DATABASE ENGINEERING

Unit 4

Normalization, Functional Dependencies and Decomposition

Section A — 45 short-answer questions (2 marks each)

Section B — 15 long-answer questions (7 marks each), fully worked

Merges the Unit IV syllabus topics, the model question bank and every Unit IV question from PYQ R25, B26 and R24

Compiled from the official five-unit syllabus, the previous-year question papers (R25, B26, R24), the class slide decks, the model question banks and the handwritten notes. Every question is answered in full; differences are shown in two-column tables and ER diagrams are drawn wherever a question needs one.

Section A — 2-Mark Questions

Q1. Define normalization.

PYQ R25 Q5(b), B26 Q5(b)

Normalization is the step-by-step process of decomposing a relation into smaller, well-structured relations so that data redundancy and the insertion, update and deletion anomalies are removed. Each step is guided by the functional dependencies present in the relation, and each higher normal form removes a stronger kind of dependency.

Q2. Why is normalization needed?

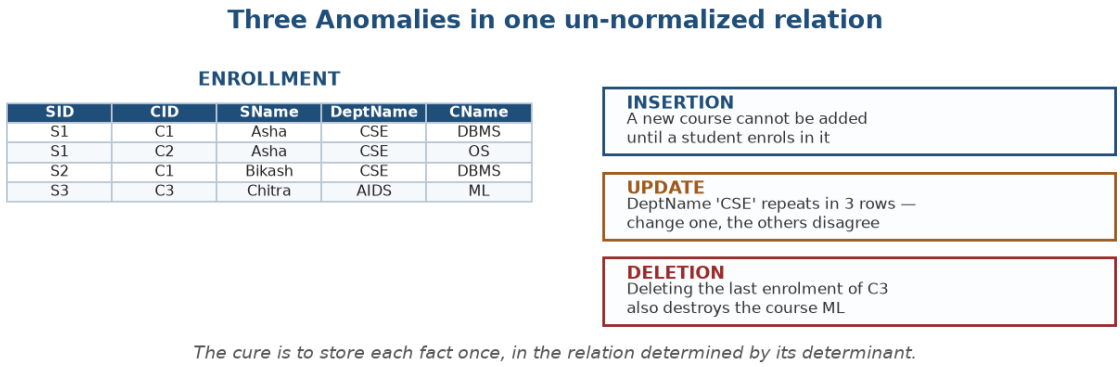
Because an un-normalized relation stores the same fact in many rows. That wastes storage, makes updates error-prone, prevents some facts from being recorded at all and can destroy unrelated facts when a row is deleted. Normalization stores each fact once, in the relation where its determinant belongs.

Q3. What are anomalies? Name the three types.

PYQ R24 Q1(j)

Anomalies are the problems that arise when an un-normalized relation is modified. The three types are the **insertion anomaly** (a fact cannot be recorded without another unrelated fact), the **update anomaly** (the same fact appears in many rows and must be changed in all of them) and the **deletion anomaly** (deleting one fact accidentally removes another).

Q4. Explain the insertion anomaly with an example.



In ENROLL(StudentID, CourseID, StudentName, CourseName, Grade), a new course cannot be recorded until at least one student enrolls in it, because the course attributes exist only as part of an enrollment row. The course is known to the university but cannot be stored.

Q5. Explain the update anomaly with an example.

If the name of department D1 changes, every row in which D1 appears must be updated. If even one row is missed, the relation holds two different names for the same department and becomes inconsistent.

Q6. Explain the deletion anomaly with an example.

If the last enrollment for a course is deleted, the only stored record of that course disappears with it. Deleting one fact — the enrollment — has destroyed a different fact — the course details.

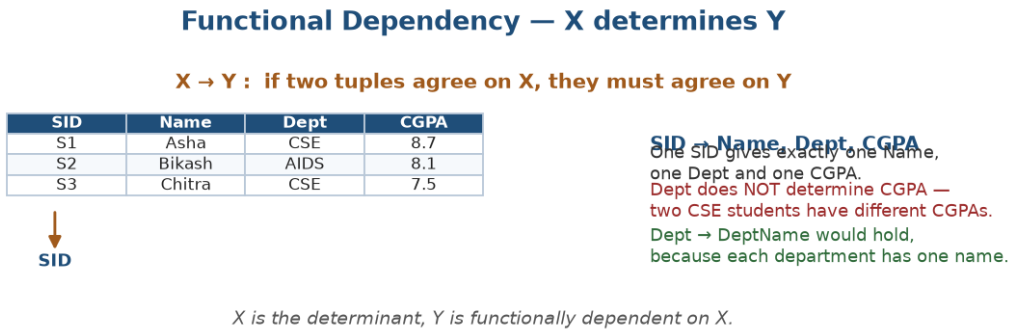
Q7. State the advantages of normalization.

Less redundancy, so less storage; better consistency because each fact is stored once; fewer modification anomalies; clearer and more explicit dependencies; easier enforcement of integrity constraints; and a logical design that is easier to extend and to understand.

Q8. State the limitations of normalization.

More tables are created, so queries need more joins; complex joins can hurt performance on read-heavy workloads; the design becomes more detailed and harder to explain; and some reporting queries become slower, which is why controlled **denormalization** is sometimes used deliberately.

Q9. Define functional dependency.



PYQ B26 Q1(k)

A **functional dependency** X → Y between two sets of attributes of a relation holds if, whenever two tuples agree on all attributes of X, they must also agree on all attributes of Y. X is the **determinant** and Y is functionally dependent on X. Example: StudentID → StudentName, Dept.

Q10. What is a trivial functional dependency?

An FD X → Y is **trivial** when Y is a subset of X, because the dependency is satisfied by the definition of a tuple. Example: {StudentID, Name} → Name, or AB → A. A trivial FD carries no information about the data.

Q11. What is a non-trivial and a completely non-trivial FD?

An FD X → Y is **non-trivial** when Y is not a subset of X, for example A → B. It is **completely non-trivial** when X and Y have no attribute in common at all, for example AB → CD. Normalization is concerned only with non-trivial dependencies.

Q12. Define full functional dependency.

Y is **fully functionally dependent** on X if X → Y holds and no proper subset of X determines Y. Example: in ENROLL(StudentID, CourseID, Grade), the FD {StudentID, CourseID} → Grade is full, because neither StudentID alone nor CourseID alone determines Grade.

Q13. Define partial dependency.

PYQ R25 Q1(k)

A **partial dependency** exists when a non-prime attribute depends on only a proper subset of a composite candidate key. Example: in ENROLL(StudentID, CourseID, StudentName, CourseName, Grade), StudentID → StudentName is partial with respect to the key {StudentID, CourseID}. Partial dependencies are what 2NF removes.

Q14. Define transitive dependency.

PYQ R25 Q1(k)

A **transitive dependency** exists when a non-prime attribute depends on another non-prime attribute rather than directly on the key. If X → Y and Y → Z, then X → Z holds transitively. Example: EmpID → DeptID and DeptID → DeptName, so EmpID → DeptName transitively. Transitive dependencies are what 3NF removes.

Q15. Differentiate partial dependency from transitive dependency.

PYQ R25 Q1(k)

Partial Dependency	Transitive Dependency
A non-prime attribute depends on part of a composite key	A non-prime attribute depends on another non-prime attribute
The determinant is a proper subset of the candidate key	The determinant is a non-prime attribute
Possible only when the key is composite	Possible with any key, including a single attribute
Removed by 2NF	Removed by 3NF
Example: StudentID → StudentName	Example: DeptID → DeptName when EmpID → DeptID

Q16. Define multivalued dependency.

