ALLAMA IQBAL OPEN UNIVERSITY

USMAN NOTES
www.usmannotes.com

Name:

Registration No.:

Course Code: 3579

Program: Postgraduate

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Database Application (3579) • Level: Postgraduate

Semester: Autumn 2026

Assignment No. 2

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE — For Study & Reference

Note: This guide is prepared for study and reference purposes only. Students are advised to read the relevant parts of the course book, understand the concepts, try the SQL examples and design exercises themselves in a practice database, and then write the assignment in their own words. Copying this guide word for word is not recommended. No units are printed in the question booklet for this course, so none are shown on this title page.

USMAN NOTES www.usmannotes.com

Question 1: Normalisation and the Normal Forms (20 Marks)

Question as printed: a) Why is normalization an essential process in database design, and how does it contribute to efficient data organization and consistency? b) How do different normal forms progressively improve database structure, and in what ways do they help eliminate insertion, deletion, and update anomalies during the transformation of unstructured data into a well-organized schema?

Introduction

A database can store data and still be badly designed. The classic symptom is a single wide table in which everything about several different things has been mixed together — an order table that repeats the customer's name and address on every order line, or a student table that repeats the tutor's name and phone number for every student the tutor supervises. Such a table wastes storage, contradicts itself and makes ordinary operations dangerous. Normalisation, the process introduced by E. F. Codd as part of the relational model, is the disciplined remedy: a step-by-step method, based on functional dependencies, for decomposing tables so that each table describes one subject and each fact is stored once. This answer explains why normalisation is essential and how it produces efficient organisation and consistency, and then works through the normal forms in sequence, showing how each removes particular insertion, deletion and update anomalies.

Part (a): Why Normalisation Is Essential

The Problem It Solves: Anomalies

The defects of an unnormalised design are conventionally grouped as three anomalies, and they are worth stating precisely because the normal forms are best understood as their cure. An update anomaly occurs when a fact stored in many rows must be changed in all of them: if a tutor changes her phone number and one row is missed, the database now contains two numbers for the same tutor and nobody can tell which is right. An insertion anomaly occurs when a fact cannot be recorded at all because another, unrelated fact is missing: a newly appointed tutor with no students yet cannot be entered in a student table whose key requires a

student. A deletion anomaly occurs when deleting one fact unintentionally destroys another: if the only student of a course leaves and his row is deleted, the course's own details, stored only in that row, are lost with him. All three arise from the same root cause — several subjects, and therefore several kinds of fact, forced into one table.

Contribution to Organisation and Consistency

Normalisation contributes to efficient organisation in a direct way: decomposing a mixed table into tables each devoted to a single subject – Student, Course, Tutor, Enrolment — means each kind of fact is stored in exactly one place, so storage is not wasted on repetition and the structure of the database mirrors the structure of the real world it models, which makes it easier to understand, query and extend. Its contribution to consistency is equally direct: because a fact such as a tutor's phone number now appears once, an update changes it once and the new value is automatically the value everywhere; because subjects are separated, inserting a tutor no longer requires a student, and deleting a student no longer endangers a course. Normalisation also clarifies design decisions. Working out the functional dependencies — which attributes determine which others — forces the designer to discover and record the organisation's real rules, such as whether a course has one tutor or several, before those rules are built into tables. The process therefore improves both the product and the designer's understanding.

Part (b): The Normal Forms, Step by Step

Normal forms are cumulative: a table must satisfy first normal form before second is tested, and so on. In ordinary practical design, third normal form, or Boyce–Codd normal form, is the

usual target; higher forms (fourth and fifth) deal with rarer dependency types and are applied when those situations are present.

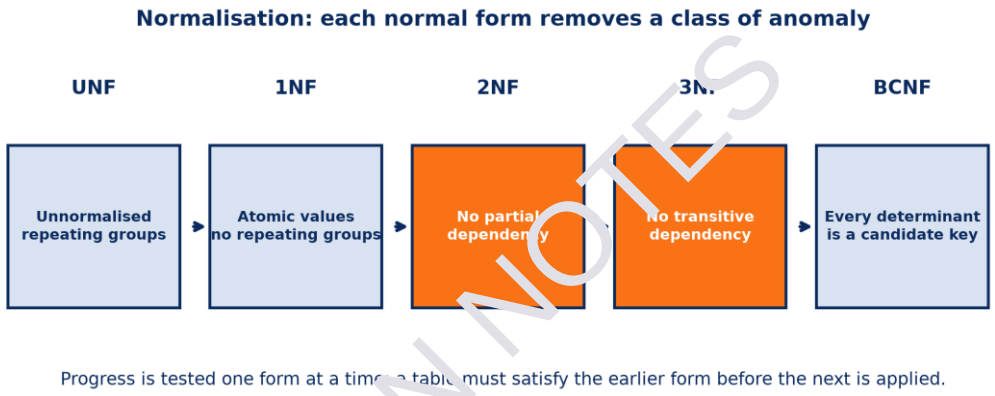


Figure 1: The normal forms in sequence. Each form retains the guarantees of the forms before it and removes one further class of anomaly.

