DATABASE ENGINEERING

Unit 4

Normalization, Functional Dependencies and Decomposition

Topic-wise study notes with importance ratings

35 topics — anomalies, the full dependency theory, closure and minimal cover, all six normal forms, and decomposition

Diagrams and worked tables are given wherever a concept needs one

Topic-wise study notes. Every topic carries an importance rating from one to three stars: three stars means the topic is asked almost every year and must be known perfectly, two stars means it is regularly asked and should be prepared, one star means it is supporting material to read once. The notes explain the concept only; worked questions and answers are kept in a separate file.

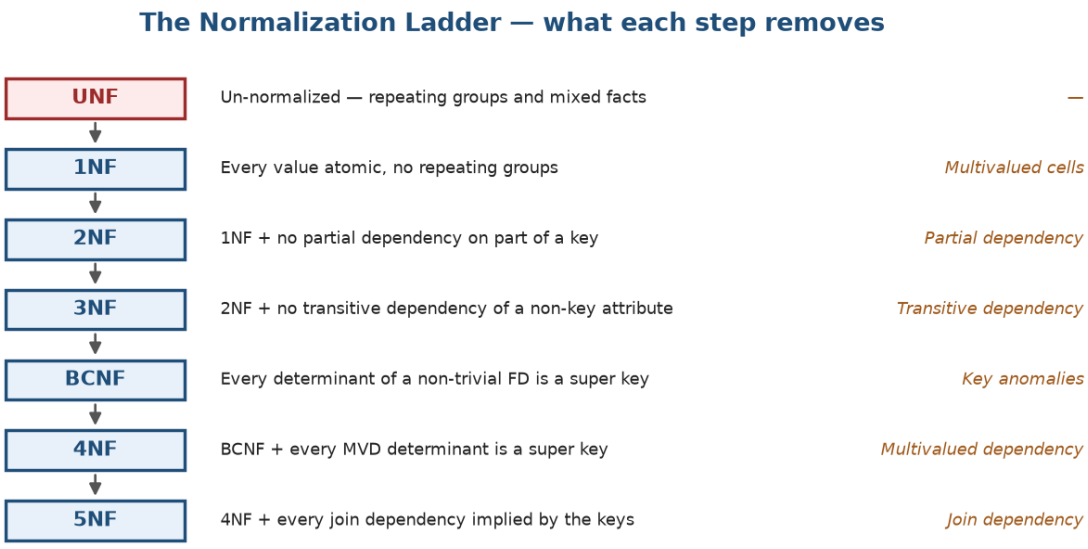
Index

Topic, importance rating and page number. ★★★ must know, ★★ should know, ★ read once.

Topic	Importance	Page
4.1 Normalization — Introduction	★★★	4
4.2 Anomalies of an Un-normalized Relation	★★★	4
4.3 Need for Normalization	★★★	5
4.4 Advantages and Limitations of Normalization	★★	5
4.5 Functional Dependency — Definition	★★★	6
4.6 Trivial and Non-trivial Functional Dependencies	★★	6
4.7 Full Functional Dependency	★★	6
4.8 Partial Dependency	★★★	7
4.9 Transitive Dependency	★★★	7
4.10 Multivalued Dependency	★★	7
4.11 Join Dependency	★★	8
4.12 Inclusion Dependency	★★	8
4.13 Dependency Diagram	★★	8
4.14 Armstrong's Axioms	★★★	9
4.15 Derived Inference Rules	★★	9
4.16 Closure of a Set of Functional Dependencies (F+)	★★	9
4.17 Attribute Closure (X+)	★★★	10
4.18 Finding Candidate Keys	★★★	10
4.19 Minimal Cover	★★★	11
4.20 Non-redundant Cover	★★	11
4.21 Minimizing the Left-hand Side of an FD	★★	12
4.22 Equivalence of FD Sets	★★★	12
4.23 First Normal Form (1NF)	★★★	12
4.24 Second Normal Form (2NF)	★★★	13
4.25 Third Normal Form (3NF)	★★★	13
4.26 Boyce-Codd Normal Form (BCNF)	★★★	14
4.27 3NF versus BCNF	★★★	14
4.28 Fourth Normal Form (4NF)	★★	15
4.29 Fifth Normal Form (5NF)	★★	15
4.30 Lossless Join Decomposition	★★★	16
4.31 Lossy Decomposition and Spurious Tuples	★★★	16
4.32 Dependency Preservation	★★★	17
4.33 The 3NF Synthesis Algorithm	★★	17
4.34 BCNF Decomposition Algorithm	★★	18

Topic	Importance	Page
4.35 Denormalization	★	18

4.1 Normalization — Introduction ★★★



Normalization is the step-by-step process of decomposing a relation into smaller, well-structured relations so that data redundancy and the modification anomalies disappear.

