

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT I

Introduction: History – Purpose – View of Data – Database languages – Data Models – Data Storage and Querying – Transaction management – Database Architecture – Two tier – Three tier – Database users and Authorization.

Relational Algebra – Structure- keys – schema diagrams – Relational operations – Formal Relational Query Languages – Relational Algebra – Tuple Relational calculus – Domain Relational Calculus.

SQL Overview – Data Definition – basic structure – basic operations – Set Operations – Null Values – Aggregate Functions – Nested Subqueries – Modifications of the Databases – Join Expression – Integrity Constraints – Views – Authorization – Functions – Procedures – Triggers – Recursive Queries.

Unit I- Two Marks

1. What is DBMS?

DBMS is a collection of interrelated data and a set of programs to access that data. The goal is to provide the environment that is both convenient and efficient to use in retrieving and storing data base information.

2. What are the advantages of DBMS?

- Centralized Control
- Data Independence allows dynamic change and growth potential.
- Data Duplication is eliminated with control Redundancy.
- Data Quality is enhanced.
- Controlling redundancy
- Restricting unauthorized access
- Providing multiple user interfaces
- Enforcing integrity constraints.
- Providing back up and recovery

3. What are the Disadvantages of DBMS?

- Cost of software, hardware and migration is high.
- Complexity of Backup
- Problem associated with centralization.

4. List out the applications of DBMS.

- Banking
- Airlines
- Universities
- Credit card transactions
- Tele communication
- Finance
- Sales
- Manufacturing
- Human resources

5. What are the disadvantages of File Systems?

- Data Redundancy and Inconsistency.
- Difficulty in accessing data.
- Data Isolation
- Integrity problems
- Security Problems
- Concurrent access anomalies

6. Give the levels of data abstraction?

- a. Physical level
- b. Logical level
- c. View level

7. Define the terms

- a. Physical schema
- b. Logical schema.

Physical schema: The physical schema describes the database design at the physical level, which is the lowest level of abstraction describing how the data are actually stored.

Logical schema: The logical schema describes the database design at the logical level, which describes what data are stored in the database and what relationship exists among the data.

8. What is conceptual schema?

The schemas at the view level are called subschemas that describe different views of the database.

9. Define data model?

A data model is a collection of conceptual tools for describing data, data relationships, data semantics and consistency constraints.

10.What is storage manager?

A storage manager is a program module that provides the interface between the low level data stored in a database and the application programs and queries submitted to the system.

11.What are the components of storage manager?

The storage manager components include

- a) Authorization and integrity manager
- b) Transaction manager
- c) File manager
- d) Buffer manager

12.What is the purpose of storage manager?

The storage manager is responsible for the following

- a) Interaction with the file manager
- b) Translation of DML commands in to low level file system commands
- c) Storing, retrieving and updating data in the database

13.List the data structures implemented by the storage manager

The storage manager implements the following data structure

- a) Data files
- b) Data dictionary
- c) Indices

14.What is a data dictionary?

A data dictionary is a data structure which stores meta data about the structure of the database ie. the schema of the database.

15.Explain Referential Integrity.

Referential Integrity means relationship between tables. Foreign keys are used. Foreign key is the column whose values are derived from the Primary key of the same or some other table.

Format for creating a foreign key is given below.

Syntax:

```
create table <table name>(columnname data type (size) constraint
constraint_name references parent table name);
```

16. Define Instances and schemas.

Instances:

The collection of information stored in the database at a particular moment is called the instance of the database.

Schemas:

The overall design of the database is called the schema of the database.

17. Define and explain the two types of Data Independence.

Two types of data independence are

Physical data independence:

The ability to modify the physical schema without causing application programs to be rewritten in the higher levels. Modifications in physical level occur occasionally whenever there is a need to improve performance.

Logical data independence:

The ability to modify the logical schema without causing application programs to be rewritten in the higher level (external level). Modifications in physical level occur frequently more than that in physical level, whenever there is an alteration in the logical structure of the database.

18. Define transaction.

A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Properties of transaction are atomicity, consistency, isolation, and durability.

19. Define the type types of DML.

Two types:

Procedural DML:

It requires a user to specify what data are needed and how to get those data.

Non-Procedural DML:

It requires a user to specify what data are needed without specifying how to get those data.

20. List out the functions of DBA.

- Schema definition
- Storage structure and access-method definition
- Schema and physical modification
- Granting of authorization for data access
- Integrity constraint specification

21. What is the need for DBA?

The need of DBA is to have central control of both the data and the programs that access those data. The person who has such central control over the system is called the DataBase Administrator. [

22. Define weak and strong entity sets?

Weak entity set

Entity set that do not have key attribute of their own are called weak entity sets.

Strong entity set

Entity set that has a primary key is termed a strong entity set.

23. What does the cardinality ratio specify?

Mapping cardinalities or cardinality ratios express the number of entities to which another entity can be associated. Mapping cardinalities must be one of the following:

- One to one
- One to many
- Many to one
- Many to many

24. Explain the two types of participation constraint.

Total

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.

Partial

If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be partial.

25. Explain DML pre-compiler.

DML pre-compiler converts DML statements embedded in an application program to normal procedure calls in the host language. The pre-compiler must interact with the DML compiler to generate the appropriate code.

26. Define file manager and buffer manager.

File manager:

File manager manages the allocation of space on disk storage and the data structures used to represent information stored on disk.

Buffer manager:

Buffer manager is responsible for fetching data from the disk storage into the main memory, and deciding what data to cache in memory.

27. Define Data Dictionary.

DDL statements are compiled into a set of tables that is stored in a special file called data dictionary. Data dictionary contains the meta-data, which in turn is data about the data.

28. What is a query language?

The query language is the language in which a user request information from the database. These languages are usually on a level higher than that of a standard programming language.

29. What in procedural and non procedural languages?

In procedural languages, the user instructs the system to perform a sequence of operation on the database to compute the desired result. In nonprocedural language, the user describes the desired information without giving a specific procedure for obtaining that information.

30. What in Cartesian –product operation?

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

31. Define domain?

For each attribute, there is a set of permitted values called the domain. For the attribute branch name, for example, the domain is the set of all branches name.

32. Define relation?

Relation to be a subset of a Cartesian product of a list of domains.

33. Define tuples?

A tuple variable is a variable that stands for a tuple, in other words a tuple variable is a variable whose domain is the set of all tuples.

34. What is the relation schema?

The concept of a relation schema corresponds to the programming –language notion of type definition. We use Account –schema to denote the relation schema for relation account. *Account-schema = (account-number, branch-name, balance)*. We denote the fact that account is a relation on Account –schema by *Account (Account-schema)*.

35. What is database schema?

Database schema in the logical design of the database.

36. What is database instance?

Database instance is a snapshot of the data in the database at a given instant in time.

37.What is relation instance?

The concept of a relation instance corresponds to the programming language notion of a value of a variable.

38.What is relational algebra?

The relational algebra is a procedural query language .It consists of set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operation in the relational algebra is select, project, union, set difference, Cartesian product and rename.

39.What is select operation?

The select operation selects tuples that satisfy a given predicate .we use the lower case Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ .The argument relation in parentheses after the σ

40.Describe extended relational operation?

The basic relation –algebra operation have been extended in several ways. The simple extension is to allow arithmetic operations as part of projection. An important extension is to allow aggregate operations such as computing the sum of the elements of a set or their average.

41.What is meant by aggregate function?

Aggregate function takes a collection of values and return a single value as a result.

42.What is meant by multiset?

The collections on which aggregate functions operates can have multiple Occurrences of value; the order in which the value appear in not relevant .such collections are called as multisets.

43.What is outer join?

Outer join operation is an extension of the join operation to deal with missing information. Suppose that we have the relations with the following schemas ,which contains data on full-time table name.

44.What are the types of outer join?

- left outer join
- right outer join
- full outer join

45.What is meant by null values?

The special values are called as Null values indicates “value unknown or non-existent”, any arithmetic operation such as (+,-,*, /) involving null values must return a null result.

46. In SQL operation how the different relational operations deal with null values?

- select
- join
- projection
- union, intersection, difference
- generalized projection
- aggregate
- outer join

47. Define modification of database?

It contains following operations like

- deletion
- insertion
- update

48. Define view?

We define a view by using the create view statement . To define a view, we must give the view a name ,and must state the query the computes the view. The form of create view statement is Create view v as <query expression>.

49. What is meant by tuple relation calculus?

The tuple relation calculus, by contrast is a, non procedural query language .It describes the desired information without giving the specific procedure for obtaining that information. A query in the tuple relational calculus is expressed as $\{t|P(t)\}$

50. What is meant by domain relation calculus?

A second form of relational calculus are called domain relational calculus.

51. What are the parts of SQL languages?

There are several parts in SQL language

- Data-definition language(DDL)
- Interactive data manipulation language(DML)
- View definition
- Transaction control
- Embedded SQL and dynamic SQL
- Integrity
- Authorization

52. Explain the basic structure SQL?

The basic structure of an SQL expression consist of three clauses;

Select from where

53. Define select clause?

The **select** clause corresponds to the projection operation of the relational algebra. It is used to list the attribute desired in the result of query.

54. Define from clause?

The **from** clause corresponds to Cartesian –product operation of the relational algebra. It lists the relations, to be scanned in the evaluation of the expression.

55. Define where clause?

The where clause corresponds to selection predicate the relational algebra.

56. Write query for rename operation?

The query is **old- name as new-name**. eg:

```
select customer- name ,borrower .loan- number ,amount
from borrower, Loan
where borrower .loan -number =loan. Loan- number.
```

The result of this query is a relation with following attributes:

Customer –name ,loan-number, amount.

57. Define tuple variable?

The **as** clause is particularly useful in defining the notion of tuple variables, as is done in the tuple relational calculus. A tuple variable in SQL must be associated with a particular relation. Tuple variable are defined in the **from** clause by way of the **as** clause.

58. Define ordering the display of tuples?

The **order by** clause causes the tuples in the result of query to appear in sorted order . By default, the **order by** clause list item in ascending order to specify the sorted order, we may specify **desc** for descending order asc for ascending order.

59. What are the set operations available in SQL?

The SQL operations are

- union operation(\cup)
- intersection operation(\cap)
- except operation($-$)

60. What is union operation?

To find all the customer having a loan, and account , or both at bank, we write query as follow(**Select** customer- name **from** depositor)

Union

(**select** customer name **from** borrower)

The union operation automatically eliminates duplicates. If we want to retain all duplicates we must write **union all** in the place of **union**.

61.Describe intersection operation?

To find all the customer having both a loan, and account at a bank.

*(Select **distinct** customer- name **from** depositor)*

intersection

*(select **distinct** customer name **from** borrower)*

The **intersection** operation automatically eliminates duplicates . If we want to retain all duplicates we must write **intersection all** in the place of **intersection**.

62.Define aggregate function?

Aggregate functions are the functions **that** take a collection (a set or multiset) of values as input and return a single value.

63.What are the 5 built-in aggregate functions in SQL?

- Average : **avg**
- Minimum : **min**
- Maximum :**max**
- Total : **sum**
- Count :**count**

64.Define group by clause?

In SQL using **group by** clauses, the attribute or attributes given in the **group by** clauses are used to form groups. Tuple with same values on all attributes in **group by** clause are placed in one group .we write query as: **Select branch name, avg (balance) from account group by branch name**

65.Define nested subqueries?

SQL provides a mechanism for nesting subqueries. A subquery is a select- from-where expression that is nested within another query.

66.Define with clause?

The with clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.

67.What is meant Transaction?

A transaction consists of a sequence of query and / or update statements. The SQL standard specifies that the transaction begins implicitly when an SQL statement is executed.

68.What are the types of transaction available in SQL?

Commit work: commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database . After the transaction is committed , a new transaction is automatically started.

Roll back work : It causes the current transaction to be rolled back ; that is ,it undoes all the updates performed by SQL statements in the transaction

69.Domain types in SQL?

The SQL standard support variety of built in domain types including

- Char (n)
- Varchar (n)
- Int
- Smallint
- numeric(p,d)
- real, double precision
- float(n)
- date
- time
- timestamp-(combination of date and time)

70.What is meant by check constraint?

Check (P): The check clause specify the predicate P that must be satisfied by the every tuple in the relation Create table account

```
(account-name char(10),  
branch-name char(10),  
balance integer,  
primary key(account-number),  
check (balance>=0))
```

71.Define ODBC?

The open database connectivity (ODBC) standard defines a way for a application program to communicate with a database server. ODBC defines an application program interface (**API**) that application can use to open a connection with database send queries and update and get back results.

72.What is meant JDBC?

JDBC standard defines an API the java program can use to connect to database server. JDBC-java database connectivity

73.Describe Query- by- Example(QBE)?

The QBE is the name of both a data manipulation language and an early database system that includes this language .

74.What is meant by condition box?

The condition box that allows the expression of general constrain over any of the domain variables. QBE allows logical expression to appear in a condition box . the logical operation are the words **AND** and **OR**, or the symbols "&"and"|".

75.What is meant by datalog?

The datalog is a non procedural query language based on the logic-programming prolog. As in the relation calculus a user describes the information decide without giving a specific procedure for obtaining that information. The syntax of datalog resembles that a prolog.

76.Describe user inter face and tools?

Although many people interact with database, few people use query language to directly interact with the database system. Most of the people interact with the database system through one of this following means:

- * Forms and GUI
- * Report generator
- Data analysis tools

77.Define forms and GUI?

Allows user to enter the values that complete predefined queries. The GUI (Graphical User Interface)provides an easy to use way to interact with the database System.

78.What is relational model?

The relational model used the basic concept of a relation or table. The columns or fields in the table identify the attributes such as name, age, and so. A tuple or row contains all the data of a single instance of the table such as a person named Doug. In the relational model, every tuple must have a unique identification or key based on the data.

79.Name any three RDBMS packages

Oracle (48.8%), IBM (20.2%), Microsoft (17.0%), SAP including Sybase (4.6%), and Teradata (3.7%).

80.What is integrity constraint?

Integrity constraint ensure that changes made to the database by authorized users do not result in a loss of data consistency.

81.What is domain constraint?

Domain constraint are the most elementary form of integrity constraint they are tested easily by the system whenever a new data item is entered into database

82.What is referential integrity?

To ensure a a value that appears in one relation for a given set of attributes also appears for certain set of attributes in another relation.

83.What is database modification?

- insert
- delete
- update

84.What is referential integrity in SQL?

Foreign key can be specified as a part of the sql 'create table' statement by using the foreign key clause. A foreign key references the primary key attributes of the referenced table.

Example:

Create table stud (rno number (2),status varchar(5), foreign key(no) references student (regno));

85.What is assertions?

An assertion is a predicate expressing a condition that we wish the database always to satisfy.

An assertion in sol takes the form,

Query: Create assertion<assertion-name>check<predicate>

86.What is a trigger?

A trigger is a statement that the system executes automatically as a side effect of a modification to the database. There are two requirements to design a trigger mechanisms

- specify when a trigger is to executed. This is broken up into an event that causes the trigger to be checked and a condition that must be satisfied for trigger execution to proceed.
- specify the actions to be taken when the trigger executes.

87.Why triggers are needed?

Trigger are useful mechanisms for altering humans or for staring certain task automatically when certain conditions are met.

88.When not to use to trigger?

Triggers should not be used, when an insert trigger on a relation has an action that causes another insert action to be stimulated.

- Long chain of triggers should not be used, because it considers them as error.

89.What is security?

Data stored in the database need protection from unauthorized access and malicious destruction.

90.What is security violation?

- Unauthorized reading of data.
- Unauthorized modification of data
- Unauthorized destruction of data

91.What are levels of security?

- Database system
- Operating system

- Network
- Physical
- Human

92.What are the several form of authorization?

- Read authorization-allows reading, but not modification of data
- Insert authorization- allows insertion of new data but not modification of existing data
- Update authorization- allows modification but not deletion of data.
- Delete authorization-allows deletion
- Index authorization-allows the creation of new relations
- Resource authorization- allows the creation of new relations
- Alteration authorization- allows the addition or deletion of attributes in a relation
- Drop authorization-allows the deletion of relations

93.What is authorization graph?

The passing of authorization from one user to another can be represented by an authorization graph.

94.What is notion of roles?

Roles include teller, branch manager, system administrator. The notion of roles capture the schema. The set of roles is created in the database. Authorization can be granted to roles. Each database user is granted a set of roles.

95.What is an audit trail? (APRIL 2011)

An audit trail is a log of changes(insert, delete, update) to the database along with information such as which user performed the change and when the change was performed.

96.What is encryption?

Encryption forms the basis of good schemes for authenticating users to a database. The various provisions that a database may make for authorization may still not provide sufficient protection for high sensitive data.. in such cases data may be stored in encrypted form.

97.What are the encryption techniques?

A good encryption technique has the following techniques

- It is relatively simple for authorization users to encrypt and decrypt data.
- It depends not on the secrecy of the algorithm, but rather on a parameter of the algorithm called the encryption key.
- Its encryption key is extremely difficult for an intruder to determine.

98.What is authentication?

It refers to the task of verifying the identity of the person./software connecting to the database.

99. What is authorization?

The ultimate aim of authorization is to give authority to the database administrator. The database administrator may authorize new users, restructure the database using read, insert, update, delete, index, resource, alter, drop authorization.

100. What are the forms of assertion?

- Domain constraint
- Referential integrity constraint

101. What are the parts of SQL language?

The SQL language has several parts:

- data – definition language
- Data manipulation language
- View definition
- Transaction control
- Embedded SQL
- Integrity
- Authorization

102. What are the categories of SQL command?

SQL commands are divided in to the following categories:

1. data - definition language
2. data manipulation language
3. Data Query language
4. data control language
5. data administration statements
6. transaction control statements

103. What are the three classes of SQL expression?

SQL expression consists of three clauses:

Select

From

where

104. Give the general form of SQL query?

Select A , A , An

1 2

Fro m R , R , R

1 2 m

Where P

105. What is the use of rename operation?

Rename operation is used to rename both relations and attributes.

It uses the as clause, taking the form:

Old-name as new-name

106. Define tuple variable?

Tuple variables are used for comparing two tuples in the same relation. The tuple variables are defined in the from clause by way of the as clause.

107. List the string operations supported by SQL?

- 1) Pattern matching Operation
- 2) Concatenation
- 3) Extracting character strings
- 4) Converting between uppercase and lower case letters.

108. List the set operations of SQL?

- 1) Union
- 2) Intersect operation
- 3) The except operation

109. What is the use of Union and intersection operation?

Union : The result of this operation includes all tuples that are either in r1 or in r2 or in both r1 and r2. Duplicate tuples are automatically eliminated.

Intersection: The result of this relation includes all tuples that are in both r1 and r2.

110. What are aggregate functions? And list the aggregate functions supported by SQL?

Aggregate functions are functions that take a collection of values as input and return a single value. Aggregate functions supported by SQL are

Average: avg

Minimum: min

Maximum: max

Total: sum

111. What is the use of group by clause?

Group by clause is used to apply aggregate functions to a set of tuples. The attributes given in the group by clause are used to form groups. Tuples with the same value on all attributes in the group by clause are placed in one group.

112. What is the use of sub queries?

A sub query is a select-from-where expression that is nested with in another query. A common use of sub queries is to perform tests for set membership, make set comparisons, and determine set cardinality.

113. What is view in SQL? How is it defined? NOV 2014

Any relation that is not part of the logical model, but is made visible to a user as a virtual relation is called a view. We define view in SQL by using the create view command. The form of the create view command is

Create view v as <query expression>

114. What is the use of with clause in SQL?

The with clause provides a way of defining a temporary view whose definition is available only to the query in which the with clause occurs.

115. List the table modification commands in SQL?

- Deletion
- Insertion
- Updates
- Update of a view

116. List out the statements associated with a database transaction?

- Commit work
- Rollback work

117. What is transaction?

Transaction is a unit of program execution that accesses and possibly updated various data items.

118. List the SQL domain Types?

SQL supports the following domain types.

- 1) Char(n) 2) varchar(n) 3) int 4) numeric(p,d)
- 5) float(n) 6) date.

119. What is the use of integrity constraints?

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Thus integrity constraints guard against accidental damage to the database.

120. Mention the 2 forms of integrity constraints in ER model?

Key declarations

Form of a relationship

121. What is trigger? NOV 2010

Triggers are statements that are executed automatically by the system as the side effect of a modification to the database.

122. What are domain constraints?

A domain is a set of values that may be assigned to an attribute .all values that appear in a column of a relation must be taken from the same domain.

123. What are referential integrity constraints?

A value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

124. What is assertion? Mention the forms available.

- An assertion is a predicate expressing a condition that we wish the database always to satisfy.
- Domain integrity constraints.
- Referential integrity constraints

125. Give the syntax of assertion?

Create assertion <assertion name> check <predicate>

126. What is the need for triggers?

Triggers are useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met.

127. List the requirements needed to design a trigger.

- The requirements are
- Specifying when a trigger is to be executed.
- Specify the actions to be taken when the trigger executes.

128. Give the forms of triggers?

- The triggering event can be insert or delete.
- For updated the trigger can specify columns.
- The referencing old row as clause
- The referencing new row as clause
- The triggers can be initiated before the event or after the event.

129. What does database security refer to?

Database security refers to the protection from unauthorized access and malicious destruction or alteration.

130. List some security violations (or) name any forms of malicious access.

- Unauthorized reading of data
- Unauthorized modification of data
- Unauthorized destruction of data.

131. List the types of authorization.

- Read authorization
- Write authorization
- Update authorization
- Drop authorization

132. What is authorization graph?

Passing of authorization from one user to another can be represented by an authorization graph.

133. List out various user authorization to modify the database schema.

- Index authorization
- Resource authorization
- Alteration authorization
- Drop authorization

134. What are audit trails?

An audit trail is a log of all changes to the database along with information such as which user performed the change and when the change was performed

135. Mention the various levels in security measures.

- Database system
- Operating system
- Network
- Physical
- human

136. Name the various privileges in SQL?

- Delete
- Select
- Insert
- update

137. Mention the various user privileges.

- All privileges directly granted to the user or role.
- All privileges granted to roles that have been granted to the user or role.

138. Give the limitations of SQL authorization.

- The code for checking authorization becomes intermixed with the rest of the application code.
- Implementing authorization through application code rather than specifying it declaratively in SQL makes it hard to ensure the absence of loopholes.

139. Give some encryption techniques?

- DES
- AES
- Public key encryption

140. What does authentication refer?

Authentication refers to the task of verifying the identity of a person.

141. List some authentication techniques.

- Challenge response scheme
- Digital signatures
- Nonrepudiation

11 MARK QUESTION AND ANSWERS

1. Explain in detail about Database management systems(6 Marks)

Database Management System (DBMS) is a Collection of interrelated data s Set of programs to access the data s DBMS contains information about a particular enterprise s DBMS provides an environment that is both convenient and efficient to use.

- Database Applications:
- Banking: all transactions
- Airlines: reservations
- Universities: registration, grades
- Sales: customers, products, purchases
- Manufacturing: production, inventory, orders, supply chain
- Human resources: employee records, salaries, tax deductions

Database applications were built on top of file systems. Drawbacks of using file systems to store data:

- Data redundancy and inconsistency Multiple file formats, duplication of information in different files.
- Difficulty in accessing data Need to write a new program to carry out each new task
- Data isolation — multiple files and formats
- Integrity problems Integrity constraints (e.g. account balance > 0) become part of program code Hard to add new constraints or change existing ones Database System Concepts
- Atomicity of updates Failures may leave database in an inconsistent state with partial updates carried out E.g. transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users Concurrent accessed needed for performance Uncontrolled concurrent accesses can lead to inconsistencies – E.g. two people reading a balance and updating it at the same time
- Security problems s Database systems offer solutions to all the above problems

2. Explain in detail about view of data and levels of Abstraction(11 Marks)

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained.

Data Abstraction:

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

Levels of Abstractions:

- **Physical level** describes how a record (e.g., customer) is stored.
- **Logical level:** describes data stored in database, and the relationships among the data. The Logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.
- **View level.** The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database. Figure 1.1 shows the relationship among the three levels of abstraction.

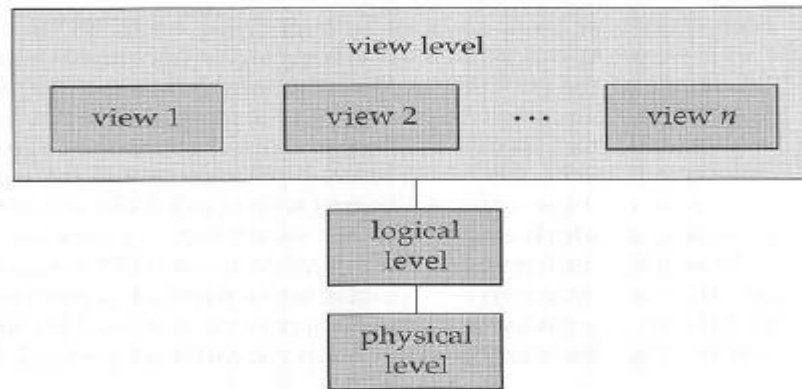


Fig 1.1 Three levels of abstraction.

- For example, in a Pascal-like language/ we may declare a record as follows:

type customer = record

name: string;

street: string;

city: string;

Street: string;

end;

- This code defines a new record type called customer with four fields. Each field has a name and a type associated with it. A banking enterprise may have several such record types, including
 - **Account with fields account_ number and balance**
 - **Employee with fields employee-name and salary**
- At the physical level, a customer, account, or employee record can be described as a block of consecutive storage locations (for example, words or bytes). The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest level storage details from database programmers. Database administrators, on the other hand, may be aware of certain details of the physical organization of the data.
- At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction.

- Finally, at the view level, computer users see a set of application programs that hide details of the data types. Similarly, at the view level several views of the database are defined, and database users see these views. In addition to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database. For example, tellers in a bank see only that part of the database that has information on customer accounts; they cannot access information about salaries of employees.

3. Explain in detail about Instances & Schemas(5 Marks)

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

The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an instance of a database schema.

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

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit physical data independence if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

4. Explain in detail about Data Models(6 Marks)

The data model is defined as a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, Logical, and view level. There are a number of different data models. The data models can be classified in four different categories:

1. Relational Model
2. The Entity-Relationship Model
3. Object-Based Data Model.
4. Semi Structured Data Model

Relational Model: The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

Entity-Relationship Model: The entity-relationship (E-R) data model is based on a perception of a real world that consists of a collection of basic objects called entities and of relationships among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design.

Object-Based Data Model: The object-oriented data model is another data model that has seen increasing attention. The object-oriented model can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and relational data model.

Semi structured Data Model: The semi structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The Extensible Markup Language (XML) is widely used to represent semi structured data.

5. Explain in detail about Database Languages and its types(11 Marks)

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

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

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

There are basically two types:

- Procedural DMLs require a user to specify what data are needed and how to get those data.
- Declarative DMLs (also referred to as nonprocedural DMLs) require a user to specify what data are needed without specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data. A query is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a query language. Although technically incorrect, it is common practice to use the terms query language and data manipulation language synonymously.

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

The data values stored in the database must satisfy certain consistency constraints. For example, suppose the balance on an account should not fall below RS. 100. The DDL provides

facilities to specify such constraints. The database systems check these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, database systems concentrate on integrity constraints that can be tested with minimal overhead.

Domain Constraints: A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

Referential Integrity: There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation (referential integrity). Database modifications can cause violations of referential integrity. When a referential integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

Assertions: An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, "Every loan has at least one customer who maintains an account with a minimum balance of Rs.1000.00 must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

Authorization: We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of authorization, the most common being, **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; update authorization, which allows modification, but not deletion, of data and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization

The DDL, just like any other programming language, gets as input some instructions (statements) and generates some output. The output of the DDL is placed in the data dictionary, which contains metadata-that is, data about data. The data dictionary is considered to be a special type of table, which can only be accessed and updated by the database system itself (not a regular user). A database system consults the data dictionary before reading or modifying actual data.

6. Describe the Structure of Relational Databases (This Questions consists of 4 Parts where each having the weightage of asking separately in 6 marks, 11 marks and 5 Marks)

A relational database consists of a collection of tables, each of which is assigned a unique name. A row in a table represents a relationship among a set of values. Informally, a table is an entity set, and a row is an entity. Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name. In what follows, we introduce the concept of relation.

- 1. Basic Structure of Relational Database(11 Marks)**
- 2. Database Schema & Schema Diagram of Relational Database(11 Marks)**
- 3. Keys of Relational Database(6 Marks)**
- 4. Query Languages(4 marks)**

(i).Basic Structure of Relational Database(11 Marks)

Consider the account table of Figure 1.2 shown below. It has three column headers: account_number, branch-name and balance.

<i>account_number</i>	<i>branch_name</i>	<i>balance</i>
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

1.2 Account Relation

For each attribute, there is a set of permitted values, called the domain of that attribute. For the attribute branch-name for example, the domain is the set of all branch names. Let D_1 denote the set of all account numbers, D_2 the set of all branch names, and D_3 the set of all balances. Any row of account must consist of a 3-tuple (u_1, u_2, u_3) where ' u_1 ' is an account number (that is, u_1 is in domain D_1), u_2 is a branch name (that is, u_2 is in domain D_2), and u_3 is a balance (that is, u_3 is in domain D_3). In general, account will contain only a subset of the set of all possible rows. Therefore, account is a subset of $D_1 \times D_2 \times D_3$.

In general, a table of n attributes must be a subset of $D_1 \times D_2 \times \dots \times D_{n-1} \times D_n$. Mathematicians define a relation to be a subset of a Cartesian product of a list of domains.

This definition corresponds almost exactly with our definition of table. The only difference is that we have assigned names to attributes, whereas mathematicians rely on numeric "names," using the integer 1 to denote the attribute whose domain appears first in the list of domains, 2 for the attribute whose domain appears second, and so on. Because tables are essentially relations, we shall use the mathematical terms relation and tuple in place of the terms table and row. A tuple variable is a variable that stands for a tuple; in other words, a tuple variable is a variable whose domain is the set of all tuples.

In the account relation of Figure 1.2. There are seven tuples. Let the tuple variable "t" refer to the first tuple of the relation. We use the notation t[account-number] to denote the value of "t" on the account-number attribute. Thus, t[account-number]= "A-101," and t[branch-name]= "Downtown". Alternatively we may write t[1] to denote the value of tuple t on the first attribute (account-number), t[2] to denote branch_name and so on. Since a relation is a set of tuples, we use the mathematical notation of $t \in r$ to denote that tuple t is in relation r.

The order in which tuples appear in a relation is irrelevant, since a relation is a set of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 1.2 shown above , or are unsorted, as in Figure 1.3 as shown below , does not matter; the relations in the two figures are the same, since both contain the same set of tuples. We require that, for all relations r, the domains of all attributes of r be atomic. A domain is atomic if elements of the domain are considered to be indivisible units. For example, the set of integers is an atomic domain, but the set of all sets of integers is a non atomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts-namely, the integers composing the set. The important issue is not what the domain itself is, but rather how we use domain elements in our database. The domain of all integers would be non atomic if we considered each integer to be an ordered list of digits. In all our examples, we shall assume atomic domains.

account_number	branch_name	balance
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

1.3 Account Relation with unordered tuples

It is possible for several attributes to have the same domain. For example, suppose that we have a relation customer that has the three attributes customer-name, customer -street, and customer city, and a relation employee that includes the attribute employee -name. It is possible that the attributes customer_name and employee name will have the same domain: the set of all person names, which at the physical level is the set of all character strings. The domains of balance and branch_name, on the other hand, certainly ought to be distinct. It is perhaps less clear whether customer_name and branch_name should have the same domain. At the physical level, both customer names and branch names are character strings. However, at the logical level, we may want customer_name and branch_name to have distinct domains.

One domain value that is a member of any possible domain is the null value, which signifies that the value is unknown or does not exist. For example, suppose that we include the attribute telephone-number in the customer relation. It may be that a customer does not have a telephone number, or that the telephone number is unlisted. We would then have to resort to null values to signify that the value is unknown or does not exist.

(ii). Database Schema & Schema Diagram of Relational Database (11 Marks)

The database schema, which is the logical design of the database, and the database instance, which is a snapshot of the data in the database at a given instant in time. The concept of a relation corresponds to the programming-language notion of a variable. The concept of a relation schema corresponds to the programming-language notion of type definition.

It is convenient to give a name to a relation schema, just as we give names to type definitions in programming languages. We adopt the convention of using lowercase names for relations, and names beginning with an uppercase letter for relation schemas. Following this notation, we use Account-schema to denote the relation schema for relation account. Thus,

“Account_schema = (account_number, branch_name, balance)”

We denote the fact that account is a relation on Account_schema by account (Account-schema).

In general, a relation schema consists of a list of attributes and their corresponding domains. The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time; similarly the contents of a relation instance may change with time as the relation is updated. However, we often simply say "relation" when we actually mean "relation instance."

As an example of a relation instance consider the branch relation of Figure 1.4. as shown below

<i>branch_name</i>	<i>branch_city</i>	<i>assets</i>
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Figure1.4. Branch Relation

The schema for that relation is

“Branch_schema=(branch_name, branch_city, assets)”

Note that the attribute branch-name appears in both Branch_schema and Account _schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations. For example, suppose we wish to find the information about all of the accounts maintained in branches located in Brooklyn. We look first at the branch relation to find the names of all the branches located in Brooklyn. Then, for each such branch, we look in the account relation to find the information about the accounts maintained at that branch.

Let us continue our banking example. We need a relation to describe information about customers. The relation schema is

“Customer_schema = (customer_name, customer_street, customer_city)”

<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Figure1.5 Customer Relation

The above Figure 1.5 shows a sample relation customer (Customer_schema). Note that we have omitted the customer-id attribute that we used in previous diagrams, because now we want to have smaller relation schemas in our running example of a bank database.

We assume that the customer name uniquely identifies a customer-obviously this may not be true in the real world, but the assumption makes our examples much easier to read. In a real-world database, the customer-id (which could be a social-security number or an identifier generated by the bank) would serve to uniquely identify customers. We also need a relation to describe the association between customers and accounts. The relation schema to describe this association is

“Depositor_schema = (customer_name, account_number)”

Fig 1.6 shows a sample relation depositor (Depositor-schema).

<i>customer_name</i>	<i>account_number</i>
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Figure1.6 Depositor Relation

It would appear that, for our banking example, we could have just one relation schema, rather than several. That is, it may be easier for a user to think in terms of one relation schema, rather than in terms of several. Suppose that we used only one relation for our example, with schema.

**“(Branch_name, branch_city, assets, customer_name, customer_street
Customer_city, account_number, balance)”**

Observe that, if a customer has several accounts, we must list her address once for each account. That is, we must repeat certain information several times. This repetition is wasteful and is avoided by the use of several relations, as in our example' In addition, if a branch has no accounts (a newly created branch, say, that has no customers yet), we cannot construct a

complete tuple on the preceding single relation, because no data concerning customer and account are available yet. To represent incomplete tuples, we must use null values that signify that the value is unknown or does not exist. Thus, in our example, the values for **customer_name**, **customer_street**, and so on must be null. By using several relations, we can represent the branch information for a bank with no customers without using null values. We simply use a tuple on Branch_schema to represent the information about the branch, and create tuples on the other schemas only when the appropriate information becomes available.

Schema Diagram

A database schema, along with primary key and foreign key dependencies, can be depicted pictorially by schema diagrams. Figure 1.7 shows the schema diagram for our banking enterprise. Each relation appears as a box, with the attributes listed inside it and the relation name above it. If there are primary key attributes, a horizontal line crosses the box, with the primary key attributes listed above the line in gray. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation. Many database systems provide design tools with a graphical user interface for creating schema diagrams.

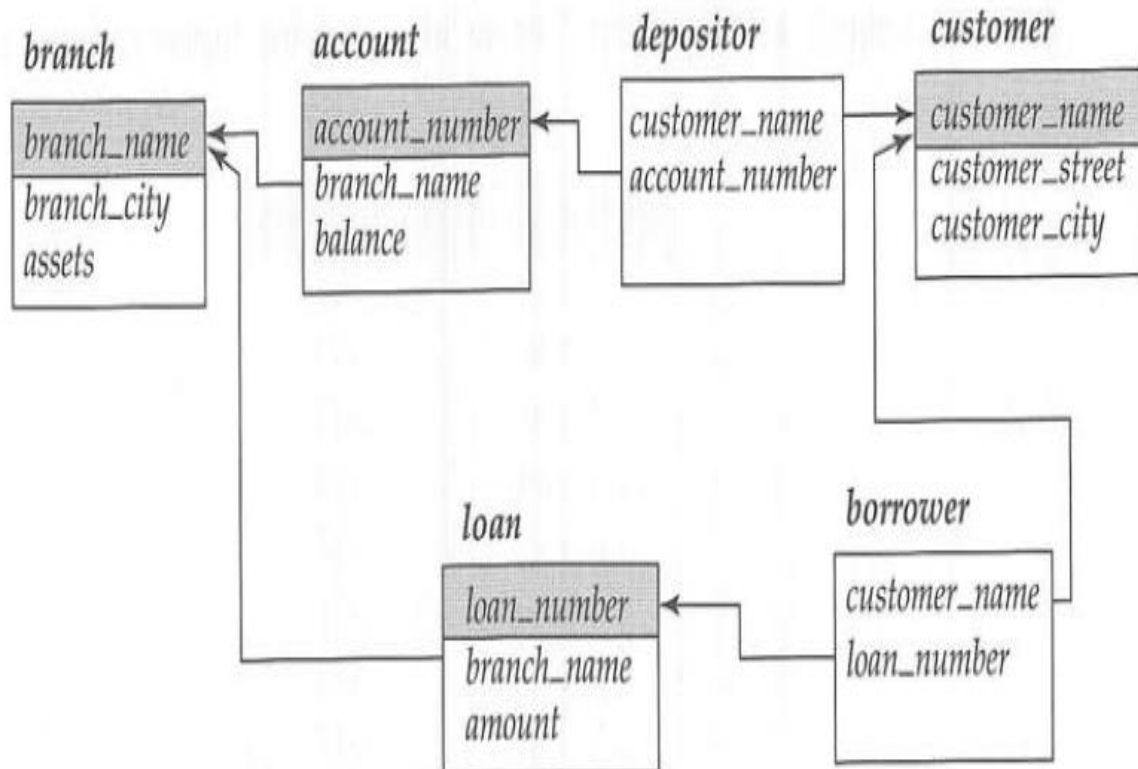


Figure1.6 Schema Diagram of a Banking Enterprise

(iii). Keys of Relational Database (6 Marks)

We must have a way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can uniquely identify the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.

Keys: $K \subseteq R$

K is a super key of R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$. By “possible r” we mean a relation r that could exist in the enterprise we are modeling.

Example: {customer_name, customer_street} and {customer_name} are both super keys of Customer, if no two customers can possibly have the same name. In real life, an attribute such as customer_id would be used instead of customer_name to uniquely identify customers, but we omit it to keep our examples small, and instead assume customer names are unique.

Types of keys:

1) K is a **candidate key** if K is minimal

Example: {customer_name} is a candidate key for Customer, since it is a super key and no subset of it is a super key.

2) **Primary key:** a candidate key chosen as the principal means of identifying tuples within a relation should choose an attribute whose value never, or very rarely, changes.

E.g. email address is unique, but may change.

3) **Foreign Keys(2 MARKS NOV 2012),APRIL 2014.**

A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a foreign key.

E.g. customer_name and account_number attributes of depositor are foreign keys to customer and account respectively.

Only values occurring in the primary key attribute of the referenced relation may occur in the foreign key attribute of the referencing relation.

(iv) Query Languages (4 marks)

A query language is a language in which a user requests information from the database. These languages are usually on a level higher than that of a standard programming

Language. Query languages can be categorized as either procedural or nonprocedural. In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a nonprocedural language, the user describes the desired information without giving a specific procedure for obtaining that information.

Most commercial relational database systems offer a query language that includes elements of both the procedural and the nonprocedural approaches. There are a number of "pure" query languages: The relational algebra is procedural, whereas the tuple relational calculus and domain relational calculus are nonprocedural. These query languages are terse and formal, lacking the "syntactic sugar" of commercial languages, but they illustrate the fundamental techniques for extracting data from the database.

Relational algebra consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are select, project, union, set difference, Cartesian product, and rename. In addition to the fundamental operations, there are several other operations—namely, set intersection, natural join, division, and assignment. However, a complete data manipulation language includes not only a query language, but also a language for database modification. Such languages include commands to insert and delete tuples, as well as commands to modify parts of existing tuples.

7. Describe Relational Algebra and its operations(11 Marks).

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

The operations can be divided into,

- **Basic operations:** Select, Project, Union, rename, set difference and Cartesian product
- **Additional operations:** Set intersections, natural join, division and assignment.
- **Extended operations:** Aggregate operations and outer join

The fundamental operations are :

1. Select
2. Project
3. Union
4. Set difference
5. Cartesian product and
6. Rename.

Example: Consider simple Relation r

A	B	C	D
A	a	1	7
A	b	5	7
B	b	12	3
B	b	23	10

$\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
A	a	1	7
B	b	23	10

SELECTION OPERATION

- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Defined as:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

- Where p is a formula in propositional calculus consisting of **terms** connected by : \wedge (**and**), \vee (**or**), \neg (**not**)
- Each **term** is one of:
 - <Attribute> op <attribute> or <constant>
 - where op is one of: $=, \neq, >, \geq, <, \leq$
- **Select:** It selects tuples that satisfy a given predicate, To denote selection. (Sigma) is used.

Syntax

σ condition (table name)

Example

$\sigma_{sal > 1000}(emp)$ ----- It selects tuples whose employee sal is > 1000

PROJECT OPERATION

- It selects attributes from the relation.
- Symbol for project

Syntax:

$$\Pi_{\text{<Attribute list>}} (\text{Table name})$$

Example:

$$\Pi_{\text{eid,sal}} (\text{Employee})$$

- Combining Select & Project Operation

$$\Pi_{\text{eid,sal}} \left(\sigma_{\text{Sal} > 1000} (\text{employee}) \right)$$

- Selects tuples where $\text{sal} > 1000$ & from them only eid and salary attributes are selected.

UNION OPERATION

Consider a query to find the names of all bank customers who have either an account or a loan or both. Note that the customer relation does not contain the information, since a customer does not need to have either an account or a loan at the bank.

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

$$\Pi_{\text{customer-name}} (\text{borrower})$$

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

$$\Pi_{\text{customer-name}} (\text{depositor})$$

The binary operation union, denoted, as in set theory, by \cup . So the expression needed is $\Pi_{\text{customer-name}} (\text{borrower}) \cup \Pi_{\text{customer-name}} (\text{depositor})$. Therefore, for a union operation $r \cup s$ to be valid, we require that two conditions hold:

- The relations r and s must be of the same arity. That is, they must have the same number of attributes.
- The domains of the i th attribute of r and the i th attribute of s must be the same, for all i .

THE SET DIFFERENCE OPERATION:

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

3. We can find all customers of the bank who have an account but not a loan by writing $\Pi_{\text{customer-name}}(\text{depositor}) - \Pi_{\text{customer-name}}(\text{borrower})$

THE CARTESIAN-PRODUCT OPERATION

1. The Cartesian-product operation, denoted by a cross (\times), allows us to combine information from any two relations.
2. We write the Cartesian product of relations r_1 and r_2 as $(r_1 \times r_2)$.
3. Suppose that we want to find the names of all customers who have a loan at the Perryridge branch. We need the information in both the loan relation and the borrower relation to do so.
4. If we write $\sigma_{\text{branch-name} = \text{"Perryridge"}}(\text{borrower} \times \text{loan})$. However, the customer-name column may contain customers
5. Finally, since we want only customer-name, we do a projection
 $\Pi_{\text{customer-name}}(\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}}(\sigma_{\text{branch-name} = \text{"Perryridge"}}(\text{borrower} \times \text{loan})))$

THE RENAME OPERATION:

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

Consider the following example query:

Find the names of all customers who live on the same street and in the same city as Smith."
 We can obtain Smith's street and city by writing

$\Pi_{\text{customer-street, customer-city}}(\sigma_{\text{customer-name} = \text{"Smith"}}(\text{customer}))$

However, in order to find other customers with this street and city, we must reference the customer relation a second time. In the following query, we use the rename operation on the preceding expression to give its result the name *smith-addr*, and to rename its attributes to street and city, instead of customer-street and customer-city:

**$\Pi_{\text{customer.customer-name}}$
 $(\sigma_{\text{customer.customer-street} = \text{smith-addr.street}} \wedge$
 $\text{Customer.customer-city} = \text{smith-addr.city}$
 $(\text{customer} \times \rho_{\text{smith-addr}}(\text{street, city}))$
 $(\Pi_{\text{customer-street, customer-city}}(\sigma_{\text{customer-name} = \text{"Smith"}}(\text{customer}))))$**

ADDITIONAL OPERATIONS:

The fundamental operations of the relational algebra are sufficient to express any relational-algebra query. However, if we restrict ourselves to just the fundamental operations, certain common queries are lengthy to express. We define additional operations that do not add any power to the algebra, but simplify common queries.

The various additional operations are:

- i) The Set-Intersection Operation
- ii) The Natural-Join Operation
- iii) The Division Operation
- iv) The Assignment Operation

THE SET-INTERSECTION OPERATION

1. The first additional-relational algebra operation that we shall define is set intersection (\cap).
2. Suppose that we wish to find all customers who have both a loan and an account. Using set intersection, we can write:

$\Pi_{\text{customer-name}}(\text{borrower}) \cap \Pi_{\text{customer-name}}(\text{depositor})$

THE NATURAL-JOIN OPERATION

1. It is often desirable to simplify certain queries that require a Cartesian product.
2. Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian product.

Consider the query “Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount”

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

The natural join is a binary operation that allows us to combine certain selections and Cartesian product into one operation.

THE DIVISION OPERATION:

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

Step 1:

We can obtain all branches in Brooklyn by the expression

$$r1 = \Pi_{\text{branch-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch}))$$

Step 2:

We can find all (customer-name, branch-name) pairs for which the customer has an account at a branch by writing

$$r2 = \Pi_{\text{customer-name, branch-name}} (\text{depositor} \bowtie \text{account})$$

Step 3:

Now, we need to find a customer who appears in r2 with every branch name in r1.operation that provides exactly those customers is divide operation. We formulate the query by writing

$$\Pi_{\text{customer-name, branch-name}} (\text{depositor} \bowtie \text{account})$$

$$\div \Pi_{\text{branch-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch}))$$

THE ASSIGNMENT OPERATION:

1. It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
2. The assignment operation, denoted by \leftarrow , works like assignment in a programming language.

To illustrate this operation, consider the definition of division

$$\text{temp1} \leftarrow \Pi_{R-S} (r)$$

$$\text{temp2} \leftarrow \Pi_{R-S} ((\text{temp1} \times s) - \Pi_{R-S}, S(r))$$

$$\text{Result} = \text{temp1} - \text{temp2}$$

The evaluation of an assignment does not result in any relation being displayed to the user. Rather, the result of the expression to the right of the \leftarrow is assigned to the relation variable on the left of the \leftarrow . This relation variable may be used in subsequent expressions.

Thus various relational algebra operations are explained.

- 8. Explain in detail about SQL with its structure and example(This question was asked with some selective queries with example)(11 marks)**

SQL**History**

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)

- ANSI and ISO standard SQL:
 - SQL86
 - SQL89
 - SQL92
 - SQL: 1999 (language name became Y2K compliant!)
 - SQL:2003
- Commercial systems offer most, if not all, SQL92 features, plus varying feature sets from later standards and special proprietary features.
 - Not all examples here may work on your particular system.

SQL is nothing but a set of commands/statements and are categorized into following five groups viz., DQL: Data Query Language, DML: Data Manipulation Language, DDL: Data Definition Language, TCL: Transaction Control Language, DCL: Data Control Language.

DQL: SELECT

DML: DELETE, INSERT, UPDATE

DDL: CREATE, DROP, TRUNCATE, ALTER

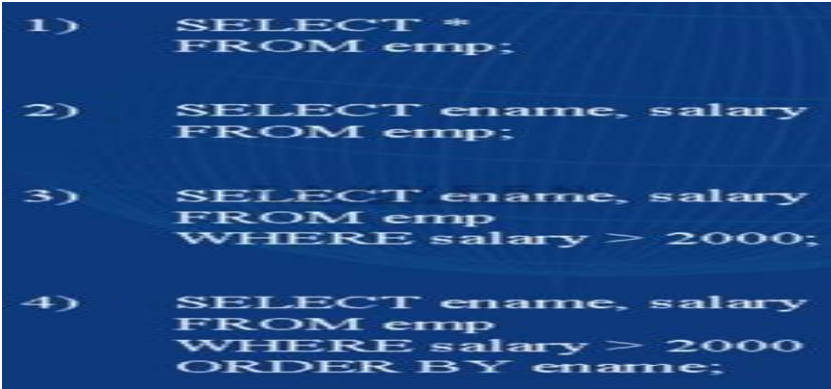
TCL: COMMIT, ROLLBACK, SAVEPOINT

DCL: GRANT, REVOKE

Data Definition Language: Allows the specification of not only a set of relations but also information about each relation, including

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Select



```

1) SELECT *
   FROM emp;

2) SELECT ename, salary
   FROM emp;

3) SELECT ename, salary
   FROM emp
  WHERE salary > 2000;

4) SELECT ename, salary
   FROM emp
  WHERE salary > 2000
  ORDER BY ename;
  
```

The SELECT statement as the name says is used to extract data from Oracle Database. The syntax for the simplest SELECT statement is as follows.

```
SELECT column_name1, column_name2, ...  
FROM table_name1;
```

Example:

```
SELECT * FROM emp;
```

This command will display all the fields of the table emp and all of the records.

Example:

```
SELECT ename, sal FROM emp WHERE sal > 2000;
```

The result of this statement will be only two columns of emp table and only those records where salary is greater than 2000.

Example:

```
SELECT ename, salary  
FROM emp  
WHERE sal > 2000  
ORDER BY ename;
```

The output of this statement will be exactly the same as the one above except that the output will be sorted based on ename column.

SQL Operators

Comparison Operators	Arithmetic Operators
1. =	1. +
2. <	2. -
3. >	3. /
4. <=	4. *
5. >=	
6. ANY or SOME	
7. ALL	
Logical Operators	Other
1. NOT	• IN
2. AND	• BETWEEN
3. OR	• EXISTS
	• LIKE
	• IS NULL
	• IS NOT NULL

Figure - SQL Operators: Comparison, Arithmetic, Logical & Other.

Example:

```
SELECT ename, sal, sal + sal*5/100 "Next Year Sal"  
FROM emp;
```

Logical Operators and the operators in the Other category can be best understood by looking at their respective real world examples.

Example:

```
SELECT sal  
FROM emp  
WHERE deptno = 30 AND sal > 2000;
```

The output of the query will be only one column i.e. sal and only those records will be displayed where department number is 30 and the salary is greater than 2000. So when you use AND operator it means both conditions needs to satisfy for the record to appear in the output but in case of OR, either first condition needs to be true or the second one e.g.

```
SELECT sal  
FROM emp  
WHERE deptno = 30 OR sal > 2000;
```

Example:

```
SELECT *  
FROM emp  
WHERE job IN ('CLERK','ANALYST');
```

The output of the query will be all the columns of emp table but only those records where job column contains either "CLERK" or "ANALYST". You can also use IN operator with as follows.

```
SELECT *  
FROM emp  
WHERE sal IN (SELECT sal  
FROM emp
```

```
WHERE deptno = 30);
```

By having NOT before IN can complete invert the results like in the following example. Such types of queries fall under the category called “Sub-Queries” which we will discuss in the article ahead in this chapter. There is a special technique to interpret them.

```
SELECT *  
FROM emp  
WHERE sal NOT IN (SELECT sal  
FROM emp  
WHERE deptno = 30);
```

Example:

```
SELECT *  
FROM emp  
WHERE sal BETWEEN 2000 AND 3000;
```

Only those records will be displayed where the salary is between 2000 and 3000 including both 2000 and 3000.

Example:

```
SELECT ename, deptno  
FROM dept  
WHERE EXISTS (SELECT *  
FROM emp  
WHERE dept.deptno = emp.deptno);
```

TRUE if a sub-query returns at least one row. In other words the output will be two columns from dept table, all the records only if the query after EXISTS results in at least one record.

Example:

```
SELECT sal  
FROM emp
```



```
WHERE ename LIKE 'SM%';
```

The output will be only those salaries from emp (employee) table where ename (employee name) begins with “SM”. Another variation of above query is as follows.

```
ename LIKE 'SMITH_'
```

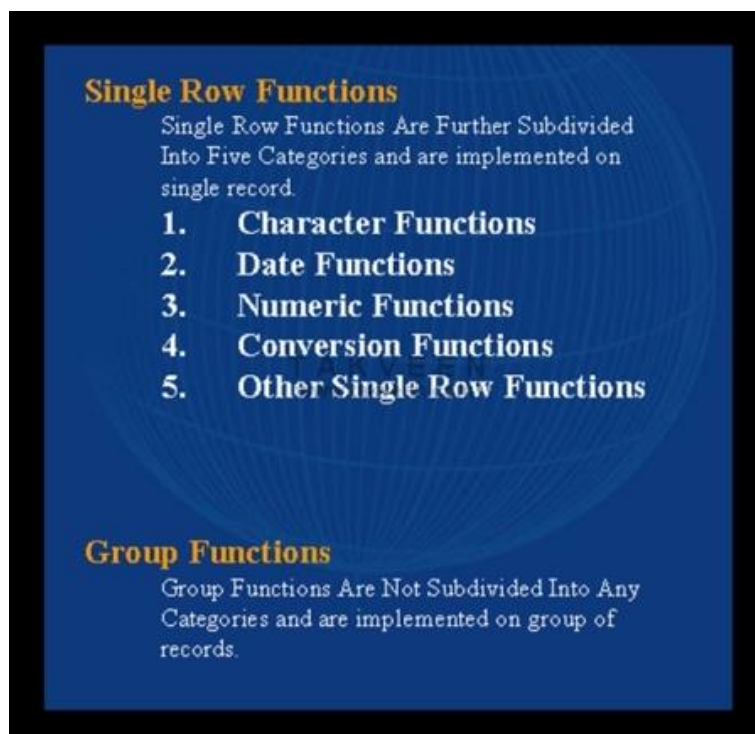
The output will be only those records where ename begins with “SMITH” and there should not be more than one character after it.