Unnormalised Form and First Normal Form (1NF)

An unnormalised collection may contain repeating groups: one row per student with a list of courses and marks inside a single cell. Such a structure cannot be queried properly — 'find all students taking Physics' becomes text searching — and the same cell tries to hold several facts at once. First normal form requires that every attribute value be atomic, that there be no repeating groups, and that every row be identifiable by a primary key. The repeating group is moved into its own rows, so a student taking three courses occupies three enrolment rows, each with one course and one mark. This single step already makes the data queryable and removes the crudest insertion problem, in which adding a course meant altering the format of a list inside a cell.

Second Normal Form (2NF): No Partial Dependency

Second normal form applies where a primary key is composite, made of more than one column. It requires that every non-key attribute depend on the whole key, not on part of it. Suppose an Enrolment table is keyed by (StudentID, CourseID) and also contains StudentName: the name depends only on StudentID, part of the key, so it is repeated on every enrolment row of that student — a standing invitation to update and inconsistency anomalies. The remedy is decomposition: StudentName moves to a Student table keyed by StudentID alone, CourseName likewise moves to a Course table, and the Enrolment table keeps only facts that genuinely depend on both, such as the mark. After this step, a student's name is stored once, so correcting it is a single update, and a student or a course can be recorded before any enrolment exists, removing the corresponding insertion anomaly.

Third Normal Form (3NF): No Transitive Dependency

Third normal form requires that non-key attributes depend only on the key — in the familiar summary, on 'the key, the whole key, and nothing but the key'. The target is the transitive dependency, in which an attribute depends on the key through another non-key attribute. If a Student table contains DepartmentID and also DepartmentName and DepartmentPhone, the department's details depend on DepartmentID, not directly on the student, and they are repeated for every student in the department: the update anomaly returns at one remove, a new department cannot be inserted until it has a student, and deleting a department's last student deletes the department. Decomposing into Student and Department tables cures all three. Third normal form is the standard of a well-designed operational database: most

unintended redundancy and the great majority of practical anomalies are removed by the time it is reached.

Boyce–Codd Normal Form (BCNF) and Beyond

Boyce–Codd normal form strengthens 3NF by requiring that every determinant — every attribute or set of attributes on which some other attribute depends — be a candidate key. It matters in the less common situation where a table has several overlapping candidate keys; a table can satisfy 3NF yet still contain a dependency on a non-key determinant and retain a residual anomaly, which BCNF decomposes away, occasionally at the cost of not preserving every dependency in a single table. Fourth normal form deals with multivalued dependencies, where two independent lists of facts about the same subject have been combined into one table and multiply each other, and fifth normal form deals with certain join dependencies; both are specialist refinements applied when the situation they describe actually occurs. The practical lesson is one of judgement: normalise deliberately, usually to 3NF or BCNF, understand the dependencies involved, and stop when the design is clean rather than pursuing form for its own sake.

Table 1: A worked transformation in outline

Stage	Structure	Anomaly removed / remaining
Unnormalised	One row per student; list of courses and marks in one cell	Data not properly queryable; format changes to insert
1NF	Separate enrolment rows; atomic values; primary key	Repeating groups removed; data queryable

2NF	Student and Course details removed from Enrolment	Partial dependency removed; name stored once
3NF	Department details removed from Student	Transitive dependency removed; department independent
BCNF	Every determinant made a candidate key	Residual anomalies from overlapping keys removed

Balance: Normalisation and Deliberate Denormalisation

A complete answer should note the recognised trade-off. Fully normalised data are spread across many tables, so some reports must join several tables to assemble a complete picture, and in very high-volume reporting or analytical systems a designer may deliberately denormalise — reintroducing a specific, documented duplication — to gain speed. This does not refute normalisation; it depends on it. Only a designer who knows where each fact belongs can decide which duplication is worth its cost, keep it under control, and maintain it with defined procedures. The unnormalised table arrived at by ignorance has the costs of duplication without any of the control.

Conclusion

Normalisation is essential because it attacks the root cause of the insertion, deletion and update anomalies — the mixing of several subjects in one table — rather than treating their symptoms. It produces efficient organisation by storing each fact once in a table devoted to one subject, and consistency by making every update a single update. First normal form removes

repeating groups, second removes partial dependencies on part of a composite key, third removes transitive dependencies, and Boyce–Codd and the higher forms remove the residual cases. Applied with judgement, usually to 3NF or BCNF, normalisation transforms unstructured, anomaly-ridden data into a well-organised schema that can be trusted and maintained.

Question 2: SQL — The Bridge Between Users and Databases (20 Marks)

Question as printed: a) How does SQL serve as a bridge between users and database systems, and why is it fundamental for managing and querying data? b) In what ways do SQL operations support both the definition and manipulation of data, and how do advanced querying techniques like JOINS, aggregation, and subqueries enhance data analysis and retrieval?

Introduction

Almost nobody who uses a database touches the stored data directly. Between the user and the files on disk stands the DBMS, and the language in which the two communicate is SQL — Structured Query Language. SQL grew out of the relational model, was standardised internationally, and is implemented, with small variations, by virtually every relational DBMS. It is declarative: the user states what result is required, and the DBMS works out how to obtain it. This answer explains in what sense SQL is a bridge between users and database systems and why it is fundamental, and then examines its two great families of operations — defining data and manipulating data — together with the advanced querying techniques of joins, aggregation

and subqueries that turn retrieval into analysis. Students should type and test the short examples in a practice database; the examples here use small Student, Course and Enrolment tables for illustration.

