INFS1200 — Week 11 Notes

Table of contents

Normalisation and Relational Database Schema Design **2**
- Outline . . . 2
- Key concepts recap . . . 2
- Approaching normality . . . 3
- First Normal Form (1NF) . . . 3
- Full and partial functional dependencies . . . 4
- Second Normal Form (2NF) . . . 4
- Transitive dependency . . . 5
- Third Normal Form (3NF) . . . 5
- Boyce-Codd Normal Form (BCNF) . . . 5
- Question 8 — Partial vs. transitive dependency . . . 6
- Question 9 — Validating normal form . . . 7
- Question 10 — Validating normal form . . . 7
- Question 11 — Validating normal form . . . 7
- Question 12 — Validating normal form . . . 7
- Question 13 — BCNF and 3NF . . . 8
- Relational database schema design . . . 8
- Question 14 — Determining which FDs apply . . . 8
- Minimal cover . . . 9
- Question 15 — Minimal cover . . . 10
- 3NF synthesis algorithm . . . 10
- Question 16 — 3NF synthesis . . . 11
- BCNF decomposition algorithm . . . 11
- Question 17 — BCNF decomposition . . . 12
- Question 18 — BCNF decomposition (with a lossy-join check) . . . 12
- Denormalisation . . . 12
- Summary . . . 13

Applied Class 10: Normalisation, BCNF Decomposition and 3NF Synthesis **13**
- Section A — BCNF decomposition (top-down) . . . 13

Section B — 3NF synthesis (bottom-up) 17

Case Study 9: Payroll System (BCNF Decomposition and 3NF Synthesis) 19
The corrected payroll schema 19
Section A — BCNF decomposition 20
Section B — 3NF synthesis 23

Reference material 24
3NF Synthesis 24
Decomposition and BCNF Design 28
Minimal Cover 36
Normal Forms (1NF to BCNF) 41

Normalisation and Relational Database Schema Design

Module 4, part 2. Continues 2026-04-27-database-design-guidelines-and-functional-dependencies[1] — now that we can identify FDs and keys, we use them to formally measure and improve schema quality via **normalisation**.

Outline

- Normalisation: 1NF, 2NF, 3NF, BCNF
- Relational database schema design: BCNF decomposition, minimal cover, 3NF synthesis

Key concepts recap

- **Superkey** — a set of attributes such that no two tuples share the same values for them (its closure covers every attribute in `R`).
- **(Candidate) key** — a *minimal* superkey (removing any attribute breaks the superkey property). Minimal shortest; a relation can have several candidate keys, and every superkey contains at least one candidate key.
- **Primary key** — one candidate key, chosen as *the* key (only one per relation).
- **Prime attribute** — an attribute that belongs to *any* candidate key (not necessarily the primary key).
- **Non-prime attribute** — belongs to no candidate key.

[1] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf

Approaching normality

Functional dependencies reveal redundancy: in `R [A, B, C]` with no FDs, there's no redundancy; but given `A → B`, several tuples can share the same `A` value, and whenever they do, they're *forced* to repeat the same `B` value too — that repetition is the redundancy **normalisation** identifies and removes.

Normalisation is the process of taking a relational schema through a series of tests, using FDs and (candidate/primary) keys, to certify which **normal form** it satisfies. Schemas that fail a test are decomposed into smaller relations with better properties.

Normal forms overview

NF	Outcome	Test (for every non-trivial `X → A` in `F`)
1NF	Identifies non-atomic values	Relation has no multivalued attributes or nested relations
2NF	Identifies partial dependencies (removes some anomalies)	`X` is *not* a proper subset of a candidate key, **or** `A` is a prime attribute
3NF	Identifies partial + transitive dependencies (removes most anomalies)	`X` is a superkey, **or** `A` is a prime attribute
BCNF	Identifies *all* anomalies (at the cost of not preserving all FDs)	`X` is a superkey

`BCNF` `3NF` `2NF` `1NF` — each normal form is strictly more restrictive than the last.

First Normal Form (1NF)

A relation schema is in 1NF if every attribute's domain contains only **atomic (simple, indivisible) values** — no set of values, no tuple of values (nested attributes), and no combination of both.

Non-1NF example — an `items` column holding a comma-separated list, or a nested `name = {firstName, familyName}` sub-structure, both violate 1NF. Normalizing to 1NF either:

- **repeats** the non-nested key columns per value (e.g. one row per `(customerName, orderNum, item)` combination) — introduces redundancy; or

- **flattens** into fixed columns (`item1`, `item2`, …) — introduces a lack of flexibility (a hard cap on how many items one order can have, and `null`s for orders with fewer items).

Neither is fully satisfactory — this motivates 2NF/3NF/BCNF, which address *how* to further decompose an already-1NF relation.

Full and partial functional dependencies

- `X → Y` is a **full functional dependency** if removing *any* attribute from `X` breaks the dependency (no proper subset of `X` determines `Y`).
- `X → Y` is a **partial dependency** if *some* attribute can be removed from `X` and the FD still holds.

Example — `Address [houseNum, street, postcode, state, value]`, `F = {{houseNum, street, postcode} → {state, value}, postcode → state}`:

- `{houseNum, street, postcode} → value` is a **full** dependency (no subset of the LHS determines `value`).
- `{houseNum, street, postcode} → state` is a **partial** dependency, since `postcode → state` alone already suffices.

Second Normal Form (2NF)

A relation schema `R` is in 2NF if every **non-prime** attribute is **fully** functionally dependent on the primary key — informally, *no partial dependency*.

In the `Address` example: the (only) key is `{houseNum, street, postcode}`; `state` is non-prime and only partially dependent on the key (via `postcode → state`) — this violates 2NF, so `Address` is **not** in 2NF.

Fixing it: decompose into `Address [houseNum, street, postcode, value]` and `Postcodes [postcode, state]` (`Address.postcode → Postcodes.postcode`). 2NF identifies anomalies caused by *partial* dependencies — but **not** anomalies caused by *transitive* dependencies, covered next.

2NF doesn't fix everything: `Employee [ID, name, level, salary]` with `level → salary` doesn't violate 2NF at all (`level` isn't even a *proper subset* of the key `{ID}` — it's a whole non-key attribute), so `Employee` **is** in 2NF — yet it still has all the same anomalies from before (updating one developer's salary is still inconsistent, deleting the only "Administration" employee still loses that level's salary, etc.). 2NF alone isn't enough.

Transitive dependency

`X → Y` in `R` is a **transitive dependency** if some attribute set `Z` exists such that `Z` is neither a candidate key nor a subset of one, and both `X → Z` and `Z → Y` hold.

Example — `Employee [ID, name, level, salary]`, `F = {level → salary}`: consider `ID → salary`. `level` is neither a candidate key nor a subset of one; `ID → level` and `level → salary` both hold; therefore `ID → salary` is a transitive dependency (`salary` depends on `ID` *only through* the intermediate `level`).

Third Normal Form (3NF)

A relation schema `R` with FD set `F` is in 3NF iff, for every non-trivial FD `X → A` in `F` : `X` is a superkey, **or** `A` is a prime attribute. Equivalently — for any non-trivial `X → A` where `A` is non-prime, `X` must be a superkey.

In the `Employee` example: `level → salary` — `level` is not a superkey, and `salary` is not a prime attribute — **violates 3NF**. Fixing it: decompose into `StaffAppointment [ID, name, level]` and `StaffIncome [level, salary]`. Most 3NF relations are anomaly-free — but not *all* of them, as the next example shows.

3NF still isn't always enough — `Teach [studentID, courseID, lecturer]`, `F = {{studentID, courseID} → lecturer, lecturer → courseID}` (keys: `{studentID, courseID}`, `{studentID, lecturer}`). `Teach` **is** in 3NF (`lecturer → courseID`'s RHS, `courseID`, is a prime attribute — so it doesn't violate 3NF even though `lecturer` isn't a superkey). Yet it still has anomalies:

- **Deletion anomaly**: deleting the only row for a given `(studentID, courseID)` pair can delete the *only* record of which lecturer teaches that course.
- **Insertion anomaly**: can't record which lecturer teaches a course that currently has no enrolled students.

Boyce-Codd Normal Form (BCNF)

A relation schema `R` with FD set `F` is in BCNF iff, for every non-trivial FD `X → A` in `F` : `X` is a superkey. Informally: whenever a set of attributes determines another attribute, it must determine *all* attributes of `R`.

In `Teach`: `lecturer → courseID` — `lecturer` is **not** a superkey — violates BCNF, so `Teach` is **not** in BCNF (even though it is in 3NF).

The catch: BCNF can lose dependencies

Decomposing `Teach` to fix the BCNF violation: split into `[lecturer, courseID]` (from `lecturer → courseID`) and `[studentID, lecturer]`. Each piece individually satisfies its own local FDs and is in BCNF. But now rejoin them:

```
lecturer  courseID          studentID  lecturer
John      INFS1200          1234       John
Jane      INFS1200          1234       Jane
                ↓ join ↓
studentID  courseID  lecturer
1234       INFS1200  John
1234       INFS1200  Jane
```

The rejoined table **violates** the original FD `{studentID, courseID} → lecturer` (both rows share `{1234, INFS1200}` but disagree on `lecturer`) — even though neither decomposed piece violated *any* FD on its own. **Decomposing into BCNF can fail to preserve all the original FDs.** A **dependency-preserving** decomposition is one where, if `R` is split into `R1, ..., Rn` with FD sets `F1, ..., Fn` (each `Fi` containing the FDs of `F` whose attributes fall entirely within `Ri`), then `(F1 ... Fn) = F` — i.e. no FD information is lost by the split.

BCNF vs 3NF trade-off:

	BCNF	3NF
Removes all anomalies		
Preserves all FDs		

Most organisations aim for one or the other depending on which guarantee matters more for their use case.

Question 8 — Partial vs. transitive dependency

What's the difference between partial and transitive dependency?

> **i** Solution
>
> **Partial dependency**: an attribute depends on only a *subpart* of the primary key. **Normalizing to 2NF** solves this. **Transitive dependency**: a non-prime attribute depends on *other non-prime attributes* (rather than directly on the key). **Normalizing to 3NF** solves this.

Question 9 — Validating normal form

Given `R [A, B, C, D, E, F]` with `{A, B} → {C, D, E}`, `C → F`, `E → {A, B}` — what is the highest normal form?

> **i** Solution
>
> **2NF.** Keys: `{A, B}`, `{E}`. No partial dependencies exist (both `{A,B} → {C,D,E}` and `E → {A,B}` have a *whole* key as LHS), so 2NF holds. But `C → F`: `C` is not a superkey, and `F` is non-prime — violates both 3NF and BCNF.

Question 10 — Validating normal form

Given `R [A, B, C, D]` with `{A, B} → {C, D}`, `{C, D} → A`, `D → B` — what is the highest normal form?

> **i** Solution
>
> **3NF.** Keys: `{A, B}`, `{C, D}`, `{A, D}` — every attribute is prime. Since every attribute is prime, 2NF and 3NF automatically hold (every FD's RHS is a prime attribute). But `D → B`: `D` is not a superkey — violates BCNF.

Question 11 — Validating normal form

Given `R [A, B, C, D, E]` with `B → {C, D}`, `A → E` — what is the highest normal form?

> **i** Solution
>
> **1NF.** Key: `{A, B}`. Both `A` and `B` are proper subsets of the key — `A → E` and `B → {C, D}` are both **partial dependencies**, violating 2NF.

Question 12 — Validating normal form

Given `R [A, B, C, D, E]` with `A → {B, C, D, E}`, `E → A` — what is the highest normal form?

> **i** Solution
>
> **BCNF.** Keys: `{A}`, `{E}`. The LHS of every FD (`A` and `E`) is itself a superkey — satisfies BCNF, the strictest test.

Question 13 — BCNF and 3NF

Given `R [A, B, C, D]` *with* `{A, C, D} → B`, `{A, C} → D`, `D → C`, `{A, C} → B` — *which is true?*

A. Neither BCNF nor 3NF B. BCNF but not 3NF C. 3NF but not BCNF D. Both BCNF and 3NF

> **i** Solution
>
> **C.** Keys: `{A, C}`, `{A, D}`. `D → C`: `D` is not a superkey, so **BCNF is violated**. But `C` *is* a prime attribute (appears in key `{A, C}`), so this same FD does **not** violate 3NF — `R` is in 3NF but not BCNF.