Example:

```
SELECT ename, deptno  
FROM emp  
WHERE comm IS NULL;
```

The output will be ename and deptno but only those records where comm field has NULL value. NULL is a special value and just keep in mind that its not Zero. It can be visualized as empty field occupying zero byte

Functions



Single Row Functions:

Single Row functions are further subdivided into five categories viz., Character, Data, Numeric, Conversion & other functions. First we will start with Character Functions or more precisely “Single Row Character Functions.”

Character Functions

Following are the functions that fall under this category.

CHR

LTRIM

INSTR

ASCII

RTRIM

INSTRB

CONCAT

TRIM

LENGTH

INITCAP

REPLACE

LENGTHB

LOWER

SOUNDEX

UPPER

SUBSTR

LPAD

SUBSTRB

RPAD

Example:

```
SELECT CHR(67)||CHR(65)||CHR(84) "Pet" FROM DUAL;
```

Output:

Pet

CAT

Example:

```
SELECT ASCII('Q') FROM DUAL;
```

Output:

ASCII('Q')

81

Example:

```
SELECT CONCAT(ename, ' is a good boy') "Result"
FROM emp
WHERE empno = 7900;
```

Output:

Result

JAMES is a good boy

Example:

```
SELECT INITCAP('the king') "Capitals" FROM DUAL;
```

Output:

Capitals

The King

Example:

```
SELECT LOWER('THE KING') "Lowercase" FROM DUAL;
```

Output:

Lowercase

the king

Similarly we can use the UPPER function.

Example:

```
SELECT LPAD('Page 1',15,'*+') "LPAD example" FROM DUAL;
```

Output:

LPAD example

++*+*+*Page 1

Similarly we can use RPAD.

Example:

```
SELECT LTRIM('121SPIDERMAN','12') "Result" FROM DUAL;
```

Output:

Result

SPIDERMAN

Similarly we can use RTRIM.

Example:

```
SELECT TRIM (0 FROM 001234567000) Result" FROM DUAL;
```

Output:

Result

1234567

```
SELECT TRIM (LEADING 0 FROM 001234567000) Result" FROM DUAL;
```

Output:

Result

1234567000

Similarly we can replace LEADING with TRAILING to omit trailing zeros in 001234567000 and the output will then be 001234567.

Example:

```
SELECT REPLACE('KING KONG','KI','HO') Result" FROM DUAL;
```

Output:

Changes

HONG KONG

Example:

```
SELECT ename FROM emp
WHERE SOUNDEX(ename) = SOUNDEX('SMYTHE');
```

Output:

ENAME

SMITH

This function allows you to compare words that are spelled differently, but sound alike in English. You must have noticed that if you do a search in google (www.google.com) using wrong spelling e.g. I made a wrong spelled word “Neus” search on google and it came up with, “Did you mean News?”. That is basically the beauty of this function.



Figure 7: Google’s implementation of SOUNDEX function. _____

Example:

```
SELECT SUBSTR('SPIDERMAN',7,3) "Result"
FROM DUAL;
```

Output:

Result

MAN

Similarly we can use SUBSTRB; for a single-byte database character set, SUBSTRB is equivalent to SUBSTR. Floating-point numbers passed as arguments to SUBSTRB are automatically converted to integers. Assume a double-byte database character set:

```
SELECT SUBSTRB(' SPIDERMAN',7,4,3) "Result" FROM DUAL;
```

Output:

Result

DE

Example:

```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2) "Instring" FROM DUAL;
```

Output:

Result

14

Similarly we can use INSTRB; for a single-byte database character set, INSTRB is equivalent to INSTR. Lets suppose a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Result" FROM DUAL;
```

Output:

Result

27

Example:

```
SELECT LENGTH('SPIDERMAN') "Result"FROM DUAL;
```

Output:

Result

9

Similarly we can use LENGTHB; for a single-byte database character set, LENGTHB is equivalent to LENGTH. Lets suppose a double-byte database character set.

```
SELECT LENGTHB ('SPIDERMAN') "Result"FROM DUAL;
```

Output:

Result

14

DATE FUNCTIONS:

Following functions fall under this category.

ADD_MONTHS

MONTHS_BETWEEN

LAST_DAY

ROUND

SYSDATE

TRUNC

Example:

```
SELECT ADD_MONTHS(hiredate,1)FROM emp WHERE ename = 'SMITH';
```

Example:

```
SELECT SYSDATE, LAST_DAY(SYSDATE) "Last", LAST_DAY(SYSDATE) - SYSDATE "Days Left"
FROM DUAL;
```

Output:

```
SYSDATE Last Days Left
```

```
-----
```

```
23-OCT-97 31-OCT-97 8
```

Example:

```
SELECT MONTHS_BETWEEN(SYSDATE, hiredate) "Months of Service"
```

```
FROM DUAL;
```

Example:

```
SELECT ROUND (TO_DATE ('27-OCT-92'),'YEAR')
```

```
"New Year" FROM DUAL;
```

Output:

```
New Year
```

```
-----
```

```
01-JAN-93
```

Example:

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')
```

```
"New Year" FROM DUAL;
```

Output:

New Year

01-JAN-92

NUMERIC FUNCTIONS:

The following functions fall under this category.

ABS

ROUND

SIGN

TRUNC

CEIL

SQRT

FLOOR

MOD

Example:

```
SELECT ABS(-25) "Result"
```

```
FROM DUAL;
```

Output:

Result

25

Example:

```
SELECT SIGN(-15) "Result"
```

```
FROM DUAL;
```

Output:

Result

-1

Example:

```
SELECT CEIL(25.7) "Result"
```

```
FROM DUAL;
```

Output:

Result

26

Example:

SELECT FLOOR(25.7) "Result" FROM DUAL;

Output:

Result

25

Example:

SELECT ROUND(25.29,1) "Round" FROM DUAL;

Output:

Round

25.3

SELECT ROUND(25.29,-1) "Round" FROM DUAL;

Output:

Round

30

Example:

SELECT TRUNC(25.29,1) "Truncate" FROM DUAL;

Output:

Truncate

25.2

Example:

SELECT TRUNC(25.29,-1) "Truncate" FROM DUAL;

Output:

Truncate

20

-1 will truncate (make zero) first digit left of the decimal point of 25.29

Example:

```
SELECT MOD(11,4) "Modulus"
```

```
FROM DUAL;
```

Output:

Modulus

3

Example:

```
SELECT SQRT(25) "Square root"
```

```
FROM DUAL;
```

Output:

Square root

5

CONVERSION FUNCTIONS:

The following functions fall under this category

TO_CHAR

TO_DATE

TO_NUMBER

Example:

```
SELECT TO_CHAR(HIREDATE, 'Month DD, YYYY') "Result"
```

```
FROM emp
```

```
WHERE ename = 'BLAKE';
```

Output:

Result

May 01, 1981

Example:

```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Result"
```

```
FROM DUAL;
```

Output:

Result

\$10,000.00-

Example:

```
SELECT TO_DATE('January 15, 1989, 11:00 A.M.',  
'Month dd, YYYY, HH:MI A.M.',  
'NLS_DATE_LANGUAGE = American')  
FROM DUAL;
```

Example:

```
SELECT TO_NUMBER('$10,000.00-', 'L99G999D99MI') "Result"  
FROM DUAL;
```

Output:

Result

-1000

OTHER SINGLE ROW FUNCTIONS:

The following functions fall under this category.

NVL

VSIZE

Example:

```
SELECT ename, NVL(TO_CHAR(COMM), 'NOT APPLICABLE') "COMMISSION" FROM emp
WHERE deptno = 30;
```

Output:

```
ENAME COMMISSION
```

```
-----
```

```
ALLEN 300
```

```
WARD 500
```

```
MARTIN 1400
```

```
BLAKE NOT APPLICABLE
```

```
TURNER 0
```

```
JAMES NOT APPLICABLE
```

Example:

```
SELECT ename, VSIZE (ename) "BYTES"
FROM emp
WHERE deptno = 10;
```

Output:

```
ENAME BYTES
```

```
-----
```

```
CLARK 5
```

```
KING 4
```

```
MILLER 6
```

GROUP FUNCTIONS:

A group function as the name says, gets implemented on more than one record within a column. They can be better understood by looking at their real world examples. There are five very important group functions.

AVG

COUNT

MAX

MIN

SUM

Example:

```
SELECT AVG(sal) "Average" FROM emp;
```

Output:

Average

2077.21429

Example:

```
SELECT COUNT(*) "Total" FROM emp;
```

Output:

Total

18

Total number of records will be returned with a table.

Example:

```
SELECT MAX(sal) "Maximum" FROM emp;
```

Output:

Maximum

5000

On the same line we can find out minimum value using the MIN group function.

Example:

```
SELECT SUM(sal) "Total" FROM emp;
```

Output:

Total

29081

9. Explain in detail about Basic Structure of SQL.(11 marks)

SQL keywords : SQL keywords fall into several groups.

(i). Data retrieval: The most frequently used operation in transactional databases is the data retrieval operation. When restricted to data retrieval commands, SQL acts as a declarative language.

- **SELECT** is used to retrieve zero or more rows from one or more tables in a database. In most applications, SELECT is the most commonly used Data Manipulation Language command. In specifying a SELECT query, the user specifies a description of the desired result set, but they do *not* specify what physical operations must be executed to produce that result set. Translating the query into an efficient query plan is left to the database system, more specifically to the query optimizer.
 - Commonly available keywords related to SELECT include:
 - FROM is used to indicate from which tables the data is to be taken, as well as how the tables JOIN to each other.
 - WHERE is used to identify which rows to be retrieved, or applied to GROUP BY. WHERE is evaluated before the GROUP BY.
 - GROUP BY is used to combine rows with related values into elements of a smaller set of rows.
 - HAVING is used to identify which of the "combined rows" (combined rows are produced when the query has a GROUP BY keyword or when the SELECT part contains aggregates), are to be retrieved. HAVING acts much like a WHERE, but it operates on the results of the GROUP BY and hence can use aggregate functions.
 - ORDER BY is used to identify which columns are used to sort the resulting data.

Example 1:

```
SELECT * FROM books WHERE price > 100.00 and price < 150.00 ORDER BY title
```

This is an example that could be used to get a list of expensive books. It retrieves the records from the *books* table that have a *price* field which is greater than 100.00. The result is sorted alphabetically by book title. The asterisk (*) means to show all columns of the *books* table. Alternatively, specific columns could be named.

Example 2:

```
SELECT books.title, count(*) AS Authors FROM books JOIN book_authors ON  
books.book_number = book_authors.book_number GROUP BY books.title
```

Example 2 shows both the use of multiple tables in a join, and aggregation (grouping). This example shows how many authors there are per book. Example output may resemble:

Title	Authors
-----	-----
SQL Examples and Guide	3
The Joy of SQL	1
How to use Wikipedia	2
Pitfalls of SQL	1
How SQL Saved my Dog	1

Data manipulation: First there are the standard Data Manipulation Language (DML) elements. DML is the subset of the language used to add, update and delete data.

- INSERT is used to add zero or more rows (formally tuples) to an existing table.
- UPDATE is used to modify the values of a set of existing table rows.
- MERGE is used to combine the data of multiple tables. It is something of a combination of the INSERT and UPDATE elements. It is defined in the SQL:2003 standard; prior to that, some databases provided similar functionality via different syntax, sometimes called an "update".
- TRUNCATE deletes all data from a table (non-standard, but common SQL command).
- DELETE removes zero or more existing rows from a table.

Example:

```
INSERT INTO my_table (field1, field2, field3) VALUES ('test', 'N', NULL);
```

```
UPDATE my_table SET field1 = 'updated value' WHERE field2 = 'N';
```

```
DELETE FROM my_table WHERE field2 = 'N';
```

Data transaction: Transaction, if available, can be used to wrap around the DML operations.

- BEGIN WORK (or START TRANSACTION, depending on SQL dialect) can be used to mark the start of a database transaction, which either completes completely or not at all.

- COMMIT causes all data changes in a transaction to be made permanent.
- ROLLBACK causes all data changes since the last COMMIT or ROLLBACK to be discarded, so that the state of the data is "rolled back" to the way it was prior to those changes being requested.

COMMIT and ROLLBACK interact with areas such as transaction control and locking. Strictly, both terminate any open transaction and release any locks held on data. In the absence of a BEGIN WORK or similar statement, the semantics of SQL are implementation-dependent.

Example:

```
BEGIN WORK;
```

```
UPDATE inventory SET quantity = quantity - 3 WHERE item = 'pants';
```

```
COMMIT;
```

Data definition: The second group of keywords is the Data Definition Language (DDL). DDL allows the user to define new tables and associated elements. Most commercial SQL databases have proprietary extensions in their DDL, which allow control over nonstandard features of the database system.

The most basic items of DDL are the CREATE and DROP commands.

- CREATE causes an object (a table, for example) to be created within the database.
- DROP causes an existing object within the database to be deleted, usually irretrievably.

Some database systems also have an ALTER command, which permits the user to modify an existing object in various ways -- for example, adding a column to an existing table.

Example:

```
CREATE TABLE my_table (
my_field1 INT,
my_field2 VARCHAR (50),
my_field3 DATE NOT NULL,
PRIMARY KEY (my_field1, my_field2) )
```

All DDL statements are auto commit so while dropping a table need to have close look at its future needs.

Data control

The third group of SQL keywords is the Data Control Language (DCL). DCL handles the authorization aspects of data and permits the user to control who has access to see or manipulate data within the database.

Its two main keywords are:

- **GRANT** — authorizes one or more users to perform an operation or a set of operations on an object.
- **REVOKE** — removes or restricts the capability of a user to perform an operation or a set of operations.
- **Example:**

```
GRANT SELECT, UPDATE ON my_table TO some_user, another_user;
```

10.Explain in detail about SET operations in SQL(5 Marks)

The set operators in SQL are based on the same principles, except they don't have a complement, and can determine the 'difference' between two sets. Here are the operators which we apply to combine two queries:

- **union** - all elements in both queries are returned;
- **intersect** - elements common to both queries are returned;
- **except** - elements in the first query are returned *excluding* any that were returned by the second query.

These are powerful ways of manipulating information, but take note: you can only apply them if the results of the two queries (that are going to be combined) have the same format - that is, the same number of columns, and identical column types! (Although many SQLs try to be helpful by, for example, coercing one data type into another, an idea which is superficially helpful and fraught with potential for errors). The general format of such queries is illustrated by:

```
select columns1 from table1 union select columns2 from table2
```

Different strokes:

Different vendor implementations of SQL have abused the SQL-92 standard in different ways. For example, Oracle uses **minus** where SQL-92 uses **except**.

An outer join could be used (with modification for NULLs if these little monstrosities are present) to achieve the same result as **except**. Similarly, an inner join (with select distinct) can do what **intersect** does.

Set operators can be combined (as you would expect when playing around with sets) to achieve results that simply cannot be obtained using a single set operator.

Note that there are some restrictions on using **order by** with set operators - **order by** may only be used once, no matter how big the compound statement, and the select list must contain the columns being used for the sort.

Pseudoset operators

Not content with implementing set operators, SQL database creators have also introduced what are called "pseudoset operators". These operators don't fit conveniently into set theory, because they allow multiple rows (redundancies) which are forbidden in true sets. We use the pseudoset operator **union all** to combine the outputs of two queries (all that is done is that the results of the second query are appended to the results of the first). Union all does exactly what we required from a FULL OUTER JOIN.

11. Write Short Notes on Null Values(5 Marks)

SQL allows the use of null values to indicate absence of information about the value of an attribute. We can use the special keyword null in a predicate to test for a null value. Wherever a value can appear in SQL, in particular in place of a column value in some row. The deviation from the relational model arises from the fact that the implementation of this ad hoc concept in SQL involves the use of three-valued logic, under which the comparison of NULL with itself does not yield true but instead yields the third truth value, unknown; similarly the comparison NULL with something other than itself does not yield false but instead yields unknown. It is because of this behavior in comparisons that NULL is described as a mark rather than a value.

The relational model depends on the law of excluded middle under which anything that is not true is false and anything that is not false is true; it also requires every tuple in a relation body to have a value for every attribute of that relation. This particular deviation is disputed by some if only because E.F. Codd himself eventually advocated the use of special marks and a 4-valued logic, but this was based on his observation that there are two distinct reasons why one might want to use a special mark in place of a value, which led opponents of the use of such logics to discover more distinct reasons and at least as

many as 19 have been noted, which would require a 21-valued logic. SQL itself uses NULL for several purposes other than to represent "value unknown". For example, the sum of the empty set is NULL, meaning zero, the average of the empty set is NULL, meaning undefined, and NULL appearing in the result of a LEFT JOIN can mean "no value because there is no matching row in the right-hand operand".

12. Write Short Notes on Nested Sub Queries(6 Marks)

NESTED SUBQUERIES: SQL provides a mechanism for nesting sub queries. A subquery is a select-from where expression that is nested within another query. A common use of sub queries is to perform tests for set membership, make set comparisons, and determine set cardinality.

Subqueries: Wouldn't it be nice if you could perform one query, temporarily store the result(s), and then use this result as part of another query? You can, and the trickery used is called a subquery. The basic idea is that instead of a 'static' condition, you can insert a query as part of a where clause! An example is:

Select * from tablename **where** value > (insert select statement here);

Note that in the above query, the inner **select** statement must return just one value (for example, an average). There are other restrictions - the subquery must be in parenthesis, and it must be on the right side of the conditional operator (here, a greater than sign). You can use such sub queries with =, >, <, >=, <= and <>, but not {to the best of my knowledge?} with **between.. And**.

Multiple select sub queries can be combined (using logical operators) in the same statement, but avoid complex queries if you possibly can!

Subqueries that return multiple values

In the above, we made sure that our subquery only returned a single value. Can you think of a way you might use a subquery that returns a list of values? (Such a thing is possible)! Yes, you need an operator that works on lists. An example of such an operator is **in**, which we've encountered before. The query should look something like:

select*from tablename **where** value **in** (insert select statement here);

The assumption is that the nested **select** statement returns a list. The outer shell of the statement can then use **in** to get cracking on the list, looking for a match of value within the list! There is a surprisingly long list of operators that resemble **in**, and can be used in a similar fashion. Here it is:

Operator	What it does
not in	There is no match with any value retrieved by the nested select statement.
in	We know how this works. Note that = any is a synonym for in that you'll sometimes encounter!
> any	The value is greater than any value in the list produced by the inner submit statement. This is a clumsy way of saying "Give me the value if it's bigger than the smallest number retrieved"!
>= any < any <= any	Similar to > . Usage should be obvious.
> all	Compare this with > any - it should be clear that the condition will only succeed if the value is bigger than the largest value in the list returned by the inner select statement!
>= all < all <= all	If you understand > all , these should present no problem!
= all	You're not likely to use this one much. It implies that (to succeed) all the values returned by the inner subquery are equal to one another and the value being tested!

(It is even possible in some SQL implementations to retrieve a 'table' (multiple columns) using the inner **select** statement, and then use **in** to simultaneously compare multiple values with the rows of the table produced by this inner select. You'll probably never need to use something like this. Several other tricks are possible, including the creation of virtual views by using a subquery

Correlated Subqueries

Whew, we're nearly finished with the sub queries, but there is one more distinct flavour The correlated subquery is a nested **select** statement that can (using trickery) refer to the outer

select statement containing it. By so doing, we can successively apply the inner select statement to each line generated by the outer statement! The trick that we use is to create an alias for the outer select statement, and then refer to this alias in the inner select statement, thus constraining the inner select to dealing with the relevant row. For an example, we return to our tedious drug table:

DrugDosing		
Dosing	DoseMg	Frequency
D1	30	OD
D2	10	OD
D3	200	TDS

Let's say we wanted (for some obscure reason) all doses that are greater than average dose, for each dosing frequency. [Meaningless, but it serves the purposes of illustration].

Select Dose Mg, Frequency **from** DrugDosing fred **where** DoseMg>(selectavg(DoseMg) **from** DrugDosing **where** Frequency = fred. Frequency) ;

The sense of this statement should be clear - we use the outer **select** to choose a row (into which we put DoseMg, and Frequency). We then check whether this row is a candidate (or not) using the **where** statement. What does the **where** statement check? Well, it makes sure that DoseMg is greater than a magic number. The magic number is the average dose for a particular Frequency, the frequency associated with the current row. The only real trickery is how we use the label fred to refer to the current line from within the inner select statement. This label fred is called an alias. We'll learn a lot more about aliases later (Some will argue that aliases are so important you should have encountered them long before, but we disagree).

Correlated sub queries are not the only way of doing the above. Using a temporary view is often more efficient, but it's worthwhile knowing both techniques.

13. Write Short Notes on SQL Views(6 Marks)(2 Marks)

View gives a particular user access to selected portions of a table. A view is however more than this - it can limit the ability of a user to update parts of a table, and can even amalgamate rows, or throw in additional columns derived from other columns. Even more complex applications of views allow several tables to be combined into a single view!

How do we make a view?

Interestingly enough, you use a **select** statement to specify the view you wish to create. The syntax is:

Create view *nameofview* **as select** *here have details of select statement*

A variant that you will probably use rather often is:

Create or replaceview *nameofview* **as select** *here have details of select statement*

(Otherwise you have to explicitly destroy a view - SQL won't simply overwrite a view without the **or replace** instruction, but will instead give you an irritating error).

One *cannot* use an **order by** statement (or something else called a **for update** clause) within a view. There is a whole lot of other convenient things you can do to views. Where you include summary statistics (eg count, sum, etc) in a view it is termed an *aggregate view*. Likewise, using **distinct**, you can have a view on the possible values of a column or grouping of columns.

You can even create a view that is derived from several tables (A *multi-table view*). This is extremely sneaky, as you can largely avoid complex join statements in code which pulls data out of several tables.

How do we limit access to a view?

This is implicit in the way we sneakily use a select statement - to limit access to certain columns, for example, we just **select** the column-names we want access to, and ignore the rest! - if you **insert** a row, then SQL doesn't know what to put into the column entry that's not represented in the view, so it will insert either NULL, or the default value for that column. (Likewise, with **delete**, the entire row will be deleted, even the column entry that is invisible).

It is also obvious how we limit access to certain *rows* - we use a **where** clause that only includes the rows we want in the view. Note that (depending on your selection criterion) it is possible to insert a row into a view (and thus the underlying database) and then not be able to see this row in the view! With (in)appropriate selection criteria for the view, one can also alter the properties of rows visible in the view so that they now become hidden!

More draconic constraints are possible. The option **with read only** prevents any modifications to the view (or the underlying database); while **with check option** prevents

you from creating rows in the view that cannot be selected (seen) in the view itself. If you use the "with check option", then you should follow this with a name, otherwise SQL will create an arbitrary and quite meaningless name for the constraint that will only confuse you when an error occurs!

How do we remove a view?

More things we can do with views

One trick is to create a "virtual view" and then use this as a "table" which you can update. Instead of specifying a table name, you specify (in parenthesis) the select statement that defines the "virtual view", and all actions are performed on this temporary table! This is particularly useful where you don't have the system authority to create a view, yet need the power of a view.

14. Explain in detail Unified Modeling Language(5 Marks)APRIL 2014.(2 Marks)NOV 2014

UML is officially defined at the Object Management Group (OMG) by the UML metamodel, a Meta-Object Facility metamodel (MOF). Like other MOF-based specifications, the UML metamodel and UML models may be serialized in XML. UML was designed to specify, visualize, construct, and document software-intensive systems.

UML is not restricted to modeling software. UML is also used for business process modeling, systems engineering modeling, and representing organizational structures. The Systems Modeling Language (SysML) is a Domain-Specific Modeling language for systems engineering that is defined as a UML 2.0 profile.

UML has been a catalyst for the evolution of model-driven technologies, which include model-driven development (MDD), model-driven engineering (MDE), and model-driven architecture (MDA). By establishing an industry consensus on a graphic notation to represent common concepts like classes, components, generalization, aggregation, and behaviors, UML has allowed software developers to concentrate more on design and architecture.

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages, supported by the OMG.

UML is extensible, offering the following mechanisms for customization: profiles and stereotype. The semantics of *extension by profiles* have been improved with the UML 2.0 major revision.

15.Explain in detail about the various Data Models(11 Marks)

Data Models is a collection of tools for describing

- Data
- Data relationships
- Data semantics
- Data constraints

Entity Relationship Model: E-R model of real world 5 Entities (objects) E.g. customers, accounts, bank branch 5 Relationships between entities E.g. Account A-101 is held by customer Johnson Relationship set depositor associates customers with accounts s Widely used for database design 5 Database design in E-R model usually converted to design in the relational model (coming up next) which is used for storage and processing

Relational Model Attributes

Example of tabular data in the relational model

customer- customer- account- customer- Customer-id street city number

name Johnson 192-83-7465

Alma A-101 Palo Alto

Smith 019-28-3746 North A-215 Rye

Data Definition Language (DDL)

It's a **Specification** notation for defining the database schema

E.g. create table account (account-number char (10), balance integer) s DDL compiler generates a set of tables stored in a data dictionary s Data dictionary contains metadata (i.e., data about data) database schema, Data storage and definition language in which the storage structure and access methods used by the database system are specified Usually an extension of the data definition language

Data Manipulation Language (DML)

It's a Language for accessing and manipulating the data organized by the appropriate data model. DML also known as query language is two classes of languages

- Procedural – user specifies what data is required and how to get those data

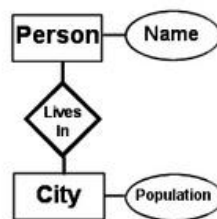
- Nonprocedural – user specifies what data is required without specifying how to get those data s SQL is the most widely used query language

Entity-relationship (ER)

Definition: An entity-relationship (ER) diagram is a specialized graphic that illustrates the interrelationships between entities in a database. ER diagrams often use symbols to represent three different types of information. Boxes are commonly used to represent entities. Diamonds are normally used to represent relationships and ovals are used to represent attributes.

Also Known As: ER Diagram, E-R Diagram, entity-relationship model

Examples: Consider the example of a database that contains information on the residents of a city. The ER diagram shown in the image above contains two entities -- people and cities. There is a single "Lives In" relationship. In our example, due to space constraints, there is only one attribute associated with each entity. People have names and cities have populations. In a real-world example, each one of these would likely have many different attributes.



Entity Sets in a database can be modeled as:

- A collection of entities,
- Relationship among entities.

an entity is an object that exists and is distinguishable from other objects. Example: specific person, company, event, plant s Entities have attributes. Example: people have names and addresses is an entity set is a set of entities of the same type that share the same properties. Example: set of all persons, companies, trees, holidays

Entity Sets customer and loan customer-id customer- customer- customer- loan- amount
name street city number

Attributes

An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set. Example: customer = (customer-id, customer-name, customer-street, customer-city) loan = (loan-number, amount) s Domain – the set of permitted values for each attribute is Attribute types:

- Simple and composite attributes.
- Single-valued and multi-valued attributes E.g. multivalued attribute: phone-numbers
- Derived attributes Can be computed from other attributes E.g. age, given date of birth

Relationship Sets

A relationship is an association among several entities Example: Hayes depositor A-102 customer entity relationship set account entity s A relationship set is a mathematical 2 entities, each taken from entity sets $\{(e_1, e_2, \dots, e_n) \mid e_i \in \text{relation among } n \text{ } E_n\}$ where (e_1, e_2, \dots, e_n) is a relationship. An attribute can also be property of a relationship set. s For instance, the depositor relationship set between entity sets customer and account may have the attribute access-date Degree of a Relationship Set Refers to number of entity sets that participate in a relationship set. s Relationship sets that involve two entity sets are binary (or degree two). Generally, most relationship sets in a database system are binary. Relationship sets may involve more than two entity sets. E.g. Suppose employees of a bank may have jobs (responsibilities) at multiple branches, with different jobs at different branches. Then there is a ternary relationship set between entity sets employee, job and branch Relationships between more than two entity sets are rare. Most relationships are binary.

Mapping Cardinalities (2 MARKS NOV 2012)

It Express the number of entities to which another entity can be associated via a relationship set. Most useful in describing binary relationship sets. For a binary relationship set the mapping cardinality must be one of the following types:

- One to one
- One to many
- Many to one
- Many to many

Mapping Cardinalities One to one One to many Note: Some elements in A and B may not be mapped to any elements in the other Mapping Cardinalities Many to one Many to many Note: Some elements in A and B may not be mapped to any elements in the other set

Mapping Cardinalities affect ER Design Can make access-date an attribute of account, instead of a relationship attribute, if each account can have only one customer s I.e., the relationship from account to customer is many to one, or equivalently, customer to account is one to many

E-R Diagrams

- Rectangles represent entity sets.
- Diamonds represent relationship sets.
- Lines link attributes to entity sets and entity sets to relationship sets.
- Ellipses represent attributes
- Double ellipses represent multivalued attributes.
- Dashed ellipses denote derived attributes.
- Underline indicates primary key attributes

Roles

Entity sets of a relationship need not be distinct, The labels “manager” and “worker” are called roles; they specify how employee entities interact via the works-for relationship set. Roles are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles. s Role labels are optional, and are used to clarify semantics of the relationship

Cardinality Constraints

We express cardinality), signifying “one,” or an constraints by drawing either a directed line (undirected line (—), signifying “many,” between the relationship set and the entity set. s E.g.: One-to-one relationship:

- A customer is associated with at most one loan via the relationship borrower
- A loan is associated with at most one customer via borrower

One-To-Many Relationship In the one-to-many relationship a loan is associated with at most one customer via borrower, a customer is associated with several (including 0) loans via borrower

Many-To-One Relationships In a many-to-one relationship a loan is associated with several (including 0) customers via borrower, a customer is associated with at most one loan via borrower

Participation of an Entity Set in a Relationship Set

Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set E.g. participation of loan in borrower is total s every loan must have a customer associated to it via borrower. Partial participation: some entities may not participate in any relationship in the relationship set s E.g. participation of customer in borrower is partial

Keys

A super key of an entity set is a set of one or more attributes whose values uniquely determine each entity.

A candidate key of an entity set is a minimal super key

Customer-id is candidate key of customer

Eg: - Account-number is candidate key of account s Although several candidate keys may exist, one of the candidate keys is selected to be the primary key.

Keys for Relationship Sets s The combination of primary keys of the participating entity sets forms a super key of a relationship set.

(customer-id, account-number) is the super key of depositor

NOTE: this means a pair of entity sets can have at most one relationship in a particular relationship set. E.g. if we wish to track all access-dates to each account by each customer, we cannot assume a relationship for each access. We can use a multivalued attribute though. Must consider the mapping cardinality of the relationship set when deciding the what are the candidate keys. Need to consider semantics of relationship set in selecting the primary key in case of more than one candidate key

Cardinality Constraints on Ternary Relationship

We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint. E.g. an arrow from works-on to job indicates each employee works on at most one job at any branch. If there is more than one arrow, there are two ways of defining the meaning. E.g. a ternary relationship R between A, B and C with arrows to B and C could mean 1. each A entity is associated with a unique entity from B and C or 2. each pair of entities from (A, B) is associated with a unique C entity, and each pair (A, C) is associated with a unique B. Each alternative has been used in different formalisms. To avoid confusion we outlaw more than one arrow. DBMS Notes by Ankur Shukla 2.27 www.ankurshukla.com

Binary Vs. Non-Binary Relationships

Some relationships that appear to be non-binary may be better represented using binary relationships. E.g. A ternary relationship parents, relating a child to his/her father and mother, is best replaced by two binary relationships, father and mother. Using two binary relationships allows partial information (e.g. only mother being known). But there are some relationships that are naturally non-binary. E.g. works-on.

Design Issues

Use of entity sets vs. attributes: Choice mainly depends on the structure of the enterprise being modeled, and on the semantics associated with the attribute in question. Use of entity sets vs. relationship sets: Possible guideline is to designate a relationship set to describe an action that occurs between entities. Binary versus n-ary relationship sets: Although it is possible to replace any nonbinary (n-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets, an n-ary relationship set shows more clearly that several entities participate in a single relationship. Placement of relationship attributes.

Weak Entity Sets (2 MARKS)

An entity set that does not have a primary key is referred to as a weak entity set. The existence of a weak entity set depends on the existence of an identifying entity set.

It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set. Identifying relationship depicted using a double diamond. The discriminator (or partial key) of a weak entity set is the set of attributes that

distinguishes among all the entities of a weak entity set. The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator. We depict a weak entity set by double rectangles. We underline the discriminator of a weak entity set with a dashed line. payment-number – discriminator of the payment entity set. Primary key for payment – (loan-number, payment-number)

Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship. If loan-number were explicitly stored, payment could be made a strong entity, but then the relationship between payment and loan would be duplicated by an implicit relationship defined by the attribute loan-number common to payment and loan

Specialization

Top-down design process; we designate sub groupings within an entity set that are distinctive from other entities in the set. These sub groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set. s Depicted by a triangle component labeled ISA (E.g. customer "is a" person). s Attribute inheritance – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Generalization A bottom-up design process – combine a number of entity sets that share the same features into a higher-level entity set. Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way. The terms specialization and generalization are used interchangeably.

Aggregation Consider the ternary relationship works-on, which we saw earlier s Suppose we want to record managers for tasks performed by an employee at a branch. Relationship sets works-on and manages represent overlapping information

- Every manages relationship corresponds to a works-on relationship
- However, some works-on relationships may not correspond to any manages relationships So we can't discard the works-on relationship
- Eliminate this redundancy via aggregation
- Treat relationship as an abstract entity
- Allows relationships between relationships

- Abstraction of relationship into new entity Without introducing redundancy, the following diagram represents:
- An employee works on a particular job at a particular branch
- An employee, branch, job combination may have an associated

16.Explain in detail about the modification of the database(6 Marks)

Modification of the Database: The content of the database may be modified using the following operations:

- Deletion
- Insertion
- Updating

All these operations are expressed using the assignment operator.

Deletion A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database. Can delete only whole tuples; cannot delete values on only particular attributes. A deletion is expressed in relational $r-E$ where r is a relation and E is a relational algebra query.

Delete all account records in branch-name = "Perryridge"

Insertion To insert data into a relation, we either: specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted s in relational algebra.

Updating A mechanism to change a value in a tuple without changing all values in the tuple s . Use the generalized projection $F_1, F_2, \dots, F_l, (r)$ s . Each F_i is either the i th operator to do this task r attribute of r , if the i th attribute is not updated, or, if the attribute is to be updated F_i is an expression, involving only constants and the attributes of r , which gives the new value for the attribute

Views

In some cases, it is not desirable for all users to see the entire logical model (i.e., all the actual relations stored in the database.) Consider a person who needs to know a customer's loan number but has no need to see the loan amount. This person should see a relation customer-name, loan-number (borrower described, in the relational algebra, by loan). Any relation that

is not of the conceptual model but is made visible to a user as a “virtual relation” is called a view.

View Definition s A view is defined using the create view statement which has the form create view v as <query expression where <query expression> is any legal relational algebra query expression. The view name is represented by v. Once a view is defined, the view name can be used to refer to the virtual relation that the view generates. View definition is not the same as creating a new relation by evaluating the query expression. Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

17.Explain in detail Embedded SQL and Dynamic SQL (11 Marks)

Embedded SQL is a method of combining the computing power of a programming language and the database manipulation capabilities of SQL. It allows programmers to embed SQL statements in programs written in C/C++, Fortran, COBOL, Pascal, etc.

Embedded SQL statements are SQL statements written within application programming languages and preprocessed by a SQL preprocessor before the application program is compiled. There are two types of embedded SQL: static and dynamic.

What is host language?

The SQL standard defines embedding of SQL as *embedded SQL* and the language in which SQL queries are embedded is referred to as the *host language*. A popular host language is C and the mixed C and embedded SQL is called Pro*C in Oracle and Sybase database management systems. Other embedded SQL precompilers are Pro*COBOL, Pro*FORTRAN, Pro*PL/I, Pro*Pascal, and SQL*Module (for Ada).

Using sqlca

- sta-select.sqc (a sample static embedded SQL program)
- dyn-select.sqc (a sample dynamic embedded SQL program)
- makefile (a sample makefile for these two programs)

Using sqlstate (for questions on how to use sqlstate, contact drosu@cs.toronto.edu)

- sta-select.sqc (a sample static embedded SQL program)
- dyn-select.sqc (a sample dynamic embedded SQL program)
- makefile (a sample makefile for these two programs)

Note: before you compile these programs, you need to create the following table and insert some tuples in it.

```
--*****
```

```
create table video(video_id int not null, \
video_title varchar(30), \
director varchar(20), \
primary key(video_id))
```

```
insert into video values(100,'Titanic','John')
```

```
insert into video values(200,'Mask','Mary')
```

```
insert into video values(300,'Mary had a little lamb','Jim')
```

Dynamic SQL

What Is Dynamic SQL?

Most database applications do a specific job. For example, a simple program might prompt the user for an employee number, then update rows in the EMP and DEPT tables. In this case, you know the makeup of the UPDATE statement at precompile time. That is, you know which tables might be changed, the constraints defined for each table and column, which columns might be updated, and the datatype of each column.

However, some applications must accept (or build) and process a variety of SQL statements at run time. For example, a general-purpose report writer must build different SELECT statements for the various reports it generates. In this case, the statement's makeup is unknown until run time. Such statements can, and probably will, change from execution to execution. They are aptly called *dynamic SQL* statements.

Unlike static SQL statements, dynamic SQL statements are not embedded in your source program. Instead, they are stored in character strings input to or built by the program at run time. They can be entered interactively or read from a file.

Advantages and Disadvantages of Dynamic SQL

Host programs that accept and process dynamically defined SQL statements are more versatile than plain embedded SQL programs. Dynamic SQL statements can be built interactively with input from users having little or no knowledge of SQL.

For example, your program might simply prompt users for a search condition to be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. A more complex program might allow users to choose from menus listing SQL operations, table and view names, column names, and so on. Thus, dynamic SQL lets you write highly flexible applications.

However, some dynamic queries require complex coding, the use of special data structures, and more runtime processing. While you might not notice the added processing time, you might find the coding difficult unless you fully understand dynamic SQL concepts and methods.

When to Use Dynamic SQL

In practice, static SQL will meet nearly all your programming needs. Use dynamic SQL only if you need its open-ended flexibility. Its use is suggested when one of the following items is unknown at precompile time:

- text of the SQL statement (commands, clauses, and so on)
- the number of host variables
- the datatypes of host variables
- references to database objects such as columns, indexes, sequences, tables, usernames, and views

Requirements for Dynamic SQL Statements

To represent a dynamic SQL statement, a character string must contain the text of a valid SQL statement, but *not* contain the EXEC SQL clause, or the statement terminator, or any of the following embedded SQL commands:

- CLOSE
- DECLARE
- DESCRIBE

- EXECUTE
- FETCH
- INCLUDE
- OPEN
- PREPARE
- WHENEVER

In most cases, the character string can contain *dummy* host variables. They hold places in the SQL statement for actual host variables. Because dummy host variables are just placeholders, you do not declare them and can name them anything you like. For example, Oracle makes no distinction between the following two strings:

```
'DELETE FROM EMP WHERE MGR = :mgr_number AND JOB = :job_title'
```

```
'DELETE FROM EMP WHERE MGR = :m AND JOB = :j'
```

How Dynamic SQL Statements Are Processed

Typically, an application program prompts the user for the text of a SQL statement and the values of host variables used in the statement. Then Oracle *parses* the SQL statement. That is, Oracle examines the SQL statement to make sure it follows syntax rules and refers to valid database objects. Parsing also involves checking database access rights, reserving needed resources, and finding the optimal access path.

Next, Oracle *binds* the host variables to the SQL statement. That is, Oracle gets the addresses of the host variables so that it can read or write their values.

Then Oracle *executes* the SQL statement. That is, Oracle does what the SQL statement requested, such as deleting rows from a table.

The SQL statement can be executed repeatedly using new values for the host variables.

Methods for Using Dynamic SQL

This section introduces four methods you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, then offers guidelines for choosing

the right method. Later sections show you how to use the methods, and include sample programs that you can study.

The four methods are increasingly general. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods 1 and 2, and so on. However, each method is most useful for handling a certain kind of SQL statement, as the following table shows:

Method	Kind of SQL Statement
1	non query without host variables
2	non query with known number of input host variables
3	query with known number of select-list items and input host variables
4	query with unknown number of select-list items or input host variables

Note: The term *select-list item* includes column names and expressions such as SAL * 1.10 and MAX(SAL).

Method 1

This method lets your program accept or build a dynamic SQL statement, then immediately execute it using the EXECUTE IMMEDIATE command. The SQL statement must not be a query (SELECT statement) and must not contain any placeholders for input host variables. For example, the following host strings qualify:

```
'DELETE FROM EMP WHERE DEPTNO = 20'
```

```
'GRANT SELECT ON EMP TO scott'
```

With Method 1, the SQL statement is parsed every time it is executed.

Method 2

This method lets your program accept or build a dynamic SQL statement, then process it using the PREPARE and EXECUTE commands. The SQL statement must not be a query. The number of placeholders for input host variables and the datatypes of the input host variables must be known at precompile time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (ENAME, JOB) VALUES (:emp_name, :job_title)'
```

```
'DELETE FROM EMP WHERE EMPNO = :emp_number'
```

With Method 2, the SQL statement is parsed just once, but can be executed many times with different values for the host variables. SQL data definition statements such as CREATE and GRANT are executed when they are PREPARED.

Method 3

This method lets your program accept or build a dynamic query, then process it using the PREPARE command with the DECLARE, OPEN, FETCH, and CLOSE cursor commands. The number of select-list items, the number of placeholders for input host variables, and the datatypes of the input host variables must be known at precompile time. For example, the following host strings qualify:

```
'SELECT DEPTNO, MIN(SAL), MAX(SAL) FROM EMP GROUP BY DEPTNO'
```

```
'SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :dept_number'
```

Method 4

This method lets your program accept or build a dynamic SQL statement, then process it using descriptors ("Using Method 4"). The number of select-list items, the number of placeholders for input host variables, and the datatypes of the input host variables can be unknown until run time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (<unknown>) VALUES (<unknown>)'
```

```
'SELECT <unknown> FROM EMP WHERE DEPTNO = 20'
```

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or input host variables.

Guidelines

With all four methods, you must store the dynamic SQL statement in a character string, which must be a host variable or quoted literal. When you store the SQL statement in the string, omit the keywords EXEC SQL and the ';' statement terminator.

With Methods 2 and 3, the number of placeholders for input host variables and the datatypes of the input host variables must be known at precompile time.

Each succeeding method imposes fewer constraints on your application, but is more difficult to code. As a rule, use the simplest method you can. However, if a dynamic SQL statement will be executed repeatedly by Method 1, use Method 2 instead to avoid reparsing for each execution.

Method 4 provides maximum flexibility, but requires complex coding and a full understanding of dynamic SQL concepts. In general, use Method 4 only if you cannot use Methods 1, 2, or 3.

Avoiding Common Errors

If you precompile using the command-line option DBMS=V6 or DBMS=V6_CHAR, blank-pad the array before storing the SQL statement. That way, you clear extraneous characters. This is especially important when you reuse the array for different SQL statements. As a rule, always initialize (or re-initialize) the host string before storing the SQL statement. Do *not* null-terminate the host string. Oracle does not recognize the null terminator as an end-of-string sentinel. Instead, Oracle treats it as part of the SQL statement.

If you precompile with the command-line option DBMS=V7 (the default), make sure that the string is null terminated before you execute the PREPARE or EXECUTE IMMEDIATE statement.

Regardless of the value of DBMS, if you use a VARCHAR variable to store the dynamic SQL statement, make sure the length of the VARCHAR is set (or reset) correctly before you execute the PREPARE or EXECUTE IMMEDIATE statement.

18. Elaborate the concept of Advanced SQL

Built-in Data Types in SQL

- **date:**Dates, containing a (4 digit) year, month and date

Example: **date**'2005-7-27'

- **time:**Time of day, in hours, minutes and seconds.

Example: **time**'09:00:30'**time**'09:00:30.75'

- **timestamp:** date plus time of day

Example: **timestamp**'2005-7-27 09:00:30.75'

- **interval:**period of time

Example: interval '1'day

Subtracting a date/time/timestamp value from another gives an interval value

Interval values can be added to date/time/timestamp values

Can extract values of individual fields from date/time/timestamp

Example: **extract(year from**r.starttime)

Can cast string types to date/time/timestamp

Example: **cast**<string-valued-expression> **as date**

Example: **cast**<string-valued-expression> **as time**

create domain*Dollars***numeric**(12, 2)

create domain*Pounds***numeric**(12,2)

- We cannot assign or compare a value of type Dollars to a value of type Pounds.
- However, we can convert type as below

(**cast***r.AasPounds*)

(Should also multiply by the dollar-to-pound conversion-rate)

Large Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
- **blob**: binary large object --object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- **clob**: character large object --object is a large collection of character data

When a query returns a large object, a pointer is returned rather than the large object itself.

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- A checking account must have a balance greater than \$10,000.00
- A salary of a bank employee must be at least \$4.00 an hour
- A customer must have a (non-null) phone number

Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check (P)**, where *P* is a predicate

- **Not Null Constraint**

Declare *branch_name* for *branch* is **not null**

branch_name **char**(15) **not null**

Declare the domain *Dollars* to be **not null**

create domain*Dollars* **numeric**(12,2)**not null**

The Unique Constraint

- **unique**(*A1, A2, ..., Am*)

The unique specification states that the attributes *A1, A2, ..., Am* form a candidate key. Candidate keys are permitted to be non null (in contrast to primary keys).

The check clause

- **check** (*P*), where *P* is a predicate

Example: Declare *branch_name* as the primary key for *branch* and ensure that the values of *assets* are non-negative. **create table** *branch*(*branch_name* **char**(15), *branch_city* **char**(30), *assets* **integer**, **primary key** (*branch_name*), **check**(*assets* >= 0))

- The **check** clause in SQL-92 permits domains to be restricted:
- Use **check** clause to ensure that an *hourly_wage* domain allows only values greater than a specified value.

create domain *hourly_wage* **numeric**(5,2)

constraint *value_test* **check**(*value* >= 4.00)

- The domain has a constraint that ensures that the *hourly_wage* is greater than 4.00
- The clause **constraint** *value_test* is optional; useful to indicate which constraint an update violated.

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
- Example: If "Perryridge" is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch "Perryridge".
- Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
- The primary key clause lists attributes that comprise the primary key.
- The unique key clause lists attributes that comprise a candidate key.
- The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

Referential Integrity in SQL – Example

create table *customer*

(*customer_name* **char**(20), *customer_street* **char**(30), *customer_city* **char**(30),

primary key(*customer_name*)

create table *branch*(*branch_name***char**(15),*branch_city***char**(30),*assets***numeric**(12,2),
primary key(*branch_name*)

create table*account*

(*account_number***char**(10),*branch_name***char**(15),*balance***integer**,**primary key**
(*account_number*),**foreign key**(*branch_name*)**references** *branch*)

create table *depositor*(*customer_name***char**(20),*account_number***char**(10),**primary**
key(*customer_name*, *account_number*),**foreign key**(*account_number*)**references** *account*,
foreign key(*customer_name*)**references** *customer*)

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- An assertion in SQL takes the form

create assertion <assertion-name> **check**<predicate>

- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
- This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- Asserting for all X , $P(X)$ is achieved in a round-about fashion using not exists X such that not $P(X)$

Assertion Example

- Every loan has at least one borrower who maintains an account with a minimum balance or \$1000.00

create assertion *balance_constraint* **check**(**not exists** (**select * from** *loan* **where not exists** (**select * from** *borrower*, *depositor*, *account* **where** *loan.loan_number* = *borrower.loan_number* **and** *borrower.customer_name* = *depositor.customer_name* **and** *depositor.account_number* = *account.account_number* **and** *account.balance* >= 1000)))

Assertion Example

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

create assertion *sum_constraint* **check**(**not exists** (**select * from** *branch* **where** (**select** **sum**(*amount*) **from** *loan* **where** *loan.branch_name* = *branch.branch_name*)

>= (select sum (amount)from account where loan.branch_name =branch.branch_name)))

Authorization

Forms of authorization on parts of the database:

- **Read** -allows reading, but not modification of data.
- **Insert** -allows insertion of new data, but not modification of existing data.
- **Update** -allows modification, but not deletion of data.
- **Delete** -allows deletion of data.

Forms of authorization to modify the database schema (covered in Chapter 8):

- **Index** -allows creation and deletion of indices.
- **Resources** -allows creation of new relations.
- **Alteration** -allows addition or deletion of attributes in a relation.
- **Drop** -allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

grant<privilege list>**on** <relation name or view name> **to**<user list>

<user list> is: a user-id

- **public**, which allows all valid users the privilege granted

A role Granting a privilege on a view does not imply granting any privileges on the underlying relations.

The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view

Example: grant users *U1*, *U2*, and *U3* **select** authorization on the *branch* relation:

grant select on branch to U1, U2, U3

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

The **revoke** statement is used to revoke authorization.

revoke <privilege list>**on** <relation name or view name> **from** <user list>

Example:

revoke select on branch from U1, U2, U3<privilege-list> may be **all** to revoke all privileges the revoke may hold.

If <revokee-list> includes **public**, all users lose the privilege except those granted it explicitly. If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

All privileges that depend on the privilege being revoked are also revoked.

19. Table: EMP Emp(empno, ename, job, mgr, hiredate, salary, comm., deptno)

a. List all the employees who have at least one person reporting to them.

SELECT DISTINCT(A.ENAME) FROM EMP A, EMP B WHERE A.EMPNO = B.MGR; or SELECT ENAME FROM EMP WHERE EMPNO IN (SELECT MGR FROM EMP);

b. List the employee details if and only if more than 10 employees are present in department no 10.

SELECT * FROM EMP WHERE DEPTNO IN (SELECT DEPTNO FROM EMP GROUP BY DEPTNO HAVING COUNT(EMPNO)>10 AND DEPTNO=10);

c. List the name of the employees with their immediate higher authority.

SELECT A.ENAME "EMPLOYEE", B.ENAME "REPORTS TO" FROM EMP A, EMP B WHERE A.MGR=B.EMPNO;

d. List all the employees who do not manage any one.

SELECT * FROM EMP WHERE EMPNO IN (SELECT EMPNO FROM EMP MINUS SELECT MGR FROM EMP);

e. List the employee details whose salary is greater than the lowest salary of an employee belonging to deptno 20.

SELECT * FROM EMP WHERE SAL > (SELECT MIN(SAL) FROM EMP GROUP BY DEPTNO HAVING DEPTNO=20);

20. Discuss with examples various operations of Relation Algebra Operations.

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

The operations can be divided into,

- **Basic operations:** Select, Project, Union, rename, set difference and Cartesian product
- **Additional operations:** Set intersections, natural join, division and assignment.
- **Extended operations:** Aggregate operations and outer join

The fundamental operations are :

1. Union
2. Select
3. Project
4. Set difference
5. Cartesian product and Rename.

For Example:

UNION OPERATION

Consider a query to find the names of all bank customers who have either an account or a loan or both. Note that the customer relation does not contain the information, since a customer does not need to have either an account or a loan at the bank.

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

$\Pi_{\text{customer-name}}(\text{borrower})$

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

$\Pi_{\text{customer-name}}(\text{depositor})$

The binary operation union, denoted, as in set theory, by \cup . So the expression needed is **$\Pi_{\text{customer-name}}(\text{borrower}) \cup \Pi_{\text{customer-name}}(\text{depositor})$** . Therefore, for a union operation $r \cup s$ to be valid, we require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.
2. The domains of the i^{th} attribute of r and the i^{th} attribute of s must be the same, for all i .

UNIT II

Database Design and the ER Model: Overview of the Design Process – The Entity – Relational Model – Constraints – Removing Redundant Attributes – Entity – Relationship Design Issues – Extended E-R Features – Alternative Notations for Modeling data – Other Aspects of Database Design – Storage and File structure – Indexing and Hashing – Basic Concepts – Ordered Indices – B+ - Tree Index Files – Static Hashing – Dynamic Hashing – Comparison of Ordered Indexing and Hashing – Bitmap Indices – Index Definition in SQL.

Unit II- Two Marks

1. What is an entity relationship model?

The entity relationship model is a collection of basic objects called entities and relationship among those objects. An entity is a thing or object in the real world that is distinguishable from other objects.

2. What are attributes? Give examples.

An entity is represented by a set of attributes. Attributes are descriptive properties possessed by each member of an entity set.

Example: possible attributes of customer entity are customer name, customer id, customer street, customer city.

3. What is relationship? Give examples

A relationship is an association among several entities.

Example: A depositor relationship associates a customer with each account that he/she has.

4. Define Entity, Entity Set, and extensions of entity set. Give one example for each.(NOV 2012)

- Entity – object or thing in the real world. Egs., each person, book, etc.
- Entity set – set of entities of the same entity that share the same properties or attributes. Eg., Instructor, Department, section, etc
- Extensions of entity set – individual entities that constitute a set. Eg., individual bank customers

5. Define and give examples to illustrate the four types of attributes in database. (NOV 2010)

Simple and Composite attribute

- Simple (address)
- Composite (street, city, district)

Single-valued and multi-valued attribute

- Single-valued (roll no)
- Multi-valued (colors = {R,B,G})

Null attribute

- No values for attribute

Derived attribute

- Age derived from Birth date
- Single-valued and multi-valued attribute
 - Single-valued (roll no)
 - Multi-valued (colors = {R,B,G})
- Null attribute
 - No values for attribute
- Derived attribute
 - Age derived from Birth date

6. Define the terms

i) Entity type & ii) Entity set

Entity type: An entity type defines a collection of entities that have the same attributes.

Entity set: The set of all entities of the same type is termed as an entity set.

7. What is meant by the degree of relationship set?

The degree of relationship type is the number of participating entity types.

8. Define the terms

i) Key attribute

ii) Value set

Key attribute: An entity type usually has an attribute whose values are distinct from each individual entity in the collection. Such an attribute is called a key attribute.

Value set: Each simple attribute of an entity type is associated with a value set that specifies the set of values that may be assigned to that attribute for each individual entity.

9. Define relationship and participation.

- Relationship is an association among several entities.
- Association between entity sets is referred as participation.

10. Define mapping cardinality or cardinality ratio.

Mapping cardinality or cardinality ratio is the way to express the number of entities to which another entity can be associated through a relationship set. These are most useful in describing binary relationship sets.

11. Explain the four types of mapping cardinality with example.

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

- One-to-one

An entity in A is associated with at most one entity in B and an entity in B is associated with at most one entity in A. Eg., Roll no entity in Student info entity set and marks details entity set.

- One-to-many

An entity in A is associated with any number of entities in B. But an entity in B is associated with at most one entity in A. Eg., one customer with many loans.

- Many-to-one

An entity in A is associated with at most one entity in B. But an entity in B can be associated with any number of entities in A. Eg., street and city associated to a single person.