Part (a): SQL as a Bridge, and Why It Is Fundamental

What the Bridge Connects

On one side of the bridge stand people and application programs with questions expressed in terms of the real world: which students have not paid their fees, what is the average mark in a course, how many accounts were opened this month. On the other side stand tables, indexes, storage structures and access methods that no ordinary user should need to know about. SQL connects them by offering a single, comparatively small language, close to the logical structure of the data, in which a question can be expressed as a statement about tables and columns. The DBMS receives the statement, checks the user's authority, plans an efficient execution using its optimiser and indexes, performs the work and returns the result as a table. The same bridge serves every class of user: the database administrator defining structures, the developer embedding statements in a program, the office user working through a form or report tool that generates SQL behind the scenes, and the analyst querying directly.

Why SQL Is Fundamental

- It is the standard interface to relational data: because SQL is standardised, skills and applications transfer between products, and an organisation is not locked into one vendor's private language.

- It is declarative and set-oriented: one statement operates on a whole set of rows at once, replacing the record-by-record loops of file processing, and the optimiser — not the user — chooses the access path, so queries keep working efficiently as data grow and storage is reorganised.
- It covers the whole life of the data in one language: defining tables, inserting and changing data, querying, controlling transactions and granting access are all SQL, so there is one coherent tool rather than a separate utility for each task.
- It enforces the database's protections: statements are executed under the user's authorisations and the tables' constraints, so the bridge is also a checkpoint — a user cannot use SQL to read data he or she has no right to see or to store data that violate the rules.

For these reasons SQL is not merely one feature of a DBMS among others. Remove it, and users and programs lose their common, controlled means of reaching the data at all; the database becomes a store that can be filled by its own internal tools but not genuinely used. Every later development — graphical query builders, reporting dashboards, application frameworks and object-relational mappers — ultimately rests on generating or executing SQL against the relational database.

Part (b): Definition, Manipulation and Advanced Querying

Defining Data: DDL

The Data Definition Language is the part of SQL that creates and changes the structures in which data live. CREATE TABLE defines a table with its columns, data types and constraints —

primary keys, foreign keys, unique, check and not-null rules — so the definition of the data and the rules protecting it are established together, in the same statement. ALTER TABLE evolves the structure as requirements change, adding a column or a constraint, and DROP removes structures that are no longer needed. CREATE VIEW defines a stored, named query that presents a selected and possibly combined picture of the underlying tables, and CREATE INDEX builds an index to speed retrieval on chosen columns. DDL therefore supports the adaptability discussed elsewhere in this course: the schema is data about data, held in the catalogue, inspectable and changeable by statement rather than by rewriting files and programs.

```
CREATE TABLE Student (
  StudentID   INT PRIMARY KEY,
  FullName    VARCHAR(60) NOT NULL,
  DepartmentID INT,
  FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
);
```

Manipulating Data: DML, and Control

The Data Manipulation Language works on the contents of the structures. INSERT adds rows, UPDATE changes rows that satisfy a condition, DELETE removes such rows, and SELECT retrieves data — the operation from which all reporting and analysis begin. SELECT's basic form names the columns required, the tables to read, and a WHERE condition selecting rows, with ORDER BY for presentation. Alongside DML stand the control facilities: transaction statements (COMMIT and ROLLBACK) that make a group of changes permanent or undo it as a unit, and the Data Control Language (GRANT and REVOKE) by which the administrator gives and withdraws

permissions. A well-formed UPDATE or DELETE always deserves a moment's care, because its WHERE clause is the difference between changing one student's record and changing every student's record — a practical illustration of why operations are tested and transacted rather than issued carelessly.

```
SELECT FullName FROM Student
WHERE DepartmentID = 5
ORDER BY FullName;
```

Joins: Answering Questions That Span Tables

Because normalisation deliberately distributes facts among related tables, most real questions span several of them, and the join is the operation that reassembles them. An inner join combines rows from two tables where the matching condition holds — enrolments joined to students on StudentID and to courses on CourseID produce a result showing each student's name, course title and mark, although no single table contains all three. Outer joins retain rows that have no match, which is exactly what is needed to list all students including those not yet enrolled on any course, or all courses including those with no enrolments. Self-joins relate a table to itself, for cases such as an employee table in which each row names the employee's manager in the same table. Joins are what make a normalised database fully usable: data are stored once, in their proper tables, and combined at query time in whatever combination the question requires.

```
SELECT s.FullName, c.CourseTitle, e.Mark
FROM Enrolment e
```

```
JOIN Student s ON s.StudentID = e.StudentID
JOIN Course  c ON c.CourseID  = e.CourseID;
```

Aggregation: From Rows to Information

Aggregation converts detail into summary, which is usually what decision-makers actually need. The aggregate functions — COUNT, SUM, AVERAGE (AVG), MAX and MIN — collapse a set of rows into a value, and GROUP BY partitions rows into groups so that an aggregate is produced for each group: the number of students in each department, the average mark in each course, total fee receipts in each month. The HAVING clause then filters groups by their aggregate values, selecting, for example, only courses whose average mark exceeds a threshold — a condition that cannot be expressed in WHERE, which filters individual rows before grouping. Aggregation is the point at which retrieval becomes analysis, and it is performed by the DBMS over the whole data set rather than by exporting rows into a spreadsheet and summarising a possibly stale copy.

```
SELECT CourseID, COUNT(*) AS Students, AVG(Mark) AS AverageMark
FROM Enrolment
GROUP BY CourseID
HAVING AVG(Mark) >= 50;
```

