Normalisation and Relational Database Schema Design

Table of contents

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**.

Today's 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.

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

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

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?

> **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?

> **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?

> **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

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)*
`{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]

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?

i 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

i 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

You should now be able to test a relation schema against 1NF/2NF/3NF/ BCNF given a set of FDs, decompose a universal relation into lossless-join anomaly-free BCNF relations (top-down), and compute 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).

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