The guiding idea

Each fact should be stored **once**, in the relation determined by its determinant. Normalization is not about making tables small; it is about making each relation hold exactly one kind of fact.

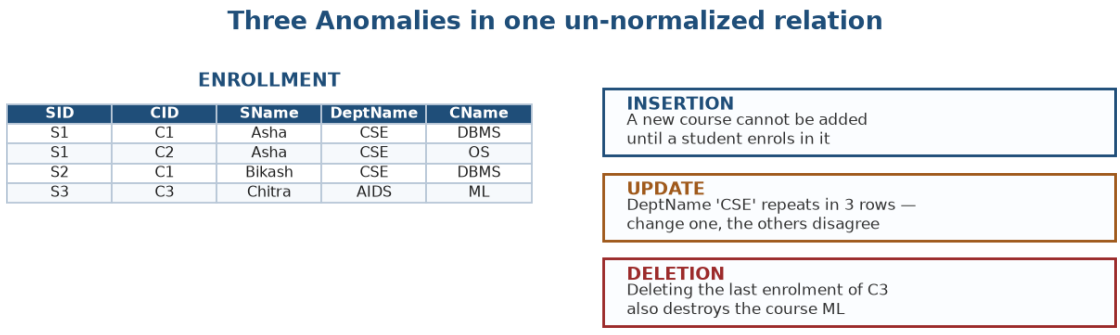
What drives the process

Functional dependencies. Every normal form is defined as a condition on the dependencies present in a relation, so the first step in any normalization question is to find the keys and the FDs.

The ladder

UNF → 1NF → 2NF → 3NF → BCNF → 4NF → 5NF. Each higher form removes a stronger kind of dependency, and each implies all the forms below it.

4.2 Anomalies of an Un-normalized Relation ★★★



The cure is to store each fact once, in the relation determined by its determinant.

An **anomaly** is a problem that arises when an un-normalized relation is modified. There are three kinds.

Anomaly	What happens	Example
Insertion	A fact cannot be recorded without an unrelated fact	A new course cannot be added until a student enrols
Update	The same fact appears in many rows, so one change needs many updates	Changing a department name across several rows
Deletion	Deleting one fact destroys another	Deleting the last enrolment removes the course itself

Why they all have the same cause

Each anomaly is a symptom of **redundancy**: the same fact is stored more than once. Remove the redundancy and all three disappear together, which is exactly what normalization does.

The test to apply in an exam

Take the relation, identify the key, list the FDs, and ask three questions: can every fact be inserted? Does every fact have exactly one home? Can anything be deleted without collateral damage? The answers name the anomalies.

4.3 Need for Normalization ★★★

The problems it solves

- Redundancy is removed, so storage is not wasted on repeated facts.
- Inconsistency cannot arise, because there is only one copy to update.
- Insertion is always possible, because no fact depends on an unrelated one being present.
- Deletion never destroys unrelated data.
- Dependencies become explicit, which makes the schema easier to reason about and to extend.

The problem it does not solve

Normalization does not improve performance — it usually costs performance, because more tables mean more joins. That is a deliberate trade: correctness first, speed recovered later through indexes, views or controlled denormalization.

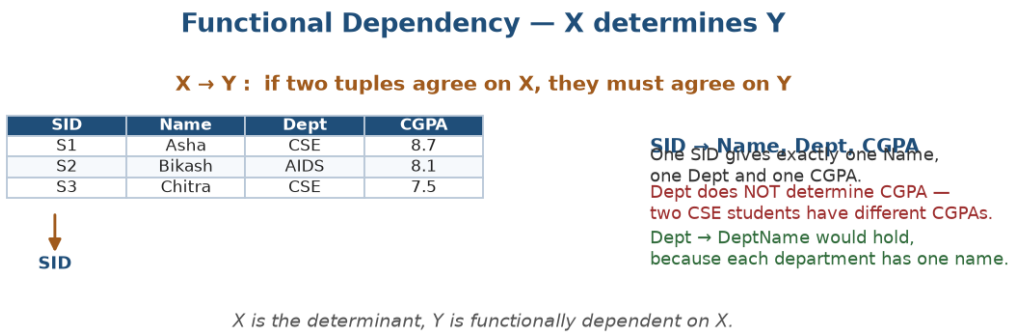
4.4 Advantages and Limitations of Normalization ★★

Advantages	Limitations
Less redundancy, less storage	More tables are created
Better consistency	Queries need more joins
Fewer modification anomalies	Complex joins can hurt performance
Clearer dependencies	Design becomes more detailed
Easier integrity enforcement	Harder to explain to non-specialists
Easier extension of the schema	Reporting systems often denormalize again

The practical conclusion

Normalize to 3NF for a transaction-processing schema, then denormalize deliberately where a measured performance problem exists. Doing it in that order means the redundancy is a decision rather than an accident.

