INFS1200 — Week 10 Notes

Table of contents

Database Design Guidelines and Functional Dependencies **2**
- Today's outline . . . 2
- Informal design guidelines . . . 2
- Functional dependencies . . . 4
- Question 1 — Functional dependencies . . . 5
- Question 2 — Anomalies . . . 6
- Question 3 — Possible keys . . . 6
- Question 4 — Possible superkeys . . . 6
- Question 5 — Finding keys . . . 8
- Question 6 — Finding keys . . . 9
- Question 7 — Finding keys . . . 9
- Summary . . . 9

Applied Class 9: Functional Dependencies **10**
- Section A — Anomalies and functional dependencies . . . 10
- Section B — Closures . . . 13
- Section C — Candidate keys . . . 14
- Section D — Highest normal form . . . 15

Case Study 8: Dirt Road Driving (Payroll System — Anomalies and FDs) **16**
- The payroll schema . . . 17
- Section A — Anomalies . . . 19
- Section 2 — Functional dependencies and highest normal form . . . 20

Reference material **21**
- Attribute Closure and Finding Keys . . . 21
- Decomposition and BCNF Design . . . 26
- Functional Dependencies . . . 33
- Normal Forms (1NF to BCNF) . . . 38
- NULLs in SQL . . . 45

Update Anomalies and Informal Design Guidelines 48

Database Design Guidelines and Functional Dependencies

Module 4, part 1. This module concerns *database design theory* — how to measure the quality of a relational schema, and how to fix a poor one.

Today's outline

- Informal design guidelines
- Functional dependencies (FDs) — definition, keys, closure

Informal design guidelines

Four informal measures of relational schema quality:

1. Make sure the **semantics of the attributes are clear** in the schema.
2. **Reduce redundant values** in tuples.
3. **Reduce null values** in tuples.
4. **Disallow spurious tuples** (don't allow lossy joins).

Guideline 1 — one relation, one meaning

Design each relation so its meaning is easy to explain. Don't combine attributes from multiple entity/relationship types into one relation — this confuses the entity's meaning and causes redundancy.

```
EMPLOYEE [Ename, Ssn, Bdate, Address, Dnumber]     -- FK: Dnumber
DEPARTMENT [Ename, Dnumber, Dmgr_ssn]              -- FK: Dmgr_ssn

-- vs. combining them into one relation:
EMP_DEPT [Ename, Ssn, Bdate, Address, Dnumber, Dname, Dmgr_ssn]
```

Guideline 2 — avoid update anomalies

Design base relations so that no insertion, deletion, or modification anomalies occur. If anomalies can't be avoided, applications must update relations carefully enough to preserve database integrity.

Motivating example — an `Employee [ID, Name, Level, Salary]` table where salary is fixed per level (Developer=60,000, Manager=700,000, Driver=50,000, Administration=50,000):

ID	Name	Level	Salary
1	Paris	Developer	60,000
2	Anna	Manager	700,000
3	Ben	Manager	700,000
4	Rose	Driver	50,000
5	Jack	Developer	60,000
6	Charlie	Administration	50,000

- **Modification anomaly**: updating one developer's salary makes the "Developer" salary inconsistent with the others.
- **Deletion anomaly**: deleting Charlie loses the fact that Administration pays 50,000 (Charlie was the only Administration row).
- **Insertion anomaly**: can't record a Cook's salary until an employee actually holds that position; inserting a new Developer row with a *different* salary makes the Developer salary inconsistent.

These aren't just textbook abstractions — a widely reported 2021 issue with Australia's vaccine certificate system arose from essentially this class of problem: state vaccination staff recorded details in a way that didn't precisely match federal records, so contradicting datasets failed to reconcile automatically.

Decomposition

A **decomposition** of relation `R` replaces it with two or more relations such that (1) each new relation's attributes are a subset of `R`'s (no foreign attributes), and (2) every attribute of `R` appears in at least one new relation.

Decomposing the `Employee` example correctly:

```
Employee [ID, Name, Level]
Level_Salary [Level, Salary]
```

removes all three anomaly types — modifying one developer's salary in `Level_Salary` doesn't touch `Employee`; deleting an employee's row doesn't touch `Level_Salary`; a Cook's salary can be stored in `Level_Salary` before anyone holds that role. (An *incorrect* decomposition — e.g. splitting into `[ID, Name, Salary]` and `[Salary, Level]` — does **not** fix the anomalies, since `Salary` isn't a valid link back to `Level` on its own.)

The join operation and lossless joins

`R1  R2` (natural join): concatenate each tuple of `R1` with every tuple of `R2` agreeing on their common attributes.

A decomposition of `R` into `R1` and `R2` is a **lossless join decomposition** if, for every legal instance, `R = R1  R2` — i.e. breaking `R` apart and rejoining it gives back exactly `R`, no more, no less.