A **multivalued dependency** X ↠ Y holds in R(X, Y, Z) when for each value of X the set of Y values is independent of the set of Z values, so every combination of Y and Z must appear. Example: STUDENT(StudentID, Hobby, Language) with StudentID ↠ Hobby and StudentID ↠ Language. Multivalued dependencies are what 4NF removes.

Q17. Define join dependency.

A **join dependency** states that a relation can be reconstructed exactly by joining specified projections of itself: $R = \pi_{R1}(R) \bowtie \pi_{R2}(R) \bowtie \ldots \bowtie \pi_{Rn}(R)$. It generalises the multivalued dependency and is what 5NF removes.

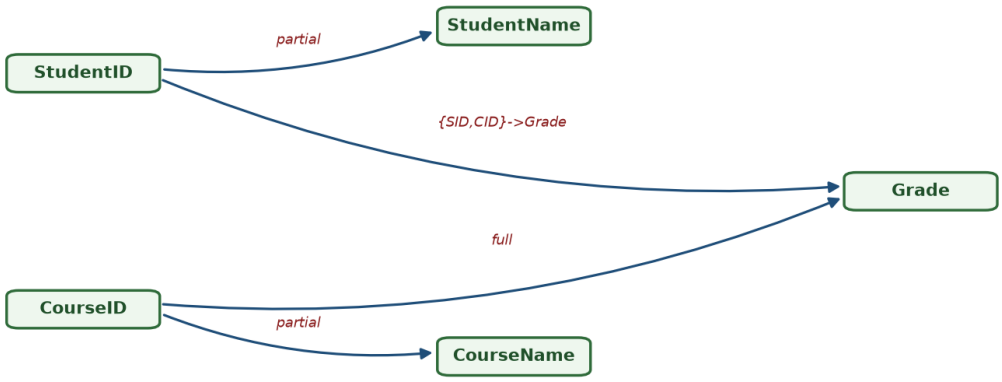
Q18. Define inclusion dependency.

An **inclusion dependency** states that the set of values of one attribute must be contained in the set of values of another attribute, written R.A ⊆ S.B. It expresses value containment between relations, and a foreign key is the most common practical example: ENROLLMENT.StudentID ⊆ STUDENT.StudentID.

Q19. What is a dependency diagram?

A **dependency diagram** is a picture of a relation in which each attribute is drawn as a node and each functional dependency as an arrow from the determinant to the dependent attribute. An arrow leaving only one part of a composite key marks a partial dependency, and a chain such as A → B → C marks a possible transitive dependency.

Dependency Diagram — ENROLL(StudentID, CourseID, StudentName, CourseName, Grade)



Arrows leaving only one part of the composite key mark partial dependencies (2NF violations).

Q20. State Armstrong's axioms.

PYQ R25 Q1(I)

Axiom	Statement	Example
Reflexivity	If Y ⊆ X then X → Y	{A,B} → A
Augmentation	If X → Y then XZ → YZ	A → B implies AC → BC
Transitivity	If X → Y and Y → Z then X → Z	A → B, B → C implies A → C

These three rules are **sound** (they generate only true dependencies) and **complete** (they can generate every dependency implied by F).

Q21. State the derived inference rules.

Rule	Statement
Union (additivity)	If X → Y and X → Z then X → YZ
Decomposition (projectivity)	If X → YZ then X → Y and X → Z
Pseudotransitivity	If X → Y and WY → Z then WX → Z
Composition	If X → Y and Z → W then XZ → YW

All of them can be proved from reflexivity, augmentation and transitivity.

Q22. What is the closure of a set of functional dependencies, F+?

PYQ B26 Q1(I)

F+ is the set of all functional dependencies that can be logically derived from the given set F using Armstrong's axioms. It contains F itself plus every implied dependency, including all trivial ones, so it is usually very large and is not listed explicitly — in practice only attribute closures are computed.

Q23. What is the closure of an attribute set, X+?

X+ is the set of all attributes functionally determined by X under F. It is computed by starting with X and repeatedly adding the right-hand side of every FD whose left-hand side is already contained in the set. If X+ contains every attribute of the relation, X is a super key.

Q24. Give the algorithm to compute X+.

1. Set X+ = X. 2. For each FD Y → Z in F, if Y ⊆ X+, add Z to X+. 3. Repeat step 2 until no new attribute can be added. 4. The resulting set is X+. To test whether X → Y is implied by F, compute X+ and check whether Y ⊆ X+.

Q25. What is a minimal cover?



PYQ R25 Q1(j)

A **minimal cover** (canonical cover) of F is an equivalent set of dependencies that has no unnecessary parts. Its three conditions are: every FD has exactly one attribute on the right-hand side; no attribute on any left-hand side is extraneous; and no whole FD is redundant.

Q26. What is a non-redundant cover and how does it differ from a minimal cover?

A **non-redundant cover** is a set of FDs from which no entire dependency can be removed without changing the closure. A minimal cover satisfies this **and** additionally has a single attribute on every right-hand side and no extraneous left-hand attribute. So every minimal cover is non-redundant, but a non-redundant cover need not be minimal.

Q27. How is the left-hand side of an FD minimized?

PYQ R24 Q1(k)

To test whether attribute A is extraneous in the left-hand side of XA → Y, remove A and compute the closure of the remaining X under F. If that closure already contains Y, then A was extraneous and the FD can be rewritten as X → Y. Example: with AB → C and A → B, removing B from AB → C leaves A, and A+ contains B and hence C, so B is extraneous and the FD becomes A → C.

Q28. How are redundant functional dependencies removed?

PYQ R24 Q1(l)

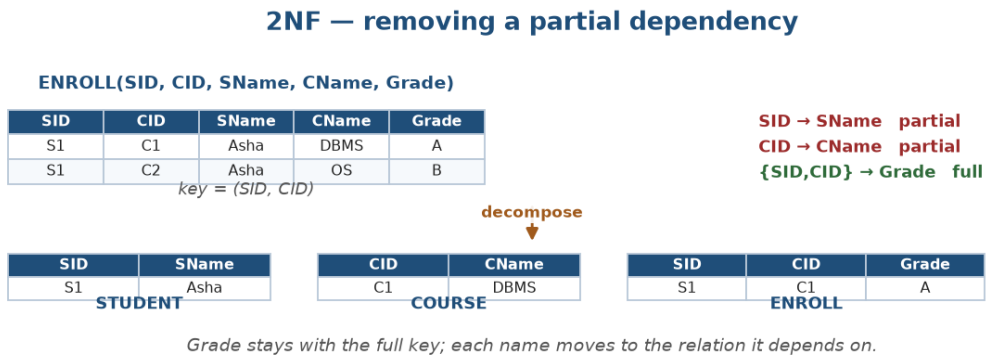
For each FD X → Y in turn, temporarily remove it, compute X+ using the remaining FDs, and check whether Y is still in X+. If it is, the FD is redundant and is dropped permanently. Example: in F = {A → B, B → C, A → C}, removing A → C leaves A+ = {A, B, C}, which contains C, so A → C is redundant and the cover reduces to {A → B, B → C}.

Q29. Define First Normal Form.