- Many-to-many

An entity in A is associated with any number of entities in B. But an entity in B can be associated with any number of entities in A. Eg., same loan by several business partners.

12. Differentiate total participation and partial participation. (i.e., write Definition and Example with illustration)

- Total – participation of an entity set, E in relationship, R is total if every entity in E participates in at least one relationship in R. Eg., loan entity set.
- Partial – if only some entities in E participate in R. Eg., payment weak entity set.

13. Define E-R diagram.

Overall structure of a database can be expressed graphically by E-R diagram for simplicity and clarity.

14. Define weak Entity set. Give an example and explain why it is weak entity set.

Entity set with no sufficient attributes to form a primary key.

- Payment entity set is weak since duplication exists

15. Define discriminator or partial key of a weak entity set. Give example.

Set of attributes that allow distinction to be made among all those entities in the entity set that depend on one particular strong entity. Eg., payment_no in payment entity set. It is also called as partial key.

16. Define strong and weak entity sets? **NOV 2014**

An entity set may not have sufficient attributes to form a primary key. Such an entity set is termed a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.

17. Define the term Canonical Form? APR 2014

Sets of functional dependencies may have redundant dependencies that can be inferred from the others

- For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, B \rightarrow C\}$
- Parts of a functional dependency may be redundant
- E.g.: on RHS: $\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$ can be simplified to $\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$
- E.g.: on LHS: $\{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$ can be simplified to $\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$

18. What is E-R modeling? APR 2015

Entity-Relationship model (ER model) is a data model for describing the data or information aspects of a business domain or its process requirements, in an abstract way that lends itself to ultimately being implemented in a database such as a relational database. The main components of ER models are entities (things) and the relationships that can exist among them.

19. What is logical data independence?

Logical data is data about database, that is, it stores information about how data is managed inside. For example, a table (relation) stored in the database and all its constraints, applied on that relation.

Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If any changes on table format, it should not change the data residing on the disk.

20. What are the types of indices? (April 2014)

Ordered indices: Based on a sorted ordering of the values.

Hash indices: Based on a uniform distribution of values across a range of bucket. The bucket to which a value is assigned is determined by a function, called a hash function.

21. What are the techniques of indexing & hashing?(Nov 2014)

- Access types
- Access time
- Insertion time
- Deletion time
- Space overhead

22. What is dense index?

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

23. What is sparse index?

An index record appears for only some of the search key values in a file. Such a

24. What is a multilevel index?

Indices with two or more levels are called multilevel indices, multilevel indices are closely related to the tree structures such as binary tree used for memory indexing.

25.What are B+ tree index files?

A B+ tree index takes the form of a balanced tree in which every path from the root of the tree to the leaf of the tree is of the same length.

26.What is the use of static hashing?

To access a file in sequential organization binary search or indexing is used, this creates more i/o operations to avoid this static hashing is used.

27.What is a hash function?

A hash functions distributes the stored key uniformly across all the buckets, so that every bucket has the same number of records.

28.What is skew?

Some buckets are assigned more records than the others, so a bucket may over flow even when other buckets still have space. This situation is called **bucket skew**

29.What are the two reasons for the occurrence of skew?

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

30.What is a hash index?

A hash index organizes the search keys, with their associated pointers into a hash file structure.

31.Define dynamic hashing?

Dynamic hashing techniques allow the hash function to be modified dynamically to accommodate the growth are shrinkage of the database, one form of the dynamic hashing is called **extendable hashing**.

32.What is a bitmap indices?

Bitmap indices are a specialized type of index designed for easy querying on multiple keys, although each bitmap index is built on a single key.

33.Define covering indices?

Covering indices are the indices that store the values of some attributes along with the pointers to the record. Storing extra attribute values is useful with secondary indices, since they allow us to answer some queries using just the index, without even looking up the actual records.

34.What is Hashing? (April 2013)

Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value. It is also used in many encryption algorithms.

35.List out the different types of indices.(April 2011),NOV 2015

Ordered indices: Based on a sorted ordering of the values.

Hash indices: Based on a uniform distribution of values across a range of bucket. The bucket to which a value is assigned is determined by a function, called a hash function

36.What is B+ tree? (APRIL 2015)

A dynamic data structure which adjusts efficiently under inserts and deletes. data entries are stored at the leaf level. Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.

37.Give example for Composite Attribute.(NOV 2015)

Abdul Kadhira Shahid is the name of a student. Here we can combine, three names to a single value.

38.How primary indices differ from secondary indices. .(NOV 2015)

Primary index:

A primary index is an index on a set of fields that includes the unique primary key for the field and is guaranteed not to contain duplicates. Also Called a **Clustered index**. eg. Employee ID can be Example of it.

Secondary index:

A Secondary index is an index that is not a primary index and may have duplicates. eg. Employee name can be example of it. Because Employee name can have similar values.

Unit II- Eleven Marks

1. Explain in detail about Entity Set? (11 Marks)

An **entity** is a “thing” or “object” in the real world that is distinguishable from all other objects. For example, each person in an enterprise is an entity. An entity has a set of properties, and the values for some set of properties may uniquely identify an entity. An entity may be concrete, such as a person or a book, or it may be abstract, such as a loan, or a holiday, or a concept.

An **entity set** is a set of entities of the same type that share the same properties, or attributes. The set of all persons who are customers at a given bank, for example, can be defined as the entity set customer. An entity is represented by a set of **attributes**. Attributes are descriptive properties possessed by each member of an entity set. The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the customer entity set are customer-id, customer-name, Customer Street, and customer-city. In real life, there would be further attributes, such as street number, apartment number, state, postal code, and country etc.

Each entity has a **value** for each of its attributes. For instance, a particular customer entity may have the value 321-12-3123 for customer-id, the value Jones for customer name, the value Main for customer-street, and the value Harrison for customer-city. The customer-id attribute is used to uniquely identify customers, since there may be more than one customer with the same name. For each attribute, there is a set of permitted values, called the **domain**, or **value set**, of that attribute. The domain of attribute customer-name might be the set of all text strings of a certain length.

A database thus includes a collection of entity sets, each of which contains any number of entities of the same type. An attribute, as used in the E-R model, can be characterized by the following attribute

types:

- **Simple** and **composite** attributes. In our examples thus far, the attributes have been simple; that is, they are not divided into subparts. **Composite** attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute name could be structured as a composite attribute consisting of first-name, middle-initial, and last-name.
- **Single-valued** and **multivalued** attributes. The attributes in our examples all have a single value for a particular entity. For instance, the loan-number attribute for a specific loan entity refers to only one loan number. Such attribute are said to be **single valued**.

- An employee may have zero, one, or several phone numbers, and different employees may have different numbers of phones. This type of attribute is said to be **multivalued**.
- **Derived** attribute. The value for this type of attribute can be derived from the values of other related attributes or entities. For instance, let us say that the customer entity set has an attribute loans-held, which represents how many loans a customer has from the bank. We can derive the value for this attribute by counting the number of loan entities associated with that customer.

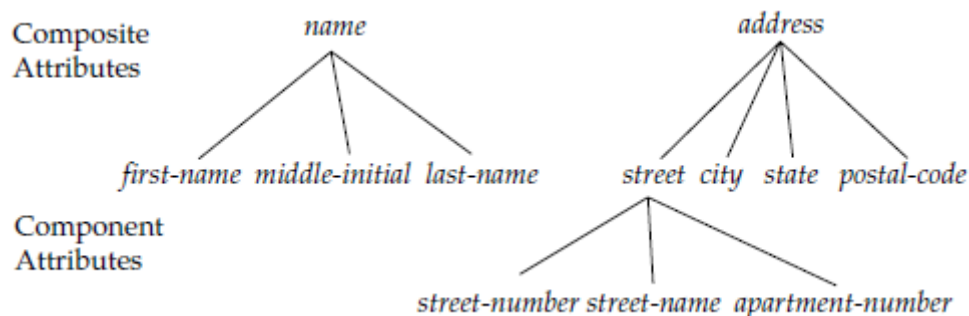
An attribute takes a **null** value when an entity does not have a value for it. The null value may indicate “not applicable”—that is, that the value does not exist for the entity. For example, one may have no middle name. Null can also designate that an attribute value is unknown. An unknown value may be either missing (the value does exist, but we do not have that information) or not known (we do not know whether or not the value actually exists).

321-12-3123	Jones	Main	Harrison
019-28-3746	Smith	North	Rye
677-89-9011	Hayes	Main	Harrison
555-55-5555	Jackson	Dupont	Woodside
244-66-8800	Curry	North	Rye
963-96-3963	Williams	Nassau	Princeton
335-57-7991	Adams	Spring	Pittsfield

L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-19	500
L-11	900
L-16	1300

customer

loan



2. Explain relationship set? (11 Marks)

A **relationship** is an association among several entities. For example, we can define a relationship that associates customer Hayes with loan L-15. This relationship specifies that Hayes is a customer with loan number L-15.

A **relationship set** is a set of relationships of the same type. Formally, it is a mathematical relation on $n \geq 2$ (possibly nondistinct) entity sets. If E_1, E_2, \dots, E_n are entity sets, then a relationship set R is a subset of

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

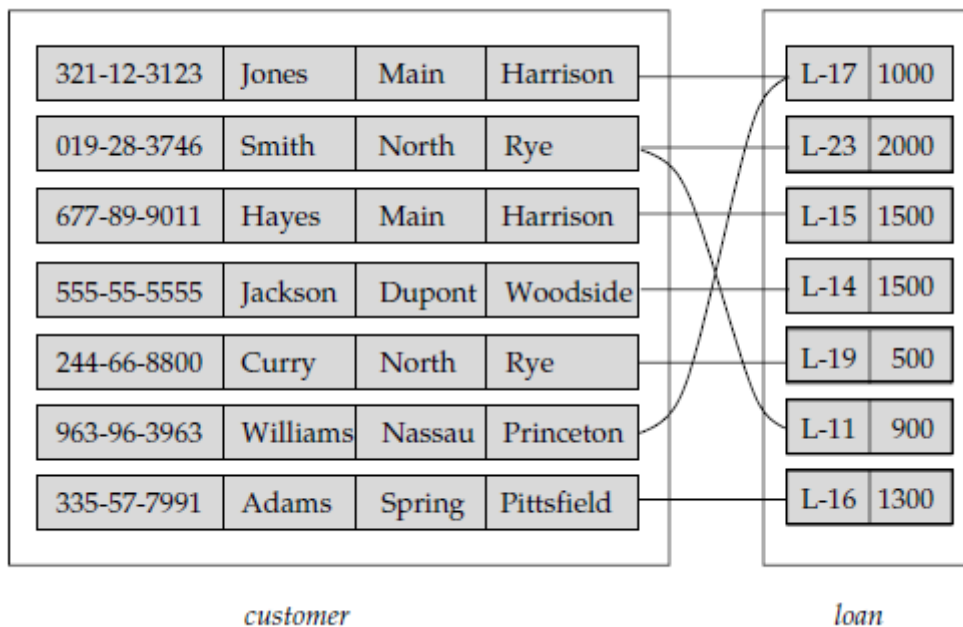
where (e_1, e_2, \dots, e_n) is a relationship.

As another example, consider the two entity sets loan and branch. We can define the relationship set loan-branch to denote the association between a bank loan and the branch in which that loan is maintained.

The association between entity sets is referred to as participation; that is, the entity sets E_1, E_2, \dots, E_n **participate** in relationship set R . A **relationship instance** in an E-R schema represents an association between the named entities.

The function that an entity plays in a relationship is called that entity's **role**. Since entity sets participating in a relationship set are generally distinct, roles are implicit and are not usually specified. The same entity set participates in a relationship set more than once, in different roles. In this type of relationship set, sometimes called a **recursive** relationship set, explicit role names are necessary to specify how an entity participates in a relationship instance. A relationship may also have attributes called **descriptive attributes**. Consider a relationship set depositor with entity sets customer and account. However, there can be more than one relationship set involving the same entity sets. In our example the customer and loan entity sets participate in the relationship set borrower. Additionally, suppose each loan must have another customer who serves as a guarantor for the loan. Then the customer and loan entity sets may participate in another relationship set, guarantor.

The relationship sets borrower and loan-branch provide an example of a **binary** relationship set that is, one that involves two entity sets. Most of the relationship sets in a database system are binary. The number of entity sets that participate in a relationship set is also the **degree** of the relationship set. A binary relationship set is of degree 2; a ternary relationship set is of degree 3.



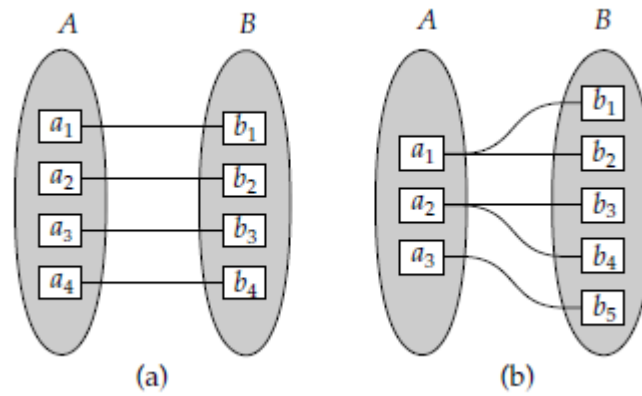
3.Explain in detail about constraints and keys? (11 Marks)

a) Mapping Cardinalities (TWO MARKS NOV 2012)

Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set. Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets. In this section, we shall concentrate on only binary relationship sets.

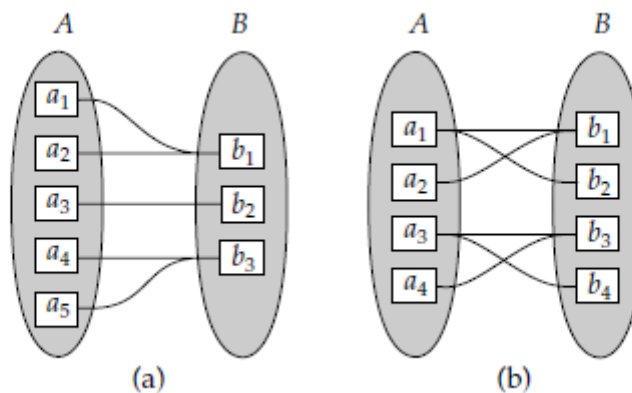
For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:

- **One to one.** An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A. (See Figure 2.4a.)
- **One to many.** An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with at most one entity in A. (See Figure 2.4b.)
- **Many to one.** An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A. (See Figure 2.5a.)
- **Many to many.** An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A. (See Figure 2.5b.)



b) Participation Constraints

The participation of an entity set E in a relationship set R is said to be **total** if every entity in E participates in at least one relationship in R. If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be **partial**. For example, we expect every loan entity to be related to at least one customer through the borrower relationship. Therefore the participation of loan in the relationship set borrower is total. In contrast, an individual can be a bank customer whether or not she has a loan with the bank. Hence, it is possible that only some of the customer entities are related to the loan entity set through the borrower relationship, and the participation of customer in the borrower relationship set is therefore partial.



Keys

A key allows us to identify a set of attributes that suffice to distinguish entities from each other. Keys also help uniquely identify relationships, and thus distinguish relationships from each other.

a) Entity Sets

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely an entity in the entity set. For example, the customer-id attribute of the entity set customer is sufficient to distinguish one customer entity from another. Thus, customer-id is a superkey. Similarly, the combination of customer-name and customer-id is a superkey for the entity set customer. The customer-name attribute of customer is not a superkey, because several people

might have the same name. The concept of a superkey is not sufficient for our purposes, since, as we saw, a superkey may contain extraneous attributes. If K is a superkey, then so is any superset of K . We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**. It is possible that several distinct sets of attributes could serve as a candidate key.

Although the attributes `customerid` and `customer-name` together can distinguish customer entities, their combination does not form a candidate key, since the attribute `customer-id` alone is a candidate key. We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set. A key (primary, candidate, and super) is a property of the entity set, rather than of the individual entities. Any two individual entities in the set are prohibited from having the same value on the key attributes at the same time.

The primary key should be chosen such that its attributes are never, or very rarely, changed. For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed to never change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

b) Relationship sets

The primary key of an entity set allows us to distinguish among the various entities of the set. Let R be a relationship set involving entity sets E_1, E_2, \dots, E_n . Let $\text{primary-key}(E_i)$ denote the set of attributes that forms the primary key for entity set E_i . Assume for now that the attribute names of all primary keys are unique, and each entity set participates only once in the relationship. The composition of the primary key for a relationship set depends on the set of attributes associated with the relationship set R .

If the relationship set R has no attributes associated with it, then the set of attributes

$\text{primary-key}(E_1) \cup \text{primary-key}(E_2) \cup \dots \cup \text{primary-key}(E_n)$

describes an individual relationship in set R .

If the relationship set R has attributes a_1, a_2, \dots, a_m associated with it, then the set of attributes

$\text{primary-key}(E_1) \cup \text{primary-key}(E_2) \cup \dots \cup \text{primary-key}(E_n) \cup \{a_1, a_2, \dots, a_m\}$

describes an individual relationship in set R .

In both of the above cases, the set of attributes

primary-key(E1) \cup primary-key(E2) $\cup \dots \cup$ primary-key(En)

forms a superkey for the relationship set.

In case the attribute names of primary keys are not unique across entity sets, the attributes are renamed to distinguish them; the name of the entity set combined with the name of the attribute would form a unique name. The role name is used instead of the name of the entity set, to form a unique attribute name. The structure of the primary key for the relationship set depends on the mapping cardinality of the relationship set.

4. Explain in detail about design issues? (11 Marks)

a) Use of Entity Sets versus Attributes

Consider the entity set employee with attributes employee-name and telephone-number. We must redefine the employee entity set as:

- The employee entity set with attribute employee-name
- The telephone entity set with attributes telephone-number and location
- The relationship set emp-telephone, which denotes the association between employees and the telephones that they have.

Treating a telephone as an attribute telephone-number implies that employees have precisely one telephone number each. Treating a telephone as an entity telephone permits employees to have several telephone numbers (including zero) associated with them. However, we could instead easily define telephone-number as a multivalued attribute to allow multiple telephones per employee. The main difference then is that treating a telephone as an entity better models a situation where one may want to keep extra information about a telephone, such as its location, or its type (mobile, video phone, or plain old telephone), or who all share the telephone. Thus, treating telephone as an entity is more general than treating it as an attribute and is appropriate when the generality may be useful.

b) Use of Entity Sets versus Relationship Sets

We assumed that a bank loan is modeled as an entity. An alternative is to model a loan not as an entity, but rather as a relationship between customers and branches, with loan-number and amount as descriptive attributes. Each loan is represented by a relationship between a customer and a branch. If every loan is held by exactly one customer and is associated with exactly one branch, we may find satisfactory the design where a loan is represented as a relationship. However, with this design, we cannot represent conveniently a situation in which several customers hold a loan jointly. To handle such a situation, we must define a separate relationship

for each holder of the joint loan. Then, we must replicate the values for the descriptive attributes loan-number and amount in each such relationship. Each such relationship must, of course, have the same value for the descriptive attributes loan-number and amount. Two problems arise as a result of the replication: (1) the data are stored multiple times, wasting storage space, and (2) updates potentially leave the data in an inconsistent state, where the values differ in two relationships for attributes that are supposed to have the same value. The problem of replication of the attributes loan-number and amount is absent in the original design, because there loan is an entity set. One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

Binary versus n-ary Relationship Sets

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

We replace the relationship set R by an entity set E, and create three relationship sets:

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

If the relationship set R had any attributes, these are assigned to entity set E; further, a special identifying attribute is created for E (since it must be possible to distinguish different entities in an entity set on the basis of their attribute values). For each relationship (a_i, b_i, c_i) in the relationship set R, we create a new entity e_i in the entity set E. Then, in each of the three new relationship sets, we insert a relationship as follows:

- (e_i, a_i) in RA
- (e_i, b_i) in RB
- (e_i, c_i) in RC

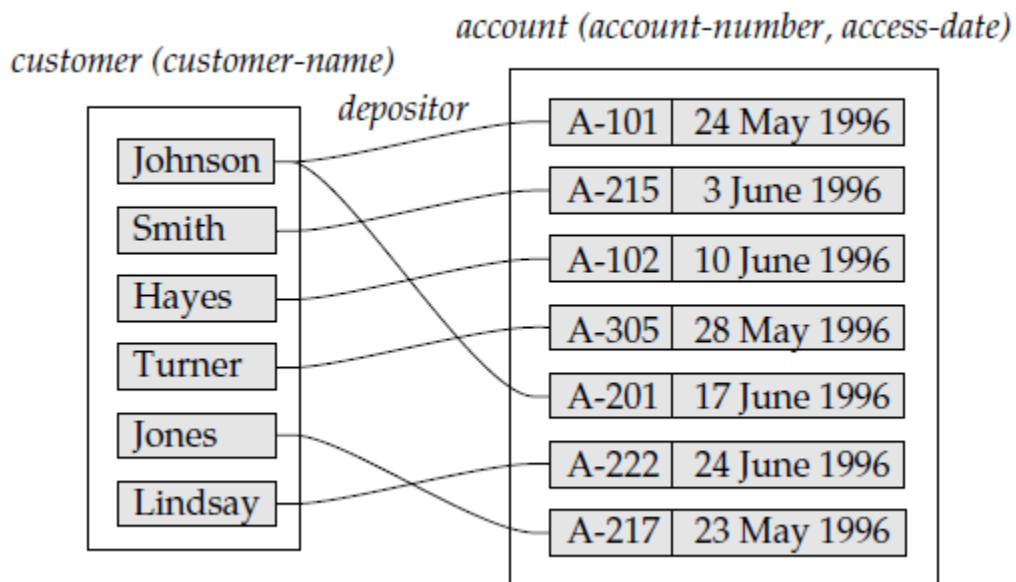
We can generalize this process in a straightforward manner to n-ary relationship sets. Thus, conceptually, we can restrict the E-R model to include only binary relationship sets. However, this restriction is not always desirable.

- An identifying attribute may have to be created for the entity set created to represent the relationship set. This attribute, along with the extra relationship sets required, increases the complexity of the design and overall storage requirements.
- A n-ary relationship set shows more clearly that several entities participate in a single relationship.
- There may not be a way to translate constraints on the ternary relationship into constraints on the binary relationships.

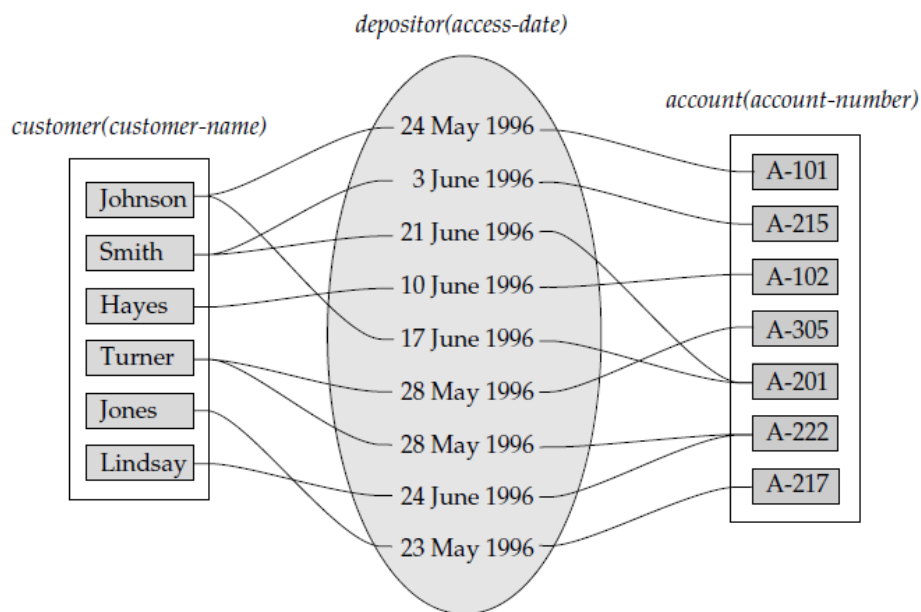
Placement of Relationship Attributes

The cardinality ratio of a relationship can affect the placement of relationship attributes. Thus, attributes of one-to-one or one-to-many relationship sets can be associated with one of the participating entity sets, rather than with the relationship set. For instance, let us specify that depositor is a one-to-many relationship set such that one customer may have several accounts, but each account is held by only one customer. In this case, the attribute access-date, which specifies when the customer last accessed that account, could be associated with the account entity set.

The choice of attribute placement is more clear-cut for many-to-many relationship sets. Example, let us specify the perhaps more realistic case that depositor is a many-to-many relationship set expressing that a customer may have one or more accounts, and that an account can be held by one or more customers. If we are to express the date on which a specific customer last accessed a specific account, access-date must be an attribute of the depositor relationship set, rather than either one of the participating entities. If access-date were an attribute of account, for instance, we could not determine which customer made the most recent access to a joint account. When an attribute is determined by the combination of participating entity sets, rather than by either entity separately, that attribute must be associated with the many-to-many relationship set.



Access-date as attribute of the account entity set.



Access-date as attribute of the depositor relationship set.

5. Explain Entity-Relationship Diagram? (11 Marks) **APRIL 2014, NOV 2015**

The following major components:

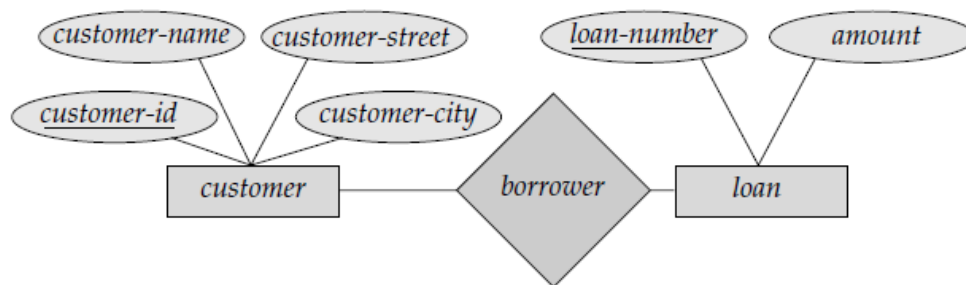
- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationship sets
- **Lines**, which link attributes to entity sets and entity sets to relationship sets
- **Double ellipses**, which represent multivalued attributes

- **Dashed ellipses**, which denote derived attributes
- **Double lines**, which indicate total participation of an entity in a relationship set
- **Double rectangles**, which represent weak entity

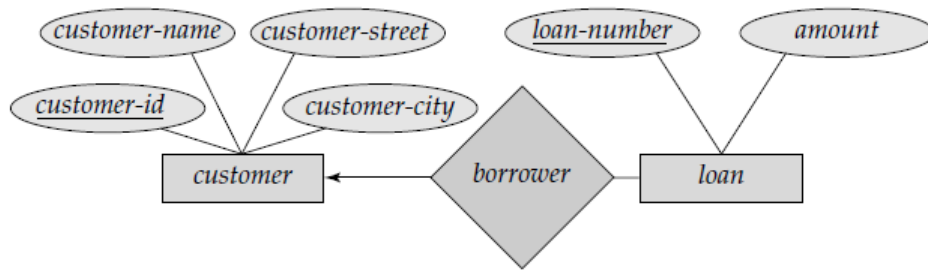
Consider the entity-relationship diagram which consists of two entity sets, customer and loan, related through a binary relationship set borrower. The attributes associated with customer are customer-id, customer-name, customer-street, and customer-city. The attributes associated with loan are loan-number and amount.

The relationship set borrower may be many-to-many, one-to-many, many-to-one, or one-to-one. To distinguish among these types, we draw either a directed line (\rightarrow) or an undirected line (—) between the relationship set and the entity set in question.

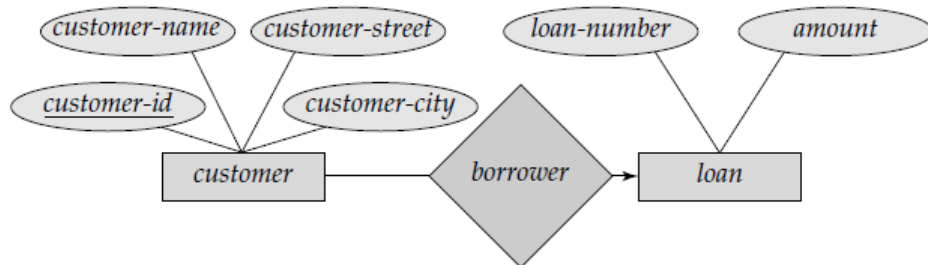
- A directed line from the relationship set borrower to the entity set loan specifies that borrower is either a one-to-one or many-to-one relationship set, from customer to loan; borrower cannot be a many-to-many or a one-to-many relationship set from customer to loan.
- An undirected line from the relationship set borrower to the entity set loan specifies that borrower is either a many-to-many or one-to-many relationship set from customer to loan.



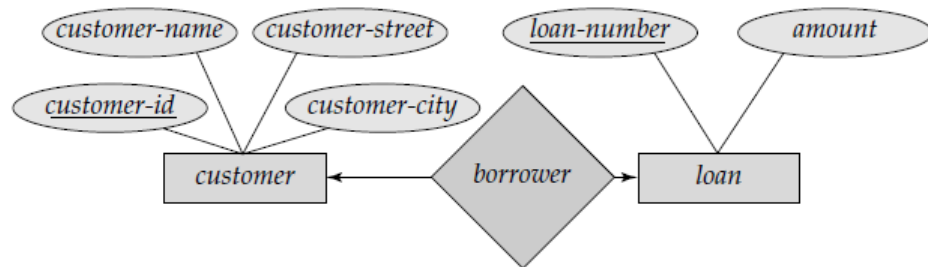
E-R diagram corresponding to customers and loans.



(a)

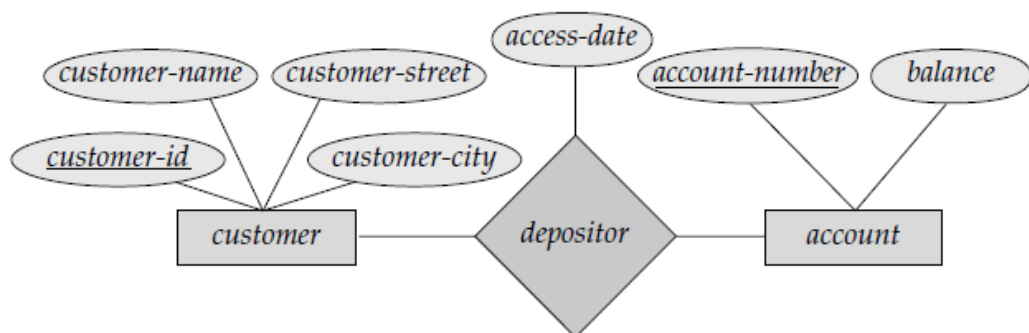


(b)



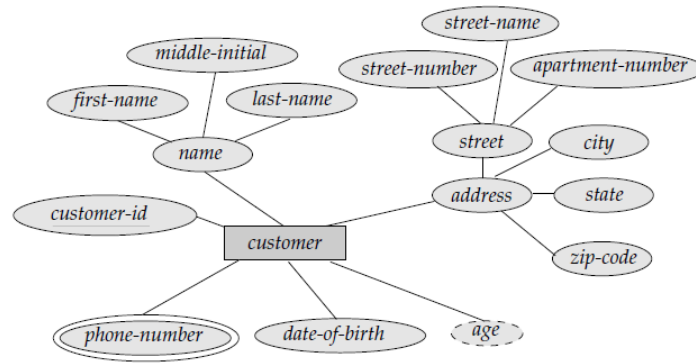
(c)

Relationships. (a) one to many. (b) many to one. (c) one-to-one.

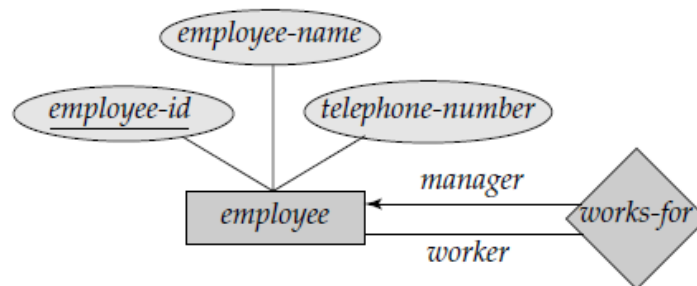


E-R diagram with an attribute attached to a relationship set.

A multivalued attribute phone-number, depicted by a double ellipse, and a derived attribute age, depicted by a dashed ellipse.



E-R diagram with composite, multivalued, and derived attributes.



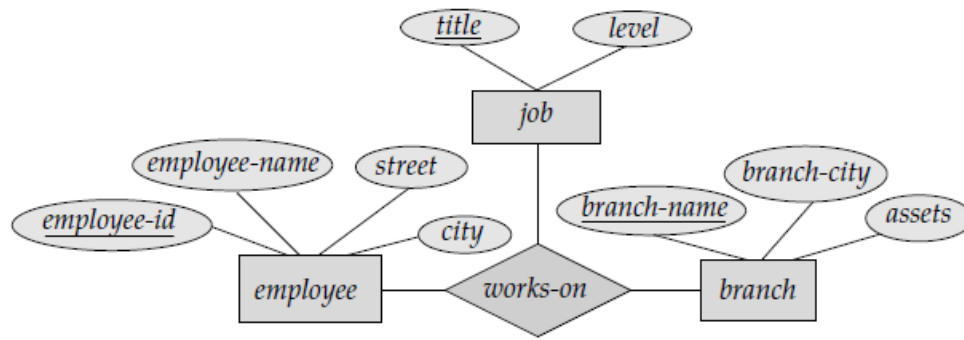
E-R diagram with role indicators

The role indicators manager and worker between the employee entity set and the works-for relationship set. Nonbinary relationship sets can be specified easily in an E-R diagram.

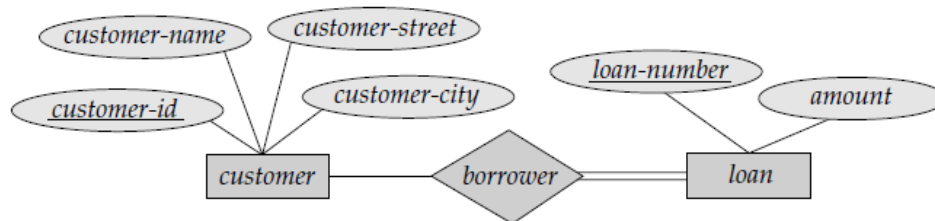
This consists of the three entity sets employee, job, and branch, related through the relationship set works-on.

Suppose there is a relationship set R between entity sets A_1, A_2, \dots, A_n , and the only arrows are on the edges to entity sets $A_{i+1}, A_{i+2}, \dots, A_n$. Then, the two possible interpretations are:

1. A particular combination of entities from A_1, A_2, \dots, A_i can be associated with at most one combination of entities from $A_{i+1}, A_{i+2}, \dots, A_n$. Thus, the primarykey for the relationship R can be constructed by the union of the primary keys of A_1, A_2, \dots, A_i .
2. For each entity set A_k , $i < k \leq n$, each combination of the entities from the other entity sets can be associated with at most one entity from A_k . Each set $\{A_1, A_2, \dots, A_{k-1}, A_{k+1}, \dots, A_n\}$, for $i < k \leq n$, then forms a candidate key.



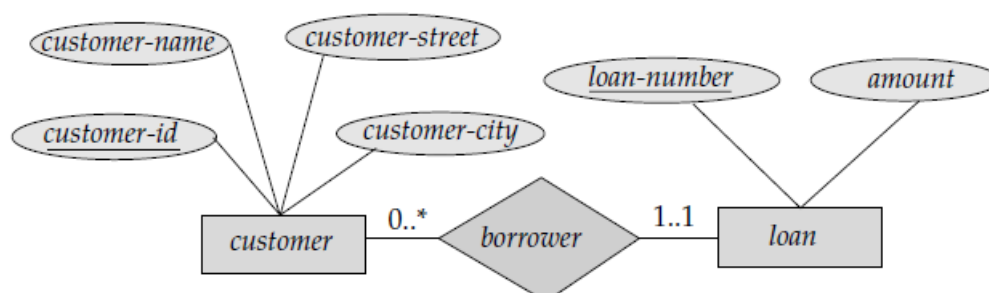
E-R diagram with a ternary relationship.



Total participation of an entity set in a relationship set.

Double lines are used in an E-R diagram to indicate that the participation of an entity set in a relationship set is total; that is, each entity in the entity set occurs in at least one relationship in that relationship set. For instance, consider the relationship borrower between customers and loans. A double line from loan to borrower, as in Figure 2.14, indicates that each loan must have at least one associated customer.

It is easy to misinterpret the 0..* on the edge between customer and borrower, and think that the relationship borrower is many to one from customer to loan—this is exactly the reverse of the correct interpretation. If both edges from a binary relationship have a maximum value of 1, the relationship is one to one. If we had specified a cardinality limit of 1..* on the edge between customer and borrower, we would be saying that each customer must have at least one loan.



Cardinality limits on relationship sets.

6. Explain Weak Entity Sets? (11 Marks)

An entity set may not have sufficient attributes to form a primary key. Such an entity set is termed a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.

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

For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying or owner entity set**. Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**. The identifying relationship is many to one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.

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

For example, the discriminator of the weak entity set payment is the attribute payment-number, since, for each loan, a payment number uniquely identifies one single payment for that loan. The discriminator of a weak entity set is also called the partial key of the entity set. The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator.

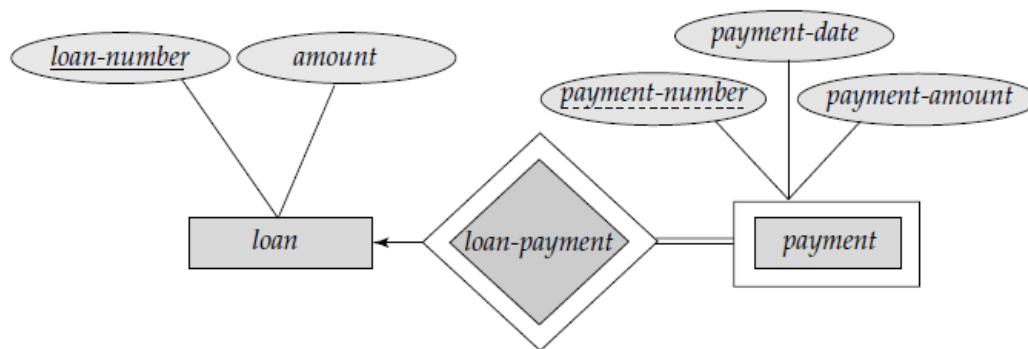
The identifying relationship set should have no descriptive attributes, since any required attributes can be associated with the weak entity set.

A weak entity set can participate in relationships other than the identifying relationship. For instance, the payment entity could participate in a relationship with the account entity set, identifying the account from which the payment was made. A weak entity set may participate as owner in an identifying relationship with another weak entity set. It is also possible to have a weak entity set with more than one identifying entity set. A particular weak entity would then be

identified by a combination of entities, one from each identifying entity set. The primary key of the weak entity set would consist of the union of the primary keys of the identifying entity sets, plus the discriminator of the weak entity set.

In E-R diagrams, a doubly outlined box indicates a weak entity set, and a doubly outlined diamond indicates the corresponding identifying relationship.

The database designer may choose to express a weak entity set as a multivalued composite attribute of the owner entity set. In our example, this alternative would require that the entity set loan have a multivalued, composite attribute payment, consisting of payment-number, payment-date, and payment amount. A weak entity set may be more appropriately modeled as an attribute if it participates in only the identifying relationship, and if it has few attributes.



E-R diagram with a weak entity set.

7. Explain specialization and generalization? (11 Marks)

Extended E-R Features

Specialization

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

- customer
- employee

Each of these person types is described by a set of attributes that includes all the attributes of entity set person plus possibly additional attributes. For example, customer entities may be

described further by the attribute customer-id, whereas employee entities may be described further by the attributes employee-id and salary. The process of designating subgroupings within an entity set is called **specialization**. The specialization of person allows us to distinguish among persons according to whether they are employees or customers.

The bank could then create two specializations of account, namely savings-account and checking account. The entity set savings-account would have all the attributes of account and an additional attribute interest-rate. The entity set checking account would have all the attributes of account, and an additional attribute overdraft amount.

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

- officer
- teller
- secretary

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

In terms of an E-R diagram, specialization is depicted by a triangle component labeled **ISA**. The label ISA stands for “is a” and represents, for example, that a customer “is a” person. The ISA relationship may also be referred to as a **superclass-subclass** relationship. Higher- and lower-level entity sets are depicted as regular entity set that is, as rectangles containing the name of the entity set.

Generalization

The refinement from an initial entity set into successive levels of entity sub groupings represents a **top down** design process in which distinctions are made explicit. The design process may also proceed in a **bottom-up** manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features. The database designer may have first identified a customer entity set with the attributes name, street, city, and customer-id, and an employee entity set with the attributes name, street, city, employee-id, and salary.

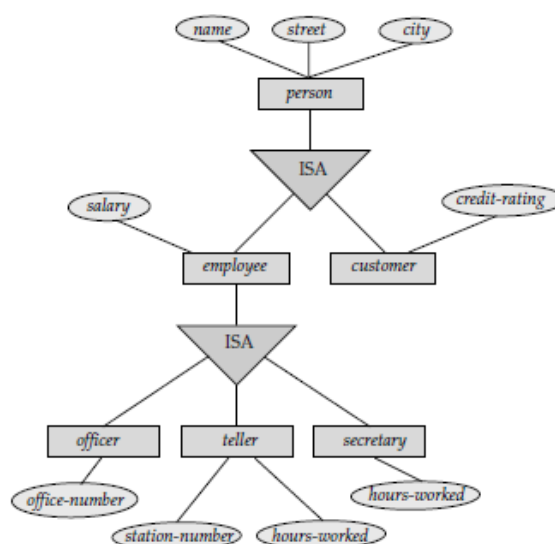
There are similarities between the customer entity set and the employee entity set in the sense that they have several attributes in common. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a higher-level entity set and one or more lower-level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

Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The person entity set is the superclass of the customer and employee subclasses. generalization is a simple inversion of specialization.

Specialization stems from a single entity set; it emphasizes differences among entities within the set by creating distinct lower-level entity sets. These lower-level entity sets may have attributes, or may participate in relationships, that do not apply to all the entities in the higher-level entity set.

If customer and employee neither have attributes that person entities do not have nor participate in different relationships than those in which person entities participate, there would be no need to specialize the person entity set.

Generalization proceeds from the recognition that a number of entity sets share some common features (namely, they are described by the same attributes and participate in the same relationship sets). On the basis of their commonalities, generalization synthesizes these entity sets into a single, higher-level entity set. Generalization is used to emphasize the similarities among lower-level entity sets and to hide the differences; it also permits an economy of representation in that shared attributes are not repeated.



Specialization and generalization.

Attribute Inheritance

A crucial property of the higher- and lower-level entities created by specialization and generalization is **attribute inheritance**. The attributes of the higher-level entity sets are said to be **inherited** by the lower level entity sets. For example, customer and employee inherit the attributes of person. Thus, customer is described by its name, street, and city attributes, and additionally a customer-id attribute; employee is described by its name, street, and city attributes, and additionally employee-id and salary attributes.

A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher level entity (or superclass) participates. The officer, teller, and secretary entity sets can participate in the works-for relationship set, since the superclass employee participates in the works-for relationship. Attribute inheritance applies through all tiers of lower-level entity sets. The above entity sets can participate in any relationships in which the person entity set participates. Whether a given portion of an E-R model was arrived at by specialization or generalization, the outcome is basically the same:

- A higher-level entity set with attributes and relationships that apply to all of its lower-level entity sets
- Lower-level entity sets with distinctive features that apply only within a particular lower-level entity set

Employee is a lower-level entity set of person and a higher-level entity set of the officer, teller, and secretary entity sets. In a hierarchy, a given entity set may be involved as a lower-level entity set in only one ISA relationship; that is, entity sets in this diagram have only **single inheritance**. If an entity set is a lower-level entity set in more than one ISA relationship, then the entity set has **multiple inheritance**, and the resulting structure is said to be a lattice.

Constraints on Generalizations

the database designer may choose to place certain constraints on a particular generalization. One type of constraint involves determining which entities can be members of a given lower-level entity set. Such membership may be one of the following:

- **Condition-defined.** In condition-defined lower-level entity sets, membership is evaluated on the basis of whether or not an entity satisfies an explicit condition or predicate. For example, assume that the higher level entity set account has the attribute account-type. All account entities are evaluated on the defining account-type attribute. Only those

entities that satisfy the condition `account-type = "savings account"` are allowed to belong to the lower-level entity set `person`. All entities that satisfy the condition `account-type = "checking account"` are included in `checking account`. Since all the lower-level entities are evaluated on the basis of the same attribute (in this case, on `account-type`), this type of generalization is said to be **attribute-defined**.

- **User-defined.** User-defined lower-level entity sets are not constrained by a membership condition; rather, the database user assigns entities to a given entity set.

A second type of constraint relates to whether or not entities may belong to more than one lower-level entity set within a single generalization. The lower-level entity sets may be one of the following:

- **Disjoint.** A disjointness constraint requires that an entity belong to no more than one lower-level entity set. In our example, an account entity can satisfy only one condition for the `account-type` attribute; an entity can be either a savings account or a checking account, but cannot be both.
- **Overlapping.** In overlapping generalizations, the same entity may belong to more than one lower-level entity set within a single generalization. A given employee may therefore appear in more than one of the team entity sets that are lower-level entity sets of employee. Thus, the generalization is overlapping. As another example, suppose generalization applied to entity sets `customer` and `employee` leads to a higher level entity set `person`. The generalization is overlapping if an employee can also be a customer.

A final constraint, the **completeness constraint** on a generalization or specialization, specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/specialization. This constraint may be one of the following:

- **Total generalization or specialization.** Each higher-level entity must belong to a lower-level entity set.
- **Partial generalization or specialization.** Some higher-level entities may not belong to any lower-level entity set.

Partial generalization is the default. We can specify total generalization in an E-R diagram by using a double line to connect the box representing the higher-level entity set to the triangle symbol.

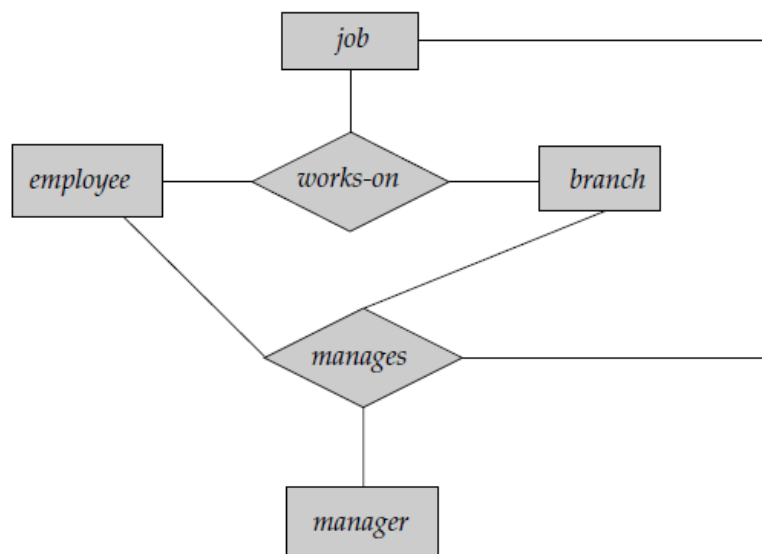
8. Explain Aggregation? (11 Marks)

One limitation of the E-R model is that it cannot express relationships among relationships

One alternative for representing this relationship is to create a quaternary relationship manages between employee, branch, job, and manager. (A quaternary relationship is required—a binary relationship between manager and employee would not permit us to represent which (branch, job) combinations of an employee are managed by which manager.)

It appears that the relationship sets works-on and manages can be combined into one single relationship set. If the manager were a value rather than an manager entity, we could instead make manager a multivalued attribute of the relationship works-on.

Aggregation is an abstraction through which relationships are treated as higher level entities.



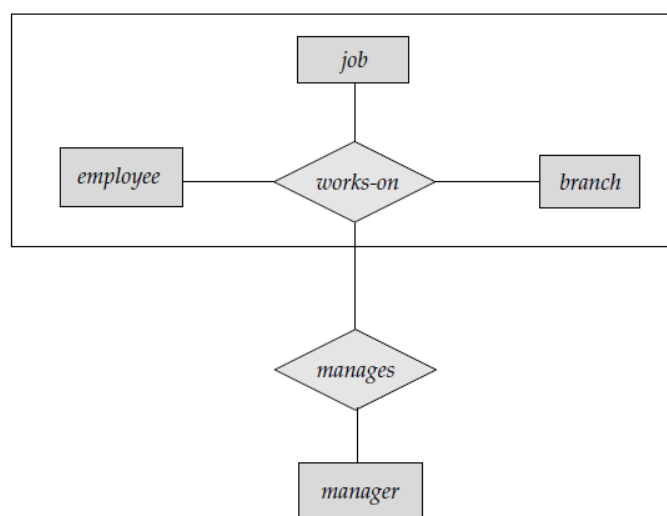
E-R diagram with redundant relationships.

Alternative E-R Notations

An entity set may be represented as a box with the name outside, and the attributes listed one below the other within the box. The primary key attributes are indicated by listing them at the top, with a line separating them from the other attributes.

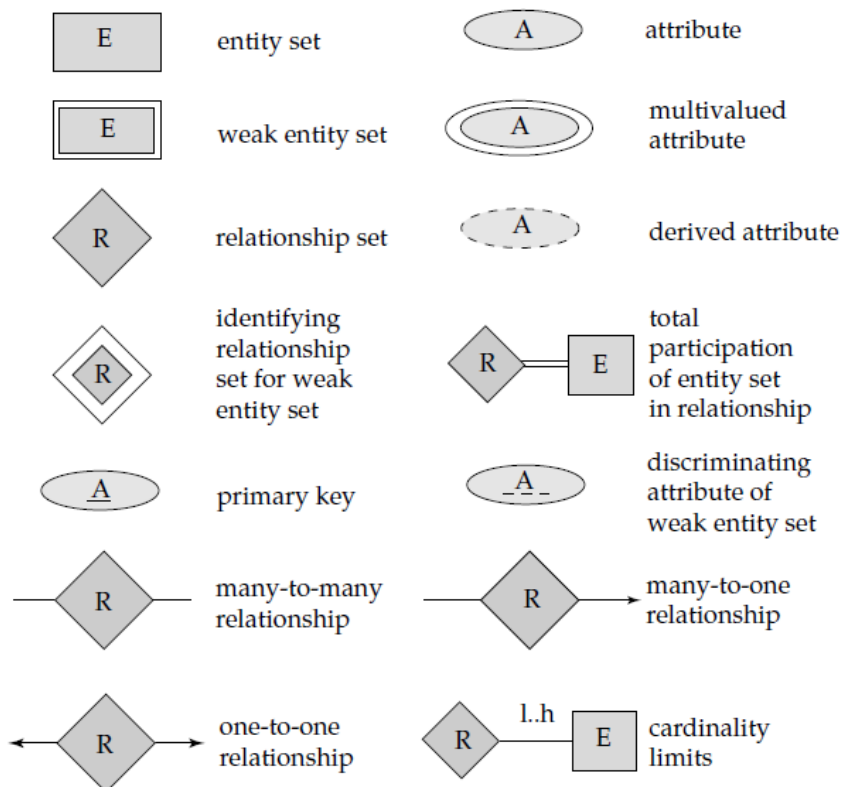
Cardinality constraints can be indicated in several different ways. The labels * and 1 on the edges out of the relationship are sometimes used for depicting many-to-many, one-to-one, and many-

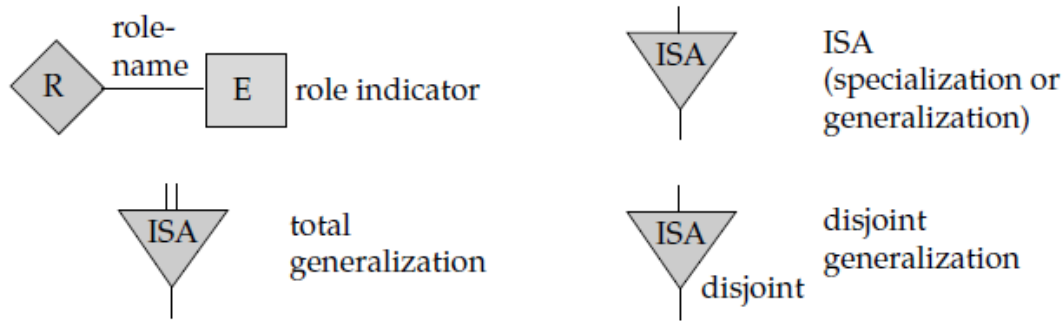
to-one relationships. Cardinality constraints in such a notation are shown by “crow’s foot” notation.



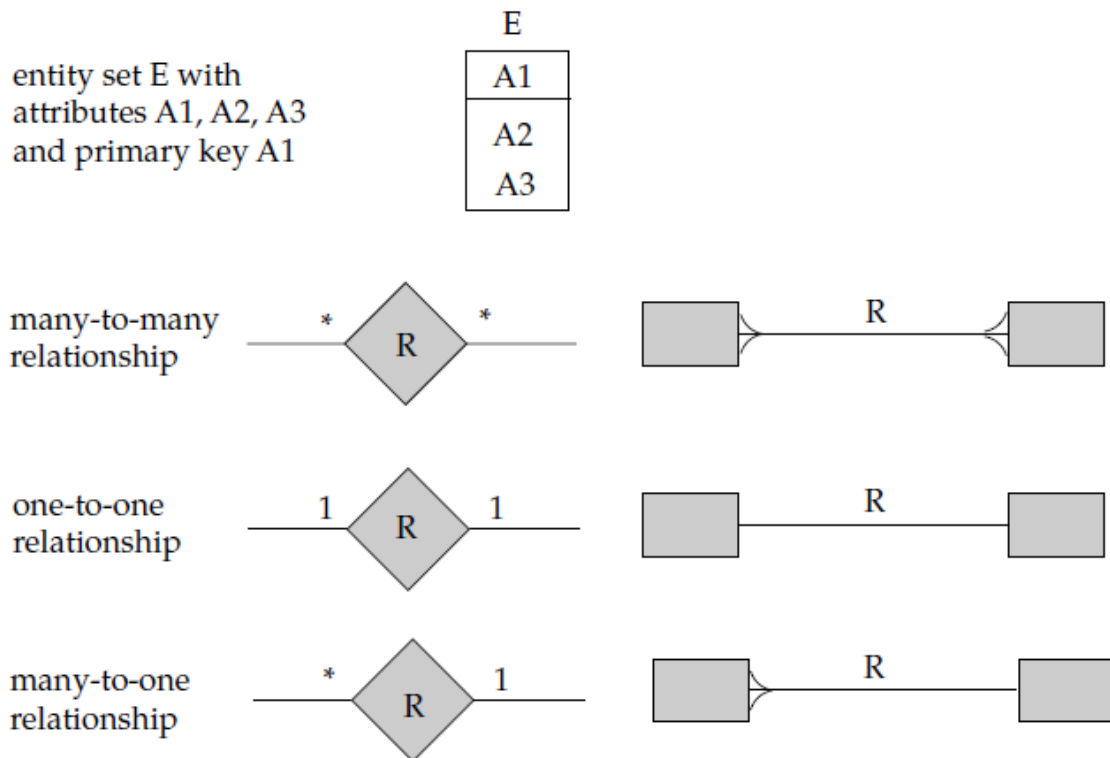
E-R diagram with aggregation

Design of an E-R Database Schema





Symbols used in the E-R notation.