4.5 Functional Dependency — Definition ★★★



X is the determinant, Y is functionally dependent on X.

A **functional dependency** X → Y holds in a relation R if, whenever two tuples agree on all the attributes of X, they must also agree on all the attributes of Y.

Term	Meaning
X	The determinant — the left-hand side
Y	The dependent — the right-hand side
X → Y	X functionally determines Y

Reading a dependency

StudentID → StudentName means that once the student ID is known, the name is fixed. It does **not** mean the reverse; many students may share a name, so StudentName does not determine StudentID.

Where dependencies come from

From the meaning of the data, not from the current rows. A dependency that happens to hold in today's small table but is not a rule of the business is a coincidence, and must not be used for design.

4.6 Trivial and Non-trivial Functional Dependencies ★★

Type	Condition	Example
Trivial	Y is a subset of X	AB → A, {StudentID, Name} → Name
Non-trivial	Y is not a subset of X	A → B
Completely non-trivial	X and Y share no attribute	AB → CD

Why only non-trivial ones matter

A trivial dependency is true in every relation by definition, so it carries no information about the data and never causes redundancy. Normalization is concerned only with non-trivial dependencies.

4.7 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 dependency {StudentID, CourseID} → Grade is full, because neither StudentID alone nor CourseID alone determines the grade.

Why it matters

Full dependency is the positive condition of 2NF. A relation is in 2NF when every non-prime attribute is fully dependent on every candidate key.

4.8 Partial Dependency ★★★

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) with key {StudentID, CourseID}, the dependencies StudentID → StudentName and CourseID → CourseName are partial.

When it is possible

Only when the key is composite. If every candidate key is a single attribute, partial dependency cannot occur, and a 1NF relation is automatically in 2NF.

4.9 Transitive Dependency ★★★

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. DeptName depends on the key only through DeptID.

Partial versus transitive

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
Possible only with a composite key	Possible with any key
Removed by 2NF	Removed by 3NF
Example: StudentID → StudentName	Example: DeptID → DeptName when EmpID → DeptID

4.10 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. One student with two hobbies and two languages needs four rows, and adding a third hobby needs two more.

Why no FD explains it

No functional dependency is violated — nothing determines anything else. The redundancy comes from two independent multivalued facts sharing one relation, which is why a separate normal form

(4NF) was needed.

4.11 Join Dependency ★★

A **join dependency** states that a relation can be reconstructed exactly by joining specified projections of itself: $R = \pi_{R1}(R) \bowtie \pi_{R2}(R) \bowtie \ldots \bowtie \pi_{Rn}(R)$

Example

SUPPLIER(Sname, Part, Project) with the join dependency *((Sname, Part), (Part, Project), (Sname, Project)) can be split into those three projections and joined back without loss.

Its place

A join dependency generalises the multivalued dependency, and it is what 5NF removes. A relation is in 5NF when every join dependency in it is implied by its candidate keys.

4.12 Inclusion Dependency ★★

An **inclusion dependency** states that the values of one attribute set must be contained in the values of another, written $R.A \subseteq S.B$.

Example

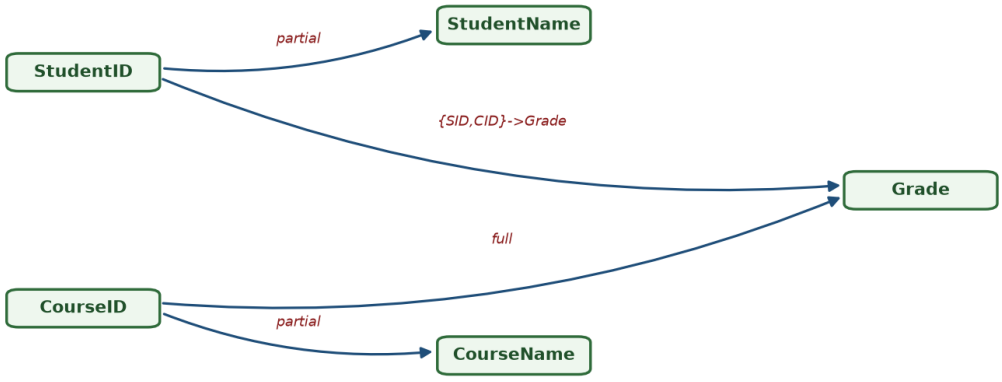
ENROLLMENT.StudentID ⊆ STUDENT.StudentID — every student appearing in an enrolment must exist in the student relation.