PYQ R25 Q5(b)

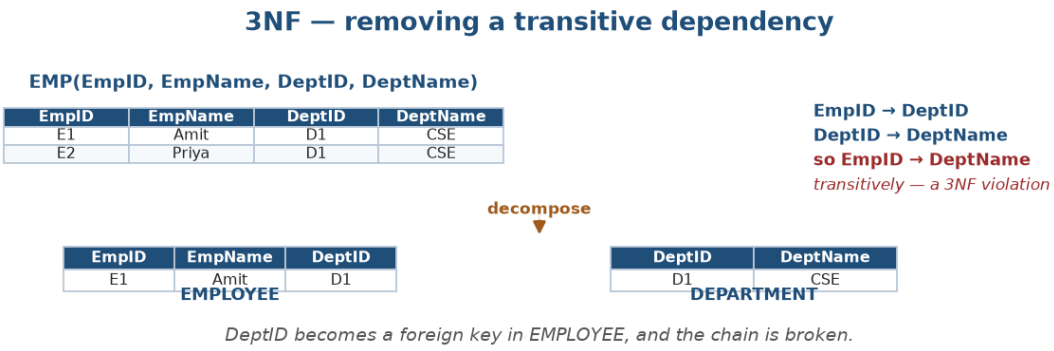
A relation is in **1NF** if every attribute holds only atomic, indivisible values and there are no repeating groups or multivalued cells. A column storing "9876, 9123" in one cell violates 1NF and must be split into two rows.

Q30. Define Second Normal Form.



A relation is in **2NF** if it is in 1NF and every non-prime attribute is fully functionally dependent on the whole candidate key — that is, there is no partial dependency. If every candidate key is a single attribute, partial dependency cannot arise and a 1NF relation is automatically in 2NF.

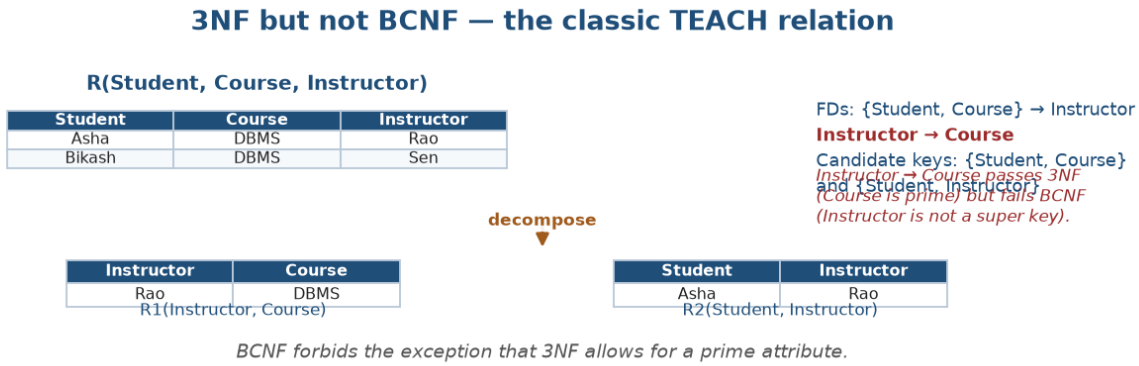
Q31. Define Third Normal Form.



PYQ R25 Q5(b)

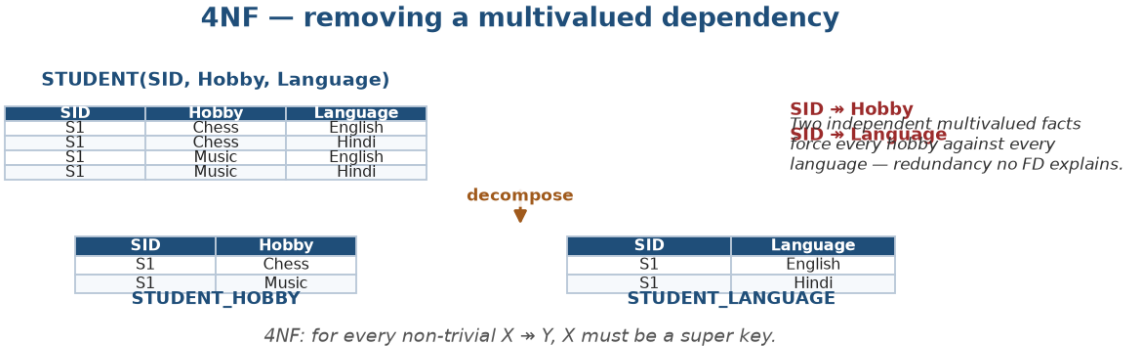
A relation is in **3NF** if it is in 2NF and there is no transitive dependency of a non-prime attribute on a candidate key. Formally, for every non-trivial FD X → A, either X is a super key or A is a prime attribute.

Q32. Define Boyce-Codd Normal Form.



A relation is in **BCNF** if for every non-trivial functional dependency X → Y, X is a super key. BCNF is stricter than 3NF because 3NF allows X → A when A is prime even though X is not a super key, while BCNF does not allow that exception.

Q33. Define Fourth Normal Form.



A relation is in **4NF** if it is in BCNF and, for every non-trivial multivalued dependency X ↠ Y, X is a super key. It removes the redundancy caused by two independent multivalued facts stored in the same relation.

Q34. Define Fifth Normal Form.

A relation is in **5NF** (Project-Join Normal Form) if it is in 4NF and every join dependency in it is implied by its candidate keys — that is, the relation cannot be decomposed further without losing information. It deals with cases where a relation is 4NF but still contains a cyclic join dependency.

Q35. What is lossless join decomposition?

A decomposition of R into R1, R2, ..., Rn is **lossless** if the natural join of the fragments reproduces exactly the original relation, with no extra tuples. For a binary decomposition the test is that the common attributes must functionally determine all attributes of at least one fragment: (R1 ∩ R2) → R1 or (R1 ∩ R2) → R2.

Q36. What is a lossy decomposition?

A decomposition is **lossy** if the natural join of the fragments produces tuples that were not in the original relation — called **spurious tuples**. Lossy decomposition destroys information, and it is never acceptable; the word "lossy" refers to loss of information, not loss of rows.

Q37. What is dependency preservation?

A decomposition is **dependency preserving** if every functional dependency of the original set can be enforced by checking the fragments individually, without joining them. Formally, the union of the projections of F onto each fragment must imply F.

Q38. What is denormalization?

Denormalization is the deliberate reintroduction of redundancy into a normalized design — by merging relations or by storing derived values — to reduce the number of joins and improve read performance. It trades update anomalies and consistency risk for speed, and is used in data warehouses and reporting systems.

Q39. Why can BCNF decomposition lose dependencies?

Because a BCNF decomposition must split on every dependency whose determinant is not a super key, and the determinant and the dependent attribute may end up in different fragments. 3NF decomposition, by contrast, can always be made both lossless and dependency preserving, which is why 3NF is often the practical target.

Q40. What is a prime attribute?

Model question — Unit 4

A **prime attribute** is an attribute that belongs to **at least one candidate key** of the relation. Every other attribute is non-prime. The distinction matters because 2NF and 3NF are defined in terms of non-prime attributes: a partial or transitive dependency matters only when the dependent attribute is non-prime.

Q41. What is an extraneous attribute?

Model question — Unit 4

An attribute is **extraneous** in a functional dependency if removing it does not change the closure of the set. In XA → Y, A is extraneous when X alone already determines Y, so the dependency can be written X → Y. Removing extraneous attributes is the second step of the minimal-cover algorithm.

Q42. Differentiate lossless decomposition from dependency preservation.

Model question — Unit 4

Lossless decomposition	Dependency preservation
Joining the fragments reproduces exactly R	Every FD can be checked inside one fragment
Concerns information content	Concerns constraint enforcement
Tested by (R1 ∩ R2) → R1 or R2	Tested by projecting F onto each fragment
A BCNF decomposition is always lossless	A BCNF decomposition may lose dependencies

The two properties are independent — a decomposition can have one without the other.