Lossy join example — decomposing `R [A, B, C]` into `R1 [A, B]` and `R2 [B, C]` when `B` does *not* uniquely 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
```

Here "loss" means loss of *information*, not loss of tuples — two extra ("spurious") rows appear because `B = 2` maps to *two different* `C` values (`3` and `9`), so the join can't tell which `A` goes with which `C`.

Guideline 4 — join on keys

Design relation schemas so they can be joined using equality conditions on primary/foreign keys, in a way that guarantees no spurious tuples are generated.

Functional dependencies

Motivating question: how can we be *sure* that all employees at the same level have the same salary, rather than it just happening to be true of the current data? Databases let you declare this formally via a **functional dependency (FD)**: `level → salary` ("level determines salary" — if we know an employee's level, we know their salary).

Formal definition

An FD `X → Y` holds on relation `R` if, for every legal instance `r` of `R` and all tuple pairs `t1, t2` `r`:

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y]$$

i.e. if two tuples agree on `X`, they must agree on `Y`. `X → Y` is a constraint between two attribute sets `X` and `Y` — it restricts which tuples can legally appear together in an instance of `R`.

Crucially: an FD is a statement about *all* allowable instances. You can check whether a *given* instance **violates** an FD, but you can never *prove* an FD holds just by looking at one instance — FDs must be identified from the application's real-world semantics (business rules), not reverse-engineered from a data sample.

Question 1 — Functional dependencies

*Given `R [A, B, C, D]` with rows `(1,2,3,4), (2,3,4,6), (6,7,8,9), (1,3,4,5)` — which FDs **cannot** be true?*

A. `B → C` B. `B → D` C. `D → B` D. All of the above can be true E. None of the above can be true

> **Solution**
>
> **B.** Rows 2 and 4 both have `B = 3`, but row 2 has `D = 6` while row 4 has `D = 5` — two tuples agreeing on `B` disagree on `D`, so `B → D` is **impossible**. `B → C` and `D → B` are both still *possible* given this instance (no counterexample rows exist for either) — remember, "possible" here just means "not yet contradicted", not "proven".

Fixing anomalies via FDs

Given `level → salary`, decomposing `Employee` into `[ID, Name, Level]` and `[Level, Salary]` fixes all three anomaly types from before — updating a developer's salary in `[Level, Salary]` no longer creates inconsistency; deleting an employee no longer loses level/salary mappings; a new level's salary can be recorded independent of whether any employee holds it yet.

Question 2 — Anomalies

*Given `R [A, B, C, D]` with `D → {A, C}` and rows `(1,4,2,5)`, `(2,3,4,3)`, `(1,1,2,5)` — which is **not** an example of an update anomaly?*

A. Deleting `<2,3,4,3>` B. Inserting `<3,5,3,3>` C. Modifying `<1,1,2,5>` to `<1,2,2,5>` D. Inserting `<1,null,2,4>` E. Modifying `<1,1,2,5>` to `<1,2,3,5>`

> **i** Solution
>
> **C.** Modifying `B` from `1` to `2` (row `<1,1,2,5>` → `<1,2,2,5>`) doesn't touch `A`, `C`, or `D` at all — `B` isn't constrained by `D → {A, C}`, so no anomaly results. A is an anomaly (deletes the only row with `D=3`, losing the fact that `D=3 → {A=2, C=4}`). B is an anomaly (row 2, `<2,3,4,3>`, already establishes `D=3 → {A=2, C=4}`; inserting `<3,5,3,3>` gives `D=3` a *different* `A`/`C` pair, `{A=3, C=3}`, contradicting it). D is an anomaly (a `null` in the primary key violates entity integrity). E is an anomaly (both existing `D=5` rows have `C=2`; changing one to `C=3` breaks `D → C` consistency).

Keys

A **key** is a *minimal* set of attributes that uniquely identifies a relation's tuples — equivalently, a minimal set of attributes that functionally determines *all* attributes in the relation. A **superkey** is any set of attributes (not necessarily minimal) that uniquely identifies the relation.

Question 3 — Possible keys

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

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

> **i** Solution
>
> **E.** `B` doesn't determine `D` or `A` (`{B} = {B, C}`, missing `A, D`) — not a key. `C` is symmetric to `B` (`{C} = {B, C}`) — also not a key. `{B, D}` *does* determine everything (`D` alone already does, via `D → {A,B,C}`), but it's **not minimal** — `D` alone is already a key, so `{B, D}` is a superkey, not a (candidate) key.

Question 4 — Possible superkeys

Same FDs as Question 3 — which is a superkey?

A. `D` B. `{B, D}` C. `{B, C, D}` D. All are superkeys E. None are superkeys

Solution

D. `D → {A, B, C}` means `D` alone determines every attribute — `D` is a key, and therefore *every* superset of `D` (`{B,D}`, `{B,C,D}`) is automatically a superkey too.

Explicit and implicit FDs; closure of F

Given a set of *explicit* FDs, further *implicit* (inferred) FDs can be derived — e.g. from `ID → level` and `level → salary`, we can infer `ID → salary`. The notation `F  X → Y` means `X → Y` can be inferred from `F` (`X` = left-hand side/LHS, `Y` = right-hand side/RHS).

- **Trivial FDs** hold regardless of `F` (the LHS already contains the RHS) — e.g. `A → A`, `{A,B,C} → {A,B}`.
- **Non-trivial FDs** depend on the specific `F` — e.g. `A → B`.

The **closure of F**, written `F` , is the set of *all* FDs (trivial and non-trivial) implied by `F`. `F` can be computed via Armstrong's Axioms, but that's outside this course's scope — instead we focus on **attribute closure**.

Attribute closure (X)

`X` is the set of all attributes determined by `X` under `F`:

```
X+ := X;
repeat
    old X+ := X+;
    for each FD Y → Z in F:
        if Y  X+ then X+ := X+  Z;
until (old X+ = X+);
```

Example — `Employee (ID, level, salary, name)`, `F = {ID → level, level → salary, ID → name}`:

1. `ID  = {ID}`
2. `ID  = {ID, level}` (using `ID → level`)
3. `ID  = {ID, level, salary}` (using `level → salary`)
4. `ID  = {ID, level, salary, name}` (using `ID → name`)

Larger example — `R [pNumber, pName, pLocation, dNum, dName, mgrSSN, mgrStartDate]`, `F = {pNumber → {pName, pLocation, dNum}, dNum → {dName, mgrSSN, mgrStartDate}}`:

1. `{pNumber}  = {pNumber}`

2. {pNumber} = {pNumber, pName, pLocation, dNum} (FD1)
3. {pNumber} = {pNumber, pName, pLocation, dNum, dName, mgrSSN, mgrStartDate} (FD2) — the full relation, so pNumber is a key.

Finding a superkey — show {sName, pNum} is a superkey for SupplierPart (sName, city, status, pNum, pName, qty) with F = {sName → city, city → status, pNum → pName, {sName, pNum} → qty}:

```
{sName, pNum}+ = {sName, pNum}
{sName, pNum}+ = {sName, pNum, city}          using sName → city
{sName, pNum}+ = {sName, pNum, city, status}  using city → status
{sName, pNum}+ = {sName, pNum, city, status, pName}   using pNum → pName
{sName, pNum}+ = {sName, pNum, city, status, pName, qty}  using {sName,pNum}→qty
```

Since the closure covers every attribute of SupplierPart, {sName, pNum} is a superkey.

Tips for finding keys

Given a relation R and FD set F: S R is a key iff (1) S = R, and (2) no proper subset S' S also has S' = R. For n attributes there are 2 subsets to consider in the worst case, but two shortcuts help:

1. **If an attribute never appears on the RHS of any FD**, it must be part of *every* key (nothing else can ever produce it).
2. **If S is already a key, don't test any superset of S** — it'll be a superkey, not a (minimal) key.
3. A relation can have **multiple keys** of different sizes.

To fully show {sName, pNum} is a **key** (not just a superkey) for SupplierPart, also confirm minimality: {sName} = {sName, city, status} and {pNum} = {pNum, pName} — neither proper subset covers all attributes, so {sName, pNum} is minimal.

Question 5 — Finding keys

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

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

> **i** Solution
>
> **D.** {B} = {B, C, D, E, F} — misses A. {B, E} = same, still misses A. {E, F} = {D, E, F} — misses far more. {A, B} = {A, B, C, D, E, F} — everything. A never

appears on any RHS, so (per the tip above) it *must* be in every key — confirming why options A–C, none of which include `A`, can never be keys.

Question 6 — Finding keys

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

A. `{A, B, D, E}` B. `{B, C, E}` C. `{C, D, E}` D. All of these are keys E. None of these are keys

Solution

B. `{A, B, D, E}` is a *superkey* but not minimal: since `D → A` already holds, `A` is redundant once `D` is present, so this isn't the smallest determining set. `{C, D, E} : D → C` (redundant, already have `C`), `D → A` adds `A`, `{C,E} → A` (redundant), `{A,E} → D` (redundant, already have `D`) — final closure `{A, C, D, E}`, missing `B`, so it's not even a superkey. `{B, C, E} : {C,E} → A` adds `A` (→ `{A,B,C,E}`), then `{A,E} → D` adds `D` (→ `{A,B,C,D,E}`) — every attribute, and no proper subset of `{B,C,E}` achieves this, so it's a minimal key.

Question 7 — Finding keys

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

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

Solution

C. `{A, B} = {A, B, E}` (via `{A,B}→E`) — misses `C, D, F`. `{A, B, C} = {A, B, C, E}` (`C→{B,E}` is redundant here) — still misses `D, F`. `{A, D} = {A, D}` — none of the three FDs' left-hand sides (`{A,B}`, `C`, `{E,D}`) are subsets of `{A, D}` alone, so nothing can be added at all. `{A, C, D} : C → {B, E}` adds `B, E` (→ `{A,B,C,D,E}`), then `{E,D} → F` adds `F` — every attribute. `{A, C, D}` is minimal and complete — a key.

Summary

You should now be familiar with informal design guidelines, functional dependencies (their formal definition and how to identify them), keys/superkeys, and how to compute attribute

closure. These are the foundation for normalisation, covered next lecture. See week10-tutorial-applied-class-9-functional-dependencies[1] and week10-tutorial-case-study-8-dirt-road-driving[2] for practice.

Applied Class 9: Functional Dependencies

Practice for 2026-04-27-database-design-guidelines-and-functional-dependencies[3].

Section A — Anomalies and functional dependencies

Question A.1

Based on the following data, provide an example and explanation of an insertion, deletion and modification anomaly.

A	B	C	D	E
2	2	1	5	6
2	3	1	5	4
3	4	5	3	2
3	5	5	1	3

Functional dependencies: `{A} → {C}`, `{B} → {D, E}`.

> **Working**
>
> - **Insertion anomaly**: insert anything into `B`, `D` and `E` (with `B`, `D`, `E` values that don't already exist in the data) without also inserting a value into `A` — the row is incomplete but there's no `A` to attach it to.
> - **Deletion anomaly**: deleting the tuple with `B = 4` loses the information for the FD `B → {D, E}`, specifically `4 → {3, 2}` — no other row records that mapping.
> - **Modification anomaly**: updating `{2, 2, 1, 5, 6}` to `{2, 4, 1, 5, 6}` creates an inconsistency in `B → {D, E}` — it now implies both `4 → {3, 2}` (from the existing row with `B=4`) and `4 → {5, 6}` (from the just-modified row).

[1] courses/infs1200/week10-tutorial-applied-class-9-functional-dependencies.pdf
[2] courses/infs1200/week10-tutorial-case-study-8-dirt-road-driving.pdf
[3] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf

Question A.2

Based on `A → B, B → C, {C, D} → E`, fill in the blanks (`?`) below so that no FD is violated.

A	B	C	D	E
1	2	1	6	2
1	?	1	4	?
2	4	2	7	4
3	?	?	4	?

i Working

A	B	C	D	E
1	2	1	6	2
1	2	1	4	3
2	4	2	7	4
3	2	1	4	3

Row 2's `B` is **forced**: `A = 1` also appears in row 1 (`B = 2`), and `A → B` requires matching `A` values to share the same `B`. Row 2's `E` is then constrained by `{C, D} → E`: row 2 has `{C=1, D=4}`, a combination that will recur in row 4 — both rows sharing that `{C, D}` pair must agree on `E` (here, `3`).
Row 4's `B` and `C` are **not strictly forced** by the given FDs (`A = 3` doesn't recur elsewhere, so `A → B` places no constraint on it) — but choosing `B = 2, C = 1` is a valid, self-consistent choice: it matches row 1/2's `B → C` mapping (`B=2 → C=1`), and then `{C=1, D=4} → E` correctly forces row 4's `E` to match row 2's (`E = 3`), since both rows now share `{C=1, D=4}`. Note the knock-on: once `B = 2` is chosen for row 4, `B → C` makes `C = 1` no longer free either.

A	B	C	D	E
1	2	1	6	2
1	**2**	1	4	**3**
2	4	2	7	4
3	**2**	**1**	4	**3**

rows 1–2 share $A = 1$, so $A \to B$ forces row 2's B

rows 2 and 4 share $\{C = 1, D = 4\}$, so $\{C, D\} \to E$ forces their E to agree

purple = a cell that was ?

Figure 1: Which FD forces which cell. Purple values are the ones that were `?`. Cyan pairs the rows that share an `A` value with the `B` values that `A → B` therefore ties together; lavender pairs the rows that share `{C, D}` with the `E` values that `{C,D} → E` ties together.

Question A.3

Based on the following data, identify which options are ***potential*** *functional dependencies.*