Its practical form

A foreign key is the common case, enforced by referential integrity. Inclusion dependencies matter in schema integration, where two databases must be merged and their value sets reconciled.

4.13 Dependency Diagram ★★

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



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

A **dependency diagram** draws each attribute as a node and each functional dependency as an arrow from the determinant to the dependent attribute.

How to read one

- Underline the candidate key first.
- An arrow leaving only **one part** of a composite key marks a **partial** dependency.

- A **chain** such as A → B → C marks a possible **transitive** dependency.
- Attributes with no incoming arrow from the key are the ones to look at when deciding the normal form.

Why it is useful

It turns a list of dependencies into a picture, so redundancy becomes visible. It is also the quickest way to decide where a relation should be split.

4.14 Armstrong's Axioms ★★★

Axiom	Statement	Example
IR1 Reflexivity	If Y ⊆ X then X → Y	{A, B} → A
IR2 Augmentation	If X → Y then XZ → YZ	A → B implies AC → BC
IR3 Transitivity	If X → Y and Y → Z then X → Z	A → B and B → C imply A → C

Why they matter

The three rules are **sound** — they generate only dependencies that are true in every relation satisfying F — and **complete** — every dependency implied by F can be generated from them. Completeness is what justifies computing attribute closure as the test for implication.

The proof of reflexivity

If two tuples agree on X they agree on every subset of X, so they agree on Y. This is why a set of attributes always determines its own subsets, and why trivial dependencies need no checking.

4.15 Derived Inference Rules ★★

Rule	Statement	Derived from
Union	If X → Y and X → Z then X → YZ	Augmentation then transitivity
Decomposition	If X → YZ then X → Y and X → Z	Reflexivity then transitivity
Pseudotransitivity	If X → Y and WY → Z then WX → Z	Augmentation then transitivity
Composition	If X → Y and Z → W then XZ → YW	Augment each, then combine

The caution on decomposition

Decomposition applies to the **right-hand side** only. From X → YZ you may conclude X → Y and X → Z, but from XZ → Y you may not conclude X → Y.

4.16 Closure of a Set of Functional Dependencies (F+) ★★

F+ is the set of all functional dependencies that can be logically derived from the given set F using Armstrong's axioms.

What it contains

F itself, every dependency implied by F, and every trivial dependency — so F+ is usually enormous and is never written out in full.

What is done instead

Attribute closure, X+, is computed for the specific attributes a question asks about. To test whether X → Y follows from F, compute X+ and check whether Y is inside it.

4.17 Attribute Closure (X+) ★★★

X+ is the set of all attributes functionally determined by X under F.

The algorithm

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 final set is X+.

Worked example

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.

The uses of closure

Finding super keys and candidate keys; testing whether an FD is implied; checking whether an FD is redundant; and driving the minimal-cover algorithm.

4.18 Finding Candidate Keys ★★★

The shortcut

Any attribute that **never appears on the right-hand side** of any FD must belong to every candidate key. Start with those attributes and compute the closure; add attributes only while the closure is not the whole relation.

Worked example 1

R(A, B, C, D, E), F = {A → B, B → C, CD → E}. A and D never appear on a right-hand side, so start with AD. (AD)+ = {A, B, C, D, E}, so **{A, D}** is the candidate key.

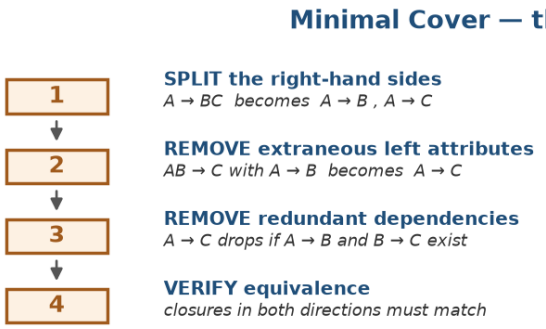
Worked example 2

R(A, B, C, D, E), F = {A → B, BC → D, D → E}. A+ = {A, B} and C+ = {C}, neither complete. (AC)+ = {A, B, C, D, E}, so **{A, C}** is the candidate key. The other closures are D+ = {D, E} and (BC)+ = {B, C, D, E}.

Worked example 3

R(A, B, C, D, E, F, G, H, I, J, K), F = {A → BC, A → DE, B → F, F → GH, D → IJ, K → B}. A and K never appear on a right-hand side, and (A, K)+ is all eleven attributes, so **{A, K}** is the only candidate key.

4.19 Minimal Cover ★★★



A **minimal cover** (canonical cover) Fc is an equivalent set of dependencies with three properties: every right-hand side has a single attribute; no left-hand-side attribute is extraneous; and no whole dependency is redundant.