Alternative E-R notations.

9. Explain about Design Phases? (11 Marks)NOV 2014

A high-level data model serves the database designer by providing a conceptual framework in which to specify, in a systematic fashion, what the data requirements of the database users are, and how the database will be structured to fulfill these requirements. The initial phase of database design, then, is to characterize fully the data needs of the prospective database users. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements.

Next, the designer chooses a data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase provides a detailed overview of the enterprise.

In a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data.

Database Design for Banking Enterprise

Data Requirements

The major characteristics of the banking enterprise:

- The bank is organized into branches. Each branch is located in a particular city and is identified by a unique name. The bank monitors the assets of each branch.
- Bank customers are identified by their customer-id values. The bank stores each customer's name, and the street and city where the customer lives. Customer may have accounts and can take out loans. A customer may be associated with a particular banker, who may act as a loan officer or personal banker for that customer.
- Bank employees are identified by their employee-id values. The bank administration stores the name and telephone number of each employee, the names of the employee's dependents, and the employee-id number of the employee's manager. The bank also keeps track of the employee's start date and, thus, length of employment.
- The bank offers two types of accounts savings and checking accounts. Accounts can be held by more than one customer, and a customer can have more than one account. Each account is assigned a unique account number. The bank maintains a record of each account's balance, and the most recent date on which the account was accessed by each customer holding the account. In addition, each savings account has an interest rate, and overdrafts are recorded for each checking account.
- A loan originates at a particular branch and can be held by one or more customers. A loan is identified by a unique loan number. For each loan, the bank keeps track of the loan amount and the loan payments. Although a loanpayment number does not uniquely identify a particular payment among those for all the bank's loans, a payment number does identify a particular payment for a specific loan. The date and amount are recorded for each payment.

Entity Sets Designation

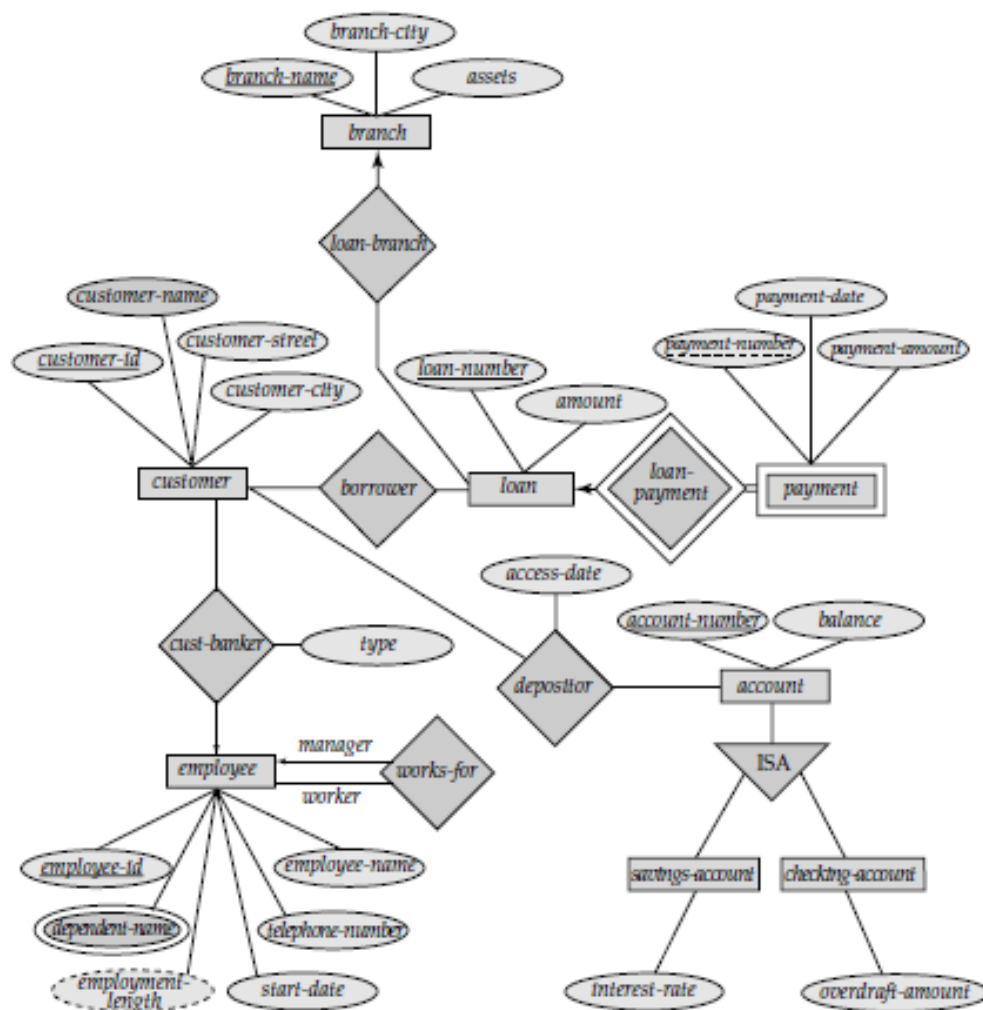
- The branch entity set, with attributes branch-name, branch-city, and assets.

- The customer entity set, with attributes customer-id, customer-name, customerstreet; and customer-city. A possible additional attribute is banker-name.
- The employee entity set, with attributes employee-id, employee-name, telephonenumber, salary, and manager. Additional descriptive features are the multivalued attribute dependent-name, the base attribute start-date, and the derived attribute employment-length.
- Two account entity sets—savings-account and checking-account—with the common attributes of account-number and balance; in addition, savings-account has the attribute interest-rate and checking account has the attribute overdraft-amount.
- The loan entity set, with the attributes loan-number, amount, and originatingbranch.
- The weak entity set loan-payment, with attributes payment-number, paymentdate, and payment-amount.

Relationship Sets Designation

- borrower, a many-to-many relationship set between customer and loan.
- loan-branch, a many-to-one relationship set that indicates in which branch a loan originated.
- loan-payment, a one-to-many relationship from loan to payment, which documents that a payment is made on a loan.
- depositor, with relationship attribute access-date, a many-to-many relationship set between customer and account, indicating that a customer owns an account.
- cust-banker, with relationship attribute type, a many-to-one relationship set expressing that a customer can be advised by a bank employee, and that a bank employee can advise one or more customers. Note that this relationship set has replaced the attribute banker-name of the entity set customer.
- works-for, a relationship set between employee entities with role indicators manager and worker; the mapping cardinalities express that an employee worksfor only one manager and that a manager supervises one or more employees.

E-R Diagram



Reduction of an E-R Schema to Tables

Tabular Representation of Strong Entity Sets

Let E be a strong entity set with descriptive attributes a_1, a_2, \dots, a_n . We represent this entity by a table called E with n distinct columns, each of which corresponds to one of the attributes of E . Each row in this table corresponds to one entity of the entity set E .

Consider the entity set loan of the E-R diagram. This entity set has two attributes: loan-number and amount . We represent this entity set by a table called loan , with two columns.

Let D_1 denote the set of all loan numbers, and let D_2 denote the set of all balances. Any row of the loan table must consist of a 2-tuple (v_1, v_2) , where v_1 is a loan (that is, v_1 is in set D_1) and v_2 is an amount (that is, v_2 is in set D_2). In general, the loan table will contain only a subset of the set of all possible rows. We refer to the set of all possible rows of loan as the Cartesian product of D_1 and D_2 , denoted by $D_1 \times D_2$.

In general, if we have a table of n columns, we denote the Cartesian product of

D_1, D_2, \dots, D_n by

$D_1 \times D_2 \times \dots \times D_{n-1} \times D_n$

<i>loan-number</i>	<i>amount</i>
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

<i>customer-id</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
019-28-3746	Smith	North	Rye
182-73-6091	Turner	Putnam	Stamford
192-83-7465	Johnson	Alma	Palo Alto
244-66-8800	Curry	North	Rye
321-12-3123	Jones	Main	Harrison
335-57-7991	Adams	Spring	Pittsfield
336-66-9999	Lindsay	Park	Pittsfield
677-89-9011	Hayes	Main	Harrison
963-96-3963	Williams	Nassau	Princeton

Tabular Representation of Weak Entity Sets

Let A be a weak entity set with attributes a_1, a_2, \dots, a_m . Let B be the strong entity set on which A depends. Let the primary key of B consist of attributes b_1, b_2, \dots, b_n . We represent the entity set A by a table called A with one column for each attribute of the set:

$\{a_1, a_2, \dots, a_m\} \cup \{b_1, b_2, \dots, b_n\}$

This entity set has three attributes: payment-number, payment-date, and payment-amount. The primary key of the loan entity set, on which payment depends, is loan-number. Thus, we represent payment by a table with four columns labeled loan-number, paymentnumber, payment-date, and payment-amount.

Tabular Representation of Relationship Sets

Let R be a relationship set, let a_1, a_2, \dots, a_m be the set of attributes formed by the union of the primary keys of each of the entity sets participating in R , and let the descriptive attributes (if any) of R be b_1, b_2, \dots, b_n . We represent this relationship set by a table called R with one column for each attribute of the set:

$\{a_1, a_2, \dots, a_m\} \cup \{b_1, b_2, \dots, b_n\}$

This relationship set involves the following two entity sets:

- customer, with the primary key customer-id
- loan, with the primary key loan-number

Since the relationship set has no attributes, the borrower table has two columns, labelled customer-id and loan-number.

<i>loan-number</i>	<i>payment-number</i>	<i>payment-date</i>	<i>payment-amount</i>
L-11	53	7 June 2001	125
L-14	69	28 May 2001	500
L-15	22	23 May 2001	300
L-16	58	18 June 2001	135
L-17	5	10 May 2001	50
L-17	6	7 June 2001	50
L-17	7	17 June 2001	100
L-23	11	17 May 2001	75
L-93	103	3 June 2001	900
L-93	104	13 June 2001	200

Redundancy of Tables

A relationship set linking a weak entity set to the corresponding strong entity set is treated specially. The weak entity set payment is dependent on the strong entity set loan via the relationship set loan payment. The primary key of payment is {loan-number, payment-number}, and the primary key of loan is {loan-number}. Since loan-payment has no descriptive attributes, the loan-payment table would have two columns, loan-number and payment number. The table for the entity set payment has four columns, loan-number, payment number, payment-date, and payment-amount. Every(loan-number, payment-number) combination in loan-payment would also be present in the payment table, and vice versa. Thus, the loan-payment table is redundant.

<i>customer-id</i>	<i>loan-number</i>
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

Combination of Tables

Consider a many-to-one relationship set AB from entity set A to entity set B. Using our table construction scheme outlined previously, we get three tables: A, B, and AB. Suppose further that the participation of A in the relationship is total; that is, every entity a in the entity set A must participate in the relationship AB. Then we can combine the tables A and AB to form a single table consisting of the union of columns of both tables.

The double line in the E-R diagram indicates that the participation of account in the account-branch is total. Hence, an account cannot exist without being associated with a particular branch. Further, the relationship set account-branch is many to one from account to branch. Therefore, we can combine the table for account-branch with the table for account and require only the following two tables:

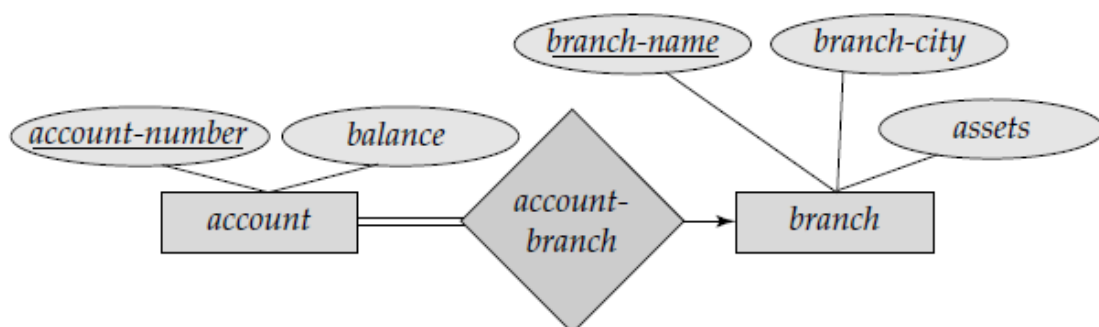
- account, with attributes account-number, balance, and branch-name
- branch, with attributes branch-name, branch-city, and assets

Composite Attributes(2 MARKS NOV 2010)

We handle composite attributes by creating a separate attribute for each of the component attributes; we do not create a separate column for the composite attribute itself. Suppose address is a composite attribute of entity set customer, and the components of address are street and city. The table generated from customer would then contain columns address-street and address-city; there is no separate column for address.

Multivalued Attributes

We have seen that attributes in an E-R diagram generally map directly into columns for the appropriate tables. Multivalued attributes, however, are an exception; new tables are created for these attributes.



Tabular Representation of Generalization

There are two different methods for transforming to a tabular form an E-R diagram that includes generalization.

1. Create a table for the higher-level entity set. For each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the primary key of the higher-level entity set. We have three tables:

- account, with attributes account-number and balance
- savings-account, with attributes account-number and interest-rate
- checking-account, with attributes account-number and overdraft-amount

2. An alternative representation is possible, if the generalization is disjoint and complete—that is, if no entity is a member of two lower-level entity sets directly below a higher-level entity set, and if every entity in the higher level entity set is also a member of one of the lower-level entity sets. Here, do not create a table for the higher-level entity set. Instead, for each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the higher-level entity set. We have two tables.

- savings-account, with attributes account-number, balance, and interest-rate
- checking-account, with attributes account-number, balance, and overdraftamount

The savings-account and checking-account relations corresponding to these tables both have account-number as the primary key.

Tabular Representation of Aggregation

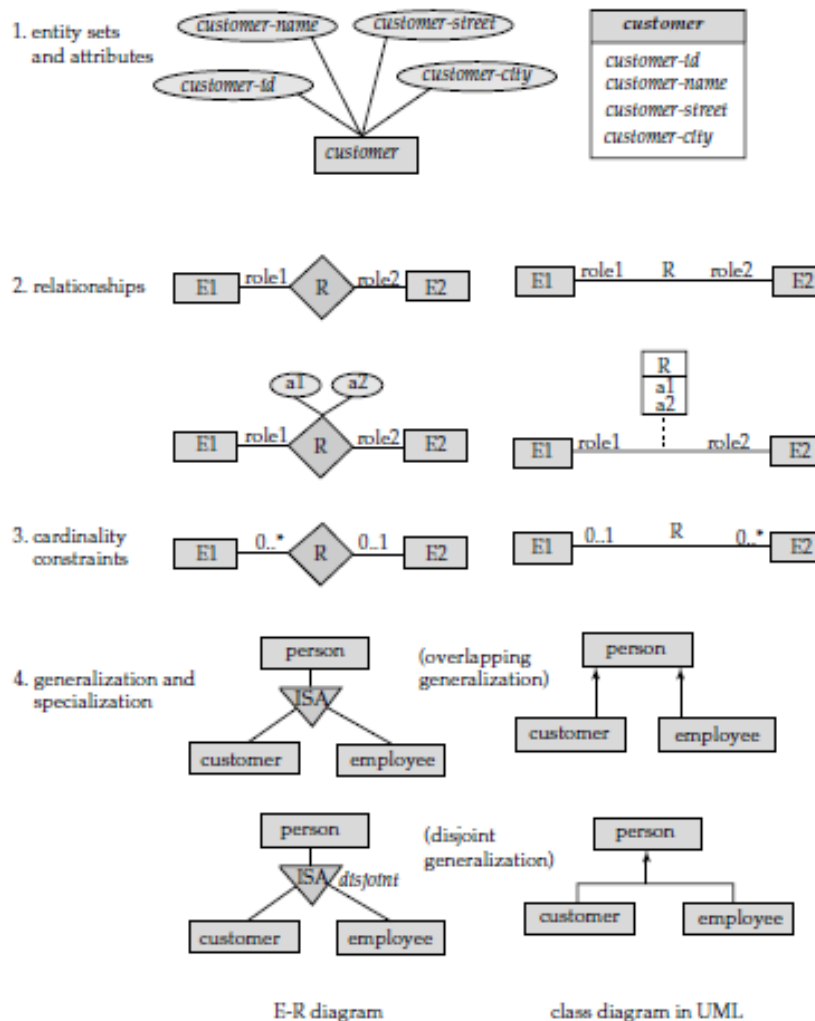
The table for the relationship set manages between the aggregation of works-on and the entity set manager includes a column for each attribute in the primary keys of the entity set manager and the relationship set works-on. It would also include a column for any descriptive attributes, if they exist, of the relationship set manages. We then transform the relationship sets and entity sets within the aggregated entity.

The Unified Modeling Language UML**

Entity-relationship diagrams help model the data representation component of a software system. Data representation, however, forms only one part of an overall system design. Other components include models of user interactions with the system, specification of functional modules of the system and their interaction, etc. The **Unified Modeling Language (UML)**, is a proposed standard for creating specifications of various components of a software system. Some of the parts of UML are:

- **Class diagram.** A class diagram is similar to an E-R diagram.

- **Use case diagram.** Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as withdrawing money or registering for a course).
- **Activity diagram.** Activity diagrams depict the flow of tasks between various components of a system.
- **Implementation diagram.** Implementation diagrams show the system components and their interconnections, both at the software component level and the hardware component level.



Symbols used in theUML class diagram notation.

Nonbinary relationships cannot be directly represented in UML. Cardinality constraints are specified in UML in the same way as in E-R diagrams, in the form l..h, where l denotes the minimum and h the maximum number of relationships an entity can participate in. However, you should be aware that the positioning of the constraints is exactly the reverse of the positioning of constraints in E-R diagrams,

15. Explain Overall Database Design Process? (11 Marks) NOV2014

Several ways are:

1. R could have been generated when converting a E-R diagram to a set of tables.
2. R could have been a single relation containing all attributes that are of interest. The normalization process then breaks up R into smaller relations.
3. R could have been the result of some ad hoc design of relations, which we then test to verify that it satisfies a desired normal form.

E-R Model and Normalization

Suppose an employee entity had attributes department-number and department-address, and there is a functional dependency department-number \rightarrow department-address. We would then need to normalize the relation generated from employee.

Functional dependencies can help us detect poor E-R design. If the generated relations are not in desired normal form, the problem can be fixed in the E-R diagram. That is, normalization can be done formally as part of data modeling. Alternatively, normalization can be left to the designer's intuition during E-R modeling, and can be done formally on the relations generated from the E-R model.

The Universal Relation Approach

The second approach to database design is to start with a single relation schema containing all attributes of interest, and decompose it. One of our goals in choosing a decomposition was that it be a lossless-join decomposition.

Dangling tuples may occur in practical database applications. They represent incomplete information, as they do in our example, where we wish to store data about a loan that is still in the process of being negotiated. The relation $r_1 \sqcup r_2 \sqcup \dots \sqcup r_n$ is called a **universal relation**, since it involves all the attributes in the universe defined by $R_1 \cup R_2 \cup \dots \cup R_n$.

The normal forms that we have defined generate good database designs from the point of view of representation of incomplete information. Returning again to the we would not want to allow storage of the following fact: "There is a loan (whose number is unknown) to Jones in the amount of \$100." This is because loan-number \rightarrow customer-name amount and therefore the only way that we can relate customer-name and amount is through loan-number.

Denormalization for Performance

One alternative to computing the join on the fly is to store a relation containing all the attributes of account and depositor. This makes displaying the account information faster. However, the balance information for an account is repeated for every person who owns the account, and all copies must be updated by the application, whenever the account balance is updated. The process of taking a normalized schema and making it non-normalized is called **denormalization**, and designers use it to tune performance of systems to support time-critical operations.

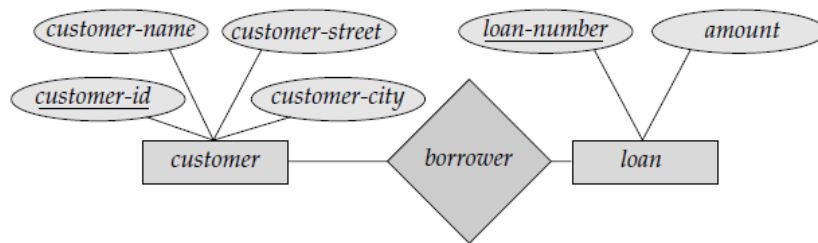
Other Design Issues

Consider a company database, where we want to store earnings of companies in different years. A relation `earnings(company-id, year, amount)` could be used to store the earnings information. The only functional dependency on this relation is `company-id, year → amount`, and the relation is in BCNF. An alternative design is to use multiple relations, each storing the earnings for a different year. Let us say the years of interest are 2000, 2001, and 2002; we would then have relations of the form `earnings-2000`, `earnings-2001`, `earnings-2002`, all of which are on the schema `(company-id, earnings)`. The only functional dependency here on each relation would be `company-id → earnings`, so these relations are also in BCNF. However, this alternative design is clearly a bad idea—we would have to create a new relation every year, and would also have to write new queries every year, to take each new relation into account. Queries would also be more complicated since they may have to refer to many relations.

16. Discuss the various concepts of ER Model with an example. **APRIL 2015**

The following major components:

- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationship sets
- **Lines**, which link attributes to entity sets and entity sets to relationship sets
- **Double ellipses**, which represent multivalued attributes
- **Dashed ellipses**, which denote derived attributes
- **Double lines**, which indicate total participation of an entity in a relationship set
- **Double rectangles**, which represent weak entity
 - Diagram



E-R diagram corresponding to customers and loans.

17. Explain about Database Design Process APRIL 2015

Database Design for Banking Enterprise

Data Requirements

The major characteristics of the banking enterprise:

- The bank is organized into branches. Each branch is located in a particular city and is identified by a unique name. The bank monitors the assets of each branch.
- Bank customers are identified by their customer-id values. The bank stores each customer's name, and the street and city where the customer lives. Customer may have accounts and can take out loans. A customer may be associated with a particular banker, who may act as a loan officer or personal banker for that customer.

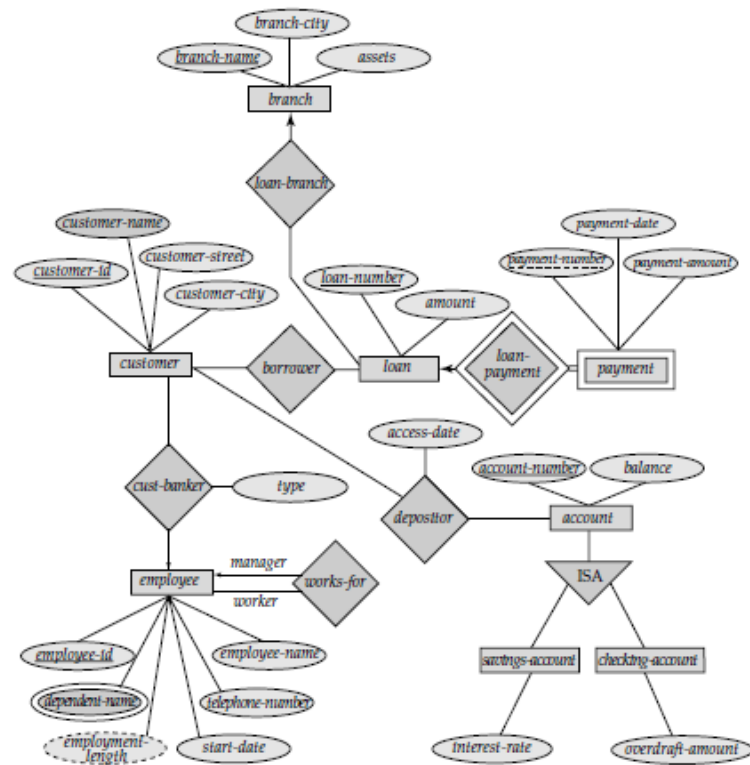
Entity Sets Designation

- The branch entity set, with attributes branch-name, branch-city, and assets.
- The customer entity set, with attributes customer-id, customer-name, customerstreet; and customer-city. A possible additional attribute is banker-name.

Relationship Sets Designation

- borrower, a many-to-many relationship set between customer and loan.
- loan-branch, a many-to-one relationship set that indicates in which branch a loan originated.

E-R Diagram



Reduction of an E-R Schema to Tables

1. Explain authorization in SQL. With example (April 2011)

Authorization:

We may assign a user several forms of authorization on parts of the database. For example,

- **Read authorization** allows reading, but not modification, of data.
- **Insert authorization** allows insertion of new data, but not modification of existing data.
- **Update authorization** allows modification, but not deletion, of data.
- **Delete authorization** allows deletion of data.

We may assign the user all, none, or a combination of these types of authorization. In addition to these forms of authorization for access to data, we may grant a user authorization to modify the database schema:

- **Index authorization** allows the creation and deletion of indices.
- **Resource authorization** allows the creation of new relations.
- **Alteration authorization** allows the addition or deletion of attributes in a relation.
- **Drop authorization** allows the deletion of relations.

Authorization in SQL

- ✚ The SQL language offers a fairly powerful mechanism for defining authorizations.

Privileges in SQL:

- ✚ The SQL standard includes the privileges **delete** , **insert** ,**select** ,and **update** .

- ✚ The **select** privilege corresponds to the **read** privilege.
- ✚ SQL also includes a **references** privilege that permits a user/role to declare foreign keys when creating relations.
- ✚ If the relation to be created includes a foreign key that references attributes of another relation, the user/role must have been granted **references** privilege on those attributes.

The SQL data-definition language includes commands to grant and revoke privileges. The **grant** statement is used to confer authorization. The basic form of this statement is:

grant <privilege list >**on** <relation name or view name >**to** <user/role list >

The *privilege list* allows the granting of several privileges in one command.

✚ The following **grant** statement grants users U 1 ,U 2 ,and U 3 **select** authorization on the *account* relation:

grant select on account to U 1 ,U 2 ,U 3

- ✚ In the below example, the **grant** statement gives users U 1 ,U 2 ,and U 3 update authorization on the *amount* attribute of the *loan* relation:

grant update (amount) on loan to U 1 ,U 2 ,U 3

2. Explain Database design process in detail (April 2011)

Database design

Database design is the process of producing a detailed data model of a database. This logical data model contains all the needed logical and physical design choices and physical storage parameters needed to generate a design in a Data Definition Language, which can then be used to create a database. A fully attributed data model contains detailed attributes for each entity.

The term database design can be used to describe many different parts of the design of an overall database system. Principally, and most correctly, it can be thought of as the logical design of the base data structures used to store the data. In the relational model these are the tables and views. In an Object database the entities and relationships map directly to object classes and named relationships. However, the term database design could also be used to apply to the overall process of designing, not just the base data structures, but also the forms and queries used as part of the overall database application within the Database Management System or DBMS.

Design process

The process of doing database design generally consists of a number of steps which will be carried out by the database designer. Not all of these steps will be necessary in all cases. Usually, the designer must:

- Determine the data to be stored in the database
- Determine the relationships between the different data elements
- Superimpose a logical structure upon the data on the basis of these relationships.

Within the relational model the final step can generally be broken down into two further steps, that of determining the grouping of information within the system, generally determining what are the basic objects about which information is being stored, and then determining the relationships between these groups of information, or objects. This step is not necessary with an Object database.

The tree structure of data may enforce a hierarchical model organization, with a parent-child relationship table. An Object database will simply use a one-to-many relationship between instances of an object class. It also introduces the concept of a hierarchical relationship between object classes, termed inheritance

Determining data to be stored

In a majority of cases, the person who is doing the design of a database is a person with expertise in the area of database design, rather than expertise in the domain from which the data to be stored is drawn e.g. financial information, biological information etc. Therefore the data to be stored in the database must be determined in cooperation with a person who does have expertise in that domain, and who is aware of what data must be stored within the system.

This process is one which is generally considered part of requirements analysis, and requires skill on the part of the database designer to elicit the needed information from those with the domain knowledge. This is because those with the necessary domain knowledge frequently cannot express clearly what their system requirements for the database are as they are unaccustomed to thinking in terms of the discrete data elements which must be stored.

Conceptual schema

Once a database designer is aware of the data which is to be stored within the database, they must then determine how the various pieces of that data relate to one another. When performing this step, the designer is generally looking out for the dependencies in the data, where one piece of information is dependent upon another i.e. when one piece of information changes, the other will also. For example, in a list of names and addresses, the address is dependent upon the name, because if the name is different then the associated address is different too. However, the inverse is not necessarily true, i.e. when the address changes name may be the same. For example, assuming the normal situation where two people can have the same address, but one person cannot have two addresses.

(**NOTE:** A common misconception is that the relational model is so called because of the stating of relationships between data elements therein. This is not true. The relational model is so named such because it is based upon the mathematical structures known as relations.)

Logically structuring data

Once the relationships and dependencies amongst the various pieces of information have been determined, it is possible to arrange the data into a logical structure which can then be mapped

into the storage objects supported by the database management system. In the case of relational databases the storage objects are tables which store data in rows and columns.

Each table may represent an implementation of either a logical object or a relationship joining one or more instances of one or more logical objects. Relationships between tables may then be stored as links connecting child tables with parents. Since complex logical relationships are themselves tables they will probably have links to more than one parent.

In an Object database the storage objects correspond directly to the objects used by the Object-oriented programming language used to write the applications that will manage and access the data. The relationships may be defined as attributes of the object classes involved or as methods that operate on the object classes.

3. Describe Magnetic Disk and Flash storage in detail (April 2012)

Magnetic disks

Physical Characteristics of Disks

1. The storage capacity of a single disk ranges from 10MB to 10GB. A typical commercial database may require hundreds of disks.
 - Each disk *platter* has a flat circular shape. Its two surfaces are covered with a magnetic material and information is recorded on the surfaces. The platter of *hard disks* are made from rigid metal or glass, while *floppy disks* are made from flexible material.
 - The disk surface is logically divided into *tracks*, which are subdivided into *sectors*. A sector (varying from 32 bytes to 4096 bytes, usually 512 bytes) is the smallest unit of information that can be read from or written to disk. There are 4-32 sectors per track and 20-1500 tracks per disk surface.
 - The **arm** can be positioned over any one of the tracks.
 - The **platter** is spun at high speed.
 - To read information, the arm is **positioned** over the correct track.
 - When the data to be accessed passes under the head, the **read** or **write** operation is performed.
2. A disk typically contains multiple platters (see Figure 10.2). The read-write heads of all the tracks are mounted on a single assembly called a *disk arm*, and move together.
 - Multiple disk arms are moved as a unit by the **actuator**.
 - Each arm has two heads, to read disks above and below it.
 - The set of **tracks** over which the heads are located forms a **cylinder**.
 - This cylinder holds that data that is accessible within the disk latency time.
 - It is clearly sensible to store related data in the same or adjacent cylinders.

3. Disk platters range from 1.8" to 14" in diameter, and 5"1/4 and 3"1/2 disks dominate due to the lower cost and faster seek time than do larger disks, yet they provide high storage capacity.
4. A *disk controller* interfaces between the computer system and the actual hardware of the disk drive. It accepts commands to r/w a sector, and initiate actions. Disk controllers also attach *checksums* to each sector to check read error.
5. *Remapping of bad sectors*: If a controller detects that a sector is damaged when the disk is initially formatted, or when an attempt is made to write the sector, it can logically map the sector to a different physical location.
6. SCSI (*Small Computer System Interconnect*) is commonly used to connect disks to PCs and workstations. Mainframe and server systems usually have a faster and more expensive bus to connect to the disks.
7. Head crash: why cause the entire disk failing (?).
8. A *fixed dead disk* has a separate head for each track -- very many heads, very expensive. *Multiple disk arms*: allow more than one track to be accessed at a time. Both were used in high performance mainframe systems but are relatively rare today.

4. Explain Join operation in detail (April 2012)

Join (SQL)

A JOIN clause in SQL combines records from two tables in a relational database and results in a new ("temporary") table, also called a "joined table". Also you can think about the JOIN as a SQL operation that relates tables by means of values that they share in common.

SQL specifies two types of joins: INNER and OUTER.

A programmer writes a join predicate to identify the records for JOINing. If the predicate evaluates true, then the combined record is inserted into the joined (temporary) table; otherwise, it does not contribute. Any predicate supported by SQL can become a join-predicate, for example, WHERE-clauses. As a special case, a table (base table, view, or joined table) can join to itself in a **self-join**.

Mathematically, a join consists of a relation composition. It provides the fundamental operation in relational algebra and generalizes function composition.

Sample tables

All subsequent explanations on join types in this article make use of the following two tables. The rows in these tables serve to illustrate the effect of different types of joins and join-predicates.

Department Table	
DepartmentID	DepartmentName
31	Sales

33	Engineering
34	Clerical
35	Marketing

Employee Table

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34
Jasper	36

Note: The "Marketing" Department currently has no listed employees. On the other hand, the employee "Jasper" has no link to any currently valid Department in the Department Table.

Inner join

An inner join essentially combines the records from two tables (A and B) based on a given join-predicate. The SQL-engine computes the cross-product of all records in the tables. Thus, processing combines each record in table A with every record in table B. Only those records in the joined table that satisfy the join predicate remain. This type of join occurs the most commonly in applications, and represents the default join-type.

SQL:2003 specifies two different syntactical ways to express joins. The first, called "explicit join notation", uses the keyword JOIN, whereas the second uses the "implicit join notation". The implicit join notation lists the tables for joining in the FROM clause of a SELECT statement, using commas to separate them. Thus, it always computes a cross-join, and the WHERE clause may apply additional filter-predicates. Those filter-predicates function comparably to join-predicates in the explicit notation.

One can further classify inner joins as equi-joins, as natural joins, or as cross-joins.

Programmers should take special care when joining tables on columns that can contain NULL values, since NULL will never match any other value (or even NULL itself), unless the join condition explicitly uses the IS NULL or IS NOT NULL predicates.

As an example, the following query takes all the records from the Employee table and finds the matching record(s) in the Department table, based on the join predicate. The join predicate compares the values in the DepartmentID column in both tables. If it finds no match (i.e., the department-id of an employee does not match the current department-id from the Department table), then the joined record remains outside the joined table, i.e., outside the (intermediate) result of the join.

Example of an explicit inner join:

```
SELECT *  
FROM employee  
    INNER JOIN department  
        ON employee.DepartmentID = department.DepartmentID
```

Example of an implicit inner join:

```
SELECT *  
FROM employee, department  
WHERE employee.DepartmentID = department.DepartmentID
```

Explicit Inner join result:

Employee.LastName	Employee.DepartmentID	Department.Department Name	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Steinberg	33	Engineering	33
Rafferty	31	Sales	31

Notice that the employee "Jasper" and the department "Marketing" do not appear. Neither of these have any matching records in the respective other table: no department has the department ID 36 and no employee has the department ID 35. Thus, no information on Jasper or on Marketing appears in the joined table.

Types of inner joins

Equi-join

An **equi-join** (also known as an **equijoin**), a specific type of comparator-based join, or *theta join*, uses only equality comparisons in the join-predicate. Using other comparison operators (such as

<) disqualifies a join as an equi-join. The query shown above has already provided an example of an equi-join:

```
SELECT *  
FROM employee  
    INNER JOIN department  
        ON employee.DepartmentID = department.DepartmentID
```

The resulting joined table contains two columns named DepartmentID, one from table Employee and one from table Department.

SQL:2003 does not have a specific syntax to express equi-joins, but some database engines provide a shorthand syntax: for example, MySQL and PostgreSQL support USING(DepartmentID) in addition to the ON ... syntax.

Natural join

A natural join offers a further specialization of equi-joins. The join predicate arises implicitly by comparing all columns in both tables that have the same column-name in the joined tables. The resulting joined table contains only one column for each pair of equally-named columns.

The above sample query for inner joins can be expressed as a natural join in the following way:

```
SELECT *  
FROM employee NATURAL JOIN department
```

The result appears slightly different, however, because only one DepartmentID column occurs in the joined table.

Employee.LastName	DepartmentID	Department.DepartmentName
Smith	34	Clerical
Jones	33	Engineering
Robinson	34	Clerical
Steinberg	33	Engineering
Rafferty	31	Sales

Using the NATURAL JOIN keyword to express joins can suffer from ambiguity at best, and leaves systems open to problems if schema changes occur in the database. For example, the removal, addition, or renaming of columns changes the semantics of a natural join. Thus, the safer approach involves explicitly coding the join-condition using a regular inner join.

The Oracle database implementation of SQL selects the appropriate column in the naturally-joined table from which to gather data. An error-message such as "ORA-25155: column used in NATURAL join cannot have qualifier" may encourage checking and precisely specifying the columns named in the query.

Outer joins

An outer join does not require each record in the two joined tables to have a matching record in the other table. The joined table retains each record—even if no other matching record exists. Outer joins subdivide further into left outer joins, right outer joins, and full outer joins, depending on which table(s) one retains the rows from (left, right, or both).

(For a table to qualify as *left* or *right* its name has to appear after the FROM or JOIN keyword, respectively.)

No implicit join-notation for outer joins exists in SQL:2003.

Left outer join

The result of a **left outer join** for tables A and B always contains all records of the "left" table (A), even if the join-condition does not find any matching record in the "right" table (B). This means that if the ON clause matches 0 (zero) records in B, the join will still return a row in the result—but with NULL in each column from B. This means that a **left outer join** returns all the values from the left table, plus matched values from the right table (or NULL in case of no matching join predicate).

For example, this allows us to find an employee's department, but still to show the employee even when their department does not exist (contrary to the inner-join example above, where employees in non-existent departments get filtered out).

Example of a left outer join(new):

```
SELECT *  
FROM employee  
LEFT OUTER JOIN department  
ON employee.DepartmentID = department.DepartmentID
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Jones	33	Engineering	33
Rafferty	31	Sales	31
Robinson	34	Clerical	34
Smith	34	Clerical	34
Jasper	36	NULL	NULL
Steinberg	33	Engineering	33

Right outer join

A right outer join closely resembles a left outer join, except with the tables reversed. Every record from the "right" table (B) will appear in the joined table at least once. If no matching row from the

"left" table (A) exists, NULL will appear in columns from A for those records that have no match in A.

A right outer join returns all the values from the right table and matched values from the left table (NULL in case of no matching join predicate).

Example right outer join:

```
SELECT *  
FROM employee  
RIGHT OUTER JOIN department  
ON employee.DepartmentID = department.DepartmentID
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Steinberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	Marketing	35

Full outer join

A **full outer join** combines the results of both left and right outer joins. The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

Example full outer join:

```
SELECT *  
FROM employee  
FULL OUTER JOIN department  
ON employee.DepartmentID = department.DepartmentID
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Jasper	36	NULL	NULL
Steinberg	33	Engineering	33
Rafferty	31	Sales	31

NULL	NULL	Marketing	35
------	------	-----------	----

Some database systems do not support this functionality directly, but they can emulate it through the use of left and right outer joins and unions. The same example can appear as follows:

```
SELECT *
FROM employee
    LEFT JOIN department
        ON employee.DepartmentID = department.DepartmentID
UNION
SELECT *
FROM employee
    RIGHT JOIN department
        ON employee.DepartmentID = department.DepartmentID
WHERE employee.DepartmentID IS NULL
```

Implementation

Much work in database-systems has aimed at efficient implementation of joins, because relational systems commonly call for joins, yet face difficulties in optimising their efficient execution. The problem arises because (inner) joins operate both commutatively and associatively. In practice, this means that the user merely supplies the list of tables for joining and the join conditions to use, and the database system has the task of determining the most efficient way to perform the operation. A query optimizer determines how to execute a query containing joins. A query optimizer has two basic freedoms:

1. **Join order:** Because joins function commutatively, the order in which the system joins tables does not change the final result-set of the query. However, join-order **does** have an enormous impact on the cost of the join operation, so choosing the best join order becomes very important.
2. **Join method:** Given two tables and a join condition, multiple algorithms can produce the result-set of the join. Which algorithm runs most efficiently depends on the sizes of the input tables, the number of rows from each table that match the join condition, and the operations required by the rest of the query.

Many join-algorithms treat their inputs differently. One can refer to the inputs to a join as the "outer" and "inner" join operands, or "left" and "right", respectively. In the case of nested loops, for example, the database system will scan the entire inner relation for each row of the outer relation.

One can classify query-plans involving joins as follows:

left-deep

using a base table (rather than another join) as the inner operand of each join in the plan

right-deep

using a base table as the outer operand of each join in the plan

bushy

neither left-deep nor right-deep; both inputs to a join may themselves result from joins

These names derive from the appearance of the query plan if drawn as a tree, with the outer join relation on the left and the inner relation on the right (as convention dictates).

Join algorithms

Three fundamental algorithms exist for performing a join operation.

Nested loops

Use of nested loops produces the simplest join-algorithm. For each tuple in the outer join relation, the system scans the entire inner-join relation and appends any tuples that match the join-condition to the result set. Naturally, this algorithm performs poorly with large join-relations: inner or outer or both. An index on columns in the inner relation in the join-predicate can enhance performance.

The "block nested loops" (BNL) approach offers a refinement to this technique: for every block in the outer relation, the system scans the entire inner relation. For each match between the current inner tuple and one of the tuples in the current block of the outer relation, the system adds a tuple to the join result-set. This variant means doing more computation for each tuple of the inner relation, but far fewer scans of the inner relation.

Merge join

If both join relations come in order, sorted by the join attribute(s), the system can perform the join trivially, thus:

1. For each tuple in the outer relation,
 1. Consider the current "group" of tuples from the inner relation; a group consists of a set of contiguous tuples in the inner relation with the same value in the join attribute.
 2. For each matching tuple in the current inner group, add a tuple to the join result. Once the inner group has been exhausted, advance both the inner and outer scans to the next group.

Merge joins offer one reason why many optimizers keep track of the sort order produced by query plan operators—if one or both input relations to a merge join arrives already sorted on the join attribute, the system need not perform an additional sort. Otherwise, the DBMS will need to perform the sort, usually using an external sort to avoid consuming too much memory.

Hash join

A hash join algorithm can produce equi-joins. The database system pre-forms access to the tables concerned by building hash tables on the join-attributes. The lookup in hash tables operates

much faster than through index trees. However, one can compare hashed values only for equality, not for other relationships.

5. Write short notes on (April 2013)

(a) Assertions.

(b) Security and Authorization.

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- Domain constraints and referential-integrity constraints are special forms of assertions.
- Assertions can be easily tested and can apply to a wide range of database applications.
- Examples are
 - The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.
 - Every loan has at least one customer who maintains an account with a minimum balance of \$1000.00.
- An assertion in SQL takes the form (Syntax)

create assertion <assertion-name> **check** <predicate>

- Since SQL does not provide a “for all X , $P(X)$ ” construct (where P is a predicate), we are forced to implement the construct by the equivalent “not exists X such that not $P(X)$ ” construct, which can be written in SQL.

create assertion *sum-constraint* **check**

(not exists (select * from branch
where (select sum(*amount*) from loan
where *loan.branch-name* = *branch.branch-name*)
>= (select sum(*balance*) from account
where *account.branch-name* = *branch.branch-name*)))

- When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

The high overhead of testing and maintaining assertions has led some system developers to omit support for general assertions, or to provide specialized forms of assertions that are easier to test

(b) Security and Authorization

- Security of data is an important concept in DBMS because it is essential to safeguard the data against any unwanted users.
- 5 different levels of security
 - DB System Level
 - Operating System Level
 - Network Level
 - Physical Level
 - Human Level

1. DB system level

- Authentication (verification) and authorization mechanism to allow specific users access only to required data.

2. Operating System Level:

- Protection from invalid logins
- File-level access protection
- Protection from improper use of “superuser” authority,
- Protection from improper use of privileged machine instructions.

3. Network level:

- Each site must ensure that it communicates with treated sites.
- Links must be protected from theft or modification of messages.

Mechanisms Used:

- * Identification protocol (password based).
- * Cryptography.

4. Physical Level:

- * Protection of equipment from floods, power failure etc..
- * Protection of disks from theft etc...
- * Protection of network and terminal cables from wire tapes etc...

Solution:

- * Physical security by locks etc...
- * Software techniques to detect physical security breaches.

5. Human Level:

Protection from stolen passwords etc...

Solution:

- * Frequent change of passwords.
- * Data audits

- ❖ The data stored in the database need protection from unauthorized access and malicious destruction or alteration, in addition to the protection against accidental introduction of inconsistency that integrity constraints provide.

Security Violations

Among the forms of malicious access are:

- Unauthorized reading of data (theft of information)
- Unauthorized modification of data
- Unauthorized destruction of data

Database security refers to protection from malicious access. Absolute protection of the database from malicious abuse is not possible, but the cost to the perpetrator can be made high enough to deter most if not all attempts to access the database without proper authority.

To protect the database, we must take security measures at several levels:

- ✚ **Database system** .Some database-system users may be authorized to access only a limited portion of the database.Other users may be allowed to issue queries,but may be forbidden to modify the data.It is the responsibility of the database system to ensure that these authorization restrictions are not violated.
- ✚ **Operating system** .No matter how secure the database system is,weakness in operating-system security may serve as a means of unauthorized access to the database.
- ✚ **Network** .Since almost all database systems allow remote access through terminals or networks,software-level security within the network software is as important as physical security,both on the Internet and in private networks.
- ✚ **Physical** .Sites with computer systems must be physically secured against armed or surreptitious entry by intruders.
- ✚ **Human** .Users must be authorized carefully to reduce the chance of any user giving access to an intruder in exchange for a bribe or other favors.

Authorization:

We may assign a user several forms of authorization on parts of the database. For example,

- **Read authorization** allows reading,but not modification,of data.
- **Insert authorization** allows insertion of new data,but not modification of existing data.
- **Update authorization** allows modification,but not deletion,of data.
- **Delete authorization** allows deletion of data.

We may assign the user all, none, or a combination of these types of authorization. In addition to these forms of authorization for access to data, we may grant a user authorization to modify the database schema:

- **Index authorization** allows the creation and deletion of indices.
- **Resource authorization** allows the creation of new relations.
- **Alteration authorization** allows the addition or deletion of attributes in a relation.
- **Drop authorization** allows the deletion of relations.

Authorization in SQL

- ✚ The SQL language offers a fairly powerful mechanism for defining authorizations.

Privileges in SQL:

- ✚ The SQL standard includes the privileges **delete**, **insert**, **select**, and **update**.
- ✚ The **select** privilege corresponds to the **read** privilege.
- ✚ SQL also includes a **references** privilege that permits a user/role to declare foreign keys when creating relations.
- ✚ If the relation to be created includes a foreign key that references attributes of another relation, the user/role must have been granted **references** privilege on those attributes.

The SQL data-definition language includes commands to grant and revoke privileges. The **grant** statement is used to confer authorization. The basic form of this statement is:

grant <privilege list > **on** <relation name or view name > **to** <user/role list >

The *privilege list* allows the granting of several privileges in one command.

- ✚ The following **grant** statement grants users U 1, U 2, and U 3 **select** authorization on the *account* relation:

grant select on account to U 1, U 2, U 3

- ✚ In the below example, the **grant** statement gives users U 1, U 2, and U 3 update authorization on the *amount* attribute of the *loan* relation:

grant update (amount) on loan to U 1, U 2, U 3

6. Describe about Normalization using functional dependency on (April 2013), (NOV 2015)

1. Another desirable property in database design is **dependency preservation**.
 - We would like to check easily that updates to the database do not result in illegal relations being created.
 - It would be nice if our design allowed us to check updates without having to compute natural joins.

- To know whether joins must be computed, we need to determine what functional dependencies may be tested by checking each relation individually.
- Let F be a set of functional dependencies on schema R .
- Let $\{R_1, R_2, \dots, R_n\}$ be a decomposition of R .
- The **restriction** of F to R_i is the set of all functional dependencies in F^+ that include only attributes of R_i .
- Functional dependencies in a restriction can be tested in one relation, as they involve attributes in one relation schema.
- The set of restrictions F_1, F_2, \dots, F_n is the set of dependencies that can be checked efficiently.
- We need to know whether testing only the restrictions is sufficient.
- Let $F' = F_1, F_2, \dots, F_n$.
- F' is a set of functional dependencies on schema R , but in general, $F' \neq F$.
- However, it may be that $F'^+ = F^+$.
- If this is so, then every functional dependency in F is implied by F' , and if F' is satisfied, then F must also be satisfied.
- A decomposition having the property that $F'^+ = F^+$ is a **dependency-preserving** decomposition.

18.Explain organization of records in files structure in detail (Apr 2011)

We have several of the possible ways of organizing records in files are:

- Heap file organization: Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.
- Sequential file organization: Records are stored in sequential order, according to the value of a “search key” of each record.
- Hashing file organization: A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed.
- Generally, a separate file is used to store the records of each relation.
- However, in a clustering file organization, records of several different relations are stored in the same file; further related records of the different relations are stored in the same block, so that one I\o operation fetches related records from all the relations.

SEQUENTIAL FILE ORGANIZATION:

- A sequential file is designed for efficient processing of records in sorted order based on some search-key.
- A search key is any attribute or set of attributes; it need not be the primary key, or even a super key.
- To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the next record in search-key

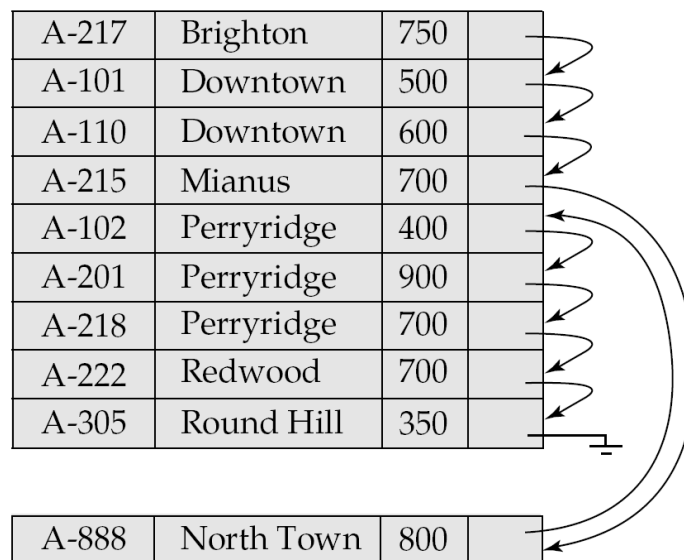
A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	



order,.

- A sequential file of account records taken from our banking example. In that example, the records are stored in search-key order, using branch name as the search key.

- It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion.
- We can manage deletion by using pointer chains, as we saw previously.
- For insertion, we apply the following rules:
- Locate the record in the file that comes before the record to be inserted in search-key order.
- If there is a free record (ie., space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an overflow block. In either case, adjust the pointers so as to chain together the records in search-key order.



order.

Sequential file after an insertion.

- The file after the insertion of the record (NorthTown,A-888,800). The structure in the allows fast insertion of new records, but forces sequential file-processing applications to process records in an order that does not match the physical order of the records.
- If relatively few records need to be stored in overflow blocks, this approach works well.
- At this point, the file should be reorganized so that it is once again physically in sequential order. Such reorganizations are costly, and must be done during times when the system load is low..
- In the extreme case in which insertions rarely occur, it is possible always to keep the file in physically sorted order. In such a case, the pointer field is not needed.

CLUSTERING FILE ORGANIZATION:

- Many relational-database systems store each relation in a separate file, so that they can take full advantage of the file system, that the operating systems provides
- The tuples of a relation can be represented as fixed-length records and can be mapped to simple file structure.
- The simple implementation of a relational database system is well suited to low-cost database implementations as in, for example, embedded systems or portable devices.
- A simple file structure reduces the amount of code needed to implement the system.
- This simple approach to relational-database implementation becomes less satisfactory as the size of the database increases.

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

To see the advantage of storing many relations in this one file, consider the following SQL query for the bank database:

```
select account-number, customer-name, customer-street, customer-city
from depositer, customer
where depositer.customer-name=customer.customer-name.
```

- This query computes a join of the depositer and customer relations.
- Thus, for each tuple of depositer, the system must locate the customer tuples with same value for customer-name.
- Ideally, these records will be located with the help of indices.
- In the worst case, each record will reside on a different block, forcing us to done block read for each record required by the query.

<i>customer-name</i>	<i>account-number</i>
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

DEPOSITOR and CUSTOMER Relation

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

- The above table file structure designed for efficient execution of queries involving depositor & customer.
- The depositor tuples for each customer-name are stored near the customer tuple for the corresponding customer-name.
- This structure mixes together tuples of two relations, but allows for efficient processing of the join.
- When a tuple of the customer relation is read, the entire block contain that tuple is copied from disk into main memory.
- Since the depositor tuples are stored on the disk near the customer-tuple, the block containing the customer tuple contains tuples of the depositor relation needed to process the query. If a customer has so many accounts that the depositor records do not fit in one block, the remaining records appear on nearby blocks.
- A clustering file organization is a file organization, such as that illustrated in that stores related records of two or more relations in each block. Our use of clustering has enhanced processing of a particular join(depositor&cxustomer),but it results in slowing processing of other types of query.
- Requires more block accesses the than it did in the scheme under which we stored each relation in a separate file.

