UNIT I

Introduction - Historical perspective - Files versus database systems - Architecture - E-R model- Security and Integrity - Data models.

INTRODUCTION

Data :

- Data is stored facts
- Data may be numerical data which may be integers or floating point numbers, and non-**numerical data such as characters, date and etc.,**

Example:

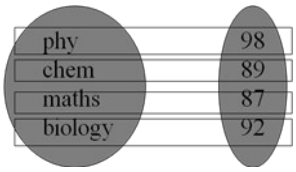
The above numbers may be anything: It may be distance in kms or amount in rupees or no of days or marks in each subject etc.,

Information :

- Information is data that have been organized and communicated in a coherent and meaningful manner.
  - Data is converted into information, and information is converted into knowledge.
  - Knowledge; information evaluated and organized so that it can be used purposefully.

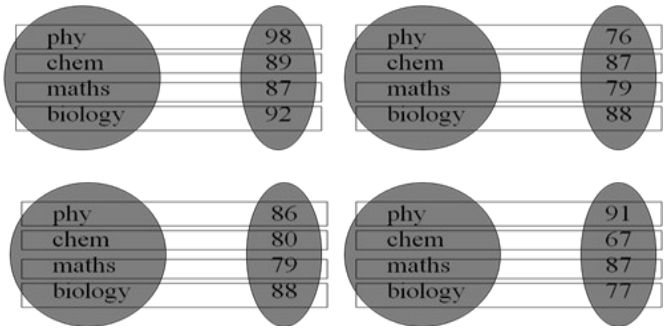
Example:

The data (information) which is used by an organization – a college, a library, a bank, a manufacturing company – is one of its most valuable resources.



Database :

- Database is a collection of information organized in such a way that a computer program can quickly select desired pieces of data.



Database Management System (DBMS) :

- A collection of programs that enables you to store, modify, and extract information from a database.
- A collection of interrelated data and a set of programs to access those data.
- The primary goal of a DBMS is to provide an environment that is both convenient and efficient to use in retrieving and storing database information.

HISTORY OF DATABASE SYSTEMS :

Techniques for data storage and processing have evolved over the years.

1950's and early 1960's :

- Magnetic tapes were developed for data storage.
- Processing of data consisted of reading from one or more tapes and writing data to a new tape.
- Data could be input from punched cards and output to printers.
- Tapes could be read only sequentially.

Late 1960's and 1970's :

-Hard disks were used for data storage.

-Hard disks allowed direct access to data

-Data freed from the difficult of sequentiality

- Network and hierarchical data models in widespread use
- Ted Codd defines the relational data model and non-procedural ways of querying data in the relational model.

1980's :

- Relational databases were so easy to use that they replaced network / hierarchical databases.
- In network / hierarchical databases, programmers had to code their queries in procedural fashion, but in relational databases all these low level tasks are carried out automatically.
- So programmers can work at logical level.

- In 1980's research on parallel and distributed databases and also on object oriented databases was started.

Early 1990's :

- The SQL language was designed for decision support applications which are query intensive.
- Tools for analyzing large amounts of data saw large growths in usage.
- Parallel database products were introduced.
- Database vendors also began to ass object-relational support to their databases.

Late 1990's :

- The major event was the explosive growth of the World Wide Web.
- DB systems now had to support very high transaction processing rates, reliability and availability (24 X 7).
- DB systems also had to support web interfaces to data.

Early 2000's :

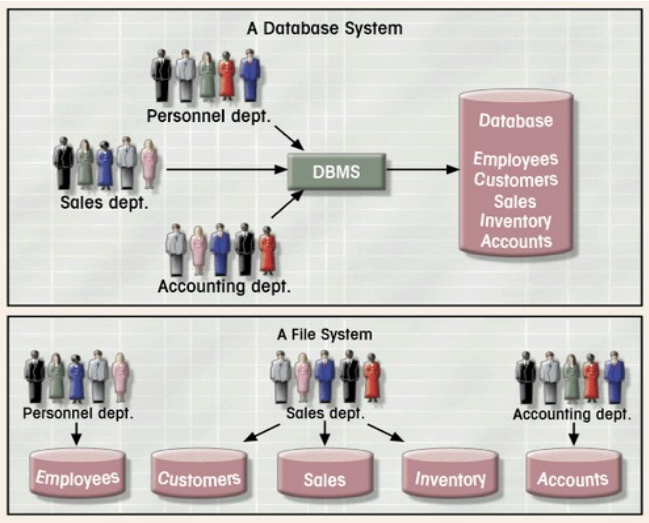
- Techniques such as Emerging of XML and XQuery and Automated database administration introduced.

DATABASE SYSTEM VS FILE SYSTEM :

The information can be either a conventional file processing system or a database system. In the conventional file processing system each and every subsystem of the information system will have its own set of files. As a result, there will be a duplication of data between various subsystems of the information system. But in database systems, there is a single centralized database which minimizes the redundancy of data to a greater extent.

Drawbacks of Conventional File Processing System :

- Data redundancy & inconsistency
- Difficulty in accessing data
- Data Isolation
- Concurrent access anomalies
- Security Problems
- Integrity Problems



I . Data redundancy & inconsistency:

The files have different formats and the programs may be written in several programming languages. The same information may be duplicated in several places. This redundancy leads to higher storage and access cost. So it may lead to data inconsistency. For example: The address of a customer may appear in savings account file and personal information file. A changed customer address may be reflected in personal information file but not in savings account records file.

II . Difficulty in accessing data :

Conventional file processing environments do not allow needed data to be retrieved in a convenient and efficient manner. Suppose the bank officer needs to find out the names of all the customers who live within the city's 78733 Zip code. The bank officer has now two choices. Either get the list of all customers and extract the needed information manually or ask the data processing dept to have a system programmer to write the necessary application program. Both alternatives are unsatisfactory.

III . Data Isolation :

Data is scattered in various files and files may be in different format, it is difficult to write new application programs to retrieve appropriate data.

IV . Integrity Problems :

The data values stored in the database must satisfy certain types of consistency constraints. These constraints are enforced in the system by adding appropriate code in the various application programs. It is hard to add new constraints or change existing ones. For Example, the balance of a bank account may change from $25 to $50.

V. Atomicity of updates

Failures may leave database in an inconsistent state with partial updates carried out. Example: Consider a program to transfer $50 from account A to B. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account A but was not credited to account B, resulting in an inconsistent database state. Transfer

of funds from one account to another should be atomic – it must happen in its entirety or not at all.

VI. Concurrent access anomalies :

In order to improve the overall performance of the system and obtain a faster response time. Many systems allow multiple users to update the data simultaneously. In such environment interaction of concurrent updates may result in inconsistent data.

Example : Consider bank account A containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account A at about the same time, they may both read the value $500, and write back $450 and $400 respectively. But the correct value is $350.

VII. Security Problems :

Not every user of the database system should be able to access all the data. It is Hard to provide user access to some, but not all, data. Example : In a banking system, payroll personnel need to see only the information about the various employees. They do not need access information about customer accounts.

Advantages of Database :

Database is a way to consolidate and control the operational data centrally. The advantages of having a centralized control of data are

- Redundancy can be reduced.
- Inconsistency can be avoided.
- The data can be shared.
- The standards can be enforced
- Security can be enforced.
- Integrity can be enforced.

VIEW OF DATA :

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these files major purpose 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 and retrieved efficiently.

Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency is to use complex data structures to represent data in the database. The developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system. The design of complex data structures for the representation of data in the database are

- Physical Level
- Conceptual Level

- View Level

i) **Physical Level:** The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.

(ii) Logical / Conceptual Level: The next higher level of data abstraction describes what data are actually stored in the database and the relationship that exists among the data. Here the entire database is described in terms of a smaller no of relatively simple structures. The user of this level need not know the complexities of the physical level.

(iii) View Level / External Level: The highest level of abstraction describes only part of the entire database. When we use simple structures at the conceptual level, some complexities remain because of the large size of the database. To simplify the interaction with the system, the view level of abstraction is defined. The system may provide many views of the same database.

Levels of Abstraction in a DBMS

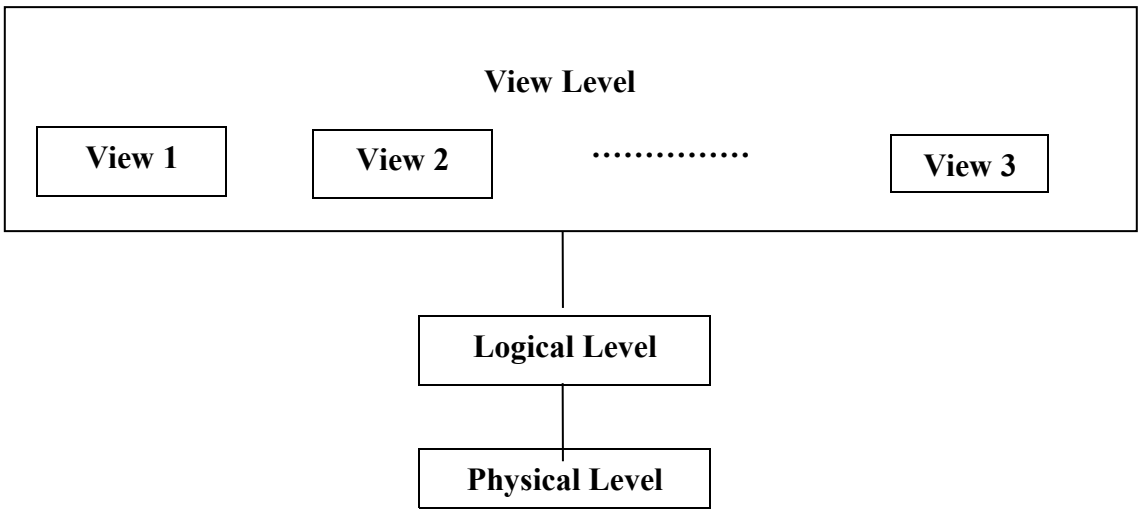
1.Conceptual Schema / Logical Schema:

It describes all relations that are stored in the database. For University, a conceptual schema is:

– Students(sid:string,Age:integer)

– Faculty (fid: string, salary: real)

– Courses (cid: string, cname: string, credits:integer)



The three levels of data abstraction

In the university example, these relations contain information about ***entities***, such as students and faculty, and about ***relationships***, such as students' enrollment in courses.

2. Physical Schema :

- It specifies additional storage details.

- It summarizes how the relations described in the conceptual schema are stored on secondary storage devices such as disks and tapes.

- Creation of data structures called ***indexes***, to speed up data retrieval operations.

3. External Schema :

- Each external schema consists of a collection of one or more *views* and relations from the conceptual schema. A *view* is conceptually a relation, but the records in a view are not stored in the DBMS.A *user* creates any view from data already stored.

  - For example: we might want to allow students to find out the names of faculty members teaching courses.

This is the view associated:

Courseinfo (cid:string, fname:string)

  – A user can treat a view just like a relation and ask questions about the records in the view, even though the records in the view are not *stored explicitly*.

Instances and Schemas

Databases change over time as information is inserted and deleted.

Schema : The logical structure of the database is called the database schema.

Analogous to type information of a variable in a program.

Types of Schema

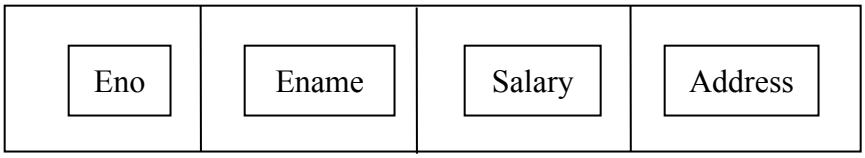
Physical schema: It describes the database design at the physical level.

Logical schema : It describes the database design at the logical level.

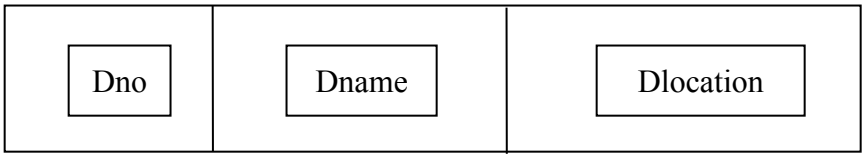
Subschema : A database may also have several schemas at the view level, sometimes called sub schemas, that describes different views of the database.

DB Schema Diagram for a Company:

Employee:



Department :



Instance : The collection of information stored in the database at a particular moment is called an instance of the database. Analogous to the value of a variable.

Instance Ex:

Eno	Ename	Salary	Address
1	A	10,000	First street
2	B	20,000	Second street
3	C	30,000	Third street

Data Independence:

The ability to modify a schema definition in one level without affecting a schema definition in the next higher level is called data independence.

There are the levels of data independence:

1. **Logical data independence:** The ability to change the logical (conceptual) schema without changing the External schema (User View) is called logical data independence. For example, the addition or removal of new entities, attributes, or relationships to the conceptual schema should be possible without having to change existing external schemas or having to rewrite existing application programs.
2. **Physical data independence:** The ability to change the physical schema without changing the logical schema is called physical data independence. For example, a change to the internal schema, such as using different file organization or storage structures, storage devices, or indexing strategy, should be possible without having to change the conceptual or external schemas.
3. **View level data independence:** always independent no effect, because there doesn't exist any other level above view level.

DATA MODEL

- A collection of conceptual tools for describing data, data relationships, data semantics & consistency constraints.

- A data model provides a way to describe the design of a database at the physical, logical and view level.

Relational Model:

- It 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.
- Each table contains records of a particular type.
- Each record type defines a fixed no. of fields, or attributes.

- It is the most widely used data model.

Table name: AGENT

AGENT_CODE	AGENT_LNAME	AGENT_FNAME	AGENT_INITIAL	AGENT_AREACODE	AGENT_PHONE
501	Alby	Alex	B	713	228-1249
502	Hahn	Leah	F	615	882-1244
503	Okon	John	T	615	123-5589

Link through AGENT code

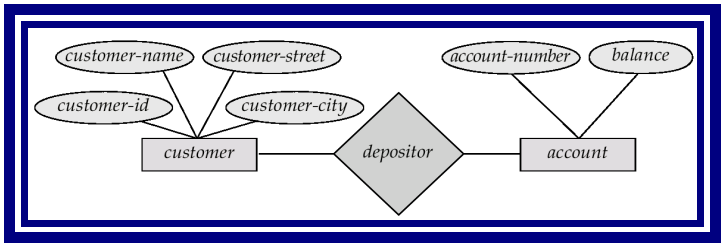
Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_RENEW_DATE	AGENT_CODE
10010	Ramas	Alfred	A	615	844-2573	05-Apr-2002	502
10011	Dunne	Leona	K	713	894-1238	16-Jun-2002	501
10012	Smith	Kathy	W	615	894-2285	29-Jan-2001	502
10013	Olowski	Paul	F	615	894-2180	14-Oct-2002	502
10014	Orlando	Myron		615	222-1672	28-Dec-2002	501
10015	O'Brian	Amy	B	713	442-3381	22-Sep-2002	503
10016	Brown	James	G	615	297-1228	25-Mar-2002	502
10017	Williams	George		615	290-2556	17-Jul-2002	503
10018	Farriss	Anne	G	713	382-7185	03-Dec-2002	501
10019	Smith	Olette	K	615	297-3809	14-Mar-2002	503

Relational Database Model

Entity Relationship Model

- It 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.



Example of schema in the entity-relationship model

customer-id	customer-name	customer-street	customer-city
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

customer-id	account-number
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

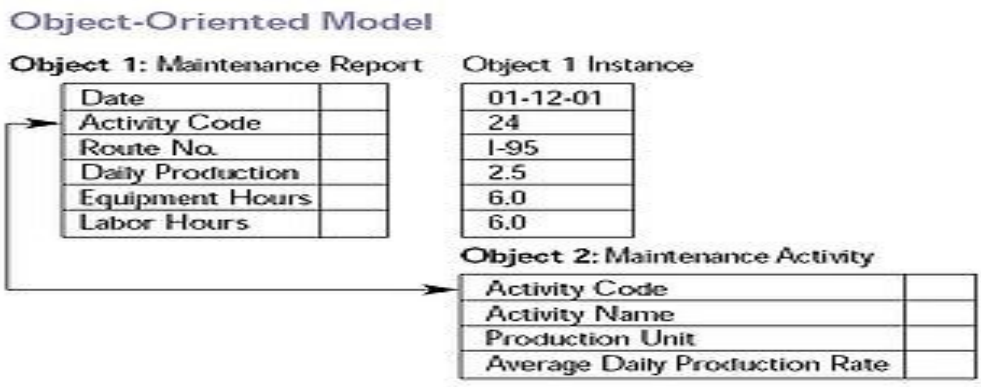
(c) The *depositor* table

account-number	balance
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

Object Oriented Data Model

- An object-oriented database is a database model in which information is represented in the form of objects as used in object-oriented programming.

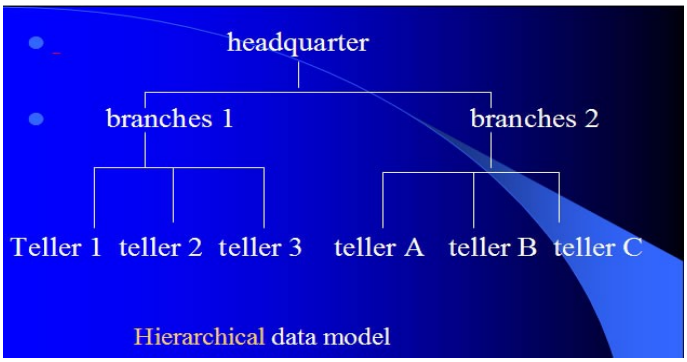


Object Relational data Model:

- It combines the features of the object oriented data model and relational data model.

Hierarchical data model:

- A **hierarchical data model** is a data model in which the data is organized into a tree-like structure.
- Logically represented by an upside down tree
- Each parent can have many children
- Each child has only one parent



Network data model

A network is a type of mathematical graph that captures relationships between objects using connectivity.

- The following are some key terms related to the network data model:
- A **node** represents an object of interest.
- A **link** represents a relationship between two nodes. A link may be **directed** or **undirected**
- A **path** is an alternating sequence of nodes and links, beginning and ending with nodes, and usually with no nodes and links appearing more than once.

- A network consists of a set of nodes and links. Each link (sometimes also called an edge or a segment) specifies two nodes.

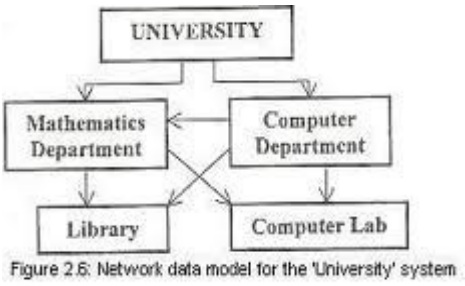


Figure 2.6: Network data model for the 'University' system

Database Languages:

A database system provides two different types of languages:

1. **Data Definition Language** (DDL) - To specify the database schema
2. **Data Manipulation Language** (DML) - To express the queries and updates.

Data Definition Language:

- Used by the DBA and database designers to specify the conceptual schema of a database.
- In some DBMSs, separate storage definition language (SDL) and view definition language (VDL) are used to define internal and external schemas.
- SDL is typically realized via DBMS commands provided to the DBA and database designers.
- DDL compiler generates a set of tables stored in a data dictionary.
- A **data dictionary** is a file that contains metadata (ie. Data about data)

Data Manipulation Lnaguage :

DML is a language that enables users to access or manipulate as organized as the appropriate data model. The types of accesses are :

- Retrieval of information stored in the database
- Insertion of new item into the data base.
- Deletion of information from the database.
- Modification of information stored in the database.

The DML languages are basically two types. They are

(i) Procedural DML'S

(ii) Non Procedural DML'S / Declarative DML'S

Procedural DML'S :

The procedural DML'S require a user to specify what data are needed and how to get those data.

Non Procedural DML'S :

Declarative DML'S require a user to specify what data are needed without specifying how to get those data. Declarative DML'S are usually easier to learn and use than are procedural DML's.

A **Query** is a statement requesting the retrieval of information. The portion of a DML that involve information retrieval is called **Query Language**.

TRANSACTION MANAGEMENT

Properties :

Atomicity -- all-or-none requirement

Consistency -- Correctness

Durability -- persistence requirement

- 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.

- If the db was consistent when a transaction started, the db must be consistent when the transaction terminates.

- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

- The db system must perform **failure recovery** (ie) detect system failures and restore the db to the state that existed prior to the occurrence of the failure.

- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Database Users and Administrators

- People who work with database can be categorized as,
- Database users and
- Database administrators

Database Users:

There are four different types of users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

Naive Users:

Naive users are those users who do not have any technical knowledge about the DBMS. They use the database through application programs by using simple user interface. They perform all operations by using simple commands provided in the user interface.

Ex : View balance of an account

The typical user interface for naïve users is a forms interface, where the user can fill in appropriate fields of the form. They also read reports.

Application Programmers :

- They are computer professionals who write application programs.
- They can choose from many tools to develop user interfaces.
- Rapid Application Development tools are tools used to construct forms and reports with minimal efforts.

Sophisticated Users:

Sophisticated users are the users who are familiar with the structure of database and facilities of DBMS. Such users can use a query language such as SQL to perform the required operations on databases.

Specialized Users:

They write specialized database applications that do not fit into the traditional data-processing framework.

Database Administrator :

A Person who has the central control over the system (ie) control of both data and programs to access those data, is called DBA.

Functions of DBA include:

Schema Definition:

DBA creates the original db schema by executing DDL statements.

Schema and Physical organization modification:

The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization.

Granting of authorization for data access:

Providing authorization for different types of users. So he can regulate the user access.

Routine maintenance :

- Periodically backing up the db, either onto tapes or onto remote servers to prevent loss of data.
- Ensuring that enough free disk space is available for normal operations and upgrading disk space as required.

- Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

DATABASE SYSTEM STRUCTURE

- A Database system is partitioned into modules that deal with each of the responsibilities of the overall system.
- The functional components of a database system can be broadly divided into,
    - the storage manager and
    - query processor components.

Storage Manager

- A storage manager is a program module that provides the interface between the low level data stored in the database and the application programs and queries submitted to the system.
- The storage manager translates the various DML statements into low level file system commands.
- The storage manager is responsible for storing, retrieving and updating data in the database.

The storage manager components include:

Authorization and integrity manager

- Which tests for the satisfaction of integrity constraints and checks the authority of users to access data.

Transaction manager

- Which ensures that the database remains in a consistent correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.

File Manager

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

Buffer manager

- Which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory.

The storage manager implements several data structures as part of the physical system implementation:

- **Data files,** which store the database itself.

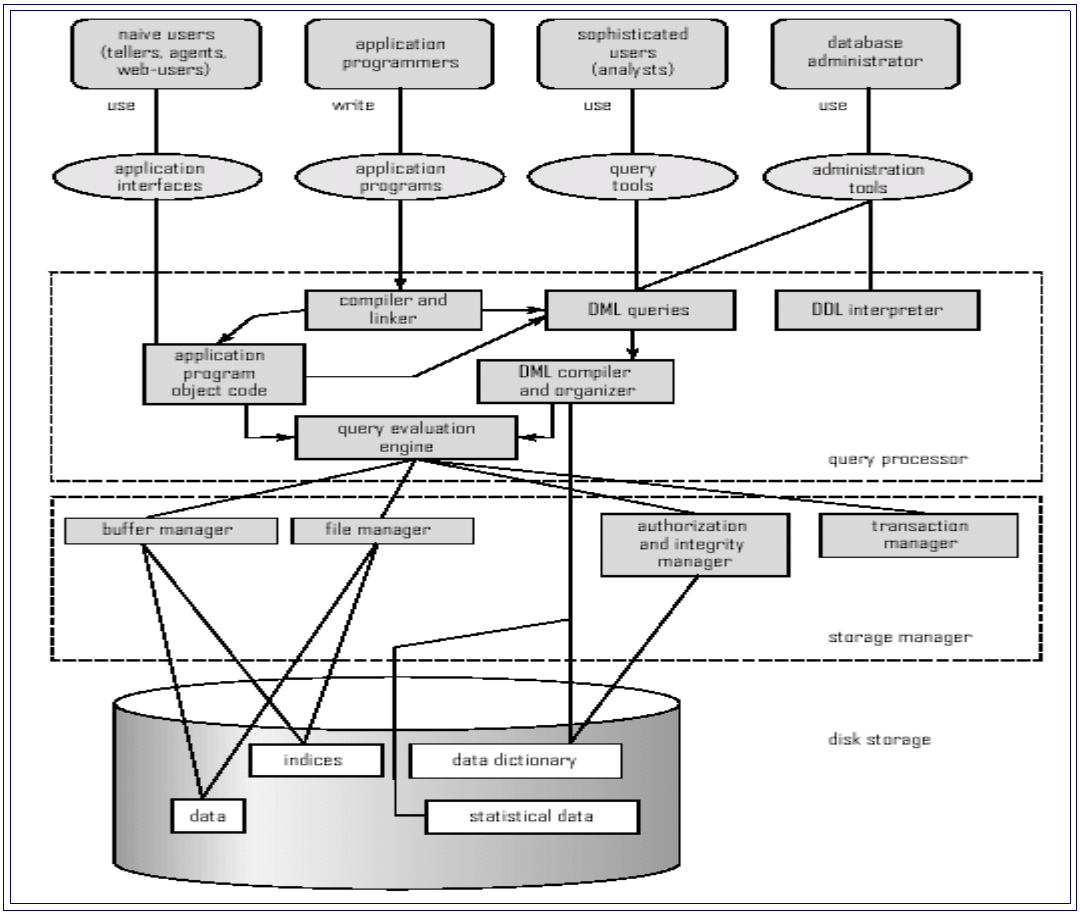
• **Data dictionary,** which stores metadata (ie) data about data in particular the schema of the database.

- names of the tables, names of attributes of each table, length of attributes, and number of rows in each table.
- Relationships
- Constraints
- Access Authorization

Indices:

It provide fast access to data items holding particular values.

Overall System Structure



The Query Processor

The query processor components include

DDL interpreter, which interprets DDL statements and records the definitions in the data dictionary.

DML Compiler and Query optimizer - The DML commands such as insert, update, delete, retrieve from the application program are sent to the DML compiler for compilation into object code for database access. The object code is then optimized in the best way to execute a query by the query optimizer and then send to the data manager.

Query evaluation engine - which executes low level instructions generated by the DML compiler.

ENTITY-RELATIONSHIP DATA MODEL

It is a high level conceptual data model that describes the structure of db in terms of entities, relationship among entities & constraints on them..

Basic Concepts of E-R Model:

- Entity
- Entity Set
- Attributes
- Relationship
- Relationship set
- Identifying Relationship

Entity

An Entity is a thing or object in the real world that is distinguishable from all other objects.

Ex : a Person , a book.

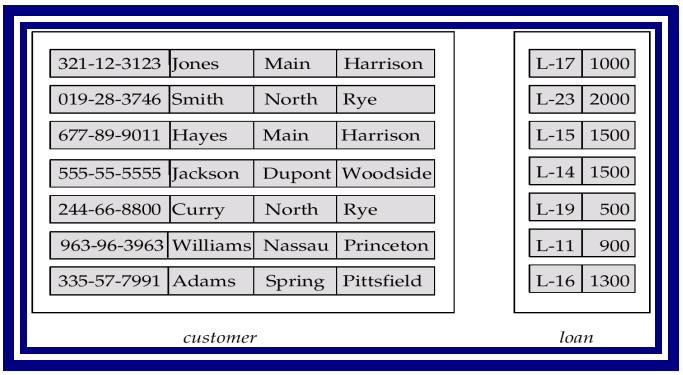
- An entity has a set of properties and the values for some set of properties may uniquely identify an entity.
- An entity may be concrete Ex : a person or a book
- An entity may be abstract Ex: a load or a holiday

Entity Set

- An **Entity Set** is a set of entities of the same type that share the same properties or attributes.

Ex : set of all customers in a bank.

Entity Sets *customer* and *loan*



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

customer

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

loan

Attributes:

A set of properties that describe an entity.

Ex : attributes of customer entity are, Customer id , Cust name and city.

attributes of account entity are, Account no and balance.

Each entity has a value for each of its attributes.

Ex : Customer id – C101 Cust name – 'Ram'

For each attribute there is a set of permitted values called the domain or value set of that attribute.

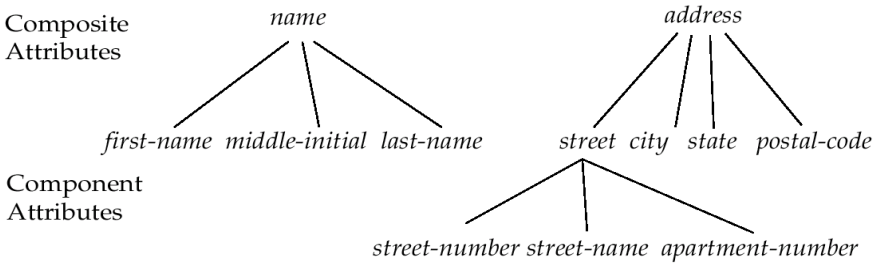
Ex : for Cust name – set of all strings of a certain length

Types of Attributes

Simple attributes : Attributes that can't be divided into subparts are called Simple or Atomic attributes.

Ex : EmployeeNumber , Age

Composite attributes : They can be divided into sub parts. Ex : Name and Address.



Single-Valued attribute : An attribute that has a single value for a particular entity.

Ex : Age , Loan number for a particular entity refers to only one loan number.

Multi-Valued attribute : An attribute has a set of values for a specific entity.

Ex : phone number of an employee . One employee can has more than one phone number and degree

Derived attribute : The value for this type of attribute can be derived from the values of other related attributes or entities.

Ex : Age of a person can be can be calculated from person's date of birth and present date.

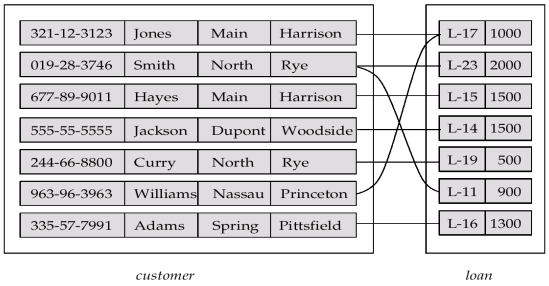
Key Attribute: An attribute which is used to uniquely identify records. Ex : eid, sno, dno

Relationship:

It is an association among several entities. It specifies what type of relationship exists between entities.

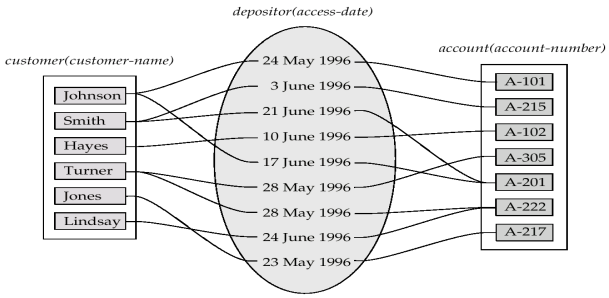
Relationship set:

It is a set of relationships of the same type.



Relationship Set *borrower*

An *attribute* can also be property of a relationship set. For instance, the *depositor* relationship set between entity sets *customer* and *account* may have the attribute *access-date.*

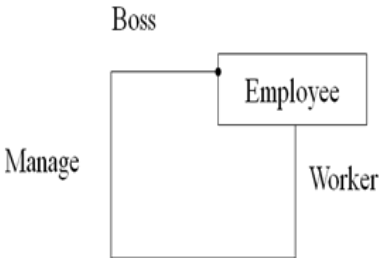


Types of relationships

i) Unary Relationship :

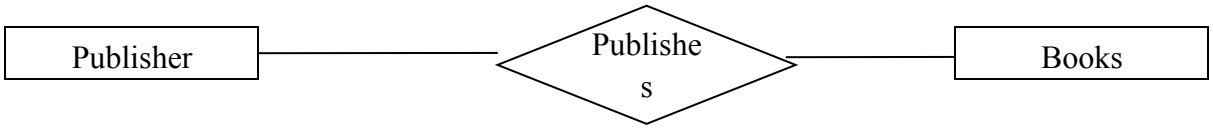
A Unary relationship exists when an association is maintained within a single entity.

Ex: Boss and worker distinguish the two employees anticipating in the manage association.



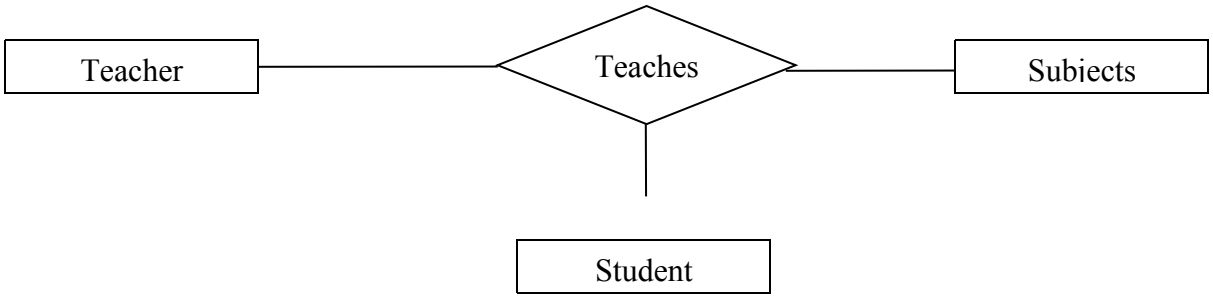
ii) Binary Relationship :

A binary relationship exists when 2 entities are involved / associated. For Example: The book – Publisher relationship is binary relationship.



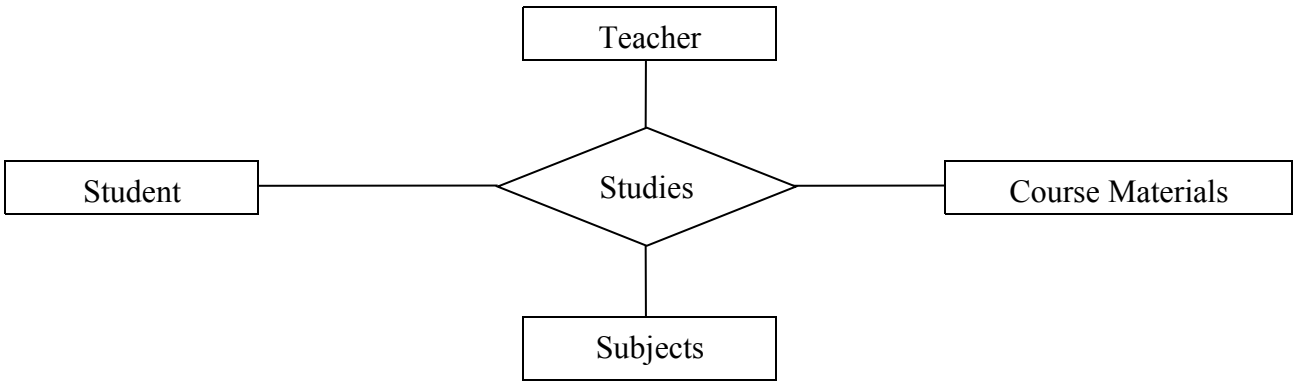
iii) Ternary Relationship :

A ternary relationship exists when 3 entities are associated. For example the entities teacher , subject, and student are related using a ternary Relationship called 'Teaches'.



iv) Quaternary Relationship:

A Quaternary relationship exists when 4 entities associated. An example of Quaternary relationship is ' Studies' where 4 entities are involved such as student , teacher , subject & course material.



Constraints:

An E-R enterprise schema may define certain constraints to which the contents of a database system must conform. There are 2 main important types of constraints. They are

- Mapping Cardinalities
- Participation Constraints

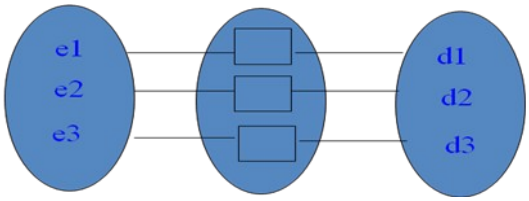
1.Mapping Cardinalities:

Mapping Cardinalities express the no of entities to which another entity can be associated via a relationship set. For a binary relationship set R between entity sets A and B, the mapping cardinalities must be one of the following:

- One to One
- One to Many
- Many to One
- Many to Many

One to One:

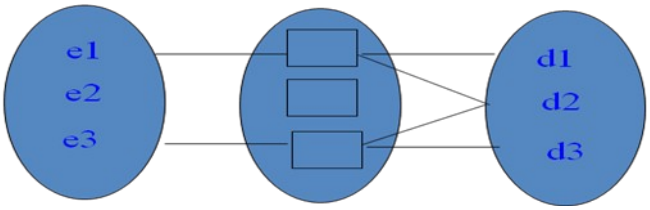
An entity in A is associated with at most one entity in B and an entity in B is associated with at most one entity in A.



Employee works for Department

One-to-many:

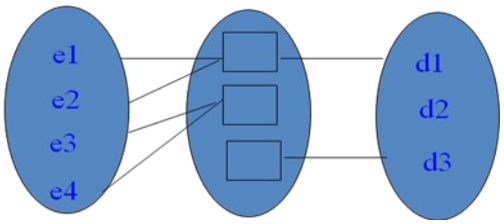
An entity in A is related to any number of entities in B, but an entity in B is related to at most one entity in A.



Employee works for Department

Many-to-One

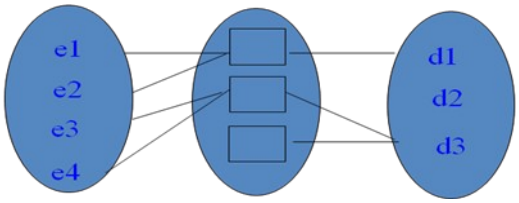
An entity in A is related to at most one entity in B, but an entity in B is related to any number of entities in A.



Employee works for Department

Many-to-Many

An entity in A is related to any number of entities in B, but an entity in B is related to any number of entities in A.



Employee works for Department

2) Participation Constraints :

i) Total Participation :

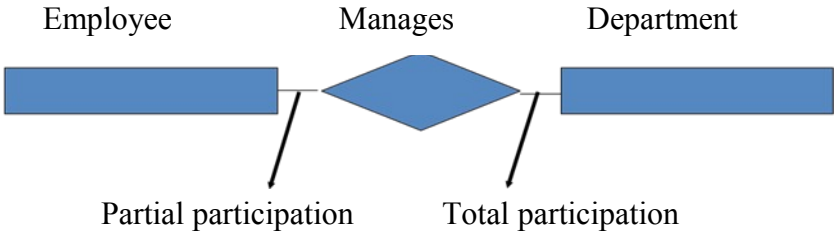
The participation of an entity set E in a relationship set R is said to be **Total** if every entity in E participates in at least one relationship in R.

