join attributes 1>}), (\text{<join attributes 2>})} R_2$ OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
SET DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t(X)$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z)/R_2(Y)$

3.7 The Tuple Relational Calculus

The tuple relational calculus is a nonprocedural language. (The relational algebra was procedural.) We must provide a formal description of the information desired.

A query in the tuple relational calculus is expressed as $\{t \mid P(t)\}$

i.e. the set of tuples t for which predicate P is true.

We also use the notation

- $t[a]$ to indicate the value of tuple t on attribute a.
- $t \in r$ to show that tuple t is in relation r.

Example Queries

(*i*) For example, to nd the branch-name, loan number, customer name and amount for loans over $1200:

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

This gives us all attributes, but suppose we only want the customer names. (We would use project in the algebra.) We need to write an expression for a relation on scheme (cname).

$$\{t \mid \exists s \in \text{borrow}\ (t[\text{cname}] = s[\text{cname}] \wedge s[\text{amount}] > 1200)\}$$

In English, we may read this equation as "the set of all tuples t such that there exists a tuple s in the relation borrow for which the values of t and s for the cname attribute are equal, and the value of s for the amount attribute is greater than 1200."

The notation $\exists t \in r(Q(t))$ means "there exists a tuple t in relation r such that predicate $Q(t)$ is true". How did we get the above expression? We needed tuples on scheme *cname* such that there were tuples in borrow pertaining to that customer name with amount attribute > 1200.

The tuples t get the scheme *cname* implicitly as that is the only attribute t is mentioned with.
Let's look at a more complex example.
Find all customers having a loan from the SFU branch, and the the cities in which they live:

$$\{t \mid \exists s \in \text{borrow}(t[cname] = s[cname] \wedge s[bname] = \text{"SFU"}$$
$$\wedge \exists u \in customer(u[cname] = s[cname] \wedge t[ccity] = u[ccity]))\}$$

In English, we might read this as "the set of all (*cname,ccity*) tuples for which cname is a borrower at the SFU branch, and ccity is the city of *cname*".
Tuple variable s ensures that the customer is a borrower at the SFU branch. Tuple variable u is restricted to pertain to the same customer as s, and also ensures that ccity is the city of the customer.
The logical connectives ^ (AND) and U (OR) are allowed, as well as ¬ (negation).
We also use the existential quantier ∃ and the universal quantifier ∀.

Some more examples:

(*i*) Find all customers having a loan, an account, or both at the SFU branch:

$$\{t \mid \exists s \in \text{borrow}(t[cname] = s[cname] \wedge s[bname] = \text{"SFU"})$$
$$\vee \exists u \in \text{deposit}(t[cname] = u[cname] \wedge u[bname] = \text{"SFU"})\}$$

Note the use of the connective.
As usual, set operations remove all duplicates.

(*ii*) Find all customers who have **both** a loan and an account at the SFU branch.
Solution: simply change the _ connective in 1 to a ^.

(*iii*) Find customers who have an account, but **not** a loan at the SFU branch.

$$\{t \mid \exists u \in deposit(t[cname] = u[cname] \wedge u[bname] = \text{"SFU"})$$
$$\wedge \neg \exists s \in borrow(t[cname] = s[cname] \wedge s[bname] = \text{"SFU"})\}$$

(*iv*) Find all customers who have an account at all branches located in Brooklyn. (We used division in relational algebra.)

For this example we will use implication, denoted by a pointing nger in the text, but by) here. The formula $P \Rightarrow Q$ means P implies Q, or, if P is true, then Q must be true.

$$\{t \mid \forall u \in branch\,(u[bcity] = \text{"Brooklyn"} \Rightarrow$$
$$\exists s \in deposit(t[cname] = s[cname] \wedge u[bname] = s[bname]))\}$$

In English: The set of all cname tuples t such that for all tuples u in the branch relation, if the value of u on attribute bcity is Brooklyn, then the customer has an account at the branch whose name appears in the bname attribute of u.
Division is dicult to understand. Think it through carefully.

Formal Denitions

(*i*) A tuple relational calculus expression is of the form

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

where P is a **formula**. Several tuple variables may appear in a formula.

(*ii*) A tuple variable is said to be a **free variable** unless it is quantied by a ∃ or a ∀. Then it is said to be a **bound variable**.

(*iii*) A formula is built of **atoms**. An atom is one of the following forms:

- $s \in r$, where s is a tuple variable, and r is a relation ($\notin$ is not allowed).

- $s[x]\ \Theta\ u[y]$, where s and u are tuple variables, and x and y are attributes, and Θ is a comparison operator ($<;\leq;=;\neq,>;\geq;$).

  $s[x]\ \Theta\ c$, where c is a constant in the domain of attribute x.

(*iv*) Formulae are built up from atoms using the following rules:

- An atom is a formula.
- If P is a formula, then so are $\neg P$ and (P).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$ and $P_1 \Rightarrow P_2$.
- If $P(s)$ is a formula containing a free tuple variable s, then $\exists s \in r\,(P(s))$ and $\forall\, s \in r\,(P(s))$ are formulae also.

(*v*) Note some equivalences:

- $P_1 \wedge P_2 = \neg(\neg P_1 \vee \neg P_2)$
- $\forall t \in r(P(t)) = \neg\exists t \in r\,(:P(t))$
- $P_1 \Rightarrow P_2 = \neg P_1 \vee P_2$

Safety of Expressions

(*i*) A tuple relational calculus expression may generate an infinite expression, *e.g.* $\{t \mid \neg(t \in \text{borrow})\}$.

(*ii*) There are an infinite number of tuples that are not in borrow! Most of these tuples contain values that do not appear in the database.

(*iii*) **Safe Tuple Expressions**

We need to restrict the relational calculus a bit.

- The domain of a formula P, denoted dom(P), is the set of all values referenced in P.
- These include values mentioned in P as well as values that appear in a tuple of a relation mentioned in P.
- So, the domain of P is the set of all values explicitly appearing in P or that appear in relations mentioned in P.
- dom($t \in$ borrow $\wedge$ t[*amount*] < 1200) is the set of all values appearing in *borrow*.
- dom($t \mid \neg(t \in$ borrow)) is the set of all values appearing in *borrow*.

We may say an expression $\{t \mid P(t)\}$ is **safe** if all values that appear in the **result** are values from dom(P).

(*iv*) A safe expression yields a nite number of tuples as its result. Otherwise, it is called **unsafe**.

Expressive Power of Languages

The tuple relational calculus restricted to safe expressions is equivalent in expressive power to the relational algebra.

3.8 The Domain Relational Calculus

Domain variables take on values from an attribute's domain, rather than values for an entire tuple.

Formal Denitions

(*i*) An expression is of the form

$$\{(x_1; x_2, \ldots, x_n) \mid P(x_1; x_2, \ldots, x_n)\}$$

where the x_i; $1 \leq i \leq n$, represent domain variables, and P is a **formula**.

(*ii*) An atom in the domain relational calculus is of the following forms

- $(x_1, \ldots x_n) \in r$ where r is a relation on n attributes, and x_i; $1 \le i \le n$, are domain variables or constants.
- $x \Theta y$, where x and y are domain variables, and is a comparison operator.
- $x \Theta c$, where c is a constant.

(*iii*) Formulae are built up from atoms using the following rules:

- An atom is a formula.
- If P is a formula, then so are $\neg P$ and (P).
- If P_1 and P_2 are formulae, then so are $P_1 \wedge P_2$, $P_1 \wedge P_2$ and $P_1 \Rightarrow P_2$.
- If $P(x)$ is a formula where x is a domain variable, then so are $\exists x(P(x))$ and $\forall x(P(x))$.

Example Queries

(*i*) Find branch name, loan number, customer name and amount for loans of over $1200.

$$\{(b, l, c, a) | (b, l, c, a) \in \text{borrow} \wedge a > 1200\}$$

(*ii*) Find all customers who have a loan for an amount > than $1200.

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

(*iii*) Find all customers having a loan from the SFU branch, and the city in which they live.

$$\{(c, x) | \exists b, l, a((b, l, c, a) \in \text{borrow}$$
$$\wedge b = \text{"SFU"} \wedge \exists y((c < y, x) \in \text{customer}))\}$$

(*iv*) Find all customers having a loan, an account or both at the SFU branch.

$$\{(c) | \exists b, l, a((b, l, c, a) \in \text{borrow} \wedge b = \text{"SFU"})$$
$$V \exists b, a, n((b, a, c, n) \in \text{deposit} \wedge b = \text{"SFU"}\}$$

(*v*) Find all customers who have an account at all branches located in Brooklyn.

$$\{(c) | \forall x, y, z(\neg((x, y, z) \in \text{branch})$$
$$V z \neq \text{"Brooklyn"} V (\exists a, n((x, a, c, n) \in \text{deposit}))))\}$$

If you nd this example dicult to understand, try rewriting this expression using implication, as in the tuple relational calculus example. Here's my attempt:

$$\{((cn) | \forall bn, as, bc$$
$$(((bn, as, bc) \in \text{branch} \wedge bc = \text{"Brooklyn"}) \Rightarrow \exists acct, bal((bn, acct, cn, bal) \in \text{deposit}))\}$$

I've used two letter variable names to get away from the problem of having to remember what x stands for.

Safety of Expressions

(*i*) As in the tuple relational calculus, it is possible to generate innite expressions. The solution is similar for domain relational calculusIrestrict the form to safe expressions involving values in the domain of the formula.

Read the text for a complete explanation.

Expressive Power of Languages

(*i*) All three of the following are equivalent:

The relational algebra.

The tuple relational calculus restricted to safe expressions.

The domain relational calculus restricted to safe expressions.

Example-3.1 Consider the following scheme

Supplier (sid, Sname, rating)

Parts (Pid, pname, color)

Catalog (sid, Pid, cost)

The catalog relation lists the prices charged for parts by suppliers.

Write the following queues in relational algebra.

(*i*) Find name of suppliers who supply some red part.

(*ii*) Find the sids of suppliers who supply some red part or green part

(*iii*) find the sids of supplier who supply some red part and some green part

(*iv*) Find the sids of supplier who supply every part

(*v*) Find the sids of suppliers who supply every red part

(*vi*) Find the pids of parts supplied by two different supplier.

(*vii*) Find pair of sids such that the supplier with the first sid charges more for some part than the supplier with the second sid.

(*viii*) Find the sid of supplier who supply most expensive part.

Solution:

(*i*) First find the sids of suppliers who supply some red parts

$temp_1 = \pi_{sid}(\sigma_{color='red'}(parts) \bowtie catalog)$

Name of supplier who supply some red part $\pi_{sname}(temp \bowtie supplier)$

(*ii*) Sids of supplier who supply some red part or green part

$\pi_{sid}(\sigma_{color='red' \vee color='green'}(parts) \bowtie catalog)$

(*iii*) Sid of supplier who supply some red part and some green part.

$temp_2 = \pi_{sid}(\sigma_{color='green'}(parts) \bowtie catalog)$

$temp_1 \cap temp2$

(*iv*) Sids of supplier who supply every parts

$\pi_{sid, Pid}(catalog) / \pi_{Pid}(parts)$

(*v*) Sid's of supplier who supply every red part

$\pi_{sid, Pid}(catalog) / \pi_{Pid}(\sigma_{color='red'}(parts)$

(*vi*) $\rho(C_1, catalog)$

$\rho(C_2, catalog)$

$$\pi_{Pid}\left(\sigma_{\substack{C_1.Pid = C_2.Pid \\ C_2.Sid \neq C_2.Sid}}(C_1 \times C_2)\right)$$

(*vii*) $$\pi_{C_1 Sid,\, C_2 sid}\left(\sigma_{\substack{C_1.Pid = C_2.Pid \\ C_1.Sid \neq C_2.Sid \\ C_1.cost > C_2.cost}}(C_1 \times C_2)\right)$$

(*viii*) First find the supplier who supplies atleast one part whose cost is less than some supplier

$$Temp_3 = \pi_{C_1.Sid, C_1.Pid, C_1.cost}\left(\sigma_{C_1.cost < C_2.cost}(C_1 \times C_2)\right)$$

$\pi_{Sid}(catalog - temp_3)$

Summary

- We introduced the basic relational algebra operations and illustrated the types of queries for which each is used. First, we discussed the unary relational operators SELECT and PROJECT, as well as the RENAME operation. Then, we discussed binary set theoretic operations requiring that relations on which they are applied be union (or type) compatible; these include UNION, INTERSECTION, and SET DIFFERENCE. The CARTESIAN PRODUCT operation is a set operation that can be used to combine tuples from two relations, producing all possible combinations. It is rarely used in practice; however, we showed how CARTESIAN PRODUCT followed by SELECT can be used to define matching tuples from two relations and leads to the JOIN operation. Different JOIN operations called THETA JOIN, EQUIJOIN, and NATURAL JOIN were introduced.
- We discussed recursive queries, for which there is no direct support in the algebra but which can be handled in a step-by-step approach, as we demonstrated. Then we presented the OUTER JOIN and OUTER UNION operations, which extend JOIN and UNION and allow all information in source relations to be preserved in the result.
- The last two sections described the basic concepts behind relational calculus, which is based on the branch of mathematical logic called predicate calculus. There are two types of relational calculus: (1) the tuple relational calculus, which uses tuple variables that range over tuples (rows) of relations, and (2) the domain relational calculus, which uses domain variables that range over domains (columns of relations). In relational calculus, a query is specified in a single declarative statement, without specifying any order or method for retrieving the query result. Hence, relational calculus is often considered to be a higher-level *declarative* language than the relational algebra, because a relational calculus expression states *what* we want to retrieve regardless of *how* the query may be executed.
- We discussed the syntax of relational calculus queries using both tuple and domain variables. We introduced query graphs as an internal representation for queries in relational calculus.We also discussed the existential quantifier and the universal quantifier. We saw that relational calculus variables are bound by these quantifiers. We described in detail how queries with universal quantification are written, and we discussed the problem of specifying safe queries whose results are finite. It is the quantifiers that give expressive power to the relational calculus, making it equivalent to the basic relational algebra. There is no analog to grouping and aggregation functions in basic relational calculus, although some extensions have been suggested.

Student's Assignments 1

Q.1 Suppose relation R(ABC) has the tuples:

R	A	B	C
	1	2	3
	1	2	3
	3	2	1

How many tuples resulted by given RA expression?

$\pi_{AB}(R) \underset{R.B < S.B}{\bowtie} \rho_{S(A,B)}(\pi_{BC}(R))$

(a) 6 (b) 2
(c) 3 (d) 4

Q.2 Consider the following schema:
Student (Sid, Sname, Sage)
Course (Cid, CourseName, Credit)
Enrolled (Sid, Cid)

$Q: p_{Sid}(\text{Student}) - p_{Sid}(s_{credit<5}(\text{course}) \bowtie \text{Enrolled})$

Which of the following is correct about relationanl algebra *Q*?

(a) Finds Sid of students who enrolled atleast one course with credit greater than 5.
(b) Finds Sid of students who enrolled all courses with credit ≥ 5.
(c) Finds Sid of students who enrolled all courses with credit less than 5.
(d) None of these

Q.3 Which of the following is not a basic relational algebra operator?

(a) Union (b) Intersection
(c) Set difference (d) Cross product

Q.4 Consider two relations $R(A_1, A_2, A_3, \ldots, A_m)$ and $S(A_1, A_2, A_3 \ldots A_n)$ where $m > n$. Find the number of attributes that appears in R/S.

(a) $\left\lfloor \frac{n}{m} \right\rfloor$ (b) $\left\lfloor \frac{m}{n} \right\rfloor$
(c) $m - n$ (d) $m + n$

Q.5 Relation supplier (Sid, Sname, Saddress) has 100 tuples and relation catalog (Sid, Pid, Pname) has 50 tuples. Find maximum and minimum tuples that can be present in (supplier ⋈ catalog). (Assume no foreign key present in the relations)

(a) 100, 100 (b) 50, 50
(c) 50, 0 (d) 100, 50

Q.6 Consider the selection of the form $S_{A \le 200}(r)$ where r is a relation with 1000 tuples. Assume attribute values of A among the tuples are uniformly distributed in interval [1, 500]. Finds the number of tuples return by given query.

(a) 100 (b) 200
(c) 300 (d) 400

Q.7 Consider the relation $P(A, B, \underline{C})$, $Q(\underline{C}, D, \underline{E})$ and $R(\underline{E}, F)$ having tuples 200, 300 and 100 respectively. Estimate number of tuples in relation $P \bowtie Q \bowtie R$

(a) 100 (b) 200
(c) 300 (d) 400

Q.8 Consider the following statements.

(*i*) If $R(A, B)$ and $S(A, B)$ are any two relations then $\pi_A(R \cup S) = \pi_A(R) \cup \pi_A(S)$.
(*ii*) If $P(A, B)$ and $Q(A, B)$ are two relations then $\pi_A(R \cap S) = \pi_A(P) \cap \pi_A(Q)$.

Which of the above statements is/are correct?

(a) Only (*i*) (b) Only (*ii*)
(c) Both (*i*) and (*ii*) (d) None of these

Q.9 Consider the relation $R(A, B, C, D)$. Which relational algebra expression to return the lowest value of A?

(a) $\pi_{R_1.A}\left(R_1 \underset{R_1.A < R_2.A}{\bowtie} R_2\right)$

(b) $\pi_A(R) - \pi_{R_1.A}\left(R_1 \underset{R_1.A > R_2.A}{\bowtie} R_2\right)$

(c) $\pi_A(R) - \pi_{R_1.A}\left(R_1 \underset{R_1.A < R_2.A}{\bowtie} R_2\right)$

(d) None of these
(R_1 and R_2 are just renames of R)

Q.10 Consider drinker data base with the following relations:
Drinker(drinkerName, age, city)
Bar (barName, owner, city)
Freq (drinkerName, barName)

Which of the following query finds all drinkers who frequents only bars on same city where he lives?

(a) $\pi_{drinkerName}$(Freq ⋈ (Drinker ⋈$_{Drinker.city = Bar.city}$ Bar))

(b) $\pi_{drinkerName}$ (Freq) − $\pi_{drinkerName}$(Freq ⋈ (Drinker ⋈ (Drinker ⋈$_{Drinker.city1 = Bar.city}$ Bar))

(c) Both (a) and (b)

(d) None of these

Q.11 Consider two relations enrolled and course as shown below

Enrolled		
Sid	Cid	Fees
S_1	C_1	10
S_1	C_2	20
S_2	C_3	30
S_3	C_4	40

Course		
Cid	Cname	Dept
C_1	ALGO	CS
C_2	DS	CS
C_3	TOC	CS
C_4	THERMO	ME

$\pi_{Sid,Cid}$ (Enrolled) / π_{Cid} ($\sigma_{Dept = 'EE'}$ (Course)).

Above relational algebra query executes over above data base table, then how many tuples are there in the result of query?

(a) 4 (b) 3

(c) 0 (d) 1

Q.12 Consider the relations P(A, B) and Q(A, B) where P has foreign key referencing Q via B and Q has foreign key referencing P via A. Which of the following is guaranteed to produce fewer than or at most the same number of tuples as any of the other?