```
select account-number,customer-name,customer-street,customer-city
from depositor,customer
where depositor.customer-name=customer.customer-name
```

- when clustering is to be used depends on the types of query that the database designer believes to be most frequent. Careful use of clustering can produce significant performance gains in query processing.

DATA-DICTIONARY STORAGE:


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

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

Clustering the Structure

Clustering the Structure with Pointer

Hayes	Main	Brooklyn	
Hayes	A-102		
Hayes	A-220		
Hayes	A-503		
Turner	Putnam	Stamford	
Turner	A-305		



- Names of the relations
- Names of the attributes of each relation
- Domains and lengths of attributes
- Names of vies defined on the database, and definitions of those views
- Integrity constraints(For example, key constraints)

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

- Names of authorized users
- Accounting information about users.
- Passwords or other information used to authenticate users

Further, the database may store statistical and descriptive data about the relations such as:

- Number of tuples in each relation
- Method of storage of each relation(for example, clustered or non-clustered)

The data dictionary may also note the storage organization (Sequential, hash or heap of relations, and the location where each relation is stored:

- If relations are stored in operating system files, the dictionary would note the names of the file(or files) containing each relation.
- If the database stores all relations in a single file, the dictionary may note the blocks containing records of each relation in a data structure such as a linked list.

In indices, we shall need to store information about each index on each of the relations:

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

All this information constitutes, in effect, a miniature database. Some database systems store this information by using special-purpose data structure and code. It is generally-preferable to store the data about the database in the database itself.

The exact choice of how to represent system data by relations must be made by the system designers. One possible representation, with primary keys underlined is:

- ✓ Relation-metadata(relation-name, number of attributes, storage-organizational,location)
- ✓ Attribute-metadata(attribute-name, relation-name, domain-type, position, length)
- ✓ User-metadata(user-name, encrypted-password, group)
- ✓ Index-metadata(index-name, relation-name, index-type, index-attribute)
- ✓ View-metadata(view-name, definition)

19.What is meant by Hash function? Also discuss about dynamic hashing

Hash Functions {NOV 2015}

1. A good hash function gives an average-case lookup that is a small constant, independent of the number of search keys.
2. We hope records are distributed uniformly among the buckets.
3. The worst hash function maps all keys to the same bucket.
4. The best hash function maps all keys to distinct addresses.
5. Ideally, distribution of keys to addresses is uniform and random.
6. Suppose we have 26 buckets, and map names beginning with *i*th letter of the alphabet to the *i*th bucket.
 - Problem: this does not give uniform distribution.
 - Many more names will be mapped to "A" than to "X".

- Typical hash functions perform some operation on the internal binary machine representations of characters in a key.
- For example, compute the sum, modulo of buckets, of the binary representations of characters of the search key.
- See Figure 11.18, using this method for 10 buckets (assuming the i th character in the alphabet is represented by integer i).

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

Figure 12.21 Hash organization of *account* file, with *branch-name* as the key.

Handling of bucket overflows

1. we handle bucket overflow by using *overflow buckets*.
2. All the overflow buckets of a given bucket are chained together in a linked list , as in fig 12.22

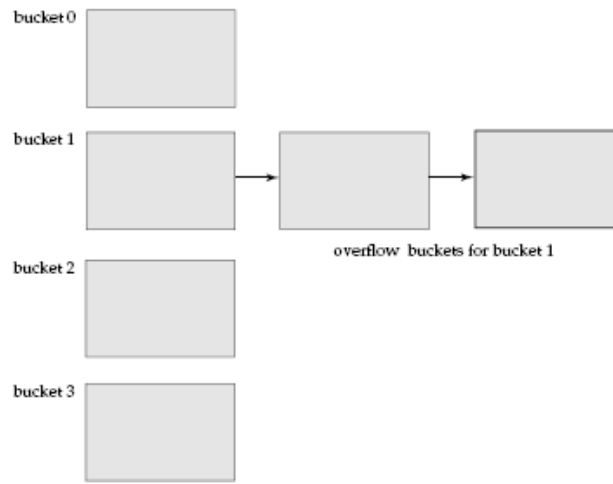


Figure 12.22 Overflow chaining in a hash structure.

3. **Open** hashing occurs where records are stored in different buckets. Compute the hash function and search the corresponding bucket to find a record.
4. **Closed** hashing occurs where all records are stored in **one** bucket. Hash function computes addresses within that bucket. (Deletions are difficult.) Not used much in database applications.
5. **Drawback to our approach:** Hash function must be chosen at implementation time.
 - Number of buckets is fixed, but the database may grow.
 - If number is too large, we waste space.
 - If number is too small, we get too many "collisions", resulting in records of many search key values being in the same bucket.
 - Choosing the number to be twice the number of search key values in the file gives a good space/performance tradeoff.

Hash Indices

1. A hash index organizes the search keys with their associated pointers into a hash file structure.
2. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).
3. Strictly speaking, hash indices are only secondary index structures, since if a file itself is organized using hashing, there is no need for a separate hash index structure on it.

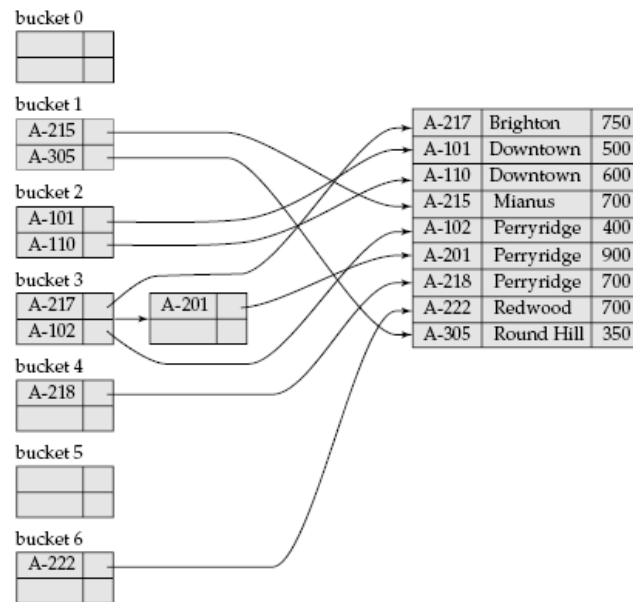


Figure 12.23 Hash index on search key *account-number* of *account* file.

DYNAMIC HASHING

1. As the database grows over time, we have three options:
 - Choose hash function based on current file size. Get performance degradation as file grows.
 - Choose hash function based on anticipated file size. Space is wasted initially.
 - Periodically re-organize hash structure as file grows. Requires selecting new hash function, recomputing all addresses and generating new bucket assignments. Costly, and shuts down database.
2. Some hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinking of the database. These are called **dynamic hash functions**.
 - **Extendable hashing** is one form of dynamic hashing.
 - Extendable hashing splits and coalesces buckets as database size changes.
 - This imposes some performance overhead, but space efficiency is maintained.
 - As reorganization is on one bucket at a time, overhead is acceptably low.
3. How does it work?

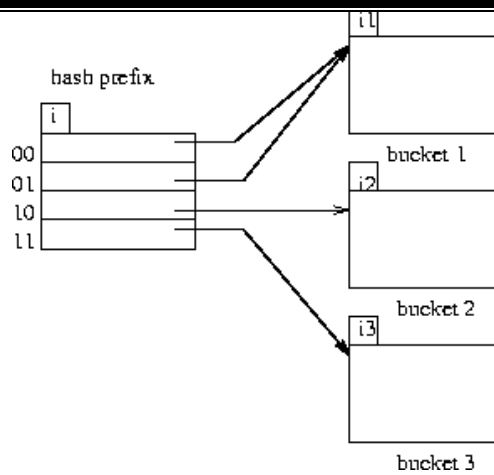


Figure 12.4 General extendable hash structure.

- We choose a hash function that is uniform and random that generates values over a relatively large range.
- Range is b -bit binary integers (typically $b=32$).
- 2^{32} is over 4 billion, so we don't generate that many buckets!
- Instead we create buckets on demand, and do not use all b bits of the hash initially.

$$0 \leq i \leq b$$

- The i bits are used as an offset into a table of bucket addresses.
- Value of i grows and shrinks with the database.
- Figure 12.4 shows an extendable hash structure.
- Note that the i appearing over the bucket address table tells how many bits are required to determine the correct bucket.
- It may be the case that several entries point to the same bucket.
- All such entries will have a common hash prefix, but the length of this prefix may be less than i .
- So we give each bucket an integer giving the length of the common hash prefix.
- The integer associated with bucket j is shown as i_j .
- Number of bucket entries pointing to bucket j is then $2^{(i-i_j)}$.

4. To find the bucket containing search key value K_i :

- Compute $h(K_i)$.
- Take the first i high order bits of $h(K_i)$.
- Look at the corresponding table entry for this i -bit string.
- Follow the bucket pointer in the table entry.

5. We now look at insertions in an extendable hashing scheme.

- Follow the same procedure for lookup, ending up in some bucket j .
- If there is room in the bucket, insert information and insert record in the file.
- If the bucket is full, we must split the bucket, and redistribute the records.
- If bucket is split we may need to increase the number of bits we use in the hash.

6. Two cases exist:

1. If $i = i_j$, then only one entry in the bucket address table points to bucket j .

- Then we need to increase the size of the bucket address table so that we can include pointers to the two buckets that result from splitting bucket j .
- We increment i by one, thus considering more of the hash, and doubling the size of the bucket address table.
- Each entry is replaced by two entries, each containing original value.
- Now two entries in bucket address table point to bucket j .
- We allocate a new bucket z , and set the second pointer to point to z .
- Set i_j and i_z to i .
- Rehash all records in bucket j which are put in either j or z .
- Now insert new record.
- It is remotely possible, but unlikely, that the new hash will still put all of the records in one bucket.
- If so, split again and increment i again.

2. If $i > i_j$, then more than one entry in the bucket address table points to bucket j .

- Then we can split bucket j without increasing the size of the bucket address table
- Note that all entries that point to bucket j correspond to hash prefixes that have the same value on the leftmost i_j bits.
- We allocate a new bucket z , and set i_j and i_z to the original i_j value plus 1.
- Now adjust entries in the bucket address table that previously pointed to bucket j .
- Leave the first half pointing to bucket j , and make the rest point to bucket z .
- Rehash each record in bucket j as before.
- Reattempt new insert.

7. Note that in both cases we only need to rehash records in bucket j .

8. Deletion of records is similar. Buckets may have to be coalesced, and bucket address table may have to be halved.
9. Insertion is illustrated for the example *deposit* file.

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Figure 12.25 Sample *account* file.

- 32-bit hash values on *branch-name* are shown in Figure

<i>branch-name</i>	$h(\text{branch-name})$
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

Figure 12.26 Hash function for *branch-name*.

- . An initial empty hash structure is shown in Figure



Figure 12.27 Initial extendable hash structure.

- We insert records one by one.
- We (unrealistically) assume that a bucket can only hold 2 records, in order to illustrate both situations described.
- As we insert the Perryridge and Round Hill records, this first bucket becomes full.
- When we insert the next record (Downtown), we must split the bucket.
- Since $i = i_0$, we need to increase the number of bits we use from the hash.
- We now use 1 bit, allowing us $2^1 = 2$ buckets.
- This makes us double the size of the bucket address table to two entries.

- We split the bucket, placing the records whose search key hash begins with 1 in the new bucket, and those with a 0 in the old bucket (Figure 11.23).
- Next we attempt to insert the Redwood record, and find it hashes to 1.
- That bucket is full, and $i = i_1$.
- So we must split that bucket, increasing the number of bits we must use to 2.
- This necessitates doubling the bucket address table again to four entries (Figure 12.28).

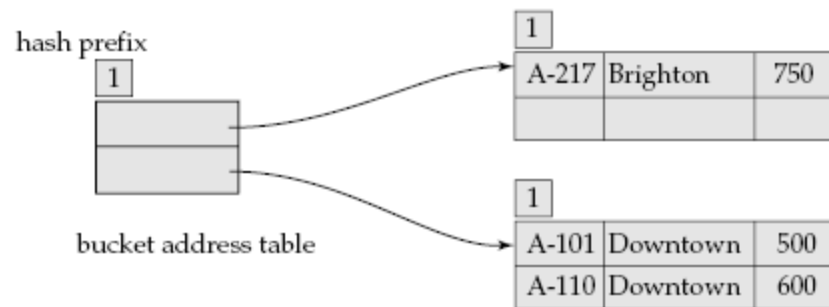


Figure 12.28 Hash structure after three insertions.

- We rehash the entries in the old bucket.
- We continue on for the deposit records of Figure 12.25, obtaining the extendable hash structure of Figure 12.31

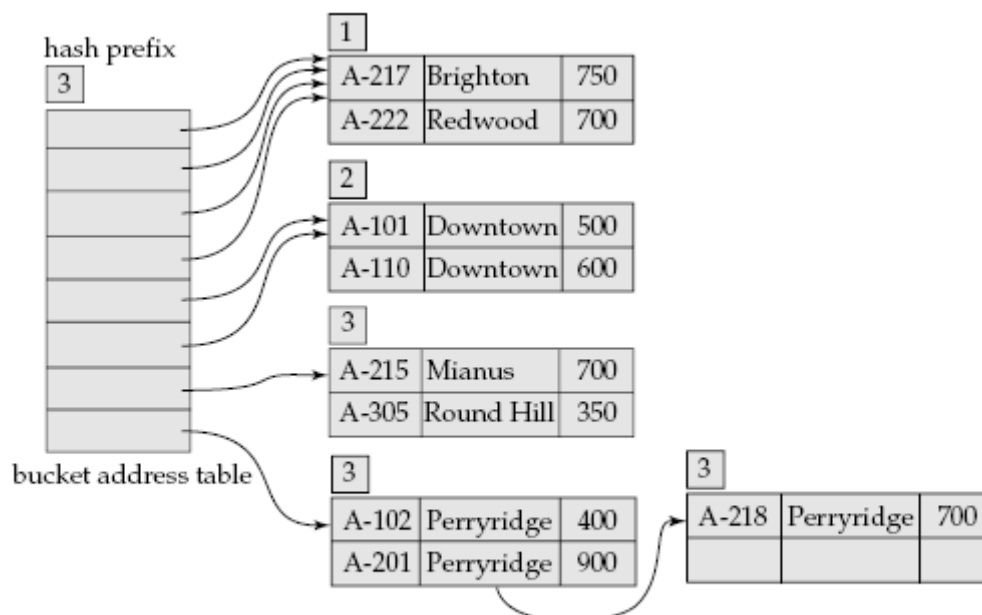


Figure 12.31 Extendable hash structure for the *account* file.

Advantages:

- Extendable hashing provides performance that does not degrade as the file grows.

- Minimal space overhead - no buckets need be reserved for future use. Bucket address table only contains one pointer for each hash value of current prefix length.

Disadvantages:

- Extra level of indirection in the bucket address table
- Added complexity

20 .Explain B-Tree index files in detail

(NOV 2010)

B+ TREE INDEX FILES

1. Primary disadvantage of index-sequential file organization is that performance degrades as the file grows. This can be remedied by costly re-organizations.
2. B +-tree file structure maintains its efficiency despite frequent insertions and deletions. It imposes some acceptable update and space overheads.
3. A B +-tree index is a *balanced tree* in which every path from the root to a leaf is of the same length.
4. Each nonleaf node in the tree must have between $\lceil n/2 \rceil$ and n children, where n is fixed for a particular tree.

Structure of a B+-Tree

1. A B +-tree index is a *multilevel* index but is structured differently from that of multi-level index sequential files.

A typical node (Figure 12.6) contains up to $n-1$ search key values K_1, K_2, \dots, K_{n-1} , and n pointers P_1, P_2, \dots, P_n . Search key values in a node are kept in sorted order.



Figure 12.6 Typical node of a B⁺-tree.

2. For leaf nodes, P_i ($i = 1, \dots, n-1$) points to either a file record with search key value K_i , or a bucket of pointers to records with that search key value. Bucket structure is used if search key is not a primary key, and file is not sorted in search key order.
Pointer P_n (n th pointer in the leaf node) is used to chain leaf nodes together in linear order (search key order). This allows efficient **sequential** processing of the file.
The range of values in each **leaf** do not overlap.
3. Non-leaf nodes form a multilevel index on leaf nodes.

A non-leaf node may hold up to n pointers and must hold $\lceil n/2 \rceil$ pointers. The number of pointers in a node is called the *fan-out* of the node.

Consider a node containing m pointers. Pointer P_i ($i = 2, \dots, m$) points to a subtree containing search key values $\geq K_{i-1}$ and $< K_i$. Pointer P_m points to a subtree containing search key values $\geq K_{m-1}$. Pointer P_1 points to a subtree containing search key values $< K_1$.

4. Figures 11.7 (textbook Fig. 11.8) and textbook Fig. 11.9 show B+-trees for the *deposit* file with $n=3$ and $n=5$.

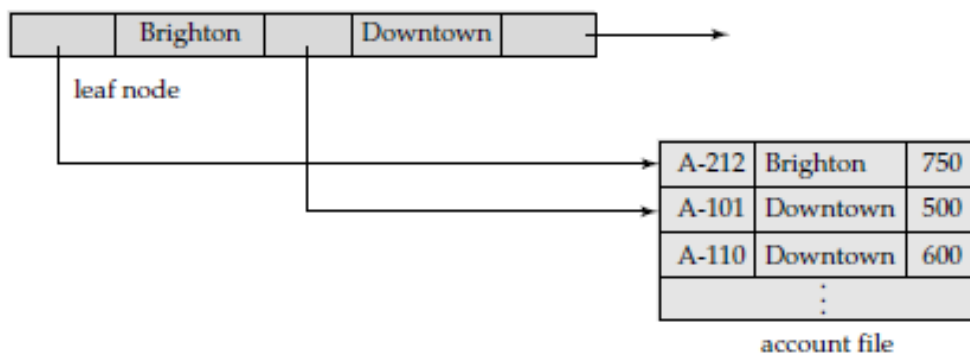


Figure 12.7 A leaf node for *account* B⁺-tree index ($n = 3$).

Queries on B+-Trees

- Suppose we want to find all records with a search key value of k .
 - Examine the root node and find the smallest search key value $K_i > k$.
 - Follow pointer P_i to another node.
 - If $k < K_1$ follow pointer P_1 .
 - Otherwise, find the appropriate pointer to follow.
 - Continue down through non-leaf nodes, looking for smallest search key value $> k$ and following the corresponding pointer.
 - Eventually we arrive at a leaf node, where pointer will point to the desired record or bucket.
- In processing a query, we traverse a path from the root to a leaf node. If there are K search key values in the file, this path is no longer than $\log_{\lceil n/2 \rceil}(K)$.

This means that the path is not long, even in large files. For a 4k byte disk block with a search-key size of 12 bytes and a disk pointer of 8 bytes, n is around 200. If $n=100$, a look-up of 1 million search-key values may take $\log_{50}(1,000,000) = 4$ nodes

to be accessed. Since root is usually in the buffer, so typically it takes only 3 or fewer disk reads.

Updates on B+-Trees

1. Insertions and Deletions:

Insertion and **deletion** are more complicated, as they may require splitting or combining nodes to keep the tree balanced. If splitting or combining are not required, insertion works as follows:

- Find leaf node where search key value should appear.
- If value is present, add new record to the bucket.
- If value is not present, insert value in leaf node (so that search keys are still in order).
- Create a new bucket and insert the new record.

If splitting or combining are not required, deletion works as follows:

- Deletion: Find record to be deleted, and remove it from the bucket.
- If bucket is now empty, remove search key value from leaf node.

2. Insertions Causing Splitting:

When insertion causes a leaf node to be too large, we **split** that node. In Figure 11.8, assume we wish to insert a record with a *bname* value of "Clearview".

- There is no room for it in the leaf node where it should appear.
- We now have n values (the $n-1$ search key values plus the new one we wish to insert).
- We put the first $\lceil n/2 \rceil$ values in the existing node, and the remainder into a new node.
- Figure 11.10 shows the result.
- The new node must be inserted into the B+-tree.
- We also need to update search key values for the parent (or higher) nodes of the split leaf node. (Except if the new node is the leftmost one)
- Order must be preserved among the search key values in each node.
- If the parent was already full, it will have to be split.
- When a non-leaf node is split, the children are divided among the two new nodes.
- In the worst case, splits may be required all the way up to the root. (If the root is split, the tree becomes one level deeper.)

- **Note:** when we start a B +-tree, we begin with a single node that is both the root and a single leaf. When it gets full and another insertion occurs, we split it into two leaf nodes, requiring a new root.

3. Deletions Causing Combining:

Deleting records may cause tree nodes to contain too few pointers. Then we must combine nodes.

- If we wish to delete "Downtown" from the B +-tree of Figure 11.11, this occurs.
- In this case, the leaf node is empty and must be deleted.
- If we wish to delete "Perryridge" from the B +-tree of Figure 11.11, the parent is left with only one pointer, and must be coalesced with a sibling node.
- Sometimes higher-level nodes must also be coalesced.
- If the root becomes empty as a result, the tree is one level less deep (Figure 11.13).
- Sometimes the pointers must be redistributed to keep the tree balanced.
- Deleting "Perryridge" from Figure 11.11 produces Figure 11.14.

4. To summarize:

- Insertion and deletion are complicated, but require relatively few operations.
- Number of operations required for insertion and deletion is proportional to logarithm of number of search keys.
- B +-trees are fast as index structures for database.

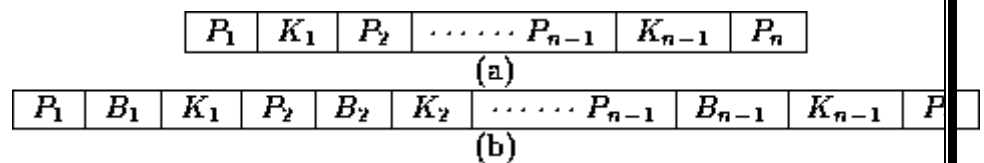
B +-Tree File Organization

1. The B +-tree structure is used not only as an index but also as an organizer for records into a file.
2. In a B +-tree file organization, the leaf nodes of the tree store records instead of storing pointers to records, as shown in Fig. 11.17.
3. Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is less than the maximum number of pointers in a non leaf node.
4. However, the leaf node are still required to be at least half full.
5. Insertion and deletion from a B +-tree file organization are handled in the same way as that in a B +-tree index.

6. When a B +-tree is used for file organization, space utilization is particularly important. We can improve the space utilization by involving more sibling nodes in redistribution during splits and merges.
7. In general, if m nodes are involved in redistribution, each node can be guaranteed to contain at least $\lceil (m-1)n/m \rceil$ entries. However, the cost of update becomes higher as more siblings are involved in redistribution.

B-TREE INDEX FILES

1. B-tree indices are similar to B +-tree indices.
 - Difference is that B-tree eliminates the redundant storage of search key values.
 - In B +-tree, some search key values appear twice.
 - A corresponding B-tree of Figure 11.18 allows search key values to appear only once.
 - Thus we can store the index in less space.



Leaf and non leaf node of a B-tree.

2. Advantages:

- Generally, the structural simplicity of B +-tree is preferred.
- Lack of redundant storage (but only marginally different).
- Some searches are faster (key may be in non-leaf node).

3. Disadvantages:

- Leaf and non-leaf nodes are of different size (complicates storage)
- Deletion may occur in a non-leaf node (more complicated)

UNIVERSITY QUESTIONS

1. Explain 4 NF and 5 NF with examples **(11 marks)** APRIL 2010, **(Question Number:14)**.
2. Discuss Assertions, Triggers, Authentication (11 Marks) **APRIL 2010, (Question Number:6 and 7)**.
3. Explain any one database system of your own with normalization **(11 marks)** NOV 2010, **(Question Number:10)**.
4. Explain referential integrity in detail **(11 marks)** NOV 2010, **(Question Number:10)**.
5. Explain database design process in detail. **(11 marks)** APRIL 2011, **(Question Number:15)**.
6. Explain Authorization in SQL with example-Refer First unit notes **11 marks (APRIL 2011, (Question Number:5))**.
7. Write about ER Diagrams(11 marks)" Refer First unit notes" **11 marks APRIL 2012,NOV 2012,APR 2014, NOV 2015(Question Number:15)**
8. Explain any two normal forms in detail. **11 marks** APRIL 2012, **(Question Number: 10)**.
9. Explain Decomposition using Functional Dependencies.(11 marks) NOV 2012, **(Question Number:10)**.
10. Explain 3 NF and 4 NF with examples **(11 marks)** APR 2014, **(Question Number:13,14)**.
11. Explain the Fundamentals of database design and ER model. NOV 2014, **(Question Number:9)**.
12. Explain in details about application design and development. Nov2014, **(Question Number:15)**.
13. Discuss the various concepts of ER Model with an example. Apr 2015, **(Question Number:16)**.
14. Explain about Database Design Process. Apr 2015, **(Question Number:17)**.
15. Explain organization of records in files structure in detail. Apr 2011, **(Question Number:18)**.
16. Explain B-Tree index files in detail. NOV 2010, **(Question Number:20)**
17. What is meant by Hash function? Also discuss about dynamic hashing
Hash Functions {NOV 2015}(Page 66) **(Question Number:19)**

UNIT -III

Relational Database Design: Features of Good Relational Designs – Atomic Domains and First Normal Form – Second Normal Form – Decomposition Using Functional Dependencies – Functional Dependency Theory – Algorithms for decomposition – Decomposition Using Multi-valued Dependencies – More Normal Forms – Database – Design Process – Modeling Temporal Data

TWO MARK

1. What is normalization? (NOV 2015)

The main goal of normalization is to reduce redundant data. Normalization is based on functional dependencies.

2. What is First Normal Form?(NOV 2010)

First normal form is also called as flat file. There are no composite attributes and every attribute is single and describe one property.

3. What is functional dependencies?(NOV 2015)

Functional dependencies play key role in differentiating good database designs from bad database design. A functional dependency is a type of constraint that is a generalization of the notion of key.

4. What is decomposition?

The process of decomposing a relation schema that has many attributes into several schemas with fewer attributes is called decomposition.

5. What is Boyce Codd Normal Form? APRIL 2014

A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, atleast one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a superkey for schema R .

6. What is Third Normal Form?

A relation is said to be Third normal form where all attributes in a relation tuple are not only functionally dependent on key attributes but also dependent on non key attributes.

7. What is Second Normal Form?

A relation is said to be in Second normal form, if it is in first normal form and non key attributes are functionally dependent on the key attributes.

8. What is Fourth Normal Form?

This schema is called as non- BCNF schema. A relation defined with multi valued dependency which is a new form of constraint is called forth normal form. It is more restrictive than BCNF.

9. What is multi valued dependency?

Multivalued dependencies are referred to as tuple generating dependencies. It do not rule out existence of certain tuples, instead they require other tuples of certain form be present in the relation.

10. Comparison of BCNF and 3CNF.

3CNF	BCNF
<ul style="list-style-type: none"> • <u>3NF</u>: For every functional dependency $X \rightarrow Y$ in a set F of functional dependencies over relation R, either: Y is a subset of X or, X is a <i>superkey</i> of R, or Y is a subset of K for some key K of R. <i>N.b.</i>, no subset of a key is a key. • It is always possible to obtain a 3NF design without sacrificing lossless-join or dependency preservation. • If we do not eliminate all transitive dependencies, we may need to use null values to represent some of the meaningful relationships. • Repetition of information occurs. 	<ul style="list-style-type: none"> • <u>BCNF</u>: For every functional dependency $X \rightarrow Y$ in a set F of functional dependencies over relation R, either: Y is a subset of X or, X is a <i>superkey</i> of R • If we cannot check for dependency preservation efficiently, we either pay a high price in system performance or risk the integrity of the data. • The limited amount of redundancy in 3NF is then a lesser evil.

11. Define Functional Dependencies.

Let $X \subseteq R$ and $Y \subseteq R$, then the Functional dependency (FD) $X \rightarrow Y$ holds on R if, in any relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[X] = t_2[X]$, it is also the case that $t_1[Y] = t_2[Y]$.

12. Define Closure of Functional Dependency.

Let F be a set of FDs, then the closure of F is the set of all FDs logically implied by F . It is denoted by F^+

13. Define normalization of data and denormalization.

Normalization:

Process of decomposing an unsatisfactory relation schema into satisfactory relation schemas by breaking their attributes, so as to satisfy the desirable properties. Denormalization: Process of storing the join of higher normal form relations as base relation, which is in the lower normal form.

14. Explain shortly the four properties or objectives of normalization.

- To minimize redundancy
- To minimize insertion, deletion, and updating anomalies.
- Lossless-join or non-additive join decomposition
- Dependency preservation

15. Define Partial Functional Dependency.

A FD $x \rightarrow y$ is a partial dependency if some attribute $A \in x$ can be removed from x and the dependency still holds; ie., for some $A \in x$, $(x - \{A\}) \rightarrow y$.

16. Define Boyce codd normal form

A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F of the form $a \rightarrow \beta$, where a

17. List the disadvantages of relational database system

Repetition of data

Inability to represent certain information.

18. What is first normal form?

The domain of attribute must include only atomic (simple, indivisible) values.

19. What are the uses of functional dependencies?

To test relations to see whether they are legal under a given set of functional dependencies. To specify constraints on the set of legal relations.

20. Explain trivial dependency?

Functional dependency of the form $a \rightarrow \beta$ is trivial if $\beta \subset a$. Trivial functional dependencies are satisfied by all the relations.

21. What are axioms?

Axioms or rules of inference provide a simpler technique for reasoning about functional dependencies.

22. What is meant by computing the closure of a set of functional dependency?

+ The closure of F denoted by F^+ is the set of functional dependencies logically implied by F.

23. What is meant by normalization of data?

It is a process of analyzing the given relation schemas based on their Functional Dependencies (FDs) and primary key to achieve the properties Minimizing redundancy Minimizing insertion, deletion and updating anomalies

24. Explain the desirable properties of decomposition.

a) Lossless-join decomposition b) Dependency preservation c) Repetition of information

25. What is 2NF?

A relation schema R is in 2NF if it is in 1NF and every non-prime attribute A in R is fully functionally dependent on primary key.

26. Define Data storage?

It stores application data in database. The place where data is backed up.

27. Define Access time and seek time?

It is the time for when a read or write request is issued to when data transfer begins. The time for repositioning the arm is called the seek time.

28. Define the average seek time?

It is the average of the seek times, measured over a sequence of a random requests.

29. Define rotational latency time?

If the head has reached the desired track, the time spent waiting for the sector to be accessed appear under the head is called rotational latency time.

30. Define MTTF?

It is used to measure the reliability of the disk. The mean time to failure of a disk is the amount of time that, on average, that expect the system to run continuously without any failure.

31. What is meant by RAID?

A variety of disk organization techniques, collectively called Redundant Array Of Independent Disks have been proposed to achieve improved performance and reliability.

32. Define Redundancy?

To store extra information that is not needed normally, but that can be used in the event of failure of the disk to rebuild the lost information.

33. What is meant by mirroring?

The simplest approach to introducing redundancy is to duplicate every disk. This technique is called mirroring or shadowing.

34. What is Mean time to repair?

It is the time it takes to replace the failed disk and to restore the data on it is called as mean time to repair.

35. Define striping?

In multiple disk , it is used to improve the transfer rate by striping data across multiple disk . It can be done by 2 methods.

1.Bit level Striping

2.Block level striping

36. What is bit level striping and block level striping?

Bit level striping:

Data striping consists of splitting the bits of each byte across multiple disk.

Block level striping:

Strips block across multiple disks. It treats the array of disks as a single large disk , and gives blocks logical number.

37. What are the different levels of RAID?

RAID level 0, RAID level 1, RAID level 2, RAID level 3, RAID level 4, RAID level 5 ,RAID level 6.

38. Define RAID level 0?

It refers to disk arrays with striping at the level of blocks but without any redundancy.

39. Define RAID level 1 and level 2?

RAID level 1: It refers disk mirroring with block striping.

RAID level 2: It is known as memory-style error-correcting-code (ECC) organization , Employs parity bit.

40. Define RAID level 3 and RAID level 4 :

RAID level 3: It is bit interleaved parity organization. It supports a lower number of IO operations per second.

RAID level 4: It is block interleaved parity organization , uses block level striping like RAID 0 and in addition keeps a parity block on a separate disk for corresponding block from N other disk.

41. Define RAID level 5 and RAID level 6:

RAID level 5: It is block interleaved distributed parity , improves on level 4 by partitioning data and parity among all N+1 disks , instead of storing data in N is disk and parity in one disk.

RAID level 6: It is P+Q redundancy scheme , is much like RAID level 5 but stores extra redundant information to guard against multiple disk failures.

42. What is mean by FILE?

File is organized logically as a sequence of records . A record is a collection of fields.

43. Define Fixed length records?

Blocks which are of fixed size is determined by the physical properties of the disk and by the operating system record sizes vary.

44. Define Variable length records?

It arises in database systems in several ways .

1. Storage of multiple record types in a file.
2. Record type that allows variable lengths for one or more fields.
3. Record types that allow repeating fields, such as arrays or multiset.

45. Define Slotted _page structure?

It is commonly used for organizing records within a block. The actual records are allocated contiguously in the block.

46. Define BLOB and CLOB?

BLOB: Binary large object

CLOB: Character large object.

47. What is heap file organization?

Any record can be placed anywhere in the file where there is a space for the record .there is no ordering if records.

48. Define sequential file organization?

Records are stored in the sequential order, according to the value of search key of each record.

49. Define hashing file organization?

A hash function is computed on some attributes of each records. The result of the hash function specifies in which block of the file the record should be placed.

50. Define Search key?

A search key is any attribute or set of attribute , it need not be the primary key or a even a super key.

51. Define data dictionary?

The relational database system needs to maintain data about the relations, such as schema of the relations. This information is called the data dictionary or system catalog.

52. What factors must be taken care while choosing RAID level?

- Monetary cost of extra disk storage requirements
- Performance requirements in terms of number of I/O operations
- Performance when a disk has failed
- Performance during rebuild

53. What is jukeboxes?

Jukeboxes are devices that store a large number of optical disk and load them automatically on demand to one of the small number drives.

54. Define hot swapping?

The faulty disks can be removed and replaced by the new ones without turning power off. Hot swapping the mean time to repair.

55. What are the buffer manager virtual schemes?

- Buffer replacement strategy
- Pinned blocks
- Forced output of blocks

56. Why is multiple key access used?

In certain type of queries it is advantageous to use multiple indices if they exist ,or to use an index built on a multi-attribute search key.

57. Define over flow chaining?

If a record must be inserted into a bucket b and b is already full, the system provides another overflow bucket for b and insert the record into the over flow bucket .if the over flow bucket is also full the system provides another over flow bucket and so on. All the over flow buckets are chained together in a linked list. Such a link list is called over flow chaining.

58. Mention about sparse index (April/May 2012)

An index record appears for only some of the search key values in a file.

59. What is meant by Dependency preservation? (April 2013)

Another desirable property in database design is **dependency preservation**.

- We would like to check easily that updates to the database do not result in illegal relations being created.
- It would be nice if our design allowed us to check updates without having to compute natural joins.
- To know whether joins must be computed, we need to determine what functional dependencies may be tested by checking each relation individually.
- Let F be a set of functional dependencies on schema R .
- Let $\{R_1, R_2, \dots, R_n\}$ be a decomposition of R .

60. State the conditions for a relation R to be in 3NF (April 2013)

A relation R is in 3NF if and only if for every non trivial FD $A \rightarrow B$ for R , one of the following two conditions are true:

- A is a super key for R

- B is an attribute in some keys

61. When we say a relation is in first Normal form? (Nov 2010)

1NF A relation R is in first normal form (1NF) if and only if all underlying domains contain atomic values only.

Example: 1NF but not 2NF

FIRST (supplier_no, status, city, part_no, quantity)

Functional Dependencies:

(supplier_no, part_no) → quantity

(supplier_no) → status

(supplier_no) → city

city → status (Supplier's status is determined by location)

11 MARKS

10.Explain in detail about Relational database design? (11 Marks)

First Normal Form

A domain is **atomic** if elements of the domain are considered to be indivisible units. We say that a relation schema R is in **first normal form** (1NF) if the domains of all attributes of R are atomic. A set of names is an example of a nonatomic value. For example, if the schema of a relation employee included an attribute children whose domain elements are sets of names, the schema would not be in first normal form. Composite attributes, such as an attribute address with component attributes street and city, also have nonatomic domains. Integers are assumed to be atomic, so the set of integers is an atomic domain; the set of all sets of integers is a nonatomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts namely, the integers making up the set.

The domain of all integers would be nonatomic if we considered each integer to be an ordered list of digits.

Some types of nonatomic values can be useful, although they should be used with care. For example, composite valued attributes are often useful, and set valued attributes are also useful in many cases, which is why both are supported in the E-R model.

Pitfalls in Relational-Database Design

Among the undesirable properties that a bad design may have are:

- Repetition of information
- Inability to represent certain information

Suppose the information concerning loans is kept in one single relation, lending, which is defined over the relation schema

Lending-schema = (branch-name, branch-city, assets, customer-name, loan-number, amount)

A tuple t in the lending relation has the following intuitive meaning:

- t[assets] is the asset figure for the branch named t[branch-name].

- $t[\text{branch-city}]$ is the city in which the branch named $t[\text{branch-name}]$ is located.
- $t[\text{loan-number}]$ is the number assigned to a loan made by the branch named $t[\text{branch-name}]$ to the customer named $t[\text{customer-name}]$.
- $t[\text{amount}]$ is the amount of the loan whose number is $t[\text{loan-number}]$.

Suppose that we wish to add a new loan to our database. Say that the loan is made by the Perryridge branch to Adams in the amount of \$1500. Let the loan-number be L-31. In our design, we need a tuple with values on all the attributes of Lendingschema. Thus, we must repeat the asset and city data for the Perryridge branch, and must add the tuple

(Perryridge, Horseneck, 1700000, Adams, L-31, 1500)

Another problem with the Lending-schema design is that we cannot represent directly the information concerning a branch (branch-name, branch-city, assets) unless there exists at least one loan at the branch. This is because tuples in the lending relation require values for loan-number, amount, and customer-name. One solution to this problem is to introduce null values, as we did to handle updates through views.

<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Sample lending relation.

Functional Dependencies

Basic Concepts

Functional dependencies are constraints on the set of legal relations. They allow us to express facts about the enterprise that we are modeling with our database.

Let R be a relation schema. A subset K of R is a **superkey** of R if, in any legal relation $r(R)$, for all pairs t_1 and t_2 of tuples in r such that $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$. That is, no two tuples in any legal relation $r(R)$ may have the same value on attribute set K . The notion of functional dependency generalizes the notion of superkey. Consider a relation schema R , and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **functional dependency** $\alpha \rightarrow \beta$

holds on schema R if, in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Using the functional-dependency notation, we say that K is a superkey of R if $K \rightarrow R$. That is, K is a superkey if, whenever $t_1[K] = t_2[K]$, it is also the case that $t_1[R] = t_2[R]$ (that is, $t_1 = t_2$). Functional dependencies allow us to express constraints that we cannot express with superkeys. Consider the schema

Loan-info-schema = (loan-number, branch-name, customer-name, amount)

which is simplification of the Lending-schema that we saw earlier. The set of functional dependencies that we expect to hold on this relation schema is

loan-number \rightarrow amount

loan-number \rightarrow branch-name

We shall use functional dependencies in two ways:

1. To test relations to see whether they are legal under a given set of functional dependencies. If a relation r is legal under a set F of functional dependencies, we say that r **satisfies** F .
2. To specify constraints on the set of legal relations. We shall thus concern ourselves with only those relations that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on schema R that satisfy a set F of functional dependencies, we say that F **holds** on R .

To distinguish between the concepts of a relation satisfying a dependency and a dependency holding on a schema, we return to the banking example. In the loan relation (on Loan-schema) we see that the dependency $\text{loan-number} \rightarrow \text{amount}$ is satisfied. In contrast to the case of customer-city and customer-street in Customer-schema.

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

The customer relation.

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-17	Downtown	1000
L-23	Redwood	2000
L-15	Perryridge	1500
L-14	Downtown	1500
L-93	Mianus	500
L-11	Round Hill	900
L-29	Pownal	1200
L-16	North Town	1300
L-18	Downtown	2000
L-25	Perryridge	2500
L-10	Brighton	2200

The loan relation.

In the branch relation of Figure 7.5, we see that $\text{branch-name} \rightarrow \text{assets}$ is satisfied, as is $\text{assets} \rightarrow \text{branch-name}$. We want to require that $\text{branch-name} \rightarrow \text{assets}$ hold on Branch-schema. However, we do not wish to require that $\text{assets} \rightarrow \text{branch-name}$ hold, since it is possible to have several branches that have the same asset value.

<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
Pownal	Bennington	300000
North Town	Rye	3700000
Brighton	Brooklyn	7100000

The branch relation.

- On Branch-schema: $\text{branch-name} \rightarrow \text{branch-city}$

branch-name \rightarrow assets

- On Customer-schema: customer-name \rightarrow customer-city
customer-name \rightarrow customer-street
- On Loan-schema: loan-number \rightarrow amount
loan-number \rightarrow branch-name
- On Borrower-schema: No functional dependencies
- On Account-schema: account-number \rightarrow branch-name
account-number \rightarrow balance
- On Depositor-schema: No functional dependencies

Closure of a Set of Functional Dependencies

More formally, given a relational schema R, a functional dependency f on R is **logically implied** by a set of functional dependencies F on R if every relation instancer(R) that satisfies F also satisfies f.

Suppose we are given a relation schema R = (A, B, C, G, H, I) and the set of functional dependencies

A \rightarrow B

A \rightarrow C

CG \rightarrow H

CG \rightarrow I

B \rightarrow H

The functional dependency

A \rightarrow H

is logically implied. That is, we can show that, whenever our given set of functional dependencies holds on a relation, A \rightarrow H must also hold on the relation. Suppose that t1 and t2 are tuples such that

t1[A] = t2[A]

Since we are given that A \rightarrow B, it follows from the definition of functional dependency that

t1[B] = t2[B]

Then, since we are given that B \rightarrow H, it follows from the definition of functional dependency that

t1[H] = t2[H]

Therefore, we have shown that, whenever t_1 and t_2 are tuples such that $t_1[A] = t_2[A]$, it must be that $t_1[H] = t_2[H]$. But that is exactly the definition of $A \rightarrow H$.

We can use the following three rules to find logically implied functional dependencies. By applying these rules repeatedly, we can find all of F^+ , given F . This collection of rules is called **Armstrong's axioms** in honor of the person who first proposed it.

- **Reflexivity rule.** If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- **Augmentation rule.** If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule.** If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Although Armstrong's axioms are complete, it is tiresome to use them directly for the computation of F^+ . To simplify matters further, we list additional rules. It is possible to use Armstrong's axioms to prove that these rules are correct.

- **Union rule.** If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition rule.** If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- **Pseudotransitivity rule.** If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Closure of Attribute Sets

To test whether a set α is a superkey, we must devise an algorithm for computing the set of attributes functionally determined by α . One way of doing this is to compute F^+ , take all functional dependencies with α as the left-hand side, and take the union of the right-hand sides of all such dependencies. However, doing so can be expensive, since F^+ can be large.

$F^+ = F$

repeat

for each functional dependency f in F^+

apply reflexivity and augmentation rules on f

add the resulting functional dependencies to F^+

for each pair of functional dependencies f_1 and f_2 in F^+

iff f_1 and f_2 can be combined using transitivity

add the resulting functional dependency to F^+

until F^+ does not change any further

Let α be a set of attributes. We call the set of all attributes functionally determined by α under a set F of functional dependencies the **closure** of α under F ; we denote it by α^+ . The input is a set F of functional dependencies and the set α of attributes. The output is stored in the variable `result`. The first time that we execute the **while** loop to test each functional dependency, we find that

- $A \rightarrow B$ causes us to include B in `result`. To see this fact, we observe that $A \rightarrow B$ is in F , $A \subseteq \text{result}$ (which is AG), so `result` := `result` $\cup B$.
- $A \rightarrow C$ causes `result` to become $ABCG$.
- $CG \rightarrow H$ causes `result` to become $ABCGH$.
- $CG \rightarrow I$ causes `result` to become $ABCGHI$.

It turns out that, in the worst case, this algorithm may take an amount of time quadratic in the size of F . There is a faster (although slightly more complex) algorithm that runs in time linear in the size of F .

```
result :=  $\alpha$ ;
```

```
while(changes to result) do
```

```
  for each functional dependency  $\beta \rightarrow \gamma$  in  $F$  do
```

```
    begin
```

```
      if  $\beta \subseteq \text{result}$  then result := result  $\cup \gamma$ ;
```

```
    end
```

There are several uses of the attribute closure algorithm:

- To test if α is a superkey, we compute α^+ , and check if α^+ contains all attributes of R .
- We can check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), by checking if $\beta \subseteq \alpha^+$. That is, we compute α^+ by using attribute closure, and then check if it contains β .
- It gives us an alternative way to compute F^+ : For each $\gamma \subseteq R$, we find the closure γ^+ , and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover

Suppose that we have a set of functional dependencies F on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies, that is, all the functional dependencies in F are satisfied in the new database state. The system must roll back the update if it violates any functional dependencies in the set F .

An attribute of a functional dependency is said to be **extraneous** if we can remove it without changing the closure of the set of functional dependencies. The formal definition of **extraneous attributes** is as follows. Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .

- Attribute A is extraneous in α if $A \in \alpha$, and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- Attribute A is extraneous in β if $A \in \beta$, and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F .

Consider an attribute A in a dependency $\alpha \rightarrow \beta$.

- If $A \in \beta$, to check if A is extraneous consider the set $F_- = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ and check if $\alpha \rightarrow A$ can be inferred from F_- . To do so, compute α^+ (the closure of α) under F_- ; if α^+ includes A , then A is extraneous in β .
- If $A \in \alpha$, to check if A is extraneous, let $\gamma = \alpha - \{A\}$, and check if $\gamma \rightarrow \beta$ can be inferred from F . To do so, compute γ^+ (the closure of γ) under F ; if γ^+ includes all attributes in β , then A is extraneous in α .

A **canonical cover** F_c for F is a set of dependencies such that F logically implies all dependencies in F_c , and F_c logically implies all dependencies in F . Furthermore, F_c must have the following properties:

- No functional dependency in F_c contains an extraneous attribute.
- Each left side of a functional dependency in F_c is unique. That is, there are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in F_c such that $\alpha_1 = \alpha_2$.

11.Explain Decomposition? (11 Marks)

Consider an alternative design in which we decompose Lending-schema into the following two schemas:

Branch-customer-schema = (branch-name, branch-city, assets, customer-name)

Customer-loan-schema = (customer-name, loan-number, amount)

Using the lending relation of Figure 7.1, we construct our new relations branch-customer (Branch-customer) and customer-loan (Customer-loan-schema):

branch-customer = Π branch-name, branch-city, assets, customer-name (lending)

customer-loan = Π customer-name, loan-number, amount (lending)

It appears that we can do so by writing

branch-customer \bowtie customer-loan

<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks

The relation branch-customer.

<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-18	2000
Glenn	L-25	2500
Brooks	L-10	2200

The relation customer-loan.

<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

The relation **branch-customer** \square **customer-loan**.

Desirable Properties of Decomposition

We illustrate our concepts with the Lendingschema

Lending-schema = (branch-name, branch-city, assets, customer-name, loan-number, amount)

The set F of functional dependencies that we require to hold on Lending-schema are

branch-name \rightarrow branch-city assets

loan-number \rightarrow amount branch-name

Lending-schema is an example of a bad database design. Assume that we decompose it to the following three relations:

Branch-schema = (branch-name, branch-city, assets)

Loan-schema = (loan-number, branch-name, amount)

Borrower-schema = (customer-name, loan-number)

Lossless-Join Decomposition

Let R be a relation schema, and let F be a set of functional dependencies on R. Let R1 and R2 form a decomposition of R. This decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies is in F+:

- $R1 \cap R2 \rightarrow R1$
- $R1 \cap R2 \rightarrow R2$

In other words, if $R1 \cap R2$ forms a superkey of either $R1$ or $R2$, the decomposition of R is a lossless-join decomposition.

We now demonstrate that our decomposition of Lending-schema is a lossless-join decomposition by showing a sequence of steps that generate the decomposition. We begin by decomposing Lending-schema into two schemas:

Branch-schema = (branch-name, branch-city, assets)

Loan-info-schema = (branch-name, customer-name, loan-number, amount)

Since $\text{branch-name} \rightarrow \text{branch-city assets}$, the augmentation rule for functional dependencies implies that

$\text{branch-name} \rightarrow \text{branch-name branch-city assets}$

Since $\text{Branch-schema} \cap \text{Loan-info-schema} = \{\text{branch-name}\}$, it follows that our initial decomposition is a lossless-join decomposition. Next, we decompose Loan-info-schema into

Loan-schema = (loan-number, branch-name, amount)

Borrower-schema = (customer-name, loan-number)

This step results in a lossless-join decomposition, since loan-number is a common attribute and **$\text{loan-number} \rightarrow \text{amount branch-name}$** .

Dependency Preservation

To decide whether joins must be computed to check an update, we need to determine what functional dependencies can be tested by checking each relation individually. Let F be a set of functional dependencies on a schema R , and let $R1, R2, \dots, Rn$ be a decomposition of R . The **restriction** of F to Ri is the set Fi of all functional dependencies in F^+ that include only attributes of Ri . Since all functional dependencies in a restriction involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.

We consider each member of the set F of functional dependencies that we require to hold on Lending-schema, and show that each one can be tested in at least one relation in the decomposition.

- We can test the functional dependency: $\text{branch-name} \rightarrow \text{branch-city assets}$ using **Branch-schema = (branch-name, branch-city, assets)**

- We can test the functional dependency: $\text{loan-number} \rightarrow \text{amount branch-name}$ using $\text{Loan-schema} = (\text{branch-name}, \text{loan-number}, \text{amount})$.

computeF+;

for each schema Ri in D **do**

begin

Fi := the restriction of F+ to Ri;

end

F_ := \emptyset

for each restriction Fi **do**

begin

F_ = F_ \cup Fi

end

computeF_+;

if(F_+ = F+) **then** return (true)

else return (false);

Repetition of Information

In the decomposition, the relation on schema Borrowerschema contains the loan-number, customer-name relationship, and no other schema does. Therefore, we have one tuple for each customer for a loan in only the relation on Borrower-schema. In the other relations involving loan-number (those on schemas Loan-schema and Borrower-schema), only one tuple per loan needs to appear.

12. Explain in detail about Boyce-Codd Normal Form? (11 Marks)

Definition

One of the more desirable normal forms that we can obtain is **Boyce-Codd normal form (BCNF)**. A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (that is, $\beta \subseteq \alpha$).
- α is a superkey for schema R.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF. Consider the following relation schemas and their respective functional dependencies:

- Customer-schema = (customer-name, customer-street, customer-city) customer-name \rightarrow customer-street customer-city
- Branch-schema = (branch-name, assets, branch-city) branch-name \rightarrow assets branch-city
- Loan-info-schema = (branch-name, customer-name, loan-number, amount) loan-number \rightarrow amount
branch-name

It is now possible to avoid redundancy in the case where there are several customers associated with a loan. There is exactly one tuple for each loan in the relation on Loan-schema, and one tuple for each customer of each loan in the relation on Borrower-schema. Thus, we do not have to repeat the branch name and the amount once for each customer associated with a loan. Often testing of a relation to see if it satisfies BCNF can be simplified:

- To check if a nontrivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF, compute α^+ (the attribute closure of α), and verify that it includes all attributes of R ; that is, it is a superkey of R .
- To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than check all dependencies in F

Decomposition Algorithm

The decomposition that the algorithm generates is not only in BCNF, but is also a lossless-join decomposition. To see why our algorithm generates only lossless-join decompositions, we note that, when we replace a schema R_i with $(R_i - \beta)$ and (α, β) , the dependency $\alpha \rightarrow \beta$ holds, and $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

result := { R };

done := false;

compute F^+ ;

while(not done) **do**

if(there is a schema R_i in result that is not in BCNF)

then begin

let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds

on R_i such that $\alpha \rightarrow R_i$ is not in F^+ , and $\alpha \cap \beta = \emptyset$;

result := (result - R_i) \cup ($R_i - \beta$) \cup (α, β);

end

else done := true;

Lending-schema = (branch-name, branch-city, assets, customer-name, loan-number, amount)

The set of functional dependencies that we require to hold on Lending-schema are

$\text{branch-name} \rightarrow \text{assets}$ branch-city

$\text{loan-number} \rightarrow \text{amount}$ branch-name

A candidate key for this schema is $\{\text{loan-number}, \text{customer-name}\}$.

We can apply the algorithm of Figure 7.13 to the Lending-schema example as follows:

- The functional dependency

$\text{branch-name} \rightarrow \text{assets}$ branch-city

holds on Lending-schema, but branch-name is not a superkey. Thus, Lending-schema is not in BCNF. We replace Lending-schema by

Branch-schema = $(\text{branch-name}, \text{branch-city}, \text{assets})$

Loan-info-schema = $(\text{branch-name}, \text{customer-name}, \text{loan-number}, \text{amount})$

Dependency Preservation

Banker-schema = $(\text{branch-name}, \text{customer-name}, \text{banker-name})$

which indicates that a customer has a “personal banker” in a particular branch. The set F of functional dependencies that we require to hold on the Banker-schema is

$\text{banker-name} \rightarrow \text{branch-name}$

$\text{branch-name customer-name} \rightarrow \text{banker-name}$

Clearly, Banker-schema is not in BCNF since banker-name is not a superkey. If we apply the algorithm we obtain the following BCNF decomposition:

Banker-branch-schema = $(\text{banker-name}, \text{branch-name})$

Customer-banker-schema = $(\text{customer-name}, \text{banker-name})$

The decomposed schemas preserve only $\text{banker-name} \rightarrow \text{branch-name}$ (and trivial dependencies), but the closure of $\{\text{banker-name} \rightarrow \text{branch-name}\}$ does not include $\text{customer-name branch-name} \rightarrow \text{banker-name}$. The violation of this dependency cannot be detected unless a join is computed.

Thus, the example shows that we cannot always satisfy all three design goals:

1. Lossless join
2. BCNF
3. Dependency preservation

13. Explain about third normal form? (11 Marks) APR 2014

We have two alternatives if we wish to check if an update violates any functional dependencies:

- Pay the extra cost of computing joins to test for violations.
- Use an alternative decomposition, third normal form (3NF), which we present below, which makes testing of updates cheaper. Unlike BCNF, 3NF decompositions may contain some redundancy in the decomposed schema.

Definition

BCNF requires that all nontrivial dependencies be of the form $\alpha \rightarrow \beta$, where α is a superkey. 3NF relaxes this constraint slightly by allowing nontrivial functional dependencies whose left side is not a superkey. A relation schema R is in **third normal form (3NF)** with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a superkey for R .
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R .

The first two alternatives are the same as the two alternatives in the definition of BCNF. The third alternative of the 3NF definition seems rather unintuitive, and it is not obvious why it is useful. It represents, in some sense, a minimal relaxation of the BCNF conditions that helps ensure that every schema has a dependency-preserving decomposition into 3NF. Its purpose will become more clear later, when we study decomposition into 3NF.

The only nontrivial functional dependencies of the form $\alpha \rightarrow \text{banker-name}$ include {customer-name, branch-name} as part of α . Since {customer-name, branch-name} is a candidate key, these dependencies do not violate the definition of 3NF.

Decomposition Algorithm

The set of dependencies F_c used in the algorithm is a canonical cover for F . Note that the algorithm considers the set of schemas R_j , $j = 1, 2, \dots, i$; initially $i = 0$, and in this case the set is empty.

let F_c be a canonical cover for F ;

$i := 0$;