Ex : 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.**

ii) Partial Participation :

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

Ex : a bank customer may or may not have a loan. Therefore, the participation of customer in the borrower relationship is **partial.**



For each department there is an Head of the Department (who is an employee again). (Dept-Head-of-emp).So now for this relation not all the employees are participating in relation Head of.So Dept-Head-of will have Total Participation and Head-of-emp will have partial participation.

Keys:

- It is used to uniquely identify entities within a given entity set or a relationship set.
- Keys in Entity set:

(i) Primary Key:

It is a key used to uniquely identify an entity in the entity set.

Ex : eno,rno,dno etc…

(ii) Super Key:

It is a set of one or more attributes that allow us to uniquely identify an entity in the entity set. Among them one must be a primary key attribute.

Ex : Eid (primary key) and ename together can be identify an entity in entity set.

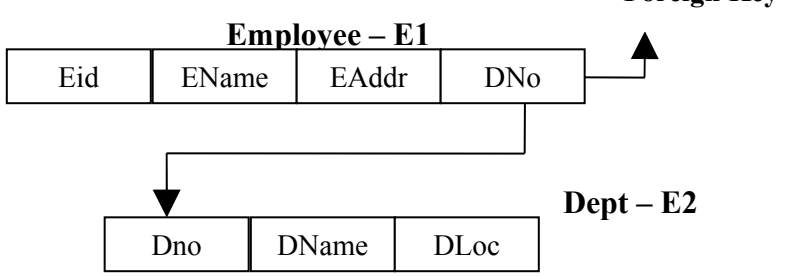
(iii) Candidate key:

They are minimal super keys for which no proper subset is a superkey.

Ex: Ename and eaddr can be sufficient to identify an employee in employee set. {eid} and {ename,eaddr} – Candidate keys

(iv) Foreign Keys

An attribute which makes a reference to an attribute of another entity type is called foreign key. Foreign keys link tables together to form an integrated da



Weak Entity Sets :

- An entity set that does not have a primary key is referred to as a weak entity set.

Strong Entity Sets :

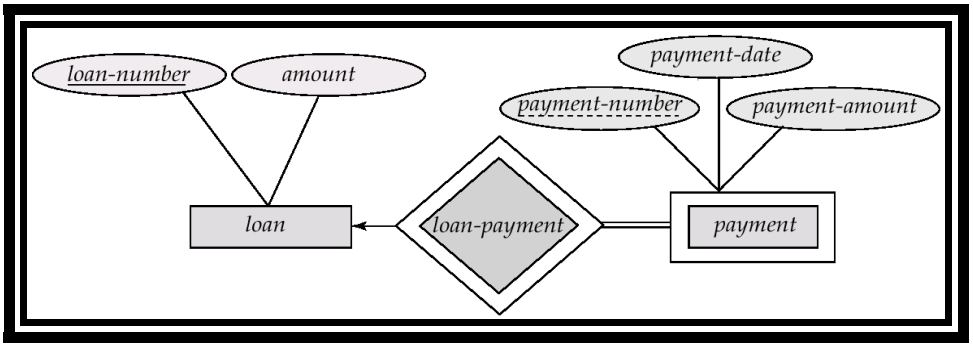
- An entity set that has a primary key is referred to as a Strong entity set.

Identifying or Owner Entity Set :

- A weak entity set must be associated with another entity set called the Identifying or Owner Entity Set.
- The existence of a weak entity set depends on the existence of a identifying entity set
- Identifying relationship depicted using a double diamond

Discriminator :

- 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)

Conceptual DB design with ER Model

- The E-R model is supported with the additional semantic concepts is called **Extended Entity Relationship Model** or **EER model.**
- The EER model includes all the concepts of the original E-R model together with the following additional concepts :
  - Specialization
  - Generalization
  - Aggregation

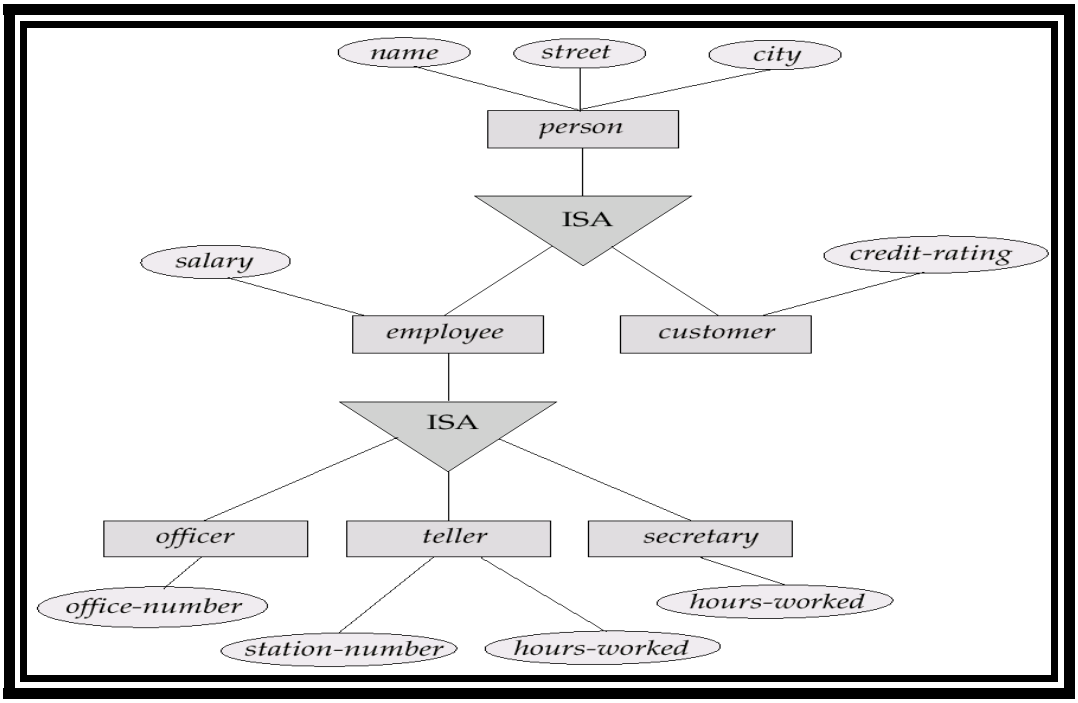
Specialization :

- "Specialization is the process of designating subgroupings within an entity set.
- This is the top – down process.
- Consider an entity set person , with attributes name , street and city. A person may be further classified as one of 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 additional attributes**.**

Generalization :

- **" Generalization** is the process of defining a more general entity type from a set of more specialized entity types".
- Generalization is a bottom – up approach.
- This approach results in the identification of a generalized superclass from the original subclasses.
- Consider that the attributes of "customer" entity are customer_id , name , street, city and an 'employee' entity attributes are employee_code , street , city and salary. Thus the entity set employee and customer have several attributes in common.
- This 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.
- **For Example: Person** is a **higher level** entity set and **customer** and **employee** are **lower level** entity sets.
- In other words, person is a **Super Class** if customer and employee are **subclasses.**
- **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.

Example



Generalization Constraints

These constraints involve determining which entities can be members of a given lower-level entity set. Such membership may be one of the following :

- Condition defined

- User defined
    - Disjoint
- Overlapping
    - Completeness Constraint
        - Total generalization / specialization
        - Partial generalization / specialization

Condition defined :

- In condition defined lower – level entity sets, membership is evaluated on the basis of whether or not an entity satisfies a condition or predicate.
    - E.g. all customers over 65 years are members of senior-citizen entity set; senior-citizen ISA person.

User defined :

- These types of constraints are defined by user.
- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.

Disjoint

- An entity can belong to only one lower-level entity set
- Noted in E-R diagram by writing disjoint next to the ISA triangle
- An account entity may be Savings – account or Checking – account . I t satisfies just one condition at a time.

Overlapping

- An entity can belong to more than one lower-level entity set
- An employee can also be a customer.

Completeness Constraint :

A final constraint, is a completeness constraint on a generalization / specialization, which 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 :

(a) Total generalization / specialization :

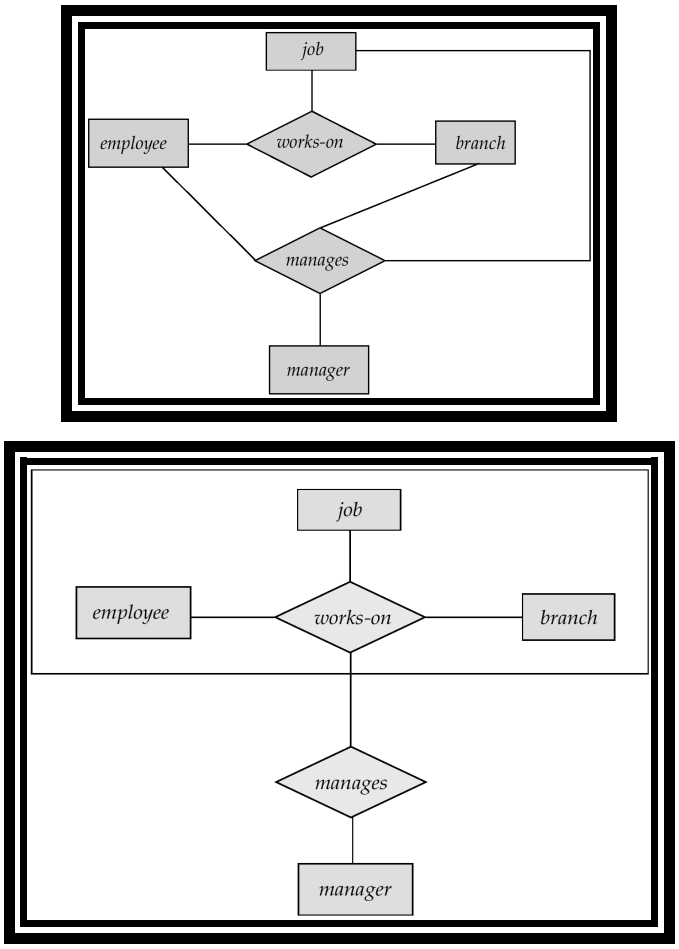
Each higher-level entity must belong to a lower-level entity set.

(b) Partial generalization / specialization :

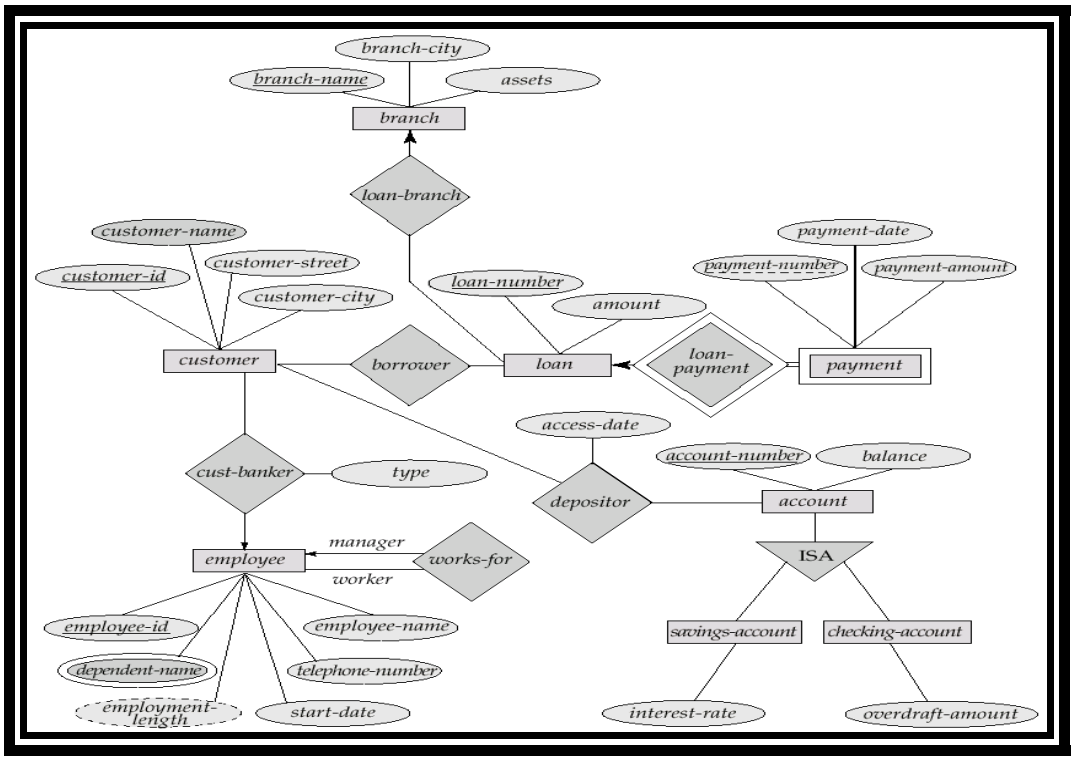
Some higher level entities may not belong to any lower level entity set.

Aggregation is an abstraction through which relationships are treated as higher – level entities . Thus, the relationship set Works-on relating the entity sets, employee , branch , and job

is considered as a higher-level entity set called **Works-on.** We can then create a binary relationship manages between Works-on and manager to represent who manages what tasks.



E-R Diagram for a Banking Enterprise

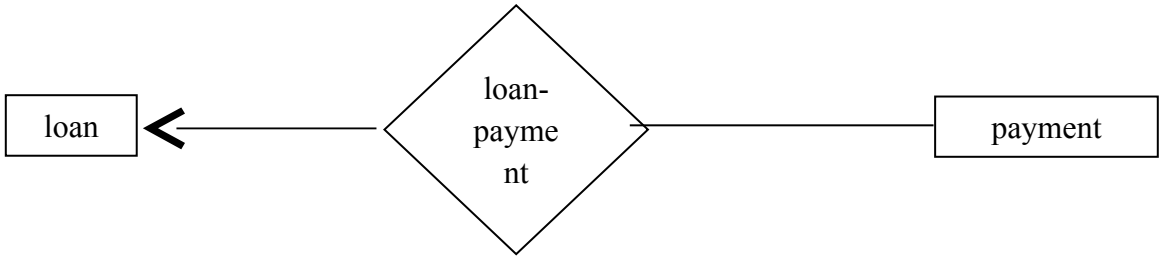


Existence Dependencies

- If the existence of entity x depends on the existence of entity y, then x is said to be existence dependent on y.

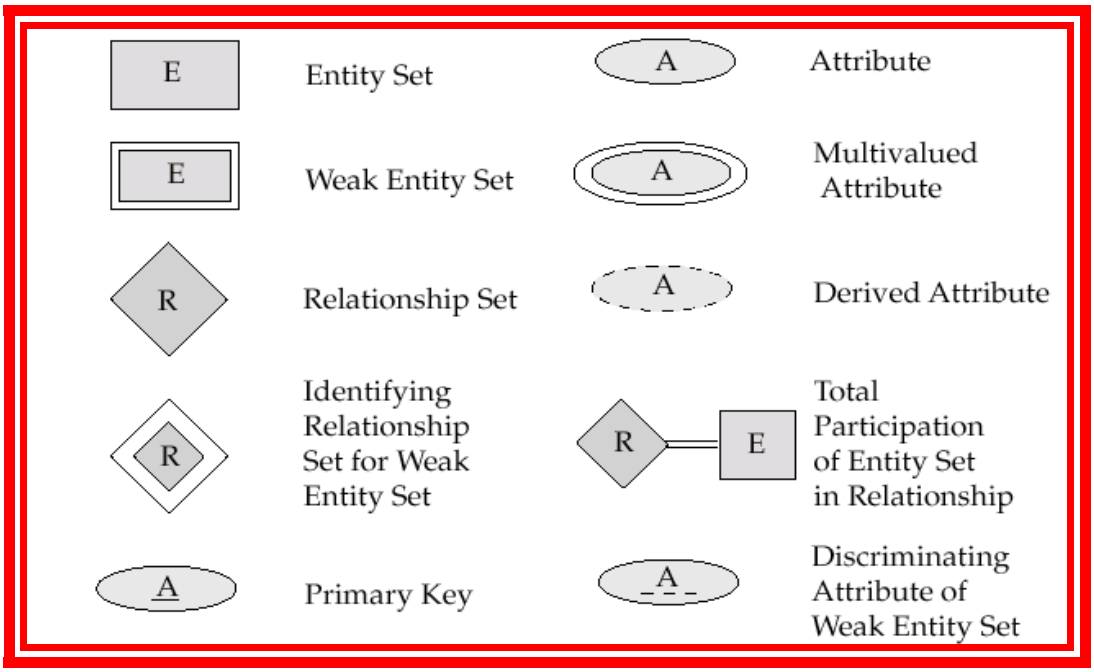
y is a dominant entity (in example below, loan)

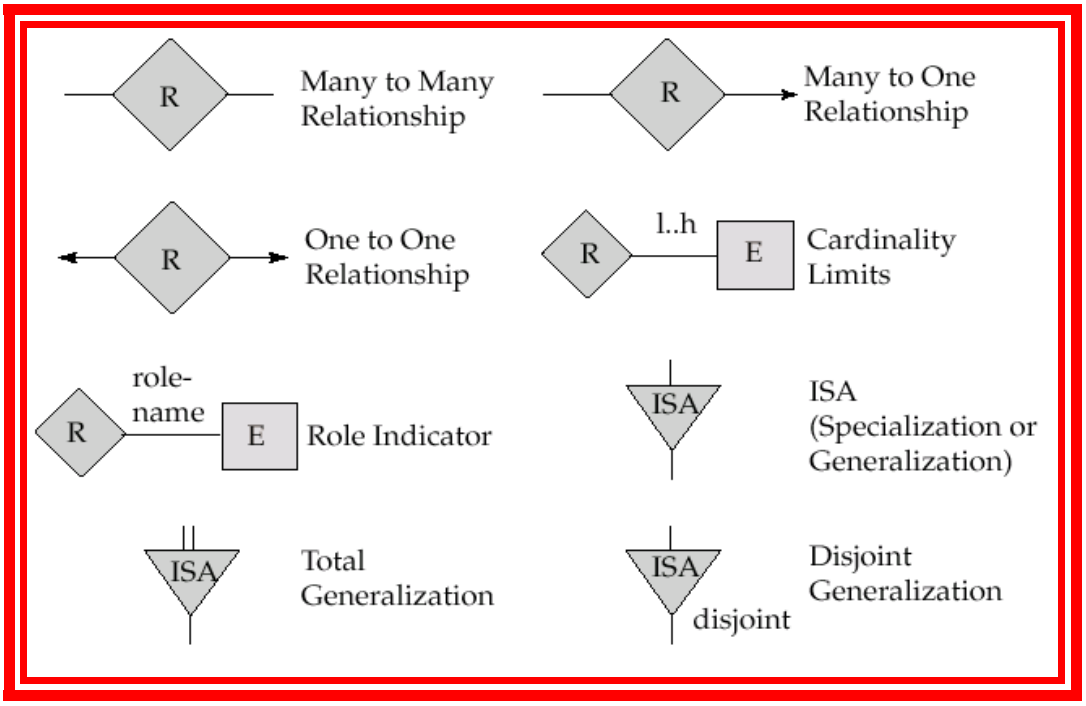
x is a subordinate entity (in example below, payment)



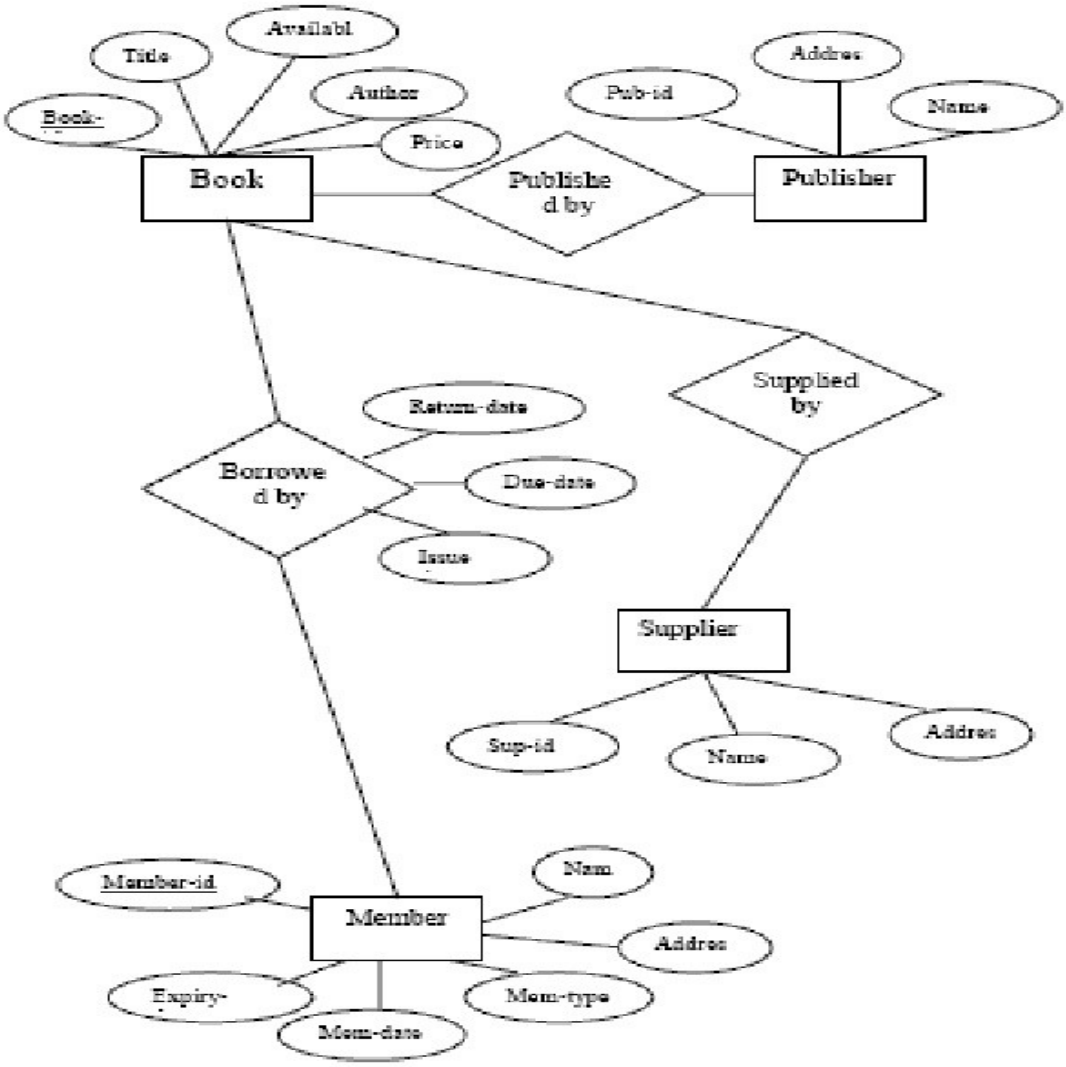
If a loan entity is deleted, then all its associated payment entities must be deleted also.

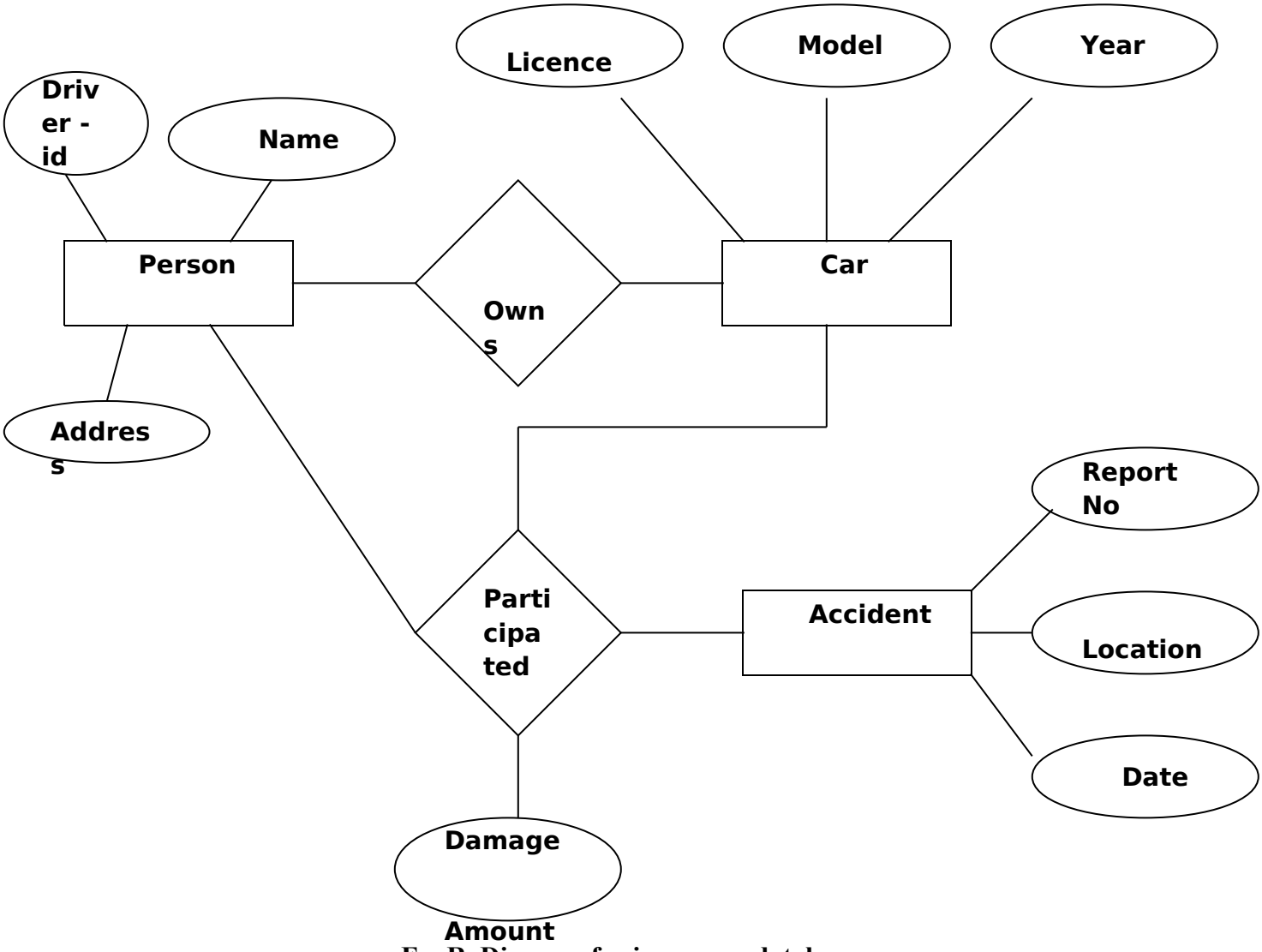
Summary of Symbols Used in E-R Notation





E-R Diagram for a Library Management Systems



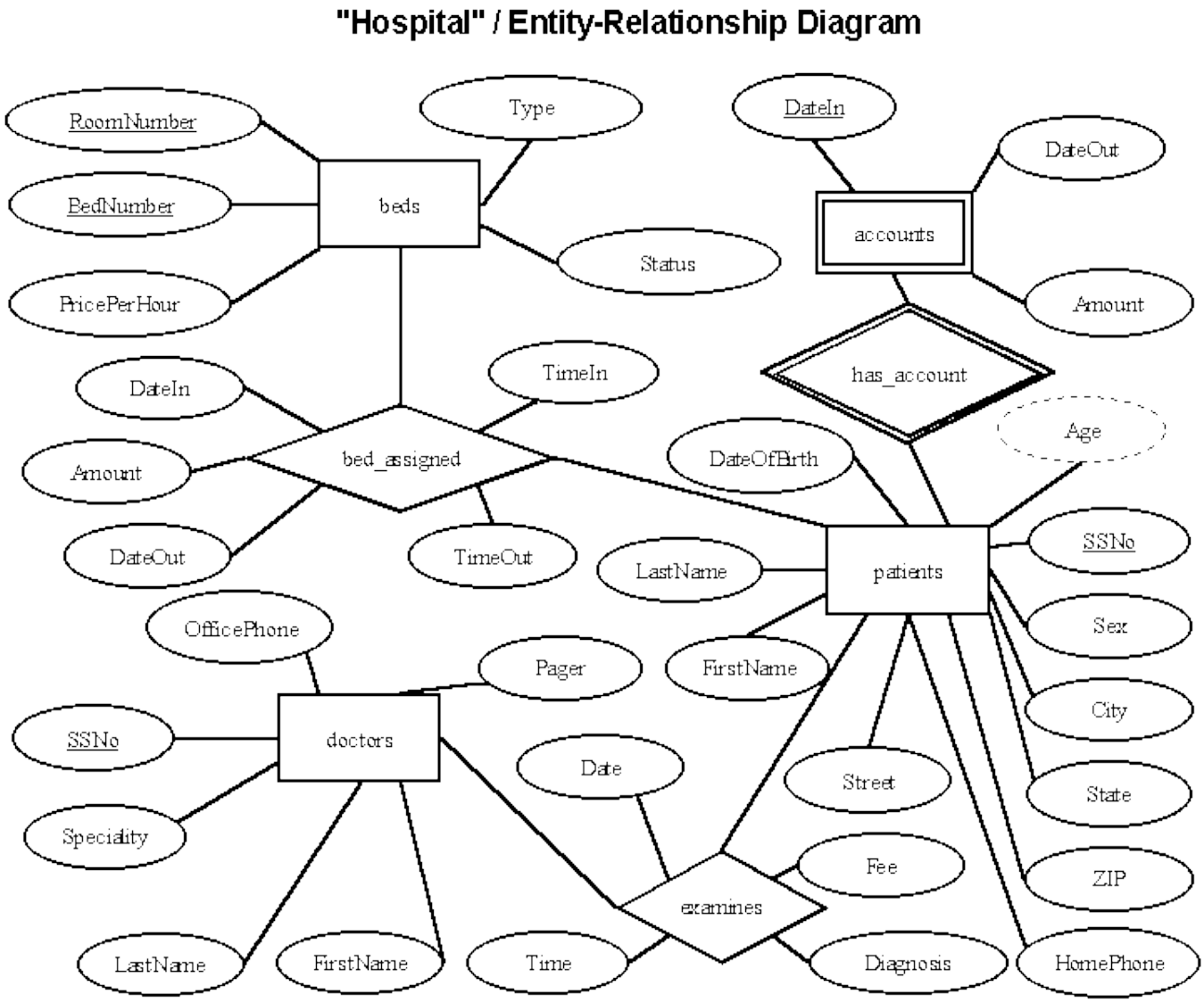


E – R Diagram for insurance database

Entity and Relationship sets for the hospital called General Hospital.

Patients, Doctors, Beds, Examines, BedAssigned, Accounts, HasAccount.

- patients, entity set with attributes SSNo, LastName, FirstName, HomePhone, Sex, DateofBirth, Age, Street, City, State, Zip.
- doctors, entity set with attributes SSNo, LastName, FirstName, OfficePhone, Pager, Specialty.
- examines, relational set with attributes Date, Time, Diagnosis, Fee.
- beds, entity set with attributes RoomNumber, BedNumber, Type, Status, PricePerHour.
- Bed_assigned, relational set with attributes DateIn, TimeIn, DateOut, TimeOut, Amount.
- accounts, weak entity set with attributes DateIn, DateOut, Amount.
- has_account, relational set with no Attributes



INTEGRITY AND SECURITY

Types of Constraints :

- Domain Constraints
- Referential Integrity
- Assertion
- Authorization

Domain Constraints:

A domain of possible values must be associated with every attribute. Domain constraints are the most elementary form of integrity constraints. They are tested easily by the system whenever a new data item is entered into the database.

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 cause the violation**.**

Assertion:

An assertion is any condition that the data base must always satisfy. Domain constraints & referential – integrity constraints are special forms of association.

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.

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.

Types of authorization:

- Read authorization
- Insert authorization
- Delete authorization
- Update authorization

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.

Delete Authorization : Which allows deletion of data.

The DDL gets as input some instructions and generates some output. The outputs of the DDL are 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.

SECURITY

The information in your database is important.

- Therefore, you need a way to protect it against unauthorized access, malicious destruction or alteration, and accidental introduction of data inconsistency.
- Some forms of malicious access:
- Unauthorized reading (theft) of data
- Unauthorized modification of data
- Unauthorized destruction of data
- To protect a database, we must take security measures at several levels.

Security Levels

- Database System: Since some users may modify data while some may only query, it is the job of the system to enforce authorization rules.
- Operating System: No matter how secure the database system is, the operating system may serve as another means of unauthorized access.
- Network: Since most databases allow remote access, hardware and software security is crucial.
- Physical: Sites with computer systems must be physically secured against entry by intruders or terrorists.
- Human: Users must be authorized carefully to reduce the chance of a user giving access to an intruder.

UNIT II RELATIONAL MODEL

The relation - Keys - Constraints - Relational algebra and Calculus - Queries - Programming and triggers

THE RELATIONAL MODEL

- The Relational Database Management System(RDBMS) has become the dominant data-processing software in use today.
- A relation is viewed as a two dimensional table of data where columns are the attributes of the data and rows, or tuples, contain records of the user data.
- Each row contains one piece of data per column.

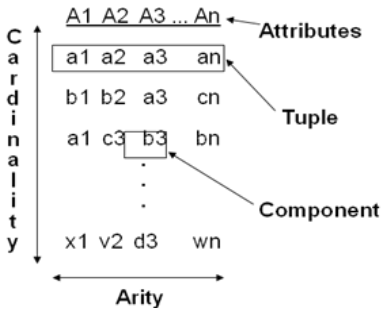
Properties of Relations

- The relation has a name that is distinct from all other relation names in the relation schema
- Each cell of the relation contains exactly one single value
- Each attribute has a distinct name
- The values of an attribute are all from the same domain
- Each tuple is distinct
- The order of attributes has no significance
- The order of tuples has no significance
- The main construct for representing data in the relational model is a relation. The relation consist of a relation schema and a relation instance.
- Relation instance is a table
- Relation schema describes the column heads for the table. Schema specifies the relation's name, the name of each field and the domain of each field.
- A relational database is a collection of relations with distinct relation names. The relational database schema is the collection of relation schemas for the relations in the database.

Relational Terminology

- Attribute (Column header)
- Relation (Table)
- Relation as table
- Rows = tuples

- Columns = components
- Names of columns = attributes
- Set of attribute names = schema
- Domain — set of values like a data type
- Cardinality = number of tuples.
- Degree / Arity = number of domains



INTEGRITY CONSTRAINTS OVER RELATIONS

An **integrity constraint (IC)** is a condition specified on a database schema and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a legal instance.

4 kinds of IC's:

1. Key Constraints
2. Attribute Constraints
3. Referential Integrity Constraints
4. Global Constraints

Key Constraints

A key constraint is a statement that a certain minimal subset of the fields of a relation is a unique identifier for a tuple.

SQL examples:

1. Primary Key:

```
CREATE TABLE branch(
        bname  CHAR(15)  PRIMARY KEY,
        bcity      CHAR(20),
        assets    INT);              or
CREATE TABLE depositor(
         cname   CHAR(15),
```

```
acct_no  CHAR(5),
PRIMARY KEY(cname, acct_no));
```

2. Candidate Keys:

```
CREATE TABLE customer (
    ssn     CHAR(9)    PRIMARY KEY,
    cname   CHAR(15),
    address CHAR(30),
    city          CHAR(10),
   UNIQUE (cname, address, city);
```

Effect of SQL Key declarations

PRIMARY (A1, A2, .., An) or

UNIQUE (A1, A2, ..., An)

Insertions: check if any tuple has same values for A1, A2, .., An as any inserted tuple. If found, reject insertion.

Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)

- 1 primary key per table, several unique keys allowed.
- Only primary key can be referenced by "foreign key" (ref integrity)
- DBMS may treat primary key differently
  - (e.g.: implicitly create an index on PK)
- NULL values permitted in UNIQUE keys but not in PRIMARY KEY

Attribute Constraints

- Attach constraints to values of attributes
- Enhances types system (e.g.: >= 0 rather than integer)

In SQL:

1. NOT NULL

e.g.:

```
CREATE TABLE branch(
    bname   CHAR(15)  NOT NULL,
    ....
    )
```

Note: declaring bname as primary key also prevents null values

2. CHECK

e.g.: CREATE TABLE depositor(

....

balance int NOT NULL,

CHECK(balance >= 0),

....

)

affect insertions, update in affected columns CHECK cond where cond is:

- Boolean expression evaluated using the values in the row being inserted or updated, and
- Does not contain subqueries; sequences; the SQL functions SYSDATE, UID, USER, or USERENV; or the pseudocolumns LEVEL or ROWNUM

Multiple CHECK constraints

- No limit on the number of CHECK constraints you can define on a column

CREATE TABLE credit_card(

....

balance int NOT NULL,

CHECK(balance >= 0),

CHECK (balance < limit),

....

)

Referential Integrity Constraints (Foreign key)

Foreign key : Set of fields in one relation that is used to `refer' to a tuple in another relation. (Must correspond to primary key of the second relation.) It prevents "dangling tuples".

Only students listed in the Students relation should be allowed to enroll for courses.

CREATE TABLE Enrolled

(sid CHAR(20), cid CHAR(20), grade CHAR(2),

PRIMARY KEY (sid,cid),

FOREIGN KEY (sid) REFERENCES Students)

CREATE TABLE Students

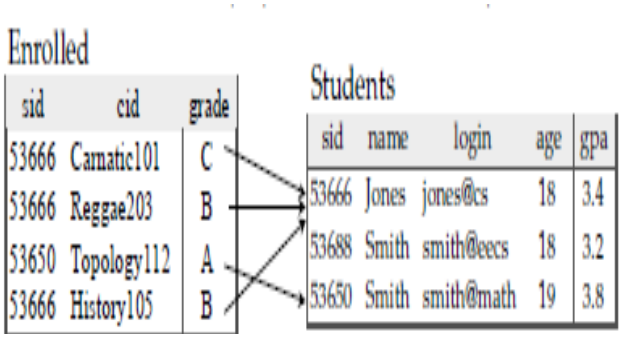
(sid: CHAR(20) PRIMARY KEY,

name: CHAR(20),

login: CHAR(10),

age: INTEGER,

gpa: REAL)



Global Constraints

Syntax: CHECK *conditional-expression.*

The conditional expression captures more General ICs than keys. The conditional expressions can use queries. The conditional expressions required to hold only if the associated table is nonempty. A CHECK constraint may be expressed over several tables; however, it is often expressed over one single table.