Q43. What is a spurious tuple?

Model question — Unit 4

A **spurious tuple** is a tuple that appears in the natural join of two fragments but was **not** present in the original relation. Spurious tuples are the visible symptom of a lossy decomposition: the join pairs values that never occurred together, so the result contains more rows than the original and the information has been destroyed.

Q44. Why is 4NF needed when BCNF already exists?

Model question — Unit 4

Because BCNF is defined only on **functional** dependencies. Two independent multivalued facts stored in one relation — hobbies and languages, for example — violate no functional dependency, so the relation can be in BCNF and still contain redundancy that no FD can explain. 4NF adds the condition that every multivalued dependency determinant must be a super key, which removes that redundancy.

Q45. Differentiate 1NF from 2NF.

Model question — Unit 4

1NF	2NF
Every value atomic, no repeating groups	1NF plus no partial dependency
Concerns the form of a value	Concerns the dependency of a non-prime attribute
Removes multivalued cells	Removes dependency on part of a composite key
Always achievable by splitting rows	Relevant only when the key is composite

Section B — 7-Mark Questions

Q1. Explain normalization and the normal forms up to BCNF with suitable examples.

PYQ R25 Q5(b), B26 Q5(b), R25 model questions

The running relation

ENROLL(StudentID, StudentName, CourseID, CourseName, InstructorID, InstructorName, Grade) Dependencies: StudentID → StudentName; CourseID → CourseName, InstructorID; InstructorID → InstructorName; {StudentID, CourseID} → Grade. The candidate key is {StudentID, CourseID}.

The anomalies present

The student name repeats for every course the student takes, so changing it needs many updates. A new course cannot be inserted without a student. Deleting the last enrollment of a course removes the course and its instructor.

First Normal Form

Every attribute must be atomic. If the relation stored Phone = "9876, 9123" in one cell it would violate 1NF; splitting into two rows fixes it. Our relation is already in 1NF, but 1NF alone removes neither partial nor transitive dependencies.

Second Normal Form — remove partial dependencies

StudentName depends only on StudentID, and CourseName and InstructorID depend only on CourseID; both are partial dependencies with respect to the composite key. Decompose:

STUDENT	COURSE	ENROLL
StudentID, StudentName	CourseID, CourseName, InstructorID	StudentID, CourseID, Grade
Key: StudentID	Key: CourseID	Key: (StudentID, CourseID)

Third Normal Form — remove transitive dependencies

In COURSE, CourseID → InstructorID and InstructorID → InstructorName, so InstructorName depends transitively on the key. Decompose further:

STUDENT	COURSE	INSTRUCTOR	ENROLL
StudentID, StudentName	CourseID, CourseName, InstructorID	InstructorID, InstructorName	StudentID, CourseID, Grade

All four relations are now in 3NF: every non-prime attribute depends directly and fully on its own key.

Boyce-Codd Normal Form — every determinant a super key

BCNF requires that for every non-trivial X → Y, X is a super key. A relation that is in 3NF but not BCNF is R(Student, Course, Instructor) with {Student, Course} → Instructor and Instructor → Course. The candidate keys are {Student, Course} and {Student, Instructor}. Instructor → Course violates BCNF because Instructor is not a super key, but it satisfies 3NF because Course is a prime attribute. The BCNF decomposition is R1(Instructor, Course) and R2(Student, Instructor).

The ladder

Normal form	Condition	Removes
1NF	Atomic values, no repeating groups	Multivalued cells
2NF	1NF plus no partial dependency	Partial dependency
3NF	2NF plus no transitive dependency of a non-prime attribute	Transitive dependency
BCNF	Every determinant is a super key	Remaining key anomalies
4NF	Every MVD determinant is a super key	Multivalued dependency
5NF	Every join dependency implied by keys	Join dependency

Q2. Explain the anomalies of an un-normalized relation, the need for normalization and its pros and cons.

Syllabus Unit IV

The un-normalized relation

SID	SName	DeptID	DeptName	CID	CName	Instructor	Grade
S1	Asha	D1	CSE	C1	DBMS	Rao	A
S1	Asha	D1	CSE	C2	OS	Sen	B
S2	Bikash	D1	CSE	C1	DBMS	Rao	B
S3	Chitra	D2	AIDS	C3	ML	Das	A

Candidate key {SID, CID}; FDs: SID → SName, DeptID; DeptID → DeptName; CID → CName, Instructor; {SID, CID} → Grade.

The three anomalies

Update anomaly — renaming department D1 requires updating every row in which D1 appears; a missed row leaves two names for one department. **Insertion anomaly** — a new course cannot be recorded until some student enrolls, because the course columns exist only inside an enrollment row. **Deletion anomaly** — deleting the last enrollment for C3 also deletes the only record of the course ML and its instructor Das.

Why normalization is needed

Each fact should be stored once, in the relation determined by its determinant. Normalization removes the redundancy that makes the anomalies possible, so updates touch one row, inserts never require unrelated facts, and deletes never destroy unrelated facts.

Advantages

Advantage	Explanation
Less redundancy	Each fact stored once
Better consistency	One copy means no disagreement
Fewer anomalies	Insert, update and delete are safe
Clear dependencies	The schema shows what determines what
Easier integrity	Constraints attach to the right relation
Easier extension	New facts need new relations, not wider rows

Limitations

Limitation	Explanation
More tables	A normalized design has many small relations
More joins	Queries must reconnect the fragments
Performance cost	Joins are expensive on large data
Complexity	Harder to explain and to maintain
Denormalization pressure	Reporting systems often undo part of it

The final design

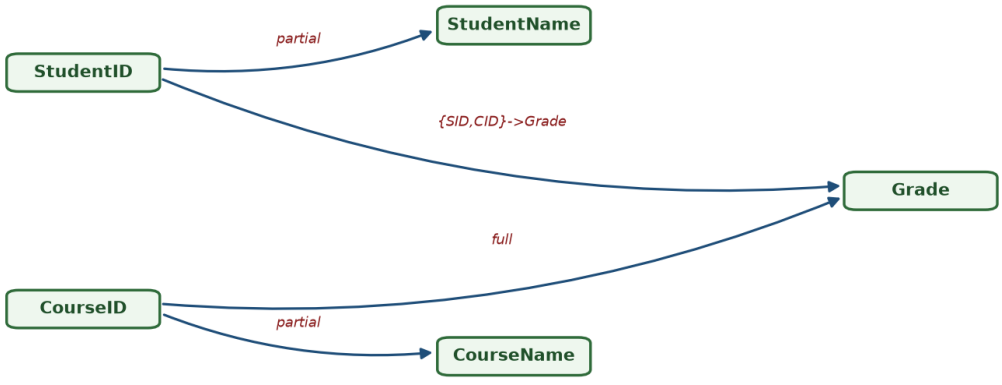
STUDENT(SID, SName, DeptID), DEPARTMENT(DeptID, DeptName), COURSE(CID, CName, Instructor), ENROLLMENT(SID, CID, Grade). Each relation is in 3NF and the anomalies are gone.

Q3. Explain all the types of functional dependency with a dependency diagram.

PYQ B26 Q1(k), R25 Q1(k), R24 Q1(k)

The diagram

Dependency Diagram — ENROLL(StudentID, CourseID, StudentName, CourseName, Grade)



Arrows leaving only one part of the composite key mark partial dependencies (2NF violations).