```
A  B  C  D  E
1  X  1  M  1
2  Y  1  M  1
3  Y  4  N  3
4  W  2  L  5
5  W  2  M  1
6  T  5  O  2
```

- `A → B` — `B → A` — `A → C` — `B → C` — `C → D` — `C → E` — `D → E`
- `{A, B} → C` — `{B, C} → E` — `{B, C, D} → E`

Working

Potential FDs: `A → B`, `A → C` (both trivially hold — every row has a distinct `A`, so there's never a repeated-`A` pair to violate anything). `D → E` (every repeated `D` value agrees on `E`: `D=M` in rows 1, 2, 5 always has `E=1`). `{A, B} → C` (trivially holds, since `A` alone is already unique per row). `{B, C, D} → E` (every `(B,C,D)` triple in the table is distinct, so it trivially holds too).
Not potential (contradicted by the data):

- `B → A`: `B=Y` appears in rows 2 and 3 with different `A` (`2` vs `3`).
- `B → C`: `B=Y` appears in rows 2 and 3 with different `C` (`1` vs `4`).
- `C → D`: `C=2` appears in rows 4 and 5 with different `D` (`L` vs `M`).
- `C → E`: `C=2` appears in rows 4 and 5 with different `E` (`5` vs `1`).
- `{B, C} → E`: `{B=W, C=2}` appears in rows 4 and 5 with different `E` (`5` vs `1`).

Section B — Closures

Question B.1

Given `R [A, B, C, D, E, F, G]` with `{A} → {D}`, `{B, C} → {A}`, `{C} → {F}`, `{F} → {E}` — find the following closures.

Working

- `{C} = {C, F, E}` (via `C→F`, then `F→E`).
- `{B, C, A} = {A, B, C, D, E, F}` (via `A→D`, `C→F`, `F→E`).
- `{B, C} = {A, B, C, D, E, F}` (via `{B,C}→A`, then `A→D`, `C→F`, `F→E`) — the same closure, so `A` adds nothing once `B` and `C` are present.

Mind `G`. Both closures stop one attribute short of `R`, so *neither* set is a superkey as the question is written: `G` appears in `R [A, B, C, D, E, F, G]` but on no side of any FD, and an attribute that never appears on a right-hand side must belong to **every** key. The candidate key is therefore `{B, C, G}` — `{B,C,G}` is all of `R`, and neither `{B, G}` nor `{C, G}` reaches `A`. (Read the relation instead as `R [A, B, C, D, E, F]`, the shape Section C.1 uses with the same style of FD set, and `{B, C}` is the candidate key with `{A, B, C}` a non-minimal superkey.) See attribute-closure[4].

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

Question B.2

Given `R [A, B, C, D, E, F]` *with* `{A} → {B, C}`, `{C, D} → {E}`, `{A, C} → {E}`, `{B} → {D}`, `{E} → {A, B}` — *find the following closures.*

> **i** Working
>
> - `{A, F} = {A, B, C, D, E, F}` (via `A→{B,C}`, `B→D`, `{C,D}→E` or `{A,C}→E`, `E→{A,B}`).
> - `{C, D, F} = {A, B, C, D, E, F}` (via `{C,D}→E`, `E→{A,B}`).
> - Both `{A, F}` and `{C, D, F}` are **minimal** — removing any single attribute from either set makes the reduced set no longer a superkey.

Section C — Candidate keys

Question C.1

`R [A, B, C, D, E, F]` *with* `{A, E} → {D}`, `{B, C} → {A}`, `{B} → {F}`, `{F} → {E}`. *Find all candidate keys.*

> **i** Working
>
> **Candidate key(s): `{B, C}`.**
> `{B, C}` : `B→F` adds `F`; `F→E` adds `E`; `{B,C}→A` adds `A`; `{A,E}→D` adds `D` — covers everything, and no proper subset of `{B,C}` does (neither `B` nor `C` alone determines the other).

Question C.2

`R [A, B, C, D, E, F]` *with* `{A} → {B, C}`, `{C, D} → {E}`, `{A, C} → {E}`, `{B} → {D}`, `{E} → {A, B}`. *Find all candidate keys.*

> **i** Working
>
> **Candidate keys: `{B, C, F}`, `{C, D, F}`, `{A, F}`, `{E, F}`.**
> `F` never appears on any FD's right-hand side, so it must be part of *every* key. Among `{A, B, C, D, E}`, the FDs form a tightly-coupled cluster (`A`, `B`/`D`, `C`/`E` combinations all inter-derive each other) — `A`, `E`, `{B,C}`, and `{C,D}` are each independently sufficient to derive the rest of that cluster, giving four minimal keys once `F` is added to each.

Question C.3

R [A, B, C, D] with {A, B} → {C, D}, {C} → {A, B, D}, {D} → {C}. Find all candidate keys.

> **Working**
>
> **Candidate keys: {C}, {A, B}, {D}.**
> {C} = {A,B,C,D} directly. {D} : D→C, then C→{A,B,D} — covers everything. {A,B} = {A,B,C,D} directly. All three are minimal.

Question C.4

R [A, B, C, D, E, F, G, H, I, J] with {A, B} → {C}, {A} → {D, E}, {B} → {F}, {F} → {G, H}, {D} → {I, J}. Find all candidate keys.

> **Working**
>
> **Candidate key: {A, B}.**
> {A, B} : A→{D,E} adds D,E; B→F adds F; {A,B}→C adds C; F→{G,H} adds G,H; D→{I,J} adds I,J — every attribute, and neither A nor B alone reaches the other's attributes.

Section D — Highest normal form

Question D.1

R [A, B, C, D, E, F] with {A, E} → {D}, {B, C} → {A}, {B} → {F}, {F} → {E}. Identify the highest normal form and justify.

> **Working**
>
> Only candidate key (CK) is {B, C} (from Question C.1). Highest normal form is **1NF**, because of the **partial dependency** {B} → {F} — B is a proper subset of the candidate key {B, C}, and F is a non-prime attribute, violating 2NF.

Question D.2

R [A, B, C, D, E] with {A} → {B, C, D, E}, {B} → {A, C, D, E}. Identify the highest normal form and justify.

Working

CKs are `{A}` and `{B}`. Highest normal form is **BCNF**, because the LHS of every FD (`A` and `B`) is itself a superkey.

Question D.3

`R [A, B, C, D, E, F, G]` with `{A} → {B, C, D}`, `{D} → {A}`, `{C} → {F, G}`. Identify the highest normal form and justify.

Working

CKs are `{A, E}` and `{D, E}`. Highest normal form is **1NF**. Decomposing `{A} → {B, C, D}` into `{A}→{B}`, `{A}→{C}`, `{A}→{D}`: `{A} → {B}` and `{A} → {C}` are both **partial dependencies**, since `A` is a proper subset of the candidate key `{A, E}` and `B`/`C` are non-prime attributes. (`{D} → {A}` is *not* a partial dependency, since `A` is a prime attribute — appearing in candidate key `{A, E}`.)

Question D.4

`R [A, B, C, D, E]` with `{A, B} → {C, E}`, `{D} → {A}`, `{A} → {D}`. Identify the highest normal form and justify.

Working

CKs are `{A, B}` and `{D, B}`. Highest normal form is **3NF** — there are no partial or transitive dependencies. It's not BCNF, though: the LHS of `{A} → {D}` (just `A`) is not a superkey, violating BCNF — but this is *not* a partial dependency, since `D` is a prime attribute (appears in candidate key `{D, B}`). The same reasoning applies symmetrically to `{D} → {A}`.

Case Study 8: Dirt Road Driving (Payroll System — Anomalies and FDs)

Practice for 2026-04-27-database-design-guidelines-and-functional-dependencies[5]. Dirt Road Driving's Director of Innovation, Peter Thompson, has asked for student help auditing their payroll/finance system's database schema — some staff feel the current design is inefficient.

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

The payroll schema

Note: this is Peter's *first* schema email — it's missing FDs for `EmployeeHistory`, `TripExpenseAllocations` and `TravelInsuranceHistory` (he sends a corrected, more complete version later, used in week11-tutorial-case-study-9-payroll-system[6]). For Section 2 below, those FDs must be identified from the sample data instead.

```
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

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

EmployeeHistory [employeeID, departmentCode, dateStarted, seniorityLevel, baseSalary, securit
EmployeeHistory.employeeID references Employee.id
EmployeeHistory.departmentCode references Department.code

TripExpenseAllocations [tripName, expenseType, quantity, organiser, startDate, endDate, locat
TripExpenseAllocations.organiser references Employee.id

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

Sample data (relevant excerpts):

```
AssetUse
employeeID  assetID  timestamp            useDuration  assetType     purchaseDate  insurance
```

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

```
2014        1200     20-12-2019 11:04:14   00:15:02     Vehicle       16-02-
2019    20,000
2014        7900     24-12-2019 18:54:01   01:12:52     Vehicle       23-10-
2019    20,000
2020        7901     01-01-2020 13:07:59   05:00:09     Power Tools   05-05-
2019    1,000
2014        1200     01-01-2020 14:47:08   02:45:36     Vehicle       16-02-
2019    20,000
```

Department

code	name	manager	buildingID	buildingName	buildingLocation	floor
MAK	Marketing	2020	0302	Dumpling Building	(-27.4907639, 152.9955379)	8
FIN	Finance	2021	0302	Dumpling Building	(-27.4907639, 152.9955379)	7
IT	Computering	2023	2023	Hotpot Building	(-27.4856679, 152.9898608)	2

EmployeeHistory

employeeID	departmentCode	dateStarted	seniorityLevel	baseSalary	securityLevel
2020	FIN	13-09-2019	Junior	70,000	Limited
2023	IT	23-09-2020	Junior	70,000	Limited
1919	IT	05-11-2019	Junior	70,000	Limited
2014	FIN	07-11-2019	Junior	70,000	Limited
2019	IT	07-11-2019	Junior	70,000	Limited
2020	MAK	09-11-2019	Executive	80,000	Full Access
2021	FIN	18-11-2019	Senior	70,000	Full Access
2022	IT	19-11-2019	Junior	70,000	Limited
2023	IT	07-01-2020	Executive	80,000	Full Access