- Constraints can be named:

```
CONSTRAINT MyConstraint
CHECK conditional-expression
```

1) Single relation (constraints spans multiple columns)

E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

All Bkln branches must have assets > 5M

```
CREATE TABLE branch (
        .....
        bcity  CHAR(15),
        assets INT,
        CHECK (NOT(bcity = 'Bkln') OR assets > 5M))
```

Affects:

insertions into branch

updates of bcity or assets in branch

2) Multiple Relations: NOT supported in Oracle.Need to be implemented as a Trigger

SQL example: every loan has a borrower with a savings account

```
CREATE ASSERTION loan-constraint (
CHECK (NOT EXISTS (
        SELECT   *
        FROM  loan AS L
        WHERE  NOT EXISTS(
                SELECT   *
                FROM borrower B, depositor D, account A
                WHERE B.cname = D.cname  AND
```

D.acct_no = A.acct_no AND L.lno = B.lno))))

RELATIONAL ALGEBRA AND CALCULUS

Query languages: Allow manipulation and retrieval of data from a database.

Relational Algebra: Queries in relational algebra are composed using a collection of operators, and each query describes a step-by-step procedure for computing the desired answer; that is, queries are specified in an operational manner. Very useful for representing execution plans.

Relational Calculus: Lets users describe what they want, rather than how to compute it. (Nonoperational, declarative.)

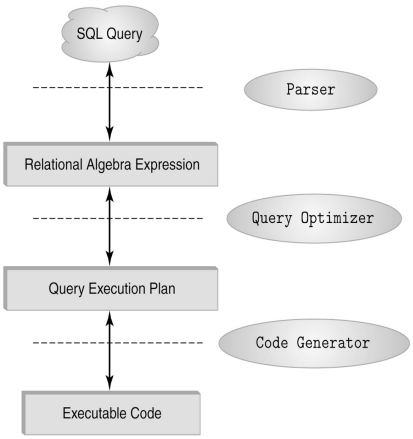
What is an Algebra?

- A language based on operators and a domain of values
- Operators map values taken from the domain into other domain values
- Hence, an expression involving operators and arguments produces a value in the domain
- When the domain is a set of all relations (and the operators are as described later), we get the *relational algebra*
- We refer to the expression as a *query* and the value produced as the *query result*

Relational Algebra

- *Domain*: set of relations
- *Basic operators*: select, project, union, set difference, Cartesian product
- *Derived operators*: set intersection, division, join
- *Procedural*: Relational expression specifies query by describing an algorithm (the sequence in which operators are applied) for determining the result of an expression.
- *Relational algebra* defines a set of operators that may work on relations. Recall that relations are simply data sets. As such, relational algebra deals with set theory. The operators in relational algebra are very similar to traditional algebra except that they apply to sets.

The Role of Relational Algebra in a DBMS



Example Instances : "Sailors" and "Reserves" relations for our examples.

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Projection (▪) Deletes unwanted columns from relation.

Cross-product (▪) Allows us to combine two relations.

Set-difference () Tuples in reln. 1, but not in reln. 2.

Union (▪) Tuples in reln. 1 and in reln. 2.

Additional operations:

Intersection, *join*, division, renaming

Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

Selection Operator

- The *selection operator* is similar to the projection operator. It produces a second relation that is a subset of the first.
- However, the selection operator produces a subset of tuples, not columns.
- The resulting relation contains all columns, but only contains a portion of the tuples.
- The selection or σ operation selects rows from a table that satisfy a ***condition***:

  Syntax : $\sigma_{<condition>} <tablename>$

  Ex: ☐*rating* ☐8 (s2)

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

Projection Operator

- A projection operation produces a second relation that is a subset of the first.
- The subset is in terms of columns, not tuples
- The resulting relation will contain a limited number of columns. However, every tuple will be listed.
- The projection or π operation selects a list of columns from a table.

Syntax : π < column list > < tablename >

Ex:

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$\pi_{sname,rating}(S2)$

age
35.0
55.5

$\pi_{age}(S2)$

Selection and Projection are usually combined:

sname	rating
yuppy	9
rusty	10

$\pi_{sname,rating}(\sigma_{rating>8}(S2))$

Set Operations

- All of these operations take two input relations, which must be union-compatible (ie)
- Both relations should have same number of fields and `Corresponding' fields taken in order from left to right have the same domains(types).

Union :

- It is denoted by U
- The union operator adds tuples from one relation to another relation
- A union operation will result in combined relation
- This is similar to the logical operator 'OR'
- Duplicate tuples are eliminated

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

Intersection Operator

- It is denoted by ∩
- An intersection operation will produce a third relation that contains the tuples that are common to the relations involved.
- This is similar to the logical operator 'AND'

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

Set Difference Operator

- It is denoted by – also called as Minus or Except
- The difference operator produces a third relation that contains the tuples that appear in the first relation, but not the second
- This is similar to a subtraction

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

Cartesian (Cross) Product

- A product operator i n one relation with every tuple in a second relation
- The resulting relation will have n x m tuples, where…

  n = the number of tuples in the first relation and

  m = the number of tuples in the second relation
- This is similar to multiplication

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Renaming operator :

- The RENAME operator is symbolized by ρ (rho).

  Syntax : ρ S(B1, B2, B3,….Bn)(R)
  - ρ is the RENAME operation.

- S is the new relation name.
- B1, B2, B3, …Bn are the new renamed attributes (columns).
- R is the relation or table from which the attributes are chosen.

- The RENAME operator is used to give a name to results or output of queries, returns of selection statements, and views of queries that we would like to view at some other point in time.
- Ex: ρ (C(1→ sid1,5→sid2), S1X R1)

Join Operator

- The join operator is a combination of the product, selection, and projection operators. There are several variations of the join operator.
  - Condition Join
  - Equijoin
  - Natural join
  - Outer join
  - Left outer join
  - Right outer join

Condition Joins:

- Defined as a cross-product followed by a selection:

$R \bowtie_c S = \sigma_c(R \times S)$ (⋈ is called the bow-tie)

where c is the condition.

Example: The condition join

$S1 \bowtie_{S1.sid<R1.sid} R1$ yields

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

- Result schema same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently.
- Sometimes called a theta-join.

Equi-Join

- A special case of condition join where the condition *c* contains only ***equalities*.**

- *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.
- The operator

S1$\bowtie_{R.sid=S.sid}$ R1 yields

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

Natural Join

- Special case of equijoin where equalities are implicitly specified on *all* fields having the same name in R and S.

- The condition c is now left out, so that the "bow tie" operator by itself signifies a natural join.

- If the two relations have no attributes in common, the natural join is simply the cross-product.

- It has the nice property that the result is guaranteed not to have two fields with the same name.

Ex: **S1$\bowtie$ R1**

sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Figure 4.13 $S1 \bowtie_{R.sid=S.sid} R1$

Division

Not supported as a primitive operator, but useful for expressing queries like:

*Find sailors who have reserved **all** boats*.

Let A have 2 fields, x and y; B have only field y:

$$A/B = \{\langle x\rangle \mid \exists\langle x,y\rangle \in A \ \forall\langle y\rangle \in B\}$$

i.e., A/B contains all x tuples (sailors) such that for every y tuple (boat) in B, there is an xy tuple in A.

Or: If the set of y values (boats) associated with an x value (sailor) in A contains all y values in B, the x value is in A/B.In general, x and y can be any lists of fields; y is the list of fields in B, and : of fields of A.

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

A

pno
p2

B1

pno
p2
p4

B2

p2
p4

B3

sno
s1
s2
s3
s4

A/B1

sno
s1
s4

A/B2

sno
s1

A/B3

RELATIONAL CALCULUS

Relational calculus consists of two calculi, the tuple relational calculus (TRC) and the domain relational calculus (DRC), that are part of the relational model for databases and provide a declarative way to specify database queries. This in contrast to the relational algebra which is also part of the relational model but provides a more procedural way for specifying queries.

The relational algebra might suggest these steps to retrieve the phone numbers and names of book stores that supply *Some Sample Book*:

1. Join book stores and titles over the BookstoreID.
2. Restrict the result of that join to tuples for the book *Some Sample Book*.
3. Project the result of that restriction over StoreName and StorePhone.

The relational calculus would formulate a descriptive, declarative way:

Get StoreName and StorePhone for supplies such that there exists a title BK with the same BookstoreID value and with a BookTitle value of *Some Sample Book*.

The relational algebra and the relational calculus are essentially logically equivalent: for any algebraic expression, there is an equivalent expression in the calculus, and vice versa. This result is known as Codd's theorem.

TUPLE RELATIONAL CALCULUS

The tuple relational calculus is a nonprocedural language.

We must provide a formal description of the information desired.

1. A query in the tuple relational calculus is expressed as

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

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

2. We also use the notation

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

For Example: Find all sailors with a rating above 7. (Sailors- S is a relation)

{ S | S ε Sailors ^ S: rating > 7 }

When this query is evaluated on an instance of the Sailors relation, the tuple variable *S* is instantiated successively with each tuple, and the test *S.rating>7* is applied. The answer contains those instances of *S* that pass this test.

Syntax of TRC Queries

Let *Rel* be a relation name, *R* and *S* be tuple variables, *a* an attribute of *R*, and *b* an attribute of *S*. Let op denote an operator in the set (<, >, =, ≤, ≥).

An **atomic formula** is one of the following:

· R ε Rel (ε is belongs to)

· R.a op S.b

· R.a op constant, or constant op R.a

A **formula** is recursively defined to be one of the following, where *p* and *q* are themselves formulas, and *p*(*R*) denotes a formula in which the variable *R* appears:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists R(p(R))$, where R is a tuple variable
- $\forall R(p(R))$, where R is a tuple variable

The **quantifiers "For any" and "For all"** are said to **bind** the variable *R*. A variable is said to be **free** in a formula or *subformula* (a formula contained in a larger formula), if the subformula does not contain an occurrence of a quantifier that binds it.

A **TRC query** is defined to be expression of the form { *T* | p(T) }, where *T* is the only free variable in the formula p.

Semantics of TRC Queries

The **answer** to a TRC query { *T* | p(T) }, is the set of all tuples *t* for which the formula *p*(*T*) evaluates to true with variable *T* assigned the tuple value *t*.

A query is evaluated on a given instance of the database. Let each free variable in a formula *F* be bound to a tuple value. For the given assignment of tuples to variables, with respect to the given database instance, *F* evaluates to (or simply 'is') true if one of the following holds:

• F is an atomic formula **R ε Rel**, and R is assigned a tuple in the instance of relation *Rel*.

• F is a comparison *R.a* op *S.b*, *R.a* op *constant*, or *constant* op *R.a*, and the tuples assigned to *R* and *S* have field values *R.a* and *S.b* that make the comparison true.

• F is of the form ¬*p*, and *p* is not true; or of the form *p* ^ *q*, and both *p* and *q* are true; or of the form *p v q*, and one of them is true, or of the form (*p) q* and *q* is true whenever *p* is true.

• F is of the form For Any *R*(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*), including the variable *R* that makes the formula *p*(*R*) true.

• *F* is of the form For all R(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*) that makes the formula *p*(*R*) true no matter what tuple is assigned to *R*.

Examples of TRC Queries

A formula *p*(*R*) includes a condition *R* ε *Rel*, and the meaning of the phrases *some tuple R* and *for all tuples R* is intuitive. We will use the notation For any *R* ε *Rel*(*p*(*R*)) for any *R*(*R* ε *Rel* ^ *p*(*R*)).

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

An Instance *S*3 of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

An Instance *R*2 of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

An Instance *B*1 of Boats

Similarly, we use the notation For all R ε Rel(p(R)) for all R(R ε Rel) p(R)).

- Find the names and ages of sailors with a rating above 7.

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge P.name = S.sname \wedge P.age = S.age)\}$$

The result of this query is a relation with two fields, *name* and *age*. The atomic formulas P.name = S.sname and P.age = S.age give values to the fields of an answer tuple P. On instances B1, R2, and S3, the answer is the set of tuples <Lubber, 55:5>, <Andy, 25:5>, <Rusty, 35.0>, <Zorba, 16.0>, and <Horatio, 35.0>.

- Find the sailor name, boat id, and reservation date for each reservation.

$$\{P \mid \exists R \in Reserves \;\; \exists S \in Sailors \\ (R.sid = S.sid \wedge P.bid = R.bid \wedge P.day = R.day \wedge P.sname = S.sname)\}$$

Answer to Query

sname	*bid*	*day*
Dustin	101	10/10/98
Dustin	102	10/10/98
Dustin	103	10/8/98
Dustin	104	10/7/98
Lubber	102	11/10/98
Lubber	103	11/6/98
Lubber	104	11/12/98
Horatio	101	9/5/98
Horatio	102	9/8/98
Horatio	103	9/8/98

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

$$\{P \mid \exists S \in Sailors \;\; \exists R \in Reserves(R.sid = S.sid \wedge R.bid = 103 \wedge P.sname = S.sname)\}$$

This query can be read as follows: "Retrieve all sailor tuples for which there exists a tuple in Reserves, having the same value in the *sid* field, and with *bid* = 103." That is, for each sailor tuple, we look for a tuple in Reserves that shows that this sailor has reserved boat 103. The answer tuple *P* contains just one field, *sname*.

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

$$\{P \mid \exists S \in Sailors \;\; \exists R \in Reserves(R.sid = S.sid \wedge P.sname = S.sname \\ \wedge \exists B \in Boats(B.bid = R.bid \wedge B.color = 'red'))\}$$

This query can be read as follows: "Retrieve all sailor tuples *S* for which there exist tuples *R* in Reserves and B in Boats such that S.sid = R.sid, R.bid = B.bid, and B.color = 'red'." Another way to write this query, which corresponds more closely to this reading, is as follows:

$$\{P \mid \exists S \in Sailors \;\; \exists R \in Reserves \;\; \exists B \in Boats$$
$$(R.sid = S.sid \wedge B.bid = R.bid \wedge B.color = 'red' \wedge P.sname = S.sname)\}$$

- Find the names of sailors who have reserved at least two boats.

$$\{P \mid \exists S \in Sailors \;\; \exists R1 \in Reserves \;\; \exists R2 \in Reserves$$
$$(S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid \wedge P.sname = S.sname)\}$$

Contrast this query with the algebra version and see how much simpler the calculus version is. In part, this difference is due to the cumbersome renaming of fields in the algebra version, but the calculus version really is simpler.

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

$$\{P \mid \exists S \in Sailors \;\; \forall B \in Boats$$
$$(\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$$

The Domain Relational Calculus

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

A DRC query has the form

$$\{ \langle x_1, x_2, \ldots\ldots x_n \rangle \mid p(\langle x_1, x_2, \ldots. X_n \rangle) \}$$

where each x_i is either a *domain variable* or a constant and

$p(\langle x_1, x_2, \ldots. X_n \rangle)$ denotes a **DRC formula** whose only free variables are the variables among the x_i ; $1 \leq I \leq n$. The result of this query is the set of all tuples $\langle x_1, x_2 \ldots. x_n \rangle$ for which the formula evaluates to true. A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let op denote an operator in the set $\{ <, >, =, \leq, \geq, \neq \}$ and let X and Y be domain variables.

An **atomic formula** in DRC is one of the following:

· $\langle x_1, x_2, \ldots. x_n \rangle \in$ Rel, where Rel is a relation with n attributes; each x_i ; $1 \leq I \leq n$ is either a variable or a constant.

· X op Y

· X op *constant*, or *constant* op X

A **formula** is recursively defined to be one of the following, where p and q are themselves formulas, and $p(X)$ denotes a formula in which the variable X appears: any atomic formula.

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists X(p(X))$, where X is a domain variable
- $\forall X(p(X))$, where X is a domain variable

Examples of DRC Queries

- Find all sailors with a rating above 7.

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge T > 7\}$$

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

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists Ir, Br, D(\langle Ir, Br, D\rangle \in Reserves \wedge Ir = I \wedge Br = 103))\}$$

This query can also be written as follows; notice the repetition of variable I and the use of the constant 103:

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists D(\langle I, 103, D\rangle \in Reserves))\}$$

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

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists \langle I, Br, D\rangle \in Reserves \wedge \exists \langle Br, BN, 'red'\rangle \in Boats)\}$$

- Find the names of sailors who have reserved at least two boats.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists Br1, Br2, D1, D2(\langle I, Br1, D1\rangle \in Reserves \wedge \langle I, Br2, D2\rangle \in Reserves \wedge Br1 \neq Br2)$$

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

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \forall B, BN, C(\neg(\langle B, BN, C\rangle \in Boats) \vee (\exists \langle Ir, Br, D\rangle \in Reserves(I = Ir \wedge Br = B))))\}$$

- Find sailors who have reserved all red boats.

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge \forall\langle B,BN,C\rangle \in Boats$$
$$(C = 'red' \Rightarrow \exists\langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))\}$$

Structured Query Language (SQL)

SQL, which is an abbreviation for **Structured Query Language**, is a language to request data from a database, to add, update, or remove data within a database, or to manipulate the metadata of the database. SQL is generally pronounced as the three letters in the name, e.g. *ess-cue-ell*, or in some people's usage, as the word *sequel*.

SQL was initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s. This version, initially called **SEQUEL** (*Structured English Query Language*), was designed to manipulate and retrieve data stored in IBM's original quasi-relational database management system, System R, which a group at IBM San Jose Research Laboratory had developed during the 1970s. The acronym SEQUEL was later changed to SQL because "SEQUEL" was a trademark of the UK-based Hawker Siddeley aircraft company.

Various aspects of the SQL Language:

The Data Manipulation Language (DML)

This subset of SQL allows users to pose queries and to insert, delete, and modify rows.

The Data Definition Language (DDL)

This subset of SQL supports the creation, deletion, and modification of definitions for tables and Views.

Triggers and Advanced Integrity Constraints

Triggers are actions executed by the DBMS whenever changes to the database meet conditions specified in the trigger.

Embedded and Dynamic SQL

Embedded SQL features allow SQL code to be called from a host language such as C or C++. Dynamic SQL features allow a query to be constructed at run-time.

Transaction Management

Various commands allow a user to explicitly control aspects of how a transaction is to be executed .

Security

SQL provides mechanisms to control users' access to data users objects such as tables and views

Advanced Features

The SQL includes object-oriented features, recursive queries, spatial data management, etc.

The Form of a Basic SQL Query

Basic Structure

- A relational database consists of a collection of relations, each of which is assigned a unique name.

The basic structure of an SQL expression consists of three clauses :

- **Select**
- **From**
- **Where**

Select – It is used to list the attributes desired in the result of a query.

From – It lists the relations to be scanned in the evaluation of the expression.

Where – It consists of a predicate involving attributes of the relations that appear in the from clause.

A typical SQL query has the form,

Select A1,A2,…..An

from r1,r2,…..rm

where P

Where,

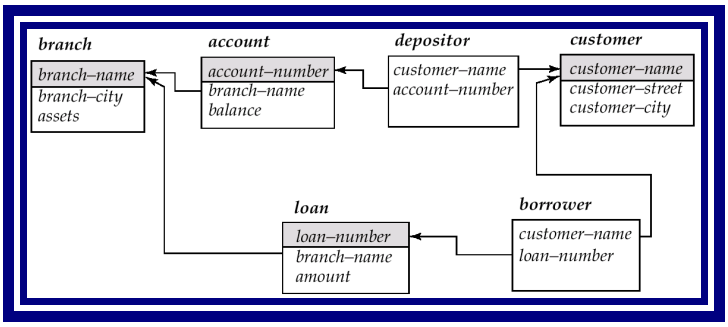
Ai – attribute

ri – relation

P – predicate

If the where clause is omitted, the predicate P is true.

Schema Used in Examples



The Select Clause :

- The result of an SQL query is a relation.

Ex : **Select branch-name from loan**

The above will find the names of all branches in the loan relation.SQL allow duplicates in relations as well as in query results. To force the elimination of duplicates, insert the keyword **distinct** after **select.**

Select distinct *branch-name*

from *loan*

The keyword **all** specifies that duplicates not be removed.

S**elect all** *branch-name*

from *loan*

- An asterisk (*) in the select clause denotes "all attributes"
- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.

The query:

S**elect** *loan-number, branch-name, amount* *100

from *loan*

The Where Clause :

The find all loan number for loans made a the Perryridge branch with loan amounts greater than $1200.

Select *loan-number*

from *loan*

where *branch-name* = 'Perryridge' **and** *amount* > 1200

Comparison results can be combined using the logical connectives **and, or,** and **not.** Comparisons can be applied to results of arithmetic expressions. SQL Includes a **between** comparison operator in order to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.

Find the loan number of those loans with loan amounts between $90,000 and $100,000 (i.e. >=$90,000 and <=$100,000)

S**elect** *loan-number*

from *loan*

where *amount* **between** 90000 **and** 100000

The from Clause :

Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

S**elect** *customer-name, borrower.loan-number, amount* **from** *borrower, loan*

where *borrower.loan-number = loan.loan- number*

The Rename Operation :

- The SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** new-name

- Find the name, loan number and loan amount of all customers; rename the column name loan-number as loan-id.

  Select loan-number as loan-id, amount

  from loan

Tuple Variables:

- Tuple variables are defined in the **from** clause via the use of the **as** clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

  Select customer-name, T.loan-number, S.amount **from** borrower

  as T, loan **as** S

  where T.loan-number = S.loan-number

String Operations :

- SQL supports a variety of string operations
- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:
- percent (%). The % character matches any substring.
- underscore (_). The _ character matches any character.
- Find the names of all customers whose street includes the substring "Main".

  select customer-name

  from customer

  where customer-street **like '**%Main%'

Ordering the Display of Tuples

- List in alphabetic order the names of all customers having a loan in Perryridge branch

  select * from *loan*

  order by branchname

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.

- E.g. **order by** *customer-name* **desc**

Set Operations

1. **Union**
2. **Intersect**
3. **Except**

The Union Operation

The **UNION** clause combines the results of two SQL queries into a single table of all matching rows. The two queries must result in the same number of columns and compatible data types in order to unite.

To find all the bank customers having a loan, an account, or both at the bank,

we write,

(select *customer name* from *depositor*)

union

(select *customer name* from *borrower*)

The union operation automatically eliminates duplicates. Thus, in the preceding query, if a customer—say, Jones—has several accounts or loans (or both) at the bank, then Jones will appear only once in the result. If we want to retain all duplicates, we must write union all in place of union:

(select *customer name* from *depositor*)

union all

(select *customer name* from *borrower*)

The Intersect Operation

The SQL INTERSECT operator takes the results of two queries and returns only rows that appear in both result sets.

To find all customers who have both a loan and an account at the bank, we write

(select customer name from depositor)

intersect

(select customer name from borrower)

The intersect operation automatically eliminates duplicates. If we want to retain all duplicates, we must write intersect all in place of intersect:

(select *customer name* from *depositor*)

intersect all

(select *customer name* from *borrower*)

The Except Operation

The SQL EXCEPT operator takes the distinct rows of one query and returns the rows that do not appear in a second result set.

To find all customers who have an account but no loan at the bank, we write

(select *customer name* from *depositor*)

except

(select *customer name* from *borrower*)

The except operation automatically eliminates duplicates.

- If we want to retain all duplicates, we must write except all in place of except:

(select *customer name* from *depositor*)

except all

(select *customer name* from *borrower*)

Nested Queries

Queries can be nested so that the results of one query can be used in another query via a relational operator or aggregation function. A nested query is also known as a **subquery**. While joins and other table operations provide computationally superior (i.e. faster) alternatives in many cases, the use of subqueries introduces a hierarchy in execution which can be useful or necessary.

Ex:

```
SELECT isbn, title, price
FROM Book
WHERE price < AVG(SELECT price FROM Book)
ORDER BY title;
```

Correlated subquery

In a SQL database query, a **correlated sub-query** is a sub-query that uses values from the outer query in its WHERE clause. The sub-query is evaluated once for each row processed by the outer query.

Here is an example for a typical correlated sub-query. In this example we are finding the list of employees (employee number and names) having more salary than the average salary of all employees in that employee's department.

```
SELECT employee_number, name
FROM employee AS e1
WHERE salary > (SELECT avg(salary)
```

```
        FROM employee
        WHERE department = e1.department);
```

Set-Comparison Operators

Find sailors whose rating is greater than that of some sailor called Horatio:

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

Aggregate Functions

Aggregate functions return a single value based upon a set of other values. If used among many other expressions in the item list of a SELECT statement, the SELECT must have a GROUP BY clause. No GROUP BY clause is required if the aggregate function is the only value retrieved by the SELECT statement.

Function	**Usage**
AVG(expression)	Computes the average value of a column by the expression
COUNT(expression)	Counts the rows defined by the expression
COUNT(*)	Counts all rows in the specified table or view
MIN(expression)	Finds the minimum value in a column by the expression
MAX(expression)	Finds the maximum value in a column by the expression
SUM(expression)	Computes the sum of column values by the expression

SQL Aggregate Function is built-in functions for counting and calculation (perform a calculation on a set of values and return a single value)
Syntax for built-in SQL functions is - SELECT function(column) FROM table

Name	**Salary**
Emil	5000
Chang	5000
Emily	4500

Nick	4000

Table Name: **A**
Below is Example for SQL built-in Function , together with SQL Statement and the answer.
AVG - Average value of columns
Select AVG(Salary) FROM A
Value = 4675
COUNT - number of rows
Select COUNT(*) FROM A
Value = 4
SELECT COUNT (DISTINCT Salary) FROM CUSTOMERS
Value = 3
MAX - Maximum or Highest number in a column
SELECT MAX(Salary) FROM CUSTOMERS
Value = 5000

MIN - Minimum or Lowest number in a column
SELECT MIN(Salary) FROM CUSTOMERS
Value = 4000

SUM - Total number in a column
SELECT SUM(Salary) FROM CUSTOMERS
Value = 18500

The GROUP BY and HAVING Clause

The HAVING clause is used in combination with the GROUP BY clause. It can be used in a SELECT statement to filter the records that a GROUP BY returns.

The syntax for the HAVING clause is:

```
SELECT column1, column2, ... column_n, aggregate_function (expression)
FROM tables
WHERE predicates
GROUP BY column1, column2, ... column_n
HAVING condition1 ... condition_n;
```

- Find the names of all branches and the average account balance

```
select branch-name, avg (balance)
from account
group by branch-name
```

- Find the names of all branches where the average account balance is more than $1,200.

```
select branch-name, avg (balance)
from account
group by branch-name
having avg (balance) > 1200
```

NULL Values:

Null is a special marker used in Structured Query Language (SQL) to indicate that a data value does not exist in the database. Introduced by the creator of the relational database model, E. F. Codd, SQL Null serves to fulfill the requirement that all true relational database management systems (RDBMS) support a representation of "missing information and inapplicable information". NULL is also an SQL reserved keyword used to identify the Null special marker.

Comparison Using Null Values

Basic SQL comparison operators always return Unknown when comparing anything with Null, so the SQL standard provides for two special Null-specific comparison predicates. The **IS NULL** and **IS NOT NULL** predicates test whether data is, or is not, Null.

Ex:

```
SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NULL
SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NOT NULL
```

Logical connectives AND, OR AND NOT

Since Null is not a member of any data domain, it is not considered a "value", but rather a marker indicating the absence of value. Because of this, comparisons with Null can never result in either True or False, but always in a third logical result, Unknown.

p	*q*	*p* OR *q*	*p* AND *q*	*p* = *q*
True	True	True	True	True
True	False	True	False	False
True	Unknown	True	Unknown	Unknown
False	True	True	False	False
False	False	False	False	True
False	Unknown	Unknown	False	Unknown
Unknown	True	True	Unknown	Unknown
Unknown	False	Unknown	False	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown

p	NOT *p*
True	False
False	True
Unknown	Unknown

Impact on SQL Constructs

- WHERE clause: eliminates rows in which qualification does not evaluate to
- true
- In the presence of null values, any row evaluating to false or unknown is eliminated.
- Eliminating such rows has a subtle but Significant impact on nested queries involving EXISTS or UNIQUE

- Duplicates: Two rows are duplicates if the corresponding columns are either equal or both contain null values
- Contrast this with comparison: Two null values when compared using = is unknown (In the case of duplicates, this comparison is implicitly treated as true which is an anomaly)
- Arithmetic operations +,−, * and/ all return null if one of their operators is null .
- COUNT(*) treats null values just like other values (included in the count)
- ALL other operators COUNT,SUM,MIN,MAX,AVG and variations using DISTINCT-discard null values
- When applied to ONLY null values, then the result is null

OUTER JOINS

Interesting variant of the join operation that rely on null values - Supported in SQL

Consider the join

Sailor $\bowtie_C$ Reserves

- Tuples of Sailors that do not match some row in Reserves according to the join condition C does not appear in the result
- In an outer join, sailor rows without a matching Reserves row appear exactly once in the result, with the columns from the Reserves relation assigned null values

Variants:

LEFT OUTER JOIN: Sailor rows without matching Reserves row appear in the result, not vice versa.

RIGHT OUTER JOIN: Reserves rows without matching Sailor row appear in the result, not vice versa.

FULL OUTER JOIN: Both Sailor and Reserves rows without matching tuples, appear in the result.

EXAMPLE : In SQL, OUTER JOIN is specified in the FROM clause.

SELECT S.sid, R.bid FROM Sailors S NATURAL LEFT OUTER JOIN Reserves R

On the instances, the result is shown

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

sid	bid
22	101
31	null
58	103

- We can disallow null values by specifying **NOT NULL**. There is an implicit NOT NULL constraint for every field listed in a PRIMARY KEY constraint.

Embedded SQL

- **Embedded SQL** is a method of combining the computing power of a programming language and the database manipulation capabilities of SQL.
- Because SQL does not use variables and control-of-flow statements, it is often used as a database sublanguage that can be added to a program written in a conventional programming language, such as C or COBOL.
- This is a central idea of embedded SQL: placing SQL statements in a program written in a host programming language
- A language in which SQL queries are embedded is referred to as a **Host Language.**
  - The structures permitted in the host language constitute **Embedded SQL.**
  - An embedded SQL program must be processed by a special preprocessor prior to compilation.
  - The preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allows run-time execution of the database accesses.
  - Then, the resulting program is compiled by the host-language compiler.
  - To identify embedded SQL requests to the preprocessor, we use the EXEC SQL statement.
- It has the form,

  EXEC SQL <embedded SQL statement> END-EXEC
- The exact syntax for embedded SQL requests depends on the language in which SQL is embedded.
- Variables of the host language can be used within embedded SQL statement, but they must be preceded by a colon(:) to distinguish them from SQL variables.

The cursor statement :

EXEC SQL declare c cursor for END EXEC

- Queries that return multiple rows of data are handled with cursors. A cursor keeps track of the current row within a result set. The DECLARE CURSOR statement defines the query.

The open statement :

EXEC SQL open c END EXEC

- the OPEN statement begins the query processing,

```
EXEC SQL
    DECLARE c CURSOR FOR
    SELECT EMPNO,JOB
    FROM EMP
```

```
            FOR UPDATE OF JOB
        END-EXEC


EXEC SQL

OPEN c
END-EXEC.

EXEC SQL
  WHENEVER NOT FOUND
   GO TO CLOSE-c
END-EXEC. EXEC SQL
  FETCH c
  INTO :EMP-NUM, :JOB-CODE
END-EXEC.

EXEC SQL
    UPDATE EMP
    SET JOB = :NEW-CODE
    WHERE CURRENT OF c
END-EXEC.

CLOSE-c.
EXEC SQL
CLOSE c
END-EXEC.
```

The fetch statement :

EXEC SQL fetch c into :cc END EXEC

- the FETCH statement retrieves successive rows of data

The close statement:

EXEC SQL close c END EXEC

- the CLOSE statement ends query processing.

ODBC and JDBC

In computing, **ODBC** (**Open Database Connectivity**) is a standard C programming language interface for accessing database management systems (DBMS). The designers of ODBC aimed to make it independent of database systems and operating systems. An application can use ODBC to query data from a DBMS, regardless of the operating system or DBMS it uses.

The ODBC architecture has four components:

Application

(Spreadsheet, Word processor, Data Access & Retrievable Tool, Development Language etc.) Performs processing by passing SQL Statements to and receiving results from the ODBC Driver Manager.

Data Source

It consists of a DBMS, the operating system the DBMS runs on, and the network (if any) used to access the DBMS.

Drivers

A *Dynamic Link Library* that Processes ODBC function calls received from the Driver Manager, submitting the resultant SQL requests to a specific data source, and returns results to the application. If necessary, the driver modifies an application's request so that the request conforms to syntax supported by the associated DBMS. An ODBC driver enables an ODBC-compliant application to use a *data source*, normally a DBMS.

ODBC Driver Manager

The Driver Manager (DM) is the software that loads a particular driver based on the connection information. An application is actually linked to the DM. When the application calls the ODBC function to connect to the DBMS, the DM parses the *connection string* and loads the appropriate driver.

JDBC driver

A **JDBC driver** is a software component enabling a Java application to interact with a database. JDBC drivers are analogous to ODBC drivers, ADO.NET data providers, and OLE DB providers.

To connect with individual databases, JDBC (the Java Database Connectivity API) requires drivers for each database. The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.

JDBC technology drivers fit into one of four categories.

Type 1 Driver - bridges

This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS. An example is JDBC-ODBC bridge, is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls.

Functions

- Translates a query by JDBC into a corresponding ODBC query, which is then handled by the ODBC driver.

Type 2 Driver - Native-API Driver

The JDBC type 2 driver, also known as the Native-API driver, is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

Type 3 Driver - Network-Protocol Driver

The JDBC type 3 driver, also known as the Pure Java Driver for Database Middleware, is a database driver implementation which makes use of a middle tier between the calling program and the database. The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.

Functions

- Follows a three tier communication approach.
- Can interface to multiple databases - Not vendor specific.
- The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database commands for that database.
- Thus the client driver to middleware communication is database independent.

Type 4 Driver - Native-Protocol Driver

The JDBC type 4 driver, also known as the Direct to Database Pure Java Driver, is a database driver implementation that converts JDBC calls directly into a vendor-specific database protocol.

An Example using JDBC

```
Connection con = null;
 Class.forName("com.mysql.jdbc.Driver");
con = DriverManager.getConnection("jdbc:mysql://localhost:3306/jdbctutorial","root","root");
 String sql = "INSERT into emp VALUES(?,?)";
  PreparedStatement prest = con.prepareStatement(sql);
  prest.setInt(1, eno); prest.setInt(2, age)
  prest.executeUpdate();
 con.close();
```

Description of code:

Connection:

This is an interface in ***java.sql*** package that specifies connection with specific database like: **MySQL**, **Ms-Access**, **Oracle** etc and java files. The **SQL** statements are executed within the context of the Connection interface.

Class.forName(String driver):

This method is static. It attempts to load the class and returns class instance and takes string type value (driver) after that matches class with given string.

DriverManager:

It is a class of **java.*sql*** package that controls a set of **JDBC** drivers. Each driver has to be register with this class.

getConnection(String url, String userName, String password):

This method establishes a connection to specified database **url**. It takes three string types of arguments like:

url: - Database url where stored or created your database

userName: - User name of MySQL

password: -Password of MySQL

INSERT table_name VALUES(field_values): Above code is used, when you want to insert values in the database table with appropriate value.

con.close():

This method is used for disconnecting the connection. It frees all the resources occupied by the database.

Integrity Constraints

Domain Constraints

- A domain of possible values must be associated with every attribute in the database.
- Declaring an attribute of a particular domain acts as a restraint on the values it can take.
- They are easily tested by the system

EX: cannot set an integer variable to "cat".

Creating New Domains:

The 'create domain' clause allows you to create your own domain types.

- EX: create domain *Dollars* numeric(12,2)
  - These create numerical domains with 12 total digits, two of which are after the decimal point.

CREATE DOMAIN Colour CHAR(15)

CONSTRAINT checkCol CHECK (VALUE IN

('RED','Blue'…))

CREATE TABLE Rainbow(Rorder Int, Rcolour Colour)

- Can have complex conditions in domain check

create domain *AccountType* **char**(10)

constraint *account-type-test*

check (**value in** ('Checking', 'Saving'))

Assertions

- An assertion is a predicate expressing a condition that we wish the database to always satisfy.
- Domain constraints and referential integrity constraints are special forms of assertions.
- Give a boolean condition that must always be true
- No action is permitted that would make the condition false
- EX1: The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

CREATE ASSERTION <name>

CHECK (<condition>)

- The condition can refer to one or several tables
- Often use **EXISTS** or **NOT EXISTS**
- Example: in an Employee table, no employee's bonus should be more than 15% of their salary

CREATE ASSERTION checkSalaryBonus

CHECK (NOT EXISTS (SELECT * FROM EMPLOYEE WHERE (Bonus > 0.15*Salary)))

- When an assertion is created, the system will test it for validity.
- If the assertion is valid, then any future modification to the database is allowed only if it does not cause the assertion to be violated.

Triggers

- A **trigger** is a statement that the system executes automatically as a side effect of a modification to the database.
- A database that has a set of associated triggers is called an **active database**.

A trigger description contains three parts:

Event : A change to the database that activates the trigger.

Condition : A Query or test that is run when the trigger is activated.

Action : A procedure that is executed when the trigger is activated and its condition is true.

- This is referred to as the event-condition-action model of triggers
- The database stores triggers just as if they were regular data.
- This way they are persistent and are accessible to all database operations.
- Once a trigger is entered into the database, the database system takes on the responsibility of executing it whenever the event occurs and the condition is satisfied.

Need for Triggers

- EX: A good use for a trigger would be, for instance, if you own a warehouse and you sell out of a particular item, to automatically re-order that item and automatically generate the order invoice.
- So, triggers are very useful for automating things in your database.

The Syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
BEGIN
 --- sql statements
END;
```

- CREATE [OR REPLACE] TRIGGER trigger_name - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- {BEFORE | AFTER | INSTEAD OF } - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- [OF col_name] - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- CREATE [OR REPLACE] TRIGGER trigger_name - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- [ON table_name] - This clause identifies the name of the table or view to which the trigger is associated.
- [REFERENCING OLD AS o NEW AS n] - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.

- [FOR EACH ROW] - This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
- WHEN (condition) - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

Example of Triggers in SQL

For Example: The price of a product changes constantly. It is important to maintain the history of the prices of the products.

We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'product_price_history' table

```
CREATE TABLE product_price_history
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );

CREATE TABLE product
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );
```

2) Create the price_history_trigger and execute it.

```
CREATE or REPLACE TRIGGER price_history_trigger
BEFORE UPDATE OF unit_price
ON product
FOR EACH ROW
BEGIN
INSERT INTO product_price_history
VALUES
(:old.product_id,
 :old.product_name,
```

```
 :old.supplier_name,
 :old.unit_price);
END;
/
```

3) Lets update the price of a product.