Subqueries: Questions inside Questions

A subquery is a SELECT nested inside another statement, and it expresses the many natural questions that are really two questions in sequence: find the students whose marks are above

the overall average (first find the average, then compare), or find the courses that no student has taken (first find the courses that have been taken, then exclude them). Subqueries appear in WHERE with operators such as IN, EXISTS and comparisons against a single aggregate value, in the FROM clause as a derived table, and in SELECT lists as a computed value per row. Correlated subqueries, which refer to the row currently being processed by the outer query, handle per-group comparisons such as students whose mark is the highest in their own course. Used with judgement — and rewritten as joins where that is clearer or faster — subqueries greatly extend what can be asked in a single, precise statement.

Conclusion

SQL serves as a bridge because it gives every user and program one standard, declarative, set-oriented language in which real-world questions can be put to the DBMS, which then plans and executes them under the database's constraints and authorisations; that combination is why SQL is fundamental rather than incidental. Its definition operations create and evolve tables, views and indexes together with their protective constraints, while its manipulation operations insert, update, delete and retrieve data under transaction and access control. Joins reassemble normalised data across tables, aggregation with grouping turns rows into decision-ready summaries, and subqueries express layered questions — together converting a store of records into an instrument of analysis and retrieval.

Question 3: Organisation of Data Storage, File Organisation and Indexing (20 Marks)

Question as printed: a) Why is the organization of data storage important in database systems, and how does it affect data retrieval performance? b) How do different file organization methods influence access speed and efficiency, and why are indexing techniques such as tree-based structures critical for optimizing large-scale data operations?

Introduction

A database is, physically, data written on storage — today usually solid-state or magnetic disks, organised by the system into pages or blocks that are read and written as units. How those pages are organised is not a mere engineering detail hidden from consideration: it largely determines how much work a retrieval must do, and therefore how long a user waits. The logical elegance of tables and SQL rests on this physical foundation, and the DBMS's task is to organise storage and provide access paths so that logical questions are answered with as little physical work as possible. This answer explains why storage organisation matters and how it affects retrieval performance, compares the principal file organisation methods, and shows why indexing — above all tree-based indexing — is critical at large scale.

Part (a): Why Storage Organisation Matters

Retrieval Performance Is Decided Physically

The dominant cost in most database operations is accessing storage: reading and writing pages is far slower than processing data already in memory, so performance is measured largely in

the number of page accesses an operation requires. Storage organisation decides that number. If the records wanted are scattered, finding them means visiting many pages; if they are clustered together, or reachable through a short access path, a few page reads suffice. Organisation also decides how expensive the other basic operations are: inserting a record may be a quick placement at the end of an unordered file or a careful insertion that maintains an ordering; deleting may leave reusable space or wasted gaps; scanning a whole table for a report may read pages in an efficient sequence or jump about. As tables grow from thousands to millions of rows, these differences are multiplied by the size of the data, which is why a design that performs acceptably on a small test database can fail in production if storage and access have not been thought about.

Storage organisation is also where several course themes meet. The buffer manager keeps frequently used pages in memory, so a good organisation that concentrates related accesses benefits twice — fewer reads, and a higher chance that the needed page is already cached. Physical organisation interacts with concurrency, because the granularity at which data are stored affects the granularity at which they can be locked, and with recovery, because the log and the data pages together must allow a consistent state to be rebuilt. Finally, organisation must accommodate growth and change: records are inserted, updated to different lengths and deleted continuously, and the structure must absorb this without steadily degrading — a requirement that distinguishes a managed DBMS file from a simple program-owned file, whose organisation is whatever its original programmer arranged.

Part (b): File Organisation Methods and Indexing

The Principal File Organisation Methods

Table 1: File organisation methods compared

Method	How records are stored	Strengths	Weaknesses / best use
Heap (unordered)	New records placed wherever space is available, usually at the end	Very fast insertion; simple	Retrieval by key requires a full scan; best for small tables and bulk loading
Sequential / ordered	Records kept in order of a key field	Fast for range and ordered retrieval and batch processing on that key	Insertion and deletion costly to maintain order; overflow handling needed
Hashing	Record's page computed by a hash function on the key	Extremely fast exact-match lookup on the hashed key	No support for range or ordered retrieval; performance depends on good distribution
Clustered / multi-table clustering	Related records from one or more tables stored together	Queries joining or reading related records touch few pages	Helps the chosen access pattern; may slow others

Each method therefore suits a pattern of use, and none suits all patterns. A heap is honest storage with no access help; an ordered file repays the work of maintaining order whenever

retrieval follows that order, as in processing accounts in account-number order or producing reports sorted by key; hashing gives near-immediate answers to the single most common kind of question — 'give me the record with exactly this key' — and nothing else; and clustering recognises that many retrievals are not for isolated records but for a record together with its related records, such as a customer with her orders, and stores them so that one page visit obtains several of them. A real DBMS combines these possibilities through its choice of table organisation and, more flexibly, through indexes, which add access paths without disturbing the underlying storage.

