6 (a) What do you understand by data analytics? What are the common steps in doing data analytics? [4]

(b) How can you differentiate the following schemas used in data warehouse? Give examples. [4]
i) Star ii) Snowflake iii) Fact-constellation

(c) The **cube** operation computes union of **group by's** on every subset of the specified attributes. [2]
Now consider the following cube operation on a multidimensional schema *sales (item_name, color, size, number)*:

select *item_name, color, size,* **sum**(*number*)
from *sales*
group by cube(*item_name, color, size*)

Convert the above SQL using only **rollup** operation that will produce the same output.

(d) Define the terms with respect the data mining: [4]
i) Classification ii) Regression

7 (a) What motivates you using object-based databases rather than relational database? [3]

(b) Consider a schema for instructor. Each instructor has: [2+4]

- *ID*
- *name* with sub-field *first_name* and *last_name* (composite attribute)
- a list of *children* (multivalued attribute)
- a list of *degree* achieved (multivalued attribute)
- a set of *phone_nos* (multivalued attribute)
- *age* (derived attribute of date attribute)

i) Create 2 tuples for the nested relation based on above schema.
ii) Show the 4NF decomposition of the nested relation mentioned in 7.b.i)

(c) Suppose that you have been hired as a consultant to choose a database system for your client's application. For each of the following applications, state what type of database system (relational, persistent programming language–based OODB, object relational; do not specify a commercial product) you would recommend. Justify your recommendation. [5]
i) A computer-aided design system for a manufacturer of airplanes.
ii) A system to track contributions made to candidates for public office.
iii) An information system to support the making of movies.

1 (a) Define RAID. Explain the necessity of RAID with examples. [4]

(b) What do you understand about P+Q Redundancy? Write notes on the RAID Level 5 with suitable examples. [5]

(c) Explain different performance measures of disks. [5]

2 (a) Define indexing. Discuss the evaluation factors of an index structure. [5]

(b) Make a B+ tree inserting the following values according to the given sequence. Consider the fanout value as 4. [Show only the tree after inserting each value] [5]
$23 \rightarrow 11 \rightarrow 67 \rightarrow 34 \rightarrow 97 \rightarrow 13 \rightarrow 42 \rightarrow 53 \rightarrow 7 \rightarrow 1 \rightarrow 100 \rightarrow 99 \rightarrow 90 \rightarrow 3 \rightarrow 77$

(c) "To make a primary index, one must choose the primary key as the search key" – explain with proper logic. [4]

3 (a) Explain the distinction between closed and open hashing. Discuss the relative merits of each technique in database applications. [3]

(b) Suppose that we are using extendable hashing on a file that contains records with the following search-key values: 2, 3, 5, 7, 11, 17, 19, 23, 29, 31, 35, 44, 47, 49, 52, 59, 63. Show the extendable (dynamic) hash structure for this file if the hash function is $h(x) = x \bmod 8$ and buckets can hold three records. [6]

(c) What are the causes of bucket overflow in a hash file organization? What can be done to reduce the occurrence of bucket overflows? [5]

4 (a) What do you understand by 'query processing' and 'query optimization'? Why is it not desirable to force users to make an explicit choice of a query processing strategy? [4]

(b) Assume (for simplicity in this exercise) that only one tuple fits in a block and memory holds at most three blocks. Show the runs created on each pass of the sort-merge algorithm when applied to sort the following tuples on the first attribute: (kangaroo, 17), (wallaby, 21), (emu, 1), (wombat, 13), (platypus, 3), (lion, 8), (warthog, 4), (zebra, 11), (meerkat, 6), (hyena, 9), (hornbill, 2), (baboon, 12) [6]

(c) Explain how you can apply the following equivalence rules to improve the efficiency of certain queries: [4]
i) $E_1 \bowtie_\theta (E_2 - E_3) \equiv (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$
ii) $\sigma_\theta({}_A\gamma_F(E)) \equiv {}_A\gamma_F(\sigma_\theta(E))$, where θ uses only attributes from A

5 (a) Compare different partitioning techniques for I/O parallelism with respect to: [5]
i) sequential search ii) point query iii) range query

(b) Histograms are good for avoiding data distribution skew but are not very useful for avoiding execution skew. Explain why. How virtual node portioning technique is used to handle skew created by I/O partitioning? [2+3]

(c) Explain the following forms of parallelism mentioning advantages and disadvantages: [4]
i) Pipelined Parallelism
ii) Independent Parallelism

1

(a) **RAID (Redundant Array of Independent Disks):** RAID is a data storage virtualization technology that combines multiple physical disk drive components into a single logical unit for data redundancy, performance improvement, or both. There are various RAID levels, each offering a different balance between performance, data redundancy, and storage capacity.