```
UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 100
```

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.

4)If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.

Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.
1) Row level trigger - An event is triggered for each row upated, inserted or deleted.
2) Statement level trigger - An event is triggered for each sql statement executed.

PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.
1) BEFORE statement trigger fires first.
2) Next BEFORE row level trigger fires, once for each row affected.
3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.
4) Finally the AFTER statement level trigger fires.

UNIT III DATA STORAGE

Disks and Files - file organizations - Indexing - Tree structured indexing - Hash Based indexing

STORAGE DATA

Classification of Physical Storage Media

Several types of data storage exist in most computer systems. These can be classified by,

- Speed with which data can be accessed
- Cost per unit of data
- Reliability

We can differentiate storage into:

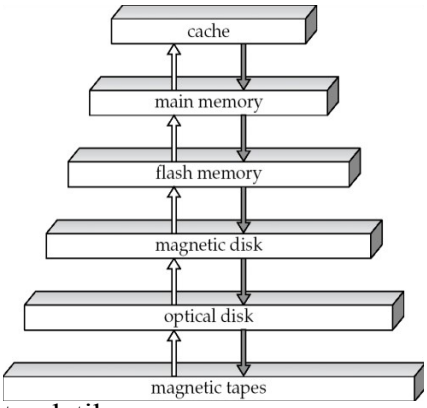
- Volatile storage: It loses contents when power is switched off
- Non-Volatile storage:
  - Contents persist even when power is switched off.
  - Includes secondary and tertiary storage,

- **Cache**
  - fastest and most costly form of storage;
  - small and volatile;
  - managed by the computer system hardware
- **Main memory**:
  - The storage medium used for data that are available to be operated on is main memory

  - generally too small (or too expensive) to store the entire database
  - capacities of up to a few Gigabytes widely used currently
  - **Volatile**
- **Flash memory**
  - Data survives power failure
  - It is a form of EEPROM
  - Data can be written at a location only once, but location can be erased and written to again
    - Can support only a limited number 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 and erase is slower
  - It is primarily used in memory cards, USB flash drives, MP3 players
- **Magnetic-disk**
  - 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
  - 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 ⇒ storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data

Storage Hierarchy



Primary Storage:

- Fastest media but volatile
- E.g. cache, main memory

Secondary Storage:

- also called on-line storage
- hierarchy, non-volatile, moderately fast access time
- E.g. flash memory, magnetic disks

Tertiary storage:

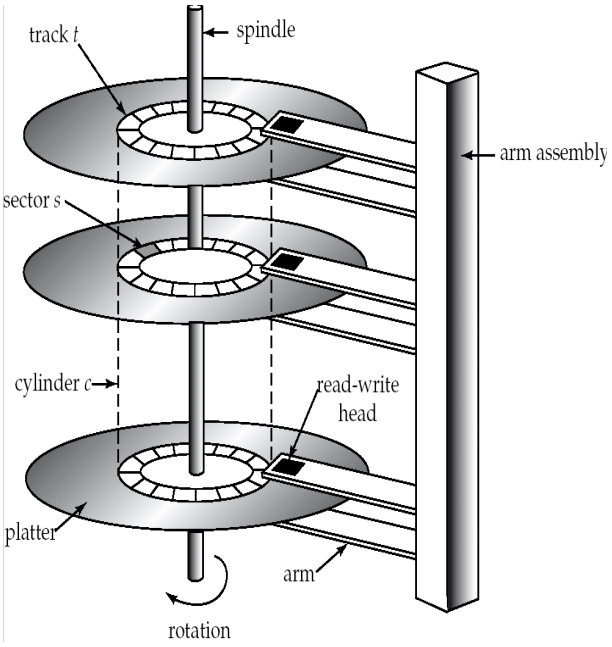
- also called off-line storage
- lowest level in hierarchy, non-volatile, slow access time
- E.g. magnetic tape, optical storage

Magnetic Disks

- Provides the bulk of secondary storage for modern computer system.

Platter :

- Platters are made from rigid metal or glass
- It has a flat circular shape.
- Its two surfaces are covered with a magnetic material and information is recorded on the surfaces.



Read / Write Head :

- Positioned just above the surface of the platter.
- It stores the information on a sector.
- Each side of a platter of a disk has a read/write head
- It moves across the platter to access different tracks.

Tracks :

- The disk surface is divided into tracks.

Sectors :

- Tracks are subdivided into sectors.
- It is the smallest unit of information to read/write to the disk.

Disk arm:

- A disk contains many platters
- The read/write heads of all the tracks are mounted on a single assembly called the disk arm.

Head disk assemblies :

- The disk platters mounted on a **Spindle**.
- The heads mounted on a disk arm are together known as head-disk assemblies.

Cylinder:

- When the head on one platter is on the ith track, the heads on all other platters are also on the ith track of their respective platters.
- So, the ith tracks of all the platters together are called the ith cylinder.

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
- performs **remapping of bad sector**.

Performance Measures of Disks

- The main measures of the qualities of a disk are,
    - Capacity
    - Access time
    - Data-transfer rate and
    - Reliability

Access time :

It is the time from when a read or write request is issued to when data transfer begins.

Seek Time :

It is the time for repositioning the arm over the correct track.

Average Seek time :

It is the average of the seek times, measured over the sequence of random requests.

Rotational latency time :

It is the time it takes for the sector to be accessed to appear under the head.

Data transfer rate :

It is the rate at which data can be retrieved from or stored to the disk.

Mean Time To Failure (MTTF) :

It is a measure of the reliability of the disk.

The MTTF of a disk is the amount of time that, on average we can expect the system to run continuously without any failure.

– Typically 3 to 5 years

RAID

RAID: Redundant Arrays of Independent Disks is a storage technology that combines multiple disk drive components into a logical unit. Data is distributed across the drives in one of several ways called "RAID levels", depending on what level of redundancy and performance is required. It is the disk organization techniques that manage a large numbers of disks, providing a view of a single disk of ,

- **high capacity** and **high speed** by using multiple disks in parallel, and
- **high reliability** by storing data redundantly, so that data can be recovered even if a disk fails
- Disk drives are most vulnerable components with shortest times between failure of any of the hardware components.

One solution is to provide a large disk array comprising an arrangement of several independent disks organized

– to improve reliability and

– increase performance.

- Performance is increased through ***data striping***

Data striping :

The data is segmented into equal-size partitions (the *striping unit*), which are transparently distributed across multiple disks.

Redundancy :

Reliability is improved through storing redundant information across the disks using a *parity* scheme or an *error-correcting* scheme. If a disk failure occurs, the redundant information is used to reconstruct the data on the failed one.

There are two choices,

- Store the redundant information either on a small number of check disks or we can distribute the redundant information uniformly over all disks.
- Using the parity scheme.

Parity Scheme :

A **parity bit** is a bit that is added to ensure that the number of bits with the value one in a set of bits is even or odd. Parity bits are used as the simplest form of error detecting code. The parity bit is set to 1 if the number of ones in a given set of bits is odd. If the number of ones in a given set of bits is already even, it is set to a 0.

Ex:

1010 → Parity bit 0

Error Correcting Scheme :

Error correction is the detection of errors and reconstruction of the original, error-free data.

There are six levels of organizing these disks:

0 -- Non-redundant Striping

1 -- Mirrored Disks

2 -- Memory Style Error Correcting Codes

3 -- Bit Interleaved Parity

4 -- Block Interleaved Parity

5 -- Block Interleaved Distributed Parity

6 -- P + Q Redundancy

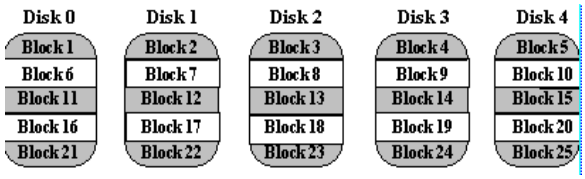
RAID 0 (block-level striping without parity or mirroring)

- RAID level 0 does not provide fault tolerance.
- This level is also known as **disk striping**, because it uses a disk file system called a **stripe set**.
- Data is divided into blocks and is spread in a fixed order among all the disks in the array.
- RAID level 0 improves read and write performance by spreading operations across multiple disks, so that operations can be performed independently.

Minimum number of drives: 2

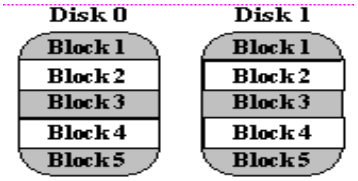
Strengths: Highest performance.

Weaknesses: No data protection; One drive fails, all data is lost.



RAID 1 (mirroring without parity or striping)

- A **RAID 1** creates an exact copy (or **mirror**) of a set of data on two or more disks.
- This is useful when read performance or reliability are more important than data storage capacity.
- RAID level 1 provides fault tolerance.
- Disk mirroring provides a redundant, identical copy of a selected disk.
- All data written to the primary disk is written to the mirror disk.



Minimum number of drives: 2

Strengths: Very high performance; Very high data protection; Very minimal penalty on write performance.

Weaknesses: High redundancy cost overhead; because all data is duplicated, twice the storage capacity is required.

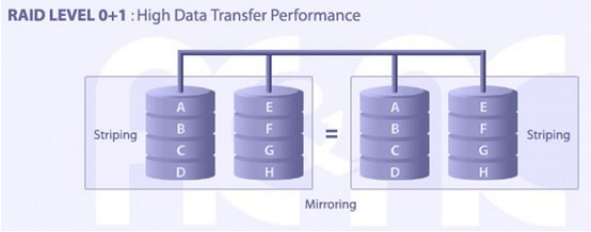
RAID – 0+1

- Combination of RAID 0 (data striping) and RAID 1 (mirroring).
- RAID 10 is another name for RAID (0+1) or RAID 0/1.
- A Mirror of Stripes: Not one of the original RAID levels, two RAID 0 stripes are created, and a RAID 1 mirror is created over them.
- Used for both replicating and sharing data among disks.

Minimum number of drives: 4

Strengths: Highest performance, highest data protection (can tolerate multiple drive failures).

Weaknesses: High redundancy cost overhead; because all data is duplicated, twice the storage capacity is required; requires minimum of four drives.



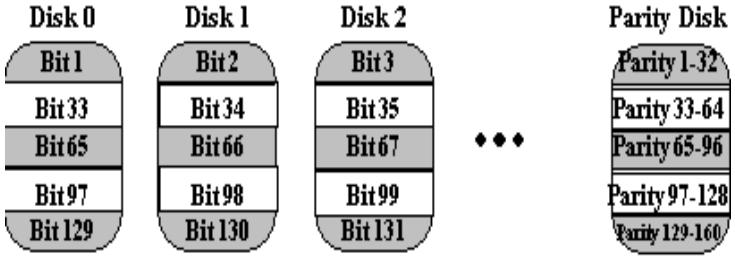
RAID 2 (bit-level striping with dedicated Hamming-code parity)

- RAID level 2 uses error correcting algorithm
- It employs disk-striping strategy
- It **breaks a file into bytes** and spreads it across multiple disks.
- The error-correction method requires several disks.
- RAID level 2 is more advanced than Level 0, because it provides fault tolerance, but is not as efficient as other RAID levels and is not generally used.

RAID 3 (byte-level striping with dedicated parity)

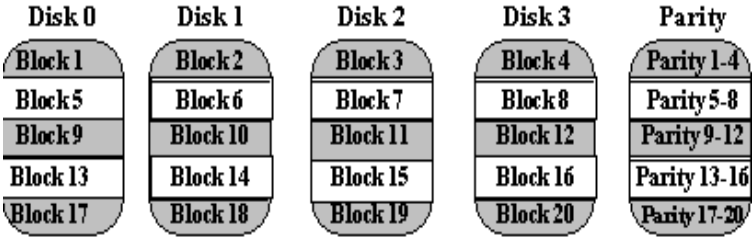
- A RAID 3 uses byte-level striping with a dedicated parity disk.
- It requires only one disk for parity data.

RAID 3 suffers from a write bottleneck, because all parity data is written to a single drive, but provides some read and write performance improvement.



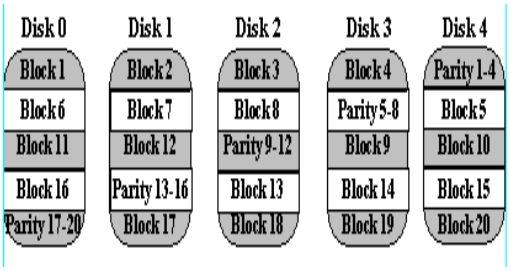
RAID 4 (block-level striping with dedicated parity)

- A **RAID 4** uses block-level striping with a dedicated parity disk.
- This allows each member of the set to act independently when only a single block is requested
- If the disk controller allows it, a RAID 4 set can service multiple read requests simultaneously.



RAID 5 (block-level striping with distributed parity)

- A **RAID 5** uses block-level striping with parity data distributed across all member disks.
- RAID level 5 is known as striping with parity.
- This is the most popular RAID level.
- It is similar to level 4 in that it stripes the data in large blocks across all the disks in the array.
- It differs in that it writes the parity across all the disks. The data redundancy is provided by the parity information.



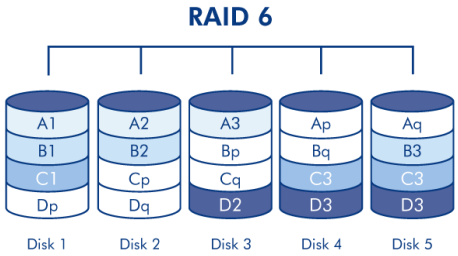
Minimum number of drives: 3

Strengths: Best cost/performance for transaction-oriented networks; Very high performance, very high data protection; Supports multiple simultaneous reads and writes; can also be optimized for large, sequential requests.

Weaknesses: Write performance is slower than RAID 0 or RAID 1.

RAID 6 (block-level striping with double distributed parity)

- Redundancy and data loss recovery capability
- **RAID 6** extends RAID 5 by adding an additional parity block; thus it uses block-level striping with two parity blocks distributed across all member disks.



Choice of RAID

There are three factors that may influence your choice of RAID level:

usability (redundant data),

performance and

cost.

If you do not need usability, RAID-0 can perform best. If usability and performance are more important than cost, RAID-1 or RAID-10(it depends on the number of disks) is a good choice. If price is as important as usability and performance, you can choose RAID-3,RAID-30,RAID-5 or RAID-50(it depends on the transmission type and the number of disk drivers)

Strengths of RAID

- Low cost, little power consumption, high transmission speed.
- Fault-tolerance function.
- Data security.

RAID applies for

- E-Mail server.
- Workgroup/file server.
- Corporation server.
- Storage LAN.
- Internet news server.

Disk Space Management

The role of disk space manager:

- It supports the concept of a ***page*** as a unit of data.
- It manages space on disk and
- It provides commands to allocate or deallocate a page, and read or write page.

The size of a page is chosen to be the size of a disk block. Pages are stored as disk blocks. Reading or writing a page can be done in one disk I/O.

Useful capability:

- This capability is essential for exploiting the advantages of sequential accessing disk blocks.
- The disk space manager hides details of the underlying hardware and allows higher levels of the s/w to think of the data as a collection of pages.
- Allocate a sequence of pages as a contiguous sequence of blocks to hold data.
- It is likely that blocks are initially allocated sequentially on disk, subsequent allocations and deallocations could in general create holes.

Keeping track of free blocks

A database grows and shrinks as records are inserted and deleted over time.

Two ways to keep track of block usage is to maintain:

(1) A list of free blocks

- Deallocating blocks are added to the free list for future use.

(2) Bitmap

- Indicating whether a block is in use or not
- One bit for each disk block
- Allowing fast identification and allocation of contiguous areas on disk

Using OS file systems to manage disk space

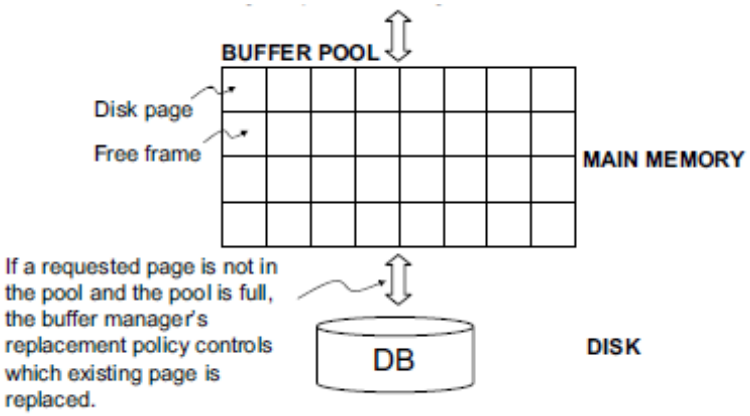
- Operating system supports the abstraction of a *file as a sequence of bytes.*

- OS translates requests " Read byte *i* of file *f* " into " Read block *m* of
- track *t* of cylinder *c* of disk *d.* "
- The entire database could reside in one or more OS files for which a number of blocks are allocated (by the OS) and initialized. The disk space manager is then responsible for managing the space in these OS files.
- DBMS requires portability and may want to access a large single file.

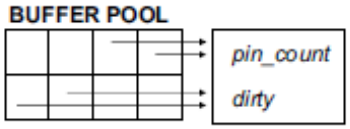
Buffer Management

The role of buffer manager:

- Brings pages from disk to main memory as needed.
- Uses replacement policy to decide which page to replace.
  - Manages the available main memory by partitioning it into a collection of pages or buffer pool. The main memory pages in the buffer pool are called frames.



- *pin_count* and *dirty* variables:



pin_count = the no. of times that the page currently in a given frame has been requested but not released

Incrementing *pin_count* is called ***pinning*** *the page.*

Decrementing *pin_count* is called ***unpinning*** *the page.*

dirty = the boolean variable indicating whether the page has been modified since it was brought into the buffer pool from disk.

Initially, the pin count for every frame is set to 0, and the dirty bits are turned off. When a page is requested the buffer manager does the following:

1. Checks the buffer pool to see if some frame contains the requested page, and if so increments the pin count of that frame. If the page is not in the pool, the buffer manager brings it in as follows:

    (a) Chooses a frame for replacement, using the replacement policy, and increments its pin count.
    (b) If the dirty bit for the replacement frame is on, writes the page it contains to disk (that is, the disk copy of the page is overwritten with the contents of the frame).
    (c) Reads the requested page into the replacement frame.

2. Returns the (main memory) address of the frame containing the requested page to the requestor.

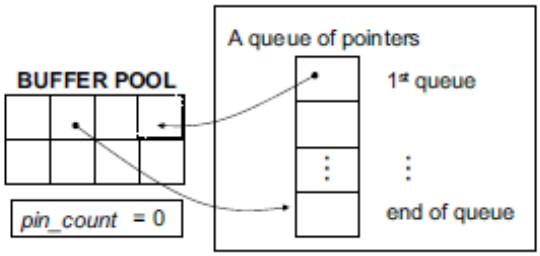
If the requestor has modified the page, it also informs the buffer manager of this at the time that it unpins the page, and the dirty bit for the frame is set. The buffer manager will not read another page into a frame until its pin count becomes 0, that is, until all requestors of the page have unpinned it.

If a requested page is not in the buffer pool, and if a free frame is not available in the buffer pool, a frame with pin count 0 is chosen for replacement. If there are many such frames, a frame is chosen according to the buffer manager's replacement policy.

The buffer replacement policies:

Least recently used (LRU)

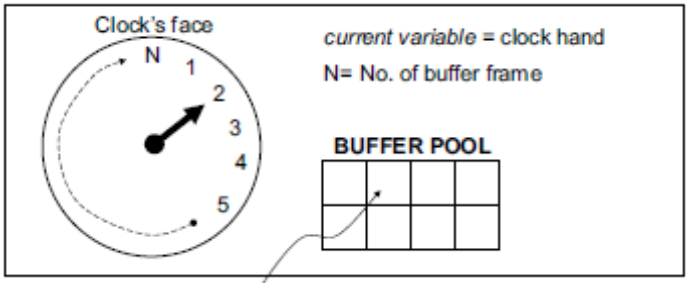
The replacement can affect the time taken for database operations.Different policies are suitable in different situations.



A page that becomes a candidate for replacement

Clock replacement

Current frame is considered for replacement. Similar behavior as LRU, but less overhead.



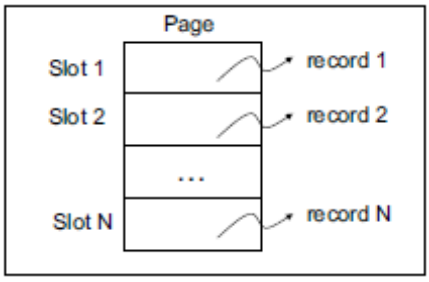
If the frame is not chosen for replacement or pin_count >0, current is incremented and the next frame is considered.

Other replacement policies include:

- First In First Out (FIFO)
- Most Recently Used (MRU)
- Random

Page Formats

- Higher levels of the DBMS see data as a collection of records.
- We consider how a collection of records can be arranged on a page.



- A record is identified by *record id* or *rid* (*page id*, *slot number)*

Alternative approaches to managing slots on a page:

- Fixed-length records
- Variable-length records

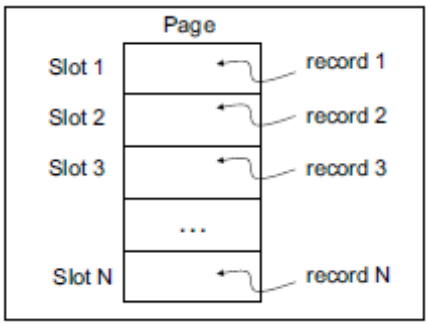
Fixed-length records

- Record slots are uniform in the same length and can be arranged within a page.
- Some slots are occupied or unoccupied by records.
- To keep track of empty slots and to locate all records on a page

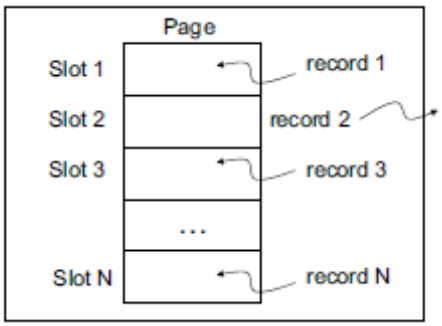
Two alternative hinges:

First alternative:

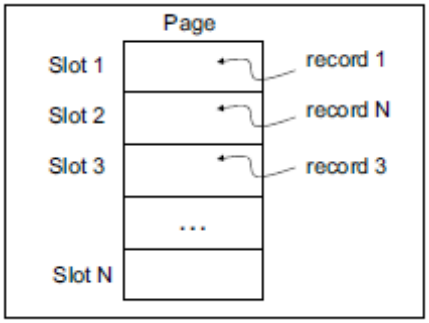
Store records in the first N slots (where N is the number of records on the page).



Whenever a record is deleted, …….

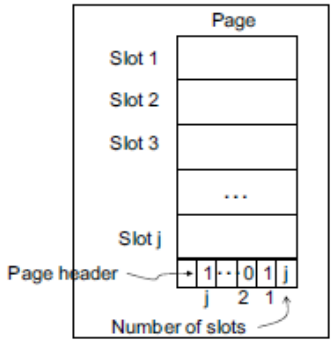


….. we move the last record on the page into the vacated slot.



Second alternative:

- Use an array of bits to handle deletions, one bit per slot.
- To locate records, the page requires scanning the bit array to find slots whose bit is *on*.
- When a record is deleted, its bit is turned off.



Variable-length records

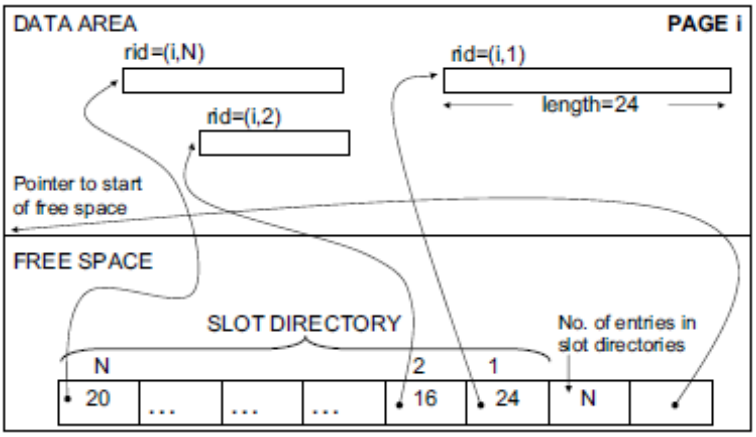
- The page cannot be divided into a fixed collection of slots.
- Problem is that when a new record is to be inserted, we have to find an empty slot of just the right length.

Important rules:

1) To insert a record, we must allocate just the right amount of space for it.

2) To delete a record, we must move records to fill the hole created by the deletion to ensure the contiguousness of all free space.

The most flexible organization for variable-length records is to maintain a ***directory of slots*** (maintain pointers).



Record Formats

- How to organize fields within record.

  We must consider whether the fields of the record are of fixed or variable length.

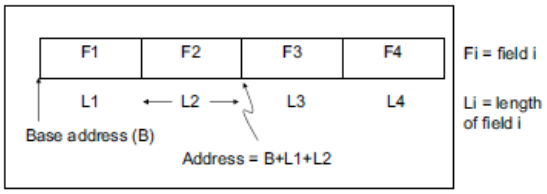
  No. of fields and field types is stored in the system catalog.

- Alternative approaches to organization of records:
  - Fixed-length records
  - Variable-length records

Fixed-length records

- Each field has a fixed length (that is the value in this field is of the same length in all records), and the number of fields is also fixed.
- The fields of such a record can be stored consecutively.
- The address of a particular field can be calculated using information about the lengths of preceding fields, which is available in the system catalog.

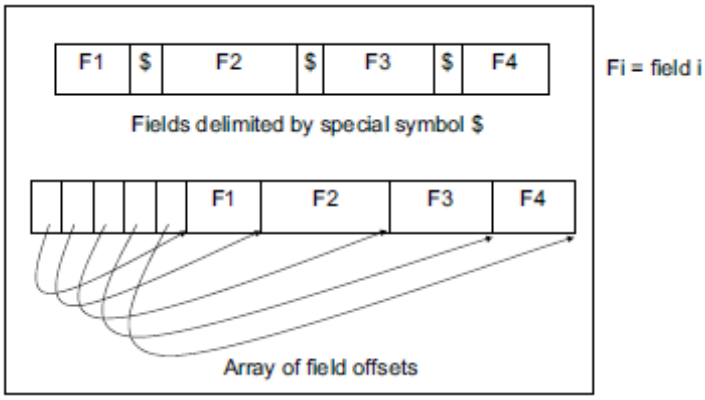
Organization of records with fixed-length fields



Variable-length records

- This organization stores fields consecutively, separated by delimiters.
- It requires a scan of the record in order to locate a desired field.
- At the beginning of a record, some space is reserved for use as an array of integer offsets– the ith integer in this array is the starting address of the ith field value relative to the start of the record.

Organization for variable-length fields



FILE ORGANIZATIONS

A file organization is a way of arranging the records in a file when the file is stored on disk.

Choosing a file organization is a design decision, hence it must be done having in mind the achievement of good performance with respect to the most likely usage of the file. The criteria usually considered important are:

1. Fast access to single record or collection of related recors.
2. Easy record adding/update/removal, without disrupting (1).
3. Storage efficiency.
4. Redundance as a warranty against data corruption.

Three basic file organizations :

Files of randomly ordered records (Heap files)

Files sorted on some field

Files that are hashed on some fields

Cost Model

Performing database operations clearly involves block I/O, the major cost factor. However, we have to additionally pay for CPU time used to search inside a page, compare a record field to a selection constant, etc.

To analyze cost more accurately, we introduce the following parameters:

Parameter	Description
b #	of pages in the file
r #	of records on a page
D	time needed to read/write a disk page
C	CPU time needed to process a record (e.g., compare a field value)
H	CPU time taken to apply a hash function to a record

Comparison of three file organizations

Search Key:

The sequence of fields on which the file is sorted or hashed is called the search key. It can be any sequence of one or more fields; it need not uniquely identify records.

Scan: Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool. There is also a CPU overhead per record for locating the record on the page (in the pool).

Search with equality selection: Fetch all records that satisfy an equality selection, for example, Find the Students record for the student with *sid* 23." Pages that contain qualifying records must be fetched from disk, and qualifying records must be located within retrieved pages.

Search with range selection: Fetch all records that satisfy a range selection, for example, Find all Students records with *name* alphabetically after `Smith.' "

Insert: Insert a given record into the file. We must identify the page in the file into which the new record must be inserted, fetch that page from disk, modify it to include the new record, and then write back the modified page. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

Delete: Delete a record that is specified using its rid. We must identify the page that contains the record, fetch it from disk, modify it, and write it back. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

HEAP files

It's the simplest possible organization: the data are collected in the file in the order in which they arrive, and it's not even required that the records have a common format across the file (different fields/sizes, same fields in different orders, etc.are possible). This implies that each record/field must be self-describing. Despite the obvious storage efficiency and the easy update, it's quite clear that this ``structure'' is not suited for easy data retrieval, since retrieving a datum basically requires detailed analysis of the file content. It makes sense only as temporary storage for data to be later structured in some way.

Sorted Files

In this scheme, all the records have the same size and the same field format, with the fields having fixed size as well. The records are sorted in the file according to the content of a field of a scalar type, called ``key''. The key must identify uniquely a records, hence different record have different keys.

Hashed

As with sequential or indexed files, a key field is required for this organization, as well as fixed record length. However, no explicit ordering in the keys is used for the hash search, other than the one implicitly determined by a hash function.

Operations

Scan

Heap file

Scanning the records of a file involves reading all b pages as well as processing each of the r records on each page:

$$\text{Scan}_{\text{heap}} = b \cdot (D + r \cdot C)$$

Sorted file

The sort order does not help much here. However, the scan retrieves the records in sorted order (which can be big plus):

$$\text{Scan}_{\text{sort}} = b \cdot (D + r \cdot C)$$

Hashed file

Again, the hash function does not help. We simply scan from the beginning (skipping over the spare free space typically found in hashed files):

$$\text{Scan}_{\text{hash}} = 1.25 \cdot b \cdot (D + r \cdot C)$$

Search with equality test (A = const)

Heap file

The equality test is (a) on a primary key, (b) not on a primary key:

(a) $\text{Search}_{heap} = 1/2 \cdot b \cdot (D + r \cdot C)$

(b) $\text{Search}_{heap} = b \cdot (D + r \cdot C)$

Sorted file (sorted on A)

We assume the equality test to be on the field determining the sort order. The sort order enables us to use binary search:

$\text{Search}_{sort} = \log2\ b \cdot D + \log2\ r \cdot C$

(If more than one record qualifies, all other matches are stored right after the first hit.)

Hashed file (hashed on A)

Hashed files support equality searching best. The hash function directly leads us to the page containing the hit (overflow chains ignored here):

(a) $\text{Search}_{hash} = H + D + 1/2 \cdot r \cdot C$

(b) $\text{Search}_{hash} = H + D + r \cdot C$

(All qualifying records are on the same page or, if present, in its overflow chain.)

Search with range selection (A >= lower AND A<=upper)

Heap file

Qualifying records can appear anywhere in the file:

$\text{Range}_{heap} = b \cdot (D + r \cdot C)$

Sorted file (sorted on A)

Use equality search (with A = lower) then sequentially scan the file until a record with A > upper is found:

$\text{Range}_{sort} = \log2\ b \cdot D + \log2\ r \cdot C + [n/r] \cdot D + n \cdot C$

(n denotes the number of hits in the range)

Hashed file (sorted on A)

Hashing offers no help here as hash functions are designed to scatter records all over the hashed file (e.g., h (h7, . . . i) = 7, h (h8, . . . i) = 0) :

$\text{Range}_{hash} = 1.25 \cdot b \cdot (D + r \cdot C)$

Insert

Heap file

We can add the record to some arbitrary page (e.g., the last page). This involves reading and writing the page:

$\text{Insert}_{heap} = 2 \cdot D + C$

Sorted file

On average, the new record will belong in the middle of the file. After insertion, we have to shift all subsequent records (in the latter half of the file):

$\text{Insert}_{sort} = \log2\ b \cdot D + \log2\ r \cdot C + 1/2 \cdot b \cdot (2 \cdot D + r \cdot C)$

Hashed file

We pretend to search for the record, then read and write the page determined by the hash function (we assume the spare 20% space on the page is sufficient to hold the new record):

$\text{Insert}_{hash} = H + D + C + D$

Delete (record specified by its rid)

Heap file

If we do not try to compact the file (because the file uses free space management) after we have found and removed the record, the cost is:

$\text{Delete}_{heap} = D + C + D$

Sorted file

Again, we access the record's page and then (on average) shift the latter half the file to compact the file:

$\text{Delete}_{sort} = D + 1/2 \cdot b \cdot (2 \cdot D + r \cdot C)$

Hashed file

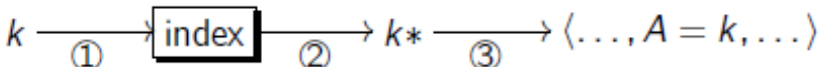
Accessing the page using the rid is even faster than the hash function, so the hashed file behaves like the heap file:

$\text{Delete}_{hash} = D + C + D$

Overview of Indexes

- If the basic organization of a file does not support a particular operation, we can additionally maintain an auxiliary structure, an index, which adds the needed support.
- We will use indexes like guides. Each guide is specialized to accelerate searches on a specific attribute A (or a combination of attributes) of the records in its associated file:
  - Query the index for the location of a record with A = k (k is the search key),
  - The index responds with an associated index entry k* (k* contains enough information to access the actual record in the file),

– Read the actual record by using the guiding information in k*; the record will have an A-field with value k



We can design the index entries, i.e., the k_, in various ways:

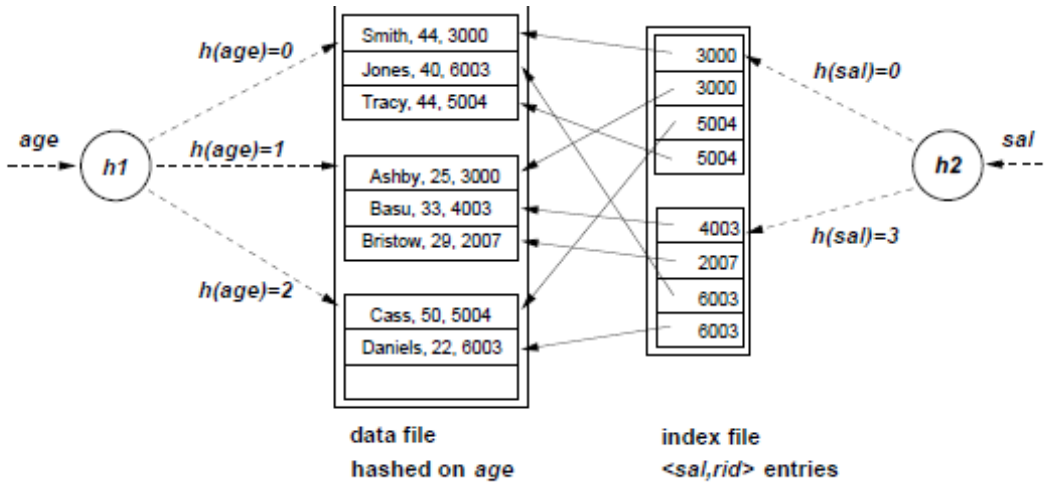
Variant	Index entry $k*$
ⓐ	$\langle k, \langle \ldots, A = k, \ldots \rangle \rangle$
ⓑ	$\langle k, rid \rangle$
ⓒ	$\langle k, [rid_1, rid_2, \ldots] \rangle$

Remarks:

- With variant a , there is no need to store the data records in addition the index—the index itself is a special file organization.
- If we build multiple indexes for a file, at most one of these should use variant a to avoid redundant storage of records.
- Variants b and c use rid(s) to point into the actual data file.
- Variant c leads to less index entries if multiple records match a search key k, but index entries are of variable length.

Example:

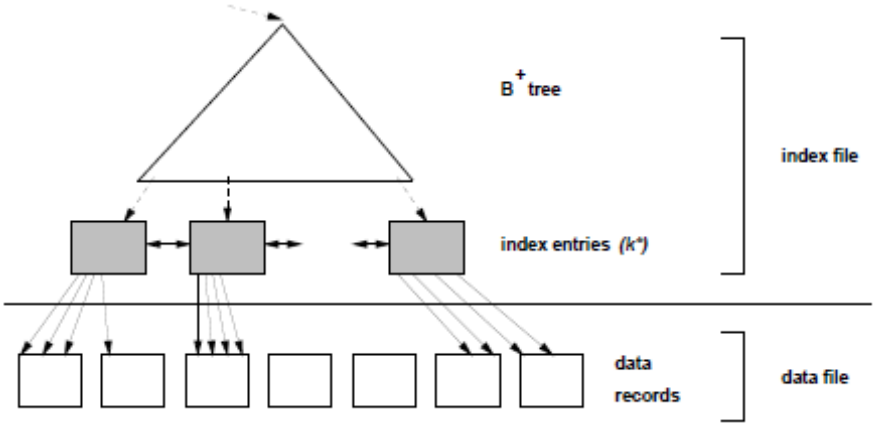
- ✓ The data file contains (name, age, sal) records, the file itself (index entry variant a) is hashed on field age (hash function h1).
- ✓ The index file contains (sal, rid) index entries (variant b), pointing into the data file.
- ✓ This file organization + index efficiently supports equality searches on the age and sal keys.



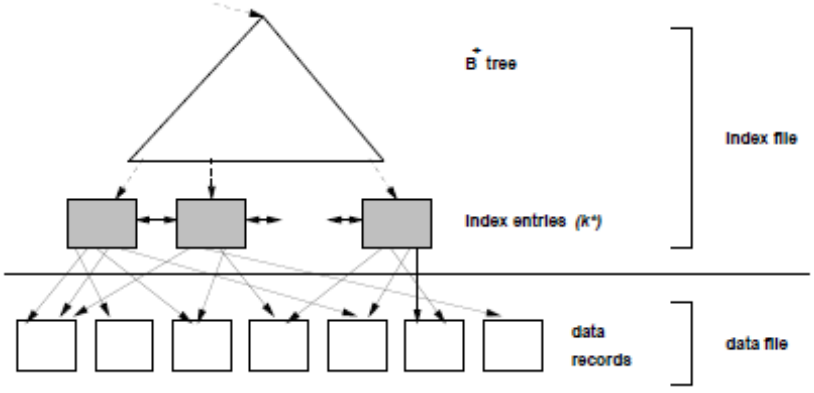
Clustered vs. Unclustered indexes

- Suppose, we have to support range selections on records such that lower <= A<= upper for field A.
- If we maintain an index on the A-field, we can
  - query the index once for a record with A = lower , and then
  - sequentially scan the data file from there until we encounter a record with field A > upper.

- This will work provided that the data file is sorted on the field A:



- If the data file associated with an index is sorted on the index search key, the index is said to be clustered.
- In general, the cost for a range selection grows tremendously if the index on A is unclustered. In this case, proximity of index entries does not imply proximity in the data file.
  - As before, we can query the index for a record with A = lower . To continue the scan, however, we have to revisit the index entries which point us to data pages scattered all over the data file:



Remarks:

If the index entries (k*) are of variant a , the index is obviously clustered by definition.

A data file can have at most one clustered index (but any number of unclustered indexes).

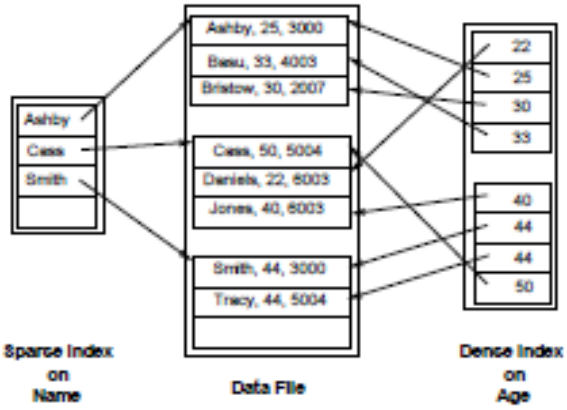
Dense vs. Sparse indexes

- A clustered index comes with more advantages than the improved speed for range selections presented above. We can additionally design the index to be space efficient:
    - To keep the size of the index file small, we maintain one index entry k* per data file page (not one index entry per data record). Key k is the smallest search key on that page.
    - Indexes of this kind are called sparse (otherwise indexes are dense).
- To search a record with field A = k in a sparse A-index, we
    - locate the largest index entry k'_ such that k' _ k, then
    - access the page pointed to by k'_, and
    - scan this page (and the following pages, if needed) to find records with h. . . ,A = k, . . . i.

    Since the data file is clustered (i.e., sorted) on field A, we are guaranteed to find matching records in the proximity.

Example:

- Again, the data file contains hname, age, sali records. We maintain a clustered sparse index on field name and an unclustered dense index on field age. Both use index entry variant b to point into the data file:



Remarks:

Sparse indexes need 2–3 orders of magnitude less space than dense indexes.

We cannot build a sparse index that is unclustered (i.e., there is at most one sparse index per file).

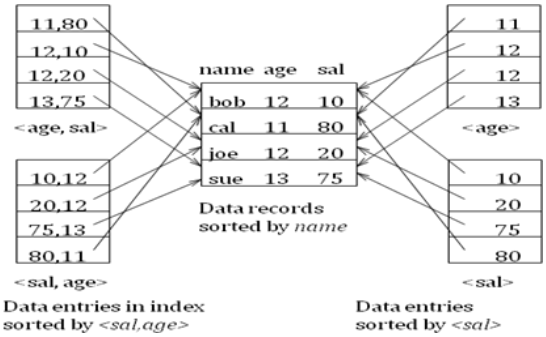
Primary vs. secondary indexes

An index on a set of fields that includes the primary key is called a primary index. An index that is not a primary index is called a secondary index.

Index on Composite Search Keys:

Search on a combination of fields.

- **Equality query**: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
  - age=20 and sal =75
- **Range query**: Some field value is not a constant. E.g.:
  - age =20; or age=20 and sal > 10



TREE STRUCTURED INDEXING

Introduction

As for any index, 3 alternatives for data entries k*:

Data record with key value k

<k, rid of data record with search key value k>

<k, list of rids of data records with search key k>

Choice is orthogonal to the indexing technique used to locate data entries k*.

Tree-structured indexing techniques support both range searches and equality searches.

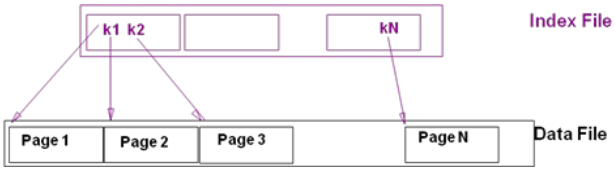
- ISAM is a static structure.
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
  - High fanout (**F**) means depth rarely more than 3 or 4.

Range Searches

- ``Find all students with gpa > 3.0''

- If data is in sorted file, do binary search to find first such student, then scan to find others.
- Cost of binary search can be quite high.

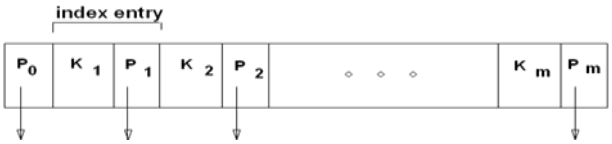
Simple idea: Create an `index' file.



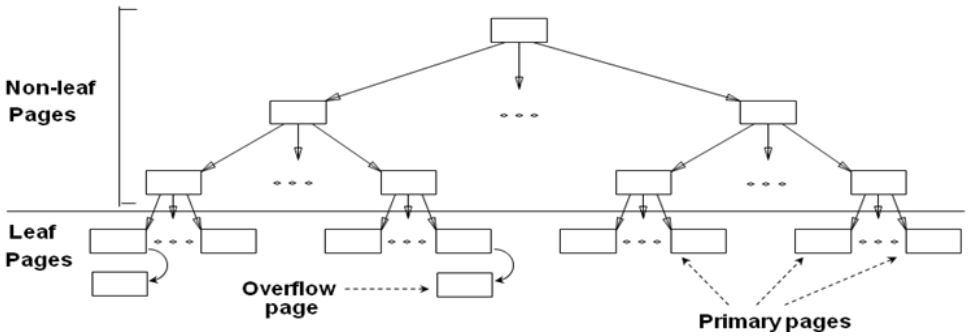
Indexed Sequential Access Method

ISAM stands for Indexed Sequential Access Method, a method for indexing data for fast retrieval. In an ISAM system, data is organized into records which are composed of fixed length fields. Records are stored sequentially, originally to speed access on a tape system. A secondary set of hash tables known as indexes contain "pointers" into the tables, allowing individual records to be retrieved without having to search the entire data set. This is a departure from the contemporaneous navigational databases, in which the pointers to other data were stored inside the records themselves. The key improvement in ISAM is that the indexes are small and can be searched quickly, thereby allowing the database to access only the records it needs. Additionally modifications to the data do not require changes to other data, only the table and indexes in question.

When an ISAM file is created, index nodes are fixed, and their pointers do not change during inserts and deletes that occur later (only content of leaf nodes change afterwards). As a consequence of this, if inserts to some leaf node exceed the node's capacity, new records are stored in overflow chains. If there are many more inserts than deletions from a table, these overflow chains can gradually become very large, and this affects the time required for retrieval of a record.

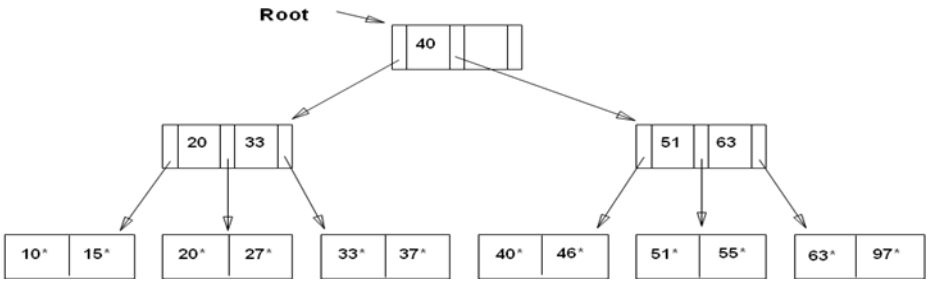


- Index file may still be quite large. But we can apply the idea repeatedly!

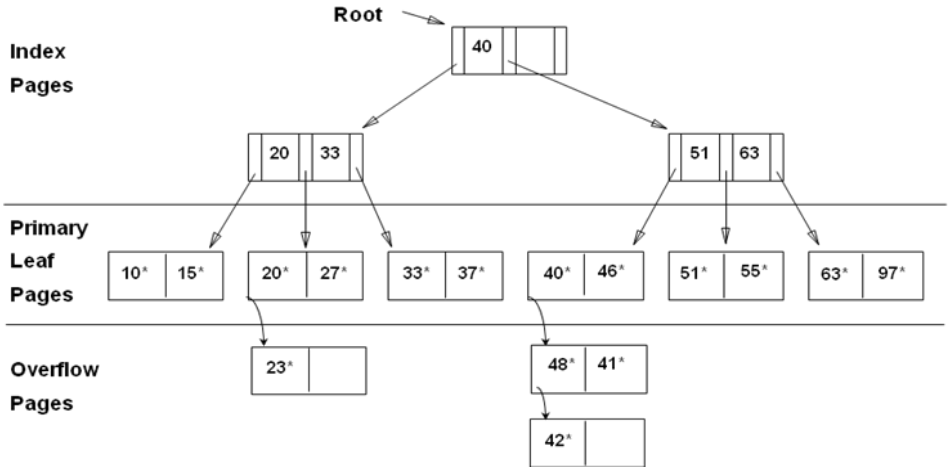


Example ISAM Tree

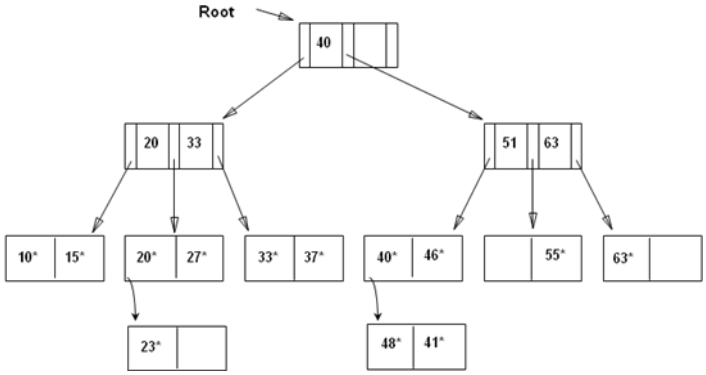
- Each node can hold 2 entries; no need for `next-leaf-page' pointers.



After Inserting 23*, 48*, 41*, 42* ...



... Then Deleting 42*, 51*, 97*

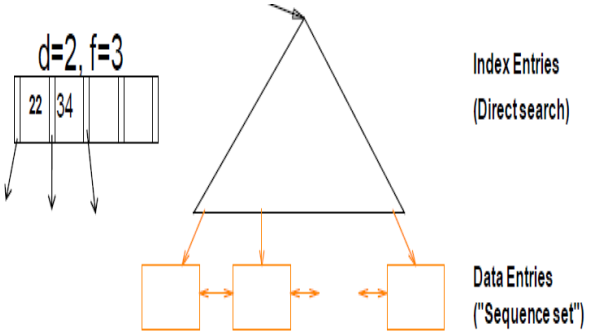


B+ TREES

A **B+ tree** is a type of tree which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a *key*. It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index

segment (usually called a "block" or "node"). In a B+ tree, in contrast to a B-tree, all records are stored at the leaf level of the tree; only keys are stored in interior nodes.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context—in particular, file systems. This is primarily because unlike binary search trees, B+ trees have very high fan-out (typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.



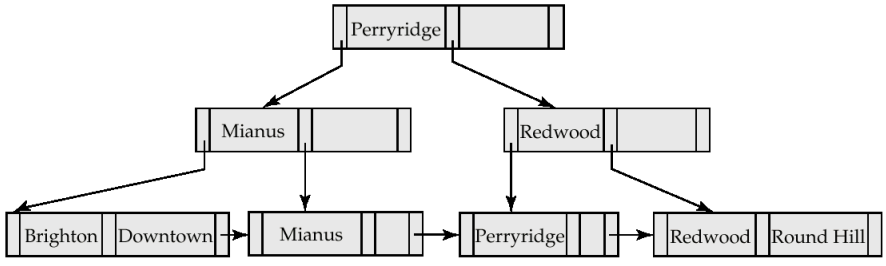
Structure of a B+ Tree

B+-Tree Node Structure

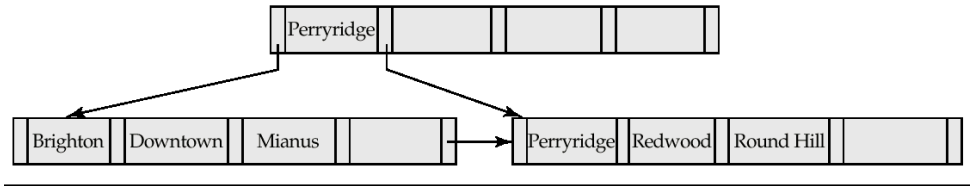
- Typical node

  - » Ki are the search-key values
  - » Pi are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered K1 < K2 < K3 < … < Kn-1

Examples of a B+-tree



B+-tree for account file (n=3)



B+-tree for account file (n=5)

Queries on B+-Trees

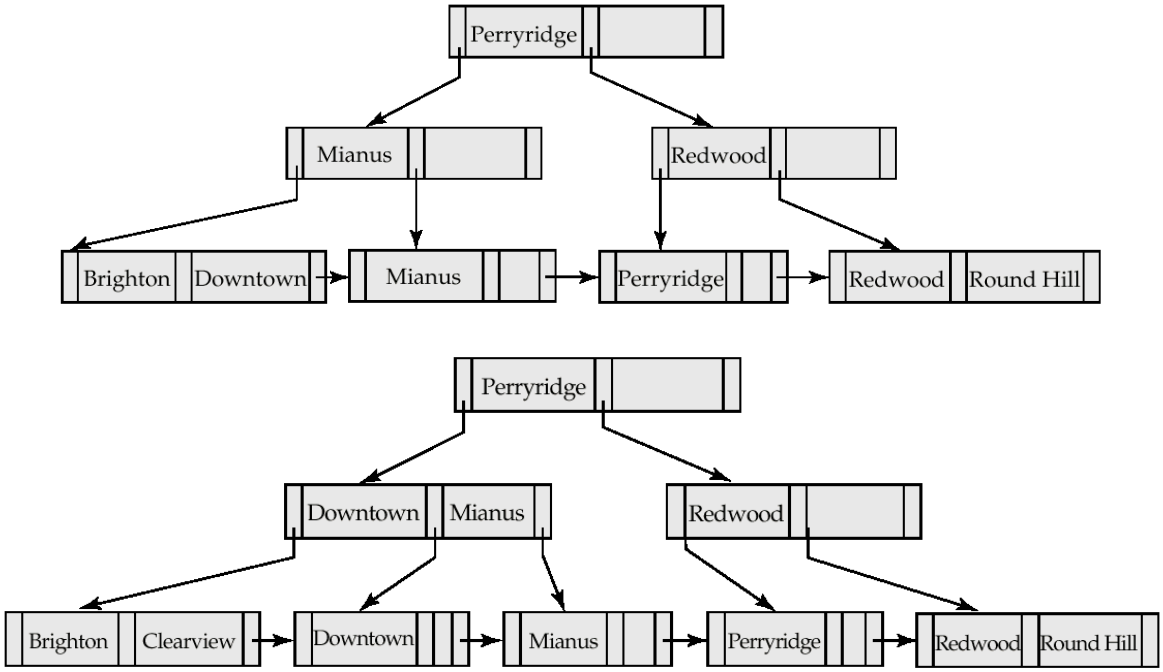
- **Find all records with a search-key value of k.**
  - » Start with the root node
  - » Examine the node for the smallest search-key value > k.
  - » If such a value exists, assume it is Ki. Then follow Pi to the child node
  - » Otherwise k ≥ Km-1, where there are m pointers in the node, Then follow Pm to the child node.
  - » If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
  - » Eventually reach a leaf node. If key Ki = k, follow pointer Pi to the desired record or bucket. Else no record with search-key value k exists.

Updates on B+-Trees: Insertion

- Find the leaf node in which the search-key value would appear
- If the search-key value is already there in the leaf node, record is added to file and if necessary pointer is inserted into bucket.
- If the search-key value is not there, then add the record to the main file and create bucket if necessary. Then:
  - » if there is room in the leaf node, insert (search-key value, record/bucket pointer) pair into leaf node at appropriate position.
  - » if there is no room in the leaf node, split it and insert (search-key value, record/bucket pointer) pair.

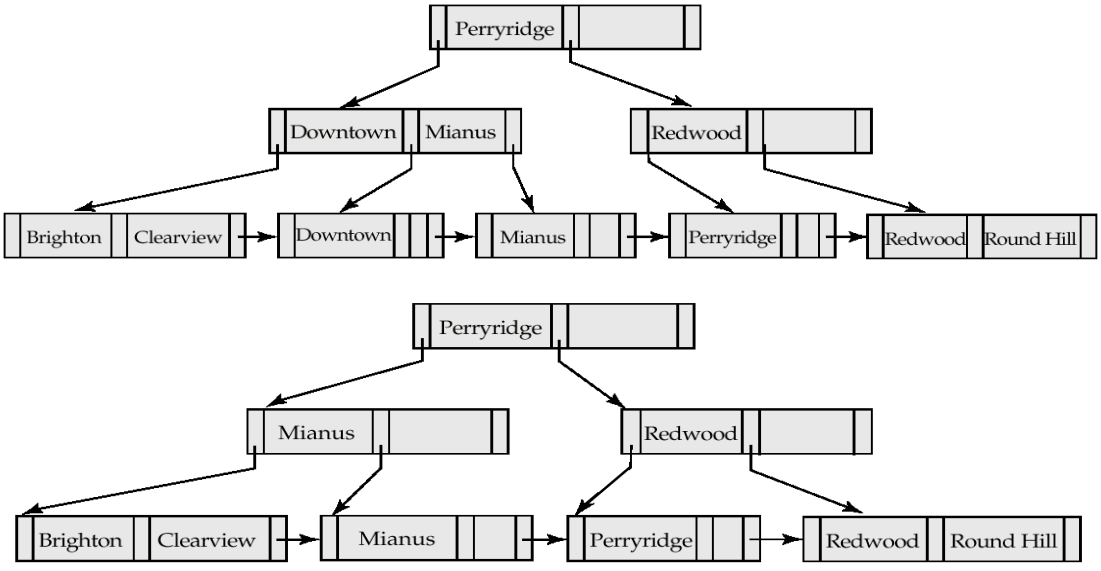
Splitting a node:

i.e. we are going to insert a node with a search key value “Clearview”. We find that “Clearview” should be in the node with Brighton and Downtown, so we must split the node. i.e. n = 3

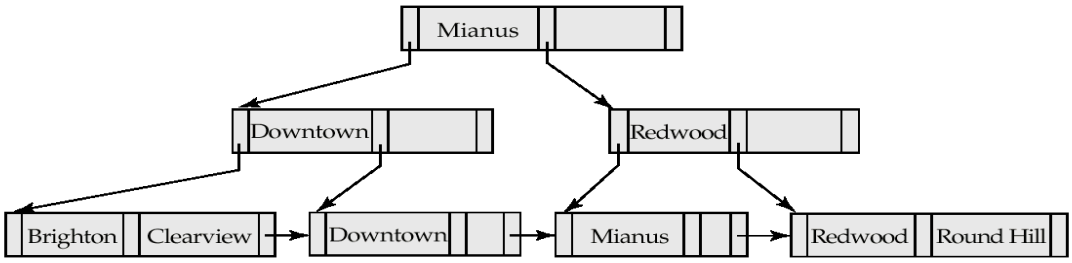


B+-Tree before and after insertion of "Clearview"

Updates on B+-Trees:Deletion



Result after deleting "Downtown" from account

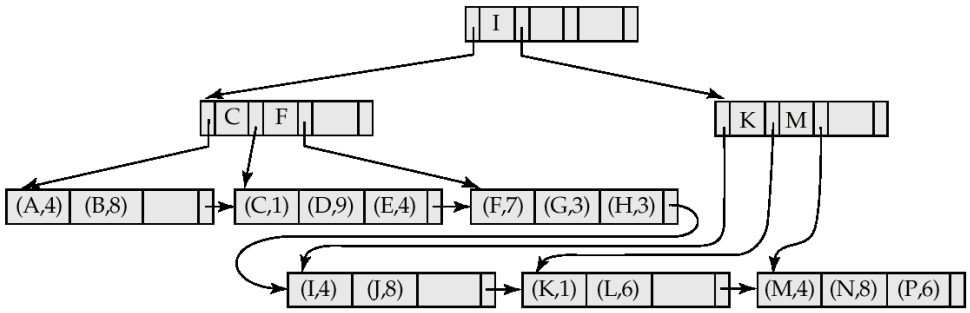


Deletion of "Perryridge" instead of "Downtown"

B+-Tree File Organization

- Index file degradation problem is solved by using B+-Tree indices. Data file degradation problem is solved by using B+-Tree File Organization.
- The leaf nodes in a B+-tree file organization store records, instead of pointers.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Leaf nodes are still required to be half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+-tree index.

Example of B+-tree File Organization



HASH BASED INDEXING

STATIC HASHING

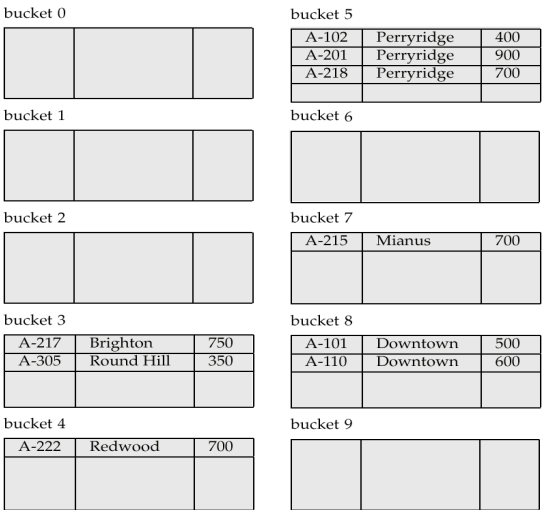
- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- **Hash function** *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B*.

- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

Hash file organization of *account* file, using *branch-name* as key,

- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i.*
- The hash function returns the sum of the binary representations of the characters modulo 10
    - E.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3
    - ie (Brighton = (2+18+9+7+8+20+14+15)%10=3)



Hash Functions

- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.

Handling of Bucket Overflows

- Bucket overflow can occur because of
    - **Insufficient buckets –** The number of buckets should be chosen such that $n_B > n_r / f_r$ where,

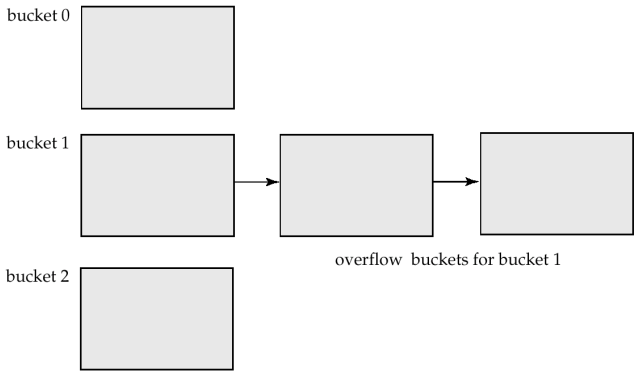
        n_B – Number of Buckets

        n_r – Total number of records

        f_r – Number of records that will fit in a bucket

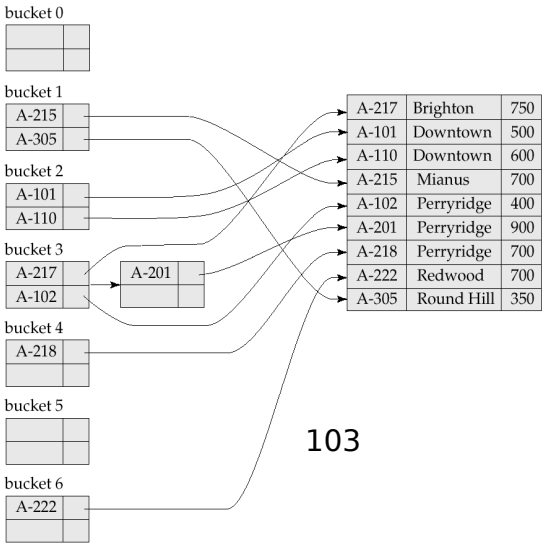
- **Skew** in distribution of records. This can occur due to two reasons:
  - multiple records have same search-key value
  - chosen hash function produces non-uniform distribution of key values

- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using **overflow buckets**.
- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list. This scheme is called **closed hashing.**
- An alternative, called **open hashing**, which does not use overflow buckets. If a bucket is full, the system inserts records in some other buckets. One policy is to use the next bucket that has space is known as **Linear Probing.** But it is not suitable for database applications.



Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.



The hash function in the above figure computes the sum of the digits of the account number modulo 7.(ie account number 215 - > (2+1+5 % 7 = 1)

Disadvantages:

- Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
- If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
- If database shrinks, again space will be wasted.
- One option is periodic re-organization of the file with a new hash function, but it is very expensive.

Dynamic Hashing:

Most databases grow larger over time. If we are to use static hashing for such a database, we have three classes of options:

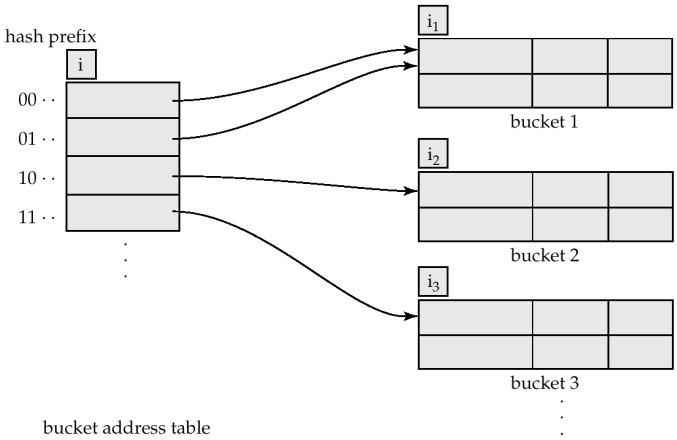
1. Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.

2. Choose a hash function based on the anticipated size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.

3. Periodically reorganize the hash structure in response to file growth.Such a reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments. This reorganization is a massive, time-consuming operation. Furthermore, it is necessary to forbid access to the file during reorganization.

- This is an alternative approach, which allows the file size to change to accommodate growth and shrinkage of the database. One type of dynamic hashing is **extendible/extendable hashing.** Extendable hashing splits and combines blocks as the database grows or shrinks. This ensures efficient space utilization. Moreover, since reorganization involves only one block at a time the associated overhead is minimal.
- **Extendable hashing** – one form of dynamic hashing

General Extendable Hash Structure



Example

- Assume that the hash function $h(k)$ returns a binary number. The first i bits of each string will be used as indices to figure out where they will go in the "directory" (hash table). Additionally, i is the smallest number such that the first i bits of all keys are different.

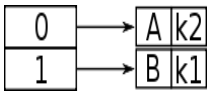
Keys to be used:

$h(k_1)$ = 100100

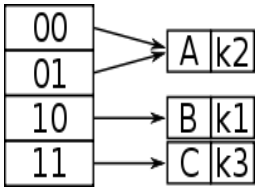
$h(k_2)$ = 010110

$h(k_3)$ = 110110

Let's assume that for this particular example, the bucket size is 1. The first two keys to be inserted, k_1 and k_2, can be distinguished by the most significant bit, and would be inserted into the table as follows:



Now, if k_3 were to be hashed to the table, it wouldn't be enough to distinguish all three keys by one bit (because k_3 and k_1 have 1 as their leftmost bit. Also, because the bucket size is one, the table would overflow. Because comparing the first two most significant bits would give each key a unique location, the directory size is doubled as follows:



And so now k_1 and k_3 have a unique location, being distinguished by the first two leftmost bits. Because k_2 is in the top half of the table, both 00 and 01 point to it because there is no other key to compare to that begins with a 0.

$h(k_4) = 011110$

Now, k_4 needs to be inserted, and it has the first two bits as 01..(1110), and using a 2 bit depth in the directory, this maps from 01 to Bucket A. Bucket A is full (max size 1), so it must be split; because there is more than one pointer to Bucket A, there is no need to increase the directory size.

What is needed is information about:

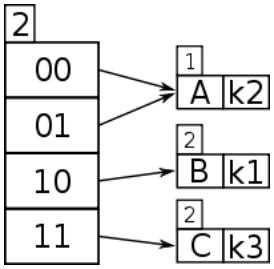
1. The key size that maps the directory (the global depth), and
2. The key size that has previously mapped the bucket (the local depth)

In order to distinguish the two action cases:

1. Doubling the directory when a bucket becomes full
2. Creating a new bucket, and re-distributing the entries between the old and the new bucket

Examining the initial case of an extendible hash structure, if each directory entry points to one bucket, then the local depth should be equal to the global depth. The number of directory entries is equal to $2^{\text{global depth}}$, and the initial number of buckets is equal to $2^{\text{local depth}}$. Thus if global depth = local depth = 0, then $2^0 = 1$, so an initial directory of one pointer to one bucket.

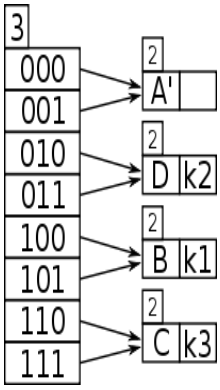
- If the local depth is equal to the global depth, then there is only one pointer to the bucket, and there is no other directory pointers that can map to the bucket, so the directory must be doubled (case1).
- If the bucket is full, if the local depth is less than the global depth, then there exists more than one pointer from the directory to the bucket, and the bucket can be split (case 2).



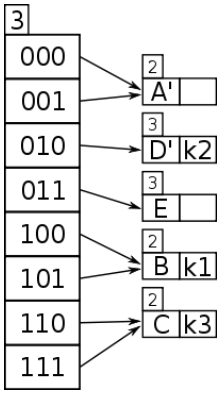
Key 01 points to Bucket A, and Bucket A's local depth of 1 is less than the directory's global depth of 2, which means keys hashed to Bucket A have only used a 1 bit prefix (i.e. 0), and the bucket needs to have its contents split using keys 1 + 1 = 2 bits in length; in general, for any local depth d where d is less than D, the global depth, then d must be incremented after a bucket split, and the new d used as the number of bits of each entry's key to redistribute the entries of the former bucket into the new buckets.

Now, $h(k_4)$= 011110 is tried again, with 2 bits 01.., and now key 01 points to a new bucket but there is still k2 in it ($h(k2)$ = 010110 and also begins with 01).If k2 had been 000110, with key 00, there would have been no problem, because k2 would have been empty.

So Bucket D needs to be split, but a check of its local depth, which is 2, is the same as the global depth, which is 2, so the directory must be split again, in order to hold keys of sufficient detail, e.g. 3 bits.



1. Bucket D needs to split due to being full.
2. As D's local depth = the global depth, the directory must double to increase bit detail of keys.
3. Global depth has incremented after directory split to 3.
4. The new entry k4 is rekeyed with global depth 3 bits and ends up in D which has local depth 2, which can now be incremented to 3 and D can be split to D' and E.
5. The contents of the split bucket D, k2, has been re-keyed with 3 bits, and it ends up in D.
6. K4 is retried and it ends up in E which has a spare slot.



Now, $h(k2)$= 010110 is in D and $h(k_4)$= 011110 is tried again, with 3 bits 011.., and it points to bucket D which already contains k2 so is full; D's local depth is 2 but now the global depth is 3 after the directory doubling, so now D can be split into bucket's D' and E, the contents of D, k2 has its $h(k2)$retried with a new global depth bitmask of 3 and k2 ends up in D', then the

new entry k4 is retried with $h(k_4)$bit masked using the new global depth bit count of 3 and this gives 011 which now points to a new bucket E which is empty. So K4 goes in Bucket E.

Linear hashing

Linear hashing is a dynamic hash table algorithm. Linear hashing allows for the expansion of the hash table one slot at a time. The frequent single slot expansion can very effectively control the length of the collision chain. The cost of hash table expansion is spread out across each hash table insertion operation, as opposed to being incurred all at once. Linear hashing is therefore well suited for interactive applications.

Hash table is a data structure that associates keys with values.

- Full buckets are not necessarily split
- Buckets split are not necessarily full
- Every bucket will be split sooner or later and so all Overflows will be reclaimed and rehashed.
- Split pointer s decides which bucket to split