(a) P ⋈ p_B(Q) (b) p_A(P) ⋈ Q

(c) P ⋈ Q (d) P ∩ Q

Q.13 Let r(R) be a relation on schema R and s(S) a relation on schema S. To perform the operation s/r. The attribute set of R and S must satisfy.

(a) R ⊆ S (b) S ⊆ R

(c) R ∩ S = ϕ (d) R − S = ϕ

Q.14 Consider the following DB relation

Student (Sid, Sname, gender, marks)

Which of the following queries correct to retrieve female students scored more marks than marks of all male student?

Q_1 $\pi_{Sid}\left(\frac{\sigma(Student)}{gender = female}\right) - \pi_{Sid}\left(\frac{Student \bowtie \rho(Student)}{gender = I, N, G, M\ female \wedge G = male \wedge marks >= M}\right)$

Q_2 $\pi_{Sid}\left(\frac{\sigma(Student)}{gender = female}\right) - \pi_{Sid}\left(\frac{Student \bowtie \rho(Student)}{gender = I, N, G, M\ female \wedge G = male \wedge marks >= M}\right)$

Q_3 $\pi_{Sid}\left(\frac{\sigma(Student)}{gender = female}\right) - \pi_{I}\left(\frac{Student \bowtie \rho(Student)}{gender = I, N, G, M\ male \wedge G = female \wedge marks >= M}\right)$

(a) Q_1 only correct

(b) Q_2 only correct

(c) Q_3 only correct

(d) None of the query correct

Q.15 Relational schema $R(A, B, C)$ and $S(C, D, E)$ with $\{C \to A, A \to B, C \to D, D \to E\}$ functional dependencies and R has n tuples S has m tuples. How many maximum tuples resulted by $R \bowtie S$?

(a) $n \times m$ (b) $n + m$

(c) min (n, m) (d) max (n, m)

Q.16 Relational algebra is

(a) Data definition language

(b) Meta language

(c) Procedural query language

(d) None of the above

Q.17 Consider the following two tables.

R_1

A	B	C
1	2	3
1	2	4
2	1	3
3	1	3

R_2

A	B	D
1	2	1
2	1	5
4	2	1
3	2	1

The number of rows where null entries are present in the table $R_1 ⟗ R_2$ (R_1 natural full outer join R_2) is _____.

Answer Key:

1. (b)	**2.** (b)	**3.** (b)	**4.** (c)	**5.** (c)
6. (d)	**7.** (a)	**8.** (a)	**9.** (b)	**10.** (b)
11. (b)	**12.** (c)	**13.** (a)	**14.** (c)	**15.** (c)
16. (c)	**17.** (7)			

Student's Assignments Explanations

1. (b)

$\pi_{AB}(R) \underset{R.B < S.B}{\bowtie} \rho_{S(A,B)}(\pi_{BC}(R))$

A	B
1	2
3	2

A	B
2	3
2	1

$\Rightarrow$

A	B	A	B
1	2	2	3
3	2	2	3

2. (b)

$S = \pi_{Sid}(\sigma_{credit < 5}(\text{course}) \bowtie \text{Enrolled})$

Computes those students who are enrolled in atleast one course with credit less than 5

$Q = \pi_{Sid}(\text{Student}) - S$

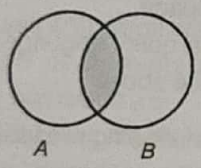
Computes students who all are enrolled in every course with credit greater than or equal to 5.

3. (b)

Intersection is derived operator. It is derived using set difference (–) operator.

$A \cap B = A - (A - B)$

$= B - (B - A)$



4. (c)

Number of attributes in R/S is $m - n$.

5. (c)

It is given that no foreign key. So it may be possible that no Sid matches between two tables, in that case natural join gives zero tuples. In best case all Sid matches then natural join gives 50 tuples.

6. (d)

A is uniformly distributed among [1, 500]. So number of tuples in [1, 100] will be equal to number of tuples in [101, 200], [201, 300], [301, 400], [401, 500] = 200

Number of tuples in [1, 200] = 400

7. (a)

$P \bowtie Q \Rightarrow$ Common attribute = C which is key for both P and Q.

$R_1 = P \bowtie Q \Rightarrow \min(200, 300) = 200$ tuples

$R_1 \bowtie R \Rightarrow$ Common attribute = E which is key for both R_1 and R.

$R_1 \bowtie R \Rightarrow \min(200, 100) = 100$ tuples

So, $|P \bowtie Q \bowtie R| = 100$ tuples

8. (a)

(*ii*) is false example $P(A, B) = \{(0, 1)\}$, $Q(A, B) = \{0, 2\}$

$\pi_A(P \cap Q) = \{\phi\}$ but $\pi_A(P) \cap \pi_A(Q) = \{0\}$

9. (b)

$$S = \pi_{R_1.A}\left(R_1 \underset{R_1.A < R_2.A}{\bowtie} R_2\right)$$

Find all values of A except the lowest.

$\pi_A(R) - S$ finds the lowest value of A.

10. (b)

(a) Find these drinker who are frequents atleast one bar on same city where he lives.

11. (b)

$|\pi_{Cid}(\sigma_{dept = 'EE'}(\text{Course}))| = 0$.

$\pi_{Sid,Cid}(\text{Enrolled}) / \pi_{Cid}(\text{empty tuples in Dept})$

All the distinct Sid will be in the result.

12. (c)

Only option (c) gives correct answer

13. (a)

To perform s/r the attribute set of r must be proper subset of attribute set s. i.e. $S \supseteq R$. s/r table contains attribute set as $S - R$.

14. (c)

15. (c)

"C" common attribute which is key for both R and S.

So one record of R maps with one record of S and vice versa.

So max matches $\min(n, m)$

Student's Assignments 2

Q.1 Consider the following COMPANY relational database schema shown

Employee (ssn, name, sex, salary)
Department (Dname, Dnumber)
Project (Pname, Pno)
Works_on (ssn, Pno)

Answer the following relational algebra queries on the above relational database schema

(a) Retrieve the names of all employees who work on every project.

(b) Retrieve the names of all employees who do not work on any project.

(c) Retrieve the average salary of all female employees.

(d) List the last names of all department manages who have no dependents.

Q.2 Consider the two tables T_1 and T_2 shown in figure. Show the results of the following operations:

(a) $T_1 \bowtie_{T_1.P=T_2.A} T_2$ (b) $T_1 \bowtie_{T_1.Q=T_2.B} T_2$

(c) $T_1 \bowtie_{T_1.P=T_2.A} T_2$ (d) $T_1 \bowtie_{T_1.Q=T_2.B} T_2$

(e) $T_1 \cup T_2$

(f) $T_1 \bowtie_{T_1.P=T_2.A AND T_1.r=T_2.c} T_2$

TABLE T_1

P	Q	R
10	a	5
15	b	8
25	a	6

TABLE T_2

A	B	C
10	b	6
15	c	3
10	b	5

Q.3 Find the results of these expression for the relational schema R and S.

R			
A	B	C	D
1	2	3	4
2	2	5	1
3	4	2	6
4	2	5	3

S		
C	D	E
1	2	4
3	4	1
5	1	6
4	2	3

(a) $R \bowtie S$ (b) $R \bowtie_{R.C=S.C} S$

(c) $R \bowtie_{R.A=S.C} S$ (d) $R \bowtie_{R.A=S.E} S$

Q.4 From the tables "R" and "S", find the following

(a) $R \cup S$ (b) $R \cap S$

(c) $R - S$

R		S	
Sno.	DEPT	Sno.	DEPT
S_1	Phy_1	S_{10}	$Maths_1$
S_2	Psy	S_3	Chem
S_3	Chem	S_{15}	Eng
S_4	Jour	S_{16}	$Math_2$

Q.5 Find the quotient for the following A/B_1, A/B_2, and A/B_3; where A, B_1, B_2, and B_3 are

Sno	Pno
S_1	P_1
S_1	P_2
S_1	P_3
S_1	P_4
S_2	P_1
S_2	P_2
S_3	P_2
S_4	P_2
S_4	P_4

B_1=

Pno
P_1

B_2=

Pno
P_2
P_4

B_3=

Pno
P_2
P_2
P_4

CHAPTER 04

SQL

4.1 Introduction

- Pronounced as SQL or sequel
- Supported by all major commercial database system
- Standardized
- Interactive via GUI on promt or embedded in program
- Declarative language based on relation algebra *i.e.* you write exactly what you want and the queries do't need to describe how to get the data out of the database
- Declarative nature of SQL leads to the component of database system called query optimizer to be extremly important. Query optimizer takes the query written in SQL language and it figures out the fastest way to execute that on the database

There are two parts of language

- **The Data Manipulation Language (DML):** This subset of SQL allows users to pose queries and to insert, delete, and modify rows.
- **The Data Definition Language (DDL):** This subset of SQL supports the creation, deletion, and modification of definitions for tables and views.

About the examples

We will present a number of sample queries using the following table definitions:
Sailors (sid, sname, rating, age)
Boats (bid, bname, color,)
Reserves (sid, bid, day)

4.2 The Form of a Basic SQL Query

The basic form of an SQL query is as follows:

```
SELECT   [DISTINCT] select-list
FROM     from-list
WHERE    qualification
```

Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause.

Such a query intuitively corresponds to a relational algebra expression involving selections, projections, and cross-products. The close relationship between SQL and relational algebra is the basis for query optimization in a relational DBMS.

A **multiset** is similar to a set in that it is an unordered collection of elements, but there could be several copies of each element, and the number of copies is significant-two multisets could have the same elements and yet be different because the number of copies is different for some elements. For example, {*a*, *b*, *b*} and {*b*, *a*, *b*} denote the same multiset, and differ from the multiset {*a*, *a*, *b*}.

Incidentally, when we want to retrieve all columns, SQL provides a convenient shorthand: We can simply write SELECT *. This notation is useful for interactive querying, but it is poor style for queries that are intended to be reused and maintained because the schema of the result is not clear from the query itself; we have to refer to the schema of the underlying table.

The SELECT clause is actually used to do projection, whereas *selections* in the relational algebra sense are expressed using the WHERE clause This mismatch between the naming of the selection and projection operators in relational algebra and the syntax of SQL is an unfortunate historical accident.

We now consider the syntax of a basic SQL query in more detail.

- The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The select-list is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the WHERE clause is a boolean combination (i.e., an expression using the logical connectives *A*ND, OR, and NOT) of conditions of the form *expression* op *expression*, where op is one of the comparison operators {<, <=, =, <>, >=, >}. An *expression* is a *column* name, a *constant*, or an (arithmetic or string) expression.
- The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain *duplicates*, that is, two copies of the same row. The default is that duplicates are not eliminated.

Although the preceding rules describe (informally) the syntax of a basic SQL query, they do not tell us the *meaning* of a query. The answer to a query is itself a relation which is a *multiset* of rows in SQL!—Whose contents can be understood by considering the following conceptual evaluation strategy:

1. Compute the cross-product of the tables in the from-list.
2. Delete rows in the cross-product that fail the qualification conditions.
3. Delete all columns that do not appear in the select-list.
4. If DISTINCT is specified, eliminate duplicate rows.

This straightforward conceptual evaluation strategy makes explicit the rows that must be present in the answer to the query. However, it is likely to be quite inefficient.

4.3 Union, Intersect, and Except

SQL provides three set-manipulation constructs that extend the basic query form presented earlier. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT.

SQL also provides other set operations: IN (to check if an element is in a given set), op *ANY*, op ALL (to compare a value with the elements in a given set, using comparison operator op), and EXISTS (to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning. We cover UNION, INTERSECT, and EXCEPT in this section.

A final point to note about UNION, INTERSECT, and EXCEPT follows. In contrast to the default that duplicates are not eliminated unless DISTINCT is specified in the basic query form, the default for UNION queries is that duplicates *are* eliminated! To retain duplicates, UNION ALL must be used; if so, the number of copies of a row in the result is always m + *n*, where *m* and *n* are the numbers of times that the row appears in the two parts of the union. Similarly, INTERSECT ALL retains duplicates—the number of copies of a row in the result is *min* (*m*, *n*) – EXCEPT ALL also retains duplicates – the number of copies of a row in the result is *m* – *n*, where *m* corresponds to the first relation.

4.4 Nested Queries

One of the most powerful features of SQL is nested queries. A nested query is a query that has another query embedded within it; the embedded query is called a subquery. The embedded query can of course be a nested query itself; thus queries that have very deeply nested structures are possible. When writing a query, we sometimes need to express a condition that refers to a table that must itself be computed. The query used to compute this subsidiary table is a subquery and appears as part of the main query. A subquery typically appears within the WHERE clause of a query. Subqueries can sometimes appear in the FROM clause or the HAVING clause.

Relational Algebra and SQL: Nesting of queries is a feature that is not available in relational algebra, but nested queries can be translated into algebra. Nesting in SQL is inspired more by relational calculus than algebra. In conjunction with some of SQL's other features, such as (multi) set operators and aggregation, nesting is a very expressive construct.

This section discusses only subqueries that appear in the WHERE clause. The treatment of subqueries appearing elsewhere is quite similar.

Qurey-1: Find the names of sailors who have reserved boat 103.

```
SELECT S. sname
FROM Sailors S
WHERE S.sid IN (SELECT  R.sid
                FROM   Reserves R
                WHERE R.bid = 103 )
```

The nested subquery computes the (multi)set of *sids* for sailors who have reserved boat 103 and the top-level query retrieves the names of sailors whose *sid* is in this set. The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested. Note that it is very easy to modify this query to find all sailors who have *not* reserved boat 103-we can just replace IN by NOT IN!

The best way to understand a nested query is to think of it in terms of a conceptual evaluation strategy. In our example, the strategy consists of examining rows in Sailors and, for each such row, evaluating the subquery over Reserves. In general, the conceptual evaluation strategy that we presented for defining the semantics of a query can be extended to cover nested queries as follows: Construct the cross-product of the tables in the FROM clause of the top-level query as before. For each row in the cross-product, while testing the qualification in the WHERE clause, (re)compute the subquery. Of course, the subquery might itself contain another nested subquery, in which case we apply the same idea one more time, leading to an evaluation strategy with several levels of nested loops.

As an example of a multiple nested query, let us rewrite the following query.

Qurey-2: Find the names of sailors who have reserved a red boat.

```
SELECT S. sname
FROM Sailors S
WHERE S.sid IN ( SELECT R.sid
                 FROM  Reserves R
                 WHERE  R.bid IN   ( SELECT B.bid
                                     FROM Boats B
                                     WHERE B.color = 'red' )
```

The innermost subquery finds the set of *bids* of red boats. The subquery one level above finds the set of *sids* of sailors who have reserved one of these boats. The top-level query finds the names of sailors whose *sid* is in this set of *sids*. To find the names of sailors who have not reserved a red boat, we replace the outermost occurrence of IN by NOT IN, as illustrated in the next query.

Qurey-3: Find the names of sailors who have not reserved a red boat.

```
SELECT S. sname
FROM Sailors S
WHERE S.sid NOT IN ( SELECT R.sid
                     FROM Reserves R
                     WHERE R.bid IN   ( SELECT B.bid
                                        FROM   Boats B
                                        WHERE B.color = 'red' )
```

This query computes the names of sailors whose *sid* is *not* in the set.

In contrast to Query Q3, we can modify the previous query (the nested version of Q2) by replacing the inner occurrence (rather than the outer occurrence) of IN with NOT IN. This modified query would compute the name of sailors who have reserved a boat that is not red, that is, if they have a reservation, it is not for a red boat.

4.5 Correlated Nested Queries

In the nested queries seen thus far, the inner subquery has been completely independent of the outer query. In general, the inner subquery could depend on the row currently being examined in the outer query (in terms of our conceptual evaluation strategy). Let us rewrite the following query once more.

Qurey-4: Find the names of sailors who have reserved boat number 103.

```
SELECT S. sname
FROM Sailors S
WHERE EXISTS ( SELECT      *
               FROM Reserves R
               WHERE R.bid = 103
                     AND   R.sid = S.sid )
```

The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty, an implicit comparison with the empty set. Thus, for each Sailor row *S*, we test whether the set of Reserves rows *R* such that *R.bid = 103* AND *S.sid = R.sid* is nonempty. If so, sailor S has reserved boat 103, and we retrieve the name. The subquery clearly depends on the current row S and must be re-evaluated for each row in Sailors. The occurrence of *S* in the subquery (in the form of the literal *S.sid)* is called a *correlation queries*.

This query also illustrates the use of the special symbol * in situations where all we want to do is to check that a qualifying row exists, and do not really want to retrieve any columns from the row. This is one of the two uses of * in the SELECT clause that is good programming style; the other is as an argument of the COUNT aggregate operation, which we describe shortly.

As a further example, by using NOT EXISTS instead of EXISTS, we can compute the names of sailors who have not reserved a red boat. Closely related to EXISTS is the UNIQUE predicate. When we apply UNIQUE to a subquery, the resulting condition returns true if no row appears twice in the answer to the subquery, that is, there are no duplicates; in particular, it returns true if the answer is empty. (And there is also a NOT UNIQUE version).

4.6 Set-Comparison Operators

We have already seen the set-comparison operators EXISTS, IN, and UNIQUE, along with their negated versions. SQL also supports op *ANY* and op ALL, where op is one of the arithmetic comparison operators {<, <=, =, <>, >=, >}. (**SOME** is also available, but it is just a synonym for ANY.)

Qurey-5: Find the sailors whose rating is better than some sailor called XYZ.

```
SELECT  S. sid
FROM    Sailors S
WHERE   S.rating > ANY ( SELECT   S2. rating
                         FROM     Sailors S2
                         WHERE    S2.sname = 'XYZ' )
```

If there are several sailors called XYZ, this query finds all sailors whose rating is better than that of *some* sailor called *XYZ*. What if there were *no* sailor called *XYZ*? In this case the comparison *S.rating* > **ANY** ... is defined to return **False**, and the query returns an empty answer set. To understand comparisons involving **ANY**, it is useful to think of the comparison being carried out repeatedly. In this example, *S. rating* is successively compared with each rating value that is an answer to the nested query. Intuitively, the subquery must return a row that makes the comparison **True**, in order for *S. rating* > **ANY** ... to return **True**.

Query-6: Find sailors whose rating is better than every sailor' called Horatio *XYZ*.

We can obtain all such queries with a simple modification to Query Q5: Just replace ANY with **ALL** in the **WHERE** clause of the outer query. If there were no sailor called *XYZ*, the comparison *S.rating* > **ALL** ... is defined to return true! The query would then return the names of all sailors. Again, it is useful to think of the comparison being carried out repeatedly. Intuitively, the comparison must be true for every returned row for *S. rating* > **ALL** ... to return **True**.

As another illustration of ALL, consider the following query.

Query-7: Find the Sailor's with the highest rating.

```
SELECT  S. sid
FROM    Sailors S
WHERE   S.rating >= ALL ( SELECT  S2. rating
                          FROM    Sailors S2 )
```

The subquery computes the set of all rating values in Sailors. The outer WHERE condition is satisfied only when *S.rating* is greater than or equal to each of these rating values, that is, when it is the largest rating value. Note that **IN** and **NOT IN** are equivalent to = **ANY** and <> **ALL**, respectively. Let us revisit a query that we considered earlier using the **INTERSECT** operator.

