Decomposition and BCNF Design

Table of contents

The two properties . . . 1
Lossless join and spurious tuples . . . 2
Testing for a lossless join . . . 3
Which FDs hold in a decomposed relation . . . 3
Dependency preservation . . . 4
The design landscape . . . 6
The BCNF decomposition algorithm . . . 6
Implicit FDs matter . . . 7
Common mistakes . . . 8
See also . . . 8

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[1] and 2026-05-04-normalisation-and-relational-database-schema-design[2].

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

[1] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[2] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf

Property	Question it answers	Guaranteed by
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.

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[3]). 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[4] 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.

[3] courses/infs1200/sql-joins.pdf
[4] courses/infs1200/update-anomalies.pdf

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, 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[5] turns into an algorithm: give every FD in the cover its own relation X Y, and preservation is automatic.

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:

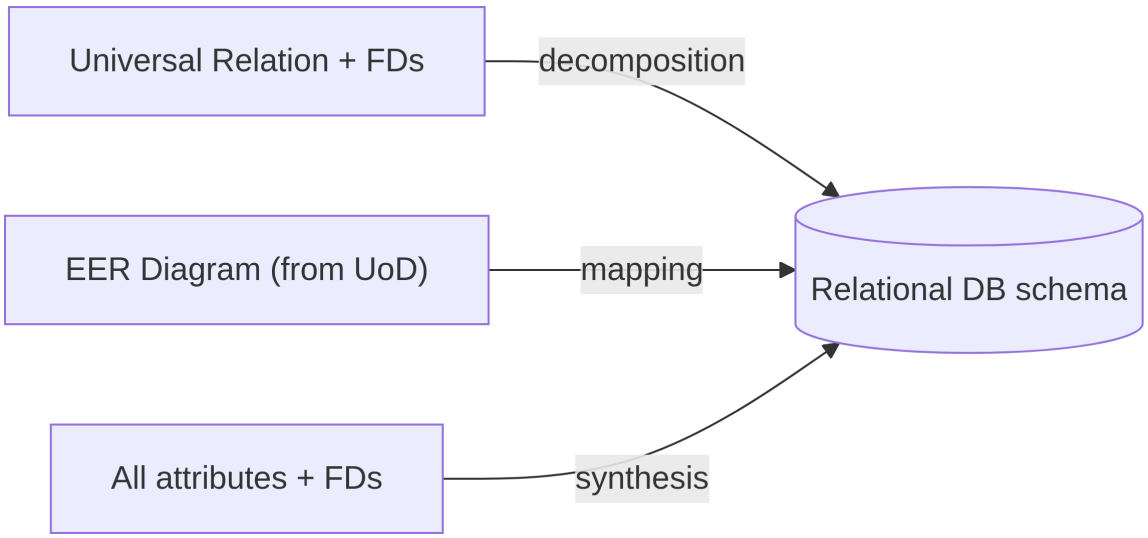
	BCNF	3NF
Removes all anomalies		
Preserves all FDs		

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

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

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[6]):



- **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[7]. Lossless-join, dependency-preserving and 3NF. See 3nf-synthesis[8].

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[9]).

The BCNF decomposition algorithm

```
D := {R};
while (some relation Q in D is not in BCNF):
```

[6] courses/infs1200/er-to-relational-mapping.pdf
[7] courses/infs1200/minimal-cover.pdf
[8] courses/infs1200/3nf-synthesis.pdf
[9] courses/infs1200/functional-dependencies.pdf

```
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 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[10] and the decomposition-tree question in 2026-05-04-normalisation-and-relational-database-schema-design[11]:

- **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[12] · 3nf-synthesis[13] · minimal-cover[14] · attribute-closure[15]

[10] courses/infs1200/week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis.pdf
[11] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[12] courses/infs1200/normal-forms.pdf
[13] courses/infs1200/3nf-synthesis.pdf
[14] courses/infs1200/minimal-cover.pdf
[15] courses/infs1200/attribute-closure.pdf

- update-anomalies[16] — guideline 4 and the informal view of spurious tuples
- 2026-05-04-normalisation-and-relational-database-schema-design[17] — Module 4, part 2
- week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis[18] · week11-tutorial-case-study-9-payroll-system[19] — practice

[16] courses/infs1200/update-anomalies.pdf
[17] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[18] courses/infs1200/week11-tutorial-applied-class-10-normalisation-bcnf-3nf-synthesis.pdf
[19] courses/infs1200/week11-tutorial-case-study-9-payroll-system.pdf