    - *s* is independent to overflowing bucket
    - At level *i*, *s* is between 0 and 2^i
    - *s* is incremented and if at end, is reset to 0.
- $h_i(k) = h(k) \bmod (2^i n)$
- h_{i+1} doubles the range of h_i

Insertion and Overflow condition

Algorithm for inserting 'k':

1. b = h0(k)
2. if b < split-pointer then
3. b = h1(k)

Searching in the hash table for 'k':

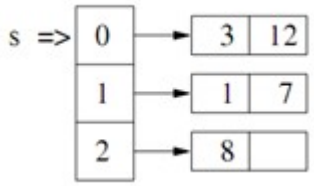
1. b = h0(k)
2. if b < split-pointer then
3. b = h1(k)
4. read bucket b and search there

Example:

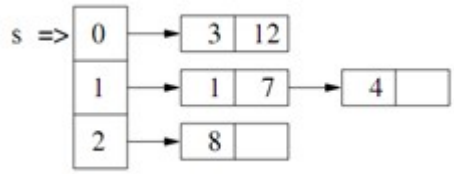
- In the following M=3 (initial # of buckets)
- Each bucket has 2 keys. One extra key for overflow.

- s is a pointer, pointing to the split location. This is the place where next split should take place.
- Insert Order: 1,7,3,8,12,4,11,2,10,13

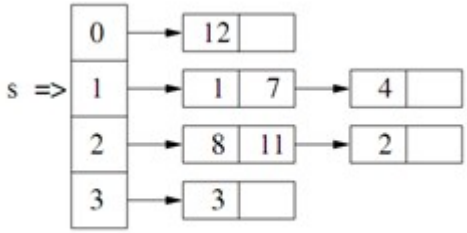
After insertion till 12:



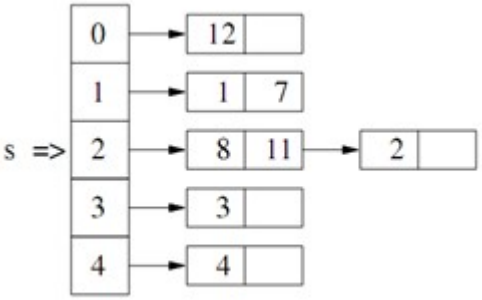
When 4 inserted overflow occurred. So we split the bucket (no matter it is full or partially empty). And increment pointer.



So we split bucket 0 and rehashed all keys in it. Placed 3 to new bucket as (3 mod 6 = 3) and (12 mod 6 = 0). Then 11 and 2 are inserted. And now overflow. *s* is pointing to bucket 1, hence split bucket 1 by re- hashing it.

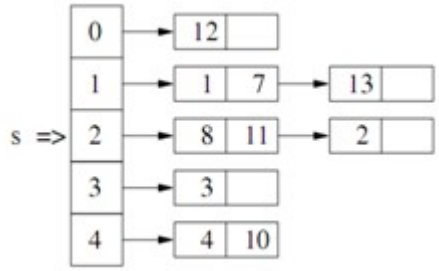


After split:

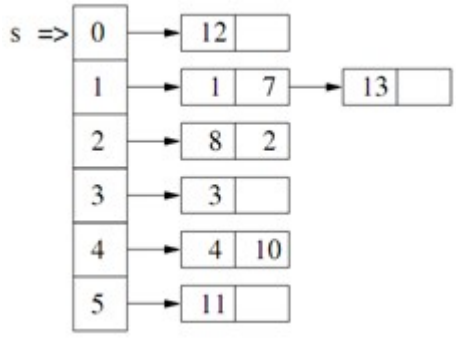


Insertion of 10 and 13: as (10 mod 3 = 1) and bucket 1 < s, we need to hash 10 again using h1(10) = 10 mod 6 = 4th bucket.

When 13 is inserted same process is done, but it end up to the same bucket. But here is an overflow, we need to split 2nd bucket.



Here is the final hash table.



s is moved to the top again as one cycle is completed. Now *s* will travel from 0 to 5th bucket, then 0 to 12, etc;

Extendible Hashing Versus Linear Hashing

The following comparison factors are used to compare the performance of linear hashing with extendible hashing:

1. storage utilization;
2. average unsuccessful search cost;
3. average successful search cost;
4. split cost;
5. insertion cost;
6. number of overflow buckets.

The simulation is conducted with the bucket sizes of 10, 20, and 50 for both hashing techniques. In order to observe their average behavior, the simulation uses 50,000 keys which have been generated randomly. According to our simulation results, extendible hashing has an advantage of 5% over linear hashing in terms of storage utilization. Successful search, unsuccessful search, and insertions are less costly in linear hashing. However, linear hashing requires a large overflow space to handle the overflow records. Simulation shows that approximately 10% of the space should be marked as overflow space

in linear hashing. Directory size is a serious bottleneck in extendible hashing. Based on the simulation results, the authors recommend linear hashing when main memory is at a premium.

UNIT IV QUERY EVALUATION AND DATABASE DESIGN

External sorting - Query evaluation - Query optimization - Schema refinement and normalization - Physical database design and tuning - Security

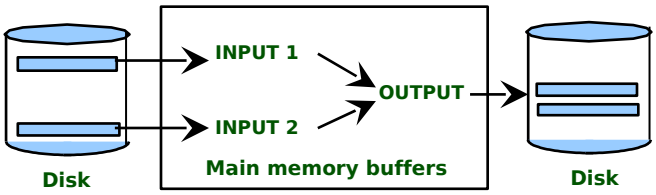
EXTERNAL SORTING

Why sort

- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing *gpa* order
- Sorting is first step in *bulk loading* B+ tree index.
- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)
- *Sort-merge* join algorithm involves sorting.

2-Way Sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.



Streaming Data Through RAM

Simple case:

– Compute f(x) for each record, write out the result

– Read a page from INPUT to Input Buffer

– Write f(x) for each item to Output Buffer

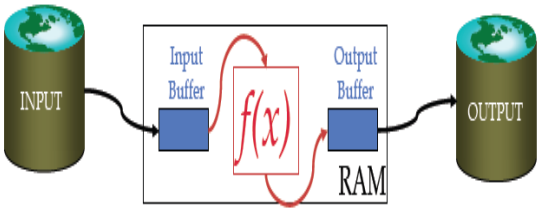
– When Input Buffer is consumed, read another page

– When Output Buffer fills, write it to OUTPUT

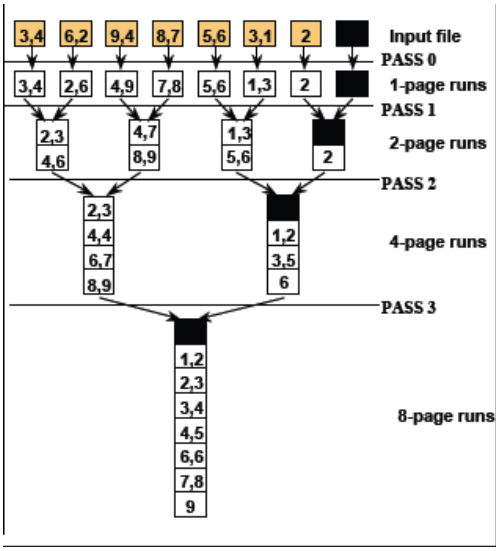
– Reads and Writes are not coordinated

– E.g., if f() is Compress(), you read many pages per write.

– E.g., if f() is DeCompress(), you write many pages per read.



Two-Way External Merge Sort

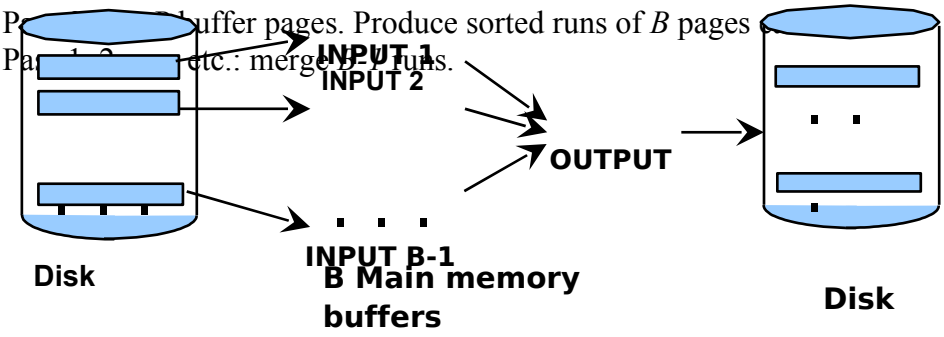


- Each pass we read + write each page in file.
- N pages in the file => the number of passes $= \lceil \log_2 N \rceil + 1$
- So total cost is: $2N(\lceil \log_2 N \rceil + 1)$
- *Idea: Divide and conquer:* sort subfiles and merge

General External Merge Sort

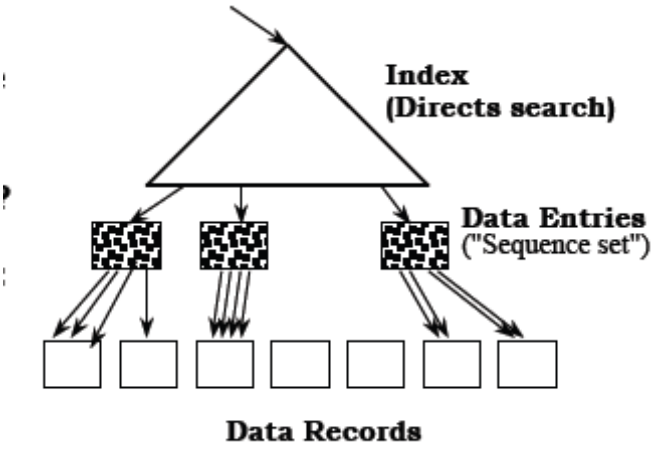
- More than 3 buffer pages. How can we utilize them
- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: use *B* buffer pages. Produce sorted runs of *B* pages each.
  - Pass 1, 2, …, etc.: merge *B-1* runs.

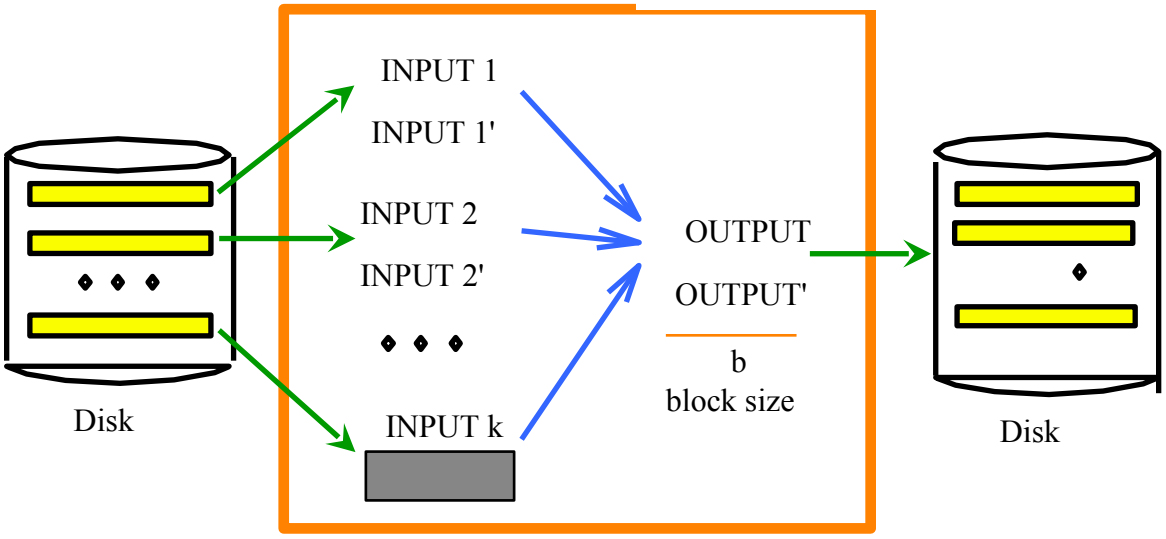
To sort a file with *N* pages using *B* buffer pages:



Double Buffering

- To reduce wait time for I/O request to complete, can prefetch into `shadow block'.
  - Potentially, more passes; in practice, most files *still* sorted in 2-3 passes.





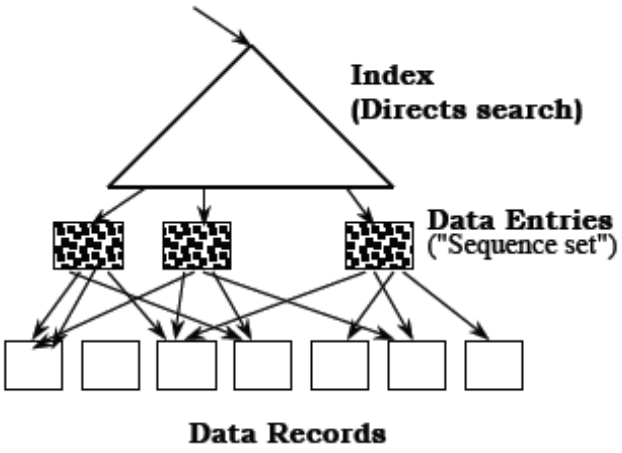
B main memory buffers, k-way merge

Clustered B+ Tree Used for Sorting

- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.

Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains *rid* of a data record. In general, one I/O per data record!



Summary

- External sorting is important
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted *runs* of size *B* (# buffer pages). Later passes: *merge* runs.
  - # of runs merged at a time depends on *B,* and *block size.*
  - Larger block size means less I/O cost per page.
  - Larger block size means smaller # runs merged.
  - In practice, # of runs rarely more than 2 or 3.
- Choice of internal sort algorithm may matter:
  - Heap/tournament sort: slower (2x), longer runs
- The best sorts are wildly fast:
  - Despite 40+ years of research, still improving!
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.

QUERY PROCESSING :

Query processing is a set of activities involving in getting the result of a query expressed in a high-level language. These activities includes parsing the queries and translate them into expressions that can be implemented at the physical level of the file system, optimizing the query of internal form to get a suitable execution strategies for processing and then doing the actual execution of queries to get the results.

The cost of processing of query is dominated by the disk access. For a given query, there are several possible strategies for processing exist, especially when query is complex. The difference between a good strategies and a bad one may be several order of magnitude. Therefore, it is worthwhile for the system to spend some time on selecting good strategies for processing query.

Overview of Query Processing

The main steps in processing a high-level query are illustrated in figure 1

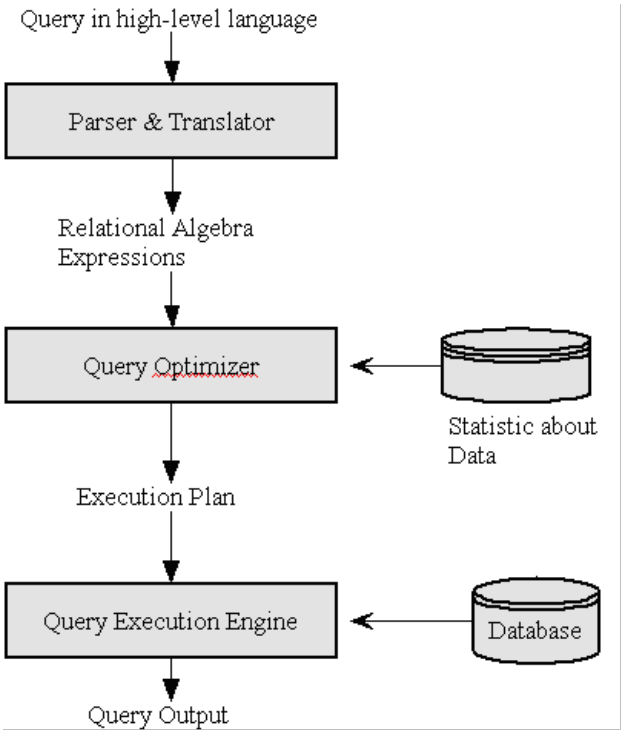


Figure 1: Steps in query processing process

The functions of Query Parser is parsing and translating a given high-level language query into its immediate form such as relational algebra expressions. The parser need to check for the syntax of the query and also check for the semantic of the query (it means verifying the relation names, the attribute names in the query are the names of relations and attributes in the database). A parse-tree of the query is constructed and then translated into relational algebra

expression. A relational algebra expression of a query specifies only partially how to evaluate a query, there are several ways to evaluate an relational algebra expression. For example, consider the query :

SELECT Salary FROM EMPLOYEE WHERE Salary >= 50,000 ;

The possible relational algebra expressions for this query are:

- $\Pi_{Salary}(\sigma_{Salary>=50000}$ (EMPLOYEE))
- $\sigma_{Salary>=50000}(\Pi_{Salary}$ EMPLOYEE)

Further, each relational algebra operation can be executed using various algorithms. For example, to implement the preceding selection, we can do a linear search in the EMPLOYEE file to retrieve the tuples with Salary >= 50000. However, if an index available on the Salary attribute, we can use the index to locate the tuples. Different algorithms might have different cost.

Thus, in order to specify fully how to evaluate a query, the system is responsible for constructing a query execution plan which made up of the relational algebra expression and the detailed algorithms to evaluate each operation in that expression. Moreover, the selected plan should minimize the cost of query evaluation. The process of choosing a suitable query execution plan is known as query optimization This process is perform by Query Optimizer.

One aspect of optimization occurs at relational algebra level. The system attempt to find an expression that is equivalent to the given expression but that is more efficient to execute. The other aspect involves the selection of a detail strategy for processing the query, this relates to choosing the processing algorithm, choosing the indices to use and so on.

Once the query plan is chosen, the Query Execution Engine lastly take the plan, executes that plan and return the answer of the query.

SELECTION OPERATION

File scan is the lowest-level operator to access data. File scans are search algorithms that locate and retrieve records that fulfill a selection condition. Allows an entire relation to be read where the relation is stored in a single, dedicated file.

Basic Algorithms

Two basic scan algorithms to implement the selection operation:

- A1 (linear search).
  - The system scans each file block and tests all records to see whether they satisfy the selection condition.
  - System terminates if the required record is found, without looking at the other records.
- A2 (binary search).
  - System performs the binary search on the blocks of the file.
  - File is ordered on an attribute, and selection condition is an equality comparison.

Selections Using Indices

Index structures referred to a access paths, since they provide a path through which data can be located and accessed. Primary index allows records to be read in an order that corresponds to the physical order in the file. Secondary index any index that is not a primary index.

A3 (primary index, equality on key).

- Use the index to retrieve a single record that satisfies the corresponding equality condition.

A4 (primary index, equality on nonkey).

- Same as A3, but multiple records may need to be fetched.

A5 (secondary index, equality).

- Retrieve a single record if the equality condition is on a key.
- Retrieve multiple records if the indexing field is not a key.

Selections Involving Comparisons

Consider the form $\sigma A \geq v\ (r)$.

- Can be implemented by linear or binary search or by using indices:

A6 (primary index, comparison).

- For A ≥ v, look for first tuple that has the value of A = v, return all the tuples starting from the tuple up to the end of the file.
- For A < v, file scan from the beginning up to (but not including) the first tuple with attribute A = v.

A7 (secondary index, comparison).

- The lowest-level blocks are scanned
  - From the smallest value up to v for < and ≤.
  - From v up to the maximum value for > and ≥.

Implementation of Complex Selections

Previously only considered simple selection conditions with an equality or comparison operation.Now consider more complex selection predicates.

- **Conjunction: $\sigma\theta 1 \cap \theta 2 \cap \ldots \cap \theta n\ (r)$**
- **Disjunction: $\sigma\theta 1 \cup \theta 2 \cup \ldots \cup \theta n\ (r)$**
- **Negation: $\sigma\text{-}\theta\ (r)$**

A9 (conjunctive selection using composite index (multiple attributes).

Search index directly if selection specifies an equality condition on 2 or move attributes and a composite index exists.

Type of index determines if A3, A4, or A5 will be used.

A10 (conjunction selection by intersection of identifiers).

Requires indices with record pointers.

Scan each index for pointers that satisfy an individual condition.

Take intersection of all retrieved pointers to get set of pointers that satisfy the conjunctive condition.

Then pointers used to retrieve actual records.

If indices not available on all conditions, then algorithm tests retrieved records against remaining conditions.

A11 (disjunctive selection by union of identifiers).

Each index is scanned for pointers to tuples that satisfy the individual condition.

Union of all the retrieved pointers gives the set of pointers to all tuples that satisfy the disjunctive condition.

Pointers then used to retrieve actual records.

JOIN OPERATIONS

Three fundamental algorithms exist for performing a join operation:

Nested loop join,

Sort-merge join and

Hash join.

Nested Loop Join

Nested loop join is a naive algorithm that joins two relations *R* and *S* by making two nested loops:

For each tuple r in R do

For each tuple s in S do

If r and s satisfy the join condition

Then output the tuple <r,s>

This algorithm will involve $n_r * b_s + b_r$ block transfers and $n_r + b_r$ seeks. Where b_r and b_s are number of blocks in relations r and s respectively, and n_r is the number of tuples in relation r.

The algorithm runs in $O(|R||S|)$ I/Os, where $|R|$ and $|S|$ is the number of tuples contained in *R* and *S* respectively. Can easily be generalized to join any number of relations.

The block nested loop join algorithm is a generalization of the simple nested loops algorithm that takes advantage of additional memory to reduce the number of times that the *S* relation is scanned.

Block-Nested Loop

A **block-nested loop** is an algorithm used to join two relations in a relational database.

This algorithm is a variation on the simple nested loop join used to join two relations *R* and *S* (the "outer" and "inner" join operands, respectively). Suppose $| R | < | S |$. In a traditional nested loop join, *S* will be scanned once for every tuple of *R*. If there are many qualifying *R* tuples, and particularly if there is no applicable index for the join key on *S*, this operation will be very expensive.

The block nested loop join algorithm improves on the simple nested loop join by only scanning *S* once for every *group* of *R* tuples. For example, one variant of the block nested loop join reads an entire page of *R* tuples into memory and loads them into a hash table. It then scans *S*, and probes the hash table to find *S* tuples that match any of the tuples in the current page of *R*. This reduces the number of scans of *S* that are necessary.

A more aggressive variant of this algorithm loads as many pages of *R* as can be fit in the available memory, loading all such tuples into a hash table, and then repeatedly scans *S*. This further reduces the number of scans of *S* that are necessary. In fact, this algorithm is essentially a special-case of the classic hash join algorithm.

The block nested loop runs in $O(P_r P_s / M)$ I/Os where *M* is the number of available pages of internal memory and P_r and P_s is size of *R* and *S* respectively in pages. Note that block nested loop runs in $O(P_r + P_s)$ I/Os if *R* fits in the available internal memory.

Sort-Merge Join

The **Sort-Merge Join** (also known as Merge-Join) is an example of a join algorithm and is used in the implementation of a relational database management system.

The basic problem of a join algorithm is to find, for each distinct value of the join attribute, the set of tuples in each relation which display that value. The key idea of the Sort-merge algorithm is to first sort the relations by the join attribute, so that interleaved linear scans will encounter these sets at the same time.

In practice, the most expensive part of performing a sort-merge join is arranging for both inputs to the algorithm to be presented in sorted order. This can be achieved via an explicit sort operation (often an external sort), or by taking advantage of a pre-existing ordering in one or both of the join relations. The latter condition can occur because an input to the join might be produced by an index scan of a tree-based index, another merge join, or some other plan operator that happens to produce output sorted on an appropriate key.

We have two relations *R* and *S* and $| R | < | S |$. *R* fits in P_r pages memory and *S* fits in P_s pages memory. So, in the worst case **Sort-Merge Join** will run in $O(P_r + P_s)$ I/Os. In the case that *R* and *S* are not ordered the worst case will be $O(P_r + P_s + 2(P_r + P_r log(P_r) + P_s + P_s log(P_s)))$ (where the last two terms are the cost of ordering both of them first).

Hash join

The **Hash join** is an example of a join algorithm and is used in the implementation of a relational database management system.

The task of a join algorithm is to find, for each distinct value of the join attribute, the set of tuples in each relation which have that value.

Hash joins require an equi-join predicate (a predicate comparing values from one table with values from the other table using the equals operator '=').

Classic hash join

The classic hash join algorithm for an inner join of two relations proceeds as follows: first prepare a hash table for the smaller relation, by applying a hash function to the join attribute of each row. Then scan the larger relation and find the relevant rows by looking in the hash table. The first phase is usually called the "build" phase, while the second is called the "probe" phase. Similarly, the join relation on which the hash table is built is called the "build" input, whereas the other input is called the "probe" input. [GG] This algorithm is simple, but it requires that the smaller join relation fits into memory, which is sometimes not the case. A simple approach to handling this situation proceeds as follows:

1. For each tuple *r* in the build input *R*
    1. Add *r* to the in-memory hash table
    2. If the size of the hash table equals the maximum in-memory size:
        1. Scan the probe input *S*, and add matching join tuples to the output relation
        2. Reset the hash table
2. Do a final scan of the probe input *S* and add the resulting join tuples to the output relation

This is essentially the same as the block nested loop join algorithm. This algorithm scans *S* more times than necessary.

Grace hash join

A better approach is known as the "grace hash join", after the GRACE database machine for which it was first implemented. This algorithm avoids rescanning the entire *S* relation by first partitioning both *R* and *S* via a hash function, and writing these partitions out to disk. The algorithm then loads pairs of partitions into memory, builds a hash table for the smaller partitioned relation, and probes the other relation for matches with the current hash table. Because the partitions were formed by hashing on the join key, it must be the case that any join output tuples must belong to the same partition. It is possible that one or more of the partitions still does not fit into the available memory, in which case the algorithm is recursively applied: an additional orthogonal hash function is chosen to hash the large partition into sub-partitions, which are then processed as before. Since this is expensive, the algorithm tries to reduce the chance that it will occur by forming as many partitions as possible during the initial partitioning phase.

Hybrid hash join

The hybrid hash join algorithm is a refinement of the grace hash join which takes advantage of more available memory. During the partitioning phase, the hybrid hash join uses the available memory for two purposes:

1. To hold the current output buffer page for each of the *k* partitions
2. To hold an entire partition in-memory, known as "partition 0"

Because partition 0 is never written to or read from disk, the hybrid hash join typically performs fewer I/O operations than the grace hash join. Note that this algorithm is memory-sensitive, because there are two competing demands for memory (the hash table for partition 0, and the output buffers for the remaining partitions). Choosing too large a hash table might cause the algorithm to recurse because one of the non-zero partitions is too large to fit into memory.

Hash anti-join

Hash joins can also be evaluated for an anti-join predicate (a predicate selecting values from one table when no related values are found in the other). Depending on the sizes of the tables, different algorithms can be applied:

Hash left anti-join

- Prepare a hash table for the **NOT IN** side of the join.
- Scan the other table, selecting any rows where the join attribute hashes to an empty entry in the hash table.

This is more efficient when the **NOT IN** table is smaller than the **FROM** table

Hash right anti-join

- Prepare a hash table for the **FROM** side of the join.
- Scan the **NOT IN** table, removing the corresponding records from the hash table on each hash hit
- Return everything that left in the hash table

This is more efficient when the **NOT IN** table is larger than the **FROM** table

Hash semi-join

Hash semi-join is used to return the records found in the other table. Unlike plain join, it returns each matching record from the leading table only once, not regarding how many matches are there in the **IN** table.

As with the anti-join, semi-join can also be left and right:

Hash left anti-join

- Prepare a hash table for the **IN** side of the join.
- Scan the other table, returning any rows that produce a hash hit.

The records are returned right after they produced a hit. The actual records from the hash table are ignored.

This is more efficient when the **IN** table is smaller than the **FROM** table

Hash right anti-join

- Prepare a hash table for the **FROM** side of the join.
- Scan the **IN** table, returning the corresponding records from the hash table and removing them

With this algorithm, each record from the hash table (that is, **FROM** table) can only be returned once, since it's removed after being returned.

This is more efficient when the **IN** table is larger than the **FROM** table

EVALUATION OF EXPRESSIONS

The output of Parsing and Translating step in the query processing is a relational algebra expression. For a complex query, this expression consists of several operations and interacts with various relations. Thus the evaluation of the expression is very costly in terms of both time and memory space. Now we consider how to evaluate an expression containing multiple operations. The obvious way to evaluate the expression is simply evaluate one operation at a time in an appropriate order. The result of an individual evaluation will be stored in a temporary relation, which must be written to disk and might be use as the input for the following evaluation. Another approach is evaluating several operations simultaneously in a pipeline, in which result of one operation passed on to the next one, no temporary relation is created.

These two approaches for evaluating expression are materialization and pipelining.

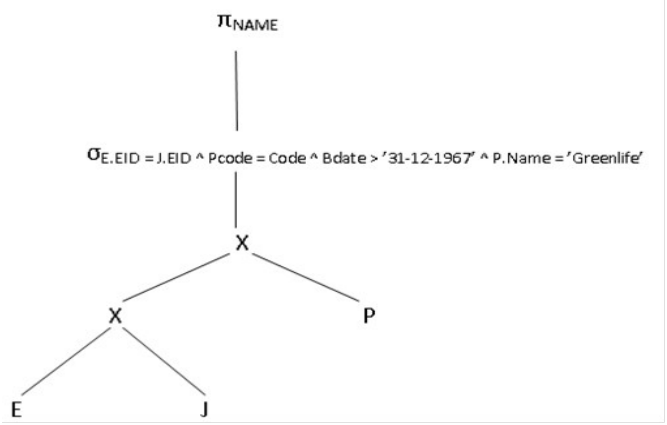
Materialization

We will illustrate how to evaluate an expression using materialization approach by looking at an example expression.

Consider the expression

$\pi_{\text{Name,Address}}(\sigma_{\text{Dname='HumanResourse'}}(\text{DEPARTMENT}) * \text{EMPLOYEE}$

The corresponding query tree is



Sample query tree for a relational algebra expression

When we apply the materialization approach, we start from the lowest-level operations in the expression. In our example, there is only one such operation: the SELECTION on DEPARTMENT. We execute this opeartion using the algorithm for it for example Retriving multiple records using a secondary index on DName. The result is stored in the temporary relations. We can use these temporary relation to execute the operation at the next level up in the tree . So in our example the inputs of the join operation are EMPLOYEE relation and temporary relation which is just created. Now, evaluate the JOIN operation, generating another temporary relation. The execution terminates when the root node is executed and produces the result relation for the query. The root node in this example is the PROJECTION applied on the temporary relation produced by executing the join.

Using materialized evaluation, every immediate operation produce a temporary relation which then used for evaluation of higher level operations. Those temporary relations vary in size and might have to stored in the disk . Thus, the cost for this evaluation is the sum of the costs of all operation plus the cost of writing/reading the result of intermediate results to disk.

Pipelining

We can improve query evaluation efficiency by reducing the number of temporary relations that are produced. To archieve this reduction, it is common to combining several operations into a pipeline of operations. For illustrate this idea, consider our example, rather being implemented seperately, the JOIN can be combined with the SELECTION on DEPARTMENT and the EMPLOYEE relation and the final PROJECTION operation. When the select operation generates a tuple of its result, that tuple is passed immediately along with a tuple from EMPLOYEE relation to the join. The join receive two tuples as input, process them , if a result tuple is generated by the join, that tuple again is passed immediately to the projection operation process to get a tuple in the final result relation.

We can implement the pipeline by create a query execution code. Using pipelining in this situation can reduce the number of temporary files, thus reduce the cost of query evaluation. And we can see that in general, if pipelining is applicable, the cost of the two approaches can differ substantially. But there are cases where only materialization is feasible.

QUERY OPTIMIZATION

Using Heuristic in Query Optimization

Heuristic optimization applies the rules to the initial query expression and produces the heuristically transformed query expressions.

A heuristic is a rule that works well in most cases but not always guaranteed. For example, a rule for transforming relational-algebra expression is "Perform selection operations as early as possible". This rule is generate based on intuitive idea that selection is the operation that results a subset of the input relation so apply selection early might reduce the immediate result size. However, there are cases perform selection before join is not a good idea. For example for the expression

$$\sigma_F(r \bowtie_C s)$$

If relations r and s both large and the selection produces a result of only one tuple so do the selection before join might be good. But, assume that r is small relation, s is very large, s has an index on the join attribute and no index on the attributes of s in selection condition then compute the join using index then do select might be better then scan the whole s to do selection first.

Transformation of Relational Expression

One aspect of optimization occurs at relational algebra level. This involves transforming an initial expression (tree) into an equivalent expression (tree) which is more efficient to execute. Two relational algebra expressions are said to be equivalent if the two expressions generate two relation of the same set of attributes and contain the same set of tuples although their attributes may be ordered differently.

The query tree is a data structure that represents the relational algebra expression in the query optimization process. The leaf nodes in the query tree corresponds to the input relations of the query. The internal nodes represent the operators in the query. When executing the query, the system will execute an internal node operation whenever its operands avaiable, then the internal node is replaced by the relation which is obtained from the preceding execution.

Equivalence Rules for transforming relational expressions

There are many rules which can be used to transform relational algebra operations to equivalent ones.

We will state here some useful rules for query optimization.

In this section, we use the following notation:

- E1, E2, E3,… : denote relational algebra expressions
- X, Y, Z : denote set of attributes
- F, F1, F2, F3 ,… : denote predicates (selection or join conditions)
- Commutativity of Join, Cartesian Product operations

$E_1 \bowtie_F E_2 \equiv E_2 \bowtie_F E_1$

$E_1 \times E_2 \equiv E_2 \times E_1$

Note that: Natural Join operator is a special case of Join, so Natural Joins are also commutative.

- Associativity of Join , Cartesian Product operations

$(E_1 * E_2) * E_3 \equiv E_1 * (E_2 * E_3)$

$(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$

$(E_1 \bowtie_{F1} E_2) \bowtie_{F2} E_3 \equiv E_1 \bowtie_{F1} (E_2 \bowtie_{F2} E_3)$

Join operation associative in the following manner: F1 involves attributes from only E1 and E2 and F2 involves only attributes from E2 and E3

- Cascade of Projection

$\pi_{X1}(\pi_{X2}(...(\pi_{Xn}(E))...)) \equiv \pi_{X1}(E)$

- Cascade of Selection

$\sigma_{F1 \wedge F2 \wedge ... \wedge Fn}(E) \equiv \sigma_{F1}(\sigma_{F2}(...(\sigma_{Fn}(E))...))$

- Commutativity of Selection

$\sigma_{F1}(\sigma_{F2}(E)) \equiv \sigma_{F2}(\sigma_{F1}(E))$

- Commuting Selection with Projection

$\pi_X(\sigma_F(E)) \equiv \sigma_F(\pi_X(E))$

This rule holds if the selection condition F involves only the attributes in set X.

- Selection with Cartesian Product and Join
- If all the attributes in the selection condition F involve only the attributes of one of the expression say E1, then the selection and Join can be combined as follows:

$\sigma_F(E_1 \bowtie_C E_2) \equiv (\sigma_F(E_1)) \bowtie_C E_2$

- If the selection condition F = F1 AND F2 where F1 involves only attributes of expression E1 and F2 involves only attribute of expression E2 then we have:

$\sigma_{F1 \wedge F2}(E_1 \bowtie_C E_2) \equiv (\sigma_{F1}(E_1)) \bowtie_C (\sigma_{F2}(E_2))$

- If the selection condition F = F1 AND F2 where F1 involves only attributes of expression E1 and F2 involves attributes from both E1 and E2 then we have:

$\sigma_{F1 \wedge F2}(E_1 \bowtie_C E_2) \equiv \sigma_{F2}((\sigma_{F1}(E_1)) \bowtie_C E_2)$

The same rule apply if the Join operation replaced by a Cartersian Product operation.

- Commuting Projection with Join and Cartesian Product
- Let X, Y be the set of attributes of E1 and E2 respectively. If the join condition involves only attributes in XY (union of two sets) then :

$\pi_{XY}(E_1 \bowtie_F E_2) \equiv \pi_X(E_1) \bowtie_F \pi_Y(E_2)$

The same rule apply when replace the Join by Cartersian Product

- If the join condition involves additional attributes say Z of E1 and W of E2 and Z,W are not in XY then :

$\pi_{XY}(E_1 \bowtie_F E_2) \equiv \pi_{XY}(\pi_{XZ}(E_1) \bowtie_F \pi_{YW}(E_2))$

- Commuting Selection with set operations

The Selection commutes with all three set operations (Union, Intersect, Set Difference) .

$\sigma_F(E_1 \cup E_2) \equiv (\sigma_F(E_1)) \cup (\sigma_F(E_2))$

The same rule apply when replace Union by Intersection or Set Difference

- Commuting Projection with Union

$\pi_X(E_1 \cup E_2) \equiv (\pi_X(E_1)) \cup (\pi_X(E_2))$

- Commutativity of set operations: The Union and Intersection are commutative but Set Difference is not.

$E_1 \cup E_2 = E_2 \cup E_1$
$E_1 \cap E_2 = E_2 \cap E_1$

Associativity of set operations: Union and Intersection are associative but Set Difference is not

$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$

$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$

- Converting a Catersian Product followed by a Selection into Join.

If the selection condition corresponds to a join condition we can do the convert as follows:

$\sigma_F(E_1 \times E_2) \equiv E_1 \bowtie_F E_2$

Example of Transformation

Consider the following query on COMPANY database: “Find the name of employee born after 1967 who work on a project named ‘Greenlife’ “.

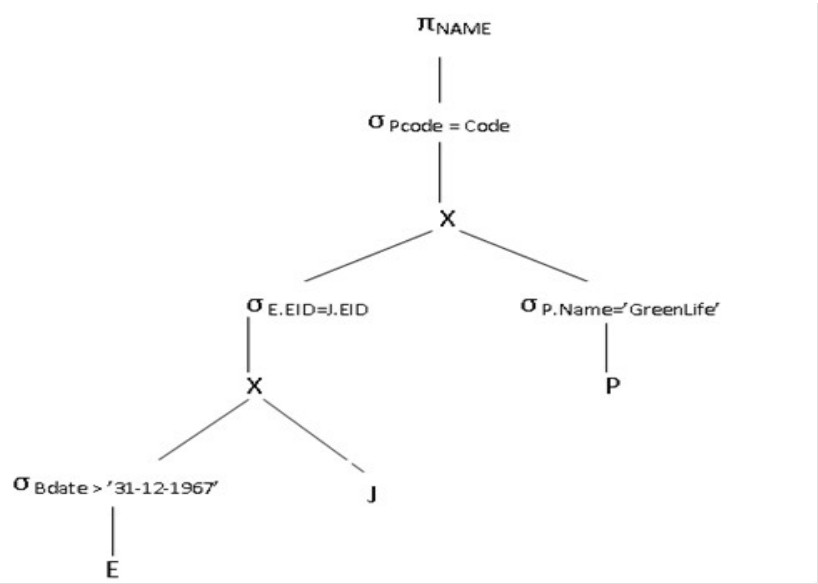
The SQL query is:

SELECT Name

FROM EMPLOYEE E, JOIN J, PROJECT P

WHERE E.EID = J.EID and PCode = Code and Bdate > ’31-12-1967’ and P.Name = Greenlife’;

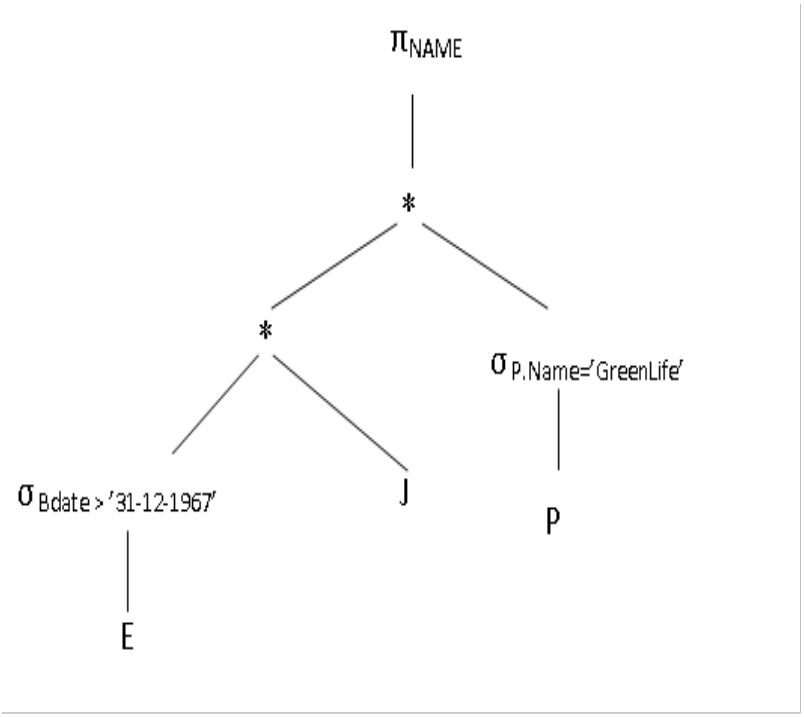
The initial query tree for this SQL query is



Initial query tree for query in example

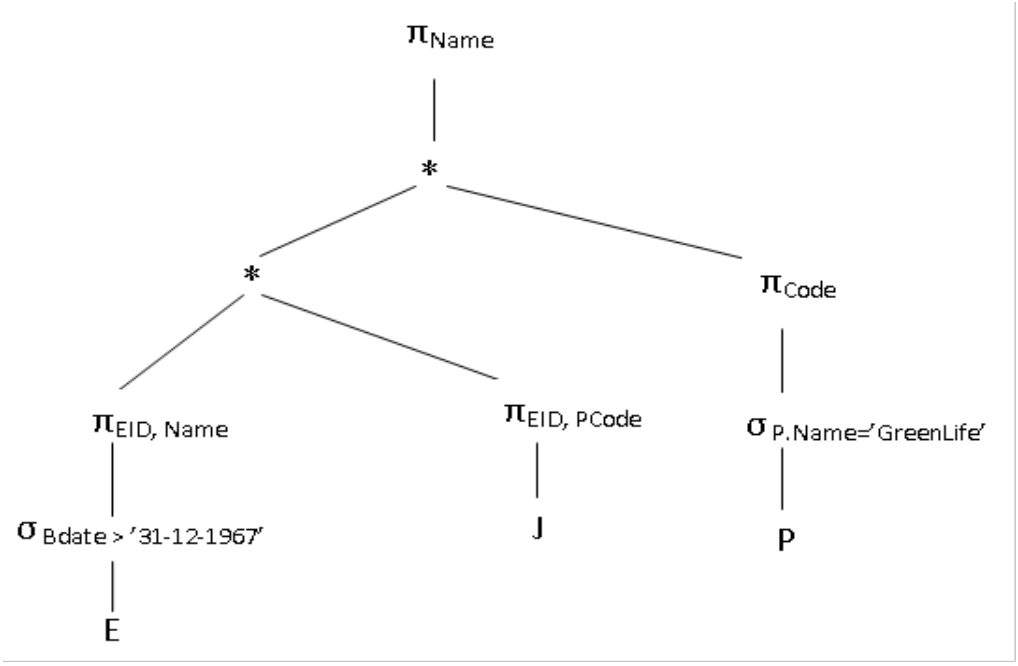
We can transform the query in the following steps:

- Using transformation rule number 7 apply on Catersian Product and Selection operations to moves some Selection down the tree. Selection operations can help reduce the size of the temprary relations which involve in Catersian Product.



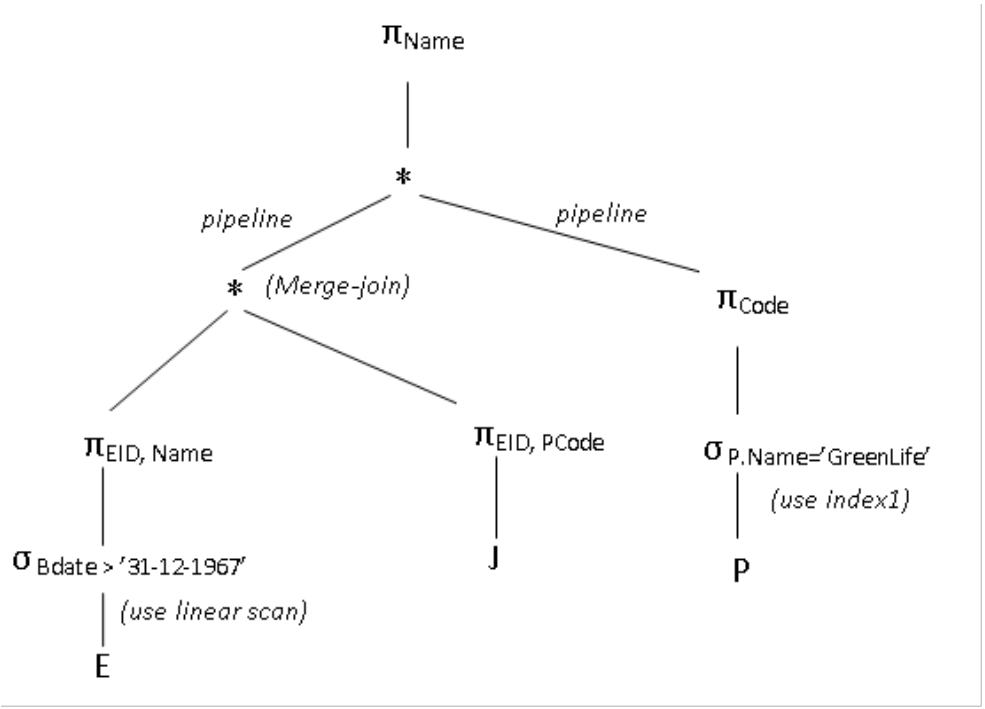
Move Selection down the tree

- Using rule number 13 to convert the sequence (Selection, Cartesian Product) in to a Join. In this situation, since the Join condition is equality comparison in same attributes so we can convert it to Natural Join.



Combine Catersian Product with subsequent Selection into Join

- Using rule number 8 to moves Projection operations down the tree.



Move projections down the query tree

Heuristic Algebraic Optimization algorithm

Here are the steps of an algorithm that utilizes equivalence rules to transfrom the query tree.

- Break up any Selection operation with conjunctive conditions into a cascade of Selection operations. This step is based on equivalence rule number 4.
- Move selection operations as far down the query tree as possible. This step use the commutativity and associativity of selection as mentioned in equivalence rule number 5,6,7 and 9.
- Rearrange the leaf nodes of the tree so most restrictive selections are done first. Most restrictive selection is the one that produces the fewest number of tuples. In addition, make sure that the ordering of leaf nodes does not cause the Catersian Product operation. This step relies on the rules of associativity of binary operations such as rule 2 and 12
- Combine a Cartesian Product with a subsequent Selection operation into a Join operation if the selection condition represents a join condition (rule 13)
- Break down and move lists of projection down the tree as far as possible. Creating new Projection operations as needed (rule 3, 6, 8, 10)
- Identify sub-tree that present groups of operations that can be pipelined and executing them using pipelining.

NORMALIZATION

Normalization is the process of efficiently organizing data in a database. There are two goals of the normalization process:

Eliminating redundant data (for example, storing the same data in more than one table) and

ensuring data dependencies make sense (only storing related data in a table).

Both of these are worthy goals as they **reduce the amount of space** a database consumes and ensure that data is logically stored.

Normalization Forms

Different normalization forms are,

First Normal Form (1NF)

A relation is said to be in the first normal form if it is already in **unnormalized form** and it **has no repeating group**.

Second Normal Form (2NF)

A relation is said to be in the second normal form if it is **already in the first normal form** and it **has no partial dependency**.

Partial Dependency :

In a relation having more than one key field, a subset of non-key fields may depend on all the key fields but another subset/a particular non-key field may depend on only one of the key fields.Such dependency is called partial dependency.

Third Normal Form (3NF)

A relation is said to be in the third normal form if it is **already in the second normal form** and it **has no transitive dependency**.

Transitive dependency:

In a relation, there may be dependency among non-key fields.Such dependency is called transitive dependency.

Boyce-Codd normal Form (BCNF)

A relation is said to be in the Boyce-Codd normal form if it is **already in the third normal form** and **every determinant is a candidate key**. It is a stronger version of 3NF.

Determinant :

A determinant is any field on which some other field is fully functionally dependent.

Example :

Alpha Book House

Pune-413001

Customer No : 1052

Customer Name : Beta school of computer science

Address : KK Nagar , Pune-01.

ISBN	Book Title	Author's name	Author's Country	Qty	Price(Rs)	Amt(Rs.)
1-213-9	DOS	Sinha	India	5	250	1250
0-112-6	DBMS	Henry	USA	6	300	1800
0-111-7	C	Herbert	USA	5	100	500
					Grand Total	3550

The invoice report shown above is represented in a form of relation. The relation is named as invoice. This is unnormalized form.

1) Invoice (Cust-no, Cust-name, Cust_add, (ISBN, Title, Author-name, Author Country, Qty, Unit_price)) – Relation1

The amount can be calculated by multiplying the Qty and Price. Similarly, the grand total can also computed by adding all the amount.

First Normal Form : (1 NF)

- The relation is in 1NF if it has no repeating groups.
- So, the invoice relation is divided into two sub-relations shown below to remove the repeating group.

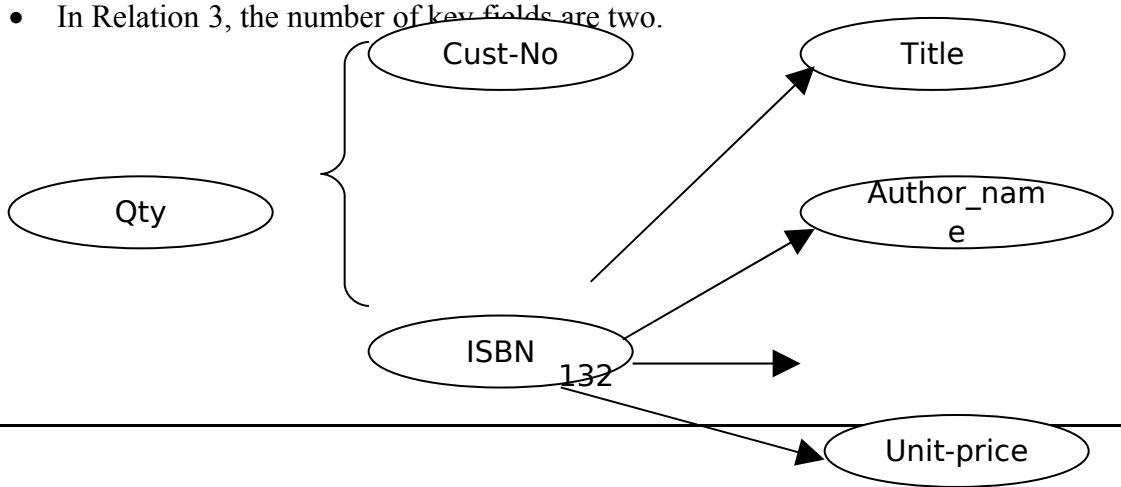
2) Customer (Cust-no, Cust-name, Cust_add) – Relation 2

3) Customer_Book (Cust-no ,ISBN, Title, Author-name, Author Country, Qty, Unit_price) – Relation 3

Now both are in 1NF.

Second Normal Form : (2 NF)

- 2 NF removes Partial Dependency among attributes of relation.
- In Relation 2, there is no partial dependency because there is only one key field.
- In Relation 3, the number of key fields are two.



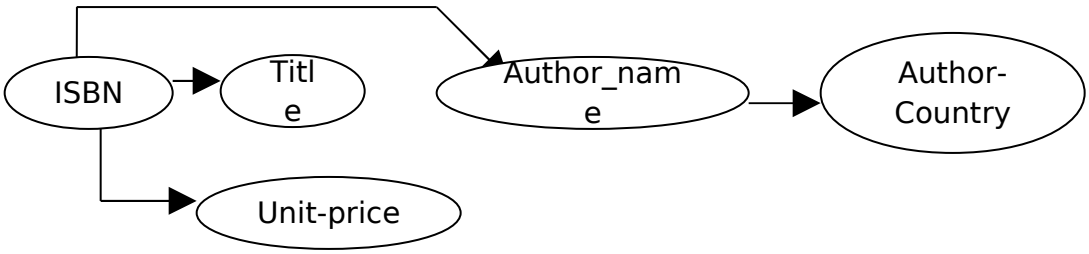
Here, Qty depends on Cust-No and ISBN, but the remaning fields depend only on ISBN. So, there partial dependency exists.Hence , Relation 3 is divided into 2 relations as,

4) Sales (Cust-no ,ISBN, Qty) – Relation 4
5) Book-Author (ISBN, Title, Author-name, Author Country, Unit_price) – Relation 5

Above 2 relations are now in 2NF.

Third Normal Form (3NF) :

- Removes transitive dependency among attributes of relation.
- In Relation 4, there in only one non-key field. So,there is no transtive dependency. Hence, Reltion 4 is in 3NF.
- In Relation 5, author's country depends on author name. This means that it has transitive dependency.



Relation 5 is divided into two relations :

6) Book (ISBN, Title, Author-name, Unit_price) – Relation 6
7) Author (Author-name, Author Country) – Relation 7

Boyce_Codd Normal Form (BCNF)

- BCNF is based on the concept of a *determinant*.
- A determinant is any attribute (simple or composite) on which some other attribute is fully functionally dependent.
- A relation is in BCNF is, and only if, every determinant is a candidate key.

Conisder a database table that stores employee information and has the attributes employee_id, first_name, last_name, title. In this table, the field employee_id determines first_name and last_name. Similarly, the tuple (first_name, last_name) determines employee_id.

Physical Database Design in Relational Databases:

1. Conceptual Schema OR logical: info to capture, tables, columns, views, etc.(concerned with the *what*)

2. Physical Schema: indexes, clustering, etc.(concerned with the *how*)

It describes base tables, file organizations, and indexes used to achieve efficient access to the data, and any associated integrity constraints and s ecurity restrictions

- For each table, need to define:
  - name of the table;
  - list of simple columns in brackets;
  - PK and, where appropriate, AKs and FKs.
  - referential integrity constraints for any FKs identified.
- For each column, need to define:
  - its domain, consisting of a data type, length, and any constraints on the domain;
  - an optional default value for the column;
  - whether the column can hold nulls.

Physical design linked tightly to query optimization DB design is usually "top-down"

- Factors that Influence Physical Database Design:

I. Analyzing the database queries and

Transactions:

- For each query, the following information

is needed.

1. The *files* that will be accessed by the query;

2. The *attributes* on which any *selection* conditions for the query are specified;
3. The *attributes* on which any *join* conditions or conditions to link multiple tables or objects for the query are specified;
4. The *attributes* whose values will be *retrieved* by the query.
5. For each update transaction or operation, the following information is needed.
    1. The files that will be updated;
    2. The type of operation on each file

    (insert, update or delete);
    3. The attributes on which selection conditions for a delete or update operation are specified;
    4. The attributes whose values will be

    changed by an update operation.

II. Analyzing the expected frequency of invocation of queries and transactions. *Insert, Select, Update, Delete Frequencies*

- What indexes should we create?
- what kind of an index should it be?

Clustered? Dynamic/static?

			Employee Table		
Transaction	***Frequency***	***% table***	**Name**	**Salary**	**Address**
Payroll Run	monthly	100	S	S	S
Add Emps	daily	0.1	I	I	I
Delete Emps	daily	0.1	D	D	D
Give Raises	monthly	10	S	U	

III. Analyzing the time constraints of queries and transactions

IV. Analyzing the expected frequencies of update operations.

- A minimum number of access paths should be specified for a file that is updated frequently.

V . Analyzing the uniqueness constraints on

Attributes.

- Attributes — that are either the primary key or constrained to be unique.

VI. Design decisions about indexing

Whether to index an attribute?

What attribute or attributes to index on?

Whether to set up a clustered index?

Whether to use a hash index over a tree

index?

Whether to use dynamic hashing for the file?

VII. Denormalization: as a design decision for speeding up queries –

The goal of denormalization is to improve the performance of frequently occurring queries and transactions.

1.Tuning:

- The process of continuing to revise/adjust the physical database design by monitoring resource utilization as well as internal DBMS processing to reveal bottlenecks such as contention for the same data or devices.

A) Goal:

- To make application run faster
- To lower the response time of queries/transactions
- To improve the overall throughput of transactions.

B) Statistics internally collected in DBMSs:

- Size of individual tables
- Number of distinct values in a column
    - The number of times a particular query or transaction is submitted/executed in an interval of time
    - The times required for different phases of query and transaction processing

C) Statistics obtained from monitoring:

- Storage statistics
- I/O and device performance statistics
- Query/transaction processing statistics
- Locking/logging related statistics
- Index statistic

D) Problems to be considered in tuning:

- How to avoid excessive lock contention?

- How to minimize overhead of logging and unnecessary dumping of data?
- How to optimize buffer size and scheduling of processes?
- How to allocate resources such as disks, RAM and processes for most efficient utilization?

E) Tuning Indexes

- Reasons to tuning indexes
- Certain queries may take too long to run for lack of an index;
- Certain indexes may not get utilized at all;
- Certain indexes may be causing excessive overhead because the index is on an attribute that undergoes frequent changes

F) Options to tuning indexes

- Drop or/and build new indexes
- Change a non-clustered index to a clustered index (and vice versa)
- Rebuilding the index

Tuning Queries

- Indications for tuning queries
- A query issues too many disk accesses
- The query plan shows that relevant indexes are not being used

• Basic idea:

– They take a workload of queries

– They use the optimizer cost metrics to estimate the cost of the workload

- Heuristics to help this go faster

• Common areas of weakness:

– Selections involving null values (bad selectivity estimates)

– Selections involving arithmetic or string expressions

– Selections involving OR conditions

– Complex subqueries (more on this later)

- Minimize the use of DISTINCT:

- Minimize the use of GROUP BY and HAVING:

- Null values, arithmetic conditions, string expressions, the use of ORs, nested queries.

– Avoid nested queries, temporary relations, complex conditions.

– Attributes mentioned in a WHERE clause are candidates for index search keys.

– Range conditions are sensitive to clustering

– Exact match conditions don't require clustering

Try to choose indexes that benefit as many queries as possible. Multi-attribute search keys should be considered

SELECT E.ename, D.mgr

FROM Emp E, Dept D

WHERE E.sal BETWEEN 10000 AND 20000

AND E.hobby='Stamps' AND E.dno=D.dno

- If condition is: 20<*age*<30 AND 3000<*sal*<5000:

Clustered tree index on <*age,sal*> or <*sal,age*> is best.

- considering a join condition:

– B+-tree on inner is very good for Index Nested Loops.

– *Clustered* B+ tree on join column(s) good for Sort-Merge.

SECURITY

- The information in your database is important.
- Therefore, you need a way to protect it against unauthorized access, malicious destruction or alteration, and accidental introduction of data inconsistency.
- Some forms of malicious access:
  - Unauthorized reading (theft) of data
  - Unauthorized modification of data
  - Unauthorized destruction of data
- To protect a database, we must take security measures at several levels.

Security Levels

- Database System: Since some users may modify data while some may only query, it is the job of the system to enforce authorization rules.
- Operating System: No matter how secure the database system is, the operating system may serve as another means of unauthorized access.
- Network: Since most databases allow remote access, hardware and software security is crucial.

- Physical: Sites with computer systems must be physically secured against entry by intruders or **terrorists**.
- Human: Users must be authorized carefully to reduce the chance of a user giving access to an intruder.

Authorization

- For security purposes, we may assign a user several forms of authorization on parts of the databases which allow:
  - Read: read tuples.
  - Insert: insert new tuple, not modify existing tuples.
  - Update: modification, not deletion, of tuples.
  - Delete: deletion of tuples.
- We may assign the user all, none, or a combination of these.
- We may also assign a user rights to modify the database schema:
  - Index: allows creation and modification of indices.
  - Resource: allows creation of new relations.
  - Alteration: addition or deletion of attributes in a tuple.
  - Drop: allows the deletion of relations.

Authorization and Views

- Views provide 'derived' tables
  - A view is the result of a SELECT statement which is treated like a table
  - You can SELECT from views just like tables

  CREATE VIEW <name>

  AS <select stmt>

- **<name>** is the name of the new view
- **<select stmt>** is a query that returns the rows and columns of the view

View Example

- Suppose a bank clerk needs to know the names of the customers of each branch, but is not authorized to see specific loan information.
  - The *cust-loan* view is defined in SQL as follows:

```
create view cust-loan as
 select branchname, customer-name
 from   borrower, loan
 where borrower.loan-number = loan.loan-number
```

- The clerk is authorized to see the result of the query:

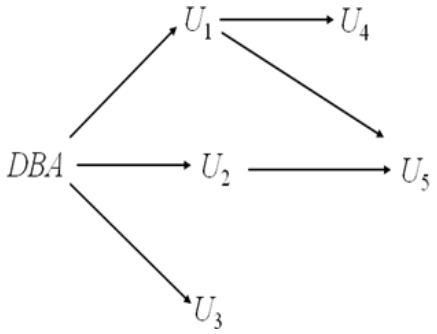
  select * from *cust-loan*

Granting of Privileges :

- The passage of authorization from one user to another may be represented by an **authorization graph.**
- The nodes of this graph are the users.
- The root of the graph is the database administrator.
- Consider graph for update authorization on loan.
- An edge $U_i \rightarrow U_j$ indicates that user U_i has granted update authorization on loan to U_j.

Authorization Grant Graph:

All edges in an authorization graph must be part of some path originating with the DBA .



Authorization in SQL

- The SQL language offers a fairly powerful mechanism for defining authorizations by using privileges.

Privileges in SQL :

- SQL standard includes the privileges:
  - Delete
  - Insert
  - Select
  - Update
  - References: permits declaration of foreign keys.

SQL includes commands to **grant and revoke** privileges.

SYNTAX :

grant <privilege list>

on <relation or view name>

to <user>

EX1:

grant select on *loan* to u1, u3, u4

grant update(amount) on *loan* to u1, u3, u4

Privilege to Grant Privileges

- By default, a user granted privileges is not allowed to grant those privileges to other users.
- To allow this, we append the term "with grant option" clause to the appropriate grant command.

EX1:

grant select on *branch* to *U1* with grant option

Revoking Privileges

- The revoke statement is used to revoke authorization.

  revoke<privilege list>

  on <relation name or view name> from <user list> [restrict|cascade]

Example:

revoke select on *branch* from U_1, U_2, U_3 cascade

- Cascade : Revocation of a privilege from a user may cause other users also to lose that privilege;
- We can prevent cascading by specifying restrict:

  revoke select on *branch* from U_1, U_2, U_3 restrict

Roles

- Roles permit common privileges for a class of users can be specified just once by creating a corresponding "role"
- Privileges can be granted to or revoked from roles, just like user
- Roles can be assigned to users, and even to other roles

create role *teller*

create role *manager*

grant select on *branch* to *teller*

grant *teller* to *manager*

UNIT V TRANSACTION MANAGEMENT

Transaction concepts - Concurrency control - Crash recovery - Decision support - Case studies

TRANSACTION MANAGEMENT

TRANSACTION CONCEPT

A transaction may be thought of as an interaction with the system, resulting in a change to the system state. While the interaction is in the process of changing system state, any number of events can interrupt the interaction, leaving the state change incomplete and the system state in an inconsistent, undesirable form. Any change to system state within a transaction boundary, therefore, has to ensure

that the change leaves the system in a stable and consistent state. A transactional unit of work is one in which the following four fundamental transactional properties are satisfied: atomicity, consistency, isolation, and durability(ACID).

Atomicity

It is common to refer to a transaction as a "unit of work." In describing a transaction as a unit of work, we are describing one fundamental property of a transaction: that the activities within it must be considered indivisible—that is, atomic. A Flute Bank customer may interact with Flute's ATM and transfer money from a checking account to a savings account. Within the Flute Bank software system, a transfer transaction involves two actions: debit of the checking account and credit to the savings account. For the transfer transaction to be successful, both actions must complete successfully. If either one fails, the transaction fails. The atomic property of transactions dictates that all individual actions that constitute a transaction must succeed for

the transaction to succeed, and, conversely, that if any individual action fails, the transaction as a whole must fail.

Consistency

A database or other persistent store usually defines referential and entity integrity rules to ensure that data in the store is consistent. A transaction that changes the data must ensure that the data remains in a consistent state—that data integrity rules are not violated, regardless of whether the transaction succeeded or failed. The data in the store may not be consistent during the duration of the transaction, but the inconsistency is invisible to other transactions, and consistency must be restored when the transaction completes.

Isolation

When multiple transactions are in progress, one transaction may want to read the same data another transaction has changed but not committed. Until the transaction commits, the changes it has made should be treated as transient state, because the transaction could roll back the change. If other transactions read intermediate or transient states caused by a transaction in progress, additional application logic must be executed to handle the effects of some transactions having read potentially erroneous data. The isolation property of transactions dictates how concurrent transactions that act on the same subset of data behave. That is, the isolation property determines the degree to which effects of multiple transactions, acting on the same subset of application state, are isolated from each other.

At the lowest level of isolation, a transaction may read data that is in the process of being changed by another transaction but that has not yet been committed. If the first transaction is rolled back, the transaction that read the data would have read a value that was not committed. This level of isolation—read uncommitted, or "dirty read"—can cause erroneous results but ensures the highest concurrency. An isolation of read committed ensures that a transaction can read only data that has been committed. This level of isolation is more restrictive (and consequently provides less concurrency) than a read uncommitted isolation level and helps avoid the problem associated with the latter level of isolation. An isolation level of repeatable read signifies that a transaction that read a piece of data is guaranteed that the data will not be changed by another transaction until the transaction completes. The name "repeatable read" for this level of isolation comes from the fact that a transaction with this isolation level can read the same data repeatedly and be guaranteed to see the same value. The most restrictive form of isolation is serializable. This level of isolation combines the properties of repeatable-read and read-committed isolation levels, effectively ensuring that transactions that act on the same piece of data are serialized and will not execute concurrently.

Durability

The durability property of transactions refers to the fact that the effect of a transaction must endure beyond the life of a transaction and application. That is, state changes made within a transactional boundary must be persisted onto permanent storage media, such as disks, databases, or file systems. If the application fails after the transaction has committed, the system should guarantee that the effects of the transaction will be visible when the application restarts. Transactional resources are also recoverable: should the persisted data be destroyed, recovery procedures can be executed to recover the data to a point in time (provided the necessary

administrative tasks were properly executed). Any change committed by one transaction must be durable until another valid transaction changes the data.

Transactions access data using two operations:

read(X), which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.

write(X), which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

Let Ti 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)
```