Query-8: Find the names of sailors who have reserved both a red and a green boat.

```
SELECT  S. sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
        AND S.sid IN ( SELECT   S2.sid
                       FROM     Sailors S2, Boats B2, Reserves R2
```

```
                WHERE   S2.sid = R2.sid AND R2.bid = B2.bid
                        AND B2.color = 'green' )
```

This query can be understood as follows: "Find all sailors who have reserved a red boat and, further, have *sids* that are included in the set of *sids* of sailors who have reserved a green boat." This formulation of the query illustrates how queries involving **INTERSECT** can be rewritten using IN, which is useful to know if your system does not support **INTERSECT**. Queries using **EXCEPT** can be similarly rewritten by using **NOT IN**. To find the sids of sailors who have reserved red boats but not green boats, we can simply replace the keyword IN the previous query by **NOT IN**.

As it turns out, writing this query (Q8) using **INTERSECT** is more complicated because we have to use *sids* to identify sailors (while intersecting) and have to return sailor names:

```
SELECT  S. sname
FROM    Sailors S
WHERE   S.sid IN    (( SELECT    R. sid
                       FROM      Boats B, Reserves R
                       WHERE     R. bid = B.bid AND B.color = 'red' )
                       INTERSECT
                    ( SELECT     R2. sid
                       FROM      Boats B2, Reserves R2
                       WHERE     R2.bid = B2.bid AND B2.color = 'green' ))
```

Our next example illustrates how the *division* operation in relational algebra can be expressed in SQL.

Query-9: Find the names of sailors who have reserved all boats.

```
SELECT  S. sname
FROM    Sailors S
WHERE   NOT EXISTS  (( SELECT    B. bid
                       FROM      Boats B
                       EXCEPT
                       (SELECT   R.bid
                       FROM      Reserves R
                       WHERE     R.sid = S.bid ))
```

Note that this query is correlated—for each sailor *S*, we check to see that the set of boats reserved by *S* includes every boat. An alternative way to do this query without using EXCEPT follows:

```
SELECT  S. sname
FROM    Sailors S
WHERE   NOT EXISTS  ( SELECT   B.bid
                      FROM     Boats B
                      WHERE    NOT EXISTS  ( SELECT   R.bid
                                             FROM     Reserved R
                                             WHERE    R.bid = B. bid
                                                      AND R.sid = S.sid ))
```

Intuitively, for each sailor we check that there is no boat that has not been reserved by this sailor.

4.7 Aggregate Operators

In addition to simply retrieving data, we often want to perform some computation or summarization. As we noted earlier in this chapter, SQL allows the use of arithmetic expressions. We now consider a powerful class of constructs for computing *aggregate values* such as *MIN* and *SUM*. These features represent a significant extension of relational algebra.

SQL supports five aggregate operations, which can be applied on any column, say *A*, of a relation:

(*i*) COUNT ([DISTINCT] *A*): The number of (unique) values in the *A* column.

(*ii*) SUM ([DISTINCT] *A*): The sum of all (unique) values in the *A* column.

(*iii*) AVG ([DISTINCT] *A*): The average of all (unique) values in the *A* column.

(*iv*) MAX (*A*): The maximum value in the *A* column.

(*v*) MIN (*A*): The minimum value in the *A* column.

Note that it does not make sense to specify DISTINCT in conjunction with MIN or MAX (although SQL does not preclude this).

Query-10: Find the average age of all sailors.

```
SELECT    AVG (S. age)
FROM      Sailors S
```

The WHERE clause can be used to restrict the sailors considered in computing the average age.

Query-11: Find the average age of sailors with a rating of 10.

```
SELECT    AVG (S.age)
FROM      Sailors S
WHERE     S.rating = 10}
```

However, finding both the name and the age of the oldest sailor is more tricky, as the next query illustrates.

Query-12: Find the name and age of the oldest sailor.

Consider the following attempt to answer this query:

```
SELECT    S.sname,  MAX  (S.age)
FROM      Sailors  S
```

The intent is for this query to return not only the maximum age but also the name of the sailors having that age. However, this query is illegal in SQL–if the SELECT clause uses an aggregate operation, then it must use only aggregate operations unless the query contains a GROUP BY clause! (The intuition behind this restriction should become clear when we discuss the GROUP BY clause in next section). Therefore, we cannot use MAX (S.age) as well as S.sname in the SELECT clause. We have to use a nested query to compute the desired answer to Q.12.

```
SELECT    S.sname, S.age
FROM      Sailors S
WHERE     S.age = (SELECT MAX (S2.age)
          FROM Sailors S2)
```

Observe that we have used the result of an aggregate operation in the subquery as an argument to a comparison operation. Stricly speaking, we are comparing an age value with the result of the subquery, which is relation. However, because of the use of the aggregate operation, the subquery is guaranteed to return a single tuple with a single field, and SQL converts such a relation to a field value for the sake of the comparison. The following equivalent query for Q.12 is legal in the SQL standard but, unfortunately, is not supported in many systems.

```
SELECT    S.sname, S.age
FROM      Sailors S
WHERE     (SELECT MAX (S2.age)
          FROM Sailors S2) = S.age
```

We can count the number of sailors using COUNT. This example illustrates the use of * as an argument to COUNT, which is useful when we want to count all rows.

Query-13: Count the number of sailors.

```
SELECT    COUNT (*)
FROM      Sailors S
```

We can think of * as shorthand for all the columns (in the cross-product of the **from-list** in the FROM clause). Contrast this query with the following query, which computes the number of distinct sailor names. (Remeber that sname is not a key!)

Query-14: Count the number of different sailor names.

```
SELECT    COUNT (DISTINCT  S.sname)
FROM      Sailors S
```

If COUNT does not include DISTINCT, then COUNT (*) gives the same answer as COUNT (x), where x is any set of attributes. In our example, without DISTINCT Q.15 is equivalent to Q.14. However, the use of COUNT (*) is better querying style, since it is immediately clear that all records contribute to the total count.

Aggregate operations offer an alternative to the ANY and ALL constructs. For example, consider the following:

Query-15: Find the names of sailors who are older than the oldest sailor with a raing of 10.

```
SELECT    S.sname
FROM      Sailors S
WHERE     S.age > (SELECT  MAX  (S2.age)
          FROM  Sailors S2
          WHERE S2.rating = 10)
```

Using ALL, this query could alternatively by written as follows:

```
SELECT    S.sname
FROM      Sailors S
WHERE     S.age >  ALL  (SELECT  S2.age
                  FROM  Sailors S2
                  WHERE S2.rating = 10)
```

However, the ALL query is more error prone—one could easily (and incorrectly!) use ANY instead of ALL, and retrieve sailors who are older than *some* sailors with a rating of 10. The use of ANY intuitively corresponds to the use of MIN, instead of MAX, in the previous query.

4.8 The Group by and Having Clauses

Thus far, we have applied aggregate operations to all (qualifying) rows in a relation. Often we want to apply aggregate operations to each of a number of groups of rows in a relation, where the number of groups depends on the relation instance (*i.e.*, is not known in advance). To write such queries, we need a major extension to the basic SQL query form, namely, the *GROUP BY* clause. In fact, the extension also includes an optional *HAVING* clause that can be used to specify qualification over groups.

Query-16: Find the age of the youngest sailor for each rating level.

If we know that ratings are integers in the range 1 to 10, we could write 10 queries of the form:

```
SELECT    MIN  (S.age)
FROM      Sailors  S
WHERE     S.rating = i
```

where $i = 1, 2, \ldots, 10$. Writing 10 such queries is tedious. More important, we may not know what rating levels exist in advance.

```
SELECT      [ DISTINCT] select-list
FROM        from-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

Using the GROUP By clause, we can write Q17 as follows:

```
SELECT      S.rating, MIN (S.age)
FROM        Sailors  S
GROUP BY    S.rating
```

Let us consider some important points concerning the new clauses:

The **select-list** in the SELECT clause consists of (1) a list of column names and (2) a list of terms having the form **aggop** (*column-name) AS newname*. Columns that are the result of aggregate operators do not already have a column name, and therefore giving the column a name with *AS* is especially useful.

Every column that appears in (1) must also appear in **grouping-list.** The reason is that each row in the result of the query corresponds to one *group*, which is a collection of rows that agree on the values of columns in groupinglist. In general, if a column appears in list (1), but not in **grouping-list**, there can be multiple rows within a group that have different values in this column, and it is not clear what value should be assigned to this column in an answer row.

We can sometimes use primary key information to verify that a column has a unique value in all rows within each group. For example, if the **grouping-list** contains the primary key of a table in the **from-list**, every column of that table has a unique value within each group.

The expressions appearing in the **group-qualification** in the HAVING clause must have a single value per group. The intuition is that the HAVING clause determines whether an answer row is to be generated for a given group. To satisfy this requirement in SQL-92, a column appearing in the **group-qualification** must appear as the argument to an aggregation operator, or it must also appear in grouping-list. In SQL:1999, two new set functions have been introduced that allow us to check whether *every* or *any* row in a group satisfies a condition; this allows us to use conditions similar to those in a WHERE clause.

If *GROUP BY* is omitted, the entire table is regarded as a single group.

Query-17: Find the average age of sailor for each rating level that has at least two sailors.

```
SELECT      S.rating, AVG (S.age) AS average
FROM        Sailors S
GROUP BY    S.rating
HAVING      COUNT (*) > 1
```

The following alternative formulation of Query Q34 illustrates clause can have a nested subquery, just like the WHERE can use *S. rating* inside the nested subquery in the *HAVING* clause because it has a single value for the current group of sailors:

```
SELECT      S. rating, AVG (S.age) AS average
FROM        Sailors S
GROUP BY    S.rating
HAVING      1 < (SELECT COUNT (*)
                 FROM    Sailors S2
                 WHERE   S.rating = S2. rating)
```

4.9 NULL Values

Thus far, we have assumed that column values in a row are always known. In practice column values can be unknown. SQL provides a special column value called *null* to use in such situations. We use *null* when the column value is either *unknown* or *inapplicable*. The presence of *null* values complicates many issues, and we consider the impact of *null* values on SQL in this section.

Consider a comparison such as *rating* = 8. If this is applied to the row for *XYZ*, is this condition true or false? Since *XYZ*'s rating is unknown, it is reasonable to say that this comparison should evaluate to the value unknown. In fact, this is the case for the comparisons *rating*> 8 and *rating* < 8 as well. Perhaps less obviously, if we compare two *null* values using <, >, =, and so on, the result is always unknown. For example, if we have *null* in two distinct rows of the sailor relation, any comparison returns unknown.

SQL also provides a special comparison operator IS NULL to test whether a column value is *null;* for example, we can say *rating* IS NULL, which would evaluate to true on the row representing *XYZ*. We can also say *rating* IS NOT NULL, which would evaluate to false on the row for *XYZ*.

Boolean expressions arise in many contexts in SQL, and the impact of *null* values must be recognized. for example, the qualification in the WHERE clause eliminates rows (in the cross-product of tables named in the FROM clause) for which the qualification does not evaluate to true. Therefore, in the presence of *null* values, any row that evaluates to false or unknown is eliminated. Eliminating rows that evaluate to unknown has a subtle but signifieant impact on queries, especially nested queries involving EXISTS or UNIQUE.

Another issue in the presence of *null* values is the definition of when two rows in a relation instance are regarded as *duplicates.* The SQL definition is that two rows are duplicates if corresponding columns are either equal, or both contain *null.* Contrast. This definition with the fact that if we compare two *null* values using =, the result is unknown! In the context of duplicates, this comparison is implicitly treated as true, which is an anomaly.

As expected, the arithmetic operations +, –, *, and / all return *null* if one of their arguments is *null.* However, nulls can cause some unexpected behavior with aggregate operations. COUNT(*) handles *null* values just like other values; that is, they get counted. All the other aggregate operations (COUNT, SUM, AVG, MIN, MAX, and variations using DISTINCT) simply discard *null* values thus *SUM* cannot be understood as just the addition of all values in the (multi) set of values that it is applied to; a preliminary step of discarding all *null* values must also be accounted for. As a special case, if one of these operators-other than *COUNT*-is applied to *only* null values, the result is again *null.*

Logical Connectives in Three-Valued Logic

(a)	**AND**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	**OR**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	**NOT**			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

Summary

- A retrieval query in SQL can consist of up to six clauses, but only the first two- SELECT and FROM - are mandatory. The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

```
SELECT <attribute list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ]
```

- The SELECT clause lists the attribute to be retrieved. The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE clause specifies the conditions for selection of tuples from these relations, including join conditions if needed. *GROUP BY* specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can, also be applied to all the selected tuples in query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.
- A query is evaluated conceptually by first applying the FROM clause (to identify all tables involved in the query or to materialize any joined tables), followed by the WHERE clause, and then by GROUP BY and HAVING. Conceptually, ORDER BY is applied at the end to sort the query result. If none of the last three clauses (GROUP BY, HAVING and ORDER BY) are specified, we can think conceptually of a query as being executed as follows: For each combination of tuples - one from each fo the relations specified in the FROM clause - evaluate the WHERE clause; if it evaluates tuple combination in the result of the query. Of course, this is not an efficient way to implement the query in a real system, and each *DBMS* has special query optimization routines to decide on anexecution plan that is efficient.
- In general, there are numerous ways to specify the same query in SQL. This flexibility in specifying queries has advantages and disadvantages. The main advantagee is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the WHERE clause, or by using joined relations in the FROM clause, or with some form of nested queries and the *IN* comparison operator. Some users may be more comfortable with one approach, whereas other may be more comfortable with another.
- The disadvantage of having numerous ways of specifying the same query is that this may not know which technique to use to specify particular types of queries. Another problem is that it may be more efficcient to execute a query specified in one way than the same query specified in an alternative way. Ideally, this should not be the case: The DBMS should process the same query in the same way regardless of how the query is specified. But this is quite difficult in practice, since each DBMS has different methods for processing queries specified in different ways. Thus, an additional burden on the user is to determine which of the aternative specifications is the most efficient. Ideally, the user should worry only about specifying the query correctly. It is the respondibility of the DBMS to execute the query efficiently.

Student's Assignment

Q.1 Consider the relations:
S (sid, sname, city)
P (pid, pname, color)
SP (sid, pid)
Consider the following SQL query
Select distinct sname
From S
Where EXISTS
(Select * From SP where SP.sid = s.sid AND SP.pid = 'P_1')
The above query represents which of the following?
(a) Get supplier names from suppliers who supply some part P_1
(b) Get supplier names who supply part P_1 only
(c) Get supplier name who does not supply part P_1
(d) None of these

Q.2 Consider the following relational schema:
Query 1:
```
Select A
FROM R
Where EXIST S(Select *
              FROM S
              Where R.A > S.C)
```
Query 2: $\pi_A(R \underset{R.A<S.C}{\bowtie} S)$
Query 3:
```
Select A
FROM R
Where A > All (Select C
               FROM S)
```
Which of the following statement is true?
(a) Query 1 and Query 2 results same but not Query 3
(b) Query 1 and Query 3 results same but not Query 2
(c) Query 2 and Query 3 results same but not Query 1
(d) All Query 1, Query 2, Query 3 results same

Q.3 Consider the following database schema:
Course(course_no, dep_name)
Enroll(stud_id, course_no, status)
```
SELECT stud_id
FROM Course C, Enroll E
WHERE C.course_no = E.course_no and
dep_name = 'CS'
EXCEPT
SELECT stud_id
FROM Course C, Enroll E
WHERE E.course_no = C.course_no AND
dep.name = 'ME'
```
Above SQL query
(a) Finds the students who are enrolled in all courses by CS department are not enrolled in all courses offered by ME department
(b) Finds the students who are enrolled in any course by CS department and not enrolled in all courses offered by ME department
(c) Finds the students who are enrolled in atleast one course by CS department and not enrolled in any courses offered by ME department
(d) Finds the students who are enrolled in all course by CS department and not enrolled in any courses offered by ME department

Q.4 Consider the relation
Project (pno, pname, budget, city)
```
Q1: SELECT pname
    FROM proj
WHERE NOT (budget < = ANY (SELECT budget
FROM proj
WHERE city = 'KANPUR'))
```
Q1 computes:
(a) Name of the projects whose budgets is less than atleast one project in KANPUR
(b) Name of the projects whose budgets is greater than some project in KANPUR
(c) Name of the projects whose budgets is less than all project in KANPUR
(d) Name of the projects whose budgets is greater than all project in KANPUR

Q.5 Consider the relations:

Proj (pid, pname, budget, city)

Q: SELECT pname
FROM proj P_1
WHERE NOT EXISTS
(SELECT budget
FROM proj P_2
WHERE city = 'DELHI'
AND P_1.budget <= P_2.budget)

Q finds project name whose budget is

(a) greater than some project in DELHI
(b) greater than all project in DELHI
(c) less than all project in DELHI
(d) less than any project in DELHI

Q.6 Consider the following relations:

Emp (eno, ename, title, city)
Project (pno, pname, budget)
Works (eno, pno)
Pay (title, salary)

Which query finds what fraction of the budget is spend on salaries for the people working on that project?

(a) SELECT *P*.pno, pname, sal/budget AS frac
From project *P*, (SELECT pno, sum (salary))
AS sal
FROM works, Emp, Pay
WHERE works.eno = Emp.eno
AND Emp.title = Pay.title
GROUP by pno) *AS Q*
WHERE *P*.pno = Q.pno
ORDER by budget

(b) SELECT *P*.pno, pname, sum(salary)/budget
AS frac
From project *P*, works *W*, Emp *E*, Pay
WHERE *P*.pno = *W*.pno
AND *W*.eno = *E*.eno
AND *E*.title = Pay.title
GROUP BY *P*.pno, budget
ORDER BY budget

(c) Both (a) and (b)
(d) None of these

Q.7 Consider the following relations:

Bank(bname, city)
Travel(pname, city)
SELECT T_1.pname
FROM Travel T_1
WHERE *NOT EXISTS* (SELECT *B*.city
From Bank *B*
WHERE *B*.bname = '*SBI*'
EXCEPT
SELECT T_2.city
FROM Travel T_2
WHERE T_1.pname = T_2.pname)

This query finds name of the persons.

(a) Who have not travelled in any city where SBI is located
(b) Who have not travelled in all city where SBI is located
(c) Who have travelled in all city where SBI is located
(d) Who have travelled in any city where SBI is located

Q.8

SELECT	1
FROM	2
WHERE	3
GROUP BY	4
HAVING	5
ORDER BY	6

What is correct order for evaluating an SQL statement? (Where order is 6 digit number)

Q.9 Which of the following queries will give the names of the employees who are earning maximum salary?

(a) Select name from emp
where sal = (select max(sal) from emp)
(b) Select name from emp
where sal >= (select sal from emp)
(c) Both (a) and (b) are correct queries but the processing time is too high in (b) than in (a)
(d) Both (a) and (b) are correct but processing is too high in (a) than in (b)

Q.10 Which of the following aggregate functions does not ignore nulls in its results?

(a) COUNT (b) COUNT(*)
(c) MAX (d) MIN

Q.11 Consider the following relational schema
employee (ename, salary, job)
Consider the following SQL queries and identify which of them are correct?
1. Select distinct ename, job from employee
2. Select ename, distinct job from employee
3. Select distinct ename from employee

Q.12 Consider the following relation schema:
Author (A_name, A_city)
Book (B_title, A_name, P_name, Price)
Publisher (P_name, P_city)
From the queries given below, which query is syntactically and logically incorrect for above schemas:
(a) SELECT B_title, Price from BOOK
Where P_name IN (Select P_name, P_city from Publisher
where P_City = "Delhi");
(b) SELECT P_name, A_name, B_title from BOOK
Where price BETWEEN 1000 AND (Select avg (price) from BOOK
where P_name = "TMH")
(c) SELECT A_name, A_city, Count (P_name)
FROM Author, BOOK
Where Author.A_name = BOOK.A_name
GROUP by A_name
having COUNT (P_name) > 5
(d) All of these

Q.13 Consider the following table orders

Order_Id	Order_price	Customer
1	500	Rajesh
2	300	Ramesh
3	100	Suresh
4	600	Rajesh
5	800	Rajesh
6	900	Suresh
7	400	Rakesh

Number of tuples if we execute the following query on the above table is _____.
Select Customer, Sum (order_price)
From Orders
Group by Customer
Having Sum (order_price) < = 1000

Q.14 Consider the following relational schema student (Sid, Sname)

Sid	Sname
1	A
2	B
3	Null
Null	Null

Now consider the following queries:
Q_1 : Select count (*) from student.
Q_2 : Select count (Sid) from student.
Query Q_1 and Q_2 returns respectively.
(a) 4 and 3 (b) 4 and 4
(c) 3 and 3 (d) Invalid table

Answers Key:

1. (a) **2.** (a) **3.** (c) **4.** (d) **5.** (b)
6. (c) **7.** (c) **8.** (234516) **9.** (a)
10. (b) **11.** (a, c) **12.** (d) **13.** (3) **14.** (a)

Student's Assignments | **Explanations**

1. (a)
Option (a) is true example

2. (a)
Q1: Retrieves A which are more than some C.
Q2: Retrieves A which are more than some C.
Q3: Retrieves A which are more than every C.

3. (c)
Query Finds students who are enrolled in atleast one course offered by CS department and not enrolled in any course offered by ME department.

4. (d)
Aabove query can be rewritten like this
SELECT pname
FROM proj
WHERE NOT NOT (budget > ALL (SELECT budget
FROM project
WHERE city = 'KANPUR'))
or