The algorithm

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

Worked example 1

F = {A → BC, B → C}. Split gives {A → B, A → C, B → C}. Testing A → C: without it, A+ = {A, B, C} because A → B and B → C, so C is derived and the FD is redundant. **Fc = {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 AB → C then applies, so A determines C. Hence the FD becomes A → C and **Fc = {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. In AC → D neither A nor C is extraneous. Testing D → B: without it, D+ = {D, A, C, B} through D → A, D → C and A → B, so B is derived and D → B is redundant. **Fc = {A → B, C → B, D → A, D → C, AC → D}.**

4.20 Non-redundant Cover ★★

A **non-redundant cover** is a set of dependencies from which no entire FD can be removed without changing the closure.

How it differs from a minimal cover

Non-redundant cover	Minimal cover
No whole FD is redundant	No whole FD is redundant
Right-hand sides may hold several attributes	Every right-hand side has one attribute
Left-hand sides may contain extraneous attributes	No left-hand attribute is extraneous

So every minimal cover is non-redundant, but a non-redundant cover need not be minimal.

The redundancy test

To test whether X → A is redundant, remove it, compute X+ with the remaining FDs, and see whether A is still there.

4.21 Minimizing the Left-hand Side of an FD ★★

To test whether attribute A is extraneous in XA → Y, remove A and compute the closure of X under F. If Y is in that closure, A was extraneous and the FD reduces to X → Y.

Example

With AB → C and A → B, removing B leaves A. A+ = {A, B}, and since A determines B the FD AB → C applies, so A determines C. Therefore B is extraneous and AB → C becomes A → C.

Why it is worth doing

A smaller left-hand side means fewer attributes must be checked on every insert, and it is a required step before the cover can be called minimal.

4.22 Equivalence of FD Sets ★★★

Two sets F and G are **equivalent** if and only if F+ = G+, which holds exactly when every FD of G is implied by F and every FD of F is implied by G.

The test

For each X → Y in G, compute X+ under F and check Y ∈ X+. Then do the reverse. Equivalently, compute a minimal cover of each set and compare them.

Worked example 1

R(W, X, Y, Z), FD1 = {W → X, X → Y, YW → Z}, FD2 = {W → X, X → Y, W → Z}. Under FD1, W+ = {W, X, Y, Z}, so W → Z is implied. Under FD2, (YW)+ = {Y, W, X, Z}, so YW → Z is implied. **The sets are equivalent.**

Worked example 2

R(A, B, C, D, E, F), F1 = {A → BC, B → DE, E → AB}, F2 = {A → BCF, B → DE, E → AB}. Under F1, A+ = {A, B, C, D, E}; F is missing, so A → F is not implied. Under F2, A → BCF implies A → BC, so F2 implies F1. **Not equivalent — F2 is strictly stronger.**

4.23 First Normal Form (1NF) ★★★

A relation is in **1NF** if every attribute holds only **atomic, indivisible** values and there are no repeating groups or multivalued cells.

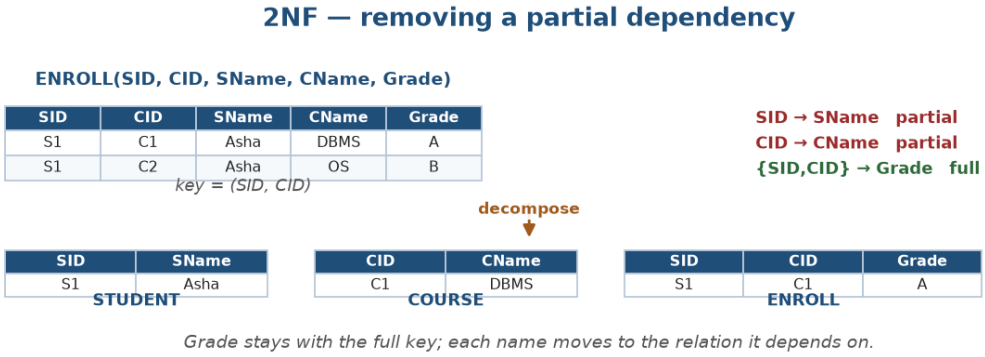
Violation and repair

Not in 1NF	In 1NF
S01, Amit, Phone = "9876, 9123"	S01, Amit, 9876 and S01, Amit, 9123

What 1NF does and does not do

It removes multivalued cells, so every value can be addressed individually. It does **not** remove partial or transitive dependencies — a relation can be in 1NF and still suffer every anomaly.

4.24 Second Normal Form (2NF) ★★★



A relation is in **2NF** if it is in 1NF and every non-prime attribute is **fully** functionally dependent on every candidate key — that is, there is no partial dependency.