Relational database schema design

Two complementary algorithms turn a *universal relation* (all attributes lumped together, plus a set of FDs) into a well-designed multi-relation schema:

1. **Decomposition** (top-down): break the universal relation apart to reach **lossless-join, anomaly-free BCNF** schemas.
2. **Synthesis** (bottom-up): build up from individual attributes to reach **lossless-join, dependency-preserving 3NF** schemas.

```
flowchart LR
    U["Universal Relation + FDs"] -->|decomposition| RDB[("Relational DB schema")]
    E["EER Diagram (from UoD)"] -->|mapping| RDB
    A["All attributes + FDs"] -->|synthesis| RDB
```

Determining which FDs apply to a decomposed relation

For a decomposed relation `S` and an FD `X → Y` from the original `F`: `X → Y` holds in `S` if `X  Y` `S`'s attributes **and** `Y  X` (computed using *all* of `F`, even attributes not in `S`).

Question 14 — Determining which FDs apply

Given `R [A, B, C, D, E]` *with* `{A,B}→C`, `{B,C}→D`, `{C,D}→E`, `{D,E}→A`, `{A,E}→B` — *which FDs hold in* `S [A, B, C, D]`*?*

A. `A → B` B. `{A, B} → E` C. `{A, E} → B` D. `{B, C, D} → A` E. None of the above

Solution

D. {B, C, D} (computed against the *full* F, using E as an intermediate even though it's not in S): {B,C}→D gives nothing new; {C,D}→E adds E → {B,C,D,E}; {D,E}→A adds A → {A,B,C,D,E} — covers everything, and A S, so {B,C,D} → A holds in S. A fails (A = {A} alone — doesn't even reach B). B and C both fail because their RHS, E, isn't even an attribute of S — an FD "holding in S" requires both sides to be within S's attributes.

Minimal cover

To decompose into 3NF (synthesis) — or just to simplify reasoning about BCNF — we first reduce F to a **minimal cover** G: an equivalent, "as small as possible" set of FDs. G is a minimal cover for F iff:

- F = G (equivalent implied FDs);
- every FD in G has a single attribute on its RHS;
- deleting any FD in G, or any attribute from any FD's LHS, changes G (i.e. every FD, and every LHS attribute, is *necessary*).

Finding a minimal cover — 3 steps

1. **RHS simplification** — split every FD so its RHS has exactly one attribute (X → {Y, Z} becomes X → Y and X → Z).
2. **LHS simplification** — for each FD X → A where X has multiple attributes including some B: if X = (X - {B}) (i.e. B is redundant in the LHS), replace it with (X - {B}) → A.
3. **FD set simplification** — delete any FD X → A if X (computed *without* that FD) still includes A — i.e. the FD is implied by the rest of F and can be dropped.

Worked example — R [A,B,C,D,E,F,G,H], F = {A → B, {A,B,C,D} → E, {E,F} → G, {E,F} → H, {A,C,D,F} → {E,G}}:

Step 1: RHS split	Step 2: LHS simplify	Step 3: delete redundant
A→B	A→B	A→B
{A,B,C,D}→E	{A,C,D}→E (B redundant, since A→B)	{A,C,D}→E
{E,F}→G	{E,F}→G	{E,F}→G
{E,F}→H	{E,F}→H	{E,F}→H
{A,C,D,F}→E	{A,C,D,F}→E	*(dropped — {A,C,D,F} already includes E without this FD)*

Step 1: RHS split	Step 2: LHS simplify	Step 3: delete redundant
{A,C,D,F}→G	{A,C,D,F}→G	*(dropped — same reason)*

Minimal cover: {A → B, {A,C,D} → E, {E,F} → G, {E,F} → H} (often written with the last two combined as {E,F} → {G,H}).

Question 15 — Minimal cover

Given R [A, B, C, D, E, F] with {D, E, F} → C, {A, B} → {D, C}, D → F — which is a minimal cover?

A. {{D,E,F}→C, {A,B}→{D,C}, D→F} B. {{D,E,F}→C, {A,B}→D, D→F} C. {{D,E}→C, {A,B}→D, {A,B}→C, D→F} D. {{D,E}→C, {A,B}→{C,D}}

i Solution

C. Step 1 (RHS split): {D,E,F}→C, {A,B}→D, {A,B}→C, D→F. Step 2 (LHS simplify): {D,E,F}→C simplifies to {D,E}→C, since D → F already makes F redundant in that LHS ({D,E} already includes F via D→F, so {D,E,F} = {D,E}). Step 3: nothing more to remove — all four remaining FDs are necessary. A keeps the redundant F in the first FD's LHS. B drops the necessary {A,B}→C FD entirely. D drops the necessary D→F FD.

3NF synthesis algorithm

```
S :=  ;
Compute a minimal cover G of F;
Combine all FDs in G with the same LHS into one;
For each X → Y in G:
    if no relation in S contains X   Y:
        add a relation with schema X   Y to S;
if any candidate key is missing from the relations:
    add a relation containing all prime attributes;
Eliminate redundant relations (one whose attributes are a
subset of another relation already in S).
```

Example — R [A, B, C, D, E], F = {{A,B}→C, C→D} (already minimal); key: {A, B, E}. Synthesis: R1 [A,B,C] (from {A,B}→C), R2 [C,D] (from C→D); the key {A,B,E} isn't contained in either, so add R3 [A,B,E]. None of R1/R2/R3 is a subset of another — final answer: R1 [A,B,C], R2 [C,D], R3 [A,B,E].

Question 16 — 3NF synthesis

Given the minimal cover `F = {{A,C}→E, {B,D}→A, A→B, E→{C,F}}` *for* `R [A, B, C, D, E, F]` *— which is a correct 3NF synthesis?*

A. `R1[A,C,E], R2[D,B,A], R3[A,B], R4[E,C,F]` B. `R1[A,C,E], R2[A,B,D], R3[A,B], R4[E,C,F], R5[A,C,D]` C. `R1[A,C,E], R2[B,D,A], R3[E,C,F], R4[A,C,D]` D. `R1[E,C,F], R5[A,B,C,D,E]`

i Solution

D. Candidate keys are `{A,C,D}, {A,D,E}, {B,C,D}, {B,D,E}` — note `F` never appears in any key. Synthesising one relation per FD gives `[A,C,E]` (`{A,C}→E`), `[A,B,D]` (`{B,D}→A`), `[A,B]` (`A→B`), `[E,C,F]` (`E→{C,F}`). `[A,B]` is a subset of `[A,B,D]` — redundant, drop it. None of the remaining three relations contains a *full* candidate key (all four keys include `D`, but neither `[A,C,E]` nor `[E,C,F]` has `D`, and `[A,B,D]` is missing `C`/`E`), so add `R5` with all **prime** attributes: `{A, B, C, D, E}` (the union of all candidate keys). Now `[A,C,E]` and `[A,B,D]` are both subsets of `R5 [A,B,C,D,E]` — redundant, drop both. `[E,C,F]` is **not** a subset of `R5` (`F` isn't in it) — keep. Final: `[E,C,F]` and `R5 [A,B,C,D,E]`.

BCNF decomposition algorithm

```
D := {R};
while (some relation Q in D is not in BCNF):
    find an FD X → Y in Q that violates BCNF;
    replace Q in D with Q1 [Q - Y] and Q2 [X   Y];
```

(The split point: `Q1` keeps everything except the "extra" attributes `Y`; `Q2` becomes the offending FD's own little BCNF-satisfying table, since `X` is now trivially a key for it.) The algorithm terminates because every 2-attribute relation is automatically in BCNF (with 0 or 1 non-trivial FDs, the LHS is always a key). Results can vary depending on which order violating FDs are processed in — that's fine, multiple correct decompositions can exist.

Worked example — `R [A, B, C, D]`, `F = {B → C, D → A}`: keys — `{B,D} = {A,B,C,D}`, the only key. `B → C` violates BCNF (`B` isn't a key) → split into `R1 [A,B,D]`, `R2 [B,C]`. `D → A` now violates BCNF in `R1` (`D` isn't a key for `R1`) → split into `R3 [B,D]`, `R4 [A,D]`. **Final**: `R2 [B,C]`, `R3 [B,D]`, `R4 [A,D]`.

Implicit FDs matter: when checking whether a relation is in BCNF during decomposition, you must also check *implied* FDs, not just the explicitly given ones — e.g. given `A → B` and `B → C`, the implicit `A → C` might be the one that actually violates BCNF in a sub-relation, even though it was never written down explicitly.

Question 17 — BCNF decomposition

Given `R [A, B, C, D, E]` with `{A,D}→B`, `C→{D,E}`, `A→E` — which is a correct BCNF decomposition tree?

> **Solution**
>
> The correct order is: apply `{A,D} → B` first (splitting off `R1 [A,B,D]`, leaving `R2 [A,C,D,E]`), **then** apply `C → {D,E}` to `R2` (splitting it into `R3 [C,D,E]` and `R4 [A,C]`). **Final**: `R1 [A,B,D]`, `R3 [C,D,E]`, `R4 [A,C]`.
> Common mistakes to watch for: writing `R3` as `[C,E]` instead of the full `[C,D,E]` (dropping an attribute of the violating FD's RHS); writing the intermediate relation as `[A,B,C,E]` instead of the correct `[A,C,D,E]` (forgetting `A → E` only removes `B`, not `D`, from the first split); or decomposing `R2` further using `A → E` when `A` is already a key for `R2` at that point (no violation left to fix).

Question 18 — BCNF decomposition (with a lossy-join check)

Given `R [A, B, C, D]` with `A → B`, `C → D`, `{A,D} → C`, `{B,C} → A` — which is a lossless-join BCNF decomposition?

A. `{{A,B}, {A,C}, {B,D}}` B. `{{A,B}, {A,C}, {C,D}}` C. `{{A,B}, {A,C}, {B,C,D}}` D. All of the above E. None of the above

> **Solution**
>
> **B.** Keys: `{A,D}`, `{A,C}`, `{B,C}` (all three give closure `{A,B,C,D}`). `A → B` violates BCNF → split into `R1 [A,C,D]`, `R2 [A,B]`. `C → D` violates BCNF in `R1` → split into `R3 [A,C]`, `R4 [C,D]`. **Final**: `{{A,B}, {A,C}, {C,D}}`.
> Why not A (`{{A,B}, {A,C}, {B,D}}`)? Concretely: take `R` rows `(1,2,5,6)`, `(1,2,3,7)`, `(8,2,9,4)`. Decomposing per option A and rejoining produces two *extra* rows not in the original `R` (e.g. `(1,2,3,4)` appears in the rejoined result despite never being an actual tuple of `R`) — this is a **lossy join**, since `{B,D}` isn't actually a valid join key here (`B=2` recurs across all three original rows with different `D` values, so joining on `B` alone reintroduces spurious combinations).

Denormalisation

Since anomalies come from redundancy, it's tempting to decompose as aggressively as possible — but heavily decomposed schemas need more `JOIN`s to answer queries. **Denormalisation** is the *deliberate*, controlled process of relaxing a normal form to improve query performance:

fewer joins (faster queries), fewer foreign keys (less storage/maintenance overhead) — useful for analytical workloads or when specific frequent queries need pre-joined results. It must be a controlled, deliberate trade-off, not an accident of poor initial design.

Summary

Three skills: testing a relation schema against 1NF/2NF/3NF/BCNF given a set of FDs, decomposing a universal relation into lossless-join anomaly-free BCNF relations (top-down), and computing a minimal cover to synthesise a lossless-join dependency-preserving 3NF schema (bottom-up). This completes Module 4. Next module: Database Security. See week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis[2] and week11-tutorial-case-study-9-payroll-system[3] for practice.

Reading: Elmasri & Navathe Chapters 14 (up to 14.6) and 15 (up to 15.5).

Applied Class 10: Normalisation, BCNF Decomposition and 3NF Synthesis

Practice for 2026-05-04-normalisation-and-relational-database-schema-design[4]. Note: "consider the FDs in the order provided" is a *default* guideline — only FDs that actually **violate** BCNF get split on; other correct decompositions can exist depending on split order.