Indexing: A Separate Access Path

An index is an auxiliary structure containing key values in a searchable order, each with pointers to the records carrying that value. It is the database counterpart of a book's index: small compared with the whole, quick to search, and pointing directly to the material required. Indexes can be created on one column or several, can enforce uniqueness, and can be added or dropped by the administrator as query patterns change — a flexibility no reorganisation of the base file can match. A clustered index determines the physical order of the table itself, so a table can have only one; non-clustered (secondary) indexes are separate structures, and a table may have several, at the price that each index must be maintained on every insertion, update and deletion and occupies storage of its own. Index selection is therefore a design judgement: index the columns by which data are actually searched, joined and ordered, and resist indexing everything.

Why Tree-Based Structures Are Critical at Scale

The standard index structure in relational systems is the B+ tree, and its virtues are precisely those that large scale demands. The tree is balanced — every leaf is the same distance from the root — and has a very high fan-out, because each internal node holds many keys and pointers; consequently the height of the tree grows extremely slowly with the number of records, and a search descends through only a few levels — typically three or four even for millions of rows — reading one page per level before reaching the leaf that points to the data. In the B+ tree all records' keys, or pointers to them, are held in the leaves, and the leaves are linked in key order, so the same structure that gives fast exact lookup also gives efficient range retrieval — all marks between 40 and 60, all accounts in a branch, all orders in a date range — by descending once and then scanning along the leaves. Updates are localised: inserting a key affects one leaf, splitting it only when full, so the structure maintains itself under continuous change without periodic wholesale reorganisation. Contrast the alternatives: a full scan reads every page of a table whose size grows with the data, and a simple sorted file pays heavily to maintain its order. The tree's logarithmic behaviour is what separates 'the system slows as we grow' from 'the system answers as quickly at ten million records as at ten thousand, for the questions it was indexed to answer'.

USMAN NOTES www.usmannotes.com

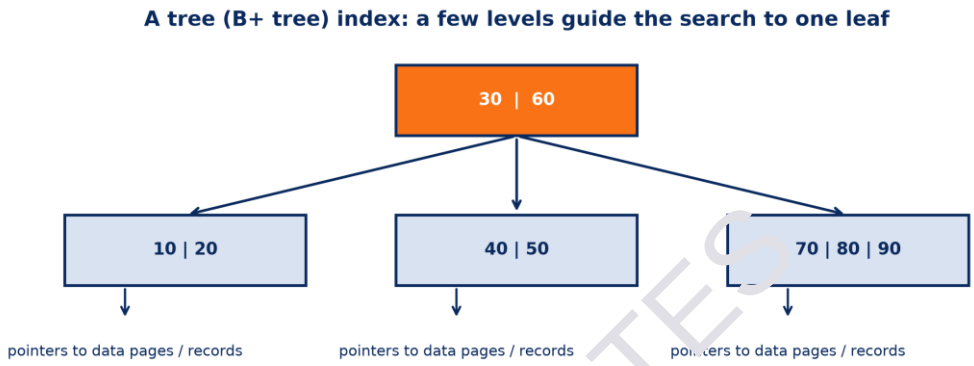


Figure 2: A B+ tree index in outline. Searching for a key such as 45 follows one path from the root to a leaf; the linked leaves then support range retrieval in key order.

Two qualifications complete a postgraduate treatment. First, an index accelerates retrieval on its key but does not help questions on other columns, and a query that needs most of a table's rows — a full statistical summary, for example — is often served better by a straightforward scan than by millions of index-guided visits; the query optimiser exists precisely to make this choice from statistics. Second, indexes interact with the rest of the system: they consume storage, they slow write-heavy work because they must be maintained, and their benefit depends on the buffer, since a tree whose upper levels are cached in memory may cost effectively a single disk read per search. Critical, therefore, does not mean unlimited: tree-based indexing is critical because it is the mechanism that makes large-scale retrieval feasible at all, and it is applied selectively, monitored, and adjusted as workloads change.

Conclusion

The organisation of data storage matters because retrieval performance is decided by how many pages an operation must touch, and organisation — together with growth, concurrency

and recovery — determines that number. Heap, ordered, hashed and clustered organisations each serve particular access patterns and fail others, which is why a DBMS supplements the base organisation with indexes: separate, maintainable access paths. Tree-based structures, above all the balanced, high-fan-out B+ tree with linked leaves, are critical for large-scale operations because they reduce exact searches to a handful of page accesses and support range retrieval in order, while maintaining themselves under continuous update. Without them, large databases would be stores that could be written but not practically read.

Question 4: Security and Recovery in Multi-User Database Systems (20 Marks)

Question as printed: a) Why are security and recovery fundamental concerns in database management systems, especially in multi-user environments? b) How do modern DBMS mechanisms ensure data protection, consistency, and system reliability in the presence of failures, unauthorized access, and concurrent transactions?