The test

Find the candidate keys. For every non-prime attribute, ask whether it depends on the whole key or only on part of it. Any attribute depending on part of a composite key is a violation.

Example

ENROLL(StudentID, CourseID, StudentName, CourseName, Grade) with key {StudentID, CourseID}. StudentID → StudentName and CourseID → CourseName are partial, so the relation is not in 2NF.

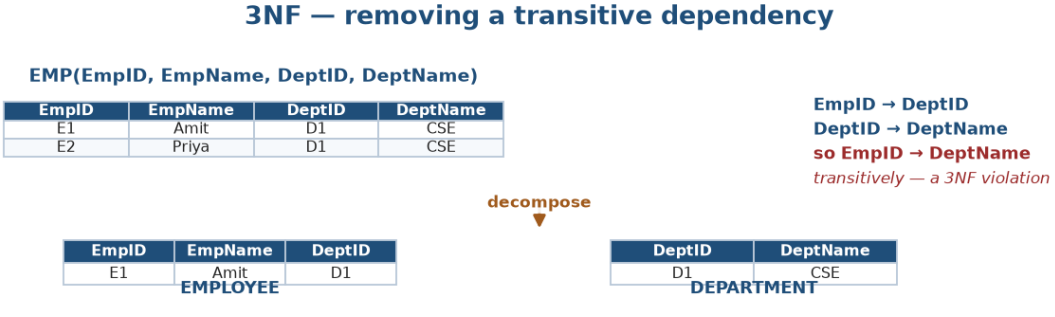
The decomposition

STUDENT(StudentID, StudentName), COURSE(CourseID, CourseName), ENROLL(StudentID, CourseID, Grade). Each attribute now depends on the whole key of its own relation.

The shortcut

If every candidate key is a single attribute, partial dependency cannot occur, and a 1NF relation is automatically in 2NF.

4.25 Third Normal Form (3NF) ★★★



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.

The formal test

For every non-trivial FD X → A, at least one of these must hold: X is a super key, **or** A is a prime attribute. The second condition is the exception that makes 3NF weaker than BCNF.

Example

EMP(EmpID, EmpName, DeptID, DeptName) with EmpID → DeptID and DeptID → DeptName. DeptName is non-prime and depends transitively on EmpID, so the relation is not in 3NF.

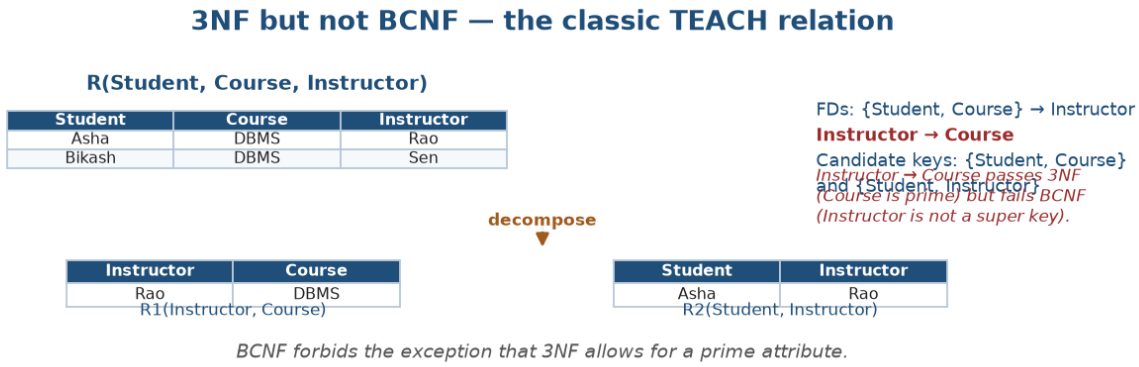
The decomposition

EMPLOYEE(EmpID, EmpName, DeptID) and DEPARTMENT(DeptID, DeptName), with DeptID as a foreign key in EMPLOYEE.

The memory rule

2NF removes partial dependency; 3NF removes the transitive dependency of a non-prime attribute.

4.26 Boyce-Codd Normal Form (BCNF) ★★★



A relation is in **BCNF** if, for every non-trivial functional dependency X → Y, X is a **super key**.

The one-line rule

Every determinant must be a super key.

Why it is stronger than 3NF

3NF allows X → A when X is not a super key provided A is **prime**. BCNF does not allow that exception. So every BCNF relation is in 3NF, but a 3NF relation need not be in BCNF.

The classic counter-example

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 prime. The BCNF decomposition is R1(Instructor, Course) and R2(Student, Instructor).