Transaction States :

Because failures occurs, transaction are broken up into states to handle various situation.

Active, the initial state; the transaction stays in this state until while it is still executing. A transition is terminated only if it has either been committed or aborted.

Partially committed, After the final statement has been executed At this point failure is still possible since changes may have been only done in main memory, a hardware failure could still occur. The DBMS needs to write out enough information to disk so that, in case of a failure, the system could re-create the updates performed by the transaction once the system is brought back up. After it has written out all the necessary information, it is committed.

Committed- after successful completion. Once committed, the transaction can no longer be undone by aborting it. Its effect could be undone only by a **compensating transaction**.

Failed, after the discovery that normal execution can no longer proceed. Once a transaction can not be completed, any changes that it made must be undone rolling it back.

Aborted, after the transaction has been rolled back the database has been restored to its state prior to the start of the transaction. The DBMS could either kill the transaction or **restart** the transaction. A transaction may only be restarted as a result of some hardware or software error, and a restarted transaction is considered a new transaction.

Concurrent Execution

DBMS usually allow multiple transaction to run at once.

The transaction could by run serially (one after another) or they could be interleaved(switching back and forth between transactions).

Pro:

Improved throughput and resource utilization.

Could take advantage of multi-processing.

Reduce waiting time.

Avoid short transaction waiting on long transaction

Improve average response time.

Con:

Creates many complications in data consistency, including cascading roll-backs.

Consistency could be compromise even if each individual transaction is correct.

Schedule:

A schedule for a set of transaction must consist of all the instruction of those transaction and must preserve the order in which the instructions appear in each individual transaction.

Schedule example

T1	T2
----------	----------
read(A)	
A:=A-50	
write(A)	
read(B)	
B:= B+ 50	
write(B)	
	read(A)
	temp: A * 0.1
	A: A-temp
	write (A)
	read(B)
	B:=B +temp

Schedule 1

Serial Schedule

In schedule 1 the all the instructions of T1 are grouped and run together. Then all the instructions of T2 are grouped and run together. This type of schedules are called serial. Concurrent transactions do not have to run serially as in the next examples.

Interleaved Schedule

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 2

Conflict Equivalent

Schedule 1 and 2 produce the same result even though they have different sequence. It could be shown that Schedule 2 could be transform into Schedule 1 with a sequence of swaps, so Schedule 1 and 2 conflict equivalent. Not all schedules are conflict equivalent. Consider Schedule 3

Inconsistent Schedule

T1	T2
----------	----------
read(A)	
A:=A-50	
	read(A)
	temp: A * 0.1
	A: A-temp
	write (A)
	read(B)
write(A)	
read(B)	
B:= B+ 50	
	B:=B +temp
write(B)	
	write(B)

Serializability

Suppose that A and B were bank accounts with initial amounts of $1,000 and $2,000 respectively. Then after Schedule 1 and Schedule 2 are run. The result is $850 for A and $2,150 for B. And the sum A+B is still $3000.

After Schedule 3 runs, account A is $950 and B $2100. The sum A+B is now $3,050 which is incorrect, since $50 was magically created in the process.

Conflict Serializability

Schedule 2 is consistent because is it conflict equivalent to Schedule 1. Since Schedule 1 is serial, Schedule two is said to be conflict serializable.

Instructions within a schedule could be swapped if they do not conflict. Instructions do not conflict if they access different data item or if they access the same data item none of the instructions are write instructions.

Schedule 2 could be turn into Schedule 1 with the by the following steps:

Swap write(A) of T2 with read(B) of T1

Swap read(B) of T1 with read(A) of T2

Swap write(B) of T1 with write(A) of T2

Swap write(B) of T1 with read(A) of T2

CONCURRENCY CONTROL

Lock-Based Protocols

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. Transaction can proceed only after request is granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

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

Any number of transactions can hold shared locks on an item,

but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

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.

Example of a transaction performing locking:

T2: **lock-S***(A)*;

read *(A)*;

unlock*(A)*;

lock-S*(B)*;

read *(B)*;

unlock*(B)*;

display*(A+B)*

Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong. A **Locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Neither *T3* nor *T4* can make progress — executing **lock-S***(B)* causes *T4* to wait for *T3* to release its lock on *B*, while executing **lock-X***(A)* causes *T3* to wait for *T4* to release its lock on *A*. Such a situation is called a **deadlock**. To handle a deadlock one of *T3* or *T4* must be rolled back and its locks released. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil. **Starvation** is also possible if concurrency control manager is badly designed. For example:

- A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
- The same transaction is repeatedly rolled back due to deadlocks.Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol

This is a protocol which ensures conflict-serializable schedules.

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

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

Two-phase locking *does not* ensure freedom from deadlocks. Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts. **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit. There can be conflict serializable schedules that cannot be obtained if two-phase locking is used. However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

Given a transaction *T*i that does not follow two-phase locking, we can find a transaction *Tj* that uses two-phase locking, and a schedule for *Ti* and *Tj* that is not conflict serializable.

Graph-Based Protocols

Graph-based protocols are an alternative to two-phase locking

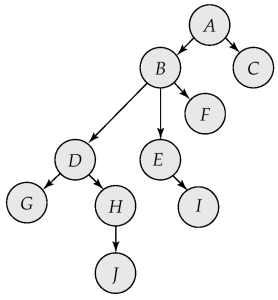
Impose a partial ordering $\rightarrow$on the set **D** = {*d1, d2 ,..., dh*} of all data items.

If *di* $\rightarrow$*dj* then any transaction accessing both *di* and *dj* must access di before accessing *dj*.

Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.

The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol



1. Only exclusive locks are allowed.

2. The first lock by *Ti* may be on any data item. Subsequently, a data *Q* can be locked by *Ti* only if the parent of *Q* is currently locked by *Ti*.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by *Ti* cannot subsequently be relocked by *Ti*
5. The tree protocol ensures conflict serializability as well as freedom from deadlock.
6. Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
    1. shorter waiting times, and increase in concurrency
    2. protocol is deadlock-free, no rollbacks are required
7. Drawbacks
    1. Protocol does not guarantee recoverability or cascade freedom
        1. Need to introduce commit dependencies to ensure recoverability
    2. Transactions may have to lock data items that they do not access.
        1. increased locking overhead, and additional waiting time
        2. potential decrease in concurrency
8. Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Multiple Granularity

Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones

Can be represented graphically as a tree (but don't confuse with tree-locking protocol)

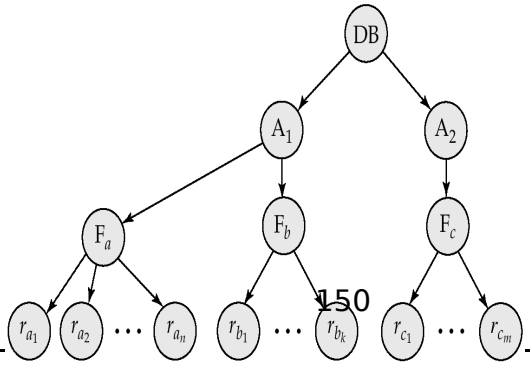
When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.

Granularity of locking (level in tree where locking is done):

fine granularity (lower in tree): high concurrency, high locking overhead

coarse granularity (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are

- database
- area
- file
- record

CRASH RECOVERY SYSTEMS

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures. Recovery algorithms have two parts

- Actions taken during normal transaction processing to ensure enough information exists to recover from failures
- Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Recovery and Atomicity

Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state. Consider transaction *Ti* that transfers \$50 from account *A* to account *B*; goal is either to perform all database modifications made by *Ti* or none at all. Several output operations may be required for *Ti* (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.

We study two approaches:

log-based recovery, and

shadow-paging

We assume (initially) that transactions 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 *Ti* starts, it registers itself by writing a <*Ti* **start**>log record

Before Ti executes **write**(*X*), a log record <*Ti, X, V1, V2*> 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 *Ti* has performed a write on data item *Xj* *Xj* had value *V1* before the write, and will have value *V2* after the write.

When *Ti* finishes it last statement, the log record <*Ti* **commit**> is written.

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 **write**s to after partial commit.

Assume that transactions execute serially

Transaction starts by writing <*Ti* ***start***> record to log.

A **write**(*X*) operation results in a log record <*Ti, X, V*> 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 *Ti* partially commits, <*Ti* **commit**> 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 <*Ti* **start**> and<*Ti* **commit**> are there in the log.

Redoing a transaction *Ti* (**redo***Ti*) sets the value of all data items updated by the transaction to the new values.

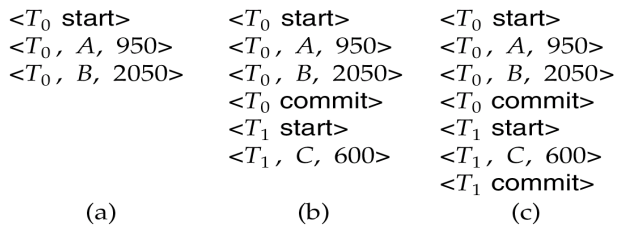
Crashes can occur while

- the transaction is executing the original updates, or
- while recovery action is being taken

example transactions *T0* and *T1* (*T0* executes before *T1*):

T0: **read** (*A*)	*T1* : **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.



If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo($T0$) must be performed since <$T0$ **commit**> is present

(c) **redo**($T0$) must be performed followed by redo($T1$) since

<$T0$ **commit**> and <Ti commit> are present

Recovery With Concurrent Transactions

We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.

- ☐ All transactions share a single disk buffer and a single log
- ☐ A buffer block can have data items updated by one or more transactions

We assume concurrency control using strict two-phase locking;

- ☐ i.e. the updates of uncommitted transactions should not be visible to other transactions
  - ➢ Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?

Logging is done as described earlier.

- ☐ Log records of different transactions may be interspersed in the log.

The checkpointing technique and actions taken on recovery have to be changed

- ☐ since several transactions may be active when a checkpoint is performed.

Checkpoints are performed as before, except that the checkpoint log record is now of the form

< **checkpoint** L>

where L is the list of transactions active at the time of the checkpoint

- ☐ We assume no updates are in progress while the checkpoint is carried out (will relax this later)

When the system recovers from a crash, it first does the following:

- Initialize *undo-list* and *redo-list* to empty
- Scan the log backwards from the end, stopping when the first <**checkpoint** *L*> record is found.

  For each record found during the backward scan:

  - if the record is <*Ti* **commit**>, add *Ti* to *redo-list*
  - if the record is <*Ti* **start**>, then if *Ti* is not in *redo-list*, add *Ti* to *undo-list*
- For every *Ti* in *L*, if *T*i is not in *redo-list*, add *Ti* to *undo-list*

At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.

Recovery now continues as follows:

- Scan log backwards from most recent record, stopping when <*Ti* **start**> records have been encountered for every *Ti* in *undo-list*.
  - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
- Locate the most recent <**checkpoint** *L*> record.
- Scan log forwards from the <**checkpoint** *L*> record till the end of the log.
  - During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

Example of Recovery

Go over the steps of the recovery algorithm on the following log:

<*T*0 **start**>

<*T*0, *A*, 0, 10>

<*T*0 **commit**>

<*T*1 **start**>

<*T*1, *B*, 0, 10>

<*T*2 **start**> /* Scan in Step 4 stops here */

<*T*2, *C*, 0, 10>

<*T*2, *C*, 10, 20>

<checkpoint {*T*1, *T*2}>

<*T*3 **start**>

<*T*3, *A*, 10, 20>

<*T*3, *D*, 0, 10>

<*T*3 **commit**>

DECISION SUPPORT

Database management systems are widely used by organizations for maintaining data that documents their everyday operations. In applications that update such operational data, transactions typically make small changes and a large number of transactions must be reliably and efficiently processed. Such **online transaction processing (OLTP)** applications have driven the growth of the DBMS industry in the past three decades and will doubtless continue to be important. DBMSs have traditionally been optimized extensively to perform well in such applications. Recently, however, organizations have increasingly emphasized applications in which current and historical data are comprehensively analyzed and explored, identifying useful trends and creating summaries of the data, in order to support high-level decision making. Such applications are referred to as **decision support**.

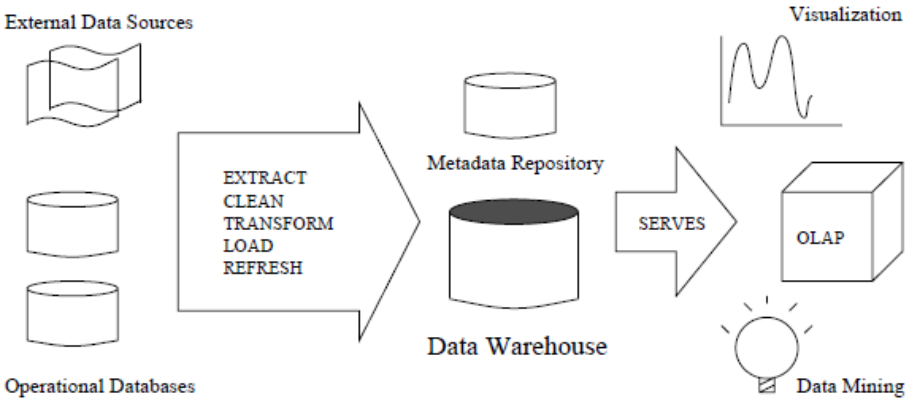
DATA WAREHOUSING

Warehouses are much larger than other kinds of databases; sizes ranging from several gigabytes to terabytes are common. Typical workloads involve ad hoc, fairly complex queries and fast response times are important. These characteristics differentiate warehouse applications from OLTP applications, and different DBMS design and implementation techniques must be used to achieve satisfactory results. A distributed DBMS with good scalability and high availability is required for very large warehouses.

An organization's daily operations access and modify **operational databases**. Data from these operational databases and other external sources are **extracted** by using *gateways*, or standard external interfaces supported by the underlying DBMSs.

A **gateway** is an application program interface that allows client programs to generate SQL statements to be executed at a server.

A Typical Data Warehousing Architecture



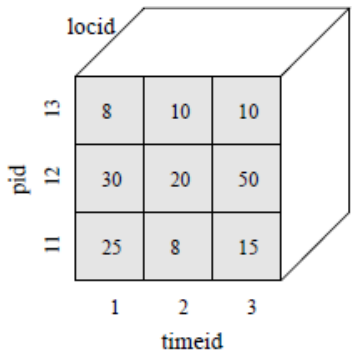
Creating and Maintaining a Warehouse

Data is **extracted** from operational databases and external sources, **cleaned** to minimize errors and fill in missing information when possible, and **transformed** to reconcile semantic mismatches. Transforming data is typically accomplished by defining a relational view over the tables in the data sources. **Loading** data consists of materializing such views and storing them in the warehouse. Unlike a standard view in a relational DBMS, therefore, the view is stored in a database that is different from the database(s) containing the tables it is defined over. The cleaned and transformed data is finally **loaded** into the warehouse.

An important task in maintaining a warehouse is keeping track of the data currently stored in it; this bookkeeping is done by storing information about the warehouse data in the system catalogs. The system catalogs associated with a warehouse are very large and are often stored and managed in a separate database called a **metadata repository**.

Multidimensional Data Model

In the multidimensional data model, the focus is on a collection of numeric measures. Each measure depends on a set of dimensions. The measure attribute in our example is sales. The dimensions are Product, Location, and Time.



OLAP systems that use arrays to store multidimensional datasets are called **multidimensional OLAP (MOLAP)** systems. This relation, which relates the dimensions to the measure of interest, is called the **fact table**.

locid	*city*	*state*	*country*
1	Ames	Iowa	USA
2	Chennai	TN	India
5	Tempe	Arizona	USA

Locations

pid	*pname*	*category*	*price*
11	Lee Jeans	Apparel	25
12	Zord	Toys	18
13	Biro Pen	Stationery	2

Products

pid	*timeid*	*locid*	*sales*
11	1	1	25
11	2	1	8
11	3	1	15
12	1	1	30
12	2	1	20
12	3	1	50
13	1	1	8
13	2	1	10
13	3	1	10
11	1	2	35
11	2	2	22
11	3	2	10
12	1	2	26
12	2	2	45
12	3	2	20
13	1	2	20
13	2	2	40
13	3	2	5

Sales

Views And De

Views nt groups of analysts within an organization are typically concerned with different aspects of the business, and it is convenient to define views that give each group insight into the business details that concern them. Once a view is defined, we can write queries or new view definitions that use it.

Views, OLAP, andWarehousing

Views are closely related to OLAP and data warehousing.

Views and OLAP: OLAP queries are typically aggregate queries. Analysts want fast answers to these queries over very large datasets, and it is natural to consider precomputing views. The idea is to choose a subset of the aggregate queries for materialization in such a way that typical CUBE queries can be quickly answered by using the materialized views and doing some additional computation. The choice of views to materialize is influenced by how many queries they can potentially speed up and by the amount

of space required to store the materialized view.

Views and Warehousing: A data warehouse is just a collection of asynchronously replicated tables and periodically maintained views. A warehouse is characterized by its size, the number of tables involved, and the fact that most of the underlying tables are from external, independently maintained databases.

CASE STUDY : THE INTERNET SHOP

REQUIREMENTS ANALYSIS

The owner of B&N has thought about what he wants and orders a concise summary: I would like my customers to be able to browse my catalog of books and to place orders over the Internet. Currently, I take orders over the phone. I have mostly corporate customers who call me and give me the ISBN number of a book and a quantity. I then prepare a shipment that contains the books they have ordered. If I don't have enough copies in stock, I order additional copies and delay the shipment until the new copies arrive; I want to ship a customer's entire order together. My catalog includes all the books that I sell. For each book, the catalog contains its ISBN number, title, author, purchase price, sales price, and the year the book was published. Most of my customers are regulars, and I have records with their name, address, and credit card number. New customers have to call me first and establish an account before they can use my Web site.

On my new Web site, customers should first identify themselves by their unique customer identification number. Then they should be able to browse my catalog and to place orders online."

CONCEPTUAL DESIGN

In the conceptual design step, DBDudes develop a high level description of the data in terms of the ER model. DBDudes has an internal design review at this point, and several questions are raised. To protect their identities, we will refer to the design team leader as Dude 1 and the

design reviewer as Dude 2:

Dude 2: What if a customer places two orders for the same book on the same day?

Dude 1: The first order is handled by creating a new Orders relationship and the second order is handled by updating the value of the quantity attribute in this relationship.