Introduction

A database is valuable precisely because many people depend on it at once — and that is also why it is vulnerable. Shared data can be read or altered by the wrong person, damaged by two users working on the same record at the same moment, or lost in the instant of a power failure halfway through a transaction. Security, which protects data against unauthorised access and change, and recovery, which restores a correct state after failure, are therefore not optional additions to a DBMS; they are among its defining functions. This answer explains why both

concerns are fundamental, with particular attention to the multi-user environment, and then describes the modern mechanisms — authentication and authorisation, views and encryption, transaction management with its ACID guarantees, concurrency control, logging and backup — by which a DBMS protects data and keeps it consistent and reliable. The discussion remains academic; assigning legal responsibility in any real incident is a matter for qualified professionals and the courts.

Part (a): Why Security and Recovery Are Fundamental

The Value and Sensitivity of Shared Data

An operational database concentrates an organisation's most sensitive information in one place: personal details, financial balances, examination results, medical or commercial records. Concentration creates value — integrated data are useful — and simultaneously creates risk, because a single breach or corruption can expose or destroy in one event what would previously have been scattered across many offices. The harm is correspondingly wide. Unauthorised disclosure can injure the individuals whose data are exposed, breach legal duties of confidentiality and data protection, and destroy the trust on which a bank, a university or a public service depends. Unauthorised or accidental alteration can be worse than disclosure, because it may go undetected while decisions, payments and certificates are generated from false data. Loss of the data altogether can halt the organisation's work entirely, since modern operations have no paper fallback for a system of this scale.

Why Multi-User Working Sharpens Both Concerns

In a single-user system most threats arrive one at a time and the user's own care is a large part of the defence. A multi-user system removes that simplicity. Dozens or thousands of users, with different rights and different levels of trustworthiness, are connected simultaneously, often over networks and from outside the building. Their work interacts: the classic concurrency problems — one user's uncommitted change being read by another, one update overwriting another, summary figures computed from a mixture of before-and-after states — can corrupt data without any malice at all, simply through interleaving. More users also mean more failures in progress at any moment: when the system crashes, it does not interrupt one tidy operation but a crowd of half-finished transactions that must be sorted into those to keep and those to undo. And the attack surface is larger in kind, not just in number — remote access, web interfaces, insider misuse and automated attacks all reach a networked multi-user database in ways they cannot reach a standalone file. Security and recovery must therefore be continuous, automatic system functions, not procedures users are asked to remember.

Part (b): The Mechanisms of Protection, Consistency and Reliability

Protection against Unauthorised Access

- Authentication and account management: the DBMS verifies the identity of each user — by password, increasingly with additional factors, and integrated with organisational sign-on — and can lock accounts, enforce password policies and expire idle sessions, so access begins from a known identity rather than an anonymous connection.

- Authorisation, privileges and roles: using SQL's GRANT and REVOKE, administrators give specified users or roles specified operations — select, insert, update, delete, execute — on specified tables, views and procedures, and nothing more. Roles model jobs (clerk, examiner, auditor, administrator), so a person changing jobs changes role rather than accumulating stray permissions.
- Views as a security boundary: a user can be authorised on a view that exposes only the rows and columns his or her work requires — a department's own students, names and results without contact or fee details — while the underlying tables remain inaccessible to that user.
- Encryption and auditing: data can be encrypted in storage and in transit so that stolen files or intercepted traffic do not disclose it, and audit facilities record who accessed or changed what and when, deterring misuse and making investigation possible. Application-level defences complement these: programs that pass user input into SQL as data rather than as code prevent injection attacks, in which a crafted input tries to make the database execute the attacker's statement.

Consistency under Concurrent Transactions

The unit of protection is the transaction: a logical unit of work, such as transferring an amount between two accounts or recording a payment together with its receipt, that must obey the ACID properties. Atomicity guarantees the transaction is all-or-nothing; consistency guarantees it takes the database from one state satisfying all constraints to another; isolation guarantees concurrent transactions do not interfere with one another's intermediate states; and durability

guarantees that once a transaction is reported as committed, its effects survive any subsequent failure. Isolation is implemented by concurrency control: two-phase locking, in which transactions acquire and release locks in a disciplined order and conflicting access to the same data is made to wait; timestamp and optimistic methods in suitable workloads; and in many modern systems multiversion concurrency control in which readers see a consistent committed version of the data without blocking writers. Deadlocks, where transactions wait for one another in a circle, are detected and resolved by aborting and restarting one transaction. The serialisable outcome the user experiences is simple to state: the result is the same as if the transactions had run one after another, even though in fact they overlapped to use the system efficiently.

Reliability in the Presence of Failures: Logging, Backup and Recovery

Recovery rests on the transaction log, a sequential record of changes in which — under the write-ahead logging rule — the log entry describing a change is written to durable storage before the change itself may be. The log records transaction beginnings, changes with their before and after values, and commit or abort decisions, with periodic checkpoints limiting how much of it must be re-examined after a crash. Recovery then classifies transactions: those committed before the failure are redone, so their effects are guaranteed present; those uncommitted or aborted are undone, so their partial effects disappear; and the database re-emerges in a consistent state, exactly as the ACID promises require. Beyond crash recovery stand the media and disaster provisions: scheduled full and incremental backups, archived logs that allow recovery to a chosen point in time, copies held on separate media or at another site or in the cloud, and mirroring or replication in systems that must continue through the failure

of a whole server. Two disciplines decide whether these mechanisms succeed in practice: backups and recovery procedures must be tested by actual restoration exercises, and the recovery objectives — how much data loss and how much downtime the organisation can accept — must be decided in advance, because they determine how frequently logs and backups are taken.