Necessity of RAID:

1. **Data Redundancy:** RAID provides data redundancy, protecting against data loss in case of a disk failure. For example, in RAID 1 (mirroring), data is copied identically on two drives. If one drive fails, the data can still be accessed from the other drive.
2. **Improved Performance:** Some RAID levels (e.g., RAID 0) improve read and write performance by striping data across multiple disks, allowing simultaneous access to data.
3. **Increased Storage Capacity:** By combining multiple disks, RAID can create a larger single storage unit, beneficial for systems that require large storage spaces.
4. **Fault Tolerance:** RAID configurations like RAID 5 or RAID 6 provide fault tolerance, allowing the system to continue functioning even if one or more drives fail.

Examples:

- **RAID 1 (Mirroring):** Ideal for critical systems where data loss cannot be tolerated, such as financial records or business databases.
- **RAID 0 (Striping):** Suitable for non-critical systems where high performance is required, like video editing workstations.
- **RAID 5 (Striping with Parity):** Balances performance, storage efficiency, and redundancy, often used in enterprise storage solutions.

(b) **P+Q Redundancy:** P+Q redundancy is a data protection method used in RAID 6. It involves storing two sets of parity information (P and Q) on each disk in the array. This allows the system to tolerate up to two simultaneous disk failures without data loss.

RAID Level 5 (Striping with Parity): RAID 5 distributes data and parity (error correction information) across all the disks in the array. Parity data is used to recover data in case of a disk failure.

Example of RAID 5: Suppose you have a RAID 5 array with four disks: A, B, C, and D. Data is written in chunks (stripes) across the disks, and parity information is also distributed.

- Stripe 1: Data block A1, B1, C1, and Parity P1
- Stripe 2: Data block A2, B2, Parity P2, and D2
- Stripe 3: Data block A3, Parity P3, C3, and D3
- Stripe 4: Parity P4, B4, C4, and D4

If disk B fails, the data on disk B can be reconstructed using the parity information and the data on the remaining disks. For example, P1 can be used to reconstruct B1.

(c) **Performance Measures of Disks:**

1. **Data Transfer Rate:** The speed at which data is transferred to and from the disk. This includes both sequential and random transfer rates. Measured in megabytes per second (MB/s).
2. **Seek Time:** The time it takes for the disk's read/write head to move to the position of the data. Measured in milliseconds (ms). Lower seek times result in faster access to data.
3. **Latency:** The delay before data transfer begins following a request. It is the time taken for the disk platter to rotate to the correct position under the read/write head. Measured in milliseconds (ms).
4. **Input/Output Operations Per Second (IOPS):** The number of read/write operations a disk can perform per second. Higher IOPS indicate better performance for random access workloads.

5. **Throughput:** The amount of data processed in a given amount of time. It considers both the transfer rate and the IOPS. Measured in MB/s or gigabytes per second (GB/s).

These measures are critical in evaluating disk performance, especially in environments requiring high-speed data access, such as databases, large-scale web applications, and high-performance computing.

2

(a) **Indexing:** Indexing in databases is a technique used to improve the speed of data retrieval operations. An index is a data structure that provides a quick way to look up records in a database table without scanning the entire table. It works similarly to an index in a book, where you can quickly locate the page number of a topic instead of searching through every page.

Evaluation Factors of an Index Structure: When evaluating the effectiveness and efficiency of an index structure, several factors are considered:

1. **Access Time:**
    - **Lookup Speed:** The time it takes to locate and retrieve data using the index. An efficient index significantly reduces lookup times compared to a full table scan.
    - **Insertion Speed:** The time it takes to insert new data into the table and update the index. Some index structures might slow down insert operations due to the need to maintain the index.
    - **Deletion Speed:** The time it takes to remove data and update the index accordingly. Like insertion, deletion operations can also be affected by the need to keep the index updated.
2. **Storage Overhead:**
    - **Space Complexity:** The amount of additional storage space required by the index. An index structure should provide a good balance between storage overhead and retrieval performance.