```

for each functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  do
  if none of the schemas  $R_j, j = 1, 2, \dots, i$  contains  $\alpha\beta$ 
  then begin
     $i := i + 1$ ;
     $R_i := \alpha\beta$ ;
  end
  if none of the schemas  $R_j, j = 1, 2, \dots, i$  contains a candidate key for  $R$ 
  then begin
     $i := i + 1$ ;
     $R_i :=$  any candidate key for  $R$ ;
  end
return ( $R_1, R_2, \dots, R_i$ )

```

Banker-info-schema = (branch-name, customer-name, banker-name, office-number)

The main difference here is that we include the banker's office number as part of the information. The functional dependencies for this relation schema are

banker-name \rightarrow branch-name office-number

customer-name branch-name \rightarrow banker-name

The **for** loop in the algorithm causes us to include the following schemas in our decomposition:

Banker-office-schema = (banker-name, branch-name, office-number)

Banker-schema = (customer-name, branch-name, banker-name)

Let us consider the three possible cases:

- B is in both α and β . In this case, the dependency $\alpha \rightarrow \beta$ would not have been in F_c since B would be extraneous in β . Thus, this case cannot hold.
- B is in β but not α . Consider two cases:
 - γ is a superkey. The second condition of 3NF is satisfied.
 - γ is not a superkey. Then α must contain some attribute not in γ . Now, since $\gamma \rightarrow B$ is in F^+ , it must be derivable from F_c by using the attribute closure algorithm on γ . The derivation could not have used $\alpha \rightarrow \beta$ — if it had been used, α must be contained in the attribute closure of γ , which is not possible, since we assumed γ is not a superkey. Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$ (since $\gamma \subseteq \alpha\beta$, and γ cannot contain B because $\gamma \rightarrow B$ is nontrivial). This would imply that B is extraneous in the right hand side of $\alpha \rightarrow \beta$, which is not possible since $\alpha \rightarrow \beta$ is in the canonical cover F_c . Thus, if B is in β , then γ must be a superkey, and the second condition of 3NF must be satisfied.

- B is in α but not β . Since α is a candidate key, the third alternative in the definition of 3NF is satisfied.

14. Explain about fourth normal form? (11 Marks) APR 2014

BC-schema = (loan-number, customer-name, customer-street, customer-city)

The astute reader will recognize this schema as a non-BCNF schema because of the functional dependency

customer-name \rightarrow customer-street customer-city

Multivalued Dependencies

Functional dependencies rule out certain tuples from being in a relation. If $A \rightarrow B$, then we cannot have two tuples with the same A value but different B values. Multivalued dependencies, on the other hand, do not rule out the existence of certain tuples. Instead, they require that other tuples of a certain form be present in the relation. For this reason, functional dependencies sometimes are referred to as **equalitygenerating dependencies**, and multivalued dependencies are referred to as **tuplegenerating dependencies**.

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$\alpha \twoheadrightarrow \beta$

holds on R if, in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that

$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$

$t_3[\beta] = t_1[\beta]$

$t_3[R - \beta] = t_2[R - \beta]$

$t_4[\beta] = t_2[\beta]$

$t_4[R - \beta] = t_1[R - \beta]$

As with functional dependencies, we shall use multivalued dependencies in two ways:

1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies
2. To specify constraints on the set of legal relations; we shall thus concern ourselves with only those relations that satisfy a given set of functional and multivalued dependencies.

From the definition of multivalued dependency, we can derive the following rule:

- If $\alpha \rightarrow \beta$, then $\alpha \rightarrow\rightarrow \beta$.

In other words, every functional dependency is also a multivalued dependency.

Definition of Fourth Normal Form

Consider again our BC-schema example in which the multivalued dependency $\text{customer-name} \rightarrow\rightarrow \text{customer-street customer-city}$ holds, but no nontrivial functional dependencies hold. We saw in the opening paragraphs of Section 7.8 that, although BC-schema is in BCNF, the design is not ideal, since we must repeat a customer's address information for each loan. We shall see that we can use the given multivalued dependency to improve the database design, by decomposing BC-schema into a **fourth normal form** decomposition.

A relation schema R is in **fourth normal form** (4NF) with respect to a set D of functional and multivalued dependencies if, for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds

- $\alpha \rightarrow\rightarrow \beta$ is a trivial multivalued dependency.
- α is a superkey for schema R .

A database design is in 4NF if each member of the set of relation schemas that constitutes the design is in 4NF.

result := {R};

done := false;

compute D^+ ; Given schema R_i , let D_i denote the restriction of D^+ to R_i

while (not done) **do**

if (there is a schema R_i in result that is not in 4NF w.r.t. D_i)

then begin

let $\alpha \rightarrow\rightarrow \beta$ be a nontrivial multivalued dependency that holds

on R_i such that $\alpha \rightarrow R_i$ is not in D_i , and $\alpha \cap \beta = \emptyset$;

result := (result - R_i) \cup ($R_i - \beta$) \cup (α, β);

end

else done := true;

Let R be a relation schema, and let R_1, R_2, \dots, R_n be a decomposition of R . To check if each relation schema R_i in the decomposition is in 4NF, we need to find what multivalued dependencies hold on each R_i . Recall that, for a set F of functional dependencies, the restriction F_i of F to R_i is all functional

dependencies in F^+ that include only attributes of R_i . Now consider a set D of both functional and multivalued dependencies. The **restriction** of D to R_i is the set D_i consisting of

1. All functional dependencies in D^+ that include only attributes of R_i

2. All multivalued dependencies of the form

$$\alpha \twoheadrightarrow \beta \cap R_i$$

where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+ .

Decomposition Algorithm

The analogy between 4NF and BCNF applies to the algorithm for decomposing a schema. It is identical to the BCNF decomposition algorithm of Figure 7.13, except that it uses multivalued, instead of functional, dependencies and uses the restriction of D^+ to R_i . If we apply the algorithm BC-schema, we find that customer-name \twoheadrightarrow loan-number is a nontrivial multivalued dependency, and customer-name is not a superkey for BC-schema. Following the algorithm, we replace BC-schema by two schemas:

Borrower-schema = (customer-name, loan-number)

Customer-schema = (customer-name, customer-street, customer-city).

This pair of schemas, which is in 4NF, eliminates the problem we encountered earlier with the redundancy of BC-schema.

Let R be a relation schema, and let D be a set of functional and multivalued dependencies on R . Let R_1 and R_2 form a decomposition of R . This decomposition is a lossless-join decomposition of R if and only if at least one of the following multivalued dependencies is in D^+ :

$$R_1 \cap R_2 \twoheadrightarrow R_1$$

$$R_1 \cap R_2 \twoheadrightarrow R_2$$

More Normal Forms Or Fifth normal forms

The fourth normal form is by no means the “ultimate” normal form. As we saw earlier, multivalued dependencies help us understand and tackle some forms of repetition of information that cannot be understood in terms of functional dependencies. There are types of constraints called **join dependencies** that generalize multivalued dependencies, and lead to another normal form called **project-join normal form (PJNF)** (PJNF is called **fifth normal form** in some books). There is a class of even more general constraints, which leads to a normal form called **domain-key normal form**.

Boyce-Codd Normal Form

1. A relation schema R is in **Boyce-Codd Normal Form (BCNF)** with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (i.e. $\beta \subseteq \alpha$).
- α is a superkey for schema R .

2. A database design is in BCNF if each member of the set of relation schemas is in BCNF.

3. Let's assess our example banking design:

Customer-schema = (*cname*, *street*, *ccity*)

$cname \rightarrow street\ ccity$

Branch-schema = (*bname*, *assets*, *bcity*)

$bname \rightarrow assets\ bcity$

Loan-info-schema = (*bname*, *cname*, *loan#*, *amount*)

$loan\# \rightarrow amount\ bname$

Customer-schema and *Branch-schema* are in BCNF.

4. Let's look at *Loan-info-schema*:

- We have the non-trivial functional dependency $loan\# \rightarrow amount$, and
- $loan\#$ is not a superkey.
- Thus *Loan-info-schema* is not in BCNF.
- We also have the repetition of information problem.
- For each customer associated with a loan, we must repeat the branch name and amount of the loan.
- We can eliminate this redundancy by decomposing into schemas that are all in BCNF.

5. If we decompose into

Loan-schema = (*bname*, *loan#*, *amount*)

Borrow-schema = (*cname*, *loan#*)

we have a lossless-join decomposition. (Remember why?)

To see whether these schemas are in BCNF, we need to know what functional dependencies apply to them.

- For *Loan-schema*, we have $loan\# \rightarrow amount\ bname$ applying.
- Only trivial functional dependencies apply to *Borrow-schema*.
- Thus both schemas are in BCNF.

We also no longer have the repetition of information problem. Branch name and loan amount information are not repeated for each customer in this design.

6. Now we can give a general method to generate a collection of BCNF schemas.

```

result := {R};
done := false;
compute F+;
while (not done) do
    if (there is a schema Ri in result that is not in BCNF)
        then begin
            let α → β be a nontrivial
functional dependency that holds on Ri
such that α → Ri is not in F+,
and α ∩ β = ∅;
            result = (result - Ri) ∪ (Ri - β) ∪ (α, β);
        end
    else done = true;

```

7. This algorithm generates a lossless-join BCNF decomposition. Why?

- We replace a schema R_i with $(R_i - \beta)$ and (α, β) .
- The dependency $\alpha \rightarrow \beta$ holds on R_i .
- $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.
- So we have $R_1 \cap R_2 \rightarrow R_2$, and thus a lossless join.

8. Let's apply this algorithm to our earlier example of poor database design:

Lending-schema = (*bname*, *assets*, *bcity*, *loan#*, *cname*, *amount*)

The set of functional dependencies we require to hold on this schema are

$bname \rightarrow assets\ bcity$

$loan\# \rightarrow amount\ bname$

A candidate key for this schema is {*loan#*, *cname*}.

We will now proceed to decompose:

- The functional dependency $bname \rightarrow assets\ bcity$
- holds on *Lending-schema*, but *bname* is not a superkey.

We replace *Lending-schema* with

Branch-schema = (*bname*, *assets*, *bcity*)

Loan-info-schema = (*bname*, *loan#*, *cname*, *amount*)

- *Branch-schema* is now in BCNF.
- The functional dependency *loan#* \rightarrow *amount* holds on *Loan-info-schema*, but *loan#* is not a superkey.

We replace *Loan-info-schema* with

Loan-schema = (*bname*, *loan#*, *amount*)

Borrow-schema = (*cname*, *loan#*)

- These are both now in BCNF.
- We saw earlier that this decomposition is both lossless-join and dependency-preserving.

9. Not every decomposition is dependency-preserving.

- Consider the relation schema
- *Banker-schema* = (*bname*, *cname*, *banker-name*)
- The set *F* of functional dependencies is
- *banker-name* \rightarrow *bname*
-
- *cname bname* \rightarrow *banker-name*
- The schema is not in BCNF as *banker-name* is not a superkey.
- If we apply our algorithm, we may obtain the decomposition
- *Banker-branch-schema* = (*bname*, *banker-name*)
- *Cust-banker-schema* = (*cname*, *banker-name*)
- The decomposed schemas preserve only the first (and trivial) functional dependencies.
- The closure of this dependency does not include the second one.
- Thus a violation of *cname bname* \rightarrow *banker-name* cannot be detected unless a join is computed.

This shows us that not every BCNF decomposition is dependency-preserving.

10. It is not always possible to satisfy all three design goals:

- BCNF.
- Lossless join.
- Dependency preservation.

11. We can see that any BCNF decomposition of *Banker-schema* must fail to preserve

12. *cname bname* \rightarrow *banker-name*

13. Some Things To Note About BCNF

- There is sometimes more than one BCNF decomposition of a given schema.
- The algorithm given produces only one of these possible decompositions.
- Some of the BCNF decompositions may also yield dependency preservation, while others may not.
- Changing the order in which the functional dependencies are considered by the algorithm may change the decomposition.
- For example, try running the BCNF algorithm on

$$R = (A, B, C, D)$$

•

$$A \rightarrow B, C$$

$$B \rightarrow D$$

$$D \rightarrow B$$

Then change the order of the last two functional dependencies and run the algorithm again. Check the two decompositions for dependency preservation.

Third Normal Form

1. When we cannot meet all three design criteria, we abandon BCNF and accept a weaker form called **third normal form (3NF)**.
2. It is always possible to find a dependency-preserving lossless-join decomposition that is in 3NF.
3. A relation schema R is in **3NF** with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
 - a. $\alpha \rightarrow \beta$ is a trivial functional dependency.
 - b. α is a superkey for schema R .
 - c. Each attribute A in $\beta - \alpha$ is contained in a candidate key for R .
4. A database design is in 3NF if each member of the set of relation schemas is in 3NF.
5. We now allow functional dependencies satisfying only the third condition. These dependencies are called **transitive dependencies**, and are not allowed in BCNF.
6. As all relation schemas in BCNF satisfy the first two conditions only, a schema in BCNF is also in 3NF.
7. BCNF is a more restrictive constraint than 3NF.
8. Our *Banker-schema* decomposition did not have a dependency-preserving lossless-join decomposition into BCNF. The schema was already in 3NF though (check it out).
9. We now present an algorithm for finding a dependency-preserving lossless-join decomposition into 3NF.

10. Note that we require the set F of functional dependencies to be in **canonical form**.

let F_c be a **canonical cover** for F ;

$i := 0$;

for each functional dependency $\alpha \rightarrow \beta \in F_c$ **do**

if none of the schemas $R_j, 1 \leq j \leq i$ contains $\alpha\beta$

then begin

$i := i + 1$;

$R_i := \alpha\beta$

end

if none of the schemas $R_j, 1 \leq j \leq i$

contains a candidate key for R

then begin

$i := i + 1$;

$R_i :=$ any candidate key for R

end

return (R_1, R_2, \dots, R_i)

Each relation schema is in 3NF. Why? (A proof is given in [Ullman 1988].)

Comparison of BCNF and 3NF

1. We have seen BCNF and 3NF.

- It is always possible to obtain a 3NF design without sacrificing lossless-join or dependency-preservation.
- If we do not eliminate all transitive dependencies, we may need to use null values to represent some of the meaningful relationships.
- Repetition of information occurs.

2. These problems can be illustrated with *Banker-schema*.

- As $banker-name \twoheadrightarrow bname$, we may want to express relationships between a banker and his or her branch.

cname	banker-name	bname
Bill	John	SFU
Tom	John	SFU
Mary	John	SFU
null	Tim	Austin

Figure 7.4: An instance of *Banker-schema*.

- Figure 7.4 shows how we must either have a corresponding value for customer name, or include a null.
 - Repetition of information also occurs.
 - Every occurrence of the banker's name must be accompanied by the branch name.
3. If we must choose between BCNF and dependency preservation, it is generally better to opt for 3NF.
 - If we cannot check for dependency preservation efficiently, we either pay a high price in system performance or risk the integrity of the data.
 - The limited amount of redundancy in 3NF is then a lesser evil.
 4. To summarize, our goal for a relational database design is
 - BCNF.
 - Lossless-join.
 - Dependency-preservation.
 5. If we cannot achieve this, we accept
 - 3NF
 - Lossless-join.
 - Dependency-preservation.
 6. **A final point:** there is a price to pay for decomposition. When we decompose a relation, we have to use natural joins or Cartesian products to put the pieces back together. This takes computational time.

Fourth Normal Form (4NF)

1. We saw that *BC-schema* was in BCNF, but still was not an ideal design as it suffered from repetition of information. We had the multivalued dependency $cname \twoheadrightarrow street\ ccity$, but no non-trivial functional dependencies.
2. We can use the given multivalued dependencies to improve the database design by decomposing it into **fourth normal form**.
3. A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
 - $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency.
 - α is a superkey for schema R .
4. A database design is in 4NF if each member of the set of relation schemas is in 4NF.

5. The definition of 4NF differs from the BCNF definition only in the use of multivalued dependencies.
 - Every 4NF schema is also in BCNF.
 - To see why, note that if a schema is not in BCNF, there is a non-trivial functional dependency $\alpha \rightarrow \beta$ holding on R , where α is not a superkey.
 - Since $\alpha \rightarrow \beta$ implies $\alpha \twoheadrightarrow \beta$, by the replication rule, R cannot be in 4NF.
6. We have an algorithm similar to the BCNF algorithm for decomposing a schema into 4NF:

```

result := {R};
done := false;
compute  $D^+$ ;
while (not done) do
  if (there is a schema  $R_i$  in result
    that is not in 4NF)
    then begin
      let  $\alpha \twoheadrightarrow \beta$  be a nontrivial multivalued
        dependency that holds on  $R_i$  such that  $\alpha \rightarrow R_i$  is not in  $D^+$ , and
         $\alpha \cap \beta = \emptyset$ ;
      result = (result -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );
    end
  else done = true;

```

7. If we apply this algorithm to *BC-schema*:
 - $cname \twoheadrightarrow loan\#$ is a nontrivial multivalued dependency and $cname$ is not a superkey for the schema.
 - We then replace *BC-schema* by two schemas:
 - $Cust-loan-schema = (cname, loan\#)$
 - $Customer-schema = (cname, street, ccity)$
 - These two schemas are in 4NF.
8. We can show that our algorithm generates only lossless-join decompositions.
 - Let R be a relation schema and D a set of functional and multivalued dependencies on R .
 - Let R_1 and R_2 form a decomposition of R .

- This decomposition is lossless-join if and only if at least one of the following multivalued dependencies is in D^+ :
 - $R_1 \cap R_2 \twoheadrightarrow R_1$
 - $R_1 \cap R_2 \twoheadrightarrow R_2$
 - We saw similar criteria for functional dependencies.
 - This says that for **every** lossless-join decomposition of R into two schemas and , one of the two above dependencies must hold.
 - You can see, by inspecting the algorithm, that this must be the case for every decomposition.
9. Dependency preservation is not as simple to determine as with functional dependencies.
- Let R be a relation schema.
 - Let R_1, R_2, \dots, R_n be a decomposition of R .
 - Let D be the set of functional and multivalued dependencies holding on R .
 - The **restriction** of D to R_i is the set D_i consisting of:
 - All functional dependencies in D^+ that include only attributes of R_i .
 - All multivalued dependencies of the form $\alpha \twoheadrightarrow \beta \cap R_i$ where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+ .
 - A decomposition of schema R is dependency preserving with respect to a set D of functional and multivalued dependencies if for every set of relations $r_1(R_1), r_2(R_2), \dots, r_n(R_n)$ such that for all i , r_i satisfies D_i , there exists a relation $r(R)$ that satisfies D and for which $r_i = \Pi_{R_i}(r)$ for all i .
10. What does this formal statement say? It says that a decomposition is dependency preserving if for every set of relations on the decomposition schema satisfying only the restrictions on D there exists a relation r on the entire schema R that the decomposed schemas can be derived from, and that r also satisfies the functional and multivalued dependencies.
11. We'll do an example using our decomposition algorithm and check the result for dependency preservation.
- Let $R = (A, B, C, G, H, I)$.
 - Let D be
 - $A \twoheadrightarrow B$
 - $B \twoheadrightarrow HI$
 - $CG \twoheadrightarrow H$
 - R is not in 4NF, as we have $A \twoheadrightarrow B$ and A is not a superkey.
 - The algorithm causes us to decompose using this dependency into

- $R_1 = (A, B)$
- $R_2 = (A, C, G, H, I)$
- R_1 is now in 4NF, but R_2 is not.
- Applying the multivalued dependency $CG \twoheadrightarrow H$ (how did we get this?), our algorithm then decomposes R_2 into
- $R_3 = (C, G, H)$
- $R_4 = (A, C, G, I)$
- R_3 is now in 4NF, but R_4 is not.
- Why? As $A \twoheadrightarrow HI$ is in D^+ (why?) then the restriction of this dependency to R_4 gives us $A \twoheadrightarrow I$.
- Applying this dependency in our algorithm finally decomposes R_4 into
- $R_5 = (A, I)$
- $R_6 = (A, C, G)$
- The algorithm terminates, and our decomposition is R_1, R_3, R_5 and R_6 .

12. Let's analyze the result.

$r_1 :$	A	B	$r_2 :$	C	G	H	$r_3 :$	A	I	$r_4 :$	A	C	G
	a_1	b_1		c_1	g_1	h_1		a_1	i_1		a_1	c_1	g_1
	a_2	b_1		c_2	g_2	h_2		a_1	i_2		a_2	c_2	g_2

Figure 7.9: Projection of relation r onto a 4NF decomposition of R .

- This decomposition is not dependency preserving as it fails to preserve $B \twoheadrightarrow HI$.
- Figure 7.9 (textbook 6.14) shows four relations that may result from projecting a relation onto the four schemas of our decomposition.
- The restriction of D to (A, B) is $A \twoheadrightarrow B$ and some trivial dependencies.
- We can see that r_1 satisfies $A \twoheadrightarrow B$ as there are no pairs with the same A value.
- Also, r_2 satisfies all functional and multivalued dependencies since no two tuples have the same value on any attribute.
- We can say the same for r_3 and r_4 .
- So our decomposed version satisfies all the dependencies in the restriction of D .
- However, there is no relation r on (A, B, C, G, H, I) that satisfies D and decomposes into r_1, r_2, r_3 and r_4 .
- Figure 7.10 (textbook 6.15) shows $r = r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$.
- Relation r does not satisfy $B \twoheadrightarrow HI$.

- Any relation s containing r and satisfying $B \twoheadrightarrow HI$ must include the tuple $(a_2, b_1, c_2, g_2, h_1, i_1)$.
- However, $\Pi_{CGHI}(s)$ includes a tuple (c_2, g_2, h_1) that is not in r_2 .
- Thus our decomposition fails to detect a violation of $B \twoheadrightarrow HI$.

A	B	C	G	H	I
a_1	b_1	c_1	g_1	h_1	i_1
a_2	b_1	c_2	g_2	h_2	i_2

Figure 7.10: A relation $r(R)$ that does not satisfy $B \twoheadrightarrow HI$.

- We have seen that if we are given a set of functional and multivalued dependencies, it is best to find a database design that meets the three criteria:
 - 4NF.
 - Dependency Preservation.
 - Lossless-join.
- If we only have functional dependencies, the first criteria is just BCNF.
- We cannot always meet all three criteria. When this occurs, we compromise on 4NF, and accept BCNF, or even 3NF if necessary, to ensure dependency preservation.

7. Explain referential integrity in detail (Nov 2010)

Referential integrity

An example of a database that has not enforced **referential integrity**. In this example, there is a foreign key (**artist_id**) value in the album table that references a non-existent artist — in other words there is a foreign key value with no corresponding primary key value in the referenced table. What happened here was that there was an artist called "Aerosmith", with an **artist_id** of "4", which was deleted from the artist table. However, the album "Eat the Rich" referred to this artist. With referential integrity enforced, this would not have been possible.

Referential integrity in a relational database is consistency between coupled tables. Referential integrity is usually enforced by the combination of a primary key or candidate key (alternate key) and a foreign key. For referential integrity to hold, any field in a table that is declared a foreign key can contain only values from a parent table's primary key or a candidate key. For instance, deleting a record that contains a value referred to by a foreign key in another table would break referential integrity. The relational database management system (RDBMS) enforces referential integrity, normally either by deleting the foreign key rows as well to maintain integrity, or by returning an error and not performing the delete. Which method is used would be defined by the definition of the referential integrity constraint.

Example

An employee database stores the department in which each employee works. The field "Department Number" in the Employee table is declared a foreign key, and it refers to the field "Index" in the Department table which is declared a primary key. Referential integrity would be broken by deleting a department from the Department table if employees listed in the Employee table are listed as working for that department, unless those employees are moved to a different department at the same time.

Database trigger

A **database trigger** is procedural code that is automatically executed in response to certain events on a particular table in a database. Triggers can restrict access to specific data, perform logging, or audit data modifications.

There are two classes of triggers, they are either "row triggers" or "statement triggers". With row triggers you can define an action for every row of a table, while statement triggers occur only once per INSERT, UPDATE, or DELETE statement. Triggers cannot be used to audit data retrieval via SELECT statements.

Each class can be of several types. There are "BEFORE triggers" and "AFTER triggers" which identifies the time of execution of the trigger. There is also an "INSTEAD OF trigger" which is a trigger that will execute instead of the triggering statement.

There are typically three triggering events that cause triggers to 'fire':

- INSERT event (as a new record is being inserted into the database).
- UPDATE event (as a record is being changed).
- DELETE event (as a record is being deleted).

The trigger is used to automate DML condition process.

The major features and effects of database triggers are that they:

- do not accept parameters or arguments (but may store affected-data in temporary tables)
- cannot perform commit or rollback operations because they are part of the triggering SQL statement (only through autonomous transactions)
- can cause mutating table errors, if they are poorly written.

Triggers in Oracle

In addition to triggers that fire when data is modified, Oracle 9i supports triggers that fire when schema objects (that is, tables) are modified and when user logon or logoff events occur. These trigger types are referred to as "Schema-level triggers".

Schema-level triggers

- After Creation
- Before Alter

- After Alter
- Before Drop
- After Drop
- Before Logoff
- After Logon

Triggers in Microsoft SQL Server

Microsoft SQL Server supports triggers either after or instead of an insert, update, or delete operation.

'Microsoft SQL Server supports triggers on tables and views with the constraint that a view can be referenced only by an INSTEAD OF trigger.

Microsoft SQL Server 2005 introduced support for Data Definition Language (DDL) triggers, which can fire in reaction to a very wide range of events, including:

- Drop table
- Create table
- Alter table
- Login events

Performing conditional actions in triggers (or testing data following modification) is done through accessing the temporary *Inserted* and *Deleted* tables.

Triggers in PostgreSQL

PostgreSQL introduced support for triggers in 1997. The following functionality in SQL:2003 is not implemented in PostgreSQL:

- SQL allows triggers to fire on updates to specific columns; PostgreSQL does not support this feature.
- The standard allows the execution of a number of other SQL statements than SELECT, INSERT, UPDATE, such as CREATE TABLE as the triggered action.

Triggers in MySQL

MySQL 5.0 introduced support for triggers. Some of the triggers MySQL supports are

- INSERT Trigger
- UPDATE Trigger
- DELETE Trigger

The SQL:2003 standard mandates that triggers give programmers access to record variables by means of a syntax such as *REFERENCING NEW AS n*. For example, if a trigger is monitoring for changes to a salary column one could write a trigger like the following:

```
CREATE TRIGGER salary_trigger
BEFORE UPDATE ON employee_table
```

```
REFERENCING NEW ROW AS n, OLD ROW AS o
FOR EACH ROW
IF n.salary <> o.salary THEN
    --make desired changes;
END IF;
```

8. Explain any one Database system of your own with normalization (Nov 2010)

Normalization is a method for organizing data elements in a database into tables.

Normalization Avoids

- Duplication of Data – The same data is listed in multiple lines of the database
- Insert Anomaly – A record about an entity cannot be inserted into the table without first inserting information about another entity – Cannot enter a customer without a sales order
- Delete Anomaly – A record cannot be deleted without deleting a record about a related entity. Cannot delete a sales order without deleting all of the customer's information.
- Update Anomaly – Cannot update information without changing information in many places. To update customer information, it must be updated for each sales order the customer has placed

Normalization is a three stage process – After the first stage, the data is said to be in first normal form, after the second, it is in second normal form, after the third, it is in third normal form

Before Normalization

1. Begin with a list of all of the fields that must appear in the database. Think of this as one big table.
2. Do not include computed fields
3. One place to begin getting this information is from a printed document used by the system.
4. Additional attributes besides those for the entities described on the document can be added to the database.

Before Normalization – Example

See Sales Order from below:

Sales Order

Fiction Company
202 N. Main
Mahattan, KS 66502

CustomerNumber: 1001
Customer Name: ABC Company
Customer Address: 100 Points
Manhattan, KS 66502

Sales Order Number: 405
Sales Order Date: 2/1/2000
Clerk Number: 210
Clerk Name: Martin Lawrence

Item Ordered	Description	Quantity	Unit Price	Total
800	widgit small	40	60.00	2,400.00
801	tingimajigger	20	20.00	400.00
805	thingibob	10	100.00	1,000.00
Order Total				3,800.00

Fields in the original data table will be as follows:

SalesOrderNo, Date, CustomerNo, CustomerName, CustomerAdd, ClerkNo, ClerkName, ItemNo,
Description, Qty, UnitPrice

Think of this as the baseline – one large table

Normalization: First Normal Form

- Separate Repeating Groups into New Tables.
- **Repeating Groups** Fields that may be repeated several times for one document/entity
- Create a new table containing the repeating data
- The primary key of the new table (repeating group) is always a composite key; Usually document number and a field uniquely describing the repeating line, like an item number.

First Normal Form Example

The new table is as follows:

SalesOrderNo, ItemNo, Description, Qty, UnitPrice

The repeating fields will be removed from the original data table, leaving the following.

SalesOrderNo, Date, CustomerNo, CustomerName, CustomerAdd, ClerkNo, ClerkName

These two tables are a database in first normal form

What if we did not Normalize the Database to First Normal Form?

Repetition of Data – SO Header data repeated for every line in sales order.

Normalization: Second Normal Form

- Remove Partial Dependencies.
- **Functional Dependency** The value of one attribute in a table is determined entirely by the value of another.
- **Partial Dependency** A type of functional dependency where an attribute is functionally dependent on only part of the primary key (primary key must be a composite key).
- Create separate table with the functionally dependent data and the part of the key on which it depends. Tables created at this step will usually contain descriptions of resources.

Second Normal Form Example

The new table will contain the following fields:

ItemNo, Description

All of these fields except the primary key will be removed from the original table. The primary key will be left in the original table to allow linking of data:

SalesOrderNo, ItemNo, Qty, UnitPrice

Never treat price as dependent on item. Price may be different for different sales orders (discounts, special customers, etc.)

Along with the unchanged table below, these tables make up a database in second normal form:

SalesOrderNo, Date, CustomerNo, CustomerName, CustomerAdd, ClerkNo, ClerkName

What if we did not Normalize the Database to Second Normal Form?

- Repetition of Data – Description would appear every time we had an order for the item
- Delete Anomalies – All information about inventory items is stored in the SalesOrderDetail table. Delete a sales order, delete the item.
- Insert Anomalies – To insert an inventory item, must insert sales order.
- Update Anomalies – To change the description, must change it on every SO.

Normalization: Third Normal Form

- Remove transitive dependencies.
- **Transitive Dependency** A type of functional dependency where an attribute is functionally dependent on an attribute other than the primary key. Thus its value is only indirectly determined by the primary key.

- Create a separate table containing the attribute and the fields that are functionally dependent on it. Tables created at this step will usually contain descriptions of either resources or agents. Keep a copy of the key attribute in the original file.

Third Normal Form Example

The new tables would be:

CustomerNo, CustomerName, CustomerAdd

ClerkNo, ClerkName

All of these fields except the primary key will be removed from the original table. The primary key will be left in the original table to allow linking of data as follows:

SalesOrderNo, Date, CustomerNo, ClerkNo

Together with the unchanged tables below, these tables make up the database in third normal form.

ItemNo, Description

SalesOrderNo, ItemNo, Qty, UnitPrice

What if we did not Normalize the Database to Third Normal Form?

- Repetition of Data – Detail for Cust/Clerk would appear on every SO
- Delete Anomalies – Delete a sales order, delete the customer/clerk
- Insert Anomalies – To insert a customer/clerk, must insert sales order.
- Update Anomalies – To change the name/address, etc, must change it on every SO.

Completed Tables in Third Normal Form

Customers: CustomerNo, CustomerName, CustomerAdd

Clerks: ClerkNo, ClerkName

Inventory Items: ItemNo, Description

Sales Orders: SalesOrderNo, Date, CustomerNo, ClerkNo

SalesOrderDetail: SalesOrderNo, ItemNo, Qty, UnitPrice

9. Briefly explain RAID (NOV 2012)(APRIL 2014)

RAID

Redundant Array of Independent Drives is also known as Redundant Array of Inexpensive Drives (or Disks), and in either respect it is referred to as RAID. RAID is an umbrella term for computer data storage schemes that divide and replicate data among multiple hard disk drives. RAID's various designs balance or accentuate two key design goals: increased data reliability and increased I/O (input/output) performance.

A number of standard schemes have evolved which are referred to as levels. There were five RAID levels originally conceived, but many more variations have evolved, notably several nested levels and many non-standard levels (mostly proprietary).

RAID combines physical hard disks into a single logical unit by using either special hardware or software. Hardware solutions often are designed to present themselves to the attached system as a single hard drive, and the operating system is unaware of the technical workings. Software solutions are typically implemented in the operating system, and again would present the RAID drive as a single drive to applications.

There are three key concepts in RAID: mirroring, the copying of data to more than one disk; striping, the splitting of data across more than one disk; and error correction, where redundant data is stored to allow problems to be detected and possibly fixed (known as fault tolerance). Different RAID levels use one or more of these techniques, depending on the system requirements. The main aims of using RAID are to improve reliability, important for protecting information that is critical to a business, for example a database of customer orders; or where speed is important, for example a system that delivers video on demand TV programs to many viewers.

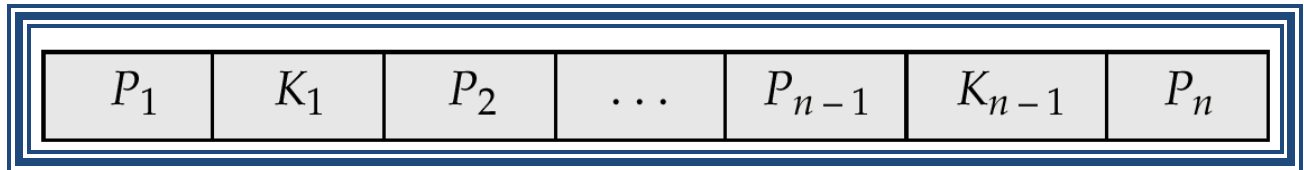
The configuration affects reliability and performance in different ways. The problem with using more disks is that it is more likely that one will go wrong, but by using error checking the total system can be made more reliable by being able to survive and repair the failure. Basic mirroring can speed up reading data as a system can read different data from both the disks, but it may be slow for writing if the configuration requires that both disks must confirm that the data is correctly written. Striping is often used for performance, where it allows sequences of data to be read off multiple disks at the same time. Error checking typically will slow the system down as data needs to be read from several places and compared. The design of RAID systems is therefore a compromise and understanding the requirements of a system is important. Modern disk arrays typically provide the facility to select the appropriate RAID configuration.

RAID systems can be designed to keep working when there is failure - disks can be hot swapped and data recovered automatically while the system keeps running. Other systems have to be shut down while the data is recovered. RAID is often used in high availability systems, where it is important that the system keeps running as much of the time as possible.

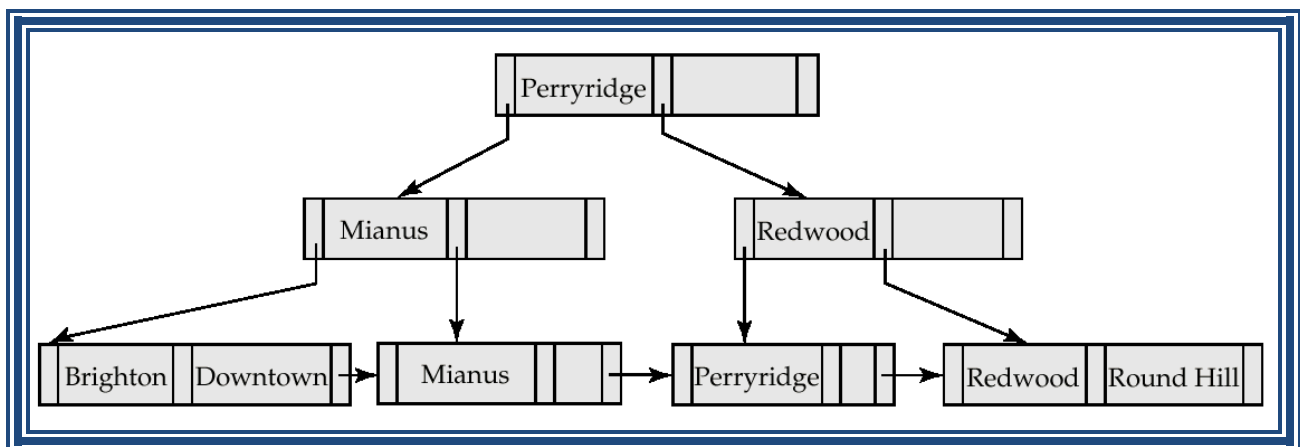
10. Explain about B+Tree index files (NOV 2012)

The Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:

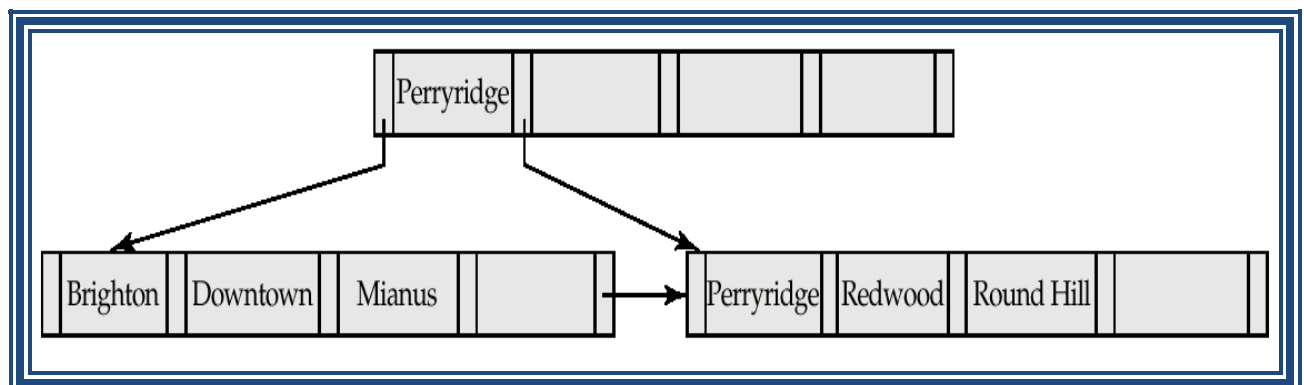
- All the search-keys in the sub tree to which P_1 points are less than K_1
- For $2 \leq i \leq n - 1$, all the search-keys in the sub tree to which P_i points have values greater than or equal to K_{i-1} and less than K_{m-1}



- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil$ and n with $n = 5$).
- Root must have at least 2 children.



B⁺-tree for *account* file ($n = 3$)



B⁺-tree for *account* file ($n = 5$)

A B⁺-tree is one in a family of multi-way search trees (others are: B-tree, 2-3-4 Tree, B*-Trees). These trees were first proposed by Bayer and McCreight in 1972 and in just a few years had replaced almost all large file access methods other than hashing. These multi-way balanced search trees are now the standard file organization for applications requiring insertion, deletion, and key range searches. They have the following advantages:

- B-trees are always height balanced, with all leaf nodes at the same level
- Update and search operations affect only a few disk pages, so performance is good.
- B-trees keep related records on the same disk page, which takes advantage of locality of reference.
- B-trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation over many thousands of records.

B⁺-tree Structure

A B⁺-tree in certain aspects is a generalization of a binary search tree (BST). The main difference is that nodes of a B⁺-tree will point to many children nodes rather than being limited to only two. Since our goal is to minimize disk accesses whenever we are trying to locate records, we want to make the height of the multi-way search tree as small as possible. This goal is achieved by having the tree branch in large amounts at each node.

A B⁺-tree of order m is a tree where each internal node contains up to m branches (children nodes) and thus store up to $m-1$ search key values -- in a BST, only one key value is needed since there are just two children nodes that an internal node can have. m is also known as the branching factor or the fanout of the tree.

1. The B⁺-tree stores records (or pointers to actual records) only at the leaf nodes, which are all found at the same level in the tree, so the tree is always height balanced.
2. All internal nodes, except the root, have between $\text{Ceiling}(m/2)$ and m children
3. The root is either a leaf or has at least two children.
4. Internal nodes store search key values, and are used only as placeholders to guide the search.

The number of search key values in each internal node is one less than the number of its non-empty children, and these keys partition the keys in the children in the fashion of a search tree. The keys are stored in non-decreasing order (i.e. sorted in lexicographical order).

5. Depending on the size of a record as compared to the size of a key, a leaf node in a B⁺-tree of order m may store more or less than m records. Typically this is based on the size of a disk block, the size of a record pointer, etc. The leaf pages must store enough records to remain at least half full.
6. The leaf nodes of a B⁺-tree are linked together to form a linked list. This is done so that the records can be retrieved sequentially without accessing the B⁺-tree index. This also supports fast processing of range-search queries as will be described later.

B⁺-tree operations

To understand the B⁺-tree operations more clearly, assume, without loss of generality, that there is a table whose primary is a single attribute and that it has a B⁺-tree index organized on the PK attribute of the table.

Searching for records that satisfy a simple condition

To retrieve records, queries are written with conditions that describe the values that the desired records are to have. The most basic search on a table to retrieve a single record given its PK value K.

Search in a B⁺-tree is an alternating two-step process, beginning with the root node of the B⁺-tree. Say that the search is for the record with key value K -- there can only be one record because we assume that the index is built on the PK attribute of the table.

1. Perform a binary search on the search key values in the current node -- recall that the search key values in a node are sorted and that the search starts with the root of the tree. We want to find the key K_i such that $K_i \leq K < K_{i+1}$.
2. If the current node is an internal node, follow the proper branch associated with the key K_i by loading the disk page corresponding to the node and repeat the search process at that node.
3. If the current node is a leaf, then:
 - a. If $K=K_i$, then the record exists in the table and we can return the record associated with K_i
 - b. Otherwise, K is not found among the search key values at the leaf, we report that there is no record in the table with the value K.

Inserting into a B⁺-tree

Insertion in a B⁺-tree is similar to inserting into other search trees, a new record is always inserted at one of the leaf nodes. The complexity added is that insertion could overflow a leaf node that is already full. When such overflow situations occur a brand new leaf node is added to the B⁺-tree at the same level as the other leaf nodes. The steps to insert into a B⁺-tree are:

1. Follow the path that is traversed as if a Search is being performed on the key of the new record to be inserted.
2. The leaf page L that is reached is the node where the new record is to be indexed.

3. If L is not full then an index entry is created that includes the search key value of the new row and a reference to where new row is in the data file. We are done; this is the easy case!
4. If L is full, then a new leaf node L_{new} is introduced to the B⁺-tree as a right sibling of L. The keys in L along with the an index entry for the new record are distributed evenly among L and L_{new} . L_{new} is inserted in the linked list of leaf nodes just to the right of L. We must now link L_{new} to the tree and since L_{new} is to be a sibling of L, it will then be pointed to by the parent of L. The smallest key value of L_{new} is **copied** and inserted into the parent of L -- which will also be the parent of L_{new} . This entire step is known as commonly referred to as a **split of a leaf node**.
 - a. If the parent P of L is full, then it is split in turn. However, this **split of an internal node** is a bit different. The search key values of P and the new inserted key must still be distributed evenly among P and the new page introduced as a sibling of P. In this split, however, the **middle key is moved to the node above** -- note, that unlike splitting a leaf node where the middle key is copied and inserted into the parent, when you split an internal node the middle key is removed from the node being split and inserted into the parent node. This splitting of nodes may continue upwards on the tree.
 - b. When a key is added to a full root, then the root splits into two and the middle key is promoted to become the new root. This is the only way for a B⁺-tree to increase in height -- when split cascades the entire height of the tree from the leaf to the root.

Deletion

Deletion from a B⁺-tree again needs to be sure to maintain the property that all nodes must be at least half full. The complexity added is that deletion could underflow a leaf node that has only the minimum number of entries allowed. When such underflow situations take place, *adjacent sibling nodes are examined*; if one of them has more than the minimum entries required then, some of its entries are taken from it to prevent a node from "under-flowing". Otherwise, if both adjacent sibling nodes are also at their minimum, then two of these nodes are merged into a single node. The steps to delete from a B⁺-tree are:

1. Perform the search process on the key of the record to be deleted. This search will end at a leaf L.
2. If the leaf L contains more than the minimum number of elements (more than $m/2 - 1$), then the index entry for the record to be removed can be safely deleted from the leaf with no further action.
3. If the leaf contains the minimum number of entries, then the deleted entry is replaced with another entry that can take its place while maintaining the correct order. To find such entries, we inspect the two sibling leaf nodes L_{left} and L_{right} adjacent to L -- at most one of these may not exist.
 - a. If one of these leaf nodes has more than the minimum number of entries, then enough records are transferred from this sibling so that both nodes have the same number of records. This is a

heuristic and is done to delay a future underflow as long as possible; otherwise, only one entry need be transferred. The placeholder key value of the parent node may need to be revised.

- b. If both L_{left} and L_{right} have only the minimum number of entries, then L gives its records to one of its siblings and it is removed from the tree. The new leaf will contain no more than the maximum number of entries allowed. This merge process combines two subtrees of the parent, so the separating entry at the parent needs to be removed -- this may in turn cause the parent node to underflow; such an underflow is handled the same way that an underflow of a leaf node.
- c. If the last two children of the root merge together into one node, then this merged node becomes the new root and the tree loses a level.

Range Search

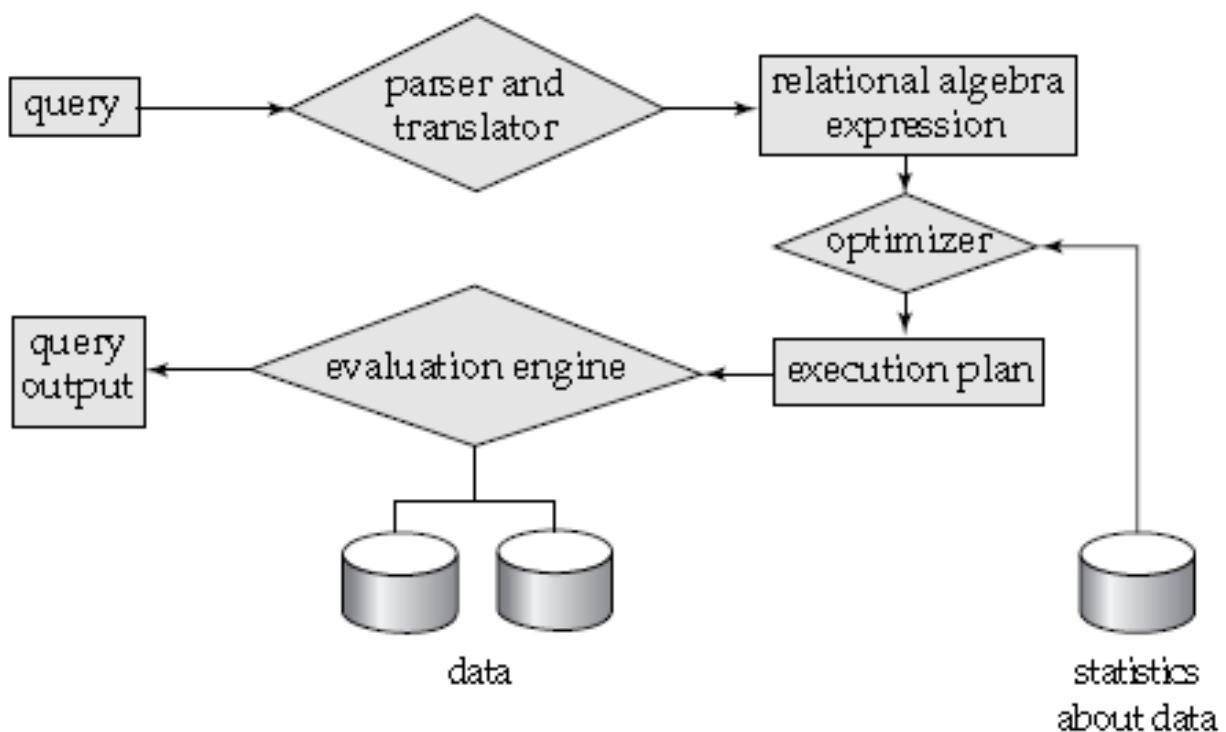
It is common to request records from a table that fall in a specified range. For example, we may want to retrieve all employees making salaries between \$50,000 and \$60,000. B⁺-trees are exceptionally good for range queries. Once the first record in the range has been found using the search algorithm described above, the rest of the records in the range can be found by sequential processing the remaining records in the leaf node, and then continuing down (actually right of the current leaf node) the linked list of leaf nodes as far as necessary. Once a record is found that has a search key value above the upper bound of the requested range, then the search completes. Note that to use a B⁺-tree index to retrieve the employees described above, the B⁺-tree index must exist that has been organized using the salary attribute of the employees table.

11.Explain Query Processing in detail.(Apr 2014)

Basic steps in query processing.

The steps involved in processing a query are

1. Parsing and translation
2. Optimization
3. Evaluation



The terms annotations and query evaluation primitive.

Annotations may state the algorithm to be used for a specific operation, or the particular index or indices to use. A relational-algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**.

Query execution plan and query execution engine

A sequence of primitive operations that can be used to evaluate a query is a **query execution plan** or **query-evaluation plan**. Figure illustrates an evaluation plan in which a particular index is specified for the selection operation. The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.

12.Describe Indexing and Hashing.(Nov 2014)

Indexing and Hashing

Hash tables are often used to implement associative arrays, sets and caches. Like arrays, hash tables provide constant-time $O(1)$ lookup on average, regardless of the number of items in the table. While theoretically the worst-case lookup time can be as bad as $O(n)$, this is for practical purposes statistically impossible unless the hash function is poorly designed or unless the set of keys is maliciously chosen with the given hash function in mind.

Compared to other associative array data structures, hash tables are most useful when large numbers of records are to be stored, especially if the size of the data set can be predicted.

Hash tables may be used as in-memory data structures. Hash tables may also be adopted for use with persistent data structures; database indexes sometimes use disk-based data structures based on hash tables, although balanced trees are more popular.

Static Hashing

Hash File Organization

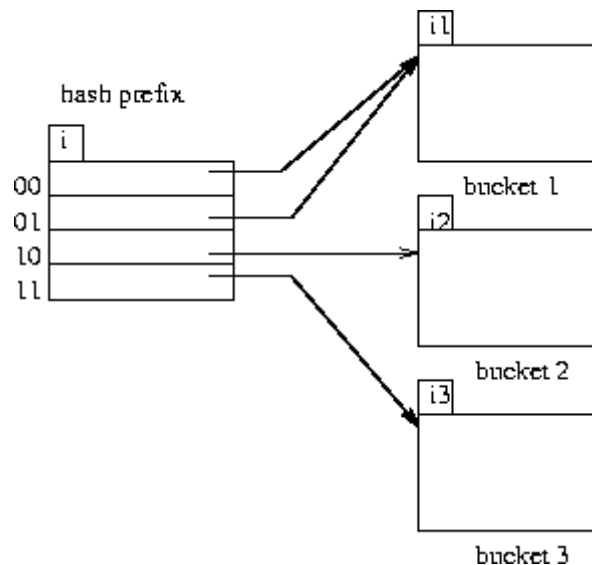
1. **Hashing** involves computing the address of a data item by computing a function on the search key value.
2. A **hash function h** is a function from the set of all search key values K to the set of all bucket addresses B .
 - We choose a number of buckets to correspond to the number of search key values we will have stored in the database.
 - To perform a lookup on a search key value K_i , we compute $h(K_i)$, and search the bucket with that address.
 - If two search keys i and j map to the same address, because $h(K_i) = h(K_j)$, then the bucket at the address obtained will contain records with both search key values.
 - In this case we will have to check the search key value of every record in the bucket to get the ones we want.
 - Insertion and deletion are simple.

Dynamic Hashing

1. As the database grows over time, we have three options:
 - Choose hash function based on current file size. Get performance degradation as file grows.
 - Choose hash function based on anticipated file size. Space is wasted initially.
 - Periodically re-organize hash structure as file grows. Requires selecting new hash function, recomputing all addresses and generating new bucket assignments. Costly, and shuts down database.

2. Some hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinking of the database. These are called **dynamic hash functions**.

- **Extendable hashing** is one form of dynamic hashing.
- Extendable hashing splits and coalesces buckets as database size changes.
- This imposes some performance overhead, but space efficiency is maintained.
- As reorganization is on one bucket at a time, overhead is acceptably low.



Index Definition in SQL

1. Some SQL implementations includes data definition commands to create and drop indices. The IBM SAA-SQL commands are

An index is created by

create index <index-name> **on** *r* (<attribute-list>)

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

To create an index on *bname* for the *branch* relation: **create index** *b-index* **on** *branch* (*bname*)

If the search key is a candidate key, we add the word **unique** to the definition:

create unique index *b-index* **on** *branch* (*bname*)

If *bname* is not a candidate key, an error message will appear.

If the index creation succeeds, any attempt to insert a tuple violating this requirement will fail.

The **unique** keyword is redundant if primary keys have been defined with integrity constraints already.

2. To remove an index, the command is
3. **drop index** <index-name>

Multiple key Access

1. For some queries, it is advantageous to use multiple indices if they exist.
2. If there are two indices on deposit, one on *bname* and one on *cname*, then suppose we have a query like
3. **select** balance
4. **from** deposit
5. **where** *bname* = "Perryridge" **and** balance = 1000
6. There are 3 possible strategies to process this query:
 - Use the index on *bname* to find all records pertaining to Perryridge branch. Examine them to see if balance = 1000
 - Use the index on balance to find all records pertaining to Williams. Examine them to see if *bname* = "Perryridge".
 - Use index on *bname* to find pointers to records pertaining to Perryridge branch. Use index on balance to find pointers to records pertaining to 1000. Take the **intersection** of these two sets of pointers.
7. The third strategy takes advantage of the existence of multiple indices. This may still not work well if
 - There are a large number of Perryridge records **AND**
 - There are a large number of 1000 records **AND**
 - Only a small number of records pertain to both Perryridge and 1000.
8. To speed up multiple search key queries special structures can be maintained.

13. Describe Query Optimization in detail. (Nov 2014)

Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex.

Query optimization is a function of many relational database management systems. The **query optimizer** attempts to determine the most efficient way to execute a given query by considering the possible query plans.

Generally, the query optimizer cannot be accessed directly by users: once queries are submitted to database server, and parsed by the parser, they are then passed to the query optimizer where optimization occurs. However, some database engines allow guiding the query optimizer with hints.

A query is a request for information from a database. It can be as simple as "finding the address of a person with SS# 123-45-6789," or more complex like "finding the average salary of all the employed married men in California between the ages 30 to 39, that earn less than their wives." Queries results are generated by accessing relevant database data and manipulating it in a way that yields the requested information. Since database structures are complex, in most cases, and especially for not-very-simple queries, the needed data for a query can be collected from a database by accessing it in different ways, through different data-structures, and in different orders. Each different way typically requires different processing time. Processing times of the same query may have large variance, from a fraction of a second to hours, depending on the way selected. The purpose of query optimization, which is an automated process, is to find the way to process a given query in minimum time. The large possible variance in time justifies performing query optimization, though finding the exact optimal way to execute a query, among all possibilities, is typically very complex, time consuming by itself, may be too costly, and often practically impossible. Thus query optimization typically tries to approximate the optimum by comparing several common-sense alternatives to provide in a reasonable time a "good enough" plan which typically does not deviate much from the best possible result.

14. Explain about Indexing concept with respect to database system. (APRIL 2015)

A search key is any attribute or set of attribute, it need not be the primary key or even a super key for easy access is called Indexing.

Types of Indexing: (5 Marks)

Ordered indices: Based on a sorted ordering of the values.

Hash indices: Based on a uniform distribution of values across a range of bucket. The bucket to which a value is assigned is determined by a function, called a hash function.

An index record appears for every search key value in the file. **(6 Marks)**

In a dense clustering index, the index record contains search key value and a pointer to the first data record with that search key value.

An index record appears for only some of the search key values in a file. Such a

Indices with two or more levels are called multilevel indices, multilevel indices are closely related to the tree structures such as binary tree used for memory indexing.

11 MARKS

1. Explain authorization in SQL. With example **(April 2011) [Question No. 01]**
2. Explain Database design process in detail **(April 2011)[Question No. 02]**
3. Describe Magnetic Disk and Flash storage in detail **(April 2012)[Question No. 03]**
4. Explain Join operation in detail **(April 2012) [Question No. 04]**
5. Write short notes on **(April 2013)[Question No. 05]**
 - (a) Assertions.
 - (b) Security and Authorization.
6. Describe about Normalization using functional dependency **(April 2013)(NOV 2015)[Question No. 06]**
7. Explain referential integrity in detail **(Nov 2010) [Question No. 07]**
8. Explain any one Database system of your own with normalization **(Nov 2010) [Question No. 08]**
9. Briefly explain RAID **(Nov 2012)(April 2014) [Question No. 09]**
10. Explain about B+Tree index files **(Nov 2012)[Question No. 10]**
11. Explain Query processing in detail. **(April 2014) [Question No. 11]**
12. Describe Indexing and Hashing.**(Nov 2014) [Question No. 12]**
13. Describe Query Optimization in detail. **(Nov 2014) [Question No. 13]**
14. Explain about Indexing concept with respect to database system. **(Apr 2015) [Question No. 14]**

UNIT -IV

Query Processing: Measures of Query Cost – Selection Operation – Sorting – Join Operation – Other Operations – Evaluation of Expressions

Query Optimization – Overview – Transformation of Relational Expressions – Estimating Statistics of Expression Results – Choice of Evaluation Plan

Transactions – Concept – A Simple Transaction Model – Storage Structure – Transaction Atomicity and Durability – Transaction Isolation – Serializability – Transaction Isolation and Atomicity – Transaction Isolation Levels – Implementation of Isolation Levels – Transactions as SQL Statements

TWO MARK

1. What is transaction? (NOV 2014)(NOV 2015)

Transaction is a unit of a program execution that accesses and possibly updates various data items

2. What are the types of transaction?

- a. begin transaction
- b. end transaction

3. What are the properties of transaction?

- Atomicity
- Consistency
- Isolation
- durability

4. What is atomicity?

Either all the operation of the transaction is reflected properly in the database or none are reflected properly in the database.

5. What is consistency?

Execution of the transaction in isolation preserves the consistency of the database.

6. What is isolation?

Even though multiple transactions may execute concurrently. The system guarantees every pair of transaction T_i and T_j . it appears to T_i that either T_j finished the execution before T_i started. Thus each transaction is unaware of other transaction executing concurrently in the system.

7. What is durability?

After the transaction completes successfully the change it has made to the database persist even if there are system failures.

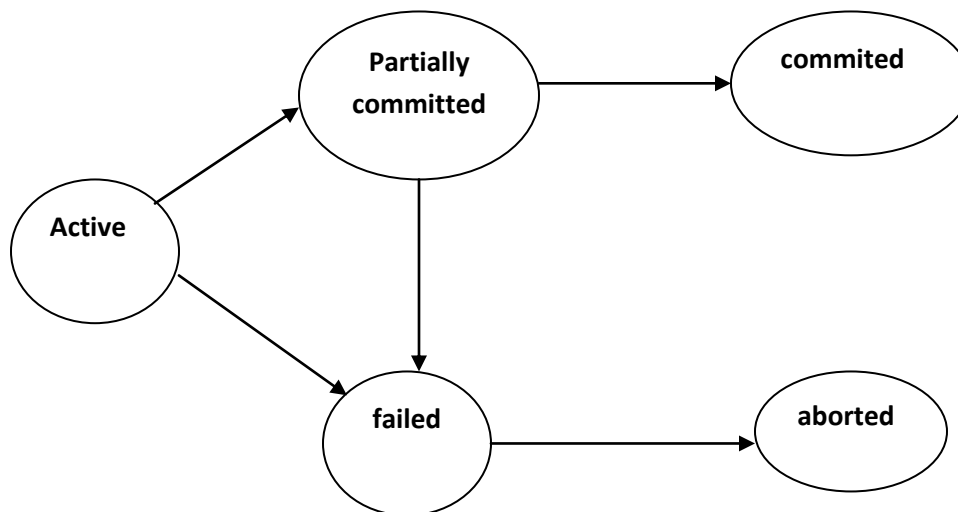
8. What are the operations of the transaction?

- Read(x) –which transfers the data items x from the database to the local buffer belonging to the transaction that executed the read operation.
- Write(x)- which transfers the data items x from the local buffer of the transaction that executed the write back to the database.

9. What is the transaction state?

- Active
- Partially committed
- Failure
- Aborted
- Committed

10. Draw the state diagram of the transaction? (APR 2014)



11. What is reduce waiting time?

Concurrent execution reduces unpredictable delays in a running transaction. Moreover it is also reduces the average response time. The average time for the transaction to be completed after it has been submitted.

12. What is serializability? (NOV 2015)

The database system must control concurrent execution of the transaction to ensure that the database state remains consistent.

13. What are the two types of serializability?

- Conflict serializability
- View serializability

14. What is conflict serializability?

- $I_i = \text{read}(q), I_j = \text{read}(q)$
- $I_i = \text{read}(q), I_j = \text{write}(q)$
- $I_i = \text{write}(q), I_j = \text{read}(q)$
- $I_i = \text{write}(q), I_j = \text{write}(q)$

15. What is view serializability?

- For each data item Q if transaction T_i reads the initial value of q in schedule S. then the transaction T_i must in schedule S' also reads the initial value Q.
- For each data item Q if the transaction T_i executes $\text{read}(Q)$ in schedule S and if that value was produced by the $\text{write}(Q)$ operation executed by the transaction T_j .
- For each data item Q the transaction that performs the final $\text{write}(Q)$ operation in schedule S must perform the final $\text{write}(Q)$ operation in schedule S' .

16. What is recoverability?

In a system that allows concurrent execution, it is necessary also to ensure that any transaction T_j that is dependent on T_i is also aborted. To achieve this we need to place restriction on the type of schedules permitted in the system.

17. What is cascading rollback?

The phenomenon in which single transaction failure leads to the series of transaction rollback is called cascading rollback.

18. What is cascadeless schedule?

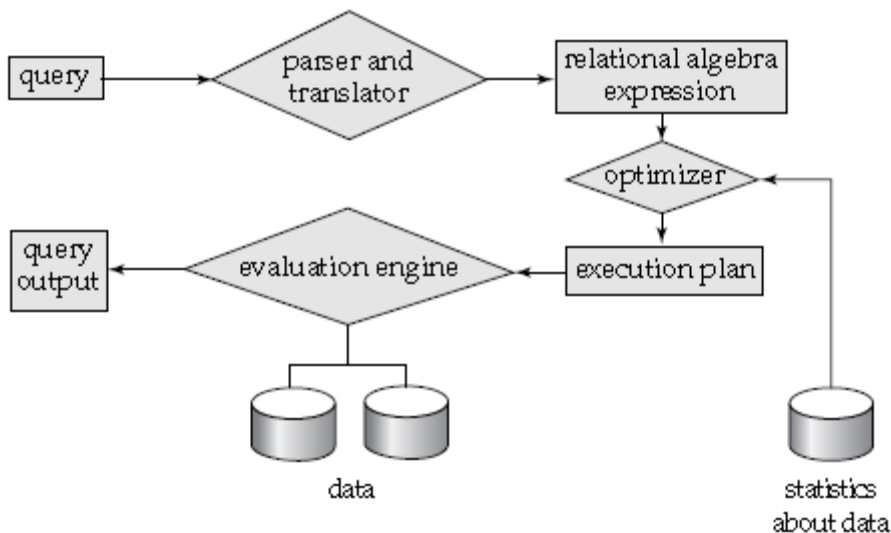
Cascading rollback is undesirable since it leads to the undoing of the significant amount of work. It is desirable to restrict the schedule to those where cascading rollbacks cannot occur. Such schedule are called cascadeless schedule.

19. Basic steps in query processing.

The steps involved in processing a query are

1. Parsing and translation
2. Optimization
3. Evaluation

20. Describe the diagrammatic representation of query processing.

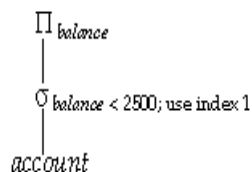


21. Explain the terms annotations and query evaluation primitive.

Annotations may state the algorithm to be used for a specific operation, or the particular index or indices to use. A relational-algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**.

22. What is query execution plan and query execution engine

A sequence of primitive operations that can be used to evaluate a query is a **query execution plan** or **query-evaluation plan**. Figure illustrates an evaluation plan in which a particular index is specified for the selection operation. The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.



23. What are the various measures of query cost.

The cost of query evaluation can be measured in terms of a number of different resources, including

- i. disk accesses ,
- ii. CPU time to execute a query
- iii. the cost of communication
- iv. the response time for a query-evaluation plan.

24. Explain the terms used in block transfer.

Rotational latency: waiting for the desired data to spin under the read–write head

Seek Time: the time that it takes to move the head over the desired track or cylinder

Sequential I/O: where the blocks read are contiguous on disk

Random I/O: where the blocks are noncontiguous

25. What are the basic algorithms in selection operation

- Linear Search A1 algorithm.
- Binary Search A2 algorithm

26. What is Linear search.

In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. For a selection on a key attribute, the system can terminate the scan if the required record is found, without looking at the other records of the relation.

27. What is Binary search.

If the file is ordered on an attribute, and the selection condition is an equality comparison on the attribute, we can use a binary search to locate records that satisfy the selection. The system performs the binary search on the blocks of the file.

28. Explain selection using index structures.

Index structures are referred to as **access paths**. Search algorithms that use an index are referred to as **index scans**.

- **A3 (primary index, equality on key).** For an equality comparison on a key attribute with a primary index, we can use the index to retrieve a single record that satisfies the corresponding equality condition.
- **A4 (primary index, equality on nonkey).** We can retrieve multiple records by using a primary index when the selection condition specifies an equality comparison on a nonkey attribute, A . The only difference from the previous case is that multiple records may need to be fetched.
- **A5 (secondary index, equality).** Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may get retrieved if the indexing field is not a key.

29. Explain the implementation of complex queries.

- **Conjunction:** A *conjunctive selection* is a selection of the form

$$\bullet \sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$$

- **Disjunction:** A *disjunctive selection* is a selection of the form

$$\bullet \sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$$

- **Negation:** The result of a selection $\sigma_{\neg \theta}(r)$ is the set of tuples of r for which the condition θ evaluates to false.

30. What is External sorting. (Nov 2014)

Sorting of relations that do not fit in memory is called **external sorting**. The most commonly used technique for external sorting is the **external sort-merge** algorithm. Let M denote the number of page frames in the main-memory buffer (the number of disk blocks whose contents can be buffered in main memory).

- In the first stage, a number of sorted **runs** are created; each run is sorted, but contains only some of the records of the relation.
- In the second stage, the runs are *merged*. Suppose, for now, that the total number of runs, N , is less than M , so that we can allocate one page frame to each run and have space left to hold one page of output.

31. What is block nested join.

Block nested-loop join is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, every tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples. As before, all pairs of tuples that satisfy the join condition are added to the result.

32. What are the types of join operation used.

- Nested loop join
- Block nested loop join
- Indexed nested loop join
- Merge join
- Hash join

33. What is hash table overflow?

Hash-table overflow occurs in partition i of the build relation s if the hash index on H_{si} is larger than main memory. Hash-table overflow can occur if there are many tuples in the build relation with the same values for the join attributes, or if the hash function does not have the properties of randomness and uniformity

34. What is Fudge factor?

We can handle a small amount of skew by increasing the number of partitions so that the expected size of each partition (including the hash index on the partition) is somewhat less than the size of memory. The number of partitions is therefore increased by a small value called the **fudge factor**.

35. What is overflow resolution?

Overflow resolution is performed during the build phase, if a hash-index overflow is detected. Overflow resolution proceeds in this way: If H_{si} , for any i , is found to be too large, it is further

partitioned into smaller partitions by using a different hash function. Similarly, H_{ri} is also partitioned using the new hash function, and only tuples in the matching partitions need to be joined.

36.What is overflow avoidance?

Overflow avoidance performs the partitioning carefully, so that overflows never occur during the build phase. In overflow avoidance, the build relation s is initially partitioned into many small partitions, and then some partitions are combined in such a way that each combined partition fits in memory. The probe relation r is partitioned in the same way as the combined partitions on s , but the sizes of H_{ri} do not matter.

37.What are the ways in which pipeline can be executed.

Pipelines can be executed in either of two ways:

- **Demand driven:** In a **demand-driven pipeline**, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple.
- **Producer driven:** operations do not wait for requests to produce tuples, but instead generate the tuples **eagerly**. Each operation at the bottom of a pipeline continually generates output tuples, and puts them in its output buffer, until the buffer is full.

38.What is query optimization.(April 2014) (April 2015)

Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex.

39.How authorization provided to the users in SQL? (April 2011)

Developing an application using a least-privileged user account (LUA) approach is an important part of a defensive, in-depth strategy for countering security threats. The LUA approach ensures that users follow the principle of least privilege and always log on with limited user accounts. Administrative tasks are broken out using fixed server roles, and the use of the **sys admin** fixed server role is severely restricted.

40.What is an audit trail? (April 2011)

An audit trail is a log of changes (insert, delete, update) to the database along with information such as which user performed the change and when the change was performed.

41.List out the various storage devices (April/May 2012)

Storage Devices are the data storage devices that are used in the computers to store the data. The computer has many types of data storage devices. Some of them can be classified as the removable data Storage Devices and the others as the non removable data Storage Devices.

The memory is of **two types**; one is the **primary memory** and the other one is the **secondary memory**.

The primary memory is the volatile memory and the secondary memory is the non volatile memory. The volatile memory is the kind of the memory that is erasable and the non volatile memory is the one where in the contents cannot be erased. Basically when we talk about the data storage devices it is generally assumed to be the secondary memory. The secondary memory is used to store the data permanently in the computer. The secondary storage devices are usually as follows: hard disk drives – this is the most common type of storage device that is used in almost all the computer systems. The other ones include the floppy disk drives, the CD ROM, and the DVD ROM. The flash memory, the USB data card etc.

42. What are the steps in query processing? (April 2013)

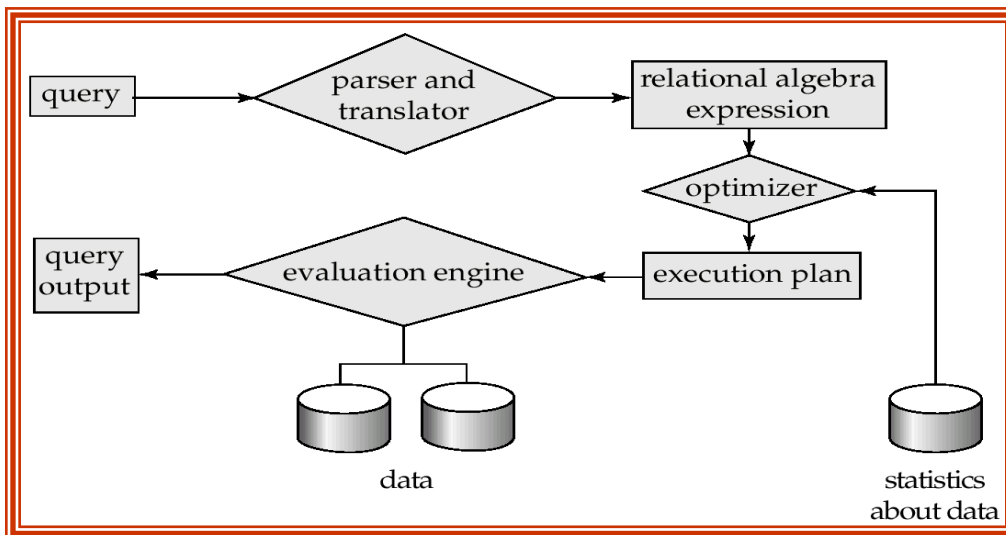
- Parsing and Translation
- Optimization
- Evaluation

11 ARKS

1. Briefly explain Query processing and Optimization in Oracle. (Nov 2013) (April 2014)
(Nov 2014) (NOV 2015)

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Parsing and translation

- translate the query into its internal form. This is then translated into relational algebra.
- Parser checks syntax, verifies relations Evaluation
- The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Optimization

A relational algebra expression may have many equivalent expressions

- ★ E.g., $\sigma_{balance < 2500}(\Pi_{balance}(account))$ is equivalent to

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

Each relational algebra operation can be evaluated using one of several different algorithms correspondingly, a relational-algebra expression can be evaluated in many ways. Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.

- ★ E.g., can use an index on *balance* to find accounts with balance < 2500,
- ★ or can perform complete relation scan and discard accounts with balance ≥ 2500

i)Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost.

- ★ Cost is estimated using statistical information from the database catalog
 - n e.g. number of tuples in each relation, size of tuples, etc.

In this chapter we study

- ★ How to measure query costs
- ★ Algorithms for evaluating relational algebra operations
- ★ How to combine algorithms for individual operations in order to evaluate a complete expression

ii)Measures of Query Cost

Cost is generally measured as total elapsed time for answering query

- ★ Many factors contribute to time cost
 - è *disk accesses, CPU, or even network communication*

Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account

- ★ Number of seeks * average-seek-cost
- ★ Number of blocks read * average-block-read-cost
- ★ Number of blocks written * average-block-write-cost
 - è Cost to write a block is greater than cost to read a block
 - data is read back after being written to ensure that the write was successful

For simplicity we just use *number of block transfers from disk* as the cost measure

- ★ We ignore the difference in cost between sequential and random I/O for simplicity
- ★ We also ignore CPU costs for simplicity

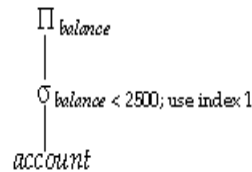
Costs depends on the size of the buffer in main memory

- ★ Having more memory reduces need for disk access
- ★ Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine ahead of actual execution
- ★ We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

Evaluation

Annotations may state the algorithm to be used for a specific operation, or the particular index or indices to use. A relational-algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**.

A sequence of primitive operations that can be used to evaluate a query is a **query execution plan** or **query-evaluation plan**. Figure illustrates an evaluation plan in which a particular index is specified for the selection operation.



The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.

2. Describe the following

(April 2012)

- A). Transaction Isolation
- B). Serializability

A) Transaction isolation

- ⇒ Transaction-processing systems usually allow multiple transactions to run concurrently.
- ⇒ Allowing multiple transactions to update data concurrently causes several complications with consistency of the data.
- ⇒ There are two good reasons for allowing concurrency:
 - **Improved throughput and resource utilization.** A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. Therefore, I/O activity can be done in parallel with processing at the CPU. All of this increases the throughput of the system correspondingly; the processor and disk utilization also increases.
 - **Reduced waiting time.** There may be a mix of transactions running on a system, some short and some long. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.
- ⇒ The motivation for using concurrent execution in a database is essentially the same as the motivation for using multiprogramming in an operating system.
- ⇒ Let T_1 and T_2 be two transactions that transfer funds from one account to another. Transaction T_1 transfers \$50 from account A to account B . It is defined as:

T_1 : read (A);