Trivial and non-trivial

X → Y is **trivial** if Y ⊆ X, such as AB → A, and always true. It is **non-trivial** if Y ⊄ X, such as A → B, and carries real information. **Completely non-trivial** means X and Y are disjoint, such as AB → CD.

Full dependency

Y depends on the whole of X and on no proper subset of X. In ENROLL, {SID, CID} → Grade is full: neither SID nor CID alone determines Grade.

Partial dependency

A non-prime attribute depends on part of a composite key. SID → SName and CID → CName in ENROLL are partial. Removing them is exactly the job of 2NF.

Transitive dependency

X → Y and Y → Z give X → Z. EmpID → DeptID and DeptID → DeptName give EmpID → DeptName. Removing this is the job of 3NF.

Multivalued dependency

X ↠ Y means that for one value of X there is a well-defined set of Y values, independent of the other attributes. STUDENT(SID, Hobby, Language) with SID ↠ Hobby and SID ↠ Language forces every hobby to be paired with every language, which is redundancy of a kind that no functional dependency can explain. This is the job of 4NF.

Join dependency

A join dependency *((R1), (R2), …, (Rn)) states that R is equal to the join of those projections. When such a dependency is not implied by the keys, the relation can be decomposed without loss, which is the job of 5NF.

Inclusion dependency

R.A ⊆ S.B requires every value of A to appear in B, as with a foreign key. It is a constraint between relations rather than within one, and it is enforced by referential integrity.

Summary table

Dependency	Notation	Removed by
Trivial	Y ⊆ X	Nothing — always valid
Full	X → Y, no proper subset works	Required for 2NF
Partial	Part of a composite key determines	2NF
Transitive	X → Y → Z	3NF
Multivalued	X ↠ Y	4NF
Join	*((R1),…,(Rn))	5NF
Inclusion	R.A ⊆ S.B	Referential integrity

Q4. Explain Armstrong's axioms and prove the derived rules.

PYQ R25 Q1(l), R24 Q1(k)

The three axioms

Axiom	Statement
IR1 Reflexivity	If Y ⊆ X then X → Y
IR2 Augmentation	If X → Y then XZ → YZ
IR3 Transitivity	If X → Y and Y → Z then X → Z

Proof of reflexivity

Take any two tuples t1 and t2 of R with t1[X] = t2[X]. Since Y is a subset of X, the tuples also agree on Y, so t1[Y] = t2[Y]. Therefore X → Y holds. This is why a set of attributes always determines any of its own subsets, and why trivial dependencies need no checking.

Proof of augmentation

Assume X → Y. Take tuples with t1[XZ] = t2[XZ]. Then t1[X] = t2[X] and t1[Z] = t2[Z]. From X → Y we get t1[Y] = t2[Y], and combining with t1[Z] = t2[Z] gives t1[YZ] = t2[YZ]. Hence XZ → YZ.

Proof of transitivity

Assume X → Y and Y → Z. If t1[X] = t2[X] then by X → Y, t1[Y] = t2[Y]; then by Y → Z, t1[Z] = t2[Z]. Hence X → Z.

Derived rule — union

Given X → Y and X → Z. By augmentation on X → Y with X, we get X → XY; by augmentation on X → Z with Y, we get XY → YZ. By transitivity on X → XY and XY → YZ, we get X → YZ.

Derived rule — decomposition

Given X → YZ. By reflexivity, YZ → Y. By transitivity, X → Y, and similarly X → Z. Note that decomposition applies to the right-hand side only.

Derived rule — pseudotransitivity

Given X → Y and WY → Z. Augment the first with W to get WX → WY. By transitivity on WX → WY and WY → Z, we get WX → Z.

Why the axioms matter

They are **sound** — every dependency they produce is true in every relation satisfying F — and **complete** — every dependency implied by F can be produced from F using them. Completeness is what justifies computing attribute closure as the test for implication, and it is the theoretical basis of minimal cover and of the synthesis algorithm for 3NF.

Q5. Explain closure of attributes and of functional dependencies, and use it to find keys.

PYQ B26 Q1(l), R25 Q1(j), R24 Q5(a)

F+ and X+

F+ is the set of all dependencies implied by F; it is enormous and never listed in full. X+, the attribute closure, is the set of all attributes determined by X, and it is what is actually computed.

Algorithm

Start with X+ = X; for every FD Y → Z in F, if Y ⊆ X+ then add Z to X+; repeat until nothing changes. To test X → Y, check Y ⊆ X+.

Worked example 1

R(A, B, C, D, E), F = {A → B, B → C, C → D, D → E}.

Step	X+	Reason
Start	{A}	Initial
1	{A,B}	A → B
2	{A,B,C}	B → C
3	{A,B,C,D}	C → D
4	{A,B,C,D,E}	D → E

A+ is the whole relation, so A is a super key and, being a single attribute, a candidate key.

Worked example 2 — the PYQ question

R(A, B, C, D, E), F = {A → B, BC → D, D → E}. Compute A+ and (BC)+. A+ = {A, B} — A determines B, and B alone cannot trigger BC → D because C is missing. (BC)+ = {B, C, D, E} — BC → D, then D → E. Further closures: D+ = {D, E}; (ABC)+ = {A, B, C, D, E}, so ABC is a super key. The attributes that never appear on a right-hand side are A and C, and (AC)+ = {A, C, B, D, E} = all attributes, so {A, C} is the candidate key.

Using closure to find candidate keys

Compute the closure of each single attribute, then of each pair, and stop as soon as the closure equals the whole relation. Any attribute that never appears on a right-hand side must belong to

every candidate key, which is the shortcut that saves time in the exam.

Worked example 3 — R(A,B,C,D,E), F = {A → B, B → C, CD → E}

A does not appear on any right-hand side, so A is in every key. A+ = {A, B, C}, which is not the whole relation, so another attribute is needed. (AD)+ = {A, D, B, C, E} = all, so {A, D} is the candidate key. This is the classic "simple method" answer.

Q6. Explain the minimal cover with the full algorithm and worked examples.

PYQ R25 Q1(j), model questions

Definition

A minimal cover Fc is an equivalent set of FDs with a single attribute on every right-hand side, no extraneous left-hand attribute, and no redundant dependency.

Algorithm

1. **Split** every right-hand side into single attributes: A → BC becomes A → B and A → C. 2. **Remove extraneous left-hand attributes**: for each FD XA → Y, compute X+ under the current set; if Y ∈ X+, drop A. 3. **Remove redundant FDs**: for each FD X → Y, compute X+ without that FD; if Y is still in X+, drop the FD. 4. **Verify** that the result is equivalent to the original by checking closures in both directions.

Worked example 1 — F = {A → BC, B → C}

Split: {A → B, A → C, B → C}. Testing A → C: with it removed, A+ = {A, B, C} because A → B and B → C, so C is derived and the FD is redundant. **Minimal cover = {A → B, B → C}.**

Worked example 2 — F = {AB → C, A → B}

Testing whether B is extraneous in AB → C: remove B, leaving A. A+ = {A, B}, and since A determines B, AB → C applies, so A determines C. Hence B is extraneous and the FD becomes A → C. **Minimal cover = {A → B, A → C}.**

Worked example 3 — F = {A → B, C → B, D → ABC, AC → D}

Split D → ABC into D → A, D → B, D → C. The set is now {A → B, C → B, D → A, D → B, D → C, AC → D}. In AC → D neither A nor C is extraneous, because A+ = {A, B} and C+ = {C, B}, neither containing D. Testing D → B: with it removed, D+ = {D, A, C, B} through D → A, D → C and A → B, so B is derived and D → B is redundant. **Minimal cover = {A → B, C → B, D → A, D → C, AC → D}.**

Worked example 4 — the PYQ question, both readings

For R(S, T, U, V, W) with F = {ST → UW, V → W}: split into ST → U, ST → W, V → W; neither ST → W nor V → W is redundant because W cannot be derived otherwise. **Minimal cover = {ST → U, ST → W, V → W}.** The attributes never on a right-hand side are S, T and V, and (STV)+ = {S, T, U, V, W}, so **candidate key = {S, T, V}**. If instead the paper reads F = {ST → UVW, V → TW}, the minimal cover is {ST → U, ST → V, ST → W, V → T, V → W} and the candidate key is {S, T}. Both readings should be shown if the question is unclear.