Dude 2: What if a customer places two orders for different books on the same day?

Dude 1: No problem. Each instance of the Orders relationship set relates the customer to a different book.

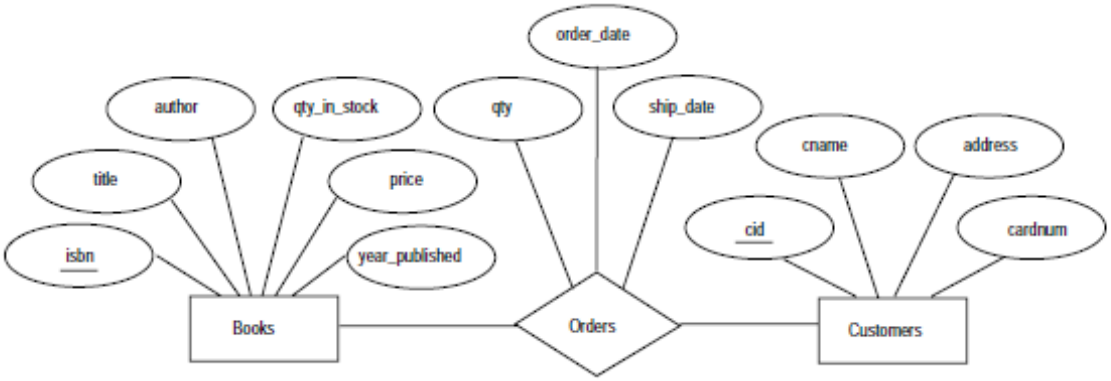
Dude 2: Ah, but what if a customer places two orders for the same book on different days?

Dude 1: We can use the attribute order date of the orders relationship to distinguish the two orders.

Dude 2: Oh no you can't. The attributes of Customers and Books must jointly contain a key for Orders. So this design does not allow a customer to place orders for the same book on different days.

Dude 1: Yikes, you're right. Oh well, B&N probably won't care; DBDudes decides to proceed with the next phase, logical database design.

LOGICAL DATABASE DESIGN



CREATE TABLE Books (isbn CHAR(10),
title CHAR(80),
author CHAR(80),
qty in stock INTEGER,
price REAL,
year published INTEGER,
PRIMARY KEY (isbn))

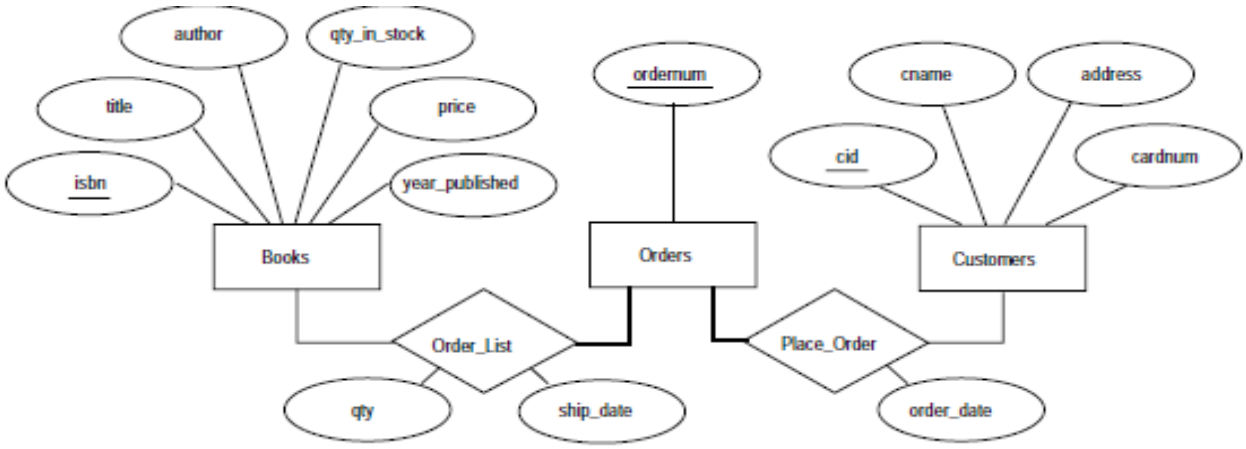
CREATE TABLE Orders (isbn CHAR(10),
cid INTEGER,
qty INTEGER,
order date DATE,
ship date DATE,
PRIMARY KEY (isbn,cid),
FOREIGN KEY (isbn) REFERENCES Books, FOREIGN KEY (cid) REFERENCES Customers)

CREATE TABLE Customers (cid INTEGER,
cname CHAR(80),
address CHAR(200),
cardnum CHAR(16),
PRIMARY KEY (cid)
UNIQUE (cardnum))

PHYSICAL DATABASE DESIGN

Next, DBDudes considers the expected workload. The owner of the bookstore expects most of his customers to search for books by ISBN number before placing an order. Placing an order involves inserting one record into the Orders table and inserting one or more records into the Orderlists relation. If a sufficient number of books is available, a shipment is prepared and a value for the *ship date* in the Orderlists relation is set. In addition, the available quantities of books in stocks changes all the time

since orders are placed that decrease the quantity available and new books arrive from suppliers and increase the quantity available.



ER Diagram Reflecting the Final Design

SECURITY

Returning to our discussion of the initial design phase, recall that DBDudes completed physical database design. Next, they address security. There are three groups of users: customers, employees, and the owner of the book shop.

The owner of the store has full privileges on all tables. Customers can query the Books table and can place orders online, but they should not have access to other customers' records nor to other customers' orders

SELECT ON Books, NewOrders, OldOrders, NewOrderlists, OldOrderlists

INSERT ON NewOrders, OldOrders, NewOrderlists, OldOrderlists

Employees should be able to add new books to the catalog, update the quantity of a book in stock, revise customer orders if necessary, and update all customer information *except the credit card information*. Thus, DBDudes creates the following view:

CREATE VIEW CustomerInfo (cid,cname,address)

AS SELECT C.cid, C.cname, C.address

FROM Customers C

They give the employee account the following privileges:

SELECT ON CustomerInfo, Books,

NewOrders, OldOrders, NewOrderlists, OldOrderlists

INSERT ON CustomerInfo, Books,

NewOrders, OldOrders, NewOrderlists, OldOrderlists

UPDATE ON CustomerInfo, Books,

NewOrders, OldOrders, NewOrderlists, OldOrderlists

DELETE ON Books, NewOrders, OldOrders, NewOrderlists, OldOrderlists

QUESTION BANK

PART – A (2 Marks)

UNIT – I

1. What is DBMS?

Redundancy is controlled, Unauthorised access is restricted, Providing multiple user interfaces, Enforcing integrity constraints, Providing backup and recovery

2. **What is a Database system?**

The database and DBMS software together is called as Database system

3. Disadvantage in File Processing System?

Data redundancy & inconsistency, Difficult in accessing data, Data isolation, Data integrity, Concurrent access is not possible, Security Problems.

4. Compare and contrast file processing system and a DBMS.

In a file processing system, one has to define the storage structure, access strategy. While we don't have to define the storage structure and access strategy in DBMS. DBMS also avoids redundancy of data to certain extent. It enables to have data independence which does not exist in a file system.

5. Differentiate between strong entity and weak entity with an example.

Strong Entity Set refers to the table wherein we can have a primary key. **Weak Entity Set** is one that does not have the primary key. A table that involves only foreign keys derived from several tables.

6. What is database?

A database is a logically coherent collection of data with some inherent meaning, representing some aspect of real world and which is designed, built and populated with data for a specific purpose

7. Define full outer join.

Full outer join does both the operations of **left Outer join** and **right Outer join** , padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation and adding them to the result of the join.

8. What do you mean by hybrid hash join.

When the join operation is performed on two or more relations by applying the hash function along with the merge algorithm, we obtain hybrid hash join.

9. Differentiate network model from hierarchical model.

In network model, the structure is represented through several complicated links; in hierarchical model, the structure of a table is represented in the form of a tree. With relational model, the M – M mapping is achieved though referential integrity constraint. But, in a network model, such M – M relation has to be decomposed in to a set of M – 1 relations. Representation of M - M mapping is possible in hierarchical model.

10. Write the sequence of operation to delete a segment in hierarchical model.

The segment to be deleted must first be retrieved via one of the "get hold" operations – "get hold unique" (GHU), "get hold next" (GHN) or " get hold next within parent" (GHNP). The delete may then be issued.

11. Define disjoint constraint, overlapping constraint, completeness constraint.

Disjoint constraint - requires that an entity belong to no more than one lower – level entity set. **Overlapping constraint**- the same entity may belong to more than one lower – level entity set within a single generalization. **Completeness** constraint **–** specifies whether or not an entity in the higher – level entity in the higher – level entity set must belong to at least one of the lower level entity sets within the generalization / specialization.

12. List two reasons why null values might be introduced into the database.

When the user does not know the details of some of the fields in a particular table, we can introduce null values. Under those circumstances, where the value does not exist. An unknown value may either be **missing or not known.**

13. Define equi join.

Equi join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation with the equality condition applied.

14. Why relational DBMS are popular than hierarchical and network DBMS.

The structure of the table is the same as we create; where as in network model, the structure is represented through several complicated links; in hierarchical model, the structure of a table is represented in the form of a tree. With relational model, the M – M mapping is achieved though referential integrity constraint. But, in a network model, such M – M relation has to be decomposed in to a set of M – 1 relations.

15. What is meant by data abstraction? What are the three levels of abstraction?

A **data abstraction** is an idea that concentrates on the essential properties of something rather than on the concrete realization or actual cases. **Levels of abstraction** – physical, conceptual, logical abstraction. (1) **Physical** level of abstraction is one that defines the storage structure of all the fields in a table. (2) **Conceptual** level acts as an interface between the physical and logical level of data. (3) **Logical** level is the user's view of the external database.

16. Define the terms relation and field, relationship

Relation refers to the association that exists among two or more tables. **Attribute/field** - defines the properties of an entity. **Relationship** is association among several entities.

17. How B- trees are different from other tree data structure?

B – trees guarantee that the insertion / deletion will keep the tree balanced.

18. What is the difference between network and hierarchical model?

The structure of the table is the same as we create; Where as in network model, the structure is represented through several complicated links; in hierarchical model, the structure of a table is represented in the form of a tree. With relational model, the M – M mapping is achieved though referential integrity constraint. But, in a network model, such M – M relation has to be decomposed in to a set of M – 1 relations.

19. What is hashing?

Each stored record occurrence is placed in the database at a location whose address may be computed by some function called the hash function of a value that appears in that occurrence – usually the primary key value.

20.Why do you need data normalization?

Normalization enables us to avoid redundancy to a certain extent. It achieves controlled redundancy on data items. It also ensures a good database design.

21. Distinguish between primary and foreign keys.

Primary Key – a field that is unique and not null. **Candidate Key** is one that has the capacity of behaving like a primary key. **Foreign Key** – a field that establishes the relationship between two / more tables. The primary key that is used to refer the details in a master table.

22. Why a member record of a set occurrence can not participate in more than one occurrence of the set at any point?

A member record of a set occurrence can not participate in more than one occurrence of the set because it is very difficult to establish M –M mapping in the network model. Such representation has to be decomposed into M - !.

23.What is meant by safety of expressions in Domain Relational Calculus?

It is possible to write expressions that may generate an **infinite** relation. An expression such as {<l, b, a > ! ┐(<l, b, a> € loan)} is unsafe because it allows values in the result that are not in the domain of the expression. Therefore, we must consider the form of the formulae within "there exists "and "for all "clauses.

24. How are the aggregate structure expressed in SEQUEL?

Aggregate functions are functions that take a collection of values as input and return a single value. Such aggregate functions are avg, max, min, sum and count.

UNIT-II

1. What is a trigger?

A piece of logic written in PL/SQL and Executed at the arrival of a SQL*FORMS event

2. What is an Entity?

It is a 'thing' in the real world with an independent existence

3. What is an Entity type?

It is a collection (set) of entities that have same attributes

4. What is an Entity set?

It is a collection of all entities of particular entity type in the database

5. What is an Extension of entity type?

The collections of entities of a particular entity type are grouped together into entity set

6. What is Weak Entity set?

An entity set may not have sufficient attributes to form a primary key, and its primary key compromises of its partial key and primary key of its parent entity, then it is said to be Weak Entity set

7. What is Relationship?

It is an association among two or more entities

8. What is Relationship set?

The collection (or set) of similar relationships

9. What is Relationship type?

Relationship type defines a set of associations or a relationship set among a given set of entity types

10. What is degree of Relationship type?

It is the number of entity type participating

11. What is Relational Algebra?

It is procedural query language It consists of a set of operations that take one or two relations as input and produce a new relation

12. What is Relational Calculus?

It is an applied predicate calculus specifically tailored for relational databases proposed by EF Codd Eg of languages based on it are DSL ALPHA, QUEL

13. How does Tuple-oriented relational calculus differ from domain-oriented relational calculus?

The tuple-oriented calculus uses a tuple variables ie, variable whose only permitted values are tuples of that relation Eg QUEL. The domain-oriented calculus has domain variables ie, variables that range over the underlying domains instead of over relation Eg ILL, DEDUCE

14. What do you mean by transaction management in database systems?

The transaction management can be done with the help of the transaction management component of the database routine that ensures ACID properties of a transaction, conflict serializability, view serializability, recoverability, cascade less schedules, etc.

15. Define logical schema in database system.

A logical schema describes the database design at the logical level.

16. List two important properties of B- Trees.

Within a node all the keys are arranged in a predefined order. All keys in the sub tree to the left of a key are predecessors and the same to the right are successors of the key. Each internal node may have at the most (m –1) keys and the node will have one more child than the number of keys it has.

17. Define dense index.

An index controlled by the DBMS is a dense, secondary index. It contains an entry fro every stored record occurrence in the indexed file.

18. Define network model.

The data is represented as records and links. It is a more general structure that allows to have any number of immediate set occurrences.

19. What are the drawbacks of hierarchical data model?

No dependent record occurrence can exist without its superior. It is possible to insert a new data to a child record without introducing a special dummy master record. If a master record is deleted that is having the entire details of the entity, then such information is lost.

20. What are the basic operations in relational algebra?

The fundamental operations of relational algebra are select (σ) , project (∏) , union, set difference (-) , Cartesian product (X) and rename.

21. Define tuple relational calculus.

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

22. What is the need for data normalization?

Normalization enables us to avoid redundancy to a certain extent. It achieves controlled redundancy on data items. It also ensures a good database design.

23. What is the purpose of views in database system?

Views are the temporary relations that are created or they are the virtual tables. Views behave like the tables. Unless otherwise specified, any changes that are performed in the views are not reflected in the original tables.

24. "The execution of full outer join query results in a Cartesian product" – justify.

Whenever a join operation is performed on two or more tables without having a common field, it 'll result in a cartesian product.

25. Define PJNF.

Project join normal form results in joining several projections derived from a given relation R. During the join, if dangling tuples are obtained, such tuples are to be eliminated by the final join.

26. Define Armstrong's axioms.

Armstrong axioms are used for finding the closure of a set of functional dependencies. (1) **Reflexivity rule -** If α is a set of attributes and β € α then α → β holds. (2) **Augmentation rule -** If α → β holds and γ is a set of attributes, then γ α → γ β holds (3) **Transitivity rule - I**f α → β holds and β → γ holds, then α → γ holds

27. What do you mean by Multi valued dependencies?

MVD s are a generalization of functional dependencies in such a way that all fields are multi valued attributes.

28. Define database management systems.

DBMS is the intermediate layer that acts as the interface between physical Vs logical data.

29. What do you mean by data base?

Data base refers to the collection of inter related data with respect to a particular entity.

30. What do you mean by attribute, row, constraints, domain, table,

Attribute / field - one that defines the properties of an entity. **Tuple / row/ record** – one that gives the relevant details of a particular instance. **Domain** is a pool of values out of which a particular value is retrieved at a given instance of time. **Constraints** are the conditions one imposes on a particular entity.

31. Mention any two differences between procedural and non procedural language.

Procedural Language - a language that has declaration part, control structures and input / output statements. E.g., C, COBOL, PL/1. **Non procedural Language** – a language that does not have the features of a procedural language. It resembles like English sentences.

32. Define data dictionary, distributed database.

Data Dictionary –contains metadata – raw data or data about data. **Distributed Database** – accessing of remote databases that are stored at geographically wide spread area.

33. Mention about various types of attributes.

Single valued attributes that have a single value for a particular entity. **Multi valued attributes** that can have several values. **Simple attributes** that are not divided into sub parts.

Composite attributes can be divided into subparts. **Derived attributes** are those values that can be derived from other related attributes or entities.

UNIT-III

1. What do you mean by recursive partitioning?

If the value of n_h is greater than or equal to the number of page frames of memory, the relations can not be partitioned in one pass, since there will not be enough buffer pages.

2. What do you mean by a key? Mention the various classification of keys.

Key is that field based on which searching is done. (1) **Super Key** – a set of one or more attributes that allows one to identify uniquely an entity in the entity set. (2) **Primary Key** – a field that is unique and not null. (3) **Candidate Key** is one that has the capacity of behaving like a primary key. (4) **Foreign Key** – a field that establishes the relationship between two / more tables. (5) **Composite Key** – a field that comprises of several key fields. (6) **Descriptive attributes** – attributes that describes a particular course of action.

3. Define schema, instance and sub schema, entity.

Schema defines the overall design of the entity. **Instance** refers to the collection of information stored in a database at a particular instant of time. **Subschema** – When we derive several schemas from the database at the view level, it is called as subschema. i.e., it describes the different views of the database.

4. Define data model.

Data Models describe the collection of conceptual tools for describing data, data relationships, data semantics and consistency constraints.

5. Define redundancy, consistency, inconsistency.

Redundancy is one where duplication of information occurs. **Consistency** refers to storing of same kind of information for a particular field in several tables. **Inconsistency** refers to the presence of irrelevant data with respect to a particular field.

6. Define entity, entity set, relationship set.

Entity refers to the object of particular interest. **Entity set** refers to the set of all entities of the same type. **Relationship set** defines the set of all relationships of the same type

7. Differentiate between strong entity set and weak entity set.

Strong Entity Set refers to the table wherein we can have a primary key. **Weak Entity Set** is one that does not have the primary key. A table that involves only foreign keys derived from several tables.

8. What do you mean by mapping cardinalities.

Mapping cardinalities enables one to express the number of entities to which another entity can be associated via a relationship set. Such mapping can either be one – to – one , one – to- many or many – to – many.

9. Why do you use database languages? Define DDL, DML and TCL.

Database languages enable us to define, retrieve and control the databases. **(1) Data definition language (DDL) - is** used to specify a database schema by a set of definitions. (2) **Data Manipulation Language (DML) - is** used to manipulate the data that is created by DDL. (3) **Data Control Language (DCL).**

10. Define query, query processor, query language.

Query is a statement requesting the retrieval of data. **Query Language** is that portion of the DML that involves information retrieval. **Query Processor** breaks down DML statements into instructions that the storage manager understands.

11. Define data mining, data warehousing.

Data mining - refers to detecting various types of patterns in large volumes of data. **Data Warehousing** - gathering data from multiple sources under unified schema, at a single site.

12. What do you mean by query optimization?

Query Optimization - 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.

13. Differentiate between 2 – tier and 3 – tier architecture.

Two – tier architecture - the application is partitioned into two parts such as user interface and the database system that resides at the server. **Three – tier architecture**- the application is partitioned into three parts such as user interface, the business logic and the database system that resides at the server.

14. What are the properties of a transaction?

ACID – **A**tomicity, **C**onsistency, **I**solation, **D**urability properties that should be present in a transaction.

15. Define participation and its classification.

Participation – the association between entity sets **(1) Total** - Every entity in E participates in at least one relationship in R **(2) Partial** - Only some of the entities in E participate in relationships in R.

16. What is a discriminator?

Discriminator / Partial key – a set of attributes that allows the distinction to be made.

17. What do you mean by connection trap?

Connection Trap refers to those inferences that can not be derived from the database. Eg., supplier S1 supplies part P1 and such part is used in job J2.

We derive from the above statement, the following facts – (1) Supplier S1 supplies part P1 (2) Part P1 is used in job J2. But, from statement 1, we can not find out , part P1 is used in which job?

18. What do you mean by specialization, generalization?

Specialization – the process of designating sub groupings within an entity set. **Generalization** – the commonality found among several entity sets.

19. Define DBA, indices, data files, null value and degree.

Database Administrator (DBA) - that controls the overall functioning of the system. **Indices** – provide faster access to data items that hold a particular value. **Data files** – Store the database itself. **Null value** – the value does not exist for an n attribute or the value is unknown. **Degree** – the number of entity sets that participate in a relationship set.

20. What is a B*-tree index?

Ans: B*-tree index is a binary tree structure that can be used to find data quickly A binary comparison is made and the tree is traversed based on that selection

21. What Is SQL?

SQL, SEQUEL (Structured English Query Language), is a language for RDBMS (Relational Database Management Systems). SQL was developed by IBM Corporation.

UNIT-IV

1. What is normalization?

It is a process of analysing the given relation schemas based on their Functional Dependencies (FDs) and primary key to achieve the properties. * Minimizing redundancy, * Minimizing insertion, deletion and update anomalies

2. What is Functional Dependency?

A Functional dependency is denoted by X Y between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuple that can form a relation state r of R The constraint is for any two tuples t1 and t2 in r if t1[X] = t2[X] then they have t1[Y] = t2[Y] This means the value of X component of a tuple uniquely determines the value of component Y .

3. When is a functional dependency F said to be minimal?

Every dependency in F has a single attribute for its right hand side. * We cannot replace any dependency X A in F with a dependency Y A where Y is a proper subset of X and still have a set of dependency that is equivalent to F. * We cannot remove any dependency from F and still have set of dependency that is equivalent to F

4. What is Multivalued dependency?

Multivalued dependency denoted by X Y specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation r of R: if two tuples t1 and t2 exist in r such that t1[X] = t2[X] then t3 and t4 should also exist in r with the following properties

* t3[x] = t4[X] = t1[X] = t2[X]

* t3[Y] = t1[Y] and t4[Y] = t2[Y]

* t3[Z] = t2[Z] and t4[Z] = t1[Z]

where [Z = (R-(X U Y))]

5. Define role, inheritance.

Role - the function that an entity plays in a relationship. **Total** – each higher – level entity must belong to a lower – level entity set. **Partial** – some higher level entities may not belong to any lower – level entity set. **Inheritance** - .the attributes of a higher – level entity may be derived by the lower level entity sets.

6. What do you mean by constraint? Mention various categories.

Constraint refers to the boundary conditions imposed on a table. **Condition – defined constraints** – membership is evaluated on the basis of whether or not an entity satisfies an explicit condition or predicate. **User – defined constraints** – are not constrained by a membership condition, rather the database user assigns entities to a given entity set.

7. Differentiate between aggregation and aggregate functions.

Aggregation – an abstraction through which relationships are treated as higher level entities. **Aggregate functions** – take a collection of values and returns a single value as a result.

8. Define SQL and its various classification.

Structured Query Language (SQL) a nonprocedural language used to define, manipulate and control databases (1) **Interactive SQL** – the query is responded by the system immediately. (2) **Embedded SQL** – the query is embedded into any one of the host languages like C, COBOL, PL/1 (3) **Dynamic SQL** allows programs to construct and submit SQL queries at run time.

9. What do you mean by join? Explain about various classification of joins.

Join - used to combine data from two / more tables. **Natural join** is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. **Left Outer join** – takes all tuples in the left relation that did not match with any tuples in the right relation, pads the tuples with null values for all other attributes from the right relation. **Right Outer join** pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join. **Theta join** - an extension to natural join operation that allows us to combine a selection and a Cartesian product into a single operation.

10. Differentiate between ODBC and JDBC

ODBC – defines a way for an application program to communicate with a database server. **JDBC** defines an API (Application Program Interface) that Java programs can use to connect to database servers.

11. What do you mean by QBE and Datalog?

QBE Queries look like tables (i.e., **skeleton tables**) and are expressed "**by example**". **GQBE** queries designed for a graphical display environment.

E.g., Access QBE **Data log** – non procedural query language based on the logic programming language Prolog.

12. What do you mean by primary key constraint, referential integrity constraint?

Primary key constraint demands that the value that is stored should be **unique and not null. Referential integrity constraint** demands that the primary key of the master table should act as the foreign key in the detail relation.

13. What do you mean by a trigger?

Trigger - a statement that the system executes automatically as a side effect of a modification to the database.

14. What do you mean by data independence?

Data independence is one where in a table, the data dependence will not occur. **Data Dependence** refers to the modification of either the data type, structure or strategy that results in the change of the application code.

15. What do you mean by atomicity, stored procedure and view?

Atomicity refers to storing of unique information in a particular field. But, achieving atomicity is impractical. **Stored Procedures** – procedures stored in the database and executed later by **Call** statement. **View** - temporary relations with the restrictions on updates.

16. What do you mean by static hashing?

The use of hash functions in which the set of bucket addresses is fixed is called static hashing. It can not accommodate databases that grow significantly larger over time.

17. What do you mean by extendable hashing?

Contrary to static hashing, this extendable hashing copes with the changes in database size by splitting and coalescing buckets as the database grows and shrinks.

18. What do you mean by a skew?

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

19. Mention different levels of RAID technology.

Level 0 – block striping, no redundancy. Level 1 – block striping, mirroring. Level 3 - bit striping, parity. Level 5 - block striping, distributed parity. Level 6 - block striping, P + Q redundancy.

20. What do you mean by striping?

Striping refers to splitting the data across multiple disks.

21. What do you mean by bit – level striping?

Bit – level striping refers to that data striping which consists of splitting the bits of each byte across multiple disks.

22.What do you mean by block -level striping?

Block level striping stripes the blocks across multiple disks. It treats the array of disks as a single large disk and it gives blocks logical numbers.

23. What do you mean by mirroring / shadowing?

Mirroring refers to duplicate every disk. A logical disk consists of two physical disks and every write is carried out on both disks.

24. Define jukeboxes.

Jukeboxes are devices that store a large number of optical disks and load them automatically on demand to one of a small number of drives (between 1 – 10) the aggregate storage capacity of such a system can be many terabytes.

25. Give examples for primary storage, secondary storage / online storage , tertiary storage / offline storage

Cache and main memory – primary storage. Magnetic disks - secondary storage / online storage. Magnetic tape, optical disk jukeboxes - tertiary storage / offline storage

26.What is 1 NF (Normal Form)?

The domain of attribute must include only atomic (simple, indivisible) values.

27.What is Fully Functional dependency?

It is based on concept of full functional dependency A functional dependency X Y is full functional dependency if removal of any attribute A from X means that the dependency does not hold any more

27.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

28.What is 3NF?

A relation schema R is in 3NF if it is in 2NF and for every FD X A either of the following is true. * X is a Super-key of R. * A is a prime attribute of R In other words, if every non prime attribute is non-transitively dependent on primary key

29. What is BCNF (Boyce-Codd Normal Form)?

A relation schema R is in BCNF if it is in 3NF and satisfies an additional constraint that for every FD X A, X must be a candidate key

30.What is 4NF?

A relation schema R is said to be in 4NF if for every Multivalued dependency X Y that holds over R, one of following is true. * X is subset or equal to (or) XY = R. * X is a super key

31.What is 5NF?

A Relation schema R is said to be 5NF if for every join dependency {R1, R2, , Rn} that holds R, one the following is true. * Ri = R for some i. * The join dependency is implied by the set of FD, over R in which the left side is key of R

UNIT-V

1. What is a partitioned table?

A partitioned table uses the new Oracle feature, range partitioning Data is stored in a location based on a range of data you have defined This range can be numeric or a data format

2. What is a nested table?
A nested table is a table that appears as a column in another table is called as nested table in database management system

3. What is a partitioned table?
A partitioned table is a table where the data is divided into smaller pieces based on the data itself is called partitioned table in database management system.

4. How are tables partitioned?
DBMS supports range partitioning The data is divided into various chunks based on ranges of data in one or more columns

5. What do you mean by hybrid hash join?

A hybrid hash join refers to the application of hash join along with merge join algorithm on two / more relations.

6. What is extension and intension?

Extension -It is the number of tuples present in a table at any instance This is time dependent. Intension - It is a constant value that gives the name, structure of table and the constraints laid on it

7. Define hot swapping.

Hot swapping refers to the removal of faulty disks and replacement of new ones without turning power off.

8. Differentiate between volatile and non - volatile storage.

Volatile storage loses its contents when the power to the device is removed. While in non – volatile storage, information is not lost even when the power is switched off.

9. What do you mean by clustering file organization?

If records of several different relations are stored on the same file and related records of the different relations are stored on the same block so that one. I /O operation fetches related records from all the relations, it is called as clustering file organization.

10. Define anchor block, overflow block.

Anchor block-which contains the first record of a chain. **Overflow block**-which contains records other than those that are the first record of a chain.

11. Define dangling pointer.

A dangling pointer is a pointer that does not point to a valid object.

12. What do you mean by dereferencing?

The action of looking up an object, given its identifiers is called dereferencing.

13. Define dangling / spurious tuples.

Any additional tuples that are generated as a result of executing joins over intermediate projections.

14. Define sparse index.

An index record appears for only some of the search key values. Each index record contains a search key value and a pointer to the first data record with that search – key value.

15. Define closed hashing, open hashing.

Closed hashing is one where there are overflow chains. Open hashing is one where the set of buckets is fixed and there are no overflow chains.

16. Define nested – loop join.

When a theta join is performed on two relations r and s such that relation r as the outer relation and s as the inner relation, it is called nested loop join.

17. Define indexed – nested loop join.

In a nested loop join, if an index is available on the inner loop's join attribute, index lookups can replace file scans. For each tuple t_r in the outer relation r, the index is used to look up tuples in s that will satisfy the join condition with tuple t_r

18. Define block nested loop 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 outer block, to generate all pairs of tuples.

19. What do you mean by merge join.

Merge join can be used to compute natural joins and equi - joins. Suppose that both relations are sorted on the attributes R intersection S. then, their join can be computed by a process similar to the merge stage in the merge sort algorithm.

20. Define hash join.

A hash function h is used to partition tuples of both relations .i.e, to partition the tuples of each of the relations into sets that have the same hash value on the join attributes.

21. What Are the Oracle Built-in Data Types?

There are 20 Oracle built-in data types, divided into 6 groups:

- Character Datatypes - CHAR, NCHAR, NVARCHAR2, VARCHAR2
- Number Datatypes - NUMBER, BINARY_FLOAT, BINARY_DOUBLE
- Long and Row Datatypes - LONG, LONG RAW, RAW
- Date time Datatypes - DATE, TIMESTAMP, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND
- Large Object Datatypes - BLOB, CLOB, NCLOB, BFILE
- Row ID Datatypes - ROWID, UROWID

PART-B (16 Marks)

UNIT-I

1. Compare file based data management with database based data management?

   Data redundancy & inconsistency

   Difficulty in accessing data

   Data Isolation

   Concurrent access anomalies

   Security Problems

   Integrity Problems

   Advantages of Database

2. Discuss in detail about the responsibilities of DBA.

   Define Database Administrator

   Functions of DBA

   Schema Definition

   Schema and Physical organization modification

   Granting of authorization for data access

   Routine maintenance

3. Discuss in detail about the overall structure of DBMS with the neat sketch.

   the storage manager

      The storage manager components

      Authorization and integrity manager

      Transaction manager

      File Manager

      Buffer manager

      Indices

query processor components.

DDL interpreter,

DML Compiler and Query optimizer

Query evaluation engine

4. Explain entities, entity relationship and mappings? Draw the E-R diagram of a banking enterprise

Define : Entity

Entity Set

Attributes

Types of Attributes

Relationship

Relationship set

Types of relationships

Constraints

Mapping Cardinalities

Participation Constraints

Draw ER diagram for banking enterprise

5. Explain in detail about various kinds of data models.

Relational Model

Entity Relationship Model

Object Oriented Data Model

Object Relational data Model

Hierarchical data model

Network data model

6. Discuss the history of DBMS.

1950's and early 1960's

Late 1960's and 1970's

1980's

Early 1990's

Late 1990's

Early 2000's

7. Explain Database Languages.

Data Definition Language

Types of Constraints

Authorization

Types of authorization

Data Manipulation Lnaguage

Procedural DML'S

Non Procedural DML'S

8.Explain ER model in detail

Attribute set

Entity set

Mapping cardinalities

Relationship sets

Keys

9. Explain about Integrity and Security in database management systems

Types Of Constraints

Domain Constraints

Authorization

Types Of Authorization

Security

Security Levels

10. Draw an ER diagram for a Hospital

Data requirements

Entity sets

Relationship sets

ER diagram

UNIT – II

1. What is relational algebra? How is it used in DBMS?

   Define Relational Algebra

   The Role of Relational Algebra in a DBMS

   Additional operations

   Selection Operator

   Projection Operator

   Set Operations

   Cartesian (Cross) Product Operator

   Renaming operator

   Join Operator

2. Explain the form of a basic SQL query with examples.

   The from Clause

   The Rename Operation

   Tuple Variables

   String Operations

   Ordering the Display of Tuples

   Set Operations

3. Explain the need for cursors.

   The cursor statement

   The open statement

   The close statement

The fetch statement

4. Discuss the strength and weaknesses of the trigger mechanism. Contrast triggers with other integrity constraints supported by SQL.

The Syntax for creating a trigger is:

Need for Triggers

Triggers

Example of Triggers in SQL

Types of PL/SQL Triggers

5. How the nested queries are obtained? Explain it?

Nested Queries

Correlated subquery

Set-Comparison Operators

6. Explain nested queries, aggregate operators, group by having clauses with suitable examples.

Aggregate Functions

The GROUP BY and HAVING Clause

Nested Queries

7. Explain in detail about embedded SQL

Embedded SQL

The cursor statement

ODBC and JDBC

8. Describe in detail about integrity constraints over relations

4 kinds of IC's

1. Key Constraints

2. Attribute Constraints

3. Referential Integrity Constraints

4. Global Constraints

9. Discuss in detail about Relational Calculus

Tuple Relational Calculus

Syntax Of Trc Queries

Semantics Of Trc Queries

Examples Of Trc Queries

The Domain Relational Calculus

Examples Of Drc Queries

10. Explain the various features of Structured Query Language

The Data Manipulation Language , The Data Definition Language

Triggers and Advanced Integrity Constraints, Embedded and Dynamic SQL

Transaction Management

Security

Advanced Features

UNIT – III

1. Explain the memory hierarchy of a computer system?

Classification of Physical Storage Media

Cache

Main memory

Flash memory

Magnetic-disk

Optical storage

Tape storage

2. Discuss the following

a. Page format b. Record Formats

Page Formats

Fixed-length records

Variable-length records

Record Formats

Fixed-length records

Variable-length records

3. Explain in detail about dynamic hashing

Dynamic Hashing

General Extendable Hash Structure

Example

4. Write a note on choice of selecting RAID levels.

RAID

Data striping

Redundancy

Parity Scheme

Error Correcting Scheme

RAID 0 , RAID 1, RAID - 0 + 1,RAID 2 ,RAID 3,RAID 4,RAID 5,RAID 6

Choice of RAID

Strengths of RAID

5. Explain in detail about the indexed sequential access method (ISAM)?

Indexed Sequential Access Method

Example ISAM Tree

6. What is B+ Trees? Explain the format of node & do the following operations

a. Search b. Insert c. Delete

B+ TREES

B+-Tree Node Structure

Queries on B+-Trees

Examples of a B+-tree for Search, Insert and Delete

7. Define in detail about static hashing? Explain in detail about linear hashing, extendible hashing?

Static Hashing

Example Of Hash File Organization

Hash Functions

Handling Of Bucket Overflows

Hash Indices

General Extendable Hash Structure , Example

Linear hashing

Insertion and Overflow condition , Example

8. Explain the concept of buffer management using buffer pool.

The role of buffer manager:

The buffer replacement policies: Least recently used (LRU) , Clock replacement

9. Explain in detail about various indices.

Overview of Indexes

Clustered vs. Unclustered indexes

Dense vs. Sparse indexes

Primary vs. secondary indexes

10. Explain in detail about Index on Composite Search Keys.

Index on Composite Search Keys

Equality query

Range query

UNIT – IV

1. Describe in detail about External Sorting

2-Way Sort: Requires 3 Buffers

Streaming Data Through Ram

Two-Way External Merge Sort

General External Merge Sort

To Sort A File With N Pages Using B Buffer Pages

Double Buffering

2. List all types of selection operations using index and explain.

Basic Algorithms

Selections Using Indices

A3 (primary index, equality on key)

A4 (primary index, equality on nonkey)

A5 (secondary index, equality)

3. List all types of Selections Involving Comparisons and explain.

A6 (primary index, comparison)

A7 (secondary index, comparison)

A9 (conjunctive selection using composite index (multiple attributes)

A10 (conjunction selection by intersection of identifiers)

A11 (disjunctive selection by union of identifiers)

Explain Join Operations

Nested loop join,

Sort-merge join and

Hash join

4. Discuss about Evaluation Of Expressions

Materialization

Pipelining

6. Describe **Query Optimization in detail**

Using Heuristic in Query Optimization

Transformation of Relational Expression

Equivalence Rules for transforming relational expressions

7**. Give one example for transforming relational expressions and explain**

Example of Transformation

8. Explain Normalization with example

Normalization Forms

First Normal Form (1NF)

Second Normal Form (2NF)

Third Normal Form (3NF)

Boyce-Codd normal Form (BCNF)

Determinant

Example

9. Explain Physical Database Design in Relational Databases

Conceptual Schema OR logical

Physical Schema

Factors that Influence Physical Database Design

10. How can we give security to the databases? Explain.

Security

Security Levels
Authorization
Authorization and Views
Granting of Privileges

UNIT –V

1. How will you test conflict serializability of a schedule?

   Conflict Equivalent

   Inconsistent Schedule

   Serializability

   Conflict Serializability

2. Explain in detail about two – Phase locking protocol.

   The Two-Phase Locking Protocol

   Phase 1: Growing Phase

   Phase 2: Shrinking Phase

3. How will you perform graph- based protocols? Explain.

   Graph-Based Protocols

   Tree Protocol

   Drawbacks

4. Explain serializability

   Conflict serializability

   View serializability

5. Explain lock based protocols

   Shared

   Exclusive

6. Explain two phase locking in detail.
    - Strict two phase locking
    - Rigorous two phase locking
7. Explain log based recovery in detail.
    - Immediate database modifications
    - Deferred modification
8. Explain ACID in detail
    - Atomicity
    - Consistency
    - Isolation
    - Durability
9. Describe Data Warehousing in detail
    - Data Warehousing
    - Creating and Maintaining a Warehouse
    - Multidimensional Data Model
    - Views And Decision Support
10. Explain how to design a database with an example
    - Requirements Analysis
    - Conceptual Design
    - Logical Database Design
    - Physical Database Design