```

A: =A - 50;
write (A);
read (B);
B: =B + 50;
write (B).

```

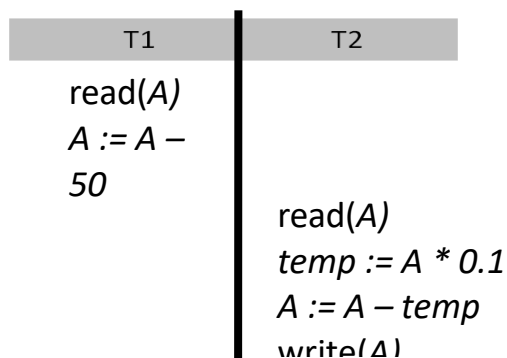
- Transaction *T2* transfers 10 percent of the balance from account *A* to account *B*. It is defined as:

```

T2: read (A);
temp: =A * 0.1;
A: =A - temp;
write (A);
read (B);
B: =B + temp;
write (B).

```

- Suppose the current values of accounts *A* and *B* are \$1000 and \$2000, respectively.
- Suppose also that the two transactions are executed one at a time in the order *T1* followed by *T2*. This execution sequence appears as follows:



Schedule 1—a serial schedule in which *T1* is followed by *T2*.

- The final values of accounts *A* and *B*, after the execution in this figure takes place, are \$855 and \$2145, respectively.

⇒ Similarly, if the transactions are executed one at a time in the order $T2$ followed by $T1$, then the corresponding execution sequence is as follows:

T1	T2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Schedule 2—a serial schedule in which $T2$ is followed by $T1$.

⇒ Again, as expected, the sum $A + B$ is preserved, and the final values of accounts A and B are \$850 and \$2150, respectively.

⇒ The execution sequences just described are called schedules.

⇒ suppose that the two transactions are executed concurrently as follows:

T1	T2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule 3—a concurrent schedule equivalent to schedule 1.

- ⇒ After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order $T1$ followed by $T2$. The sum $A + B$ is indeed preserved.
- ⇒ Not all concurrent executions result in a correct state. To illustrate, consider the schedule as follows:

T1	T2
read(A) $A := A - 50$	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(R)
Write(A) read(B) $B := B + 50$ write(B)	$B := B + temp$ write(B)

Schedule 4—a concurrent schedule.

- ⇒ After the execution of this schedule, we arrive at a state where the final values of accounts A and B are \$950 and \$2100, respectively.
- ⇒ This final state is an *inconsistent state*.

B) Serializability

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact.

Two operations:

read and write. We thus assume that, between a $read(Q)$ instruction and a $write(Q)$ instruction on a data item Q , a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction.

Thus, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions.

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)

Schedule3-showing only the read and write instructions

Conflict Serializability

Let us consider a schedule S in which there are two consecutive instructions I_i and I_j , of transactions T_i and T_j , respectively ($i \neq j$). If I_i and I_j refer to different data items, then we can swap I_i and I_j without affecting the results of any instruction in the schedule.

However, if I_i and I_j refer to the same data item Q , then the order of the two steps may matter.

Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. $I_i = \text{read}(Q)$, $I_j = \text{read}(Q)$. The order of I_i and I_j does not matter, since the same value of Q is read by T_i and T_j , regardless of the order.
2. $I_i = \text{read}(Q)$, $I_j = \text{write}(Q)$. If I_i comes before I_j , then T_i does not read the value of Q that is written by T_j in instruction I_j . If I_j comes before I_i , then T_i reads the value of Q that is written by T_j . Thus, the order of I_i and I_j matters.
3. $I_i = \text{write}(Q)$, $I_j = \text{read}(Q)$. The order of I_i and I_j matters for reasons similar to those of the previous case.
4. $I_i = \text{write}(Q)$, $I_j = \text{write}(Q)$. Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j . However, the value obtained by the next $\text{read}(Q)$ instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other $\text{write}(Q)$ instruction after I_i and I_j in S , then the order of I_i and I_j directly affects the final value of Q in the database state that results from schedule S .

The write(*A*) instruction of *T1* conflicts with the read(*A*) instruction of *T2*.

However, the write(*A*) instruction of *T2* does not conflict with the read(*B*) instruction of *T1*, because the two instructions access different data items.

View Serializability

Consider two schedules *S* and *S*₁, where the same set of transactions participates in both schedules. The schedules *S* and *S*₁ are said to be view equivalent if three conditions are met:

1. For each data item *Q*, if transaction *T_i* reads the initial value of *Q* in schedule *S*, then transaction *T_i* must, in schedule *S*₁, also read the initial value of *Q*.
2. For each data item *Q*, if transaction *T_i* executes read(*Q*) in schedule *S*, and if that value was produced by a write(*Q*) operation executed by transaction *T_j*, then the read(*Q*) operation of transaction *T_i* must, in schedule *S*₁, also read the value of *Q* that was produced by the same write(*Q*) operation of transaction *T_j*.
3. For each data item *Q*, the transaction (if any) that performs the final write(*Q*) operation in schedule *S* must perform the final write(*Q*) operation in schedule *S*₁.

T1	T2
read(A)	
A:=A-50	
Write(A)	
	read(B)
	B:=B-10
	write(B)
read(B)	

Schedule 8.

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

T3	T4	T6
Read(Q)	Write(Q)	
Write(Q)		

Schedule 9—a view-serializable schedule.

6. Discuss fixed length records in detail with an example

FIXED-LENGTH RECORDS:

For an example, let us consider a file of account records for our bank database. Each record of this file is defined as:

```

type deposit=record
    account-number:char(10);
    branch-name:char(22);
    balance:real;
end

```

- If we assume that each character occupies 1 byte and that a real occupies 8 bytes, our account record is 40 bytes long. A simple approach is to use the first 40 bytes for the first record, the next 40 bytes for the second record, and so on.
- Though, we have two problems with this simple approach:
 1. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of this file, or we must have way of marking deleted records so that they can be ignored.
 2. Unless the block happens to be a multiple of 40(which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.
 3. When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record and so on, until every record following the deleted record has been moved ahead.

FIG 1.2 ,WITH RECORD 2 DELETED AND ALL RECORDS MOVED

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

- It might be easier simply to move the final record of the file into the space occupied by the deleted record.(fig-1.3)

Fig-1.3 WITH RECORD 2 DELETED AND FINAL RECORD MOVED

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

- A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.
- At the beginning of the file, we allocate a certain number of bytes as a FILE HEADER.
- The header will contain a variety of information about the file.
- For now, all we need to store there is the address of the first record whose contents are deleted. We use this first record to store the address of the second available record and so on.
- These deleted records form a linked list, which is often referred to as a FREE LIST.

Fig 1.4, WITH FREE LIST AFTER DELETION OF RECORDS 1,4,AND 6

Header			
Record 0	A-102	Perryridge	400
Record 1			

Record 2	A-215	Mianus	700
Record 3	A-101	Downtown	500
Record 4			
Record 5	A-201	Perryridge	900
Record 6			
Record 7	A-110	Downtown	600
Record 8	A-218	Perryridge	700

- On insertion of a new record, we use the record pointed to by the header.
- We change the header pointer to point to the next available record. If no space is available, we add the new file to the end of the record.
- ◆ **NOTE:** Insertion and deletion for files of fixed-length records are simple to implement, because
- ◆ The space made available by a deleted record is exactly the space needed to insert a record. An inserted record may not fit in the space left free by a deleted record, or it may fill only part of that space.

8. What is the transactions isolation level in SQL? How to implementation of isolation level. Discuss it.

isolation levels:

Serializable

Repeatable read

Read committed

Read uncommitted

Implementation:

Locking

Timestamps

Multiple Versions and Snapshot isolation

Note:

Refer book for notes

9. Explain about Transaction control in detail. (April 2015) (11 Marks)

TRANSACTIONS

- Collections of operations that form a single logical unit of work are called as transactions.
- A transaction is a unit of program execution that accesses and possibly updates various data items.
- To ensure integrity of the data, the database system maintain the following properties of the transactions:
 - **Atomicity.** Either all operations of the transaction are reflected properly in the database, or none.
 - **Consistency.** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
 - **Isolation.** Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
 - **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transactions access data using **two operations**:

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

Transaction State:

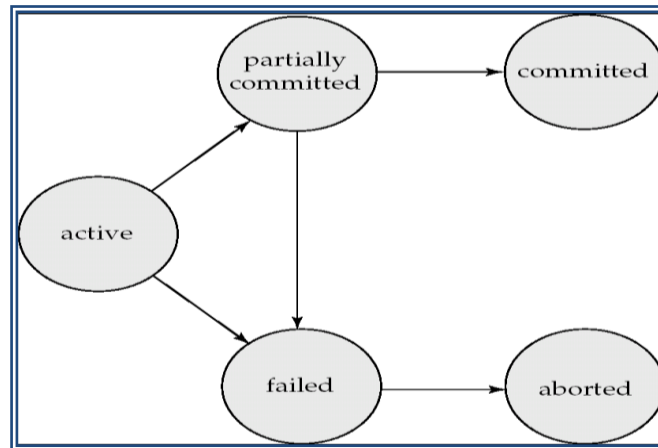
(April 2014)

A transaction must be in one of the following states:

1. Active, the initial state; the transaction stays in this state while it is executing.
 2. Partially committed, after the final statement has been execute.
 3. Failed, after the discovery that normal execution can no longer Proceed.
 4. Aborted, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
 5. Committed, after successful completion.
- ❖ A transaction starts in the active state.
 - ❖ When it finishes its final statement, it enters the partially committed state.
 - ❖ The database system then writes out enough information to disk.
 - ❖ If a transaction enters the failed state such a transaction must be rolled back.
 - ❖ Then, it enters the aborted state. At this point, the system has two options:
 - It can restart the transaction.

- It can kill the transaction.

❖ The state diagram corresponding to a transaction appears as follows:



Atomicity and

➤ Recovery-

component of a database system can support atomicity and durability by a variety of schemes.

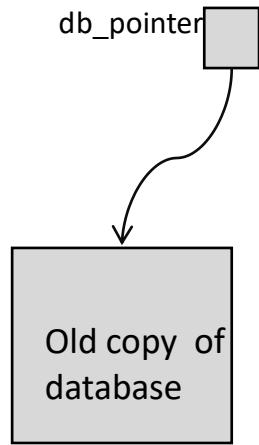
Shadow Copy

- It is simple, but extremely inefficient
- It is based on making copies of the database, called *shadow* copies, assumes that only one transaction is active at a time.
- A pointer called db-pointer is maintained on disk; it points to the current copy of the database.
- A transaction that wants to update the database,
 - first creates a complete copy of the database
 - All updates are done on the new database copy,
 - leaving the original copy, the shadow copy, untouched.
 - If at any point the transaction has to be aborted, the system merely deletes the new copy.
 - The old copy of the database has not been affected.

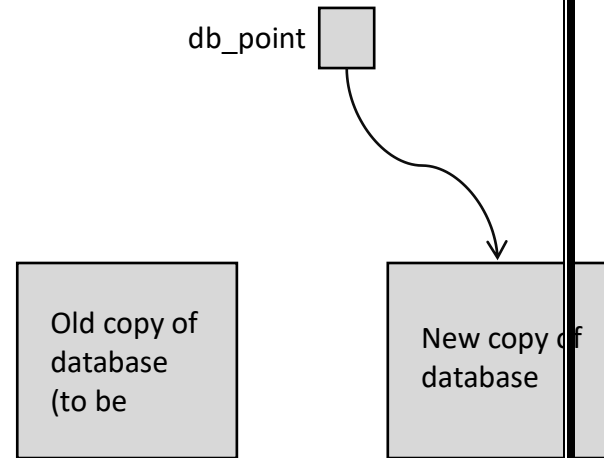
Implementation of

Durability:

management



(a) Before update



(b) After update

Shadow-copy technique for atomicity and durability.

- The transaction is said to have been *committed* at the point where the updated db-pointer is written to disk.
- If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected.
- Suppose that the system fails at any time before the updated db-pointer is written to disk.
- Then, when the system restarts, it will read db-pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database.
- Next, suppose that the system fails after db-pointer has been updated on disk.
- Before the pointer is updated, all updated pages of the new copy of the database were written to disk.
- Thus, the atomicity and durability properties of transactions are ensured by the shadow-copy implementation of the recovery-management component.

11.Explain the transactions properties in detail with examples

TRANSACTIONS

- Collections of operations that form a single logical unit of work are called as transactions.

- A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does.
- A transaction is a unit of program execution that accesses and possibly updates various data items.
- Usually, a transaction is initiated by a user program written in a High-level data-manipulation language or programming language, where it is delimited by statements (or function calls) of the form begin transaction and end transaction.
- To ensure integrity of the data, the database system maintain the following properties of the transactions:
 - Atomicity. Either all operations of the transaction are reflected properly in the database, or none are.
 - Consistency. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
 - Isolation. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
 - Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Properties of Transactions

ATOMICITY

all or nothing

CONSISTENCY

no violation of integrity constraints

ISOLATION

concurrent changes invisible to serializable

DURABILITY

committed updates persist

- These properties are often called the ACID properties; the acronym is derived from the first letter of each of the four properties.

- Transactions access data using two operations:
 1. **read(X)**, which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.
 2. **write(X)**, which transfers the data item X from the local buffer of the transaction that executed the write back to the database.
- Let T_i be a transaction that transfers \$50 from account A to account B . This transaction can be defined as

```

Ti: read (A);
A: = A - 50;
write (A);
read (B);
B: = B + 50;
write (B).

```

- Consistency: The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction.
- Atomicity: Suppose that, just before the execution of transaction T_i the values of accounts A and B are \$1000 and \$2000, respectively. Suppose that the failure happened after the write (A) operation but before the write (B) operation. In this case, the values of accounts A and B reflected in the database are \$950 and \$2000. The system destroyed \$50 as a result of this failure. Such a state is termed as inconsistent state. If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.
- Durability: The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. We can guarantee durability by ensuring that either
 - The updates carried out by the transaction have been written to disk before the transaction completes.

- Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the update when the database system is restarted after the failure.
- Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state. A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. Ensuring the isolation property is the responsibility of a component of the database system called the concurrency-control component executes transactions serially—that is, one after the other.

UNIVERSITY QUESTIONS

2. Describe the following **(April 2012) (April 2014)** [Question No.: 02]

A). Transaction Isolation & Transaction State

B). Serializability , Conflict Serializability **(NOV 2015)**

3. Discuss about recovery and Atomicity **(April 2012)** [Question No.: 05]

4. Discuss fixed length records in detail with an example. **(April 2013)** [Question No.: 06]

5. Explain storage access in detail **(Nov 2010)** [Question No.: 07]

6. What is the transactions isolation level in SQL? How to implementation of isolation level.

Discuss it. **[Question No.: 08]**

7. Explain about the Concurrency Control. **(NOV 2014)** [Question No.: 04]

8. Explain about the Recovery System for Database Management System. **(NOV 2014)**
[Question No.: 05]

9. Explain about Transaction control in detail. **(APRIL 2015)** [Question No.: 09]

10. Briefly explain Query processing and Optimization in Oracle. **(Nov 2013) (April 2014) (Nov 2014)**
(NOV 2015) [Question No.: 10]

UNIT -V

Concurrency Control – Lock Based Protocols – Deadlock Handling – Multiple Granularity – Timestamp Based Protocols – Validation Based Protocols – Multiversion Schemes – Snapshot Isolation – Insert Operations, Delete Operations and Predicate Reads – Weak Levels of Consistency – Concurrency in Index Structures – Recovery – Failure Classification – Storage – Recovery and Atomicity – Recovery Algorithm – Buffer Management – Failure with Loss of Nonvolatile Storage – Early Lock Release and Logical Undo Operations.

Case Studies IBM DB2 Universal Database – My SQL.

1. What are the two types of locks? (APR 2014)(NOV 2015)

- Shared lock (0r) Term lock- if the transaction T_i has obtained the shared mode lock(denoted by S) on item Q then T_i can read but cannot write Q
- Exclusive lock- if the transaction T_i has obtained the exclusive mode lock(denoted by X) on item Q then T_i can both read and write Q.

2. What is deadlock?

The state where neither of these transaction can ever proceed with its normal execution. This situation is called deadlock.

3. What is locking protocol?

Each transaction in the system follows the set of rules called locking protocol, indicating when a transaction may lock and unlock each of the data item.

4. What are the two phases of the locking protocol?

- Growing phase- a transaction may obtain locks but may not release any lock.
- Shrinking phase- a transaction may release lock but may not obtain any new lock

5. What is timestamp based protocol?

The method of determining the serializability order is to select an ordering among transaction in advance. The common method for doing so is to use a timestamp ordering scheme.

6. What is timestamp?

With each transaction T_i in the system , we associate the unique fixed timestamp denoted by the $TS(T_i)$. This timestamp is assigned by the database system before the transaction T_i starts the execution.

7. What are the two timestamp values?

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

8. What is timestamp ordering protocol?

Timestamp ordering protocol ensures that any conflicting read and write operation are executed in the timestamp order.

9. What is Thomas' write protocol?

The modification of the timestamp ordering protocol is called thomas' write protocol. Suppose the transaction T_i issues write(Q)

- If $TS(T_i) < R\text{-timestamp}(Q)$ then the value of Q that T_i producing was previously needed and that it had been assumed that the value would never been produced. Hence the system rejects the write operation and rolls T_i back.
- If $TS(T_i) < W\text{-timestamp}(Q)$ then T_i is attempting to write an obsolete value of Q. hence this write operation can be ignored.
- Otherwise the system executes the write operation and sets $W\text{-timestamp}(Q)$ to $TS(T_i)$.

10. What are the three phases in validation based protocol?

- Read phase
- Validation phase
- Write phase

11. What are the three different timestamp in validation based protocol?

- Start(T_i)
- Validation(T_i)
- Finish(T_i)

12. What is the optimistic concurrency control?

The transaction executes optimistically assuming they will be able to finish execution and validate at the end. This validation scheme is called optimistic concurrency control

13. What the two principle methods in the deadlock problem?

- Deadlock prevention
- Deadlock detection and recovery

14. What are the two types of deadlock prevention scheme using timestamp?

- Wait – die scheme is a non preemptive technique.
- Wound-wait scheme is a preemptive scheme

15. What is starvation?

No transaction gets rolled back repeatedly and is never allowed to make progress.

16. What are the actions to be followed to recover from deadlock?

- Selection of a victim
- Rollback
- Starvation

17. What are the classifications of failure?

- Transaction failure
 - ✓ Logical error
 - ✓ System error
- System crash
- Disk failure

18. What is meant by Log Based Recovery? (NOV 2014)(NOV 2015)

- A log is kept on stable storage.
- The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a $\langle T_i$

start>log record

- Before T_i executes write(X), a log record $\langle T_i, X, V1, V2 \rangle$ is written,
- where $V1$ is the value of X before the write, and $V2$ is the value to be written to x .
- Log record notes that T_i has performed a write on data item

19. What are the properties of transactions? (April 2015)

- **ACID property:** Atomicity, Consistency, Isolation and Durability

20. Distinguish between optimization and pessimistic locking (April 2015)

- In pessimistic locking, when a user opens a data to update it, a lock is granted. Other users can only view the data until the whole transaction of the data update is completed.

- In optimistic locking, a data is opened for updating by multiple users. A lock is granted only during the update transaction and not for the entire session. Due to this concurrency is increased and is a practical approach of updating the data.

21.What is PostgreSQL?

PostgreSQL (pronounced "post-gress-Q-L") is an open source relational database management system (DBMS) developed by a worldwide team of volunteers. PostgreSQL is not controlled by any corporation or other private entity and the source code is available free of charge.

22. What is Oracle?

This is a suite of tools for various aspects of application development, including tools for forms development, data modeling, reporting, and querying. The suite supports the UML standard for development modeling. Oracle Repository provides configuration management for database objects, forms applications, Java classes, ML files, and other types of files

23. What is IBM DB2 Universal Database?

DB2 has a long history and is considered by many to have been the first database product to use SQL (also developed by IBM) although Oracle released a *commercial* SQL database product somewhat earlier than IBM.

24. What is My SQL?

My SQL is a relational database management system (RDBMS), and ships with no GUI tools to administer My SQL databases or manage data contained within the databases. Users may use the included command line tools, or use My SQL "front-ends", desktop software and web applications that create and manage My SQL databases, build database structures, back up data, inspect status, and work with data records.

25.What is Microsoft SQL Server?

Microsoft SQL Server is a relational database management system developed by Microsoft. As a database, it is a software product whose primary function is to store and retrieve data as requested by other software applications, be it those on the same computer or those running on another computer across a network (including the Internet).

26.What is the database design tool used in Oracle?

The major database design tool used in Oracle is Oracle Designer, which translates business logic and data flow into a schema definitions and procedural scripts for application logic. Oracle Designer stores the design in Oracle Repository, which serves as a single point of metadata for the application.

27. What is a Warehouse Builder?

Warehouse Builder is a tool for design and deployment of all aspects of a data warehouse, including schema design, data mapping and transformations, data load processing, and metadata management. Oracle Warehouse Builder supports both 3NF and star schemas and can also import designs from Oracle Designer.

28. What is Oracle Discoverer?

Oracle Discoverer is a Web-based, ad-hoc query, reporting, analysis and Web publishing tool for end users and data analysts. It allows users to drill up and down on result sets, pivot data, and store calculations as reports that can be published in a variety of formats such as spreadsheets or HTML.

29. What is Oracle Express Server?

Oracle Express Server is a multidimensional database server. It supports a wide variety of analytical queries as well as forecasting, modeling and scenario management.

30. State the Object - Relational Features in Oracle. (APRIL 2014)

Oracle has extensive support for object – relational constructs, including:

- Object Types
- Collection Types
- Object Tables
- Table Functions
- Object Views
- Methods
- User – defined aggregate functions
- XML data types

31. What is a segment?

The space in a table is divided into units, called segments, that each contains data for a specific data structure. There are four types of segments.

1. Data Segments
2. Index Segments
3. Temporary Segments
4. Rollback Segments

32. What are the ways by which partitioning is provided by Oracle?

Oracle supports various kinds of horizontal partitioning of tables and indices, and this feature plays a major role in Oracles' ability to support very large databases. The several types of partitioning provided are:

1. Range Partitioning
2. Hash Partitioning

3. Composite Partitioning
4. List Partitioning

33.What are the various access methods provide by Oracle?

Data can be accessed through a variety of access method:

1. Full table scan.
2. Index scan.
3. Index fast full scan.
4. Index joins.
5. Cluster and hash cluster access.

34.State some of the major types of transformations and rewrites supported by Oracle. (NOV 2014)

Some of the major types of transformations and rewrites supported by Oracle are as follows:

1. View merging
2. Complex view merging
3. Sub query flattening
4. Materialized view rewrite
5. Star transformation.

35.What is SQL* Loader?

Oracle has a direct load utility, SQL* Loader, that supports fast parallel loads of large amounts of data from external files. It supports variety of data formats and it can perform various filtering operations on the data being loaded.

36.What is Free Space Control Record?

DB2 uses a space map record called *Free Space Control Record* (FSCR) to find free space in the table. The FCSR record usually contains a space map for 500 pages.

37.What are the various access methods supported by DB2?

DB2 supports a comprehensive set of access methods on relational tables, including:

1. Table scan
2. Index scan
3. Index-only scan
4. List pre fetch scan
5. Index and-ing scan
6. Index or-ing scan

38.State the various modes provided by DB2 for isolation.

For isolation, DB2 supports the repeatable read (RR), read stability (RS), cursor stability (CS), and uncommitted read (UR) modes.

39.What is DB2 Data Propagator?

DB2 Data Propagator is a product in the DB2 family that provides replication of data among DB2, other relational database systems such as Oracle, Microsoft SQL Server, Sybase SQL Server, and Informix, and non relational data sources.

40.State the various tools provided by DB2 for administration.

Additionally, DB2 supports tools such as:

1. Audit facility – maintaining audit trace of database actions.
8. Governor facility – controls the priority and execution times.
9. Query patroller facility – managing the query jobs in the system.
10. Trace and diagnostic facility for debugging.
11. Event monitoring facility – tracks the resources and events during system execution.

41.What are the uses of SQL Query Analyzer?

A database administrator or developer can use SQL Query Analyzer to:

1. Analyze queries.
2. Format SQL queries.
3. Use templates for stored procedures, functions, and basic SQL statements.
4. Drag object names from the Object Browser to the Query window.
5. Define personal keystroke and toolbar options.

42.What is SQL Profiler?

(APRIL 2014) (NOV 2014)

SQL Profiler is a graphical utility that allows database administrators to monitor and record database activity. SQL Profiler can display all server activity in real time, or it can create filters that focus on the actions of particular users, applications, or types of commands.

43.What are the various stages in compiling an SQL Statement?

An SQL statement is compiled as follows.

1. Parsing / binding.
2. Simplification / normalization.
3. Cost-based optimization.
4. Plan preparation.

44.How is logging implemented in SQL Server?

The transaction log records all changes made to the database and stores enough information to allow any changes to be undone (rolled back) or redone (rolled forward) in the event of a system failure or rollback request.

45. What are the different uses of memory within SQL Server process?

There are many different uses of memory within the SQL Server Process:

1. Buffer pool.
2. Dynamic memory allocation.
3. Plan and execution cache.
4. Large memory grants.

46. When we call a transaction is terminated? **(April 2011)**

- As long as you don't **COMMIT** or **ROLLBACK** a transaction, it's still "running" and potentially holding locks.
- If your client (application or user) closes the connection the database, any still running transactions will be rolled back and terminated.

47. What are errors cause a transaction failure? **(April 2011)**

- Transaction failure
 - ✓ Logical error
 - ✓ System error
- System crash
- Disk failure

48. What is the use pivot operator? **(April/May 2012)**

PIVOT can be used to generate cross tabulation reports to summarize data as it creates a more easy understandable data in a user friendly format. PIVOT rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output i.e it rotates a rows to columns and aggregations where they are required on any remaining column values that are wanted in the final output.

49. Explain a shadow copy scheme **(April 2013)**

It is simple, but efficient, scheme called the shadow copy schemes. It is based on making copies of the database called shadow copies that one transaction is active at a time. The scheme also assumes that the database is simply a file on disk.

50. Mention the two approaches to manage the Database buffer. **(April 2013)**

A buffer is an 8-KB page in memory, the same size as a data or index page. Thus, the buffer cache is divided into 8-KB pages. The buffer manager manages the functions for reading data or index pages from the database disk files into the buffer cache and writing modified pages back to disk. A page remains in the buffer cache until the buffer manager needs the buffer area to read in more data. Data is

written back to disk only if it is modified. Data in the buffer cache can be modified multiple times before being written back to disk. For more information, see Reading Pages and Writing Pages.

51.Name any four query languages

(April 2013)

- SQL
- MY SQL
- Oracle
- MS Access

52.What is locking protocol?

(Nov 2010)

A lock is a mechanism to control concurrent access to a data item. Data items can be locked in two modes :

1. exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using lock-X instruction.
2. shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction. Lock requests are made to concurrency-control manager

54. List any 4 DB2 background process.

(April 2013)

db2agent - For user process
db2cart - for archive
db2dlock - dead lock detector
db2event - event handler
db2fcmdm - FCM daemon

55. Name any 4 query Languages.

(April 2013)

DQL[Doctrine Query Language],PHQL[Phalcon Query Language], SQL[Structure Query Language] and XML Query Language.

56. What are the features of DB2? (April 2015)

CPU/Performance Improvements
Virtual Storage Enhancements
Security Extensions
Improved Catalog Concurrency

57. State any four background process in Oracle? (April 2015)

Checkpoint Process (CKPT)
System Monitor Process (SMON)
Process Monitor Process (PMON)

Recoverer Process (RECO)

58. What is your understanding about atomicity?(NOV 2015)

Atomicity. In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are.

11 MARKS

1. Explain Microsoft SQL server architecture.

(April 2012) (Nov 2014)

Microsoft SQL Server is a relational database management system developed by Microsoft. As a database, it is a software product whose primary function is to store and retrieve data as requested by other software applications, be it those on the same computer or those running on another computer across a network (including the Internet). There are at least a dozen different editions of Microsoft SQL Server aimed at different audiences and for workloads ranging from small single-machine applications to large Internet-facing applications with many concurrent users. Its primary query languages are T-SQL and ANSI SQL.

Architecture

The protocol layer implements the external interface to SQL Server. All operations that can be invoked on SQL Server are communicated to it via a Microsoft-defined format, called Tabular Data Stream (TDS). TDS is an application layer protocol, used to transfer data between a database server and a client. Initially designed and developed by Sybase Inc. for their Sybase SQL Server relational database engine in 1984, and later by Microsoft in Microsoft SQL Server, TDS packets can be encased in other physical transport dependent protocols, including TCP/IP, Named pipes, and Shared memory. Consequently, access to SQL Server is available over these protocols. In addition, the SQL Server API is also exposed over web services.^[42]

2. Give an overview of IBM DB2 process and architecture in detail.

(April 2013) (Nov 2013)

DB2 has a long history and is considered by many^[who?] to have been the first database product to use SQL (also developed by IBM) although Oracle released a *commercial* SQL database product somewhat earlier than IBM did.

The name **DB2** was first given to the Database Management System or DBMS in 1983 when IBM released **DB2** on its MVS mainframe platform. Prior to this, a similar product was named SQL/DS on the VM platform. The earlier System 38 platform also contained a relational DBMS. System Relational, or System R, was a research prototype developed in the 1970s. **DB2** has its roots back to the beginning of the 1970s when E.F. Codd, working for IBM, described the theory of relational databases and in June 1970 published the model for data manipulation. To apply the model Codd needed a relational database language he named Alpha. At the time IBM didn't believe in the potential of Codd's ideas, leaving the implementation to a group of programmers not under Codd's supervision, who violated several fundamentals of Codd's relational model; the result was Structured English QUery Language or SEQUEL. When IBM released its first relational database product, they wanted to have a commercial-

quality sublanguage as well, so it overhauled SEQUEL and renamed the basically new language (System Query Language) SQL to differentiate it from SEQUEL.

When Informix acquired Illustra and made their database engine an object-SQL DBMS by introducing their Universal Server, both Oracle and IBM followed suit by changing their database engines to be capable of object-relational extensions. In 2001, IBM bought Informix and in the following years incorporated Informix technology into the DB2 product suite. Today, DB2 can technically be considered to be an object-SQL DBMS.

For some years DB2, as a full-function DBMS, was exclusively available on IBM mainframes. Later IBM brought DB2 to other platforms, including OS/2, UNIX and Windows servers, then Linux (including Linux on zSeries) and PDAs. This process occurred through the 1990s. The inspiration for the mainframe version of DB2's architecture came in part from IBM IMS, a hierarchical database, and its dedicated database manipulation language, IBM DL/I. DB2 is also embedded in the i5/OS operating system for IBM System i (iSeries, formerly the AS/400), and versions are available for z/VSE and z/VM.

An earlier version of the code that would become DB2 LUW (Linux, Unix, Windows) was part of an Extended Edition component of OS/2 called Database Manager. IBM extended the functionality of Database Manager a number of times, including the addition of distributed database functionality that allowed shared access to a database in a remote location on a LAN. Eventually IBM declared that insurmountable complexity existed in the Database Manager code, and took the difficult decision to completely rewrite the software in their Toronto Lab. The new version of Database Manager, called DB2 like its mainframe parent, ran on the OS/2 and RS/6000 platforms, was called DB2/2 and DB2/6000 respectively. Other versions of DB2, with different code bases, followed the same '/' naming convention and became DB2/400 (for the AS/400), DB2/VSE (for the DOS/VSE environment) and DB2/VM (for the VM operating system). IBM lawyers stopped this handy naming convention from being used and decided that all products needed to be called "product FOR platform" (for example, DB2 for OS/390). The next iteration of the mainframe and the server-based products were named DB2 Universal Database (or DB2 UDB), a name that had already been used for the Linux-Unix-Windows version, with the introduction of widespread confusion over which version (mainframe or server) of the DBMS was being referred to. At this point, the mainframe version of DB2 and the server version of DB2 were coded in entirely different languages (PL/S for the mainframe and C++ for the server), but shared similar functionality and used a common architecture for SQL optimization: the Starburst Optimizer.

Over the years DB2 has both exploited and driven numerous hardware enhancements, particularly on IBM System z with such features as Parallel Sysplex data sharing. In fact, DB2 UDB Version 8 for z/OS now requires a 64-bit system and cannot run on earlier processors, and DB2 for z/OS maintains certain unique software differences in order to serve its sophisticated customers.

Although the ultimate expression of software-hardware co-evolution is the IBM mainframe, to some extent that phenomenon occurs on other platforms as well, as IBM's software engineers collaborate with their hardware counterparts.

In the mid-1990s, IBM released a clustered DB2 implementation called DB2 Parallel Edition, which initially ran on AIX. This edition allowed scalability by providing a shared nothing architecture, in which a single large database is partitioned across multiple DB2 servers that communicate over a high-speed interconnect. This DB2 edition was eventually ported to all Linux, UNIX, and Windows (LUW) platforms and was renamed to DB2 Extended Enterprise Edition (EEE). IBM now refers to this product as the Database Partitioning Feature (DPF) and sells it as an add-on to their flagship DB2 Enterprise product.

In mid 2006, IBM announced "Viper," which is the codename for DB2 9 on both distributed platforms and z/OS. DB2 9 for z/OS was announced in early 2007. IBM claimed that the new DB2 was the first relational database to store XML "natively". Other enhancements include OLTP-related improvements for distributed platforms, business intelligence/data warehousing-related improvements for z/OS, more self-tuning and self-managing features, additional 64-bit exploitation (especially for virtual storage on z/OS), stored procedure performance enhancements for z/OS, and continued convergence of the SQL vocabularies between z/OS and distributed platforms.

In October 2007, IBM announced "Viper 2," which is the codename for DB2 9.5 on the distributed platforms. There were three key themes for the release, Simplified Management, Business Critical Reliability and Agile XML development.

In June 2009, IBM announced "Cobra" (the codename for DB2 9.7 for LUW). DB2 9.7 adds data compression for database indexes, temporary tables, and large objects. DB2 9.7 also supports native XML data in hash partitioning (database partitioning), range partitioning (table partitioning), and multi-dimensional clustering. These native XML features allows users to directly work with XML in data warehouse environments. DB2 9.7 also adds several features that make it easier for Oracle Database users to work with DB2. These include support for the most commonly-used SQL syntax, PL/SQL syntax, scripting syntax, and data types from Oracle Database. DB2 9.7 also enhanced its concurrency model to exhibit behavior that is familiar to users of Oracle Database and Microsoft SQL Server.

In October 2009, IBM introduced its second major release of the year when it announced DB2 pureScale. DB2 pureScale is a database cluster solution for non-mainframe platforms, suitable for Online Transaction Processing (OLTP) workloads. IBM based the design of DB2 pureScale on the Parallel Sysplex implementation of DB2 data sharing on the mainframe. DB2 pureScale provides a fault-tolerant architecture and shared-disk storage. A DB2 pureScale system can grow to 128 database servers, and provides continuous availability and automatic load balancing.

In 2009, it was announced that DB2 can be an engine in MySQL. This allows users on the System i platform to natively access the DB2 under the IBM i operating system (formerly called OS/400), and for users on other platforms to access these files through the MySQL interface. On the System i and its predecessors the AS/400 and the System/38, DB2 is tightly integrated into the operating system, and comes as part of the operating system. It provides journaling, triggers and other features.

In October 2010, IBM announced the general availability (GA) of DB2 10 for z/OS. DB2® 10 for z/OS® expands the value delivered to businesses by IBM's industry-leading mainframe data server through innovations in key areas:

- Improved operational efficiencies for "out-of-the-box" DB2 CPU savings
- Unsurpassed resiliency for business-critical information
- Rapid application and warehouse deployment for business growth
- Enhanced business analytics and data visualization solutions with QMF

Selected features that deliver these valuable benefits to any business include:

- When compared to running on DB2 9, depending on the workload, customers may experience reduced CPU utilization
- When compared to running DB2 9, up to five to ten times more concurrent users on a single subsystem by avoiding memory constraints
- Greater concurrency for data management, data definition, and data access, including DDL, BIND, REBIND, PREPARE, utilities, and SQL
- Additional online changes for data definitions, utilities, and subsystems
- Improved security with better granularity for administrative privileges, data masking, and audit capabilities
- Temporal or versioned data to understand system and business times at the database level (Bi-temporal feature is not available on Oracle or any other competing RDBMS products)
- pureXML™ and SQL enhancements to simplify portability from other database solutions
- Productivity improved for database administrators, application programmers, and systems administrators
- QMF Classic Edition, an optional for-charge feature, providing greater interoperability with other programs plus improved queries, forms, diagnostics, performance, and resource control

3. Explain various model of locking a data item. Also explain two-phases protocol.(NOV 2015)

Lock:

- Mechanism to control concurrent access to a data item.
- Lock requests are made to concurrency-control manager.
- Transaction can proceed only after request is granted. Data items can be locked in two modes:
 1. exclusive mode (X): Data item can be both read as well as written. X-lock is requested using lock-X(A) instruction.
 2. shared mode (S): Data item can only be read. S-lock is requested using lock-S(A) instruction. Locks can be released: U-lock(A)

Locking protocol:

A set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules. Ensure serializable schedules by delaying transactions that might violate serializability.

Lock-compatibility matrix tells whether two locks are compatible or not. Any number of transactions can hold shared locks on a data item. If any transaction holds an exclusive lock on a data item no other transaction may hold any lock on that item.

Locking Rules/Protocol:

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Pitfalls of Lock-Based Protocols:

- Too early unlocking can lead to non-serializable schedules. Too late unlocking can lead to deadlocks.

Example

Transaction T1 transfers \$50 from account B to account A. Transaction T2 displays the total amount of money in accounts A and B, that is, the sum of A + B.

Early unlocking can cause incorrect results, non-serializable schedules. If A and B get updated in-between the read of A and B, the displayed sum would be wrong.

e.g., A = \$100, B = \$200, display A + B shows \$250

1. X-lock(B)
2. read B
3. B := B-50

4. write B
5. U-lock(B)
6. S-lock(A)
7. read A
8. U-lock(A)
9. S-lock(B)
10. read B
11. U-lock(B)
12. display A + B
13. X-lock(A)
14. read A
15. A := A+50
16. write A
17. U-lock(A)

T1 T2

Late unlocking causes deadlocks. Neither $T1$ nor $T2$ can make progress: executing lock-S(B) causes $T2$ to wait for $T1$ to release its lock on B . executing lock-X(A) causes $T1$ to wait for $T2$ to release its lock on A . To handle a deadlock

one of $T1$ or $T2$ must be rolled back and its locks released.

1. X-lock(B)
2. read(B)
3. B := B-50
4. write(B)
5. S-lock(A)
6. read(A)
7. S-lock(B)
8. X-lock(A)

T1 T2

Two-Phase Locking Protocol:

A locking protocol that ensures conflict-serializable schedules. It works in two phases:

- *Phase 1: Growing Phase: transaction may obtain locks transaction may not release locks*
- *Phase 2: Shrinking Phase: transaction may release locks transaction may not obtain locks*
- When the first lock is released, the transaction moves from phase 1 to phase 2.

- Properties of the Two-Phase Locking Protocol. Ensures serializability It can be shown that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).
- Does not ensure freedom from deadlocks. Cascading roll-back is possible. Modifications of the two-phase locking protocol

Strict two-phase locking

- * A transaction must hold all its exclusive locks till it commits/aborts
- * Avoids cascading roll-back

Rigorous two-phase locking

- All locks are held till commit/abort. Transactions can be serialized in the order in which they commit.
- Refine the two-phase locking protocol with lock conversions

Phase 1: Acquire a lock-S on item can acquire a lock-X on item can convert a lock-S to a lock-X (upgrade)

Phase 2: can release a lock-S, can release a lock-X, can convert a lock-X to a lock-S (downgrade)

- * Ensures serializability; but still relies on the programmer to insert the various locking instructions.
- * Strict and rigorous two-phase locking (with lock conversions) are used extensively in DBMS.

Automatic Acquisition of Locks:

- A transaction T_i issues the standard read/write instruction without explicit locking calls (by the programmer).
- The operation $\text{read}(D)$ is processed as:
 if T_i has a lock on D then
 $\text{read}(D)$
 else
 if necessary wait until no other transaction has a lock-X on D ;
 grant T_i a lock-S on D ;
 $\text{read}(D)$;
 end
- $\text{write}(D)$ is processed as:
 if T_i has a lock-X on D then
 $\text{write}(D)$
 else
 if necessary wait until no other transaction has any lock on D ;