    - **Memory Usage:** The amount of memory used by the index when loaded into memory for operations. Efficient indexes minimize memory consumption.
3. **Maintenance Cost:**
    - **Update Frequency:** How often the index needs to be updated when the underlying data changes. High update frequency can increase maintenance costs.
    - **Rebuilding Cost:** The cost associated with rebuilding the index, which might be necessary after extensive data modifications to ensure optimal performance.
4. **Complexity:**
    - **Implementation Complexity:** The complexity of implementing the index structure within the database system. Simple structures are easier to implement and maintain, while more complex structures might offer better performance but at a higher implementation cost.
    - **Operational Complexity:** The complexity of performing operations like lookups, insertions, deletions, and updates using the index.
5. **Scalability:**
    - **Handling Large Datasets:** The index's ability to efficiently handle large volumes of data. Some index structures perform well with small datasets but degrade in performance as the dataset grows.
    - **Concurrency Support:** The ability of the index to support concurrent access by multiple users without significant performance degradation. This is crucial in multi-user database environments.
6. **Adaptability:**
    - **Query Patterns:** How well the index supports various query patterns. An index that works well for range queries might not be as efficient for exact match queries, and vice versa.
    - **Data Distribution:** The ability of the index to handle different data distributions, such as uniform distribution, skewed distribution, or sparse data.

Common index structures include **B-trees**, **B+ trees**, **hash indexes**, **bitmap indexes**, and **R-trees**, each with its own strengths and weaknesses depending on the specific use case and the evaluation factors mentioned above.

Primary Index and Primary Key:

A primary index is an index on the primary key of a table. The primary key uniquely identifies each record in the table.

Why Choose the Primary Key as the Search Key for a Primary Index:

1. **Uniqueness:**
    - The primary key is unique, ensuring each entry in the index points to a unique record.
2. **Efficient Data Retrieval:**
    - Searches using the primary key are fast because the index is sorted based on the primary key values.
3. **Ordered Data Storage:**
    - Data is stored in the same order as the primary key, making range queries efficient.
4. **Improved Query Performance:**
    - Many queries involve the primary key, so indexing it speeds up these queries.

Example:

Consider a table `Students` with:

- `StudentID` (Primary Key)
- `Name`
- `DateOfBirth`
- `Major`

If `StudentID` is the primary key, creating a primary index on `StudentID` allows quick searches and efficient range queries, such as finding a student by their ID or retrieving students within a certain ID range.

3

(a) Closed vs. Open Hashing

Closed Hashing (Open Addressing):

Closed hashing, also known as open addressing, stores all elements directly within the hash table. If a collision occurs (i.e., two keys hash to the same index), the system finds another empty slot within the table using a predefined probing sequence (e.g., linear probing, quadratic probing, double hashing).

Open Hashing (Separate Chaining):

Open hashing, or separate chaining, uses linked lists to handle collisions. Each slot in the hash table points to a linked list (or chain) of all elements that hash to the same index. If a collision occurs, the element is added to the linked list at the corresponding index.

Merits in Database Applications:

- **Closed Hashing:**
  - **Pros:**
    - Typically has better space utilization since no extra pointers are needed.
    - Can achieve good cache performance because all data is stored in contiguous memory locations.
  - **Cons:**
    - Performance can degrade with high load factors due to increased probing sequences.
    - Handling deletions can be complex, requiring rehashing or marking deleted slots.
- **Open Hashing:**

- **Pros:**
  - Easier to implement and manage collisions, especially with high load factors.
  - Deletion is straightforward since it involves removing an element from a linked list.
- **Cons:**
  - Requires extra memory for pointers, leading to potential space overhead.
  - Can have poor cache performance due to non-contiguous storage of elements.

(c) Causes and Mitigation of Bucket Overflow in Hash File Organization

Causes of Bucket Overflow:

1. **High Load Factor:** When too many records are stored in relation to the number of buckets, collisions increase, leading to overflow.
2. **Poor Hash Function:** A hash function that doesn't distribute keys uniformly across buckets can result in uneven bucket loads.
3. **Static Hashing:** Fixed number of buckets can become inadequate as the dataset grows.

Mitigation Techniques:

1. **Rehashing:** Dynamically resize the hash table and rehash all elements using a new hash function.
2. **Dynamic Hashing:** Use methods like extendable hashing or linear hashing that allow the hash table to grow and shrink dynamically as needed.
3. **Improving Hash Function:** Use a more sophisticated hash function that provides a more uniform distribution of keys.
4. **Separate Chaining:** Utilize open hashing to store overflow elements in linked lists, mitigating the impact of bucket overflow.