Table 1: Threats and the mechanisms that answer them

Threat	DBMS mechanism	What it guarantees
Unauthorised reading or change	Authentication, privileges, roles, views	Only authorised operations by identified users
Data exposure from stolen files / traffic	Encryption at rest and in transit	Stolen copies are not readable data
Concurrent users corrupting each other's work	Transactions, locking / multiversion control	Results equivalent to serial execution; no lost updates
Crash during a multi-step operation	Write-ahead log; redo and undo on restart	Committed work survives; partial work disappears
Disk failure or disaster	Backups, archived logs, off-site copies, replication	Restore to a consistent point; continuity of service
Undetected misuse	Audit trail	A record of who accessed or changed what

Conclusion

Security and recovery are fundamental because a database concentrates sensitive, organisation-critical data and exposes it to many simultaneous users, so that disclosure, corruption and loss are each capable of causing serious harm — and because concurrency and failure create dangers even when every user is honest. Modern DBMS mechanisms answer each threat in kind: authentication, authorisation, roles, views, encryption and auditing protect against unauthorised access; transactions with ACID properties and disciplined concurrency control keep concurrent work consistent; and write-ahead logging, checkpoints, backup and tested recovery restore a reliable, consistent state after failures. Together they allow an organisation to share its data widely — which is the point of having a database — without sharing it carelessly.

Question 5: From a Real-World Problem to a Complete Database Solution (20 Marks)

Question as printed: a) How can a real-world problem be transformed into a structured database design, and what key steps are involved in developing a complete database solution from concept to implementation? b) How does a well-designed database system ensure data integrity, minimize redundancy, and effectively support the functional requirements of users?

Introduction

Databases do not begin with tables; they begin with a problem in the real world — a college that cannot track its students and fees reliably, a clinic losing track of appointments, a shop

whose stock figures are never right. The craft of database development is the disciplined transformation of that messy situation into a precise structure of entities, relationships, constraints and operations, and then into a working system that people use. This answer sets out the key steps of that transformation, from requirements analysis through conceptual, logical and physical design to implementation, testing and maintenance, using a small illustrative example. It then explains how a well designed result ensures integrity, minimises redundancy and supports what users actually need to do. No specific institution's system is described; the example is a teaching illustration.

Part (a): Steps from Concept to Implementation

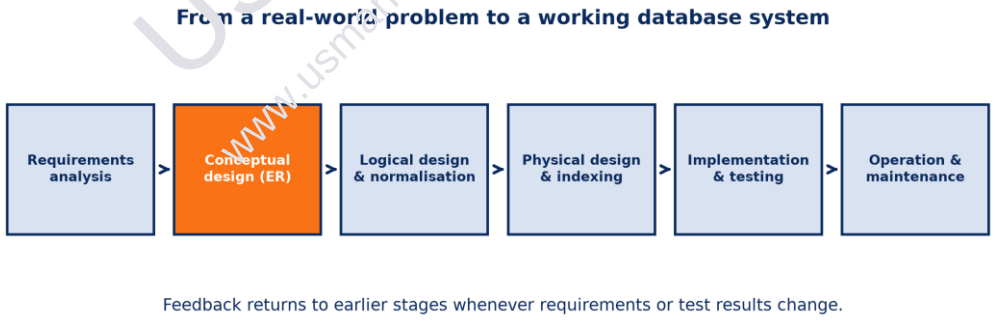


Figure 1: The main stages of database development. The process is iterative: findings at any stage may send the designer back to an earlier one.

Step 1: Requirements Analysis — Understanding the Problem

The designer first discovers what the organisation does, what data it handles, what is going wrong now, and what the new system must achieve. The techniques are interviews with managers and with the staff who do the work, observation, and study of the existing

documents — application forms, registers, receipts, reports — because the documents reveal the data actually used, as opposed to the data people believe is used. The output is a requirements specification covering the data to be kept, the transactions and questions the system must support (register a student, record a payment, list defaulters, produce a transcript), volumes and growth, the classes of user and their access needs, and the rules of the organisation that the data must obey. Errors made here are the most expensive in the whole project, because a misunderstood requirement faithfully implemented is still the wrong system, so this stage deserves time and checking with the users themselves.

Step 2: Conceptual Design — Modelling the World

The requirements are then modelled independently of any particular software, most commonly as an entity–relationship (ER) model. The designer identifies the entities about which data are kept (Student, Course, Tutor, Payment), their attributes (a student's name and date of birth; a course's title and credit hours), the primary keys that identify each instance, and the relationships between entities with their cardinalities — a student enrols on many courses, a course is taken by many students, a department has many students each belonging to one department. The ER diagram is the crucial communication device of the project: users who could not judge a table design can look at a diagram of the things they deal with daily and say whether it matches their world. Many-to-many relationships, such as student–course, are noted for resolution into an associative entity (Enrolment, carrying its own attributes such as the mark) at the next stage.