```
if  $T_i$  has a lock-S on  $D$  then
  upgrade lock on  $D$  to lock-X
else
  grant  $T_i$  a lock-X on  $D$ ;
end
write( $D$ );
end
```

All locks are released after commit or abort

Implementation of Locking:

- A lock manager can be implemented as a separate process to which transactions send lock and unlock requests.
- The lock manager replies to a lock request by sending a lock grant message (or a message asking the transaction to roll back, in case of a deadlock).
- The requesting transaction waits until its request is answered.
- The lock manager maintains a data structure called a lock table to record granted locks and pending requests.

Lock table:

- Implemented as in-memory hash table indexed on the data item being locked. Black rectangles indicate granted locks.
- White rectangles indicate waiting requests. Records also the type of lock granted/requested.
- Processing of requests:
 - New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks.
 - Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted.
 - If transaction aborts, all waiting or granted requests of the transaction are deleted. Index on transaction to implement this efficiently.

4. Explain deadlock handling in detail

(Nov 2012)

In order to understand deadlock, let us consider the following example:

T1	T2	Concurrency Controller
lock-X(A)		Lock granted
	lock-X(B)	Lock granted
lock-X(B)		Request is queued i.e T1 will wait for T2 to release the lock on data item B
	lock-X(A)	Request is again queued i.e T2 will wait for T1 to release the lock on data item A

As shown in partial schedule transaction T1 is waiting for transaction T2 to release its lock on data item B and transaction T2 is waiting for transaction T1 to release its lock on data item A. Such a cycle of transactions waiting for locks to be released is called a Deadlock.

Clearly, these two transactions will make no further progress. Now we can define the deadlock as "A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set."

More precisely, there exists a set of waiting transactions (T0, T1, Tn) such that T0 is waiting for a data item that is held by T1 and T1 is waiting for a data item that is held by T2 and

..... Tn-1 is waiting for a data item that is held by Tn and Tn is waiting for a data item that is held by T0.

None of the transactions can make progress in such a situation.

There are two principal methods for dealing with the deadlock problem.

- Deadlock prevention
- Deadlock detection

Deadlock prevention: We can use a deadlock-prevention protocol to ensure that the system will never enter a deadlock state.

Deadlock detection: In this case, we can allow the system to enter a deadlock state, and then try to recover using a deadlock detection and deadlock recovery scheme.

Both the above methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise detection and recovery are more efficient.

Deadlock Prevention

We can prevent deadlocks by giving each transaction a priority and ensuring that lower priority transactions are not allowed to wait for higher priority transactions (or vice versa). One way to assign priorities is to give each transaction a timestamp when it starts up. The lower the timestamp, the higher the transaction priority that is, the oldest transaction has the highest priority.

If a transaction Ti requests a lock and transaction Tj holds a conflicting lock, the lock manager can use one of the following two policies:

Wait-die

If T_i has higher priority, it is allowed to wait; otherwise it is aborted. It means when transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp smaller than that of T_j (that is T_i is older than T_j). Otherwise T_i is rolled back (dies).

Example

Suppose that transactions T_{22} , T_{23} and T_{24} have timestamps 5, 10 and 15 respectively. If T_{22} requests a data item held by T_{23} then T_{22} will wait. If T_{24} requests a data item held by T_{23} then T_{24} will be rolled back.

T_{22} will Wait

T_{24} will rollback

T_{22} -----> T_{23} <----- T_{24}

(5)

(10)

(15)

The wait-die scheme is non-preemptive scheme because only a transaction requesting a lock can be aborted. As a transaction grows older (and its priority increases), it tends to wait for more and more younger transactions.

Wound-wait

If T_i has higher priority, abort T_j otherwise T_i waits. It means when transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp larger than that of T_j (that is T_i is younger than T_j). Otherwise T_j is rolled back (T_j is wounded by T_i).

Example:

Returning to our previous example, with transactions T_{22} , T_{23} and T_{24} , if T_{22} requests a data item held by T_{23} then the data item will be preempted from T_{23} and T_{23} will be rolled back. If T_{24} requests a data item held by T_{23} , and then T_{24} will wait.

T_{22} access

T_{23} will be

Data item

rollback

T_{22} -----> T_{23}

(5)

(10)

Data item with T_{23}

wait for T_{23}

T_{23} <-----

T_{24}

(10)

(15)

This scheme is based on a preemptive technique and is a counterpart to the wait-die scheme. In the wait-die scheme, lower priority transactions can never wait for higher priority transactions. In the wound-wait scheme, higher priority transactions never wait for lower priority transactions. In either case no deadlock cycle can develop.

When a transaction is aborted and restarted, it should be given the same timestamp that it had originally. Reissuing timestamps in this way ensures that each transaction will eventually become the oldest transaction, and thus the one with the highest priority, and will get the locks that it requires.

Problem of starvation

Whenever transactions are rolled back, it is important to ensure that there is no starvation that is, no transaction gets rolled back repeatedly and is never allowed to make progress.

Both the wound-wait and the wait-die schemes avoid starvation. At any time, there is a transaction with the smallest timestamp. This transaction cannot be required to roll back in either scheme. Since timestamps always increase, and since transactions are not assigned new timestamps when they are rolled back, a transaction that is rolled back will eventually have the smallest timestamp. Thus it will not be rolled back again.

Differences between wait-die and wound-wait scheme

There are following differences between wait-die and wound-wait schemes:

- In the wait-die scheme an older transaction must wait for a younger one to release its data item. Thus, the older the transaction gets the more it tends to wait. By contrast, in the wound wait scheme, an older transaction never waits for a younger transaction.
- In the wait-die scheme, if a transaction T_{24} request a data item held a data item by T_{23} and T_{24} then die and roll back because T_i request a data held by T_j and $TS(T_i) \gg TS(T_j)$. Now T_{24} restarted and may reissue the same sequence of requests if data item is still held by T_{23} then T_{24} will die again. Thus T_{24} may die several times before acquiring the needed data item.

Now see what happen in case of wound-wait scheme transaction T_i is wound and rollback because T_j request a data item that it holds and $TS(T_i) > TS(T_j)$. Now suppose that data item is held by T_{23} and T_{22} request the data item held by T_{23} . Then T_{23} is wound and rollback because $TS(23) > TS(T_{22})$. When T_{23} will again restarted and request the data item T_{22} then again, $TS(23) > TS(22)$ and T_{23} will wait for transaction T_{22} and T_{23} will not rollback again. Thus, there may be few rollbacks in the wound wait scheme.

Timeout-Based Schemes

Another simple approach to deadlock handling is based on lock timeouts. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur and deadlock detection and recovery.

Uses of Timeout-Based Schemes

The timeout scheme is particularly easy to implement, and works well if transactions are short, and if long waits are likely to be due to deadlocks.

Limitations

- It is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources.
- Starvation is also a possibility with this scheme.

Hence the timeout-based scheme has limited applicability.

5. Discuss in detail about Time stamp - based protocols

TIMESTAMP BASED PROTOCOL:

- Timestamp based protocol is the locking protocols and the order between every pair of conflicting transactions at execution time by the first that both members of the pairs request that involves incompatible modes.
- Another method for determining the serializability order is to select an ordering among transaction in advance. The most common method for doing so is to use a timestamp ordering scheme.

TIMESTAMPS:

- This timestamp is assigned by the database system before the transaction T_i starts execution.
- If a transaction T_i has been assigned timestamp $TS(T_i)$ and a new transaction T_j enters the system, then $TS(T_i) < TS(T_j)$. There are two simple methods for implementing this scheme.
 1. Use the value of the *system clock* as the timestamp, (i.e.) a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
 2. Use a logical counter that is incremented after a new timestamp has been assigned and a transaction timestamp is equal to the value of the counter when the transaction enters the system.

To implement this scheme we associate with each data item Q two timestamp values.

- $W\text{-timestamp}(Q)$ denotes the largest timestamp of any transaction that executed $write(Q)$ successfully.

- $R\text{-timestamp}(Q)$ denotes the largest timestamp of any transaction that executed $read(Q)$ successfully.

THE TIMESTAMP-ORDERING PROTOCOL:

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

1. Suppose that transaction T_i issues $read(Q)$.

- If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten. Hence the read operation is rejected and T_i is rolled back.
- If $TS(T_i) \geq W\text{-timestamp}(Q)$ then the read operation is executed and $R\text{-timestamp}(Q)$ is set to the maximum of $R\text{-timestamp}(Q)$ and $TS(T_i)$.

2. Suppose that transaction T_i issues $write(Q)$.

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

The protocol can generate schedules that are not recoverable. However it can be extended to make the schedules recoverable in one several ways

- Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction.
- Recoverability and cascadelessness can also be guaranteed by using a limited form of locking whereby reads of uncommitted items are postponed until the transaction that updated the item commits.
- Recoverability alone can be ensured by tracking uncommitted writes and allowing a transaction T_i to commit only after the commit of any transaction that wrote a value that T_i read.

Example: The following schedule is possible under the timestamp ordering protocol. Since $TS(T_{14}) < TS(T_{15})$, the schedule must be conflict equivalent to schedule $\langle T_{14}, T_{15} \rangle$

read(B)
 read(B)
 $B := B - 50$
 write(B)
 read(A)
 read(A)
 display(A+B)
 $A := A + 50$
 write(A)
 display(A+B)
 T14 T15

Thomas' Write rule:

- Let us consider schedule 4 and apply the timestamp ordering protocol. Since T16 starts before T17 we shall assume that $TS(T16) < TS(T17)$.
- The read(Q) operation of T16 succeeds as does the write (Q) operation, we find that $TS(T16) < W\text{-timestamp}(Q)$, since $W\text{-timestamp}(Q) = TS(T17)$.
- Thus the write (Q) by T16 is rejected and transaction T16 must be rolled back.
- Although the rollback of T16 is required by the timestamp ordering protocol it is unnecessary. Since T17 has already written Q, the value that T16 is attempting to write is one that will never need to be read.
- Any transaction T_i with $TS(T_i) < TS(T17)$ that attempts a read(Q) will be rolled back. Since $TS(T_i) < W\text{-timestamp}(Q)$.

The modification to the timestamp-ordering protocol called Thomas write rule is this. Suppose that transaction T_i issues write(Q).

1. If $TS(T_i) < R\text{-timestamp}(Q)$, then the value of Q that T_i is producing was previously needed and it had been assumed that the value would never be produced. Hence the system rejects the write operation and rolls T_i back.
2. If $TS(T_i) < W\text{-timestamp}(Q)$ then T_i is attempting to write an obsolete value of Q. Hence the write operation can be ignored.
3. Otherwise the system executes the write operation and sets $W\text{-timestamp}(Q)$ to $TS(T_i)$.

Under Thomas' write rule the write(Q) operation of T16 would be ignored. The result is a schedule that is view equivalent to the serial schedule <T16,T17>.

6. Discuss various type of failures that occur in a system.

Failure Classification

Various types of failure that may occur in a system

1. Transaction failure

a. Logical errors - The Transaction cannot complete due to some internal error condition

b. System errors

The database system must terminate an active transaction due to an error condition. (e.g., deadlock)

2. System crash:

A power failure or other hardware or software failure causes the system to crash.

Fail stop assumption

A non-volatile storage contents are assumed to not be corrupted by system crash. Database systems have numerous integrity checks to prevent corruption of disk data

3. Disk failure

A head crash or similar disk failure destroys all or part of Disk. Destruction is assumed to be detectable: disk drives use checksums to detect failures

7. Discuss about recovery and Atomicity (or)

Discuss about the Recovery System for Database Management Systems.

(NOV 2014)

Recovery system

Recovering a system from failure crash is called recovery system or crash recovery.

Failure Classification:

Various types of failure that may occur in a system

1. Transaction failure

A. Logical errors - The Transaction cannot complete due to some internal error condition

B. System errors - The database system must terminate an active transaction due to an error condition. (e.g., deadlock)

2. System crash:

A power failure or other hardware or software failure causes the system to crash.

Fail stop assumption

A non-volatile storage contents are assumed to not be corrupted by system crash. Database systems have numerous integrity checks to prevent corruption of disk data

3. Disk failure

A head crash or similar disk failure destroys all or part of Disk. Destruction is assumed to be detectable: disk drives use checksums to detect failures

4. Recovery Algorithms

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures.

Recovery algorithms have two parts

6. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
7. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

Storage Structure :

The various data items in the database may be stored and accessed in a number of different storage media.

Storage types:

The types are:

- Volatile: It does not survive system crashes.

eg: main memory, cache memory

- Non volatile storage: It survives system crashes

eg: disk, tape, flash memory, non-volatile (battery backed up) RAM.

- Stable storage: A mythical form of storage that survives all failures approximated by maintaining multiple copies on distinct nonvolatile media.

Stable Storage Implementation

- Maintain multiple copies of each block on separate disks
 - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
 - Successful completion
 - Information arrived safely at its destination
 - Partial failure
 - Destination block has incorrect information
 - Total failure:
 - Destination block was never updated .
- Protecting storage media from failure during data transfer (one solution)
 - Execute output operation as follows (assuming two copies of each block)
 1. Write the information onto the first physical block.
 2. When the first write successfully completes, write the Same information onto the second physical block.
 3. The output is completed only after the second write successfully completes.
- copies of a block may differ due to failure during output operation. To recover from failure:
First find inconsistent blocks:
 - 1. Expensive solution:** Compare the two copies of every disk block.
 - 2. Better solution:** Record in-progress disk writes on non- volatile storage (Nonvolatile RAM or special area of disk).
 - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
 - Used in hardware RAID systems If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

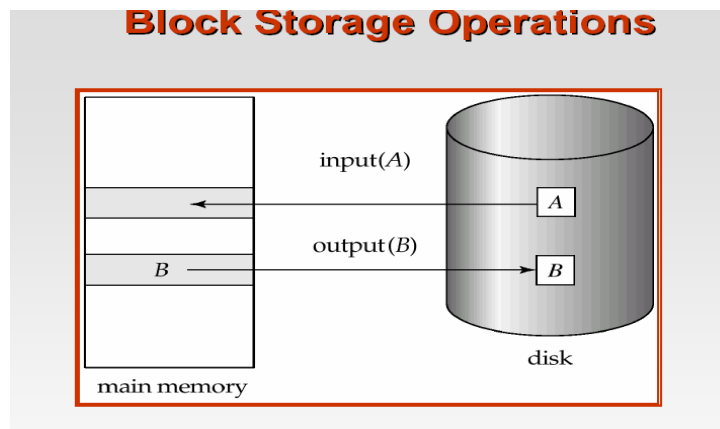
Data Access

- Physical blocks are those blocks residing on the disk.

➤ Buffer blocks are the blocks residing temporarily in main memory.

Block movements between disk and main memory are initiated through the following two operations:

- $\text{input}(B)$ transfers the physical block B to main memory.
- $\text{output}(B)$ transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
- T_i 's local copy of a data item X is called x_i .



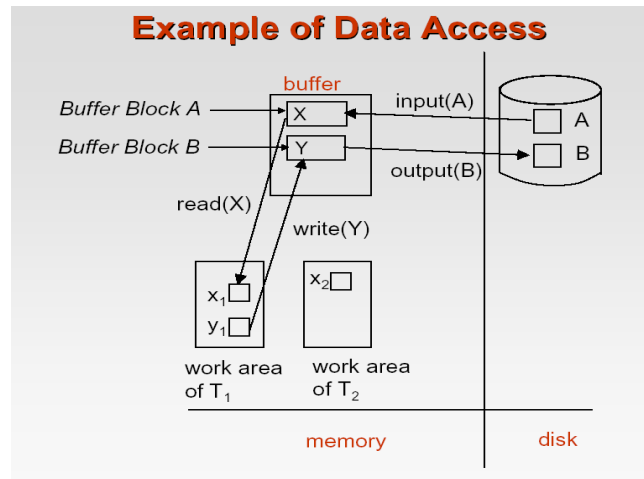
We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
- $\text{read}(X)$ assigns the value of data item X to the local variable x_i .
- $\text{write}(X)$ assigns the value of local variable x_i to data item $\{X\}$ in the buffer block.
- Both these commands may necessitate the issue of an $\text{input}(BX)$ instruction before the assignment, if the block BX in which X resides is not already in memory.

Transactions:

- Perform $\text{read}(X)$ while accessing X for the first time;
- All subsequent accesses are to the local copy. After last access, transaction executes $\text{write}(X)$.

- $\text{output}(BX)$ need not immediately follow $\text{write}(X)$. System can perform the output operation when it deems fit.



Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers \$50 from account A to account B ; goal is either to perform all database modifications made by T_i or none at all.
- Several output operations may be required for T_i (to output A and B). A failure may occur after one of these modifications have been made but before all of them are made.

To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.

Two approaches:

1. log based recovery
2. Shadow paging

Assume that a transaction run serially, that is, one after the other.

Log Based Recovery

- A log is kept on stable storage.
- The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a $\langle T_i$

start>log record

- Before T_i executes write(X), a log record $\langle T_i, X, V1, V2 \rangle$ is written,
- where $V1$ is the value of X before the write, and $V2$ is the value to be written to x .
- Log record notes that T_i has performed a write on data item

X_j had value $V1$ before the write, and will have value $V2$ after the write.

- When T_i finishes its last statement, the log record $\langle T_i$

commit> is written to x

We assume for now that log records are written directly to stable storage (that is, they are not buffered)

- Two approaches using logs
- Deferred database modification
- Immediate database modification

Deferred Database Modification

- The deferred database modification scheme records all modifications to the log, but defers all the writes to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing $\langle T_i \text{ start} \rangle$ record to log.
- A write(X) operation results in a log record $\langle T_i, X, V \rangle$ being

written, where V is the new value for X .

✓ Note: old value is not needed for this scheme.

- The write is not performed on X at this time, but is deferred. When T_i partially commits, $\langle T_i \text{ commit} \rangle$ is written to the log.
- Finally, the log records are read and used to actually execute the previously deferred writes.
- During recovery after a crash, a transaction needs to be redone if and only if both $\langle T_i \text{ start} \rangle$ and $\langle T_i \text{ commit} \rangle$ are there in the log.
- Redoing a transaction T_i (redo T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while the transaction is executing the original updates, or while recovery action is being taken

example: transactions T_0 and T_1 (T_0 executes before T_1):

T_0 : read (A)

T_1 : read (C)

A :- $A - 50$ C :- $C - 100$

Write (A) write (C)

read (B)

B :- $B + 50$

write (B)

Below we show the log as it appears at three instances of time.

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$
$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 600 \rangle$	$\langle T_1, C, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

- If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo(T_0) must be performed since $\langle T_0 \text{ commit} \rangle$ is present

(c) redo(T_0) must be performed followed by redo(T_1) since

$\langle T_0 \text{ commit} \rangle$ and $\langle T_i \text{ commit} \rangle$ are present

Immediate Database Modification

- The immediate database modification scheme allows database updates of an uncommitted transaction to be made as the writes are issued.
- since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written.
- We assume that the log record is output directly to stable storage
- Can be extended to postpone log record output, so long as prior to execution of an output(B) operation for a data block B , all log records corresponding to items B must be flushed to stable storage.

- Output of updated blocks can take place at any time before or after transaction commit.
- Order in which blocks are output can be different from the order in which they are written.

Example :

Log	input	output
Log Write Output		
<T0 start>		
<T0, A, 1000, 950>		
To, B, 2000, 2050		
	A = 950	
	B = 2050	
<T0 commit>		
<T1 start>		
<T1, C, 700, 600>		
	C = 600	
		BB, BC
<T1 commit>		
		BA

Note: BX denotes block containing X.

x1

Recovery procedure has two operations instead of one:

- undo(Ti) restores the value of all data items updated by Ti to their old values, going backwards from the last log record for Ti
- redo(Ti) sets the value of all data items updated by Ti to the new values, going forward from the first log record for Ti.
- Both operations must be idempotent That is, even if the operation is executed multiple times the effect is the same as if it is executed once
- Needed since operations may get re-executed during recovery

When recovering after failure:

- Transaction Ti needs to be undone if the log contains the record
- <Ti start>, but does not contain the record <Ti commit>.
- Transaction Ti needs to be redone if the log contains both the record
- <Ti start> and the record <Ti commit>.

8. Discuss about Database Recovery procedure in detail. (April 2015)(NOV 2015)

Recovery system

Recovering a system from failure crash is called recovery system or crash recovery.

Failure Classification: (6 Marks)

Various types of failure that may occur in a system

1. Transaction failure

A. Logical errors - The Transaction cannot complete due to some internal error condition

B. System errors - The database system must terminate an active transaction due to an error condition. (e.g., deadlock)

2. System crash:

A power failure or other hardware or software failure causes the system to crash.

Fail stop assumption

A non-volatile storage contents are assumed to not be corrupted by system crash. Database systems have numerous integrity checks to prevent corruption of disk data

3. Disk failure

A head crash or similar disk failure destroys all or part of Disk. Destruction is assumed to be detectable: disk drives use checksums to detect failures

4. Recovery Algorithms

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures.

Recovery algorithms have two parts

5. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
6. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

(5Marks)

8. Write about Microsoft SQL server in detail

Microsoft SQL Server is a relational database management system developed by Microsoft. As a database, it is a software product whose primary function is to store and retrieve data as requested by other software applications, be it those on the same computer or those running on another computer across a network (including the Internet). There are at least a dozen different editions of Microsoft SQL Server aimed at different audiences and for workloads ranging from small single-machine applications to large Internet-facing applications with many concurrent users. Its primary query languages are T-SQL and ANSI SQL.

9. Explain Multidimensional Clustering in IBM.

Multidimensional clustering tables

Multidimensional clustering (MDC) provides an elegant method for clustering data in tables along multiple dimensions in a flexible, continuous, and automatic way. MDC can significantly improve query performance, in addition to significantly reducing the overhead of data maintenance operations, such as reorganization, and index maintenance operations during insert, update, and delete operations. MDC is primarily intended for data warehousing and large database environments, and it can also be used in online transaction processing (OLTP) environments.

Comparison of regular and MDC tables

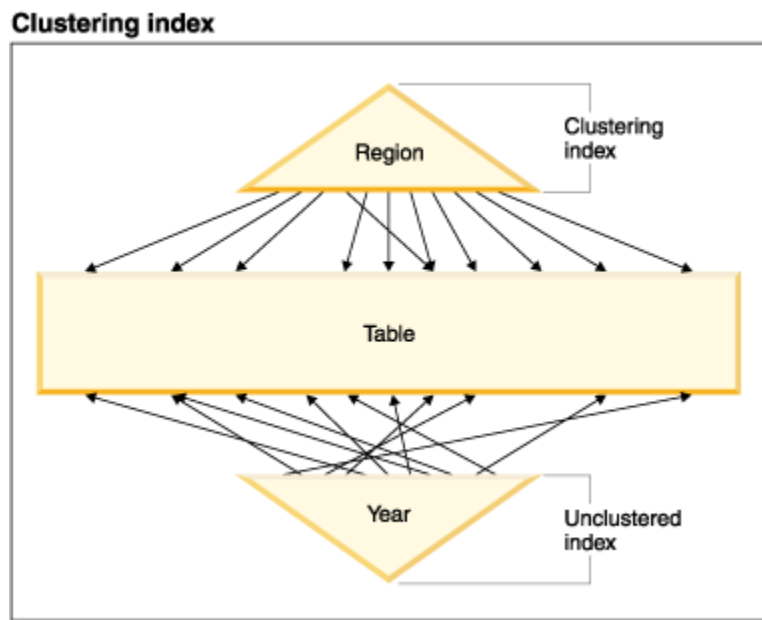
Regular tables have indexes that are record-based. Any clustering of the indexes is restricted to a single dimension. Prior to Version 8, DB2^(R) Universal Database (DB2 UDB) supported only single-dimensional clustering of data, through clustering indexes. Using a clustering index, DB2 UDB attempts to maintain the physical order of data on pages in the key order of the index when records are inserted and updated in the table. Clustering indexes greatly improve the performance of range queries that have predicates containing the key (or keys) of the clustering index. Performance is improved with a good clustering index because only a portion of the table needs to be accessed, and more efficient prefetching can be performed.

Data clustering using a clustering index has some drawbacks, however. First, because space is filled up on data pages over time, clustering is not guaranteed. An insert operation will attempt to add a record to a page nearby to those having the same or similar clustering key values, but if no space can be found in the ideal location, it will be inserted elsewhere in the table. Therefore, periodic table reorganizations may be necessary to re-cluster the table and to setup pages with additional free space to accommodate future clustered insert requests.

Second, only one index can be designated as the "clustering" index, and all other indexes will be unclustered, because the data can only be physically clustered along one dimension. This limitation is related to the fact that the clustering index is record-based, as all indexes have been prior to Version 8.1.

Third, because record-based indexes contain a pointer for every single record in the table, they can be very large in size.

Figure 26. A regular table with a clustering index



The table in Figure 26 has two record-based indexes defined on it:

- A clustering index on "Region"
- Another index on "Year"
-

The "Region" index is a clustering index which means that as keys are scanned in the index, the corresponding records should be found for the most part on the same or neighboring pages in the table. In contrast, the "Year" index is unclustered which means that as keys are scanned in that index, the corresponding records will likely be found on random pages throughout the table. Scans on the clustering index will exhibit better I/O performance and will benefit more from sequential prefetching, the more clustered the data is to that index.

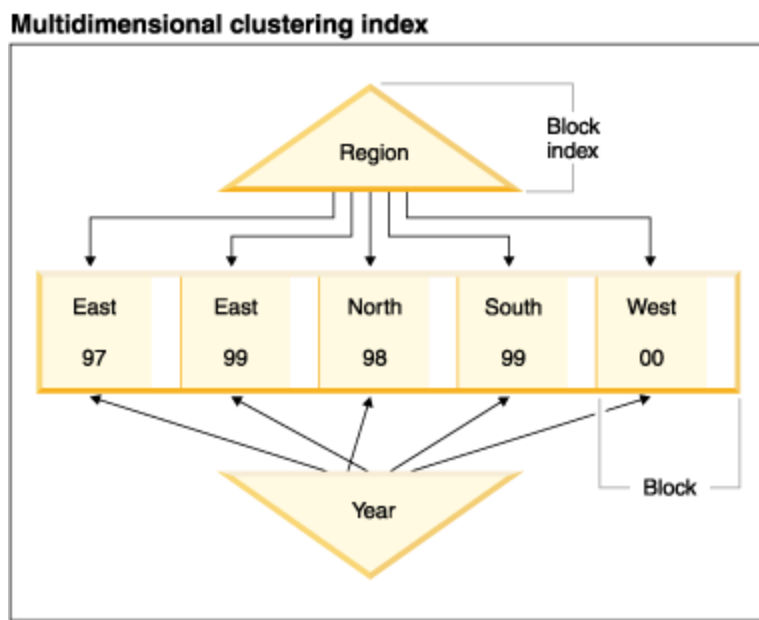
MDC introduces indexes that are block-based. "Block indexes" point to blocks or groups of records instead of to individual records. By physically organizing data in an MDC table into blocks according to clustering values, and then accessing these blocks using block indexes, MDC is able not only to address all of the drawbacks of clustering indexes, but to provide significant additional performance benefits.

First, MDC enables a table to be physically clustered on more than one key, or dimension, simultaneously. With MDC, the benefits of single-dimensional clustering are therefore extended to multiple dimensions, or clustering keys. Query performance is improved where there is clustering of one or more specified dimensions of a table. Not only will these queries access only those pages having records with the correct dimension values, these qualifying pages will be grouped into blocks, or extents.

Second, although a table with a clustering index can become unclustered over time, an MDC table is able to maintain and guarantee its clustering over all dimensions automatically and continuously. This eliminates the need to reorganize MDC tables to restore the physical order of the data.

Third, in MDC the clustering indexes are block-based. These indexes are drastically smaller than regular record-based indexes, so take up much less disk space and are faster to scan.

Figure 27. A multidimensional clustering table



Block indexes

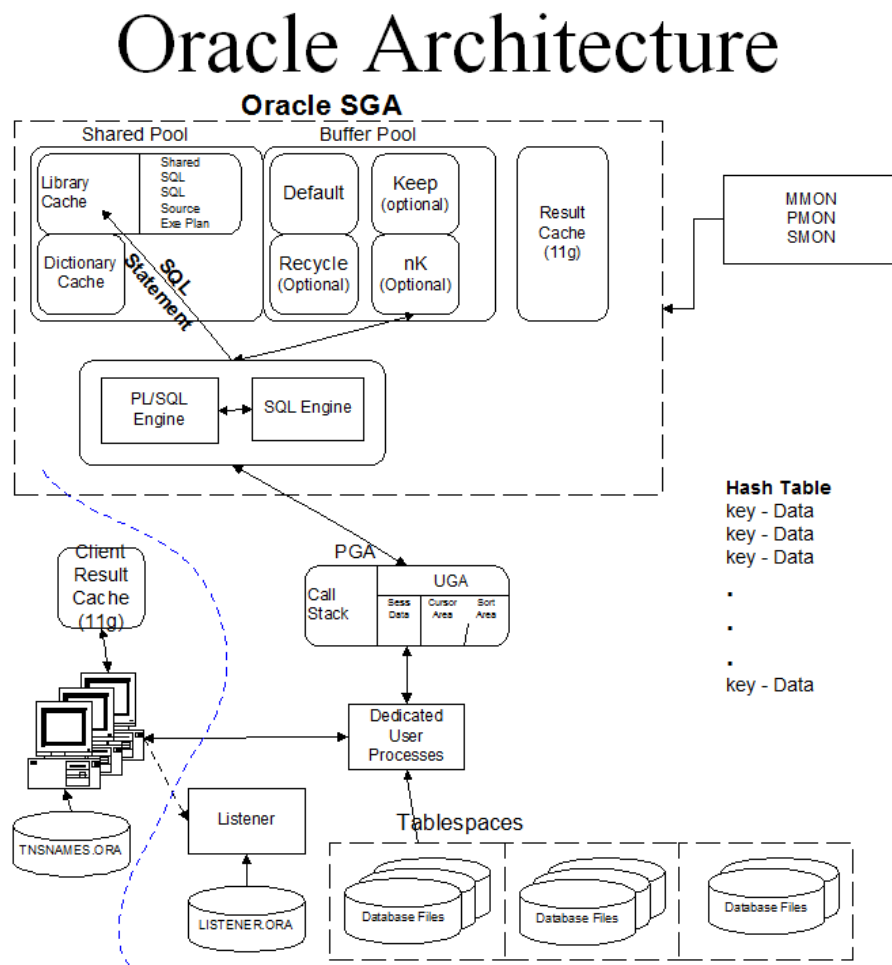
The physically organized such that records having the same "Region" and "Year" values are grouped together into separate blocks, or extents. An extent is a set of contiguous pages on disk, so these groups of records are clustered on physically contiguous data pages. Each table page belongs to exactly one block, and all blocks are of equal size (that is, an equal number of pages). The size of a block is equal to the extent size of the table space, so that block boundaries line up with extent boundaries. In this case, two block indexes are created, one for the "Region" dimension, and another for the "Year" dimension. These block indexes contain pointers only to the blocks in the table. A scan of the "Region" block index for all records having "Region" equal to "East" will find two blocks that qualify. All records, and only those records, having "Region" equal to "East" will be found in these two blocks, and will be clustered on those two sets of contiguous pages or extents. At the same time, and completely independently, a scan of the "Year" index for records between 1999 and 2000 will find three blocks that qualify. A data scan of each of these three blocks will return all records and only those records that are between 1999 and 2000, and will find these records clustered on the sequential pages within each of the blocks.

In addition to these clustering improvements, MDC tables provide the following benefits:

- Probes and scans of block indexes are much faster due to their incredibly small size in relation to record-based indexes
- Block indexes and the corresponding organization of data allows for fine-grained "partition elimination", or selective table access
- Queries that utilize the block indexes benefit from the reduced index size, optimized prefetching of blocks, and guaranteed clustering of the corresponding data
- Reduced locking and predicate evaluation is possible for some queries
- Block indexes have much less overhead associated with them for logging and maintenance because they only need to be updated when adding the first record to a block, or removing the last record from a block
- Data rolled in can reuse the contiguous space left by data previously rolled out.

10. Explain the various background process of Oracle in detail. (April 2015)

Oracle works is fundamental to tuning SQL statements. This provides an overview of important concepts in Oracle architecture and how they can impact SQL performance tuning. All users have to connect or login to Oracle. The listener is waiting on a predetermined port and 'listens' for traffic from the network. When a request is made to login, the listener process verifies the login credentials and if they pass, a dedicated user process (or if MTS...is assigned to the pool of available shared user processes) is started and a Program Global Area (PGA) is established. This PGA contains an area for any sorts the user will request as well as a cursor area for each SQL submitted. This cursor area is where the rows/result set from Oracle will be held while negotiating with the application on how to return the rows/result set.



The SQL is then passed to the library cache. Oracle performs a check sum on the SQL and arrives at a hash value from this check sum. This hash value is used to access a predetermined location in the

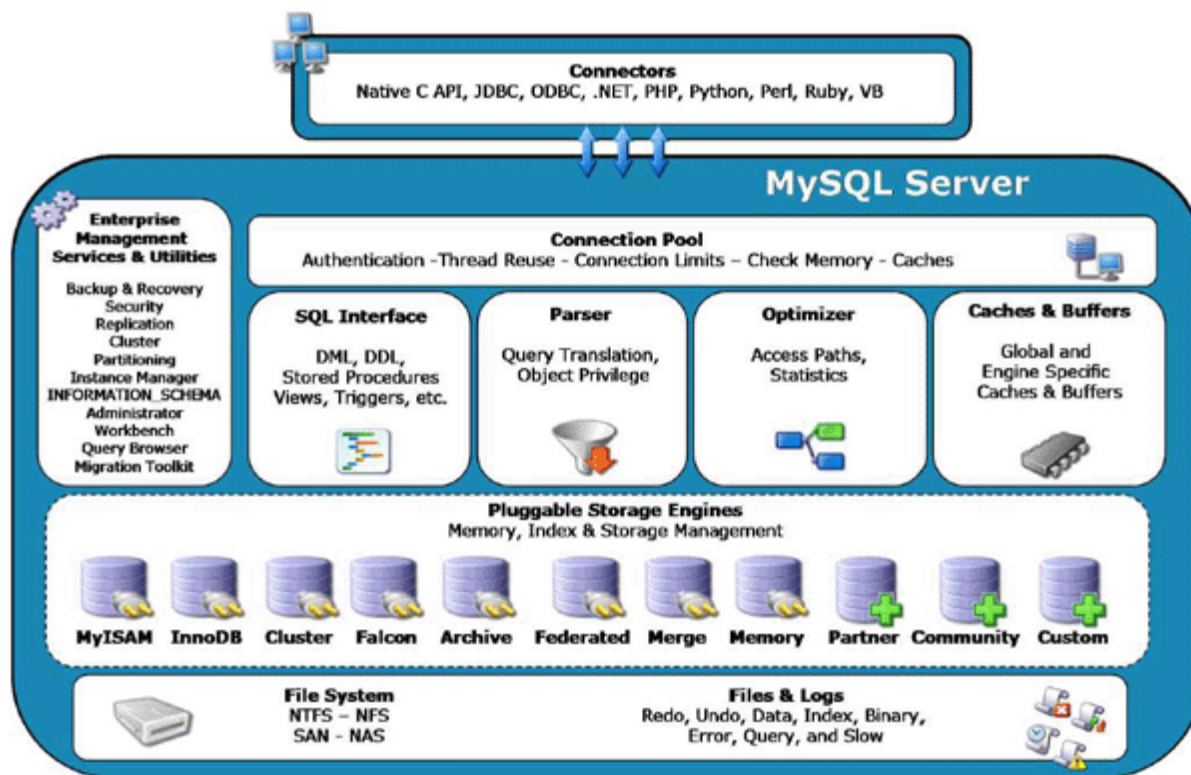
library cache. Oracle checks to see if the SQL in that slot (if there is SQL in the slot) matches the SQL being submitted.

If it is a match, then the existing explain plan is used. This is known as a 'Soft Parse.'

11. With a neat diagram discuss about architecture of MySQL in detail. (April 2015)

MySQL Server is a relational database management system developed by Microsoft. As a database, it is a software product whose primary function is to store and retrieve data as requested by other software applications, be it those on the same computer or those running on another computer across a network (including the Internet). There are at least a dozen different editions of Microsoft SQL Server aimed at different audiences and for workloads ranging from small single-machine applications to large Internet-facing applications with many concurrent users. Its primary query languages are T-SQL and ANSI SQL.

Architecture (6 Marks)



The protocol layer implements the external interface to SQL Server. All operations that can be invoked on SQL Server are communicated to it via a Microsoft-defined format, called Tabular Data Stream (TDS). TDS is an application layer protocol, used to transfer data between a database server and a client. Initially designed and developed by Sybase Inc. for their Sybase SQL Server relational database engine in 1984, and later by Microsoft in Microsoft SQL Server, TDS packets can be encased in other physical transport dependent protocols, including TCP/IP, Named pipes, and Shared memory. Consequently,

access to SQL Server is available over these protocols. In addition, the SQL Server API is also exposed over web services.

12. Explain typically available storages media in detail

(Apr 2011)

Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
 - data loss on power failure or system crash
 - physical failure of the storage device
- Can differentiate storage into:
 - **volatile storage:** loses contents when power is switched off

Nonvolatile storage:

- ☐ Contents persist even when power is switched off.
- ☐ Includes secondary and tertiary storage, as well as
 - Battery backed up main memory

Physical Storage Media

Cache

Fastest and most costly form of storage; volatile; managed by the computer system hardware

(Note: “Cache” is pronounced as “cash”)

Main memory

- Fast access (10s to 100s of nanoseconds; 1 nanosecond = 10⁻⁹ seconds)
- Generally too small (or too expensive) to store the entire database
 - Capacities of up to a few Gigabytes widely used currently
 - Capacities have gone up and per byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
- **Volatile** — contents of main memory are usually lost if a power Failure or system crash occurs.

Flash memory

- ▶ Data survives power failure
- ▶ Data can be written at a location only once, but location can be erased and written to again
 - ▶ Can support only a limited number (10K – 1M) of write/erase cycles.
 - ▶ Erasing of memory has to be done to an entire bank of memory
- ▶ Reads are roughly as fast as main memory
- ▶ But writes are slow (few microseconds), erase is slower
- ▶ NOR Flash
 - ▶ Fast reads, very slow erase, lower capacity
 - ▶ Used to store program code in many embedded devices
- ▶ NAND Flash
 - ▶ Page-at-a-time read/write, multi-page erase
 - ▶ High capacity (several GB)
 - ▶ Widely used as data storage mechanism in portable devices

Magnetic-disk

- ▶ Data is stored on spinning disk, and read/written magnetically
- ▶ Primary medium for the long-term storage of data; typically stores entire database.
- ▶ Data must be moved from disk to main memory for access, and written back for storage
- ▶ **direct-access** – possible to read data on disk in any order, unlike magnetic tape
- ▶ Survives power failures and system crashes
 - disk failure can destroy data: is rare but does happen

Optical storage

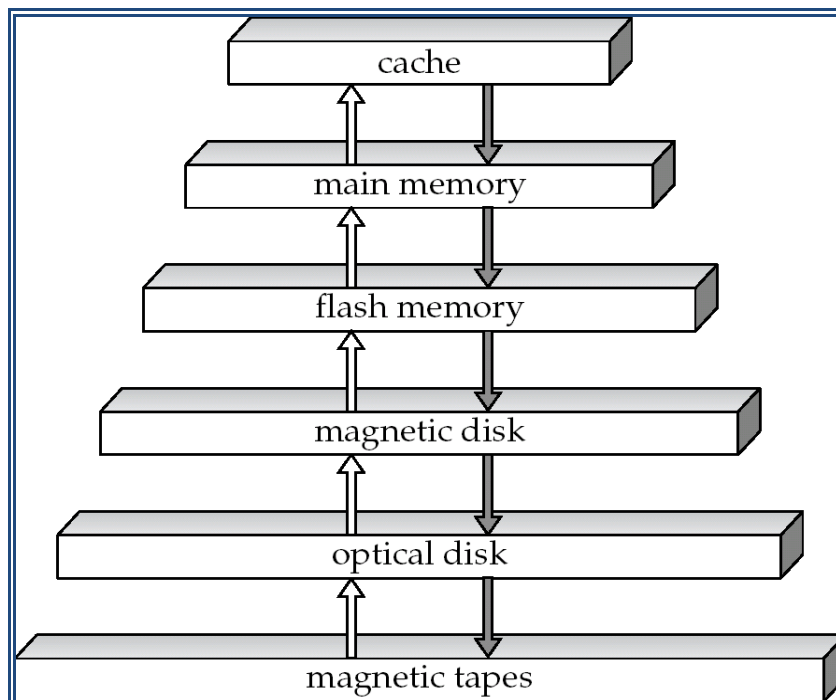
- non-volatile, data is read optically from a spinning disk using a laser
- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
- Reads and writes are slower than with magnetic disk

- **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

Tape storage

- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** – much slower than disk
- very high capacity (40 to 300 GB tapes available)
- tape can be removed from drive \Rightarrow storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
 - hundreds of terabytes (1 terabyte = 10^9 bytes) to even a petabyte (1 petabyte = 10^{12} bytes)

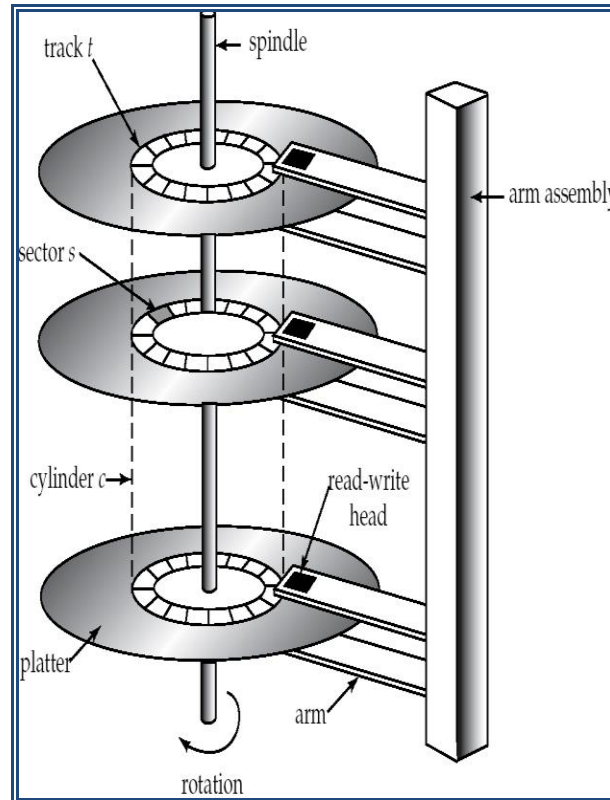
Storage Hierarchy



- primary storage: Fastest media but volatile (cache, main memory).
- secondary storage: next level in hierarchy, non-volatile, moderately fast access time

- ✓ also called on-line storage
- ✓ E.g. flash memory, magnetic disks
- tertiary storage: lowest level in hierarchy, non-volatile, slow access time
 - ✓ also called off-line storage
 - ✓ E.g. magnetic tape, optical storage

Magnetic Disk



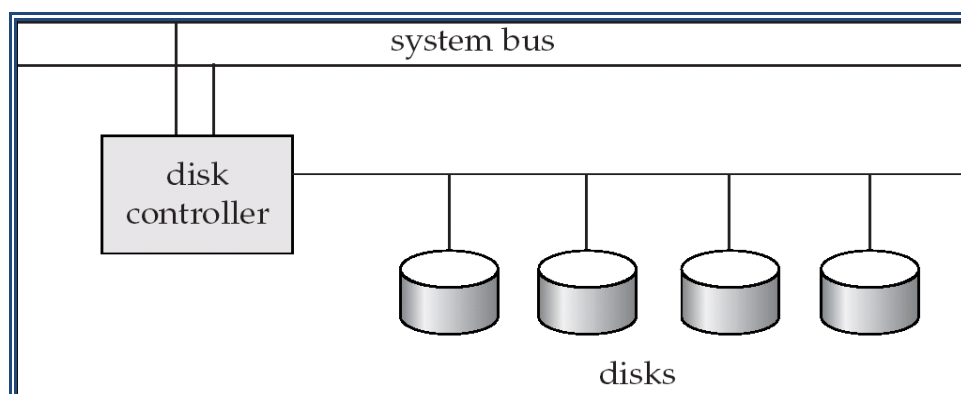
- **Read-write head**
 - ✓ Positioned very close to the platter surface (almost touching it)
 - ✓ Reads or writes magnetically encoded information.
 - ✓ Surface of platter divided into circular **tracks**
 - ✓ Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**.
 - ✓ Sector size typically 512 bytes
 - ✓ Typical sectors per track: 500 (on inner tracks) to 1000 (on outer tracks)
- To read/write a sector
 - ✓ disk arm swings to position head on right track
 - ✓ platter spins continually; data is read/written as sector passes under head

- Head-disk assemblies
 - ✓ multiple disk platters on a single spindle (1 to 5 usually)
 - ✓ One head per platter, mounted on a common arm.
- **Cylinder** consists of i^{th} track of all the platters
- Earlier generation disks were susceptible to “head-crashes” leading to loss of all data on disk
 - ✓ Current generation disks are less susceptible to such disastrous failures, but individual sectors may get corrupted

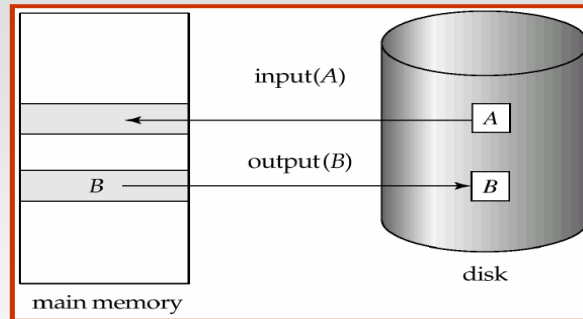
Disk controller – interfaces between the computer system and the disk drive hardware.

- accepts high-level commands to read or write a sector
- initiates actions such as moving the disk arm to the right track and actually reading or writing the data
- Computes and attaches **checksums** to each sector to verify that data is read back correctly
 - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
- Ensures successful writing by reading back sector after writing it
- Performs remapping of bad sectors

Disk Subsystem



Block Storage Operations



13. Explain storage access in detail

(APR 2010)

Storage Access

A database file is partitioned into fixed length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.

Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.

Buffer – portion of main memory available to store copies of disk blocks.

Buffer manager – subsystem responsible for allocating buffer space in main memory.

Buffer Manager

Programs call on the buffer manager when they need a block from disk. Buffer manager does the following:

- If the block is already in the buffer, return the address of the block in main memory
 1. . If the block is not in the buffer
 - . Allocate space in the buffer for the block
 - . Replacing (throwing out) some other block, if required, to make space for the new block.
- Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
- Read the block from the disk to the buffer, and return the address of the block in main memory to requester.

UNIVERSITY QUESTIONS

1. Explain deadlock handling in detail **(Nov 2012)(Question No. 8)**
2. Explain about Microsoft SQL server in detail **(April 2012) (Nov 2014) (Question No. 5)**
3. Explain various models of locking a data item. Also explain two-phase locking protocol **(Question No. 7)**
4. Explain the various background process of Oracle in detail. **(April 2015) (Question No. 9)**
5. With a neat diagram discuss about architecture of MySQL in detail. **(April 2015) (Question No. 10)**
6. Discuss about Database Recovery procedure in detail. **(April 2015)(NOV 2015) (Question No. 11)**
7. State and explain the Two-phase locking protocol with an example. **(NOV 2015) (Question No. 3)**

8. Explain about the Concurrency Control. **(NOV 2014)[Question No.: 04]**
9. Explain about the Recovery System for Database Management System. **(NOV 2014)**
[Question No.: 05]
10. Explain typically available storage media in detail **(April 2011)[Question No.: 01]**