Worked example 5 — F = {A → BC, B → C, C → D}

Split: {A → B, A → C, B → C, C → D}. A → C is redundant because A → B and B → C. **Minimal cover = {A → B, B → C, C → D}.**

Worked example 6 — F = {AB → C, A → B, C → D}

B is extraneous in AB → C, so the set becomes {A → C, A → B, C → D}. No FD is redundant. **Minimal cover = {A → B, A → C, C → D}.**

Why minimal cover matters

It gives the smallest set of constraints that the DBMS must enforce, it is the basis of the synthesis algorithm that produces a dependency-preserving 3NF decomposition, and it removes the ambiguity

of having many equivalent FD sets for the same relation.

Q7. Explain when two sets of functional dependencies are equivalent, and test the given pairs.

PYQ R25 Q5(a), B26 Q5(a)

The condition

Two sets F and G are **equivalent** (F ≡ G) if and only if F+ = G+, which holds if and only if every FD of G is implied by F and every FD of F is implied by G. In practice: for each X → Y in G check Y ⊆ X+ computed under F, and for each X → Y in F check Y ⊆ X+ computed under G. Equivalently, compute a minimal cover of each and compare.

Test 1 — R(W, X, Y, Z)

F1 = {W → X, X → Y, YW → Z} and F2 = {W → X, X → Y, W → Z}. Check F1 implies F2: under F1, W+ = {W, X, Y, Z} — W → X, X → Y, and then YW → Z gives Z. So W → Z is implied. Check F2 implies F1: under F2, (YW)+ = {Y, W, X, Z} — W → X, W → Z. So YW → Z is implied. **The two sets are equivalent.**

Test 2 — R(A, B, C, D, E, F)

F1 = {A → BC, B → DE, E → AB} and F2 = {A → BCF, B → DE, E → AB}. Check F1 implies F2: under F1, A+ = {A, B, C, D, E} — A → BC, B → DE, and E → AB adds nothing new. F is not in A+, so **A → F is not implied**. Check F2 implies F1: under F2, A+ contains F, and A → BCF implies A → BC, so F1 is implied. **The sets are not equivalent**: F2 is strictly stronger than F1.

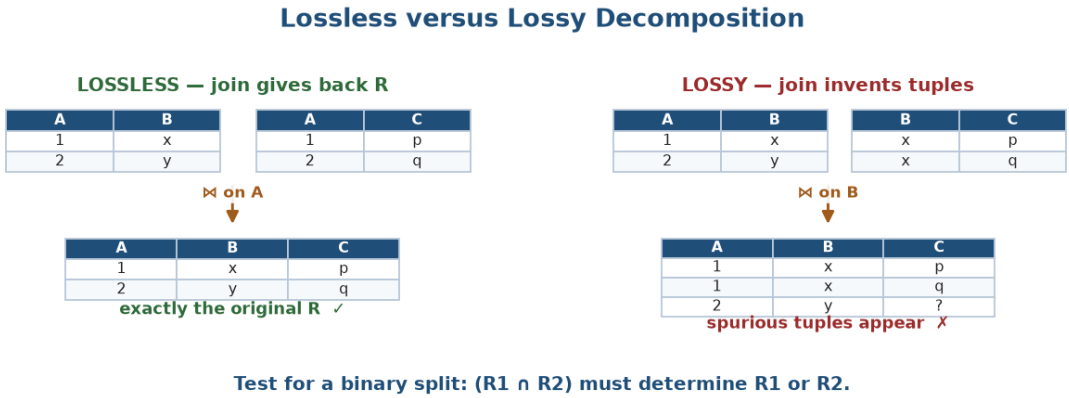
A third standard pair

F = {A → BC, AB → C, D → AC, D → E} and G = {A → BC, D → AE}. Under F, A+ = {A, B, C} and D+ = {D, A, C, E, B} = all attributes, so D → AE is implied and G ⊆ F+. Under G, D+ = {D, A, E, B, C} = all, so D → AC is implied, and AB → C follows from A → BC. Hence **F ≡ G**.

Why equivalence is asked

Because the same relation can be described by many different FD sets, and the DBA wants the smallest. Two equivalent sets describe the same set of valid relation instances, so either may be used for normalization.

Q8. Explain lossless join decomposition and dependency preservation, with the standard tests.



PYQ B26 Q1(j), model questions

Lossless join

A decomposition is **lossless** if joining the fragments reproduces exactly R. For a binary decomposition the test is: (R1 ∩ R2) → R1 or (R1 ∩ R2) → R2. The common attributes must functionally determine all attributes of at least one fragment. For more than two fragments, the **chase** or the tabular (matrix) method is used.

Why the test works

If the common attributes determine one fragment, every tuple of that fragment is fully known once the common part is known, so no combination with the other fragment can be invented. If neither is determined, the join pairs values that never occurred together and creates spurious tuples.

The PYQ question — R(A,B,C,D,E), F = {A → BC, CD → E}, into R1(A,B,C) and R2(C,D,E)

R1 ∩ R2 = {C}. Compute C+ under F: C determines nothing, so C+ = {C}. Is C → R1? No. Is C → R2? No, because CD → E needs D as well. **The decomposition is not lossless — it is lossy.** A correct lossless decomposition would be R1(A, B, C) and R2(A, C, D, E) or R1(A, B, C) and R2(C, D, E) only if C were a key of one of them.

Dependency preservation

A decomposition is **dependency preserving** if every FD of F is implied by the union of the projections of F onto the fragments. Practically, each FD should be checkable inside one fragment without a join. For the same example, A → BC lives in R1, and CD → E lives in R2, so this decomposition is dependency preserving even though it is lossy — a useful reminder that the two properties are independent.

Table of standard checks

Relation and F	Decomposition	Common attribute	Result
R(ABCD), A→B, B→C, C→D	R1(ABC), R2(CD)	C, and C → D	Lossless
R(ABCD), A→B, B→C, C→D	R1(ACD), R2(BC)	C, but C determines neither	Lossy
R(ABC), A→B, B→C	R1(AC), R2(BC)	C, but C determines neither	Lossy
R(ABC), A→B	R1(AB), R2(AC)	A, and A → AB	Lossless
R(ABC), A→B	R1(AB), R2(BC)	B, but B determines neither	Lossy
R(ABCD), A→B, C→D	R1(AB), R2(CD)	None	Lossy
R(ABCDE), A→BC, CD→E	R1(ABC), R2(CDE)	C, but C determines neither	Lossy
R(ABCDE), A→BC, CD→E, E→A	R1(ABC), R2(CDE)	C, but C determines neither	Lossy
R(XYZ), X→Y	R1(XY), R2(XZ)	X, and X → XY	Lossless
R(PQRST), P→Q, Q→R, R→S, S→T	R1(PQ), R2(QR), R3(RS), R4(ST)	Chain of keys	Lossless
R(UVWXY), U→VW, V→X, W→Y	R1(UVW), R2(VX), R3(WY)	V and W, each a key	Lossless

The design goal

A good decomposition is lossless, dependency preserving, and in a suitable normal form. A binary split on a violating FD is always lossless, which is why BCNF decompositions are lossless by construction — but they may fail dependency preservation, whereas a 3NF synthesis is always both.