4

a)**Query Processing and Query Optimization**

Query Processing:

Query processing refers to the series of steps that a database management system (DBMS) takes to execute a query and retrieve the desired data. The main stages of query processing include:

1. **Parsing:**
    - The query is parsed to check its syntax and semantics.
    - The DBMS generates a parse tree or an abstract syntax tree (AST) representing the query structure.
2. **Translation:**
    - The parsed query is translated into a logical query plan, which represents the operations needed to obtain the results in a DBMS-independent way.
3. **Optimization:**
    - The logical query plan is optimized to create an efficient physical query plan. This step involves choosing the most efficient algorithms and access paths for executing the query.
4. **Execution:**
    - The DBMS executes the physical query plan and retrieves the requested data.

Query Optimization:

Query optimization is a critical step in query processing that focuses on improving the efficiency of a query. It involves choosing the best execution strategy from multiple possible strategies. The goal is to minimize the resources required to execute the query, such as CPU time, memory, and disk I/O.

Why Not Force Users to Choose Query Processing Strategies

1. **Complexity:**

    - Database systems are complex, and the internal details of query execution strategies can be intricate. Most users do not have the expertise to understand and choose the best strategy for their queries.
2. **Efficiency:**
    - The DBMS is designed to optimize queries automatically using advanced algorithms and heuristics. Forcing users to choose strategies manually could lead to suboptimal choices, reducing query performance.
3. **User Convenience:**
    - Users generally focus on what data they need rather than how the data is retrieved. Requiring users to specify query strategies would increase the complexity of writing queries, making the system less user-friendly.
4. **Adaptability:**
    - The optimal query execution strategy can depend on many factors, such as data distribution, indexing, and current system load. The DBMS can dynamically choose the best strategy based on the current state, which users may not be able to do effectively.
5. **Maintenance:**
    - Allowing the DBMS to handle query optimization means that as the database grows or changes, the system can adapt its strategies automatically. If users had to specify strategies, they would need to continuously update their queries to maintain performance.

5

(a) Partitioning Techniques for I/O Parallelism

Partitioning Techniques:

1. **Hash Partitioning:** Data is distributed based on the hash value of a key.

2. **Range Partitioning:** Data is distributed based on ranges of the key values.
3. **Round-Robin Partitioning:** Data is distributed evenly across partitions without considering the data values.

i) Sequential Search:

- **Hash Partitioning:**
    - **Pros:** Uniform distribution across partitions.
    - **Cons:** Requires checking all partitions, leading to higher overhead.
- **Range Partitioning:**
    - **Pros:** Only the relevant partition needs to be scanned if the range is known.
    - **Cons:** Imbalanced partitions if data distribution is skewed.
- **Round-Robin Partitioning:**
    - **Pros:** Simple and ensures balanced I/O across partitions.
    - **Cons:** Requires scanning all partitions, leading to inefficiency.

ii) Point Query:

- **Hash Partitioning:**
    - **Pros:** Directly accesses the relevant partition using the hash key.
    - **Cons:** Needs a good hash function to avoid collisions.
- **Range Partitioning:**
    - **Pros:** Efficient if the point query falls within a known range.
    - **Cons:** Can be inefficient if the range partitions are not well defined.
- **Round-Robin Partitioning:**
    - **Pros:** Simple implementation.
    - **Cons:** Inefficient as it may require scanning multiple partitions.

iii) Range Query:

- **Hash Partitioning:**
    - **Pros:** Not ideal for range queries as it doesn't group similar ranges.

    - **Cons:** Requires scanning multiple partitions to cover the range.
- **Range Partitioning:**
    - **Pros:** Efficiently handles range queries by limiting the search to relevant partitions.
    - **Cons:** Can suffer from imbalances if the ranges are not uniformly distributed.
- **Round-Robin Partitioning:**
    - **Pros:** Ensures balanced load distribution.
    - **Cons:** Inefficient for range queries as it requires scanning all partitions.

(b) Histograms and Execution Skew

Histograms:

- **Pros:** Provide a good representation of data distribution, useful for query optimization.
- **Cons:** Not effective in addressing execution skew because they do not control how work is distributed during execution.

Virtual Node Partitioning:

- **Description:** This technique involves creating many more partitions (virtual nodes) than the number of physical nodes, and then mapping these virtual nodes to physical nodes.
- **Handling Skew:**
    - **Pros:** If a physical node becomes a bottleneck, its virtual nodes can be redistributed to other less loaded physical nodes, balancing the workload dynamically.
    - **Cons:** Adds complexity in managing the mapping of virtual to physical nodes.