Two things to have at hand before starting Section A:

- **Which FDs still apply to a fragment.** An FD `X → Y` holds in a decomposed relation `S` when `S` contains all of `X  Y` *and* `Y  X` — where `X` is computed against the **whole** original `F`, including attributes that aren't in `S` at all. The second half is what makes implicit FDs surface mid-algorithm, as in Question A.3.
- **Which child gets which attributes.** Splitting `Q` on a violating `X → Y` gives `Q1 [Q − Y]` and `Q2 [X  Y]`. `X` is a key of `Q2`, which is exactly what makes the split lossless. See decomposition[5].

Section A — BCNF decomposition (top-down)

Question A.1

`R [A, B, C, D, E]` with `{A} → {C}`, `{A, B} → {D, E}`. Decompose into BCNF.

[2] courses/infs1200/week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis.pdf
[3] courses/infs1200/week11-tutorial-case-study-9-payroll-system.pdf
[4] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[5] courses/infs1200/decomposition.pdf

Working

`{A,B}` = `{A,B,C,D,E}` — `{A, B}` is the only candidate key.
`{A}` → `{C}` violates BCNF (`A` isn't a superkey) → split into `R1 [A, C]` and `R2 [A, B, D, E]`. Within `R2`, `{A,B}` is a key (it already was for the whole relation), so `{A,B}` → `{D,E}` doesn't violate BCNF there — no further splitting needed.
Final answer: `R1 [A, C]` (`{A}`→`{C}`), `R2 [A, B, D, E]` (`{A,B}`→`{D,E}`).

Question A.2

`R [A, B, C, D, E]` with `{A}` → `{B}`, `{C}` → `{D, E}`. Decompose into BCNF.

Working

`{A,C}` = `{A,B,C,D,E}` — `{A, C}` is the only candidate key.
`{A}` → `{B}` violates BCNF (`A` alone isn't a superkey) → split into `R1 [A, B]` and `R2 [A, C, D, E]`. In `R2`, `{C}` → `{D, E}` violates BCNF (`C` alone isn't a superkey for `R2`) → split into `R3 [C, D, E]` and `R4 [A, C]`.
Final answer: `R1 [A, B]`, `R3 [C, D, E]`, `R4 [A, C]` (no non-trivial FD).

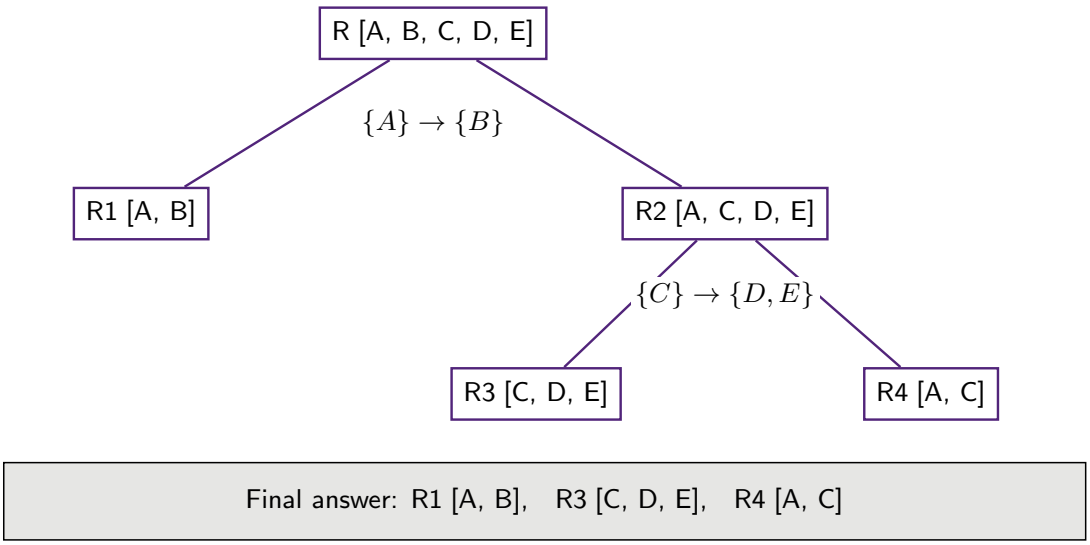
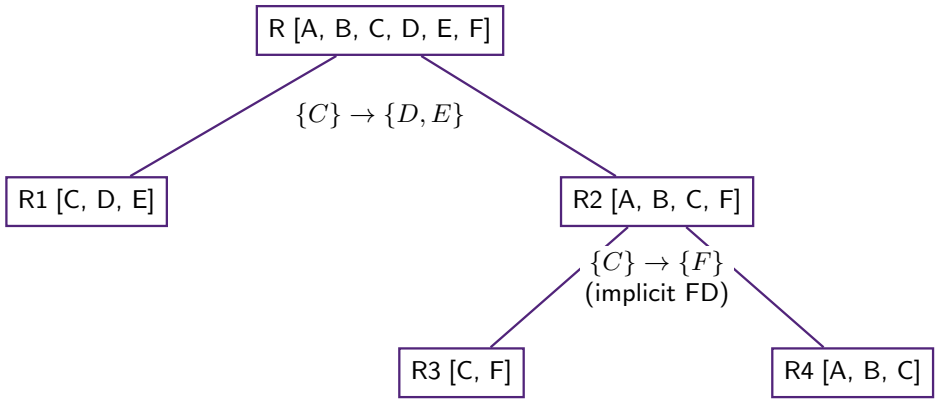


Figure 1: The decomposition tree for A.2 — and, as it turns out, for A.4 as well. Each split is labelled with the FD that violated BCNF; the leaves are the answer. Each split puts `X Y` on the left and `Q - Y` on the right. Nothing hangs on the order — it is the set of leaves that is the decomposition.

Question A.3

R [A, B, C, D, E, F] with {A} → {B, C}, {C} → {D, E}, {E} → {F}. Decompose into BCNF.

> **i** Working
>
> {A} = {A,B,C,D,E,F} — A is the only candidate key. A → {B, C} does **not** violate BCNF here, since A *is* the key — so we skip it and look for an FD whose LHS genuinely isn't a superkey.
> {C} → {D, E} violates BCNF (C isn't a superkey) → split into R1 [C, D, E] and R2 [A, B, C, F]. Within R2, the *explicit* FDs don't obviously violate BCNF — but the **implicit** FD C → F (derivable transitively via C → E and E → F) does (C isn't a superkey for R2 either) → split R2 into R3 [C, F] and R4 [A, B, C].
> **Final answer**: R1 [C, D, E] ({C}→{D,E}), R3 [C, F] ({C}→{F}, implicit), R4 [A, B, C] ({A}→{B,C}).
> Note: {E} → {F} is **lost** by this decomposition — E and F never end up together in any final relation, so that FD can no longer be checked directly against the decomposed schema.



$\{E\} \to \{F\}$ *is lost*

Final answer: R1 [C, D, E], R3 [C, F], R4 [A, B, C]

Figure 2: A.3's tree. The second split is driven by an FD that was never written down: `C → F` follows from `C → E` and `E → F`, and `C` is not a superkey of `R2`, so `R2` is not in BCNF even though none of the three given FDs says so. Miss it and you stop with `R2 [A, B, C, F]` wrongly declared finished. Each split puts `X  Y` on the left.

Question A.4

`R [A, B, C, D, E]` with `{A} → {B}`, `{C} → {D, E}`, `{A, D, E} → {C}`, `{B, C} → {A}`. Decompose into BCNF.

i Working

`{A, C}`, `{A, D, E}` and `{B, C}` are all candidate keys.
`{A} → {B}` violates BCNF → split into `R1 [A, B]` and `R2 [A, C, D, E]`. Within `R2`, `{C} → {D, E}` violates BCNF → split into `R3 [C, D, E]` and `R4 [A, C]`.
Final answer: `R1 [A, B]`, `R3 [C, D, E]`, `R4 [A, C]` — *the exact same result as Question A.2*, splitting on the same two FDs in the same order, so Figure 1 is A.4's tree unchanged, even though A.4 started with twice as many FDs. `{A, D, E} → {C}` and `{B, C} → {A}` are both **lost** in this decomposition (their attributes never all end up together in one final relation) — this is worth noticing: extra FDs don't necessarily survive decomposition, and a differently-ordered decomposition might have preserved different

(but not necessarily *more*) FDs.

Section B — 3NF synthesis (bottom-up)

Question B.1

`R [A, B, C, D, E, F, G, H]` with `{A, B} → {C}`, `{C, D} → {E}`, `{D} → {F, G}`. Synthesise into 3NF.

> **i** Working
>
> `{A,B,D,H} = {A,B,C,D,E,F,G,H}` — the only candidate key. Prime attributes: `A, B, D, H`. (`H` appears on no side of any FD, so it must be in every key; dropping any of `A`, `B` or `D` loses `C` or `E`.)
> This FD set is already a minimal cover. Step 1 splits `{D}→{F,G}` into `{D}→F` and `{D}→G`; step 2 finds nothing to drop from either compound LHS (`{A} = {A}` and `{B} = {B}` both miss `C`; `{C} = {C}` and `{D} = {D,F,G}` both miss `E`); step 3 finds no FD implied by the others.
> **Synthesis**: `R1 [A, B, C]` (`{A,B}→C`), `R2 [C, D, E]` (`{C,D}→E`), `R3 [D, F, G]` (`{D}→{F,G}`), and — since the candidate key `{A,B,D,H}` isn't contained in any of these — `R4 [A, B, D, H]` (no non-trivial FD). All four are in 3NF.

Question B.2

`R [A, B, C, D, E, F]` with `{A, B} → {C}`, `{C} → {D, E}`. Synthesise into 3NF.

> **i** Working
>
> `{A,B,F} = {A,B,C,D,E,F}` — the only candidate key. Prime attributes: `A, B, F`. Already a minimal cover.
> **Synthesis**: `R1 [A, B, C]` (`{A,B}→C`), `R2 [C, D, E]` (`{C}→{D,E}`), `R3 [A, B, F]` (no non-trivial FD, added since the key isn't otherwise present). All three are in 3NF.

Question B.3

`R [A, B, C, D, E]` with `{A} → {D, E}`, `{D} → {A}`, `{B} → {C}`, `{B, C} → {A, D}`, `{E, A} → {D}`. Synthesise into 3NF.

Working

{B} = {A,B,C,D,E} — {B} is the only candidate key. Prime attributes: B.
Minimal cover, step by step:

Step 1: RHS split	Step 2: LHS simplify	Step 3: delete redundant
{A} → D	{A} → D	{A} → D
{A} → E	{A} → E	{A} → E
{B,C} → A	{B} → A — C redundant, since B → C	{B} → A
{B,C} → D	{B} → D — same reason	*dropped* — {B} → A then {A} → D
{D} → A	{D} → A	{D} → A
{E,A} → D	{A} → D — E redundant, A alone reaches D	*dropped* — now a duplicate of {A} → D
{B} → C	{B} → C	{B} → C

leaving {A} → {D, E}, {B} → {A, C}, {D} → {A}.
Synthesis: one relation per FD gives [A, D, E], [A, B, C] and [A, D]; the last is a subset of the first, so it is eliminated and {D} → {A} rides along inside R1. The candidate key {B} is already present in R2, so no extra all-prime-attributes relation is needed.
R1 [A, D, E] ({A}→{D,E}, {D}→{A}), R2 [A, B, C] ({B}→{A,C}). Both are in 3NF.

Question B.4

R [A, B, C, D, E, F] with {A} → {B, C, D, E, F}, {B, C} → {A}, {D, E} → {B}, {C} → {D}. Synthesise into 3NF.

Working

{A}, {B, C} and {C, E} are all candidate keys. Prime attributes: A, B, C, E.
Minimal cover, step by step. Only step 1 and step 3 do anything: neither compound LHS can be shortened ({B} = {B} and {C} = {C,D} both miss A; {D} = {D} and {E} = {E} both miss B).