```
SELECT pname
FROM   proj
Where budget >  ALL (SELECT budget
FROM   proj
WHERE city = 'KANPUR')
```

which finds name of the projects whose budget is greater than all the projects in KANPUR.

5. (b)
Subquery finds budget of projects located in DELHI if that project has a larger budget than what is found in outer query not exist says subquery to return an empty result.
In that case projects found in the outer relation have budgets greater than all projects located in DELHI.

6. (c)
Both the query finds what fraction of budget is spend on salaries for the people working on that project.

7. (c)
Aabove query computes name of persons who have travelled in every city where SBI is located.

8. (234516)
- Take all tables listed in 2 compute their cross product
- Select rows of cross product that satisfy the condition in 3
- Group the selected rows by attribute in 4
- Select the groups that satisfy the condition in 5
- Project out the selected group attribute in 1
- Last the result by attribute in 6

9. (a)
Query (b) is not correct. Correct form of query (b) is

```
Select name from emp
where sal > = ALL (select sal from emp)
```

12. (d)
Query (A): When use 'IN' the inner query must retrieve only a single attribute.
Query (B): Between cannot be used with a sub-query.
Query (C): In select clause only grouping attributes and aggregate functions are allowed, when using group by.

13. (3)
The output table is

Customer	Sum (order_price)
Rakesh	400
Ramesh	300
Suresh	1000

CHAPTER 05

Transaction

5.1 Introduction

A transaction is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations – these can include insertion, deletion, modification, or retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program.

The basic database access operations that a transaction can include are as follows:

- read_item(X). Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
- write_item(X). Writes the value of program variable X into the database item named X.

The basic unit of data transfer from disk to main memory is one block. Executing a read_item(X) command includes the following steps:

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
- Copy item X from the buffer to the program variable named X.

Executive a write_item(X) command includes the following steps:

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
- Copy item X from the program variable named X into its correct location in the buffer.
- Store the updated block from the buffer back to disk (either immediately or at some later point in time).

5.2 ACID Properties

To preserve Integrity, transaction should satisfy ACID properties. ACID properties are some basic rules, which has to be satisfied by every transaction.

Atomicity

Transaction should execute all operations including commit or execute none of them. User should not have to worry about incomplete transaction (when a system crash occurs). Let A, B are two bank balances with values 100$ and 200$ respectively transaction T transfer 50$ from A to B

T: read_item (A)
$A = A$-50
Write_item (A)
read_item(B)
$B = B + 50$
write_item (B)
COMMIT

If transaction T fails any where before COMMIT, undo all the modification of database, rollback is done by **recovery management component**.

Consistency

Users are responsible for ensuring transaction consistency. That is, the user who submits a transaction must ensure that, when run to completion by itself against a 'consistent' database instance, the transaction will leave the database in a 'consistent' state. For example, the user may (naturally) have the consistency criterion that fund transfers between bank accounts should not change the total amount of money in the accounts. To transfer money from one account to another, a transaction must debit one account, temporarily leaving the database inconsistent in a global sense, even though the new account balance may satisfy any integrity constraints with respect to the range of acceptable account balances. The user's notion of a consistent database is preserved when the second account is credited with the transferred amount. If a faulty transfer program always credits the second account with one dollar less than the amount debited from the first account, the DBMS cannot be expected to detect inconsistencies due to such errors in the user program's logic.

Isolation

The isolation property is ensured by guaranteeing that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order.

For example, if two transaction T_1 and T_2 are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of) T_1 followed by executing T_2 or executing T_2 followed by executing T_1. (The DBMS provides no guarantees about which of these orders is effectively chosen.) If each transaction maps a consistent database instance to another consistent database instance, executing several transactions one after the other (on a consistent initial database instance) results in a consistent final database instance.

Durability

The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure. i.e. transaction should able to recover under any case of failure for example to recover from disk failure RAID architecture is maintained.

The *atomicity property* requires that we execute a transaction to completion. It is the responsibility of the *transaction recovery subsystem* of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

The preservation of *consistency* is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. Recall that a **database state** is a collection of all the stored data items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems. There have been attempts to define the **level of isolation** of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called *true isolation*) has, in addition to level 2 properties, repeatable reads. And last, the *durability property* is the responsibility of the *recovery subsystem* of the DBMS.

5.3 Types of Failures

Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

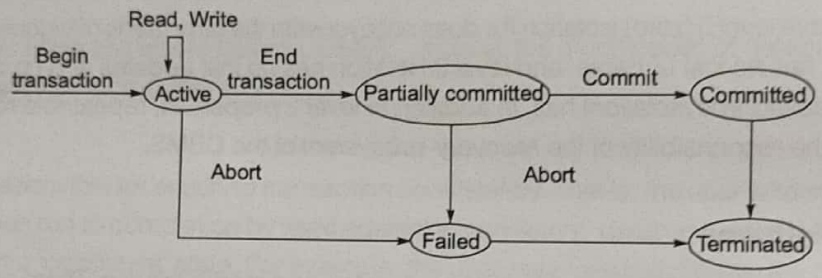
1. **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.
2. **A transaction or system error.** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Additionally, the user may interrupt the transaction during its execution.
3. **Local errors or exception conditions detected by the transaction.** During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. An exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.
4. **Concurrency control enforcement.** The concurrency control method may decide to abort a transaction because it violates serializability, or it may abort one or more transactions to resolve a state of deadlock among several transactions . Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.
5. **Disk failure.** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

5.4 Transaction States

A transaction goes into an active state immediately after it starts execution, where it can issue READ and WRITE operations. When the transaction ends, it moves to the partially committed state. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log). Once this check is successful, the transaction is said to have reached its commit point and enters the committed state.

Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.

However, a transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. The terminated state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transaction may be restarted later - either automatically or after being resubmitted by the user - as brand new transactions.



5.5 Schedule

A **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction *T* appear in a schedule must be the same as the order in which they appear in *T*. Intuitively, a schedule represents an actual or potential execution sequence. For example, the schedule in Figure shows an execution order for actions of two transaction T_1 and T_2. We move forward in time as we go down from one row to the next. We emphasize that a schedule describes the actions of transactions as seen by the DBMS. In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on; however, we assume that these actions do not affect other transactions; that is, the effect of a transaction on another transaction can be understood solely in terms of the common database objects that they read and write.

T_1	T_2
R (A)	
W (A)	R (B)
	W (B)
R (C)	
W (C)	

Note that the schedule in figure does not contain an abort or commit action for either transaction. A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a complete schedule. A complete schedule must contain all the actions of every transaction that appears in it. If the actions of different transactions are not interleaved - that is transactions are executed from start to finish, one by one - we call the schedule a **serial schedule**.

5.6 Concurrent Execution of Transaction

Now that we have introduced the concept of a schedule, we have a convenient way to describe interleaved executions of transactions. The DBMS interleaves the actions of different transactions to improve performance, but not all interleaving should be allowed. In this section, we consider what interleaving, or schedules, DBMS should allow.

Motivation for Concurrent Execution

The schedule shown in figure represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult but necessary for performance reasons. First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle and increases **system throughput** (the average number of transactions completed in a given time). Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short transaction could get stuck behind a long transaction, leading to unpredictable delays in **response time**, or average time taken to complete a transaction.

5.7 Problems because of Concurrent Execution

Two actions on the same data object conflict if at least one of them is a write. The three anomalous situations can be described in terms of when the actions of two transaction T_1 and T_2 conflict with each other: In a write-read (*WR*) conflict, T_2 reads a data object previously written by T_1; we define read-write (*RW*) and write-write(*WW*) conflicts similarly.

Reading Uncommitted Data (WR Conflicts)

The first source of anomalies is that a transaction T_2 could read a database object A that has been modified by another transaction T_1, which has not yet committed. Such a read is called a **dirty read**. A simple example illustrates how such a schedule could lead to an inconsistent database state.

Consider two transactions T_1 and T_2, each of which, run alone, preserves database consistency: T_1 transfers \$100 from A to B, and T_2 increments both A and B by 6% (e.g., annual interest is deposited into these two accounts). Suppose that the actions are interleaved so that (1) the account transfer program T_1 deducts \$100 from account A, then (2) the interest deposit program T_2 reads the current values of accounts A and B and adds 6% interest to each, and then (3) the account transfer program credits \$100 to account B. The corresponding schedule, which is the view the DBMS has of this series of events, is illustrated in figure. The result of this schedule is different from any result that we would get by running one of the two transactions first and then the other. The problem can be traced to the fact that the value of A written by T_1 is read by T_2 before T_1 has completed all its changes.

T_1	T_2
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

Reading uncommitted data.

The general problem illustrated here is that T_1 may write some value into A that makes the database inconsistent. As long as T_1 overwrites this value with a 'correct' value of A before committing, no harm is done if T_1 and T_2 run in some serial order, because T_2 would then not see the (temporary) inconsistency. On the other hand, interleaved execution can expose this inconsistency and lead to an inconsistent final database state.

Unrepeatable Read (RW Conflicts)

Transaction T_2 is updating A which is already read by uncommitted transaction T_1. If T_1 tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime. This situation could not arise in a serial execution of two transactions; it is called an unrepeatable read.

Example: Supports joint account A of 2 persons with initial value 500

T_1 : $r(A)$; $A = A\text{-}100$; $W(A)$; commit

T_2 : $r(A)$; $A = A\text{-}200$; $W(A)$; commit

T_1	T_2
$r(A)$	
$A = A\text{-}100$	
	$r(A)$
	$A = A\text{-}200$
	$W(A)$
	C_2
$W(A)$	
C_1	

This situation can never arise in a serial execution of T_1 and T_2.

Overwriting Uncommitted Data (WW Conflicts)

Transaction T_2 could overwrite the value of an object A, which has already been modified by a transaction T_1, while T_1 is still in progress. Even if T_2 does not read the value of A written by T_1, a potential problem exists as the following example illustrates.

Suppose that Harry and Larry are two employees, and their salaries must be kept equal. Transaction T_1 sets their salaries to \$2000 and transaction T_2 sets their salaries to \$1000. If we execute these in the serial order T_1 followed by T_2, both receive the salary \$1000; the serial order T_2 followed by T_1 gives each the salary \$2000. Either of these is acceptable from a consistency standpoint (although Harry and Larry may prefer a higher salary!). Note that neither transaction reads a salary value before writing it - such a write is called a blind write, for obvious reasons.

Now consider the following interleaving of actions of T_1 and T_2.

T_1	T_2
$W(A)$	
	$W(A)$
	$W(B)$
$W(A)$	

The result is not identical to the result of either of the two possible serial executions, and the interleaved schedule is therefore not serializable. It violates the desired consistency criterion that the two salaries must be equal.

Lost Update Problem

Lost update problem can occur in both serializable schedule and non serializable schedule. This problem occurs when two transactions that access the some database items have their operations interleaved in a way that makes the value of same database items incorrect. Suppose that transactions T_1 and T_2 are interleaved as below

T_1	T_2
$W(A)$	
⋮	
	$W(A)$
Failed	

Suppose T_1 fails value updated by T_2 is lost.

Lost update problem possible if simultaneous ww operations exist. It is not similar to *WW* problem because for ww problem the schedule should be non-serializable.

To avoid lost update problem, concurrency control management should restrict the simultaneous WW operation. i.e. if T_1 updates data item A then other transaction say T_2 is not allowed to write on A until commit/ rollback of T_1 then no lost update problem.

Concurrency control component allowed a schedule S only if

- S is serializable (to avoid *WW, RW, WR* problem)
- No WW operation exists (to avoid last update problem)

5.8 Serializability

A serializable schedule over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transaction in some serial order.

Two schedules S and S' are equal iff

1. Initial read operation of S and S' should be same for every data item.
2. Final updation of data item items in S and S' should be same.

Example-5.1 Consider the following schedule S

T_1	T_2
$r(A)$	
	$r(B)$
$r(C)$	
$r(B)$	
	$r(C)$
$W(C)$	
	$r(A)$

Find if S is serializable or not?

Solution:

S is serializable only if there is a serial schedule S' and S is equal to S'

Suppose $S' : T_1 \rightarrow T_2$

T_1	T_2
$r(A)$	
$r(C)$	
$r(B)$	
$W(C)$	
	$r(B)$
	$r(C)$
	$r(A)$

Schedule S and S' are not equal because initial read of data item B and C is not same in S and S'

Suppose $S' : T_2 \rightarrow T_1$

T_1	T_2
	$r(B)$
	$r(C)$
	$r(A)$
✓ $r(A)$	
✓ $r(C)$	
$W(b)$	
$W(C)$	

Schedule $S = S'$ because initial read on data items A, B and C in both S and S' are same and final writes on data items is same in S and S'.
So schedule S is serializable and the equivalent serial schedule is $T_2 \rightarrow T_1$.

5.9 Uses of Serializability

A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for another transaction to terminate, thus slowing down processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness. In practice, it is quite difficult to test for the serializability of a schedule. The interleaving of operations from concurrent transaction - which are usually executed as processes by the operating system - is typically determined by the operating system scheduler, which allocates resources to all processes. Factors such as system load, time of transaction submission, and priorities of processes contribute to the ordering of operations in a schedule. Hence, it is difficult to determine how the operations of a schedule will be interleaved beforehand to ensure serializability.