TripExpenseAllocations

tripName	expenseType	organiser	startDate	endDate	location	a
UQ Partnership	Food	2023	02-02-2020	03-02-2020	Brisbane	200
No alcohol over $70						
UQ Partnership	Hotel	2023	02-02-2020	03-02-2020	Brisbane	500
Food service not included						
UQ Partnership	Transport	2023	02-02-2020	03-02-2020	Brisbane	100
Cannot use Uber or DiDi						
Ride-share Marketing Conf.	Hotel	2020	21-03-2020	28-03-2020	Sydney	500
Food service not included						
Ride-share Marketing Conf.	Transport	2020	21-03-2020	28-03-2020	Sydney	100
Cannot use Uber or DiDi						
Investor Meeting	Food	2021	27-03-2020	28-03-2020	Cairns	200
No alcohol over $70						

TravelInsuranceHistory

```
tripName                    insuranceLevel  description                                  ma
UQ Partnership              4               Local travel without unsafe circumstances
Ride-share Marketing Conf.  1               Any travel with unsafe/govt-
conflicting circs.   10,000
Investor Meeting            1               Any travel with unsafe/govt-
conflicting circs.   10,000
```

Section A — Anomalies

Provide and explain one example each of a modification, deletion, and insertion anomaly for the ***`AssetUse`*** *and* ***`Department`*** *tables. (100 words each — the exercise must show* practically* how each anomaly applies, not just restate the textbook definition.)*

AssetUse

> **i** Working
>
> **Modification anomaly**: updating the tuple `<2014, 1200, 20-12-2019 11:04:14, 00:15:02, "Vehicle", 16-02-2019, 20,000>` to `<2014, 1200, 20-12-2019 11:04:14, 00:15:02, "Vehicle", 18-03-2019, 20,000>` — since `assetID → purchaseDate`, and `assetID = 1200` also appears in another row (with `timestamp = 01-01-2020 14:47:08`), that other row's `purchaseDate` would *also* need updating to `18-03-2019`, otherwise the table becomes inconsistent about when asset `1200` was purchased.
> **Deletion anomaly**: deleting `<2020, 7901, 01-01-2020 13:07:59, 05:00:09, "Power Tools", 05-05-2019, 1,000>` — this is the *only* row recording asset `7901`, so deleting it loses that asset's `assetType` and `purchaseDate` entirely, even though those facts are logically about the asset, not about this particular use-event.
> **Insertion anomaly**: to record a new asset that hasn't been used yet, we'd need to insert `<null, 2200, null, null, "Power Tools", 29-11-2019, 1,000>` — but `employeeID` and `timestamp` form the primary key, so a `null` primary key violates entity integrity. A new asset can't be entered until *someone* has used it.

Department

> **i** Working
>
> **Modification anomaly**: updating `<"FIN", "Finance", 2021, 0302, "Dumpling Building", (-27.4907639, 152.9955379), 7>`'s `buildingLocation` alone — since `buildingID → buildingLocation`, the *other* row with `buildingID = 0302` ("MAK") would also need updating, otherwise the two rows disagree about where building `0302` is.
> **Deletion anomaly**: deleting `<"IT", "Computering", 2023, 2023, "Hotpot Building", (-27.4856679, 152.9898608), 2>` — this is the only row referencing `buildingID = 2023`, so deleting it loses the `buildingName` and `buildingLocation` facts for that building.
> **Insertion anomaly**: recording a new building (`buildingID = 2013`, "Peking Duck Building") that isn't yet occupied by any department requires inserting `<null, null, null, 2013, "Peking Duck Building", (-27.566544, 152.917845), null>` — a `null` primary key (`code`), which violates entity integrity.

Section 2 — Functional dependencies and highest normal form

Based on the sample data above, identify all non-trivial functional dependencies (beyond the given primary keys) for the tables below, and their highest normal form. As the primary keys are given in the schema, ignore FDs for the primary key itself.

EmployeeHistory

> **i** Working
>
> **FD**: `seniorityLevel → baseSalary, securityLevel`
> Every row sharing a `seniorityLevel` agrees on `baseSalary` and `securityLevel` — all six `Junior` rows have `{70,000, Limited}`; both `Executive` rows have `{80,000, Full Access}`.
> **Highest normal form: 2NF.** This FD is a **transitive dependency** — its left-hand side (`seniorityLevel`) is not a superkey, and its right-hand side contains non-prime attributes, violating 3NF. (The LHS is also not a subset of the candidate key `{employeeID, departmentCode}`, so it doesn't violate 2NF.)

TripExpenseAllocations

> **i** Working
>
> **FDs**: `tripName → startDate, endDate, location, organiser`; `expenseType → allowance, restrictions`
> All three "UQ Partnership" rows agree on `organiser`/dates/`location`; `expenseType = 'Food'` (rows for "UQ Partnership" and "Investor Meeting") always has `allowance = 200` and the same restriction text.
> **Highest normal form: 1NF.** The first FD is a **partial dependency** — its left-hand side (`tripName`) is a prime attribute (part of the candidate key `{tripName, expenseType}`) but not itself a superkey, and its right-hand side contains non-prime attributes, violating 2NF.

TravelInsuranceHistory

> **i** Working
>
> **FD**: `insuranceLevel → description, maxCoverage`
> Both rows with `insuranceLevel = 1` ("Ride-share Marketing Conference" and "Investor Meeting") share the same `description` and `maxCoverage = 10,000`.
> **Highest normal form: 1NF.** Same shape of violation as above — the left-hand side (`insuranceLevel`) is a prime attribute but not a superkey, and the right-hand side contains non-prime attributes, violating 2NF.

Reference material

Attribute Closure and Finding Keys

Attribute closure is the one algorithm the whole of Module 4 runs on. Every later question — is this a key? is this relation in BCNF? is this FD redundant? does this FD survive a decomposition? — is answered by computing a closure. Introduced in 2026-04-27-database-design-guidelines-and-functional-dependencies[7].

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

Definition

Given a relation R, a set of FDs[8] F, and a set of attributes X R, the **closure of X** (written X) is the set of all attributes functionally determined by X under F:

$$X^{+} = \{\, A \in R \;:\; F \models X \rightarrow A \,\}$$

Computing X is the practical substitute for computing F — which would need Armstrong's Axioms and is explicitly out of scope for this course (see functional-dependencies[9]) — and it answers the only question we ever actually ask of F .

The algorithm

Start with X itself, then repeatedly apply any FD whose LHS is already fully contained in the set, until nothing more can be added:

```
X+ := X;
repeat
    old X+ := X+;
    for each FD Y → Z in F:
        if Y  X+ then X+ := X+  Z;
until (old X+ = X+);
```

Two things to watch:

- The LHS must be **entirely** inside X before the FD can fire — a partial match does nothing. {A, D} = {A, D} under {A,B} → E, C → {B,E}, {E,D} → F, because none of the three LHSs is a subset of {A, D}, so the closure never grows past its starting point.
- It's a **fixpoint** loop, not a single pass: an FD that couldn't fire earlier may fire once a later FD has added its LHS. Keep sweeping F until a full pass adds nothing.

Worked examples

Employee (id, level, salary, name) with F = {id → level, level → salary, id → name}:

[8] courses/infs1200/functional-dependencies.pdf
[9] courses/infs1200/functional-dependencies.pdf

Step	{id}	Using
0	{id}	— (start)
1	{id, level}	id → level
2	{id, level, salary}	level → salary
3	{id, level, salary, name}	id → name

{id} covers every attribute, so id is a superkey — and since it's a single attribute it's trivially minimal, so it's a key.