(c) Forms of Parallelism

i) Pipelined Parallelism:

- **Description:** Different stages of a query pipeline are executed in parallel. For example, while one stage processes a batch of data, the next stage can begin processing the previous batch's output.
- **Advantages:**
    - Reduces overall query execution time by overlapping operations.
    - Improves resource utilization.
- **Disadvantages:**
    - Requires efficient inter-stage communication.
    - Can be complex to implement and debug.

ii) Independent Parallelism:

- **Description:** Different operations of a query are executed independently and in parallel. For example, multiple independent queries or subqueries can be processed simultaneously.
- **Advantages:**
    - Simplifies implementation since tasks are independent.
    - Can lead to significant performance improvements if tasks are truly independent.
- **Disadvantages:**
    - Limited to queries that can be easily decomposed into independent tasks.
    - Potential resource contention if independent tasks compete for the same resources.

6

(a) Data Analytics

Definition: Data analytics is the process of examining datasets to draw conclusions about the information they contain, typically with the aid of specialized systems and software. It involves various techniques to analyze raw data and identify patterns, trends, and relationships that can inform decision-making.

Common Steps in Data Analytics:

1. **Data Collection:**
    - Gathering data from various sources, including databases, sensors, logs, and external datasets.
    - Example: Collecting sales data from an e-commerce platform, social media data, or customer feedback.
2. **Data Cleaning:**
    - Removing or correcting inaccuracies, inconsistencies, and errors in the data.
    - Example: Handling missing values, correcting typographical errors, and resolving data duplicates.
3. **Data Transformation:**
    - Converting data into a suitable format for analysis.
    - Example: Normalizing data, aggregating data at different levels, and creating derived metrics.
4. **Data Analysis:**
    - Applying statistical and computational techniques to examine data.
    - Example: Performing exploratory data analysis (EDA), running regression analysis, or using machine learning algorithms to find patterns.
5. **Data Visualization:**
    - Representing data and analysis results using visual elements like charts, graphs, and maps.
    - Example: Creating bar charts to show sales trends or heat maps to display geographic data.
6. **Interpretation and Reporting:**
    - Drawing insights and making recommendations based on the analysis.
    - Example: Generating reports that summarize findings, creating dashboards for ongoing monitoring, and presenting results to stakeholders.

(b) Data Warehouse Schemas

i) Star Schema:

- **Structure:**
  - Consists of a central fact table surrounded by dimension tables.
  - Fact table contains quantitative data (metrics or facts) and foreign keys to dimension tables.
  - Dimension tables contain descriptive data (attributes) related to the facts.
- **Example:**
  - **Fact Table:** `Sales` (contains columns like `SaleID`, `DateID`, `ProductID`, `StoreID`, `Amount`)
  - **Dimension Tables:** `Date` (contains `DateID`, `Date`, `Month`, `Year`), `Product` (contains `ProductID`, `ProductName`, `Category`), `Store` (contains `StoreID`, `StoreName`, `Location`)
- **Advantages:**
  - Simple and easy to understand.
  - Efficient for query performance due to fewer joins.
- **Disadvantages:**
  - Can lead to data redundancy in dimension tables.

ii) Snowflake Schema:

- **Structure:**
  - An extension of the star schema where dimension tables are normalized into multiple related tables.
  - Dimension tables are split into additional tables to reduce redundancy.
- **Example:**
  - **Fact Table:** `Sales`
  - **Dimension Tables:** `Date`, `Product`, `Store`
  - **Normalized Dimension Tables:** `ProductCategory` (contains `CategoryID`, `CategoryName`), `StoreLocation` (contains `LocationID`, `LocationName`)

- **Advantages:**
  - Reduces data redundancy and storage requirements.
  - Can improve data integrity.
- **Disadvantages:**
  - More complex queries due to additional joins.
  - Slightly slower query performance compared to star schema.

iii) Fact-Constellation Schema:

- **Structure:**
  - Also known as a galaxy schema.
  - Contains multiple fact tables that share dimension tables.
  - Used to represent multiple business processes.
- **Example:**
  - **Fact Tables:** `Sales`, `Returns`
  - **Dimension Tables:** `Date`, `Product`, `Store`, `Customer`
- **Advantages:**
  - Flexible and supports complex queries.
  - Can represent multiple business processes in a single schema.
- **Disadvantages:**
  - Increased complexity in schema design.
  - More challenging to maintain and manage.