Step 1: RHS split	Step 2: LHS simplify	Step 3: delete redundant
{A} → B	{A} → B	*dropped* — A → D, A → E, then {D,E} → B
{A} → C	{A} → C	{A} → C

{A} → D	{A} → D	*dropped* — A → C, then {C} → D
{A} → E	{A} → E	{A} → E
{A} → F	{A} → F	{A} → F
{B,C} → A	{B,C} → A	{B,C} → A
{D,E} → B	{D,E} → B	{D,E} → B
{C} → D	{C} → D	{C} → D

leaving {A} → {C, E, F}, {B, C} → {A}, {D, E} → {B}, {C} → {D}. Neither removal is visible from a single FD — both take the closure test, and both survive in either order. **Synthesis**: R1 [A, C, E, F] ({A}→{C,E,F}), R2 [B, C, A] ({B,C}→{A}), R3 [D, E, B] ({D,E}→{B}), R4 [C, D] ({C}→{D}). The candidate key {A} is already present in R1, so no extra relation is needed. All four are in 3NF.

Case Study 9: Payroll System (BCNF Decomposition and 3NF Synthesis)

Practice for 2026-05-04-normalisation-and-relational-database-schema-design[6]. Continues week10-tutorial-case-study-8-dirt-road-driving[7] — Peter has now resent Dirt Road Driving's payroll schema *with* the full set of functional dependencies included (his earlier email accidentally omitted them).

The corrected payroll schema

```
Employee [id, firstName, lastName, role]

Project [name, description, funding, projectLeader]
Project.projectLeader references Employee.id

TimeLog [employeeID, projectName, date, hoursWorked, approved]
TimeLog.employeeID references Employee.id
TimeLog.projectName references Project.name

AssetUse [employeeID, assetID, timestamp, useDuration, assetType, purchaseDate, insuranceValu
assetID → assetType, purchaseDate
assetType → insuranceValue
AssetUse.employeeID references Employee.id
```

[6] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[7] courses/infs1200/week10-tutorial-case-study-8-dirt-road-driving.pdf

```
Department [code, name, manager, buildingID, buildingName, buildingLocation, floor]
buildingID → buildingLocation, buildingName
buildingName → buildingLocation, buildingID
Department.manager references Employee.id

EmployeeHistory [employeeID, departmentCode, seniorityLevel, baseSalary, securityLevel]
seniorityLevel, securityLevel → baseSalary
EmployeeHistory.employeeID references Employee.id
EmployeeHistory.departmentCode references Department.code

TripExpenseAllocations [tripName, expenseType, quantity, organiser, startDate, endDate, loca
tripName, expenseType → quantity, description
tripName → startDate, endDate, location, organiser
expenseType → allowance, restrictions
TripExpenseAllocations.organiser references Employee.id

TravelInsuranceHistory [tripName, approved, insuranceLevel, description, maxCoverage, advise
tripName → approved, insuranceLevel, description
insuranceLevel → description, maxCoverage
TravelInsuranceHistory.tripName references TripExpenseAllocations.tripName
```

Section A — BCNF decomposition

Decompose (if needed) ***AssetUse***, ***Department***, ***EmployeeHistory***, ***TripExpenseAllocations***, *and* ***TravelInsuranceHistory*** *into BCNF, clearly stating any new tables and all foreign keys.*

AssetUse

Working

CK = {employeeID, assetID, timestamp}. FDs: {employeeID, assetID, timestamp} → {useDuration, assetType, purchaseDate, insuranceValue}, {assetID} → {assetType, purchaseDate}, {assetType} → {insuranceValue}, and the implicit {assetID} → {insuranceValue}. Highest NF: **1NF**.
Final answer:

- Asset [assetID, assetType, purchaseDate] — {assetID} → {assetType, purchaseDate}
- Insurance [assetID, insuranceValue] — {assetID} → {insuranceValue}, FK assetID references Asset.assetID

- `AssetUse [employeeID, assetID, timestamp, useDuration]` — no non-trivial FD, FK `assetID` references `Asset.assetID`

Lost: `{assetType} → {insuranceValue}` (once decomposed, `assetType` and `insuranceValue` never appear together in one relation, so this FD can no longer be directly enforced/checked).

Department

i Working

CK = `{code}`. FDs: `{code} → {name, manager, buildingID, buildingName, location, floor}`, `{buildingID} → {buildingLocation, buildingName}`, `{buildingName} → {buildingLocation, buildingID}`. Highest NF: **2NF**.
Final answer:

- `Building [buildingID, buildingLocation, buildingName]` — `{buildingID} → {buildingLocation, buildingName}`, `{buildingName} → {buildingID, buildingLocation}`
- `Department [code, name, manager, buildingID, floor]` — `{code} → {name, manager, buildingID, floor}`, FK `buildingID` references `Building.buildingID`

EmployeeHistory

i Working

CK = `{employeeID, departmentCode}`. FDs: `{employeeID, departmentCode} → {seniorityLevel, baseSalary, securityLevel}`, `{seniorityLevel, securityLevel} → {baseSalary}`. Highest NF: **2NF**.
Final answer:

- `Salary [seniorityLevel, securityLevel, baseSalary]` — `{seniorityLevel, securityLevel} → {baseSalary}`
- `EmployeeHistory [employeeID, departmentCode, seniorityLevel, securityLevel]` — `{employeeID, departmentCode} → {seniorityLevel, securityLevel}`, FKs to `Employee.id` and `Department.code`

TripExpenseAllocations

Working

CK = {tripName, expenseType}. FDs: {tripName, expenseType, quantity} → {organiser, startDate, endDate, location, allowance, restrictions, description}, {tripName, expenseType} → {quantity, description}, {tripName} → {startDate, endDate, location, organiser}, {expenseType} → {allowance, restrictions}. Highest NF: **1NF**.
Final answer:

Relation	FD
Trip [tripName, startDate, endDate, location, organiser]	{tripName} → {startDate, endDate, location, organiser}
Expense [expenseType, allowance, restrictions]	{expenseType} → {allowance, restrictions}
TripExpenseAllocations [tripName, expenseType, quantity, description]	{tripName, expenseType} → {quantity, description}, FKs to Trip.tripName and Expense.expenseType

TravelInsuranceHistory

Working

CK = {tripName} (the schema's stated primary key {tripName, approved, insuranceLevel} is **not minimal** — tripName alone is already a key). FDs: {tripName, approved, insuranceLevel} → {description, maxCoverage, advisedPrecautions}, {tripName} → {approved, insuranceLevel, description}, {insuranceLevel} → {description, maxCoverage}. Highest NF: **2NF**.
Final answer:

- InsuranceCoverage [insuranceLevel, description, maxCoverage] — {insuranceLevel} → {description, maxCoverage}
- TravelInsuranceHistory [tripName, approved, insuranceLevel, advisedPrecautions] — {tripName} → {approved, insuranceLevel}, {tripName, approved, insuranceLevel} → {advisedPrecautions}, FK insuranceLevel references InsuranceCoverage.insuranceLevel

Section B — 3NF synthesis

Decompose (if needed) `Department` *and* `TripExpenseAllocations` *into 3NF via minimal-cover synthesis.*

Department

> **Working**
>
> CK = `{code}`. Highest NF (from Section A): 2NF.
> **Minimal cover**: RHS-splitting `{code} → {name, manager, buildingID, buildingName, buildingLocation, floor}` and removing redundant attributes — `code → buildingName` and `code → buildingLocation` are both redundant (derivable transitively via `code → buildingID` then `buildingID → buildingName`/`buildingLocation`); `buildingID → buildingLocation` is redundant (derivable via `buildingID → buildingName` then `buildingName → buildingLocation`). Simplified cover: `{code} → {name, manager, buildingID, floor}`, `{buildingID} → {buildingName}`, `{buildingName} → {buildingLocation, buildingID}`.
> **Synthesis**:
>
> - `Department [code, name, manager, buildingID, floor]` — `{code} → {name, manager, buildingID, floor}`
> - `Building [buildingName, buildingLocation, buildingID]` — `{buildingID} → {buildingName}`, `{buildingName} → {buildingLocation, buildingID}` (a candidate relation `[buildingID, buildingName]` from `{buildingID}→{buildingName}` alone is redundant — it's a subset of this `Building` relation, so it's dropped.)

TripExpenseAllocations

> **Working**
>
> CK = `{tripName, expenseType}` (the stated primary key is not minimal). Highest NF (from Section A): 1NF.
> **Minimal cover**: after RHS-splitting and removing every attribute already implied by `{tripName}` or `{expenseType}` alone from the `{tripName, expenseType, quantity} → ...` FD (all seven of its RHS attributes turn out to be redundant there, since each is already determined by `tripName` or `expenseType` individually), the simplified cover is exactly: `{tripName, expenseType} → {quantity, description}`, `{tripName} → {startDate, endDate, location, organiser}`, `{expenseType} → {allowance,`

restrictions}.
Synthesis:

- TripExpenseAllocations [tripName, expenseType, quantity, description] — {tripName, expenseType} → {quantity, description}
- Trip [tripName, startDate, endDate, location, organiser] — {tripName} → {startDate, endDate, location, organiser}
- Expense [expenseType, allowance, restrictions] — {expenseType} → {allowance, restrictions}

(Same final relations as the BCNF decomposition in Section A — in this case, decomposition and synthesis happen to converge on the same schema.)

Reference material

3NF Synthesis

3NF synthesis is the bottom-up route from a universal relation to a good schema: instead of repeatedly splitting one big relation apart (see decomposition[8]), it *builds* relations up from a minimal-cover[9] — one relation per dependency. The result is guaranteed lossless-join, **dependency preserving**, and in 3NF[10]. Introduced in 2026-05-04-normalisation-and-relational-database-schema-design[11].

Dependency preservation is the reason to choose synthesis over BCNF decomposition: because every FD in the cover gets its own relation, every FD ends up checkable against a single relation. The price is that 3NF tolerates the residual anomalies BCNF would have removed.

The algorithm

```
S := ;
Compute a minimal cover G of F;
Combine all FDs in G with the same LHS into one;
For each X → Y in G:
    if no relation in S contains X  Y:
        add a relation with schema X  Y to S;
if any candidate key is missing from the relations:
```

[8] courses/infs1200/decomposition.pdf
[9] courses/infs1200/minimal-cover.pdf
[10] courses/infs1200/normal-forms.pdf
[11] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf

```
    add a relation with all prime attributes (i.e. all candidate keys);
Eliminate redundant relations from the resulting set of relations;
```

A relation is **redundant** if it is a *projection of* another relation already in the schema — that is, if its attributes are a subset of another's.

Reading it step by step:

Step	What it does	Why
Minimal cover	Strips redundant FDs and redundant LHS attributes	Redundant FDs would create redundant relations
Combine same LHS	`X → A` and `X → B` become `X → {A, B}`	Otherwise you'd get two relations that both key on `X`
One relation per FD	Schema `X  Y`; `X` is its key	Puts each dependency somewhere it can be enforced — this is what preserves dependencies
Add the key relation	If no relation contains a full candidate key, add one holding all prime attributes	Without it the decomposition need not be lossless — nothing joins the pieces back to the original key
Eliminate redundant relations	Drop any relation that is a *projection of* another — i.e. whose attributes are a subset of another's	A subset relation stores nothing new

The key relation step, precisely

This is the step that trips people up, so it's worth stating exactly.

When it fires. Check whether any relation already in `S` contains a full candidate key of `R` among its attributes. If one does, skip the step entirely. If none does, fire it. Note that the FD-derived relations often *look* key-ish — each has its own LHS as a local key — without any of them containing a candidate key of the original `R`.

What gets added. A relation containing **all the prime attributes** — the union of *every* candidate key of `R`, not one chosen candidate key. When `R` has exactly one candidate key the two readings coincide, which is why the distinction is easy to miss; when it has several, they differ and only the union is correct.

Why. The FD-derived relations are chained together by their shared LHS attributes, but nothing guarantees those chains reach the original key. The prime-attribute relation is what

makes the join reconstruct R — it is a losslessness patch, not an FD-carrying relation, and it usually carries no non-trivial FD at all.

Then re-run elimination. The prime-attribute relation is typically wide, so relations added earlier can become projections of it. Eliminate redundant relations **after** adding it, never before.

Worked example — the straightforward shape

R [A, B, C, D, E] with F = {{A,B} → C, C → D} (already a minimal cover). A, B and E never appear on a RHS, so every key contains them; {A,B,E} = {A,B,C,D,E}, so {A, B, E} is the only candidate key.