Q9. Normalize R(SSN, ENAME, PNUMBER, PNAME, PLOCATION, HOURS, CITY, STATUS) up to BCNF.

PYQ R25 Q5(b)

The relation and its dependencies

R(SSN, ENAME, PNUMBER, PNAME, PLOCATION, HOURS, CITY, STATUS) with SSN → (ENAME, CITY), PNUMBER → (PNAME, PLOCATION), {SSN, PNUMBER} → HOURS and CITY → STATUS.

Step 1 — find the candidate key

SSN+ = {SSN, ENAME, CITY, STATUS}; PNUMBER+ = {PNUMBER, PNAME, PLOCATION}; {SSN, PNUMBER}+ = all eight attributes. **Candidate key = {SSN, PNUMBER}**, and every other attribute is non-prime.

Step 2 — check 3NF and BCNF

SSN → ENAME, CITY: SSN is not a super key and ENAME and CITY are non-prime, so this violates 3NF and BCNF. PNUMBER → PNAME, PLOCATION: the same problem. CITY → STATUS: STATUS is non-prime, so this violates both as well. The relation is therefore in 1NF only with respect to these dependencies; it is **not in 3NF and not in BCNF**.

Step 3 — decompose to BCNF, one violating FD at a time

Remove CITY → STATUS: **R1(CITY, STATUS)** and the rest R2(SSN, ENAME, PNUMBER, PNAME, PLOCATION, HOURS, CITY). In R2, SSN → ENAME, CITY violates: **R3(SSN, ENAME, CITY)** and R4(SSN, PNUMBER, PNAME, PLOCATION, HOURS). In R4, PNUMBER → PNAME, PLOCATION violates: **R5(PNUMBER, PNAME, PLOCATION)** and **R6(SSN, PNUMBER, HOURS)**.

The final BCNF design

Relation	Attributes	Key
CITY_STATUS	CITY, STATUS	CITY
EMPLOYEE	SSN, ENAME, CITY	SSN
PART	PNUMBER, PNAME, PLOCATION	PNUMBER
WORKS	SSN, PNUMBER, HOURS	(SSN, PNUMBER)

Step 4 — verify

Every determinant in every relation is now a super key, so the design is in BCNF. Each split was made on a violating FD, so the decomposition is lossless. Every original FD appears inside some fragment — SSN → ENAME, SSN → CITY, PNUMBER → PNAME, PNUMBER → PLOCATION, {SSN, PNUMBER} → HOURS and CITY → STATUS — so the decomposition is also dependency preserving.

Q10. Find all candidate keys for R(A,B,C,D,E,F,G,H,I,J,K) and normalize to BCNF.

PYQ R24 Q5(a)

The relation and dependencies

R(A, B, C, D, E, F, G, H, I, J, K) with F = {A → BC, A → DE, B → F, F → GH, D → IJ, K → B}.

Step 1 — find the candidate key

The attributes that never appear on a right-hand side are A, E and K, so each must be in every candidate key. Compute closures:

Attribute set	Closure	Reason
A+	A, B, C, D, E, F, G, H, I, J	A → BC, A → DE, B → F, F → GH, D → IJ
K+	K, B, F, G, H	K → B, B → F, F → GH
(A,K)+	All eleven attributes	A gives B–J, K gives B

E is already determined by A, so it is not needed separately. **The only candidate key is {A, K}.**

Step 2 — check BCNF

A → B is non-trivial and A is not a super key, so the relation is **not in BCNF**. The same is true of B → F, F → GH, D → IJ and K → B.

Step 3 — decompose to BCNF

Relation	Attributes	Key	Dependencies held
R1	A, B, C, D, E	A	A → BC, A → DE

Relation	Attributes	Key	Dependencies held
R2	B, F	B	B → F
R3	F, G, H	F	F → GH
R4	D, I, J	D	D → IJ
R5	K, B	K	K → B
R6	A, K	(A, K)	The global key

R2 was originally B, F, G, H, but F → GH violates BCNF there because F is not a super key, so it splits into R2 and R3.

Step 4 — verify

Every determinant in R1 to R6 is a super key of its own relation, so the design is in BCNF. Each split was on a violating FD, so the decomposition is lossless. Every FD of F appears in exactly one fragment — A → BC and A → DE in R1, B → F in R2, F → GH in R3, D → IJ in R4, K → B in R5 — so it is also dependency preserving. R6 is added because the candidate key {A, K} is not contained in any other fragment, and keeping the key is what guarantees losslessness for the whole decomposition.

The answer in one line

Candidate key {A, K}; not in BCNF; decompose into R1(A,B,C,D,E), R2(B,F), R3(F,G,H), R4(D,I,J), R5(K,B) and R6(A,K).

Q11. Explain dependency-preserving decomposition with the standard questions.

PYQ model questions, B26 Q1(j)

The idea

A decomposition is dependency preserving if the union of the FDs projected onto the fragments is equivalent to the original F. Practically, the DBMS must be able to check every constraint by looking at one table, without performing a join.

How to test

For each FD X → Y in F, take the attributes of each fragment Ri, form the projection $\pi_{Ri}(F)$, and check whether X → Y is implied by the union of those projections. Equivalently, compute the closure of X using only the projected FDs and see whether Y appears.

Worked checks

Relation and F	Decomposition	Preserved ?	Reason
R(ABCD), A→B, B→C, C→D	R1(ABC), R2(CD)	Yes	A→B, B→C in R1; C→D in R2
R(ABCD), A→B, B→C, C→D	R1(ACD), R2(BC)	No	A→B cannot be checked anywhere
R(ABC), A→B, B→C	R1(AC), R2(BC)	No	A→B lost; only A→C survives in R1
R(ABCD), AB→C, C→D, D→A	R1(ABC), R2(CD)	No	D→A is lost
R(ABCDE), A→B, A→C, DE→C, DE→B, C→D	R1(ABC), R2(BCDE), R3(AE)	No	DE→B and DE→C are lost

The trade-off

A 3NF decomposition can always be made both lossless and dependency preserving, using the synthesis algorithm: compute a minimal cover, make one relation per FD, and add a relation for the candidate key if none contains it. A BCNF decomposition is always lossless but may lose dependencies, which is why some designers stop at 3NF.

The synthesis algorithm for 3NF

1. Compute a minimal cover Fc. 2. For each FD X → Y in Fc, create a relation XY. 3. If no relation contains a candidate key of R, add one. 4. Remove any relation that is a subset of another. The result is lossless, dependency preserving and in 3NF.

Q12. Explain 4NF and 5NF with examples.

Syllabus Unit IV

Why 4NF is needed

A relation can be in BCNF and still contain redundancy caused by two independent multivalued facts. STUDENT(StudentID, Hobby, Language) with StudentID ↠ Hobby and StudentID ↠ Language must store every hobby against every language, so adding one hobby forces a row for each language.