Step 3: Logical Design — Mapping to Tables and Normalising

The conceptual model is mapped to the relational model by standard rules: each entity becomes a table, attributes become columns, keys become primary keys, and relationships are represented by foreign keys or, for many-to-many cases, by a new table whose key combines the keys of the participants. The resulting schema is then refined by normalisation, as examined in Question 1 of this assignment: functional dependencies are identified, and tables are decomposed to (usually) third normal form or Boyce–Codd normal form so that each fact is stored once and the insertion, deletion and update anomalies are removed. Integrity rules discovered in the requirements are declared as constraints, and the views and queries the users require are checked against the design — if an important question cannot be answered from the schema, the design, not the question, is revised. The outcome is a complete logical schema, still independent of performance decisions and of a specific product's details.

Steps 4 to 6: Physical Design, Implementation and Operation

- Physical design: choose the DBMS and translate the logical schema into its terms — data types and sizes, the indexes that the expected queries and joins require (typically on keys and frequent search columns), any partitioning of very large tables, storage and security arrangements, and the initial decisions about controlled denormalisation, if any workload justifies it.
- Implementation: create the database with SQL DDL, load and convert existing data — which in a real project includes cleaning it, because conversion exposes the duplicates and

contradictions the old files contained — write or configure the applications, forms and reports, set up users, roles and views, and establish backup procedures from the first day.

- Testing and evaluation: test with realistic volumes, not only with a handful of sample rows; verify that transactions behave correctly, that constraints reject invalid data, that concurrent use and recovery after a simulated failure work as designed, and — decisively — that users can complete their real tasks and obtain their real reports.
- Operation, maintenance and evolution: monitor performance and growth, adjust indexes, apply changes as requirements evolve using the adaptability the DBMS provides, review access rights periodically, and feed lessons back into the design. A database is a living system accompanying the organisation, not a product finished on delivery day.

Part (b): How a Well-Designed System Delivers Integrity, Low Redundancy and User Support

Ensuring Data Integrity

Integrity is designed in at every stage and then enforced by the system. Requirements analysis discovers the rules — marks lie between 0 and 100, a payment must belong to an existing student, a student's programme must be one the institution offers. Logical design expresses them structurally through keys and relationships, and implementation declares them as constraints — primary key and entity integrity, foreign key and referential integrity, unique, check, not-null and domain constraints — which the DBMS then enforces on every transaction, from every application and user, without depending on each program remembering to check. Transactions add the multi-step dimension: a registration consisting of an enrolment, a fee

assessment and a timetable allocation either completes as a whole or is rolled back, so the database never rests in a half-registered state. Integrity in a well-designed system is thus not a hope or a periodic clean-up; it is a property maintained continuously by the structure itself.

Minimising Redundancy

Redundancy is minimised chiefly by the combination of sound entity modelling and normalisation. Because the conceptual model separates the real subjects of the data, and normalisation tests each table against its functional dependencies, each fact — a student's name, a course's title, a tutor's phone number — comes to rest in exactly one table, with all other uses reached by keys and joins at query time. The practical gains follow as they did throughout this course: updates are single and therefore consistent, storage is not spent on repetition, and insertion and deletion of one subject do not endanger facts about another. Where redundancy is deliberately retained for performance, in a designed system it is identified, documented and maintained as a decision — the opposite of the accidental duplication that accumulates in unplanned files.

Supporting the Functional Requirements of Users

A design is only well designed relative to the work it must support, which is why the development process begins and ends with users. Their transactions define the operations the schema and applications must make easy; their questions define the views, joins and reports; their roles define the access rights and the views through which they see the data; and their expected volumes define the indexes and physical choices. In the finished system this support appears as correct and current answers, forms and portals that match the tasks people actually

perform, summaries for managers drawn from the same data as the clerks' working records rather than from a separate reporting copy, and adaptability when requirements change — a new report as a new view, a new attribute as a new column, a new class of user as a new role — without rebuilding the system. Evaluation must therefore always return to the functional test: can every class of user do everything the specification promised, accurately and within an acceptable time? If yes, the design has achieved its purpose; if not, the iteration continues, regardless of how elegant the schema may look.

Table 1: How each design quality is produced

Quality	Produced mainly by	Enforced / experienced as
Integrity	Requirements rules; keys; declared constraints; transactions	Invalid data rejected at entry; multi-step work all-or-nothing
Minimal redundancy	Entity modelling; normalisation to 3NF / BCNF	Each fact stored once; related data joined at query time
Support for users	Requirements analysis; views, queries and roles designed for real tasks	Fast, correct transactions and reports for each class of user
Long-term fitness	Data independence; monitoring and maintenance	Schema evolves and performance is tuned without rebuilding

Conclusion

A real-world problem is transformed into a structured database design by a disciplined sequence: analyse requirements with the users and their documents; model the world conceptually as entities, attributes and relationships; map the model to tables and refine it by normalisation; make physical choices of indexes and storage; implement, convert and clean the data; test against real tasks and volumes; and operate, monitor and evolve the system thereafter. A solution built this way ensures integrity because organisational rules become system-enforced constraints and transactions, minimises redundancy because modelling and normalisation place each fact in exactly one table, and supports users' functional requirements because those requirements drove the design from the first interview to the final evaluation. Good databases, in short, are not found but designed — and designed from the problem outward, not from the software inward.

USMAN NOTES www.usmannotes.com