1. One relation per FD: R1 [A, B, C] (from {A,B} → C) and R2 [C, D] (from C → D).
2. Neither contains the candidate key {A, B, E} — E appears nowhere — so add R3 [A, B, E].
3. No relation is a subset of another.

Final: R1 [A, B, C], R2 [C, D], R3 [A, B, E].

R3 carries no non-trivial FD; its job is purely to hold the key together so the join is lossless. A relation like that is a normal, and expected, output of synthesis.

Worked example — same-LHS combination and subset elimination

R [A, B, C, D, E] with {A} → {D,E}, {D} → {A}, {B} → {C}, {B,C} → {A,D}, {E,A} → {D} (from week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis[12]).

{B} = {A,B,C,D,E}, and B never appears on a RHS, so {B} is the only candidate key.

Minimal cover: after RHS-splitting, {E,A} → D simplifies to {A} → D (E is redundant — A alone already reaches D) and the resulting duplicate collapses into the existing {A} → {D,E}; {B,C} → A and {B,C} → D both lose their C (redundant, since B → C), becoming {B} → A and {B} → D; and {B} → D is then dropped outright, being derivable via {B} → A then {A} → D. What remains is {A} → {D, E}, {B} → {A, C}, {D} → {A}.

Synthesis: one relation per FD gives [A, D, E], [A, B, C] and [A, D]. The last is a subset of [A, D, E], so it goes. The candidate key {B} is already inside [A, B, C], so no extra relation is needed.

Final: R1 [A, D, E] (carrying {A} → {D,E} and {D} → {A}), R2 [A, B, C] (carrying {B} → {A,C}). Both are in 3NF.

[12] courses/infs1200/week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis.pdf

Worked example — when the key relation swallows everything

Given the minimal cover `{A,C} → E`, `{B,D} → A`, `A → B`, `E → {C, F}` for `R [A, B, C, D, E, F]`. The candidate keys are `{A,C,D}`, `{A,D,E}`, `{B,C,D}` and `{B,D,E}` — note `F` appears in none of them, so the prime attributes are `{A, B, C, D, E}`.

1. One relation per FD: `[A, C, E]`, `[B, D, A]`, `[A, B]`, `[E, C, F]`.
2. `[A, B]` is a subset of `[B, D, A]` — drop it.
3. No remaining relation contains a *full* candidate key: every key contains `D`, but `[A,C,E]` and `[E,C,F]` lack `D`, and `[A,B,D]` lacks both `C` and `E`. So add `R5` with all prime attributes: `[A, B, C, D, E]`.
4. Eliminate subsets again: `[A, C, E]` and `[A, B, D]` are now both subsets of `R5` — drop both. `[E, C, F]` is not (it has `F`) — keep.

Final: `[E, C, F]` and `R5 [A, B, C, D, E]`.

This is the case that catches people out. Skipping the second subset-elimination pass leaves three redundant relations behind; skipping step 3 leaves a decomposition that isn't lossless.

Synthesis vs decomposition

	BCNF decomposition (top-down)	3NF synthesis (bottom-up)
Input	`R` + `F`	A **minimal cover** of `F`
Method	Split on each violating FD	One relation per FD, plus a key relation
Normal form reached	BCNF	3NF
Lossless join	✓	✓
Dependency preserving	✗ (not guaranteed)	✓
Anomaly-free	✓	Mostly — 3NF's prime-attribute clause can let some through

Neither is deterministic — synthesis depends on which minimal cover you land on, decomposition on the order violating FDs are processed — and both inherit any error in the FD set: a missing FD produces a sub-optimal or incorrect design either way.

The two often converge. In week11-tutorial-case-study-9-payroll-system[13], `TripExpenseAllocations` decomposes to the same three relations — `Trip`, `Expense`, `TripExpenseAllocations` — whether you take the BCNF or the synthesis route. When they differ, the difference is exactly the FDs BCNF had to sacrifice.

[13] courses/infs1200/week11-tutorial-case-study-9-payroll-system.pdf

See also

- minimal-cover[14] — compute this first; synthesis is meaningless without it
- decomposition[15] · normal-forms[16] · attribute-closure[17] · functional-dependencies[18]
- 2026-05-04-normalisation-and-relational-database-schema-design[19] — Module 4, part 2
- week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis[20] — Section B is pure synthesis drill

Decomposition and BCNF Design

A **decomposition** of a relation `R` replaces it with two or more relations such that (1) each new relation's attributes are a subset of `R`'s, and (2) every attribute of `R` appears in at least one of them. That's a low bar — it says nothing about whether the pieces can be put back together, or whether the original constraints survive. This note covers the two properties that separate a good decomposition from a destructive one, and the top-down algorithm that produces BCNF relations. Introduced in 2026-04-27-database-design-guidelines-and-functional-dependencies[21] and 2026-05-04-normalisation-and-relational-database-schema-design[22].

The two properties

Property	Question it answers	Guaranteed by
Lossless join (non-additive join)	Can `R` be reconstructed exactly by rejoining the pieces?	BCNF decomposition **and** 3NF synthesis
Dependency preservation	Can every original FD still be checked against a single relation?	3NF synthesis only

Lossless join is non-negotiable — a decomposition that fails it has corrupted the data. Dependency preservation is desirable but sometimes unattainable in BCNF, which is the whole reason 3NF remains in use.

[14] courses/infs1200/minimal-cover.pdf
[15] courses/infs1200/decomposition.pdf
[16] courses/infs1200/normal-forms.pdf
[17] courses/infs1200/attribute-closure.pdf
[18] courses/infs1200/functional-dependencies.pdf
[19] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[20] courses/infs1200/week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis.pdf
[21] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[22] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf

Lossless join and spurious tuples

The natural join `R1  R2` concatenates each tuple of `R1` with every tuple of `R2` that agrees on their common attributes (see sql-joins[23]). A decomposition of `R` into `R1` and `R2` is a **lossless join decomposition** if, for every legal instance,

$$R = R_1 \bowtie R_2$$

— breaking `R` apart and rejoining gives back exactly `R`, no more and no less.

A lossy decomposition — split `R [A, B, C]` into `R1 [A, B]` and `R2 [B, C]` when `B` does *not* determine `C`:

```
R                    R1        R2          R1   R2 (rejoined)
A  B  C              A  B      B  C        A  B  C
1  2  3              1  2      2  3        1  2  3
4  5  6      →       4  5   +  5  6   →    1  2  9   <- spurious!
7  2  9              7  2      2  9        4  5  6
                                           7  2  3   <- spurious!
                                           7  2  9
```

"Loss" here means loss of *information*, not loss of tuples — the rejoined relation has *more* rows than the original. A better name for the failure would be "addition of spurious information". Because `B = 2` maps to two different `C` values, the join can't tell which `A` belongs with which `C`, and invents both combinations. Those extra rows are the **spurious tuples** that guideline 4 of update-anomalies[24] warns about.

Note what would have saved it: if `B → C` held, each `B` value would match exactly one `R2` row, no `A` could pair with two `C`s, and the join would return exactly `R`. That is the whole idea — a split is safe when the attributes the two pieces *share* functionally determine one of them.

Testing for a lossless join

There is no general algebraic test in this course — no tableau or chase procedure, and no separately named binary-decomposition rule to quote. Two things stand in for it, and between them they cover everything the course asks:

1. The structural guarantee, when you built the decomposition yourself. The BCNF algorithm below always splits `Q` on an FD `X → Y` into `Q1 [Q - Y]` and `Q2 [X  Y]`. The two pieces share exactly `X`, and `X` is a key of `Q2` — so every `Q1` tuple matches at most one `Q2` tuple,

[23] courses/infs1200/sql-joins.pdf
[24] courses/infs1200/update-anomalies.pdf

the join can never multiply rows, and the result is lossless by construction. This is also the answer to "justify why lossless-join decompositions are preferred": they are the only ones that don't silently fabricate data.

2. The counter-example test, when you're handed a decomposition and asked whether it works. Build a small instance of R that satisfies F, project it onto the proposed pieces, natural-join them back, and look for a row that wasn't in R. A single spurious tuple settles the question; no exhaustive argument is needed.

Worked example — R [A, B, C, D] with A → B, C → D, {A,D} → C, {B,C} → A. Is {{A,B}, {A,C}, {B,D}} a lossless-join decomposition?

```
R                    A  B      A  C      B  D
A  B  C  D           1  2      1  5      2  6
1  2  5  6     →     8  2      1  3      2  7
1  2  3  7                     8  9      2  4
8  2  9  4
```

Rejoining [A,B] [A,C] [B,D] gives nine rows, including (1, 2, 3, 4) — never in R. The decomposition is **lossy**. The reason is visible without the instance: [A,B] and [B,D] meet only on B, and B determines neither piece, so every A gets paired with every D. Replacing [B,D] with [C,D] fixes it — the pieces then chain on A and on C, and each of those *is* a key of the piece it determines.

Which FDs hold in a decomposed relation

To test a piece for a normal form, or to check dependency preservation, you first need the FDs that apply to it. For a decomposed relation S and an FD X → Y from the original F:

> X → Y holds in S if X Y S's attributes **and** Y X , where X is computed against *all* of F — including attributes that aren't in S.

The second condition is the one that's easy to get wrong: attributes outside S may still be used as *intermediates* in the derivation.

Worked example — R [A, B, C, D, E] with {A,B} → C, {B,C} → D, {C,D} → E, {D,E} → A, {A,E} → B. Which FDs hold in S [A, B, C, D]?

{B,C,D} → A does. Computing {B,C,D} against the full F: {B,C} → D adds nothing new; {C,D} → E adds E (an attribute *not in S*); {D,E} → A then adds A. So A {B,C,D} , and both sides sit inside S — the FD holds. By contrast {A,B} → E and {A,E} → B both fail the *first* condition: E isn't an attribute of S at all.

Dependency preservation

When `R` is decomposed into `R1, …, Rn`, the FD set `F` is decomposed alongside it into `F1, …, Fn`, where **`Fi` contains all the FDs in `F` whose attributes fall entirely within `Ri`**. The decomposition is **dependency preserving** if

$$(F_1 \cup \cdots \cup F_n)^+ = F^+$$

— no FD information is lost, so every original constraint can still be enforced by checking a single relation, with no join required.

Testing it

The set-closure equality is never computed directly. One direction is free (every `Fi` `F`, so `( Fi)` `F` always), which leaves a finite check on the other:

1. For each piece `Ri`, write down `Fi` — the FDs of `F` with **both** sides entirely inside `Ri`. An FD that straddles two pieces belongs to neither.
2. Union them into `G = F1` … `Fn`.
3. For each FD `X → Y` in the original `F`, compute `X` **using only `G`**. If `Y` `X` for every one of them, `G = F` and the decomposition is dependency preserving. The first FD that fails is the one that was lost.

Step 3 matters because an FD absent from every `Fi` may still be *recoverable* by chaining the ones that survived — only an FD that `G` cannot derive at all is genuinely lost.

Two minimal examples — `R [A, B, C, D]` with `F = {A → B, A → D}`:

Decomposition	F1	F2	Verdict
`R1 [A, B, C]`, `R2 [B, C, D]`	`{A → B}`		`A → D` is **not preserved** — `A` and `D` never appear together
`R1 [A, B, C]`, `R2 [A, B, D]`	`{A → B}`	`{A → D}`	**All FDs preserved**

The difference is only where `A` was placed. Dragging the LHS along into the second piece is what keeps its FD checkable, and it is the instinct 3nf-synthesis[25] turns into an algorithm: give every FD in the cover its own relation `X` `Y`, and preservation is automatic.

[25] courses/infs1200/3nf-synthesis.pdf

A BCNF decomposition that loses one — `Teach [studentID, courseID, lecturer]` with `{studentID, courseID} → lecturer` and `lecturer → courseID`. The second FD violates BCNF, so split into `[lecturer, courseID]` and `[studentID, lecturer]`. Each piece is now in BCNF and neither violates any FD locally — but rejoin them:

```
lecturer  courseID          studentID  lecturer
John      INFS1200          1234       John
Jane      INFS1200          1234       Jane
                ↓ join ↓
studentID  courseID  lecturer
1234       INFS1200  John
1234       INFS1200  Jane
```

The result violates the original `{studentID, courseID} → lecturer`: two rows share `{1234, INFS1200}` but disagree on `lecturer`. The FD survived nowhere, so nothing was there to reject the insert.