A chained closure — R [pNumber, pName, pLocation, dNum, dName, mgrSSN, mgrStartDate] with F = {pNumber → {pName, pLocation, dNum}, dNum → {dName, mgrSSN, mgrStartDate}}:

```
{pNumber}+ = {pNumber}
{pNumber}+ = {pNumber, pName, pLocation, dNum}                    using FD1
{pNumber}+ = {pNumber, pName, pLocation, dNum, dName, mgrSSN,
              mgrStartDate}                                       using FD2
```

The second FD only became applicable *after* the first added dNum — the fixpoint loop in action. The closure is all of R, so pNumber is a key.

A composite LHS — showing {sName, pNum} is a superkey for SupplierPart (sName, city, status, pNum, pName, qty) with F = {sName → city, city → status, pNum → pName, {sName, pNum} → qty}:

```
{sName, pNum}+ = {sName, pNum}
{sName, pNum}+ = {sName, pNum, city}                        using sName → city
{sName, pNum}+ = {sName, pNum, city, status}                using city → status
{sName, pNum}+ = {sName, pNum, city, status, pName}         using pNum → pName
{sName, pNum}+ = {sName, pNum, city, status, pName, qty}    using {sName,pNum} → qty
```

Showing it is a **key** rather than merely a superkey takes a second half: every proper subset must fail. {sName} = {sName, city, status} and {pNum} = {pNum, pName}, neither of which is all of SupplierPart, so nothing smaller works and {sName, pNum} is minimal. "Cover all attributes, then show each proper subset doesn't" is the standard two-part answer whenever a question says *key* rather than *superkey*.

Key vocabulary

Everything below is defined *relative to a set of FDs*, and each is decided by a closure computation. See relational-keys[10] for how the same terms are used in the relational model generally.

[10] courses/infs1200/relational-keys.pdf

Term	Definition	Test
Superkey	A set of attributes for which no two tuples can share the same values	`S = R`
(Candidate) key	A *minimal* superkey — removing any attribute breaks the superkey property	`S = R` **and** no proper subset `S'` `S` has `S' = R`
Primary key	One candidate key, chosen as *the* key (only one per relation)	— (a design choice)
Prime attribute	An attribute belonging to *any* candidate key — not just the primary key	Appears in at least one candidate key
Non-prime attribute	Belongs to no candidate key	—

Minimal shortest: a relation can have several candidate keys of different sizes, and every superkey contains at least one candidate key. Prime/non-prime is the distinction the 2NF and 3NF tests turn on, so it has to be computed against *all* candidate keys, not the primary one — see normal-forms[11].

Finding all candidate keys

For `n` attributes there are `2` subsets in the worst case, but three observations cut that down to something you can do by hand:

1. **An attribute that never appears on the RHS of any FD must be in every key.** Nothing can ever derive it, so no set that omits it can have full closure. This is usually the fastest way in.
2. **Once `S` is known to be a key, skip every superset of `S`.** Those are superkeys, not candidate keys.
3. **Expect more than one key.** Stop only when you've checked every minimal combination, not at the first hit.

Worked example — `R [A, B, C, D, E, F]` with the FDs `A → {B, C}`, `{C,D} → E`, `{A,C} → E`, `B → D`, `E → {A, B}` (from week10-tutorial-applied-class-9-functional-dependencies[12], Section C). Note the attribute named `F` here has nothing to do with the usual name for the FD set:

The attribute `F` never appears on any RHS, so by tip 1 it belongs to every key. Among the remaining attributes the FDs form a tightly coupled cluster: `A` alone reaches `B, C`, then `D` (via `B → D`), then `E`; `E` alone reaches `A, B` and so everything `A` does; `{B, C}` reaches `D` then `E` then `A`;

[11] courses/infs1200/normal-forms.pdf
[12] courses/infs1200/week10-tutorial-applied-class-9-functional-dependencies.pdf

{C, D} reaches E then A, B. So there are four minimal ways to cover the cluster, each needing F added:

Candidate keys: {A, F}, {E, F}, {B, C, F}, {C, D, F}.

A negative example — R [A, B, C, D, E, F] with {A,B} → E, C → {B, E}, {E,D} → F. Is {A, B} a key? {A,B} = {A, B, E} — misses C, D, F, so it isn't even a superkey. Is {A, B, C}? {A,B,C} = {A, B, C, E} — still misses D, F. The key is {A, C, D}: C → {B, E} adds B, E, then {E,D} → F adds F, giving all six attributes, and no proper subset gets there.

Superkey key — with B → C, C → B, D → {A, B, C} on R [A, B, C, D], {B, D} has full closure but is *not* a candidate key, because D alone already does (D = R). Every superset of D is a superkey; only D itself is the key.

Where closures get used

Task	Closure computed
Is S a superkey?	S = R?
Is S a candidate key?	S = R and S' R for every S' S
Does R satisfy BCNF for X → A?	Is X = R? (see normal-forms[13])
Is an LHS attribute B redundant in X → A?	(X - {B}) = X ? (see minimal-cover[14])
Is the whole FD X → A redundant?	X computed *without* it still contains A?
Does X → Y hold in a decomposed relation S?	X Y S and Y X (see [decomposition])

See also

- functional-dependencies[15] · normal-forms[16] · minimal-cover[17] · [decomposition] · 3nf-synthesis[18]
- 2026-04-27-database-design-guidelines-and-functional-dependencies[19] — Module 4, part 1
- week10-tutorial-applied-class-9-functional-dependencies[20] — Sections B and C are pure closure/candidate-key drill

[13] courses/infs1200/normal-forms.pdf
[14] courses/infs1200/minimal-cover.pdf
[15] courses/infs1200/functional-dependencies.pdf
[16] courses/infs1200/normal-forms.pdf
[17] courses/infs1200/minimal-cover.pdf
[18] courses/infs1200/3nf-synthesis.pdf
[19] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[20] courses/infs1200/week10-tutorial-applied-class-9-functional-dependencies.pdf

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.

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`:

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

```
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, 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
```

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

```
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` $\subseteq$ `F`, so `(Fi)`$^+$ $\subseteq$ `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` $\subseteq$ `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.

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
```

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

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.