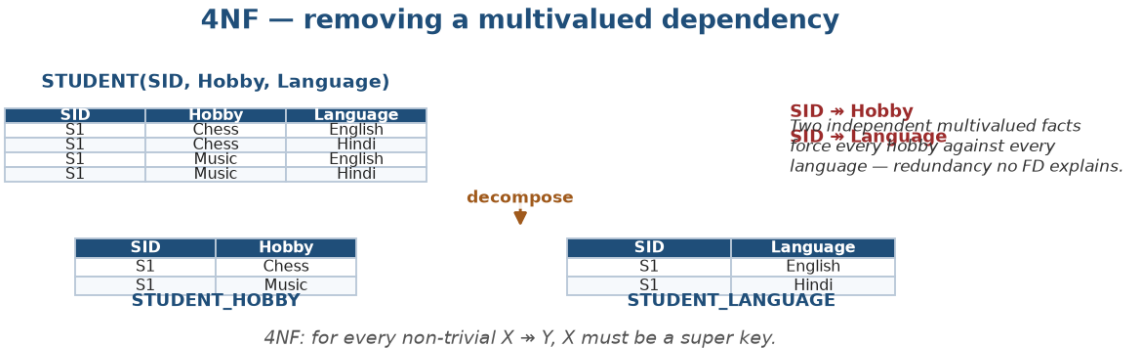
4.27 3NF versus BCNF ★★★

3NF	BCNF
Condition: X is a super key OR A is prime	Condition: X must be a super key
Allows an FD from a non-super-key determinant if the dependent is prime	Allows no such exception
Every 3NF relation need not be BCNF	Every BCNF relation is in 3NF
Decomposition can always be made lossless AND dependency preserving	Decomposition is always lossless but may lose dependencies
Practical target for most schemas	Used when the schema has overlapping candidate keys

What to say in an exam

State the condition of each, give the TEACH relation as the example that separates them, and add that 3NF synthesis always preserves dependencies while BCNF decomposition may not. Those three points carry the marks.

4.28 Fourth Normal Form (4NF) ★★



A relation is in **4NF** if it is in BCNF and, for every non-trivial multivalued dependency X ↠ Y, X is a super key.

Example

STUDENT(StudentID, Hobby, Language) with StudentID ↠ Hobby and StudentID ↠ Language is in BCNF, because no functional dependency is violated. It is not in 4NF, because StudentID is not a super key of the relation.

The decomposition

STUDENT_HOBBY(StudentID, Hobby) and STUDENT_LANGUAGE(StudentID, Language). The cross product disappears and each fact is stored once.

How to recognise one

Two independent multivalued facts stored in the same relation — hobbies and languages, skills and projects, colours and sizes. If the table contains every combination, an MVD is present.

4.29 Fifth Normal Form (5NF) ★★

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.

Example

SUPPLIER(Sname, Part, Project) with the join dependency *((Sname, Part), (Part, Project), (Sname, Project)). The relation cannot be split into two projections without loss, but it can be split into those three.

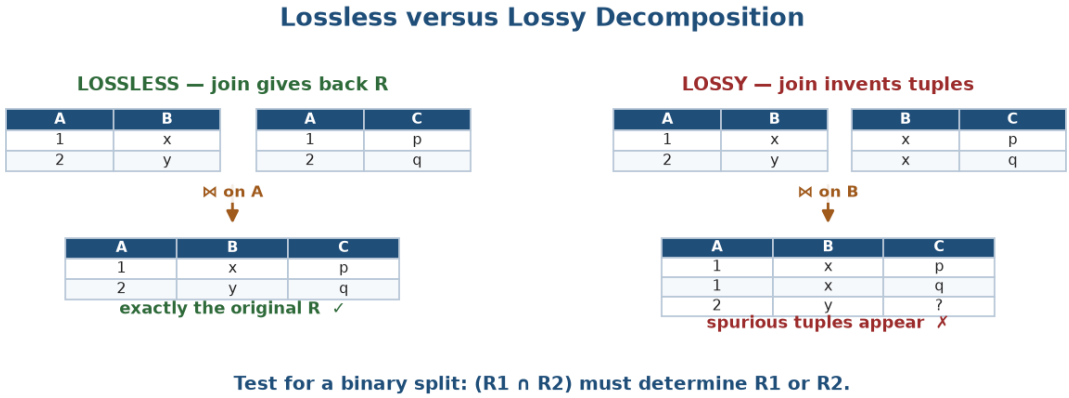
The decomposition

SUPPLIER_PART(Sname, Part), PART_PROJECT(Part, Project) and SUPPLIER_PROJECT(Sname, Project). Joining all three reproduces the original exactly.

Practical note

5NF is rarely violated in ordinary business schemas, because multivalued facts are usually separated at design time. It is examined mainly as a definition with this one example.

4.30 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.