These schemas help organize data in a data warehouse to support efficient querying and reporting, each with its own set of trade-offs.

(a) Motivation for Using Object-Based Databases over Relational Databases

1. **Complex Data Structures:**
   - Object-based databases (OODBMS) are designed to handle complex data structures like nested objects, inheritance, and polymorphism. This is particularly useful when working with data that naturally fits into an object-oriented paradigm, such as multimedia, CAD/CAM, and complex simulations.

2. **Seamless Integration with Object-Oriented Programming:**
    - OODBMS allows for seamless integration with object-oriented programming languages like Java, C++, and Python. This integration enables developers to work with data directly as objects without the need for complex object-relational mapping (ORM) layers.
3. **Enhanced Performance for Certain Operations:**
    - In scenarios where there is a need for handling large amounts of complex interrelated data with frequent read/write operations, OODBMS can offer better performance due to its ability to store objects directly without the need to flatten them into tables and rows.

(b) Schema for Instructor

Given the schema, let's create two tuples and show the 4NF decomposition.

Schema:

- `ID`
- `name` (composite attribute: `first_name`, `last_name`)
- `children` (multivalued attribute)
- `degrees` (multivalued attribute)
- `phone_nos` (multivalued attribute)
- `age` (derived attribute from `date`)

i) Two Tuples for the Nested Relation

1. Tuple 1:
    - `ID: 101`
    - `name: { first_name: "John", last_name: "Doe" }`
    - `children: ["Alice", "Bob"]`
    - `degrees: ["BSc", "MSc"]`
    - `phone_nos: ["123-4567", "234-5678"]`

    - `age: derived from date`
2. Tuple 2:
    - `ID: 102`
    - `name: { first_name: "Jane", last_name: "Smith" }`
    - `children: ["Charlie"]`
    - `degrees: ["BA", "PhD"]`
    - `phone_nos: ["345-6789"]`
    - `age: derived from date`

ii) 4NF Decomposition of the Nested Relation

To decompose the nested relation into 4NF, we must ensure that multivalued dependencies are removed.

Original Nested Relation:

```
Instructor(ID, name(first_name, last_name), children, degrees, phone_nos, age)
```

Decomposition into 4NF Relations:

1. **Instructor:**
    - `(ID, first_name, last_name, date)`
2. **Instructor_Children:**
    - `(ID, child)`
3. **Instructor_Degrees:**
    - `(ID, degree)`
4. **Instructor_Phone_Numbers:**
    - `(ID, phone_no)`

Example Data:

1. **Instructor:**

    - Tuple 1: `(101, "John", "Doe", date)`
    - Tuple 2: `(102, "Jane", "Smith", date)`
2. **Instructor_Children:**
    - Tuple 1: `(101, "Alice")`
    - Tuple 2: `(101, "Bob")`
    - Tuple 3: `(102, "Charlie")`
3. **Instructor_Degrees:**
    - Tuple 1: `(101, "BSc")`
    - Tuple 2: `(101, "MSc")`
    - Tuple 3: `(102, "BA")`
    - Tuple 4: `(102, "PhD")`
4. **Instructor_Phone_Numbers:**
    - Tuple 1: `(101, "123-4567")`
    - Tuple 2: `(101, "234-5678")`
    - Tuple 3: `(102, "345-6789")`

(c) Choosing a Database System

i) A Computer-Aided Design System for a Manufacturer of Airplanes:

- **Recommended Database System:** Persistent Programming Language-Based OODB
- **Justification:**
    - CAD systems involve complex objects with many interrelated parts, hierarchies, and inheritance.
    - OODBMS can efficiently store and manage these complex data structures.
    - Seamless integration with object-oriented programming languages facilitates direct manipulation of CAD objects.

ii) A System to Track Contributions Made to Candidates for Public Office:

- **Recommended Database System:** Relational Database
- **Justification:**

- The system involves managing large volumes of structured, tabular data with predefined relationships (contributors, candidates, contributions).
- Relational databases provide robust support for transactions, integrity constraints, and complex queries.
- The schema is relatively simple and fits well into the relational model.

iii) An Information System to Support the Making of Movies:

- **Recommended Database System:** Object-Relational Database (ORDB)
- **Justification:**
  - Movie production involves both structured data (schedules, budgets, personnel) and unstructured data (scripts, videos, images).
  - ORDB allows for the flexibility to handle both types of data within the same system.
  - Provides the benefits of a relational model (SQL support, ACID properties) while also supporting complex data types and objects.