BCNF vs 3NF:

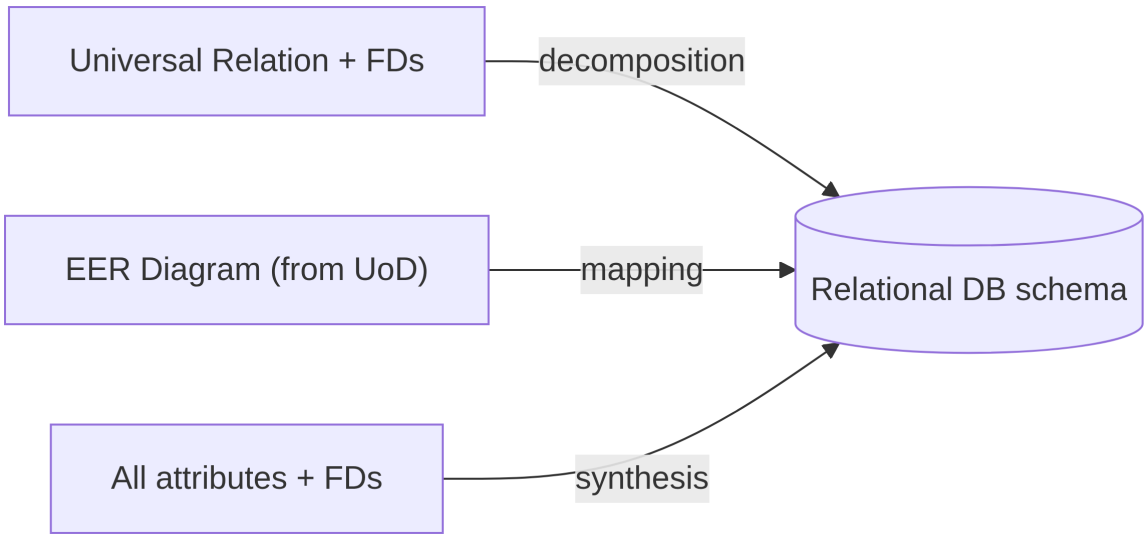
	BCNF	3NF
Removes all anomalies		
Preserves all FDs		

Organisations pick one or the other depending on which guarantee matters more for the workload.

The design landscape

Two complementary algorithms turn a **universal relation** (all attributes lumped together, plus a set of FDs) into a well-designed multi-relation schema — the third route being to derive the schema from an EER diagram in the first place (see er-to-relational-mapping[26]):

[26] courses/infs1200/er-to-relational-mapping.pdf



- **Decomposition** (top-down) — *design by analysis*. Start from the universal relation and its FDs and break it apart until every piece is in BCNF. Lossless-join and anomaly-free; may lose FDs. Covered below.
- **Synthesis** (bottom-up) — *design by synthesis*. Start from the individual attributes and the FDs, and build relations up from a minimal-cover[27]. Lossless-join, dependency-preserving and 3NF. See 3nf-synthesis[28].

Both approaches are **non-deterministic** — the answer depends on choices made along the way — and both are only as good as their input: a missing FD leads to a sub-optimal or outright incorrect design, which is why the FDs have to come from the Universe of Discourse rather than from a sample instance (see functional-dependencies[29]).

The BCNF decomposition algorithm

```
D := {R};
while (some relation Q in D is not in BCNF):
    find an FD X → Y in Q that violates BCNF;
    replace Q in D with Q1 [Q - Y] and Q2 [X   Y];
```

Why it works. After the split, `X → Y` can no longer violate BCNF in either piece: `Q1` doesn't contain `Y` any more, so the FD isn't there to test; and in `Q2`, `X` *is* a key, so the FD passes. Do that to every offending FD in turn and every violation is eventually fixed. The algorithm

[27] courses/infs1200/minimal-cover.pdf
[28] courses/infs1200/3nf-synthesis.pdf
[29] courses/infs1200/functional-dependencies.pdf

terminates because every two-attribute relation `[a, b]` is automatically in BCNF — with no FD there's no redundancy, with `a → b` then `a` is a key, with `b → a` then `b` is, and with both, both are. `Q2` is also what makes the result lossless: `X` keys it, so the rejoin on `X` can never multiply rows.

Results can vary depending on the order violating FDs are processed in. That's fine: several correct decompositions can exist for the same relation, and they may preserve different sets of FDs.

Worked example — `R [A, B, C, D]` with `B → C` and `D → A`. `{B,D} = {A,B,C,D}`, and it's the only key.

1. `B → C` violates BCNF (`B` isn't a superkey) → replace `R` with `R1 [A, B, D]` and `R2 [B, C]`.
2. `D → A` now violates BCNF in `R1` (`D` isn't a key for `R1`) → replace it with `R3 [B, D]` and `R4 [A, D]`.

Final: `R2 [B, C]`, `R3 [B, D]`, `R4 [A, D]`.

Skip FDs that don't violate — `R [A, B, C, D, E, F]` with `A → {B,C}`, `C → {D,E}`, `E → F`. Here `A` is the only candidate key, so `A → {B, C}` does **not** violate BCNF and must be left alone; the first genuine violation is `C → {D, E}`.

Implicit FDs matter

When testing a relation for BCNF you must consider *implied* FDs too, not only the explicitly listed ones. The reliable habit is to make them explicit before you start: compute every closure in step 1, and whenever `F  X → Y` for some `X → Y` not written down, **add it to `F`** and treat it like any other candidate violation.

Worked example — `R [A, B, C, D, E, F]` with `A → B`, `{D,E} → F`, `B → C` (the attribute called `F` is unrelated to the FD set `F`). Closures: `{A} = {A,B,C}`, `{B} = {B,C}`, `{D,E} = {D,E,F}`; the only key is `{A,D,E}`. Since `F  A → C`, that FD joins the set. Then:

1. `A → B` violates → `R1 [A, C, D, E, F]`, `R2 [A, B]`
2. `{D,E} → F` violates in `R1` → `R3 [A, C, D, E]`, `R4 [D, E, F]`
3. `A → C` violates in `R3` → `R5 [A, D, E]`, `R6 [A, C]`

Final: `R2 [A, B]`, `R4 [D, E, F]`, `R5 [A, D, E]`, `R6 [A, C]`. Miss the implied `A → C` and you stop one step early with `R3` wrongly declared BCNF.

The same trap appears mid-algorithm. Returning to the `A → {B,C}`, `C → {D,E}`, `E → F` relation above: splitting on `C → {D, E}` gives `[C, D, E]` and `[A, B, C, F]`. No explicit FD obviously violates BCNF in `[A, B, C, F]` — but the implicit `C → F` (from `C → E` and `E → F`) does, since `C` isn't a superkey of it. Splitting again gives `[C, F]` and `[A, B, C]`.

Final: `[C, D, E]`, `[C, F]`, `[A, B, C]` — and note `E → F` is now **lost**, since `E` and `F` never appear together in any final relation.

Common mistakes

From week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis[30] and the decomposition-tree question in 2026-05-04-normalisation-and-relational-database-schema-design[31]:

- **Dropping part of the violating FD's RHS** — splitting on `C → {D, E}` produces `[C, D, E]`, not `[C, E]`.
- **Removing the wrong attributes from the remainder** — `Q1` is `Q - Y`, so only the RHS leaves. Splitting `R [A,B,C,D,E]` on `{A,D} → B` leaves `[A, C, D, E]`, not `[A, B, C, E]`.
- **Splitting on an FD that no longer violates anything** — once `A` is a key of the sub-relation you're looking at, `A → E` is not a violation and must not be split on.
- **Not checking the result is lossless.** For `R [A, B, C, D]` with `A → B`, `C → D`, `{A,D} → C`, `{B,C} → A` (keys `{A,D}`, `{A,C}`, `{B,C}`), running the algorithm on `A → B` then `C → D` gives the correct `{A,B}`, `{A,C}`, `{C,D}`. The plausible-looking `{A,B}`, `{A,C}`, `{B,D}` is **lossy** — see the counter-example test above.

Extra FDs don't guarantee extra survivors, either. On `R [A, B, C, D, E]`, the FD set `{A → B, C → {D,E}}` and the larger set `{A → B, C → {D,E}, {A,D,E} → C, {B,C} → A}` decompose to *exactly* the same three relations `[A,B]`, `[C,D,E]`, `[A,C]` — the two extra FDs in the second case are simply lost, since their attributes never all end up together in one final relation.

See also

- normal-forms[32] · 3nf-synthesis[33] · minimal-cover[34] · attribute-closure[35]
- update-anomalies[36] — guideline 4 and the informal view of spurious tuples
- 2026-05-04-normalisation-and-relational-database-schema-design[37] — Module 4, part 2
- week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis[38] · week11-tutorial-case-study-9-payroll-system[39] — practice

[30] courses/infs1200/week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis.pdf
[31] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[32] courses/infs1200/normal-forms.pdf
[33] courses/infs1200/3nf-synthesis.pdf
[34] courses/infs1200/minimal-cover.pdf
[35] courses/infs1200/attribute-closure.pdf
[36] courses/infs1200/update-anomalies.pdf
[37] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[38] courses/infs1200/week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis.pdf
[39] courses/infs1200/week11-tutorial-case-study-9-payroll-system.pdf

Minimal Cover

A **minimal cover** (or canonical cover) `G` of an FD set `F` is an equivalent set of FDs with all the slack squeezed out — nothing can be deleted or shortened without changing what the set implies. It's the required input to 3nf-synthesis[40], and it makes reasoning about BCNF[41] easier by hand too. Introduced in 2026-05-04-normalisation-and-relational-database-schema-design[42].

Equivalence of FD sets

Two FD sets are **equivalent** when they imply exactly the same FDs: `F = G`. Equivalence is what makes the reduction safe — a minimal cover constrains the data identically to the original set, it just says so with fewer, smaller rules.

Definition

The goal is to **transform the FDs to be as small as possible** while still implying exactly the same things.

`G` is a minimal cover for `F` iff:

1. **Equivalent** — `F = G`.
2. **Nothing redundant** — deleting any FD from `G`, or any attribute from any FD's LHS, gives some `G'` with `G' G`. Every FD and every LHS attribute is *necessary*.

Informally: every FD in `G` is needed, and is as small as possible, while still closing to the same set as `F`.

The smallest illustration: `F = {A → B, B → C, A → {B, C}}` reduces to `G = {A → B, B → C}` — the third FD asserts nothing the first two don't already imply.

The textbook definition adds a third condition — **single-attribute RHS**, every FD in `G` having exactly one attribute on its right — but this course explicitly does **not require** it. Splitting the RHS is still the first working step, because condition 2's tests are only correct per-attribute; but the answer may then be written back up with same-LHS FDs recombined (`{E,F} → G` and `{E,F} → H` as `{E,F} → {G, H}`). Both presentations are accepted, and 3nf-synthesis[43] recombines them anyway as its second step.

[40] courses/infs1200/3nf-synthesis.pdf
[41] courses/infs1200/normal-forms.pdf
[42] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[43] courses/infs1200/3nf-synthesis.pdf

Finding a minimal cover — three steps

The steps run in this order, and each is decided by an attribute-closure[44] computation:

Step	What to do	Test
1. RHS simplification	Split every FD so its RHS is a single attribute: `X → {Y, Z}` becomes `X → Y` and `X → Z`	— (always applies)
2. LHS simplification	For an FD `X → A` whose LHS has several attributes, drop any attribute `B` that isn't pulling its weight	Replace `X → A` with `(X - {B}) → A` when `(X - {B}) = X`
3. FD set simplification	Delete any FD that the rest of the set already implies	Delete `X → A` if `X` , computed *without* that FD, still contains `A`

Two things make this reliable in practice:

- **Do step 1 first, completely.** A compound RHS can hide the fact that one of its attributes is redundant while another isn't — the case-study `Department` example below is exactly that.
- **In steps 2 and 3, recompute the closure against the FD set as it currently stands**, including edits already made. Each removal changes what the remaining FDs can derive.