The binary test

For R split into R1 and R2, the decomposition is lossless if at least one of these holds: (R1 ∩ R2) → R1 or (R1 ∩ R2) → R2 In words, the common attributes must functionally determine all attributes of at least one fragment.

Worked example — lossless

R(A, B, C, D) with F = {A → B, B → C, C → D}, split into R1(A, B, C) and R2(C, D). The common attribute is C and C → D holds, so C determines R2. **Lossless.**

Worked example — lossy

R(A, B, C, D, E) with F = {A → BC, CD → E}, split into R1(A, B, C) and R2(C, D, E). The common attribute is C and C+ = {C}, so C determines neither fragment. **Lossy.**

Why every split on a violating FD is lossless

If X → Y is non-trivial and R is split into XY and (R − Y), the common attributes are X and X determines XY, so the test is satisfied immediately. That is why BCNF algorithms are lossless by construction.

4.31 Lossy Decomposition and Spurious Tuples ★★★

A decomposition is **lossy** if the natural join produces tuples that were not in the original relation. These are called **spurious tuples**, and the information content has been destroyed.

The misleading name

"Lossy" refers to loss of **information**, not loss of rows. A lossy decomposition can produce **more** rows than the original, which is exactly the problem.

Why it happens

The common attribute does not determine either fragment, so the join pairs values that never occurred together. In the example above, C alone cannot say which D value belonged to which A value, so every combination is invented.

The rule

Never accept a decomposition that fails the binary test. If a decomposition is given in a question and the test fails, the answer is that it is lossy, and a correct alternative should be offered.

4.32 Dependency Preservation ★★★

A decomposition is **dependency preserving** if every FD of the original set can be enforced by checking the fragments individually, without joining them.

The test

For each FD X → Y in F, compute X+ using only the FDs that lie **inside** the fragments — those whose attributes all appear in the same Ri. If Y appears in that closure for every FD, the decomposition preserves all dependencies.

Worked example — preserved

R(A, B, C, D) with F = {A → B, B → C, C → D}, split into R1(A, B, C) and R2(C, D). Inside R1 we have A → B and B → C; inside R2 we have C → D. Every FD lives inside one fragment, so the decomposition is dependency preserving — and by the earlier test it is also lossless.

Worked example — not preserved

The same relation split into R1(A, C, D) and R2(B, C). R2 keeps B → C and R1 keeps C → D, but **A → B cannot be checked anywhere**, because A and B never appear in the same fragment.

The independence of the two properties

Losslessness and dependency preservation are independent. A decomposition can be dependency preserving but lossy, or lossless but not dependency preserving. A good design is both, and 3NF synthesis always achieves both.

4.33 The 3NF Synthesis Algorithm ★★

The steps

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

The guarantee

The result is **lossless**, **dependency preserving** and in **3NF** — all three at once. That is why 3NF is the practical target for most schemas.

Example

R(A, B, C, D) with F = {A → B, B → C, C → D}. The minimal cover is the same set, so the synthesis gives R1(A, B), R2(B, C), R3(C, D). The candidate key is A, which is contained in R1, so no extra relation is needed.

4.34 BCNF Decomposition Algorithm ★★

The steps

1. Check each FD X → Y in R. 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 guarantee and the cost

Each split is on a violating FD, so the decomposition is always **lossless**. It may, however, **lose dependencies**, because the determinant and the dependent attribute can end up in different fragments. That is the trade-off against 3NF.

Worked example

R(SSN, ENAME, PNUMBER, PNAME, PLOCATION, HOURS, CITY, STATUS) with SSN → (ENAME, CITY), PNUMBER → (PNAME, PLOCATION), {SSN, PNUMBER} → HOURS, CITY → STATUS. Candidate key {SSN, PNUMBER}. Splitting on each violation gives CITY_STATUS(CITY, STATUS), EMPLOYEE(SSN, ENAME, CITY), PART(PNUMBER, PNAME, PLOCATION) and WORKS(SSN, PNUMBER, HOURS). Every determinant is now a super key, every split was lossless, and every original FD is still checkable inside one fragment.

4.35 Denormalization ★

Denormalization is the deliberate reintroduction of redundancy into a normalized design — by merging relations or by storing derived values — to reduce joins and improve read performance.

Reason to denormalize	Cost accepted
Too many joins slow a report	Update anomalies return
Aggregates are recomputed constantly	Stored totals can drift out of date
A star schema is being built for analytics	Storage increases

The discipline

Denormalize only after measuring a real performance problem, and only in a controlled way — a materialized view or a summary table is safer than merging base tables, because the redundancy is contained in one object that can be refreshed.

Why it is in the syllabus

Because the answer to "is normalization always good?" is no. Normalization is the default for correctness; denormalization is an optimization, and a good answer says both.