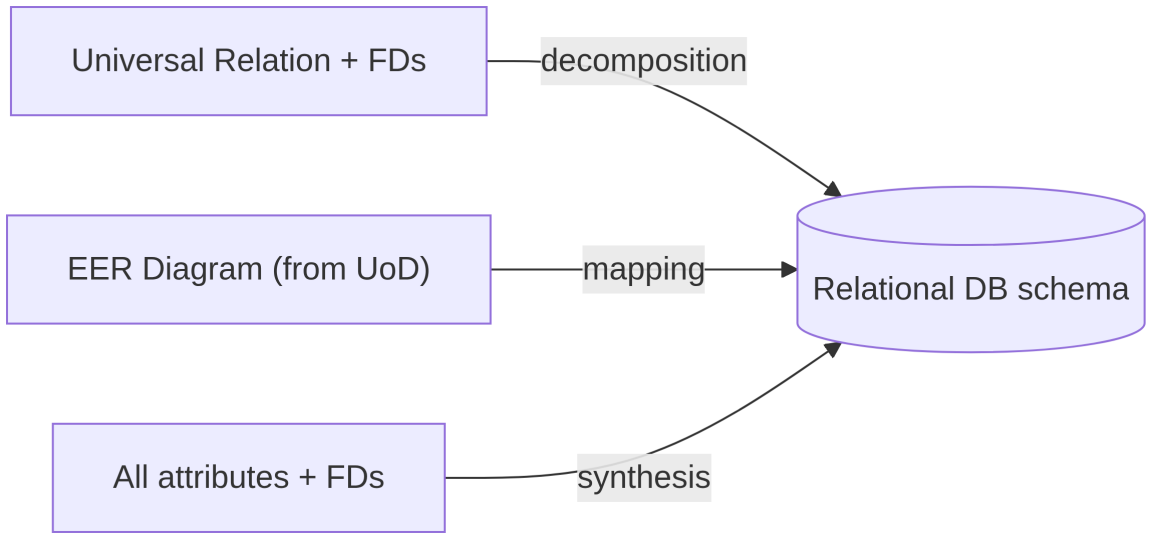
BCNF vs 3NF:

	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]):



- **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.

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

- **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 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}`.

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

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.

[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

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

Functional Dependencies

A **functional dependency (FD)** is the formal way to state a business rule of the form "if you know *this*, you know *that*". It's the unit of information every design-theory tool in this course consumes: closures, keys, normal forms and both schema-design algorithms are all defined in terms of a set of FDs. Introduced in 2026-04-27-database-design-guidelines-and-functional-dependencies[40].

The motivating question is the one left hanging by update-anomalies[41]: how can we be *sure* every employee at the same level earns the same salary, rather than it merely happening to be true of today's rows? By declaring `level → salary` — "level determines salary".

Formal definition

An FD `X → Y` holds on relation `R` if, for every legal instance `r` of `R` and every pair of tuples $t_1, t_2 \in r$:

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y]$$

[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
[40] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[41] courses/infs1200/update-anomalies.pdf

i.e. any two tuples that agree on `X` must also agree on `Y`. `X` is the **left-hand side (LHS)** or *determinant*, `Y` the **right-hand side (RHS)**. Both are *sets* of attributes, so `{sName, pNum} → qty` is a perfectly ordinary FD; single-attribute sets are written without braces.

An FD is a **constraint between two attribute sets** — it restricts which tuples may legally appear together in an instance of `R`, exactly as a key or referential-integrity constraint does (see integrity-constraints[42]). A key is in fact a special case: `X` is a superkey of `R` precisely when `X → R` holds, which is how relational-keys[43]'s "minimal set that uniquely identifies" becomes something you can *compute* rather than assert — see attribute-closure[44].

An FD is a statement about *all* instances

This is the single most important thing about FDs, and the thing exam questions probe most often:

> You can check whether a given instance **violates** an FD, but you can never **prove** an FD holds by looking at one instance.

FDs come from the semantics of the application — the Universe of Discourse and its business rules — not from reverse-engineering a data sample. A sample can only ever rule FDs *out*.

Ruling one out: given `R [A, B, C, D]` with rows `(1,2,3,4)`, `(2,3,4,6)`, `(6,7,8,9)`, `(1,3,4,5)` — `B → D` **cannot** be true, since rows 2 and 4 both have `B = 3` but disagree on `D` (`6` vs `5`). `B → C` and `D → B` are still *possible* — meaning only "not yet contradicted", never "proven".

Reading candidate FDs off a sample instance

Case-study questions do ask you to *identify* the FDs a sample table suggests. The mechanical procedure is: for each candidate `X → Y`, group the rows by their `X` value; if any group disagrees on `Y`, the FD is dead, otherwise it survives as a **potential** FD. Two traps:

- An `X` value that never repeats makes `X → Y` **trivially survive** — there is no pair of rows to contradict it. In week10-tutorial-applied-class-9-functional-dependencies[45] Section A, `A → B`, `A → C` and `{A, B} → C` all "hold" only because `A` is distinct in every row. That's evidence of nothing.

[42] courses/infs1200/integrity-constraints.pdf
[43] courses/infs1200/relational-keys.pdf
[44] courses/infs1200/attribute-closure.pdf
[45] courses/infs1200/week10-tutorial-applied-class-9-functional-dependencies.pdf

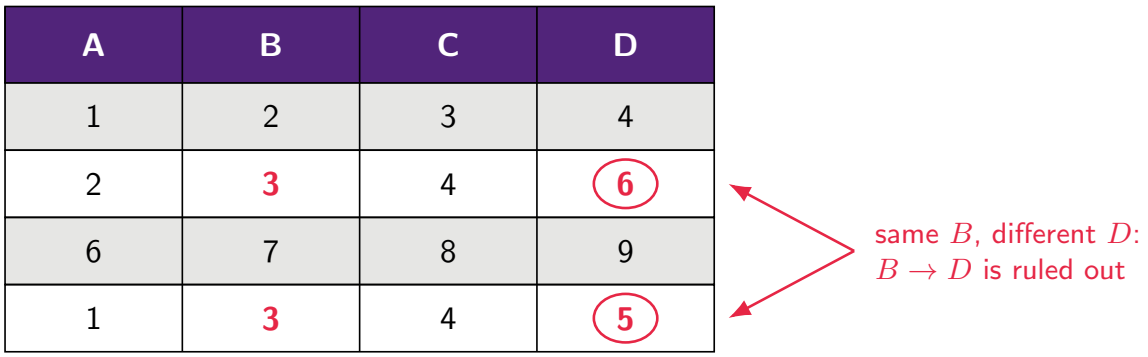


Figure 2: The only thing an instance can do to an FD. Rows 2 and 4 agree on `B` and disagree on `D`, so `B → D` is refuted outright; nothing in the table can establish `B → C` or `D → B`, which merely survive un-contradicted. The crimson ellipses mark the two `D` values that clash.

- The surviving FDs still have to make *sense*. In week10-tutorial-case-study-8-dirt-road-driving[46], the FD worth reporting on `EmployeeHistory` is `seniorityLevel → baseSalary, securityLevel` — every `Junior` row has `{70,000, Limited}` and every `Executive` row has `{80,000, Full Access}` — because seniority genuinely determines pay band in the UoD, not merely in the sample.

Trivial and non-trivial FDs

- A **trivial FD** holds regardless of `F`, because the LHS already contains the RHS — e.g. `A → A`, `{A, B, C} → {A, B}`.
- A **non-trivial FD** depends on the specific `F` — e.g. `A → B`.

Every normal-form test is stated over *non-trivial* FDs only, since a trivial FD can never be a design flaw.

Explicit and implicit FDs; the closure of F

Given a set `F` of *explicit* FDs, further *implicit* FDs follow: from `id → level` and `level → salary` we can infer `id → salary`. Implicit FDs are also called **inferred** FDs. The notation

$$F \models X \to Y$$

[46] courses/infs1200/week10-tutorial-case-study-8-dirt-road-driving.pdf

means `X → Y` can be inferred from `F` — equivalently, "`Y` can be derived from `X` under `F`", or "`X` implies `Y` under `F`".

The **closure of F**, written `F⁺`, is the set of *all* FDs — trivial and non-trivial — implied by `F`.