StudentID	Hobby	Language
S1	Chess	English
S1	Chess	Hindi
S1	Music	English
S1	Music	Hindi

The 4NF condition

A relation is in 4NF if it is in BCNF and, for every non-trivial multivalued dependency X ↠ Y, X is a super key. Here StudentID is not a super key, so the relation violates 4NF. The decomposition is STUDENT_HOBBY(StudentID, Hobby) and STUDENT_LANGUAGE(StudentID, Language), which removes the cross product.

Why 5NF is needed

A relation can be in 4NF and still be decomposable into three or more fragments without loss, because of a cyclic join dependency. The classic example is SUPPLIER(Sname, Part, Project) with the join dependency *((Sname, Part), (Part, Project), (Sname, Project)). If every supplier supplies a part to every project that uses it, the three-way table contains redundant rows that two-way projections can reconstruct.

The 5NF condition

A relation is in 5NF (Project-Join Normal Form) if it is in 4NF and every join dependency is implied by the candidate keys. When it is not, the relation is decomposed into the projections named by the join dependency: SUPPLIER_PART(Sname, Part), PART_PROJECT(Part, Project) and SUPPLIER_PROJECT(Sname, Project).

The comparison

Normal form	Removes	Condition
4NF	Multivalued dependency	Every MVD determinant is a super key
5NF	Join dependency	Every JD is implied by candidate keys
6NF	Remaining non-atomic facts	Used mainly for temporal databases

Practical note

4NF and 5NF are rarely violated in ordinary business schemas, because multivalued attributes are usually separated at design time by the ER-to-relational mapping rules. They are examined mainly as definitions with one standard example each.

Q13. Explain the three algorithms required by the syllabus — finding a candidate key, the closure of an attribute set, and the minimal cover.

Model question — Unit 4

1. Algorithm to find the closure of an attribute set X+

1. Set X+ = X. 2. For each FD Y → Z in F, if Y ⊆ X+, add Z to X+. 3. Repeat step 2 until no new attribute can be added. 4. X+ is the final set. Worked example: R(A, B, C, D, E) with F = {A → B, B → C, C → D, D → E} gives A+ = {A, B, C, D, E}, so A is a super key.

2. Algorithm to find a candidate key

1. Find the attributes that never appear on the right-hand side of any FD — they must be in every candidate key. 2. Start with that set and compute its closure. 3. Add attributes one at a time while the closure is not the whole relation. 4. Check minimality by removing each attribute in turn. Worked example: R(A, B, C, D, E) with F = {A → B, B → C, CD → E}. A and D never appear on a right-hand side, and (AD)+ = all five attributes, so **{A, D}** is the candidate key.

3. Algorithm to find a minimal cover

1. Split every right-hand side into single attributes. 2. Remove extraneous left-hand attributes. 3. Remove redundant dependencies. 4. Verify equivalence with the original set. Worked example: F = {A → BC, B → C} becomes {A → B, B → C} after A → C is found redundant.

Why the three belong together

The closure is the primitive operation; the candidate-key algorithm uses it repeatedly; and the minimal-cover algorithm uses it twice, once for extraneous attributes and once for redundant dependencies. Every normalization question is answered by chaining these three.

Q14. Explain the 3NF synthesis algorithm and the BCNF decomposition algorithm, and compare their guarantees.

Model question — Unit 4

3NF synthesis

1. Compute a minimal cover Fc of F. 2. Create one relation XY for each FD X → Y in Fc. 3. If no relation contains a candidate key of R, add one holding a candidate key. 4. Remove any relation that is a subset of another.

BCNF decomposition

1. Check each FD X → Y. If X is not a super key, the relation violates BCNF. 2. Split R into R1(XY) and R2(R − Y). 3. Repeat for each fragment until every determinant is a super key.

The comparison

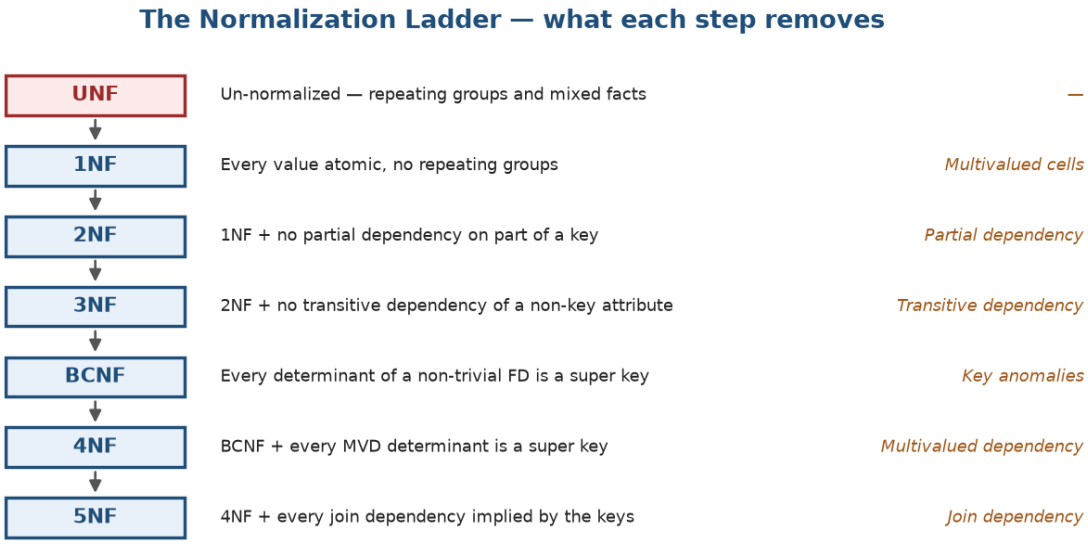
Aspect	3NF synthesis	BCNF decomposition
Starting point	Minimal cover	Violating FD in the current relation
Lossless	Always	Always
Dependency preserving	Always	Not guaranteed
Resulting form	3NF	BCNF
Number of relations	One per FD plus a key relation	Depends on the violations found

What to conclude

3NF is the safe target, because it is the strongest form that can always be reached while keeping every dependency checkable without a join. BCNF removes the last anomalies but may force a join to enforce a constraint, which is why many designers stop at 3NF and use BCNF only when the schema has overlapping candidate keys.

Q15. Explain normalization with a complete worked example from an un-normalized relation to BCNF.

Model question — Unit 4



Step 0 — the un-normalized relation

STUDENT_REPORT(RollNo, Name, Phone, Course1, Course2, Course3, DeptID, DeptName). The three course columns are a repeating group, so the relation is not even in 1NF.

Step 1 — 1NF

Split the repeating group into rows: STUDENT_COURSE(RollNo, Name, Phone, Course, DeptID, DeptName). Every value is now atomic.

Step 2 — 2NF

The key is (RollNo, Course). Name, Phone, DeptID and DeptName depend only on RollNo, which is a partial dependency. Decompose into STUDENT(RollNo, Name, Phone, DeptID, DeptName) and ENROLL(RollNo, Course).

Step 3 — 3NF

In STUDENT, RollNo → DeptID and DeptID → DeptName, so DeptName depends transitively on the key. Decompose into STUDENT(RollNo, Name, Phone, DeptID) and DEPARTMENT(DeptID, DeptName).

Step 4 — BCNF

Every remaining determinant is a super key of its relation, so the design is already in BCNF. If a case such as R(Student, Course, Instructor) with Instructor → Course appeared, it would be split into (Instructor, Course) and (Student, Instructor).

The final design

STUDENT(RollNo, Name, Phone, DeptID), DEPARTMENT(DeptID, DeptName), ENROLL(RollNo, Course). Each fact is stored once, and every relation is in BCNF.