If transactions are executed at will and then the resulting schedule is tested for serializability, we must cancel the effect of the schedule if it turns out not to be serializable. This is a serious problem that makes this approach impractical. Hence, the approach taken in most practical systems is to determine methods that ensure serializability, without having to test the schedules themselves. The approach taken in most commercial *DBMS*s is to design **protocols** (sets of rules) that - if followed by every individual transaction or if enforced by a DBMS concurrency control subsystem-will ensure serializability of all **schedules in which the transactions participate**.

5.10 Classification of Schedules

- Based on serializability (To avoid *RW*, *WR*, *WW* problems)
  Conflict serializability
  View serializability
- Based on recoverability
  irrecoverable schedule
  recoverable schedule
  cascade roll back recoverable schedule
  strict recoverable schedule

Conflict Serializable Schedule

Schedule S is conflict serializable only if there exist atleast one serial schedule $S¢$ which is conflict equivalent to S.

Two schedule S and S' are conflict equal schedules only if S' is resulted after swapping or exchanging some consecutive non conflict pairs. Two operations in a schedule are said to conflict if they belong to different transactions, access the same database item, and atleast one of the two operation is write_item operation.

Testing for conflict serializability of a schedule.

- For each transaction T_i participating in schedule S, create a node labelled T_i in the precedence graph.
- For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
- For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
- For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
- The schedule S is serializable if and only if the precedence graph has no cycles.

If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable.

In the precedence graph, an edge from T_i to T_j means that transaction T_i must come before transaction T_j in any serial schedule that is equivalent to S, because two conflicting operations appear in the schedule in that order. If there is no cycle in the precedence graph, we can create an equivalent serial schedule S' that is equivalent to S, by ordering the transaction that participate in S as follows: Whenever an edge exists in the precedence graph from T_i to T_j, T_i must appear before T_j in the equivalent serial schedule S'.

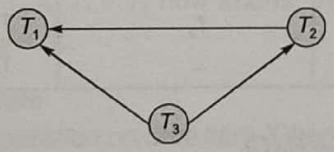
Example-5.2 Consider the schedule S given below:

$S: r_1(X); r_3(Y); r_3(X); r_2(Y); r_2(Z); w_3(Y); w_2(Z); r_1(Z); w_1(X); w_1(Z)$

Find schedule is conflict serializable or not.

Solution:

Precedence graph



S is conflict serializable with equivalent serial schedule is T_2, T_3, T_1.

View Serializable Schedule

A schedule S is said to be view serializable if it is view equivalent to some serial schedule. Two schedules S and S' are said to be view equivalent if the following three condition holds.

1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
2. Every read operation in S and S' should be same *i.e.* For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in S'.
3. Every final write of data item should be same in S and S'.

The idea behind view equivalence is that, as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to see the same view in both schedules. Conditions 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be same at the end of both schedules.

The definition of view serializability is less restrictive than that of conflict serializability under the unconstrained write assumption, where the value written by an operation $w_i(X)$ in T_i can be independent of its old value from the database. This is called a **blind write**, and it is illustrated by the following schedule S of three transaction.

T_1: $r_1(X)$; $w_1(X)$; T_2: $w_2(X)$; and T_3: $w_3(X)$:

S: $r_1(X)$; $w_2(X)$; $w_1(X)$; $w_3(X)$; c_1; c_2; c_3;

In S the operations $w_2(X)$ and $w_3(X)$ are blind writes, since T_2 and T_3 do not read the value of X. The schedule S is view serializable, since it is view equivalent to the serial schedule T_1, T_2, T_3. However, S is not conflict serializable, since it is not conflict equivalent to any serial schedule. It has been shown that any conflict-serializable schedule is also view serializable but not vice versa, as illustrated by the preceding example. There is an algorithm to test whether a schedule S is view serializable or not. However, the problem or testing of view serializability has been shown to be NP-hard, meaning that finding an efficient polynomial time algorithm for this problem is highly unlikely.

Example-5.3 Consider 3 transaction T_1, T_2 and T_3

T_1 : $w(A)$; $R(C)$; $w(B)$

T_2 : $r(A)$; $r(B)$; $w(C)$; $w(B)$

T_2 : $w(C)$; $w(B)$

Find Schedule S : $r_2(A)$; $r_2(B)$; $w_1(A)$; $w_2(C)$; $R_1(C)$; $w_3(C)$; $w_1(B)$; $w_3(B)$ is view serializable or not.

(*i*) Initial read:

Data item	Transaction making initial read	Transaction which writes on data item
A	T_2	$T_1 \Rightarrow T_2 \rightarrow T_1$
B	T_2	$T_1T_3 \Rightarrow T_2 \rightarrow (T_1T_3)$
C	–	T_3T_2

(*ii*) *WR* sequence

$$w_2(C) \rightarrow R_1(C)$$

Find other transaction which writes on $C = T_3$.

This says $T_2 \rightarrow T_1$ and T_3 should not execute between T_2 and T_1

(*iii*) Final write on data item

A : $\textcircled{T_1}$

B : T_1, T_2, $\textcircled{T_3}$

C : T_2, $\textcircled{T_3}$

(T_1, T_2) should execute before T_3.

From (*i*), (*ii*) and (*iii*) view equivalent serial schedule

$T_2 \rightarrow T_1 \rightarrow T_3$

Classification of Schedule based on Recoverability

We now extend our definition of serializability to include aborted transactions. Intuitively, all actions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with. Using this intuition, we extent the definition of a serializable schedule as follows: A serializable schedule over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of committed transactions in S.

This definition of serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations.

For example, suppose that (1) an account transfer program T_1 deducts \$100 from account A, then (2) an interest deposit program T_2 reads the current values of accounts A and B and adds % interest to each, then commits, and then (3) T_1 is aborted. The corresponding schedule is shown in below figure.

T_1	T_2
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
Abort	

Now, T_2 has read a value for A that should never have been there. (Recall that aborted transactions' effects are not supposed to be visible to other transactions) If T_2 had not yet committed, we could deal with the situation by cascading the abort of T_1 and also aborting T_2, this process recursively aborts any transaction that read data written by T_2, and so on. But T_2 has already committed and so we can't undo its action we say that such a schedule is unrecoverable. In a **recoverable schedule**, transactions commit only after (and if!) all transactions whose changes they read commit. If transaction read only the changes of committed transactions, not only is the schedule recoverable, but also aborting a transaction can be accomplished without cascading the abort to other transactions. Such a schedule is said to **avoid cascading aborts**.

There is another potential problem in undoing the actions of a transaction. Suppose that a transaction T_2 overwrites the value of an object A that has been modified by a transaction T_1, while T_1 is still in progress, and T_1 subsequently aborts. All of T_1's changes to database objects are undone by restoring the value of any object that it modified to the value of the object before T_1's changes. When T_1 is aborted and its changes are undone in this manner, T_2's changes are lost as well, ever if T_2 decides to commit. So, for example, if A originally had the value 5, then was changed by T_1 to 6, and by T_2 to 7, if T_1 now aborts, the value of A becomes 5 again. Even if T_2 commits, its changes to A is inadvertently lost.

Cascadeless Rollback Recoverable Schedule

If a transaction performs a write operation on data item X then, other transactions read operation should be delayed until commit or roll back of original transaction.

Example:

T_1	T_2
$w(A)$	
	$R(A)$

If T_1 writes on A, T_2 wants to read A then T_2 should be delayed until commit/rollback of T_1. i.e. dirty ready are not allowed.

Strict schedule: A schedule is said to be strict if a value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits strict schedule are recoverable.

Example:

T_1	T_2
$R(A)$	
	$R(A)$
	$W(C)$
	C_2
$W(A)$	
C_1	

The above schedule is strict but not serializable schedule because RW problem exist. A schedule S is correct only if S is serializable as well as strict

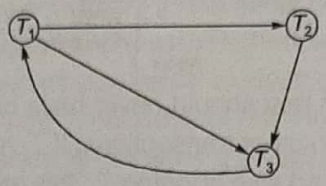
Conflict and view serializable schedule:

Example-5.4 Verify the conflict serializability of the given schedule.

$r_1(X); r_3(X); w_1(X); r_2(X); w_3(X)$

Solution:

Precedence graph ⇒



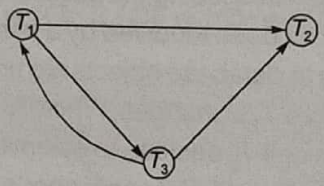
Not conflict serializable schedule

Example-5.5 Verify the conflict serializability of the given schedule.

$r_1(X); r_3(X); w_3(X); w_1(X); r_2(X)$

Solution:

Precedence graph



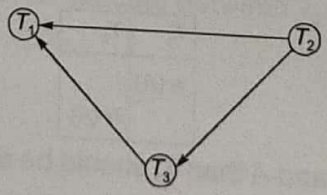
Not conflict serializable

Example-5.6 Verify the conflict serializability of the given schedule.

$r_3(X); r_2(X); w_3(X); r_1(X); w_1(X)$

Solution:

Precedence graph

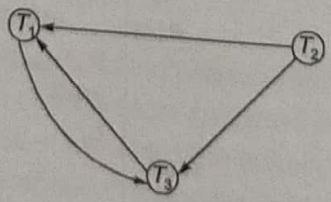


Conflict serializable schedule with equivalent serial schedule is T_2, T_3, T_1

Example-5.7 Verify the conflict serializability of the given schedule.

$r_3(X); r_2(X); r_1(X); w_3(X); w_1(X)$

Solution:



Not conflict serializable

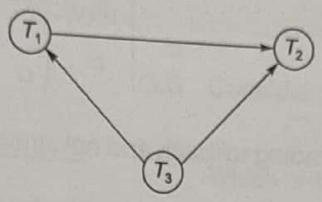
Example-5.8 Consider the schedules S_1 and S_2 given below. Draw the precedence graphs for S_1 and S_2 and state whether each schedule is serializable or not?

$S_1 : r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y)$

$S_2 : r_1(X); r_2(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y)$

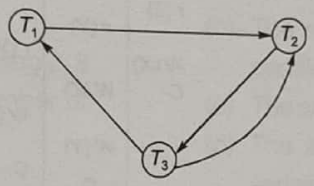
Solution:

Precedence graph for S_1 :



Conflict serializable schedule with equivalent serial schedule: T_3, T_1, T_2

Precedence graph for S_2 :



Not conflict serializable as precedence graph of S_2 has a cycle.

Example-5.9 Consider the schedules S_1, S_2 and S_3 below. Determine wheather each schedule is strict, cascadeless, recoverable or non recoverable?.

$S_1 : r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); \ c_1; w_3(Y); c_3; r_2(Y); w_2(Z); w_2(Y); c_2;$

$S_2 : r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y); \ c_1; c_2; c_3$

$S_3 : r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); c_1 \ w_2(Z); w_3(Y); w_2(Y); c_3; c_2$

Solution:

T_1	T_2	T_3
$r(X)$	$r(Z)$	
$r(Z)$		$r(X)$
$W(X)$		$r(Y)$
C		$W(Y)$
		C
	$r(Y)$	
	$W(Z)$	
	$W(Y)$	

A schedule is said to be irrecoverable if transaction T_2 reads A which is updated by uncommitted transaction T_i and commit of T_y before commit of T_i.

S_1 don't have any uncommitted read. So it is recoverable and casecadeless recoverable.

Since no lost updation hence strict recoverable.

S_2 :

T_1	T_2	T_3
$r(X)$		
	$r(Z)$	
$r(Z)$		$r(X)$
		$r(Y)$
$W(X)$		$W(Y)$
	$r(Y)$	
	$W(Z)$	
	$W(Y)$	
C		
	C	
		C

Not recoverable

Uncommitted read hence cascading rollback and not strict recoverable schedule.

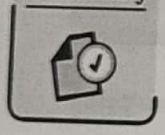
S_3 :

T_1	T_2	T_3
$r(X)$		
	$r(Z)$	
		$r(X)$
$r(Z)$		
	$r(Y)$	
		$r(Y)$
$W(X)$		
C	$W(Z)$	
		$W(Y)$
	$W(Y)$	
	C	C

Not uncommitted read hence recoverable and cascadeless.

Since lost update ($w-w$) problem hence not strict recoverable.

Summary



- In this chapter we discussed DBMS concepts for transaction processing. We introduced the concept of a database transaction and the operations relevant to transaction processing. We compared single-user systems to multiuser systems and then presented examples of how uncontrolled execution of concurrent transactions in a multiuser system can lead to incorrect results and database values.We also discussed the various types of failures that may occur during transaction execution.
- Next we introduced the typical states that a transaction passes through during execution, and discussed several concepts that are used in recovery and concurrency control methods. The system log keeps track of database accesses, and the system uses this information to recover from failures. A transaction either succeeds and reaches its commit point or it fails and has to be rolled back. A committed transaction has its changes permanently recorded in the database. We presented an overview of the desirable properties of transactions—atomicity, consistency preservation, isolation, and durability—which are often referred to as the ACID properties.
- We defined equivalence of schedules and saw that a serializable schedule is equivalent to some serial schedule. We defined the concepts of conflict equivalence and view equivalence, which led to definitions for conflict serializability and view serializability. A serializable schedule is considered correct. We presented an algorithm for testing the (conflict) serializability of a schedule. We discussed why testing for serializability is impractical in a real system, although it can be used to define and verify concurrency control protocols, and we briefly mentioned less restrictive definitions of schedule equivalence.

- Then we defined a schedule (or history) as an execution sequence of the operations of several transactions with possible interleaving. We characterized schedules in terms of their recoverability. Recoverable schedules ensure that, once a transaction commits, it never needs to be undone. Cascadeless schedules add an additional condition to ensure that no aborted transaction requires the cascading abort of other transactions. Strict schedules provide an even stronger condition that allows a simple recovery scheme consisting of restoring the old values of items that have been changed by an aborted transaction.

Student's Assignments 1

Q.1 Which of the following ACID property is maintained by concurrency control component?
(a) Atomicity (b) Consistency
(c) Isolation (d) Durability

Q.2 Consider 3 transactions T_1, T_2 and T_3 having 2, 3 and 4 operations respectively. Find number of concurrent schedules.
(a) 630 (b) 1260
(c) 2520 (d) 2440

Q.3 Which of the following schedules are conflict serializable?
S_1: $r_2(A)$, $r_1(C)$, $r_2(B)$, $w_2(B)$, $r_3(B)$, $r_1(A)$, $r_3(C)$, $w_3(C)$, $w_1(A)$
S_2: $r_2(A)$, $r_1(C)$, $r_2(B)$, $w_2(B)$, $r_1(A)$, $r_3(C)$, $r_3(C)$, $w_3(C)$, $w_1(A)$
(a) Only S_1 (b) Only S_2
(c) Both S_1 and S_2 (d) None of these

Q.4 Consider the following two schedules:
S_1: $w_1(a)$, $r_2(b)$, $r_1(b)$, $r_2(a)$, C_1, C_2
S_2: $r_1(a)$, $w_2(a)$, C_2, $w_1(a)$, C_1, $r_3(a)$, C_3
Which of the above schedule is recoverable?
(a) Only S_1 (b) Only S_2
(c) Both S_1 and S_2 (d) None of these

Q.5 Consider the following schedule:
$r_1(a)$, $r_2(a)$, $r_3(b)$, $w_2(a)$, $r_4(c)$, $r_3(b)$, $r_1(b)$, $r_2(b)$, $w_1(c)$, $w_4(a)$
The above schedule is serializable as
(a) T_2, T_1, T_3, T_4 (b) T_1, T_2, T_3, T_4
(c) T_1, T_3, T_2, T_4 (d) Not serializable

Q.6 Consider the following schedule
$r_1(x)$, $r_1(y)$, $w_2(x)$, $w_1(x)$, $w_3(x)$
Which of the following is true regarding the given schedule?
(a) The schedule is conflict serializable but not view serializable.
(b) The schedule is view serializable but not conflict serializable.
(c) The schedule is view serializable.
(d) The schedule is neither view nor conflict serializable.

Q.7 How many view equal serial schedules possible for the following schedule?
S: $w_1(A)$ $r_2(A)$ $w_3(A)$ $r_4(A)$ $w_5(A)$ $r_6(A)$ $w_7(A)$ $r_8(A)$
(a) 1 (b) 4
(c) 6 (d) 8

Q.8 Due to cascade rollback problem in schedule there is problem of
(a) Inconsistency (b) Data loss
(c) More I/O cost (d) None of these

Q.9 Consider the following statement which is true.
(a) Time complexity to check given schedule is view serializable or not is $O(2^n)$.
(b) Time complexity to check given schedule is conflict serializable or not is $O(n^2)$.
(c) Both (a) and (b)
(d) None of these

Answer Key:

1. (c)	**2.** (b)	**3.** (a)	**4.** (c)	**5.** (d)