Worked example — the mechanical case

`R [A, B, C, D, E, F, G, H]` with `F = {A → B, {A,B,C,D} → E, {E,F} → G, {E,F} → H, {A,C,D,F} → {E, G}}`:

Step 1: RHS split	Step 2: LHS simplify	Step 3: delete redundant
`A → B`	`A → B`	`A → B`
`{A,B,C,D} → E`	`{A,C,D} → E` (`B` redundant, since `A → B`)	`{A,C,D} → E`
`{E,F} → G`	`{E,F} → G`	`{E,F} → G`
`{E,F} → H`	`{E,F} → H`	`{E,F} → H`
`{A,C,D,F} → E`	`{A,C,D,F} → E`	*dropped* — `{A,C,D,F}` already contains `E` without it

[44] courses/infs1200/attribute-closure.pdf

Step 1: RHS split	Step 2: LHS simplify	Step 3: delete redundant
`{A,C,D,F} → G`	`{A,C,D,F} → G`	*dropped* — same reason (`E` then `{E,F} → G`)

Minimal cover: `{A → B, {A,C,D} → E, {E,F} → G, {E,F} → H}`, often written with the last two recombined as `{E,F} → {G, H}`.

The `B` in `{A,B,C,D} → E` is the classic step-2 case: `A → B` already puts `B` in the closure of anything containing `A`, so naming it in the LHS adds nothing.

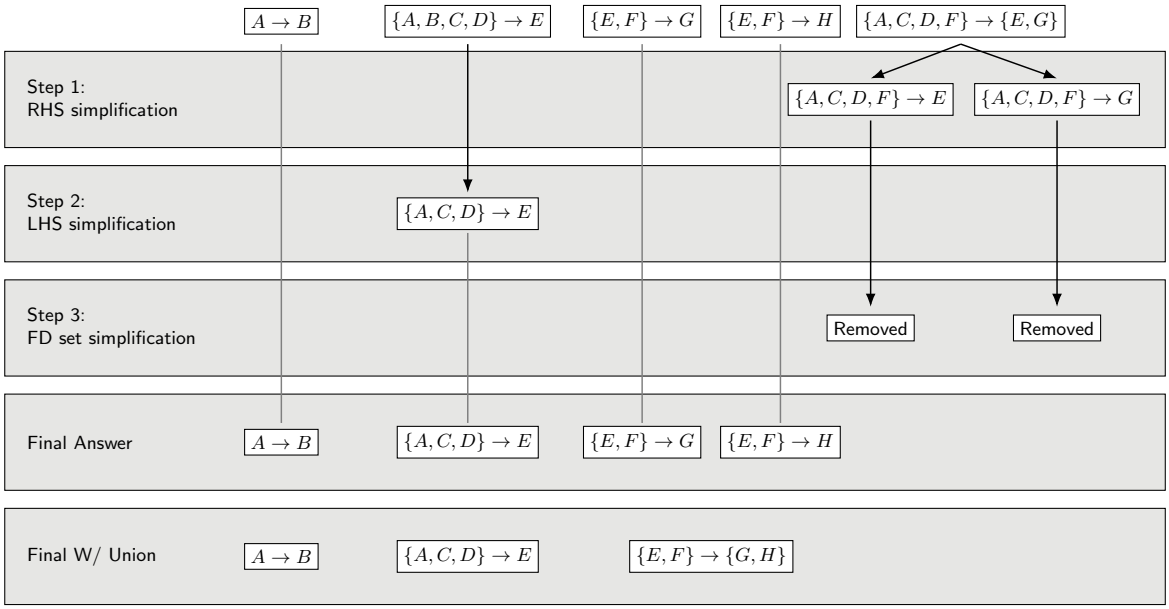


Figure 3: The same reduction seen as a flow, one band per step. Each original FD enters at the top; a black arrow means that step transformed it (split it, or shortened its LHS), a thin grey line means it passed through untouched, and the `Removed` boxes are where step 3's closure test deleted an FD outright.

Worked example — a wide right-hand side

Here step 1 does almost all the work. `R [C, S, J, D, P, Q, V]` with

`F = {C → {S, J, D, P, Q, V}, {J,P} → C, {S,D} → P, J → S}`.

Step 1: RHS split	Step 2: LHS simplify	Step 3: delete redundant
`C → S`	`C → S`	*dropped*
`C → J`	`C → J`	`C → J`
`C → D`	`C → D`	`C → D`
`C → P`	`C → P`	*dropped*
`C → Q`	`C → Q`	`C → Q`
`C → V`	`C → V`	`C → V`
`{J,P} → C`	`{J,P} → C`	`{J,P} → C`
`{S,D} → P`	`{S,D} → P`	`{S,D} → P`
`J → S`	`J → S`	`J → S`

- **Step 2 changes nothing.** Only `{J,P} → C` and `{S,D} → P` have a compound LHS, and neither attribute can go: `{J}` = `{J,S}` and `{P}` = `{P}` both miss `C`; `{S}` = `{S}` and `{D}` = `{D}` both miss `P`.
- **Step 3 deletes two.** `C → S` goes because `C → J` then `J → S` already reaches `S`. `C → P` then goes too: with it removed, `C` still reaches `S` (via `C → J`, `J → S`) and `D` directly, and `{S,D} → P` supplies `P`. Neither removal depends on the other being made first — test them in the opposite order and both still come out redundant.

Minimal cover: `{C → J, C → D, C → Q, C → V, {J,P} → C, {S,D} → P, J → S}`, or with same-LHS FDs recombined, `{C → {J, D, Q, V}, {J,P} → C, {S,D} → P, J → S}`.

With `C → P` removed, the *only* surviving statement about `P` is `{S,D} → P`, which is exactly the FD 3nf-synthesis[45] will turn into its own relation. Steps 2 and 3 are not tidying — they decide the shape of the schema you end up with.

Worked example — spotting a false answer

`R [A, B, C, D, E, F]` with `{D,E,F} → C`, `{A,B} → {D, C}`, `D → F`.

- **Step 1** gives `{D,E,F} → C`, `{A,B} → D`, `{A,B} → C`, `D → F`.
- **Step 2**: in `{D,E,F} → C`, is `F` redundant? `{D,E}` already contains `F` via `D → F`, so `{D,E}` = `{D,E,F}` — yes. It becomes `{D,E} → C`.
- **Step 3**: nothing further can go; all four are necessary.

Minimal cover: `{{D,E} → C, {A,B} → D, {A,B} → C, D → F}`.

The wrong answers to this question are instructive: keeping the redundant `F` in the first LHS (step 2 skipped), dropping `{A,B} → C` (it is *not* implied by the others — `{A,B}` without it reaches `D` and `F`, never `C`), or dropping `D → F` (which the simplified `{D,E} → C` now depends on).

[45] courses/infs1200/3nf-synthesis.pdf

Worked example — on a real schema

From week11-tutorial-case-study-9-payroll-system[46], the `Department` relation `[code, name, manager, buildingID, buildingName, buildingLocation, floor]` with `code` as the key, plus `buildingID → {buildingLocation, buildingName}` and `buildingName → {buildingLocation, buildingID}`.

After splitting the key's FD into single-attribute FDs, step 3 removes three of them:

- `code → buildingName` — derivable via `code → buildingID` then `buildingID → buildingName`.
- `code → buildingLocation` — derivable the same way.
- `buildingID → buildingLocation` — derivable via `buildingID → buildingName` then `buildingName → buildingLocation`.

Minimal cover: `{code} → {name, manager, buildingID, floor}`, `{buildingID} → {buildingName}`, `{buildingName} → {buildingLocation, buildingID}`.

Feeding that straight into 3nf-synthesis[47] yields `Department [code, name, manager, buildingID, floor]` and `Building [buildingName, buildingLocation, buildingID]` — the transitive `code → building…` redundancy removed exactly because step 3 removed the transitive FDs.

Redundancy can hide behind a chain

Step 3's test is "is `A` still reachable without this FD", which catches FDs whose work is done indirectly by two or more others. From week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis[48], `R [A, B, C, D, E, F]` with `A → {B,C,D,E,F}`, `{B,C} → A`, `{D,E} → B`, `C → D`:

- `A → B` is redundant — `A → D`, `A → E`, then `{D,E} → B`.
- `A → D` is redundant — `A → C`, then `C → D`.

leaving `A → {C, E, F}`, `{B,C} → A`, `{D,E} → B`, `C → D`. Neither removal is visible from a single FD; both need the closure test.

[46] courses/infs1200/week11-tutorial-case-study-9-payroll-system.pdf
[47] courses/infs1200/3nf-synthesis.pdf
[48] courses/infs1200/week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis.pdf

See also

- 3nf-synthesis[49] — the algorithm that consumes a minimal cover
- attribute-closure[50] · functional-dependencies[51] · normal-forms[52] · decomposition[53]
- 2026-05-04-normalisation-and-relational-database-schema-design[54] — Module 4, part 2
- week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis[55] · week11-tutorial-case-study-9-payroll-system[56] — practice

Normal Forms (1NF to BCNF)

Normalisation is the process of putting a relation schema through a series of tests — defined in terms of its FDs[57] and its keys — to certify which **normal form** it satisfies. A schema that fails a test is decomposed into smaller relations with better properties. Put another way, it is the process of *identifying redundancy in data*: an FD `A → B` is precisely the statement that every tuple sharing an `A` value is forced to repeat the same `B` value, and each normal form rules out a different way that forcing can go wrong. Introduced in 2026-05-04-normalisation-and-relational-database-schema-design[58].

Each normal form names a *class of redundancy* and rules it out. The guarantees stack: `BCNF` ⇒ `3NF` ⇒ `2NF` ⇒ `1NF`, so a BCNF relation is automatically in all the others, and the "highest normal form" of a relation is the strictest test it passes.

Overview

NF	Identifies	Test — for every non-trivial `X → A` in `F`
1NF	Non-atomic values	Relation has no multivalued attributes or nested relations
2NF	Partial dependencies (removes *some* anomalies)	`X` is **not** a proper subset of a candidate key, **or** `A` is a prime attribute

[49] courses/infs1200/3nf-synthesis.pdf
[50] courses/infs1200/attribute-closure.pdf
[51] courses/infs1200/functional-dependencies.pdf
[52] courses/infs1200/normal-forms.pdf
[53] courses/infs1200/decomposition.pdf
[54] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[55] courses/infs1200/week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis.pdf
[56] courses/infs1200/week11-tutorial-case-study-9-payroll-system.pdf
[57] courses/infs1200/functional-dependencies.pdf
[58] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf

NF	Identifies	Test — for every non-trivial `X → A` in `F`
3NF	Partial + transitive dependencies (removes *most* anomalies)	`X` is a superkey, **or** `A` is a prime attribute
BCNF	*All* anomalies — at the cost of not preserving all FDs	`X` is a superkey

Note the tests quantify over `F`, not `F` — **implied** FDs count too (see functional-dependencies[59]). Trivial FDs are always exempt.

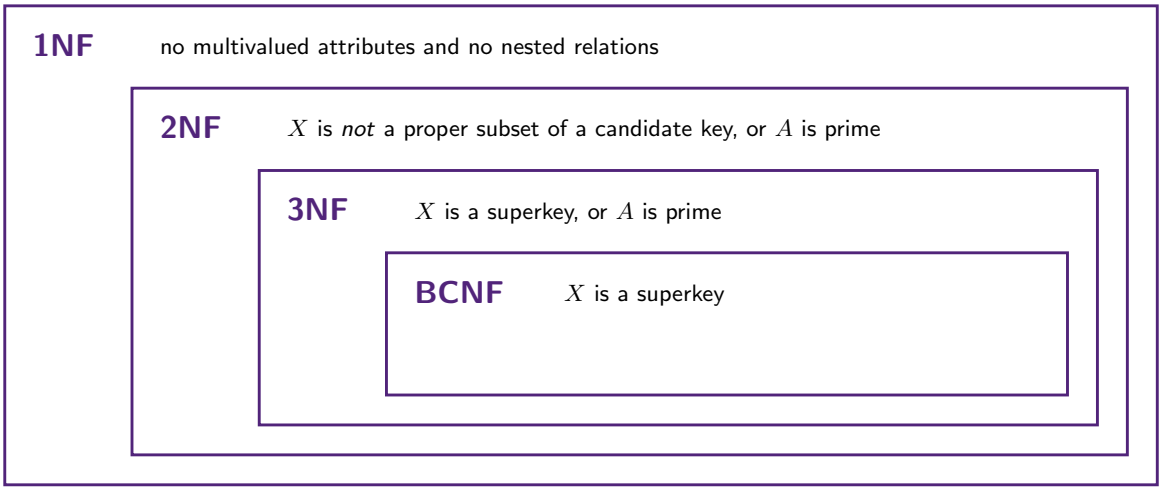


Figure 4: The four normal forms nest: each test is the previous one with an escape hatch removed, so a relation in an inner band is automatically in every band outside it. Each band carries its one-line test, and every test reads *for every non-trivial* `X → A` *in* `F`.