6. (b)	**7.** (c)	**8.** (c)	**9.** (c)	

Student's Assignments Explanations

1. (c)
Isolation is maintained by concurrency control component.

2. (b)
If T_1, T_2 and T_3 are transactions with p, q and r operations respectively

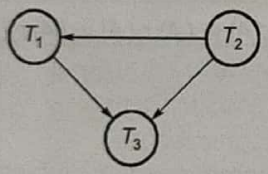
Number of concurrent schedules $= \frac{(p+q+r)!}{p!\,q!\,r!}$

$= \frac{(2+3+4)!}{2!\,3!\,4!} = \frac{9!}{2\times 6\times 4!}$

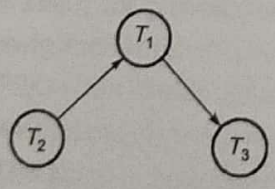
$= \frac{9\times 8\times 7\times 6\times 5}{2\times 6} = 63 \times 20$

$= 1260$

3. (a)
Schedule S_1 is conflict serializable because the precedence graph has no cycles.



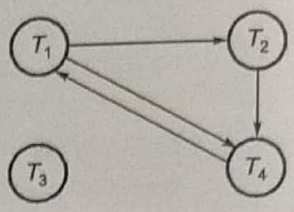
The only possible conflict-equivalent serial schedule is (T_2, T_1, T_3). Schedule S_2 is also conflict serializable having conflict equivalent serial schedule is (T_2, T_1, T_3).



4. (c)
Irrecoverable schedule.

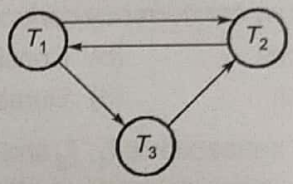
T_1	T_2
w(a)	
	r(a)
	C
C/R	

5. (d)
The precedence graph of the above schedule



Since there is a cycle T_1, T_2, T_4, T_1 so schedule is not serializable.

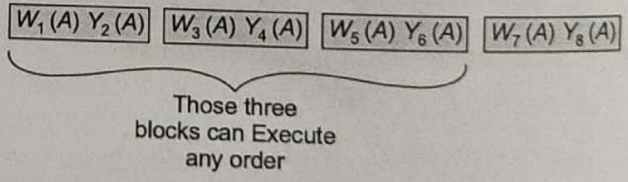
6. (b)



Precedence graph of the schedule contains cycle so the schedule is not serializable.
View serializable schedule is

T_1	T_2	T_3
$r(x)$		
$r(y)$		
$w(x)$		
	$w(x)$	
		$w(x)$

7. (c)



Student's Assignments 2

Q.1 Contains complete record of all activity that affected the contents of a database during a certain period of time ?

Q.2 Which scenario may lead to an irrecoverable schedule?

Q.3 Consider the following two schedules:

S_1:

T_1	T_2
W(a)	R(b)
R(b)	R(a)
C1	C2

S_2:

T_1	T_2	T_3
R1(a)		
	W2(a)	
	C2	
W1(a)		
C1		R3(a)
		C3

Which of the above schedule is recoverable?

Q.4 What is transaction? In what ways is it different from an ordinary C program.

Q.5 What are the potential problem may occur when DBMS executes multiple transaction concurrently

Q.6 Consider a database with object X and Y and assume that there are two transactions T_1 and T_2. T_1 reads object X and Y and writes object X T_2 reads object X and Y and writes object X and Y.

(*i*) Given an example schedule that results in
- (a) write_read conflict.
- (b) read_write conflict.
- (c) write_write conflict.

(*ii*) For schedules in (*a*), (*b*) and (*c*) show that strict 2PL disallows the schedule

Q.7 Which of the following schedule is conflict equivalent to each other

S_1: $R_2(A)$; $W_2(A)$; $R_3(C)$; $W_2(B)$; $W_3(A)$; $W_3(C)$; $R_1(A)$; $R_1(B)$; $W_1(A)$; $W_1(B)$

S_2: $R_3(C)$; $R_2(A)$; $W_3(A)$; $W_2(B)$; $W_3(A)$; $R_1(A)$; $R_1(B)$; $W_1(A)$; $W_1(B)$; $W_3(C)$

S_3: $R_2(A)$; $R_3(C)$; $W_3(A)$; $W_2(A)$; $W_2(B)$; $W_3(C)$; $R_1(A)$; $R_1(B)$; $W_1(A)$; $W_1(B)$

Q.8 Consider the following schedules S_1, S_2, S_3 below. Determine wheather each schedule is strict, cascadeless, recoverable or non recoverable. (Determing strongest recoverability condition that each schedule satisfies)

S_1: $R_1(X)$; $R_2(Z)$; $R_1(Z)$; $R_3(X)$; $R_3(Y)$; $W_1(X)$; C_1; $W_3(Y)$; C_3 $r_2(Y)$; $W_2(Z)$; $W_2(Y)$; C_2

S_2: $R_1(Z)$; $R_2(Z)$; $R_3(X)$; $R_1(Z)$; $R_2(Y)$; $R_3(Y)$; $W_1(X)$; C_1; $W_2(Z)$; $W_3(Y)$; $W_2(Y)$; C_3; C_2

Q.9 Which of the following schedules is conflict serializable for each serializable schedule, determine the equivalent serial schedule?

S_1: $R_1(A)$; $R_2(A)$; $R_3(B)$; $W_1(A)$; $R_2(C)$; $R_2(B)$; $W_2(B)$; $W_1(C)$

S_2: $R_3(A)$; $R_2(A)$; $R_1(A)$; $W_3(A)$; $W_1(A)$;

S_3: $W_3(A)$; $R_1(A)$; $W_1(B)$; $R_2(B)$; $W_2(C)$; $R_3(C)$

S_4: $R_1(A)$; $R_2(A)$; $W_1(B)$; $W_2(B)$; $R_1(B)$; $R_2(B)$; $W_2(C)$; $W_1(D)$

Q.10 For the following schedules find all view equivalent serial schedule

S_1: $R_1(A)$; $R_2(A)$; $R_3(A)$; $W_1(B)$; $W_2(B)$; $W_3(C)$

S_2: $W_1(A)$; $R_2(A)$; $W_3(A)$; $R_4(A)$; $W_5(A)$; $W_5(A)$

S_3: $R_2(A)$; $R_1(A)$; $W_1(C)$; $R_3(C)$; $W_1(B)$; $R_4(B)$; $R_4(B)$; $W_3(A)$; $R_4(C)$; $W_2(D)$; $R_2(B)$; $W_4(A)$; $W_4(B)$

CHAPTER 06

Concurrency Control Techniques

6.1 Introduction

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the data base. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

A DBMS must be able to ensure that only serializable, recoverable schedule are allowed and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a locking protocol to achieve this. A lock is a small bookkeeping object associated with a database object. A locking protocol is a set of rules to be followed by each transaction (and enforced by the DBMS) to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order. Different locking protocols use different types of locks, such as shared locks or exclusive locks.

6.2 Shared-Exclusive Locking

Shared lock (*S*): Shared lock is required for reading a data item. Many transaction may hold a lock on the same data item in shared locking mode

Exclusive lock (*X*): If a transaction is to write an data item, it must have exclusive lock to that data item.

Lock compatible table

New request by transaction T_j ↓	Transaction T_i already hold lock	
	S	*X*
S	True	False
X	False	False

6.3 Two Phase Locking Protocol (2PL)

2 *PL* insures the serializability i.e. it does not allow to execute any non serializable schedule. In 2 *PL* a transaction can not request additional locks once it releases any lock. Thus every transaction has a growing phase in which it acquires locks, followed by a shrinking phase in which it releases locks

Intuitively, an equivalent serial order of transactions is given by the order in which transactions enter their shrinking phase: If T_2 reads or writes an object written by T_1, T_1 must have released its lock on the object before T_2 requested a lock on this object. Thus, T_1 precedes T_2. (A similar argument shows that T_1 precedes T_2 if T_2 writes an object previously read by T_1). A formal proof of the claim would have to show that there is no cycle of transactions that 'precede' each other by this argument.

Example-6.1 Consider the following schedule S:

Check if schedule S is allowed by 2 *PL* or not?

T_1	T_2	T_3
$W(A)$		$R(C)$
	$R(A)$	$R(D)$
	$W(D)$	
$W(B)$		$R(A)$

Solution:

The schedule is allowed by 2 *PL* as follows:

T_1	T_2	T_3
$X(A)$		$S(C)$
$W(A)$		$R(C)$
$X(B)$ $U(A)$		
L_{P1}	$S(A)$	
	$R(A)$	
		$S(D)$
		$R(D)$
		$S(A)$ $U(D)$
		L_{p2}
	$X(D)$	
	$U(D)$	
	$U(A)$	
$W(B)$	L_{p3}	$R(A)$
$U(B)$		$U(A)$

L_P: lock point (Point between last lock and first unlock). Equivalent serial schedule is based upon the lock points of transactions $T_1 \rightarrow T_3 \rightarrow T_2$. Every schedule which is allowed by 2 *PL* is also conflict serializable schedule but converse is not true.

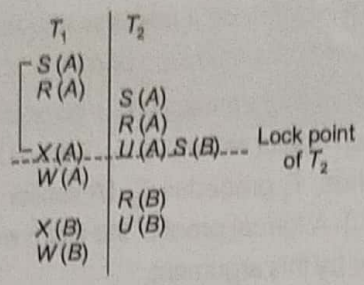
Lock Upgrading

A transaction T can upgrade from shared to exclusive on some data item if T does not perform any unlock operation.

$S(A)$
$R(A)$
No unlocks operation
$X(A)$
$W(A)$

Lock upgrading technique improves degree of concurrency.

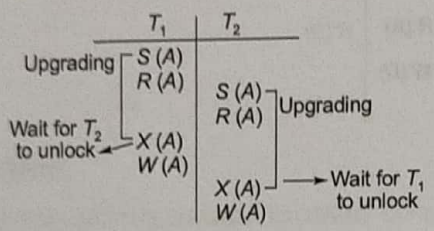
Example of a schedule which is not allowed by 2*PL*, but allowed by 2*PL* with lock upgradation equivalent serial schedule based on lock points

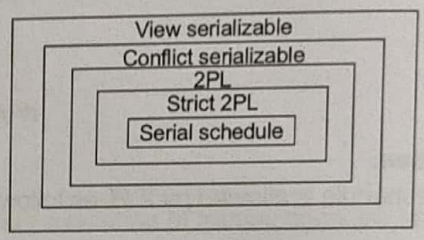


Lock upgrading may leads to additional deadlocks.

Strict 2*PL*: In strict 2*PL* transaction holds all exclusive lock until commit/abort. Shared locks can be unlocked before commit also. Hence strict 2*PL* ensures recoverability and serializability.

Limitation of 2*PL* ⇒ May not free from starvation, deadlock





6.4 Time Stamp based Concurrency Control

In lock based concurrency control, conflicting actions of different transactions are ordered by the order in which locks are obtained. If a transaction needs an item that is already locked, it may be forced to wait. Until the item is released there by ensuring serializability.

The use of locks, combined with the 2*PL* protocol, guarantees serializability of schedules. The serializable schedule produced by 2*PL* have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equivalent serial schedule.

Timestamp is a unique identified created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time. We will refer to the timestamp of transaction *T* as *TS* (*T*). Concurrency control techniques based on timestamp ordering do not use locks; hence, **deadlocks cannot occur**.

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, ... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

6.5 The Timestamp Ordering Algorithm

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the *only equivalent serial schedule permitted* has the transactions

in order of their timestamp values. This is called **timestamp ordering (TO)**. Notice how this differs from *2PL*, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction timestamps. The algorithm must ensure that, for each item accessed by *conflicting operations* in the schedule, the order in which the item is accessed does not violate the timestamp order. To do this, the algorithm associates with each database item X two timestamp (*TS*) values:

- **read_TS(X).** The **read timestamp** of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, read_$TS(X)$ = $TS(T)$, where T is the *youngest* transaction that has read X successfully.
- **write_TS(X).** The **write timestamp** of item X is the largest of all the timestamps of transactions that have successfully written item X—that is, write_$TS(X)$ = $TS(T)$, where T is the *youngest* transaction that has written X successfully.

Basic Timestamp Ordering (TO)

Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation, the **basic TO** algorithm compares the timestamp of T with read_$TS(X)$ and write_$TS(X)$ to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a *new timestamp*. If T is aborted and rolled back, any transaction T_1 that may have used a value written by T must also be rolled back. Similarly, any transaction T_2 that may have used a value written by T_1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic *TO*, since the schedules produced are not guaranteed to be recoverable. An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict. We first describe the basic *TO* algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Whenever a transaction T issues a write_item(X) operation, the following is checked:
   (*i*) If read_$TS(X)$ > $TS(T)$ or if write_$TS(X)$ > $TS(T)$, then abort and roll back T and reject the operation. This should be done because some *younger* transaction with a timestamp greater than $TS(T)$—and hence *after* T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
   (*ii*) If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_$TS(X)$ to $TS(T)$.
2. Transaction T issues a read_item(X) operation:
   (*i*) If write_$TS(X)$ > $TS(T)$, then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than $TS(T)$—and hence *after* T in the timestamp ordering—has already written the value of item X before T had a chance to read X.
   (*ii*) If write_$TS(X)$ ≤ $TS(T)$, then execute the read_item(X) operation of T and set read_$TS(X)$ to the *larger* of $TS(T)$ and the current read_$TS(X)$.

Hence, whenever the basic *TO* algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic *TO* are hence guaranteed to be *conflict serializable*, like the *2PL* protocol. However, some schedules are possible under each protocol that are not allowed under the other. Thus, *neither* protocol allows *all possible* serializable schedules. As mentioned earlier, deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Strict Timestamp Ordering (TO)

A variation of basic TO called **strict TO** ensures that the schedules are both **strict** (for easy recoverability) and (conflict) serializable. In this variation, a transaction T that issues a read_item(X) or write_item(X) such that $TS(T') >$ write_$TS(X)$ has its read or write operation *delayed* until the transaction T that *wrote* the value of X (hence $TS(T') =$ write_$TS(X)$) has committed or aborted. To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T' until T is either committed or aborted. This algorithm *does not cause deadlock*, since T waits for T' only if $TS(T) > TS(T')$.

Thomas's Write Rule: A modification of the basic TO algorithm, known as **Thomas's Write Rule**, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:

- If read_$TS(X) > TS(T)$, then abort and roll back T and reject the operation.
- If write_$TS(X) > TS(T)$, then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than $TS(T)$—and hence after T in the timestamp ordering—has already written the value of X. Thus, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
- If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_$TS(X)$ to $TS(T)$.
- Thomas write rule allows greater potential concurrency.
- Allows some view serializable schedules that are not conflict serializable.

6.6 Multiversion Concurrency Control Techniques

Other protocols for concurrently control keep the old values of a data item when the item is updated. These are known as multi version concurrency control, because several versions (values) of an item are maintained. When a transaction requires access to an item, an appropriate version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an older version of the item to maintain serializability. When a transaction writes an item, it writes a new version and the old version of the item is retained. Some multi version concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multi version techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway-for example, for recovery purposes. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data items values. The extreme case is a temporal database, which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multi version techniques, since older versions are already maintained.

Multi version Techniques Based on Timestamp Ordering

In this method, several versions $X_1, X_2, \ldots X_k$ of each data items X are maintained. For each version, the values of version X_i and the following two timestamps are kept:

- read_$TS(X_i)$. The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_i.
- write_$TS(X_i)$. The write timestamp of X_i is the timestamps of transactions that wrote the value of version X_i.

Whenever a transaction T is allowed to execute a write_item(X) operation, a new version X_{k+1} of item X is created, with both the write_$TS(X_{k+1})$ and the read_$TS(X_{k+1})$ of version X_i, the value read_$TS(X_i)$ is set to the larger of the current read_$TS(X_i)$ and $TS(T)$. To ensure serializability, the following rules are used:

- If transaction T issues a write_$TS(X)$ operation, and version i of X has the highest write_$TS(X_i)$ of all versions of X that is also less than or equal to $TS(T)$, and read_$TS(X_i) > TS(T)$, then abort and roll back transaction T; otherwise, create a new version X_j of X with read_$TS(X_j)$ = write_$TS(X_j) = TS(T)$.
- If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_$TS(X_i)$ of all versions of X that is also *less than or equal to* $TS(T)$; then return the value of X_i to transaction T, and set the value of read_$TS(X_i)$ to the larger of $TS(T)$ and the current read_$TS(X_i)$.

As we can see in case 2, a read_item(X) is always successful, since it finds the appropriate version X_i to read based on the write_TS of the various existing versions of X. In case 1, however, transaction T may be aborted and rolled back. This happens if T attempts to write a version of X that should have been read by another transaction T' whose timestamp is read_$TS(X_i)$; however, T' has already read version X_i, which was written by the transaction with timestamp equal to write_$TS(X_i)$. If this conflict occurs, T is rolled back; otherwise, a new version of X, written by transaction T, is created. Notice that, if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

- Oserve that
- Reads always succeds
- A write by T_i is rejected if some transaction T_j that should read T_i's write, has already read a version created by a transaction older than T_i.

Summary

- In this chapter we discussed DBMS techniques for concurrency control.We started by discussing lock-based protocols, which are by far the most commonly used in practice. We described the two-phase locking (2*PL*) protocol and a number of its variations: basic 2*PL*, strict 2*PL*, conservative 2*PL*, and rigorous 2*PL*. The strict and rigorous variations are more common because of their better recoverability properties. We introduced the concepts of shared (read) and exclusive (write) locks, and showed how locking can guarantee serializability when used in conjunction with the two-phase locking rule.