Read the figure from the inside out and the whole module fits on one line: BCNF asks only that every determinant is a superkey; 3NF adds the escape hatch "…or the determined attribute is prime"; 2NF weakens the superkey demand to "not a *proper subset* of a key"; 1NF stops asking about FDs at all and only asks that the values be atomic.

[59] courses/infs1200/functional-dependencies.pdf

What you need before testing

Every test above mentions candidate keys or prime attributes, so the first step is always the same: compute **all** candidate keys with attribute-closure[60], then mark each attribute **prime** (in at least one candidate key) or **non-prime**. Doing this against only the *primary* key is the most common source of wrong answers — 2NF's "proper subset of a candidate key" clause and 3NF's "or `A` is a prime attribute" escape hatch both depend on *every* candidate key, and a relation with several keys can pass against one and fail against another.

First Normal Form (1NF)

A relation is in 1NF if every attribute's domain contains only **atomic (simple, indivisible) values** — no set of values, no tuple of values (nested attributes), and no combination of the two.

An `items` column holding a comma-separated list violates 1NF; so does a `name` attribute that nests into `{firstName, familyName}`. Both standard ways of fixing it are unsatisfying:

- **Repeat** the non-nested columns once per value — one row per `(customerName, orderNum, item)` — which introduces redundancy; or
- **Flatten** into fixed columns (`item1`, `item2`, …) — which caps how many items an order can have and forces nulls on the shorter ones (guideline 3 of update-anomalies[61]).

That neither option is good is exactly the motivation for 2NF/3NF/BCNF, which say *how* to decompose an already-1NF relation.

Full and partial dependencies, and 2NF

- `X → Y` is a **full functional dependency** if removing *any* attribute from `X` breaks it — no proper subset of `X` determines `Y`.
- `X → Y` is a **partial dependency** if some attribute can be dropped from `X` and the FD still holds.

A relation `R` is in **2NF** if every **non-prime** attribute is **fully** functionally dependent on the primary key — informally, *no partial dependency on a key.*

Primary key or all candidate keys? The informal sentence above says "the primary key". The **formal** definition — the one given alongside every exam question, and the one that decides marks — is stated over *candidate keys*:

[60] courses/infs1200/attribute-closure.pdf
[61] courses/infs1200/update-anomalies.pdf

> `R` with FD set `F` is in 2NF iff, for every non-trivial `X → A` in `F` with `X  R` and `A  R`, either (1) `X` is **not** a proper subset of a candidate key for `R`, **or** (2) `A` is a prime attribute.

Use the formal version. It differs from the primary-key reading whenever a relation has more than one candidate key: a partial dependency on *any* candidate key breaks 2NF, not just one on the key you happened to nominate as primary. It also gives you a free shortcut in the other direction — if **every** attribute is prime, clause (2) is satisfied by every FD, so the relation is automatically in 2NF (and in 3NF).

Example — `Address [houseNum, street, postcode, state, value]` with `{houseNum, street, postcode} → {state, value}` and `postcode → state`. The only key is `{houseNum, street, postcode}`.

- `{houseNum, street, postcode} → value` is **full** — no subset of the LHS determines `value`.
- `{houseNum, street, postcode} → state` is **partial** — `postcode` alone already suffices.

`state` is non-prime and only partially dependent on the key, so `Address` is **not** in 2NF. Decompose into `Address [houseNum, street, postcode, value]` and `Postcodes [postcode, state]` with `Address.postcode references Postcodes.postcode`.

2NF doesn't fix much. `Employee [id, name, level, salary]` with `level → salary` satisfies 2NF — `level` isn't a *proper subset* of the key `{id}`, it's a whole non-key attribute — yet it still has all three anomalies from update-anomalies[62]. 2NF only catches redundancy hanging off *part of a composite key.*

Transitive dependencies, and 3NF

`X → Y` in `R` is a **transitive dependency** if some attribute set `Z` exists such that `Z` is neither a candidate key nor a subset of one, and both `X → Z` and `Z → Y` hold — i.e. `Y` depends on `X` only *through* an intermediate that isn't a key.

In `Employee [id, name, level, salary]` with `level → salary`: `level` is neither a candidate key nor part of one, `id → level` holds and `level → salary` holds, so `id → salary` is a transitive dependency.

`R` is in **3NF** iff, for every non-trivial `X → A` in `F` , `X` is a superkey **or** `A` is a prime attribute. Equivalently: whenever `A` is non-prime, `X` must be a superkey.

`level → salary` fails both halves — `level` isn't a superkey and `salary` isn't prime — so `Employee` violates 3NF. Decompose into `StaffAppointment [id, name, level]` and `StaffIncome [level, salary]`.

[62] courses/infs1200/update-anomalies.pdf

3NF still isn't always enough. `Teach [studentID, courseID, lecturer]` with `{studentID, courseID} → lecturer` and `lecturer → courseID` (keys: `{studentID, courseID}` and `{studentID, lecturer}`) **is** in 3NF — `lecturer → courseID` escapes via the prime-attribute clause, since `courseID` is prime. Yet:

- **Deletion anomaly** — deleting the last row for a `(studentID, courseID)` pair can destroy the only record of who teaches that course.
- **Insertion anomaly** — you can't record which lecturer teaches a course that currently has no enrolled students.

Partial vs transitive, in one line each

The distinction is a recurring exam question, and the marked answer is phrased like this:

> A **partial** dependency is where an attribute depends on only a *subpart of the primary key*; normalising to **2NF** solves it. A **transitive** dependency is where a non-prime attribute depends on *other non-prime attributes*; normalising to **3NF** solves it.

The trap in the distractors is the swap — attaching partial dependencies to 3NF and transitive ones to 2NF.

Boyce-Codd Normal Form (BCNF)

`R` is in **BCNF** iff, for every non-trivial `X → A` in `F` , `X` is a superkey. Informally: *whenever a set of attributes determines anything, it must determine everything.* BCNF is 3NF with the prime-attribute escape hatch removed.

In `Teach`, `lecturer → courseID` now fails — `lecturer` is not a superkey — so `Teach` is not in BCNF even though it is in 3NF. That gap between 3NF and BCNF is exactly the set of relations where 3NF's prime-attribute clause is letting an anomaly through.

BCNF is strictly stronger, but not strictly better: decomposing to reach it can lose FDs that 3NF would have kept.

3NF or BCNF? What each one buys

Real designs aim for one of these two, and the choice is a straight swap:

	Removes all anomalies	Preserves all FDs
BCNF		
3NF		

Neither dominates. BCNF eliminates the redundancy but a BCNF decomposition may scatter an FD's attributes across relations so that the constraint can no longer be checked without a join; 3NF keeps every FD checkable inside a single relation but leaves the `Teach`-shaped anomalies above in place. See decomposition[63] for dependency preservation and the BCNF algorithm, and 3nf-synthesis[64] for the algorithm that guarantees the 3NF side of the table.

Determining the highest normal form

1. Compute **all** candidate keys (attribute-closure[65]); mark prime and non-prime attributes.
2. Split every FD so its RHS is a single attribute — the tests are stated per-attribute, and a compound RHS can pass on one attribute and fail on another.
3. Test each non-trivial FD `X → A` from the strictest end down: is `X` a superkey (BCNF)? If not, is `A` prime (3NF)? If not, is `X` a proper subset of a candidate key (breaks 2NF) or not (breaks 3NF only)?
4. The relation's normal form is the strictest one **no** FD violates.

Relation and FDs	Candidate keys	Highest NF	Why
`R [A,B,C,D,E]`, `A → {B,C,D,E}`, `E → A`	`{A}`, `{E}`	**BCNF**	Every FD's LHS is itself a superkey
`R [A,B,C,D]`, `{A,B} → {C,D}`, `{C,D} → A`, `D → B`	`{A,B}`, `{C,D}`, `{A,D}`	**3NF**	Every attribute is prime, so nothing can break 3NF; but `D → B` has a non-superkey LHS, breaking BCNF
`R [A,B,C,D]`, `{A,C,D} → B`, `{A,C} → D`, `D → C`, `{A,C} → B`	`{A,C}`, `{A,D}`	**3NF**	`D → C`: `D` isn't a superkey (not BCNF), but `C` is prime (3NF survives)
`R [A,B,C,D,E,F]`, `{A,B} → {C,D,E}`, `C → F`, `E → {A,B}`	`{A,B}`, `{E}`	**2NF**	No FD's LHS is a *proper subset* of a key, so 2NF holds; `C → F` has a non-superkey LHS and non-prime RHS, breaking 3NF

[63] courses/infs1200/decomposition.pdf
[64] courses/infs1200/3nf-synthesis.pdf
[65] courses/infs1200/attribute-closure.pdf

Relation and FDs	Candidate keys	Highest NF	Why
`R [A,B,C,D,E]`, `B → {C,D}`, `A → E`	`{A,B}`	**1NF**	Both `A` and `B` are proper subsets of the key with non-prime RHSs — partial dependencies, breaking 2NF
`R [A,B,C,D,E,F]`, `{A,E} → D`, `{B,C} → A`, `B → F`, `F → E`	`{B,C}`	**1NF**	`B → F` is a partial dependency: `B` `{B,C}` and `F` is non-prime

The fourth and fifth rows are the pair worth internalising — the *same* shape of violation (non-superkey LHS, non-prime RHS) lands on 3NF or 2NF purely depending on whether the LHS sits inside a candidate key.

A subtlety from the tutorial: in `R [A,B,C,D,E,F,G]` with `A → {B,C,D}`, `D → A`, `C → {F,G}` (keys `{A,E}` and `{D,E}`), splitting `A → {B,C,D}` gives `A → B`, `A → C` and `A → D`. The first two are partial dependencies (`A` `{A,E}`, and `B`/`C` are non-prime) so the relation is only in 1NF — but `A → D` is *not* a partial dependency, because `D` is prime (it appears in key `{D,E}`). That's why step 2 above insists on splitting the RHS first.

Denormalisation

Since anomalies come from redundancy it's tempting to decompose as aggressively as possible — but a heavily decomposed schema needs more JOIN[66]s to answer the same query. **Denormalisation** is the *deliberate*, controlled relaxation of a normal form to improve query performance: fewer joins (faster queries) and fewer foreign keys (less storage and maintenance overhead), which suits analytical workloads or a small set of very frequent pre-joined queries.

The word "deliberate" is load-bearing. Denormalisation is a trade-off made knowingly, accepting the anomalies it reintroduces — not an excuse for a schema that was never normalised in the first place.

[66] courses/infs1200/sql-joins.pdf

See also

- functional-dependencies[67] · attribute-closure[68] · decomposition[69] · 3nf-synthesis[70] · minimal-cover[71]
- update-anomalies[72] — the anomalies each normal form is removing
- 2026-05-04-normalisation-and-relational-database-schema-design[73] — Module 4, part 2
- week10-tutorial-applied-class-9-functional-dependencies[74] · week10-tutorial-case-study-8-dirt-road-driving[75] — highest-normal-form practice

[67] courses/infs1200/functional-dependencies.pdf
[68] courses/infs1200/attribute-closure.pdf
[69] courses/infs1200/decomposition.pdf
[70] courses/infs1200/3nf-synthesis.pdf
[71] courses/infs1200/minimal-cover.pdf
[72] courses/infs1200/update-anomalies.pdf
[73] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[74] courses/infs1200/week10-tutorial-applied-class-9-functional-dependencies.pdf
[75] courses/infs1200/week10-tutorial-case-study-8-dirt-road-driving.pdf