- We presented other concurrency control protocols that are not used often in practice but are important for the theoretical alternatives they show for solving this problem. These include the timestamp ordering protocol, which ensures serializability based on the order of transaction timestamps. Timestamps are unique, system-generated transaction identifiers. We discussed Thomas's write rule, which improves performance but does not guarantee conflict serializability. The strict timestamp ordering protocol was also presented. We discussed two multiversion protocols, which assume that older versions of data items can be kept in the database. One technique, called multiversion two-phase locking (which has been used in practice), assumes that two versions can exist for an item and attempts to increase concurrency by making write and read locks compatible (at the cost of introducing an additional certify lock mode). We also presented a multiversion protocol based on timestamp ordering, and an example of an optimistic protocol, which is also known as a certification or validation protocol.

Student's Assignment

Q.1 How many concurrent schedules can be formed with 3 transactions having 3, 2 and 1 operations respectively?

Q.2 Consider the following schedule:

S : $r_1(A)$, $r_3(D)$, $w_1(B)$, $r_2(B)$, $r_4(B)$, $w_2(C)$, $r_5(C)$, $w_4(E)$, $r_5(E)$, $w_5(B)$

How many serial schedules conflict equal to schedules (S)?

(a) 10 (b) 15
(c) 8 (d) 12

Q.3 Consider the following schedule:

S : $r_2(A)$, $w_1(B)$, $w_1(C)$, $r_3(B)$, $r_2(B)$, $r_1(A)$, c_1 $r_2(C)$, c_2 $w_3(A)$ c_3

How many given statements true about schedule(S)?

(i) Schedule(S) is conflict serializable schedule.
(ii) Schedule(S) is allowed by 2PL.
(iii) Schedule(S) is strict recoverable schedule.
(iv) Schedule(S) is allowed by strict 2PL.

(a) 1 (b) 2
(c) 3 (d) 4

Answer Key:

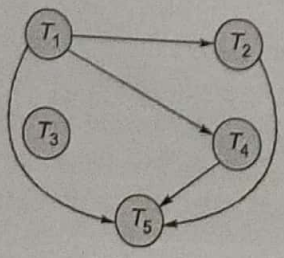
1. (60) **2.** (a) **3.** (b)

Student's Assignments Explanations

1. (60)

$$\frac{(3+2+1)!}{3!\cdot 2!} = \frac{6!}{3!\cdot 2!} = \frac{6\times5\times4\times3!}{3!\times2} = 60$$

2. (a)



Number of serial schedules conflict equal to schedule (S) is number of topological orders?

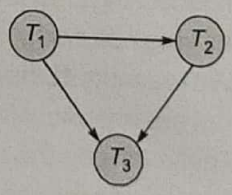
T_1 < $T_1 - T_4 - T_5$, $T_4 - T_2 - T_5$ } 2 sequences for $T_1T_2T_4T_5$

T_3 can be any where in both sequences. Total 10 topological order.

3. (b)

T_1	T_2	T_3
	S(A)	
	r_2(A)	
X(B)		
w_1(B)		
X(C)		
W(C)		
S(A)		
U(A)		
		S(B)
		r_3(B)
	S(B)	
	r_2(B)	
S(A)		
R(A)		
$c_1 \cup$ (B)		
	r_2(C)	
	$c_2 \cup$ (A) $\cup$ (B)	
		X(A)
		w_3(A)
		$c_3 \cup$ (B) $\cup$ (A)

(i)



Conflict serializable

(ii) Allowed by 2PL.
(iii) Not strict recoverable.
(iv) No allowed by strict 2PL.

CHAPTER 07

File Organization and Indexing

7.1 File Organization

The database is stored as a collection of files. Each file is a sequence of records. A record is a sequence of fields. A file descriptor (or file header) includes information that describes the files, such as the field names and their data types, and the addresses of the file blocks on disk.

Records are stored on disk blocks. The blocking factor for a file is the average number of file records stored in a disk block. For storing the records within the blocks we have two strategy.

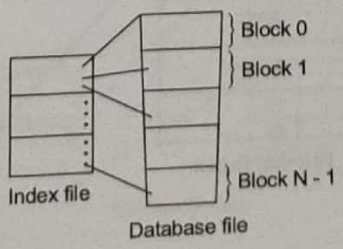
1. **Spanned Organization:** Spanned organization allow part of record to be in one block and rest of it to be on the next block.
2. **Unspanned organization:** Record not allowed to span in more than one block i.e. entire record should be in one block
   - No internal fragmentation in spanned organization
   - Less I/O cost needed in unspanned organization because to load any record only One disk block access is required.
   - If records are of fixed length then unspanned organization is preferred. For variable length records spanned organization is preferred.

   I/O cost: When record is accessed, number of memory blocks required do access the record is called as I/O cost.
   - Block is collection of records
   - Database file is collection of blocks.

Suppose a database file has N blocks and we want to search for record X then maximum number of block accessed = N (when records are unordered).

If records are ordered then maximum block accessed = $\lceil \log_2 N \rceil$

Indexing: Index file can be created for the database that index file also divided into blocks.



Each entry of index file is < search key, pointer>

Index file entry size << *DB* file entry size number of index blocks (*M*) << number of database file blocks (*N*).

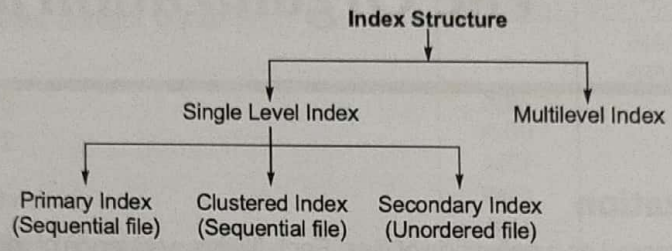
I/O cost with indexing = $\lceil \log_2 N \rceil + 1$

↓ ↓

Index file blocks database file block

7.2 Index Structure

There are two types of indexing.



Indices can also be classified as

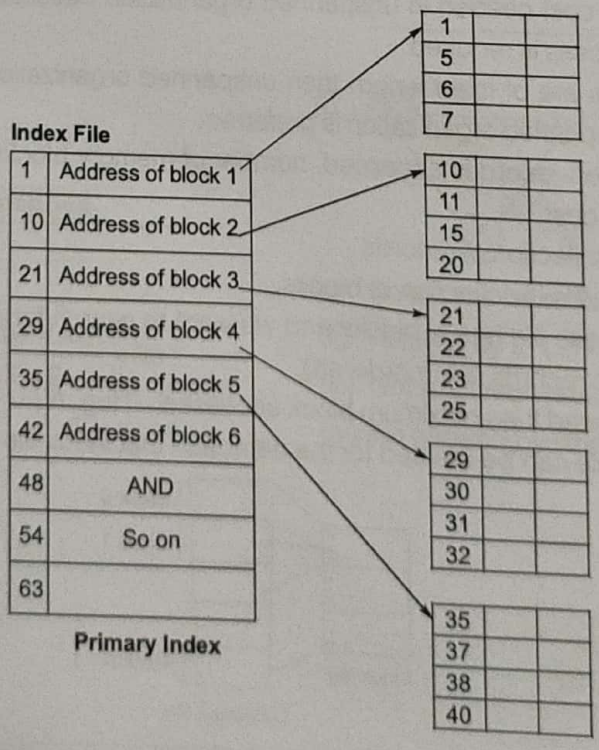
Sparse Index: has index entries for only some of the search values.

Dense Index: has index entries for every search key value (and hence every record) in the data file.

Single Level Indexing

Primary Index (Sorted Over key Attribute): Primary index is sorted file and records in it will be in this form $\boxed{K}\boxed{P_r}$ where K is primary key over the file is sorted and P_r is data block pointer.

Anchor Record: First record of each blocks is called as anchor record.



NOTE

- The index file may or may not fit in a block i.e. index file can take more than one block space. It depends on its size.
- It is clear that (from figure) primary indexing is sparse/non dense indexing. On ordered (sorted) key attribute
- Record in which empid is 1, 10, 21, 29, 35 are anchor records.
- The index entry has the key field value for the first record in the block, also called as block anchor.
- A record whose primary key value is k lies in the block whose address is $p(i)$ where k $(i) \le k < k(i+1)$.
- To get a record with a value k, we do a binary search on the index file to find the appropriate index entry i and then retrieve the data file block whose address is $p(i)$.

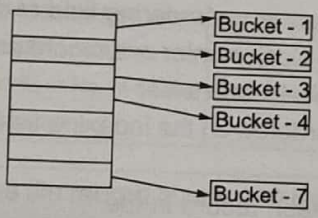
Insertion and Deletion of Records

If we want to insert new record we need to move records to make space for the new record but also change some index entries, since moving records will change the anchor records of some blocks. Solution to above problem:

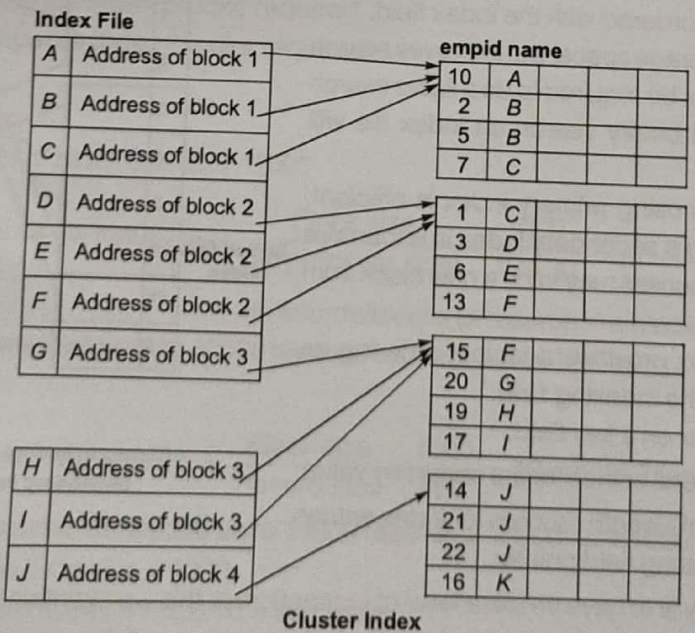
- Using an unordered overflow file.
- Linked list of overflow records for each block in data file.

Record deletion is handled by deletion markers.

Example:



Clustering Index (Sorted Over Non Key Attribute)

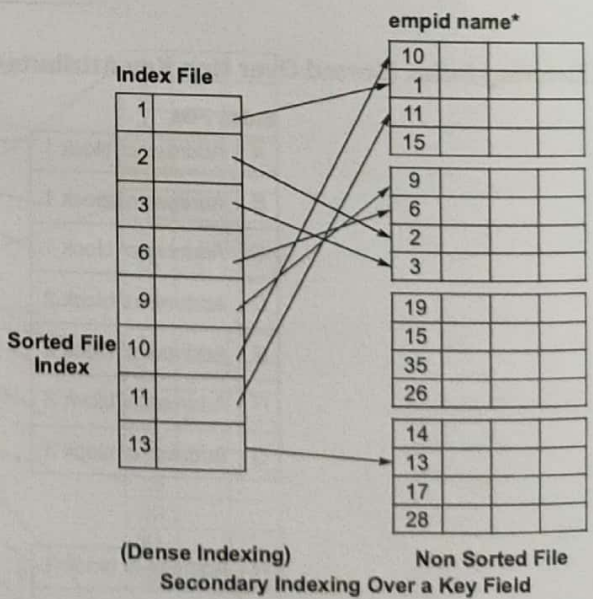


NOTE

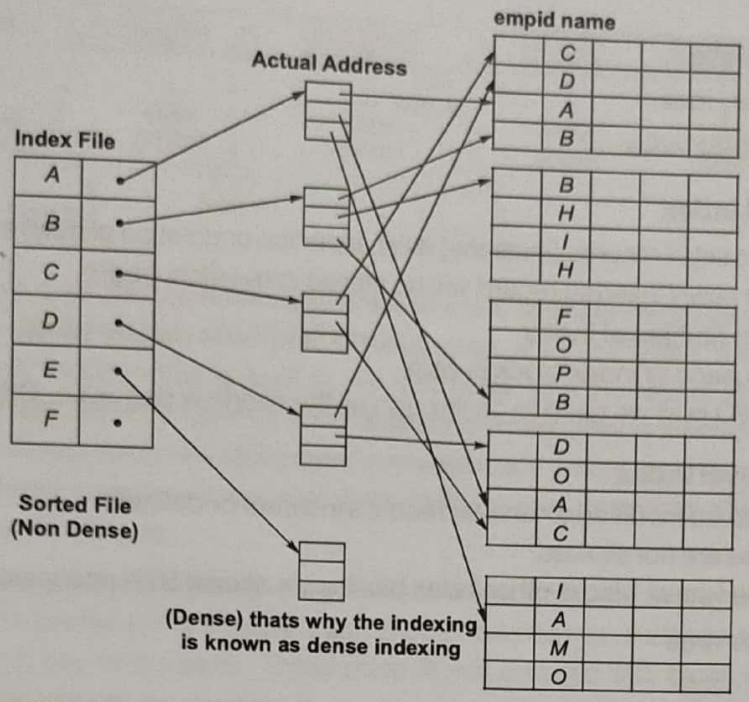
- The records are sorted over the attribute name which is not a primary key.
- Pointers are pointing the first address of the block (in figure).
- If records of a file are physically ordered on a non key field which doesn't have a distinct value for each record-the field is called clustering field. So, we create clustering index.
- There is an entry in the clustering index for each distinct value of the clustering field, containing the value and a pointer to the first block in the data file that has a record with that value for its clustering field.
- INSERTION and DELETION are still troublesome.
- To solve insertion problem we keep an extra block for each value of clustering field, all records with that value are placed in the block.
- It is a non dense or sparse index.

Secondary Index (Single Level Unordered Indices)

- It is not sparse *i.e.* it is dense.
- It provides the secondary mean of accessing a file for which some primary access already exists.
- The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non key with duplicate values.
- The index is an ordered file with 2 fields.
  (*i*) the first field is secondary key or nonordering field of data file that is the indexing field.
  (*ii*) The second field is either a block pointer or a record pointer.
- There can be many secondary indices in a file.
- The data file is not physically ordered on the indexing field, hence insertion and deletion is not so expensive.
- An index entry is created for each record in the data because we can not have block anchors here as the file is not ordered with the index field.
- It needs more storage space, but improves search time as we will not be required to do a linear search on data file and binary search on index file will suffice.
- Sequential scan using primary index is efficient, but a scan using a secondary index is expensive as each record access may fetch a new block from disk.
- Secondary index provides a logical ordering on the records by the indexing field.
- Secondary index on a key field:
  (*i*) Include several entries for the same key value.
  (*ii*) Have variable length records for index entries with a repeating field pointer.
  (*iii*) Another way is to have an extra level of indirection. In this we maintain the index file having block pointers pointing to the block, where the actual address of the records is stored.



Secondary Indexing Over a Key Field



Type of Index	No. of (First-Level) Index Entries	Dense of Nondense	Block Anchoring on The Data File
Primary	No. of blocks in data file	Nondense	
Clustering	No. of distinct index field values	Nondense	Yes
Secondary (key)	No. of records in data file	Dense	Yes/No*
Secondary (nonkey)	No. of records or No. of distinct index field values	Dense or nondense	No No

Example-7.1 In a database file record size = 100 bytes, block size = 1000 bytes key = 10 bytes, pointer size = 10 bytes. Database file consist of 10000 records.

(*i*) How many dense index blocks required

(*ii*) How many sparse index blocks required

Solution:

(*i*) Dense index

Number of index entries = 10000

$$\text{Block factor for index file} = \frac{\text{Block size}}{(\text{key+Pointer size})} = \frac{1000}{10+10}$$

= 50 records/blocks (# records in a block)

Number of required to store 10000 records = 100/50 = 200 blocks

(*ii*) Sparse index

$$\text{Block factor of database file} = \frac{\text{Block size}}{\text{Record size}} = \frac{1000}{100} = 10 \text{ record /block}$$

Database blocks required to store 10000 records = 10000/10 = 1000 blocks

entries in the index file = 1000

Sparse index blocks = 1000/50 = 20 blocks

7.3 Multilevel Index

(*i*) Static multilevel index

(*ii*) Dynamic multilevel index

7.3.1 Static Multilevel Index

Index file is build on set of records (instance) exist. Insertion or deletion of new record does not effect the index file modification. The newly inserted record will be stored in overflow pages

Limitation of Static Multilevel Index

- In worst case usage of index block is zero
- In worst case I/O cost we need to go through all the overflow pages i.e. O(N)

7.3.2 Dynamic Multilevel Index

- Index file has to be modified whenever records inserted or deleted
- Overflow pages are not allowed.
- Except last level index blocks other index blocks are atleast 50% occupied

Example: B tree, B^+ tree

7.4 B-Trees

The B-tree has constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances—namely, whenever we attempt an insertion into a node that is already full or a deletion from a node that makes it less than half full. More formally, a **B-tree of order** p, when used as an access structure on a *key field* to search for records in a data file, can be defined as follows:

(*i*) Each internal node in the *B*-tree is of the form

$$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_q>$$

Where $q \le p$. Each P_i is a **tree pointer**—a pointer to another node in the *B*-tree. Each Pr_i is a **data pointer**—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).

(*ii*) Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.

(*iii*) For all search key field values X in the subtree pointed at by P_i (the i^{th} subtree, we have: $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.

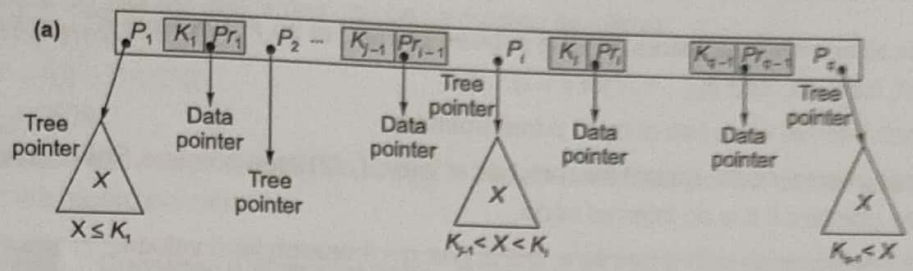
(*iv*) Each node has at most p tree pointers.

(*v*) Each node, except the root and leaf nodes, has **at least** $\lceil p/2 \rceil$ tree pointers. The root node has **at least two** tree pointers unless it is the only node in the tree.

(*vi*) A node with q tree pointers, $q \le p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).

(*vii*) All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers* P_i are NULL.

Notice that all search values K in the *B*-tree are unique because we assumed that the tree is used as an access structure on a key field. If we use a B-tree *on a nonkey field*, we must change the definition of the file pointers Pr_i to point to a block—or a cluster of blocks—that contain the pointers to the file records. This extra level of indirection is similar to discussed in secondary indexes.



A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with $p-1$ search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes. If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split.

Suppose the search field is $V = 9$ bytes long, the disk block size is $B = 512$ bytes, a record (data) pointer is $P_r = 7$ bytes, and a block pointer is $P = 6$ bytes. Each *B*-tree node can have at most p tree pointers, $p-1$ data pointers, and $p-1$ search key field values. These must fit into a single disk block if each *B*-tree node is to correspond to a disk block. Hence, we must have:

$$(p * P) + ((p-1) * (P_r + V)) \le B$$

$$(p * 6) + ((p-1) * (7+9)) \le 512$$

$$(22 * p) \le 528$$

B-trees are sometimes used as primary file organizations. In this case, whole records are stored within the *B*-tree nodes rather than just the <search key, record pointer> entries. This works well for files with a relatively small number of records and a small record size. Otherwise, the fan-out and the number of levels become too great to permit efficient access.

In summary, *B*-trees provide a multilevel access structure that is a balanced tree structure in which each node is at least half full. Each node in a *B*-tree of order p can have at most $p-1$ search values.

Most implementations of a dynamic multilevel index use a variation of the *B*-tree data structure called a **B^+-tree**. In a *B*-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B^+-tree, **data pointers are stored only at the leaf nodes of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes.** The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field. For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

The leaf nodes of the B^+-tree are usually linked to provide ordered access on the search field to the records. These leaf nodes are similar to the first (base) level of an index. Internal nodes of the B^+-tree correspond to the other levels of a multilevel index. Some search field values from the leaf nodes are *repeated* in the internal nodes of the B^+-tree to guide the search. The structure of the *internal nodes* of a B^+- tree of order p is as follows:

(*i*) Each internal node is of the form

$$<<P_1, K_1>, <P_2, K_2> \ldots, <P_{q-1}, K_{q-1}>, P_q>$$

Where $q \le p$ and each P_i is a **tree pointer.**

(*ii*) Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$.

(*iii*) For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$

(*iv*) Each internal node has at most p tree pointers.

(*v*) Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.

(*vi*) An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

The structure of the leaf *nodes* of a B^+-tree of order p is as follows:

(*i*) Each leaf node is of the form

$<<K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{next}>$

Where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node* of the B^+-tree.

(*ii*) Within each leaf node, $K_1 \leq K_2 \ldots, K_{q-1}, q \leq p$.

(*iii*) Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).

(*iv*) Each leaf node has at least $\lceil (p/2) \rceil$ values.

(*v*) All leaf nodes are at the same level.

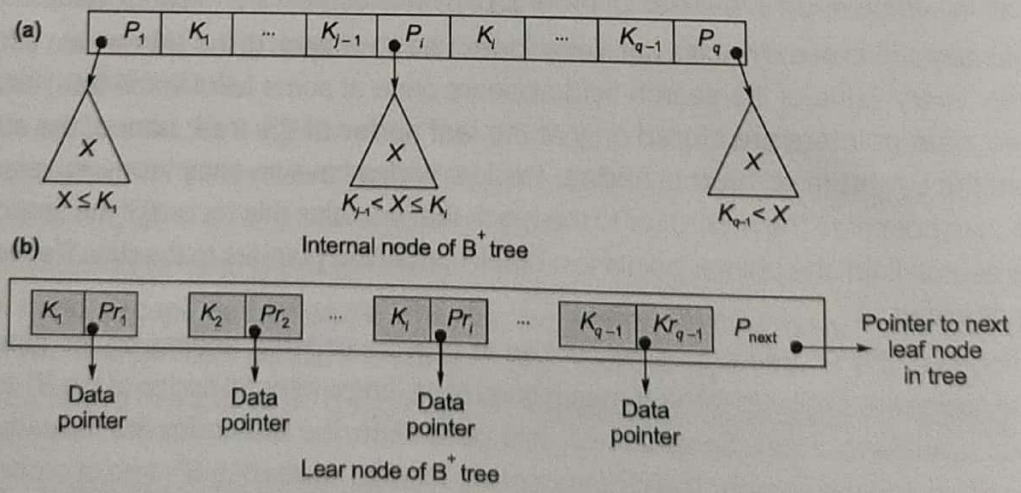
To calculate the order p of a B^+-tree, suppose that the search key field is $V = 9$ bytes long, the block size is $B = 512$ bytes, a record pointer is $Pr = 7$ bytes, and a block pointer is $P = 6$ bytes. An internal node of the B^+-tree can have up to p tree pointers and $p - 1$ search field values; these must fit into a single block.

Hence, we have:

$$(p * P) + ((p - 1) * V) \leq B$$

$$(P * 6) + ((P - 1) * 9) \leq 512$$

$$(15 * p) \leq 521$$



We can choose p to be the largest value satisfying the above inequality, which gives $p = 34$. This is larger than the value of 23 for the B-tree (it is left to the reader to compute the order of the B-tree assuming same size pointers), resulting in a larger fan-out and more entries in each internal node of a B^+-tree than in the corresponding B-tree. The leaf nodes of the B^+-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer.

Hence, the order p_{leaf} for the leaf nodes can be calculated as follows:

$(p_{leaf} * (Pr + V)) + P \le B$

$(p_{leaf} * (7 + 9)) + 6 \le 512$

$(16 * p_{leaf}) \le 506$

It follows that each leaf node can hold up to $p_{leaf} = 31$ key value/data pointer combinations, assuming that the data pointers are record pointers.

Example-7.2 Consider a B tree of order P. Find the maximum and minimum number of keys in a B tree of height h.

Solution:

Height	Minimum Nodes	Minimum Block Pointer	Minimum keys	Maximum Nodes	Maximum Block Pointer	Maximum keys
0	1	2	1	1	P	$P-1$
1	2	$2\left\lceil\frac{P}{2}\right\rceil$	$2\left(\left\lceil\frac{P}{2}\right\rceil-1\right)$	P	P^2	$P(P-1)$
2	$2\left\lceil\frac{P}{2}\right\rceil$	$2\left\lceil\frac{P}{2}\right\rceil^2$	$2\left\lceil\frac{P}{2}\right\rceil\left(\left\lceil\frac{P}{2}\right\rceil-1\right)$	P^2	P^3	$P^2(P-1)$
3	$2\left\lceil\frac{P}{2}\right\rceil^2$	$2\left\lceil\frac{P}{2}\right\rceil^3$	$2\left\lceil\frac{P}{2}\right\rceil^2\left(\left\lceil\frac{P}{2}\right\rceil-1\right)$	P^3	P^4	$P^3(P-1)$
M						
h	$2\left\lceil\frac{P}{2}\right\rceil^{h-1}$	$2\left\lceil\frac{P}{2}\right\rceil^{h}$	$2\left\lceil\frac{P}{2}\right\rceil^{h-1}\left(\left\lceil\frac{P}{2}\right\rceil-1\right)$	P^h	P^{h+1}	$P^h(P-1)$

Minimum number of keys $= 1+2\left(\left\lceil\frac{P}{2}\right\rceil-1\right)+2\left\lceil\frac{P}{2}\right\rceil\left(\left\lceil\frac{P}{2}\right\rceil-1\right)+2\left\lceil\frac{P}{2}\right\rceil^2\left(\left\lceil\frac{P}{2}\right\rceil-1\right)+\ldots 2\left\lceil\frac{P}{2}\right\rceil^{h-1}\left(\left\lceil\frac{P}{2}\right\rceil-1\right)$

$$= 1+2\left(\left\lceil\frac{P}{2}\right\rceil-1\right)\left[1+\left\lceil\frac{P}{2}\right\rceil+\left\lceil\frac{P}{2}\right\rceil^2+\ldots\left\lceil\frac{P}{2}\right\rceil^{h-1}\right]$$

$$= 1+2\left(\left\lceil\frac{P}{2}\right\rceil-1\right)\left(\frac{\left\lceil\frac{P}{2}\right\rceil^h-1}{\left\lceil\frac{P}{2}\right\rceil-1}\right) = 1+2\left\lceil\frac{P}{2}\right\rceil^h-2 = 2\left\lceil\frac{P}{2}\right\rceil^h-1$$

Example-7.3 Find the height of B tree with order P and n keys when (a) Each node have minimum node (b) Each node have maximum node.

Solution:

(a) Each node have minimum node

$$n = 2\left\lceil\frac{P}{2}\right\rceil^h-1$$

$$\left\lceil \frac{P}{2} \right\rceil^h = \frac{n+1}{2}$$

$$h = \log_{\left\lceil \frac{P}{2} \right\rceil}\left(\frac{n+1}{2}\right)$$

(b) Each node has maximum node

$$n = P^{h+1} - 1$$

$$P^{h+1} = n + 1$$

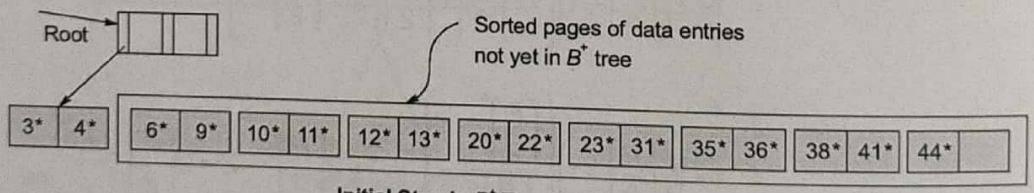
$$h = \log_P(n+1) - 1$$

7.5 Bulk Loading in B+ Tree

Entries are added to a B^+ tree in two ways. First, we may have an existing collection of data records with a B^+ tree index on it; whenever a record is added to the collection, a corresponding entry must be added to the B^+ tree as well. (Of course, a similar comment applies to deletions.) Second, we may have a collection of data records for which we want to create a B^+ tree index on some key field(s). In this situation, we can start with an empty tree and insert an entry for each data record, one at a time, using the standard insertion algorithm. However, this approach is likely to be quite expensive because each entry requires us to start from the root and go down to the appropriate leaf page. Even though the index-level pages are likely to stay in the buffer pool between successive requests, the overhead is still considerable.

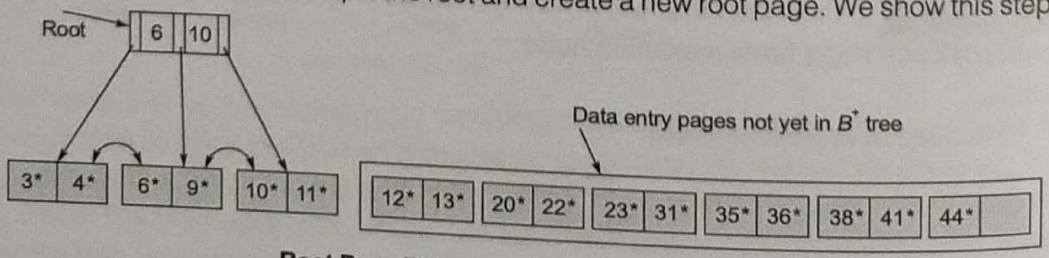
For this reason many systems provide a *bulk-loading* utility for creating a B^+ tree index on an existing collection of data records. The first step is to sort the data entries k^* to be inserted into the (to be created) B^+ tree according to the search key k. (If the entries are key-pointer pairs, sorting them does not mean sorting the data records that are pointed to, of course.) We use a running example to illustrate the bulk-loading algorithm. We assume that each data page can hold only two entries, and that each index page can hold two entries and an additional pointer (i.e., the B+ tree is assumed to be of order $d = 1$).

After the data entries have been sorted, we allocate an empty page to serve as the root and insert a pointer to the first page of (sorted) entries into it. We illustrate this process in Figure, using a sample set of nine sorted pages of data entries.

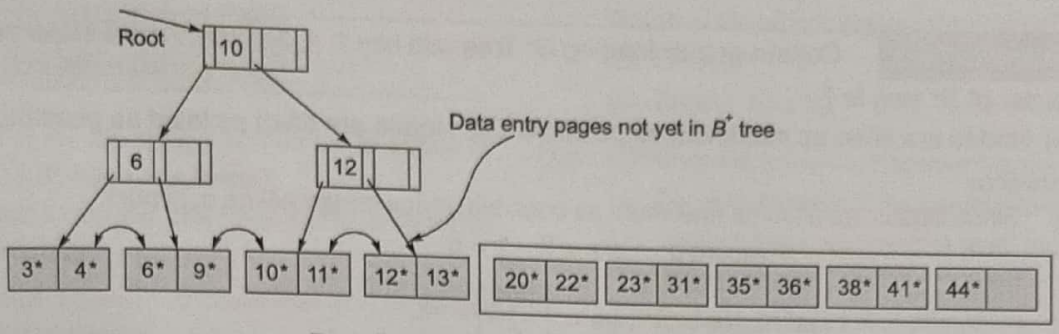


Initial Step in B^+ Tree Bulk-Loading

We then add one entry to the root page for each page of the sorted data entries. The new entry consists of Now key value on page, pointer' to page). We proceed until the root page is full; see Figure. To insert the entry for the next page of data entries, we must split the root and create a new root page. We show this step in Figure.



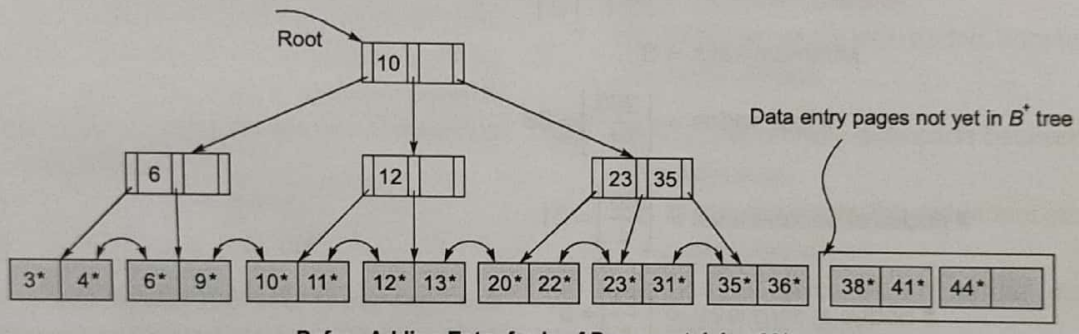
Root Page Fills up in B^+ Tree Bulk-Loading



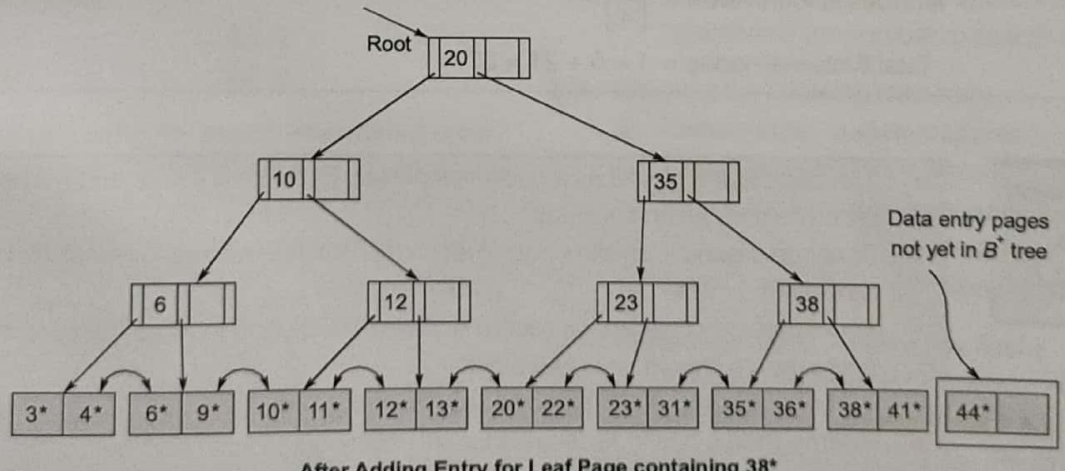
Page Split During B^+ Tree Bulk-Loading

"We have redistributed the entries evenly between the two children of the root, in anticipation of the fact that the B^+ tree is likely to grow. Although it is difficult (!) to illustrate these options when at most two entries fit on a page, we could also have just left all the entries on the old page or filled up some desired fraction of that page (say, 80 percent). These alternatives are simple variants of the basic idea.

To continue with the bulk-loading example, entries for the leaf pages are always inserted into the right-most index page just above the leaf level. 'When the right most index page above the leaf level fills up, it is split. This action may cause a split of the right-most index page one step closer to the root, as illustrated in Figures.



Before Adding Entry for Leaf Page containing 38*



After Adding Entry for Leaf Page containing 38*

Note that splits occur only on the right-most path from the root to the leaf level.

Example-7.4 Construct bulk loading B^+ Tree with key 1, 2, 3... 260. Find number of internal nodes if order of B^+ tree is 7.

(a) Nodes are filled as maximum as possible (b) Nodes are filled as least as possible

Solution:

(a) Since nodes are filled as maximum as possible, index nodes will be minimum.

$$\#\text{ record pointers} = 7 - 1 = 6$$

$$\#\text{ leaf nodes in } B^+ \text{ tree} = \left\lceil \frac{260}{6} \right\rceil = 44$$

$$\#\text{ internal nodes at the second level} = \left\lceil \frac{44}{7} \right\rceil = 7$$

$$\#\text{ internal nodes at the third level} = \left\lceil \frac{7}{7} \right\rceil = 1$$

$$\text{Total \# of internal nodes} = 7 + 1 = 8$$

(b) Since nodes are filled as least as possible, index nodes will be maximum.

$$\text{Minimum pointers} = \left\lceil \frac{P}{2} \right\rceil = \left\lceil \frac{7}{2} \right\rceil = 4$$

$$\text{Minimum keys} = 3$$

$$\#\text{ leaf nodes} = \left\lfloor \frac{260}{3} \right\rfloor = 86$$

$$\#\text{ nodes at second level} = \left\lfloor \frac{86}{4} \right\rfloor = 21$$

$$\#\text{ nodes at third level} = \left\lfloor \frac{21}{4} \right\rfloor = 5$$

$$\#\text{ nodes at fourth level} = \left\lfloor \frac{5}{4} \right\rfloor = 1$$

$$\text{Total \# internal nodes} = 1 + 5 + 21 = 27$$

Summary

- The database is stored as a collection of files. Each file is a sequence of records. A record is a sequence of fields.
- Spanned organization allow part of record to be in one block and rest of it to be on the next block.
- In unspanned organization record not allowed to span in more than one block i.e. entire record should be in one block.
- Types of indexing: Primary index and Secondary index.
- **Sparse Index:** has index entries for only some of the search values.
- **Dense Index:** has index entries for every search key value (and hence every record) in the data file.
- Types of multilevel Index (*i*) Static multilevel index and (*ii*) Dynamic multilevel index.

Student's Assignments 1

Q.1 Which of the following statement true above B tree and B+ tree node index?
[Assume order of B tree node same as order of B+ tree node]
(a) B tree index more levels than B+ tree index for large number of keys.
(b) B+ tree index more levels than B tree index for large number and keys.
(c) Both B tree B+ tree best for sequential access of records.
(d) B tree index nodes more than B+ tree for large number of keys.

Q.2 Suppose a phone book contain 500 pages and each page can contain upto 500 records. Suppose we want to search for a particular name in a phone book. Give a worst case bound on number of pages that must be looked to perform a search using an index for the name of the first entry of each page.
(a) 1 (b) 2
(c) 9 (d) 500

Q.3 Given a data file with 100 records per page and 1000 pages and on index page capacity of 512 index entries, how deep should be the B^+ tree to index this file.
(a) 1 (b) 2
(c) 3 (d) 4

Q.4 In a database file, the search key field is 9 bytes long the block size is 512 bytes, a record pointer is 6 bytes and block pointer is 7 bytes. The largest possible order of a non leaf node in B^+ tree implementing this file structure {order defines maximum number of keys present}
(a) 23 (b) 31
(c) 32 (d) 42

Q.5 Consider the following statement
(*i*) Second level index in multi level indexing will be always a primary index.
(*ii*) All indexed files will be always ordered files.
Which of the above statement is false?
(a) Only (*i*) (b) Only (*ii*)
(c) Both (*i*) and (*ii*) (d) None of these

Q.6 Which of the following statement is/are true?
(a) For any data file it is possible to construct two separate sparse first level indexes on different keys.
(b) For any data file, it is possible to construct two separate dense first level indexes on different keys.
(c) For any data file, it is possible to construct a sparse first level index and a dense second level index both should be useful.
(d) All the above

Q.7 *B* tree allows
(a) Sequential access only
(b) Random access only
(c) Both sequential and random access
(d) None of these

Q.8 What is the primary distinction between *B* and B^+ tree indices.
(a) *B* tree eliminates the redundant storage of search key values
(b) B^+ tree eliminates the redundant storage of search key values
(c) deletion in a *B* tree is more complicated
(d) Look-up in *B* tree is proportional to the logarithm of the number of search key

Q.9 Consider the following statement
(*i*) Primary index is always sparse
(*ii*) Secondary index may or may not be dense
Which of the above statement is/are true
(a) Only (*i*) (b) Only (*ii*)
(c) Both (*i*) and (*ii*) (d) None of these

Q.10 Consider a Table *T* with a key field k. *A* and *B* tree of order *P* denotes the maximum number of record pointers in a *B* tree. Assume *K* is 10 bytes long disk block size is 512 Bytes. Record pointer is 8 Bytes and block pointer size is 5 Bytes long in order for each *P* tree node to fit in a single disk block the maximum value of *P*.

(a) 20 (b) 22
(c) 23 (d) 32

Q.11 Consider a sorted file ordered on a key field consisting of 1024 blocks. The number of block access required to insert a record is _____.

Q.12 Match List-I (Type of Memory) with List-II (Used as) and select the correct answer using the code given below the lists:

List-I

A. Primary index
B. Clustering index
C. Secondary index (key)
D. Secondary index (non-key)

List-II

1. Non-key and non-ordered
2. key and non-ordered
3. Key and ordering
4. Non-key and ordered

Code:

	A	B	C	D
(a)	4	1	2	3
(b)	3	2	4	1
(c)	4	3	1	2
(d)	3	4	2	1

Q.13 The number of secondary indices that can be built on a file with 4 attributes is _____.

Q.14 Suppose size of disk block 1000 bytes and search key of 12 bytes, pointer size 8 bytes. How many minimum number of records in data file which loads 3 level dense B^+ tree index?

(a) 1200 (b) 1250
(c) 15000 (d) 15625

Answer Key:

1. (b) **2.** (b) **3.** (b) **4.** (b) **5.** (c)
6. (b) **7.** (b) **8.** (a) **9.** (b) **10.** (b)
11. (10) **12.** (d) **13.** (3) **14.** (a)

Student's Assignments Explanations

2. (b)

If the book were organized as a heap file in a linear search so worst case bound is 500. By using binary search the worst case bound is $\lceil \log_2 500 \rceil = 9$ using the fact that the book is ordered by names with an index for the name of the first entry on each page we get worst case bound is 2 because the index has 500 pages, so entire index files in one page 1 access to the index and 1 to the data.

3. (b)

One level B^+ tree node will have 512 index entries which could only refer to 512 pages. A 2 level B^+ tree have 512×512 index entries at the leaf level. There are 262, 144 entire. This is enough to address the 1000 pages of file. More over sufficient to address each record individually. Since there are only 100000 records.

4. (b)

Internal node in B^+ contains tree pointer and search key.

$$(P+1)\cdot 7 + P\cdot 9 \le 512$$
$$7P + 7 + 9P \le 512$$
$$16P \le 505$$
$$P \le 31.5$$
$$\text{order}(P) = 31$$

5. (c)

Both statements are false.

7. (b)

B tree allows only Random access while B^+ tree allows both sequential and random access.

8. (a)

B tree eliminates the redundant storage of search key because in B tree every node have similar structure. Non leaf node also contain record pointer.

9. (b)

Secondary index may or may not be dense

10. (b)

$$(10+8)P+(P+1)\cdot 5 \le 512$$
$$18P+5P+5 \le 512$$
$$23P \le 507$$
$$P \le 22.04$$
$$P = 22$$

11. (10)

The number of block accesses required to insert, search or delete a record in a sorted file of n blocks = $\log_2 n = \log_2 1024 = \log_2 2^{10} = 10$

12. (d)

Primary index is built over ordered key field. Clustering index is built over ordered non-key field. Secondary index (key) is built over non-ordered key field. Secondary index (non key) is built over non-ordered non-key field.

13. (3)

If there are n attributes then data file will be ordered based on only one attribute, remaining $(n-1)$ attributes are unordered. As per definition of secondary index, it is based on unordered attribute, so number of secondary indices that can be built on a file with 4 attributes is 3.

14. (a)

Order P:

$P * \text{pointer} + (P-1) * \text{Key} \le \text{Block}$

$P * 8 + (P-1)\,12 \le 1000$

$20P \le 1012$

$$P = \left\lfloor \frac{1012}{20} \right\rfloor = 50$$

Level	Minimum nodes	Minimum B_P	Minimum keys
1	1	2	1
2	2	2 * 25	24
3	50	–	50 * 24
			1200

Student's Assignments 2

Q.1 Consider a B tree of order P where order denotes maximum number of block pointer find minimum and maximum number of keys present of height h. (root node is present at height zero)

Q.2 A B^+ tree of order P is a tree in which each internal node has between P to 2 P keys. The root has between 1 to 2 P keys. What is the maximum number of internal nodes in B^+ tree?

Q.3 Consider a file consist of 30,000 fixed length record of size 100 bytes. Each disk block size is 1024 bytes and block pointer size is 6 bytes. A multilevel index on key field of length 9 bytes is constructed. Find out total number of blocks required to access a record by searching multilevel index?

Q.4 Suppose blocks hold either three records, or ten (key, pointer) pairs. How many blocks do we need to hold a data file and dense index to store n records?

Q.5 Consider a B tree of order P, where P denotes maximum number of tree pointers in a B tree node. Assume that key field is 10 bytes long, disk block size is 512 bytes, data pointer is 8 bytes long.

Find the maximum value of P in order to fit B tree in a single disk block.

Q.6 Consider a file of 16384 records. Each record is 32 byte long and key fields of size 6 bytes. The file is ordered on a non-key field and index is build on nonkey an average 16 records per cluster.

The block size is 1024 bytes and the size of block pointer is 10 bytes. If secondary index is built on the key field of the file, and multilevel index scheme is used to store the secondary index. Find the number of index blocks.

Q.7 Suppose values 8, 5, 1, 7, 3, 12, 9, 6 are inserted into an empty B tree of order 3 show the final structure of B tree.

Q.8 Find the number of different B tree with 3 keys, 4 pointer nodes, when data file has 6 records?