Armstrong's axioms and what is actually examinable

`F⁺` can be computed by applying **Armstrong's Axioms** as inference rules, but doing so is *difficult and time consuming*, and it is **outside the scope of this course**. The individual rules — reflexivity, augmentation, transitivity, and the derived union, decomposition and pseudotransitivity rules — are outside it too, so there is nothing to memorise and no question can ask you to name or apply one.

What replaces them is attribute-closure[47] (`X⁺`), which answers the only question the rest of the module ever asks of `F`: given `X`, which attributes are determined? Every inference you actually need — the transitive chain `id → level → salary`, say — falls out of one closure computation, and *that* is the examinable skill.

`F⁺` still matters conceptually, because every normal-form test is phrased "for every non-trivial `X → A` **in** `F⁺`" — the implied FDs count too. Missing an implicit FD is the classic way to declare a relation BCNF-compliant when it isn't; in practice you compute the closures first, spot that `F⁺ ⊇ A → C`, add `A → C` to `F`, and *then* run the algorithm. See [decomposition].

Why FDs create redundancy

FDs are also the precise account of *where redundancy comes from*. In `R [A, B, C]` with no FDs at all, no value constrains any other and there is no redundancy. Add `A → B`, and any two tuples sharing an `A` value are now *forced* to repeat the same `B` value. That forced repetition is exactly what normalisation identifies and removes — which is why the normal-form tests in normal-forms[48] are all conditions on the LHS of an FD relative to the keys.

A running instance makes the mechanism concrete. `Employee [ID, name, level, salary]` with `level → salary` stores `Developer, 60,000` twice and `Manager, 700,000` twice — not by accident but *because* the FD forbids anything else. Project the relation onto `[ID, name, level]` and `[level, salary]` and each level's salary is recorded exactly once, with nothing lost: the natural join on `level` rebuilds the original rows. The two halves are named `StaffAppointment [ID, name, level]` and `StaffIncome [level, salary]` in the 3NF treatment; see normal-forms[49] and [decomposition].

[47] courses/infs1200/attribute-closure.pdf
[48] courses/infs1200/normal-forms.pdf
[49] courses/infs1200/normal-forms.pdf

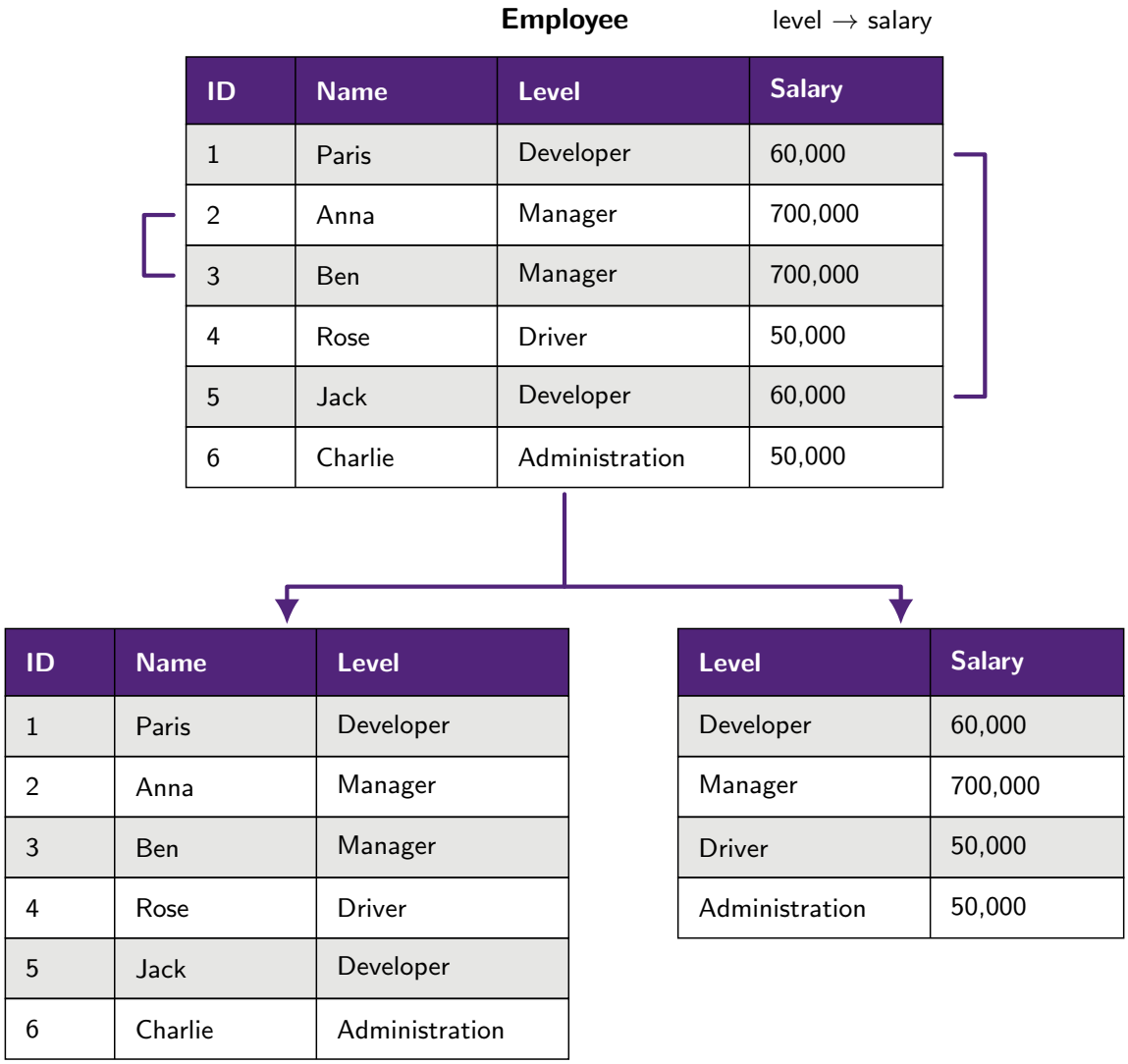


Figure 3: An FD is a redundancy generator, and the split is the cure. `level → salary` forces every Developer row to carry `60,000` and every Manager row `700,000` — the purple brackets mark the two groups the FD ties together. Splitting off a `[level, salary]` table records each level's salary once; the natural join on `level` rebuilds the six original rows.

The three anomalies that repetition causes — and which the split fixes — are catalogued in update-anomalies[50].

See also

- attribute-closure[51] · normal-forms[52] · minimal-cover[53] · [decomposition]
- update-anomalies[54] — the symptoms FDs explain
- 2026-04-27-database-design-guidelines-and-functional-dependencies[55] — Module 4, part 1
- week10-tutorial-applied-class-9-functional-dependencies[56] — FD identification and closure 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

[50] courses/infs1200/update-anomalies.pdf
[51] courses/infs1200/attribute-closure.pdf
[52] courses/infs1200/normal-forms.pdf
[53] courses/infs1200/minimal-cover.pdf
[54] courses/infs1200/update-anomalies.pdf
[55] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[56] courses/infs1200/week10-tutorial-applied-class-9-functional-dependencies.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`
2NF	Partial dependencies (removes *some* anomalies)	`X` is **not** a proper subset of a candidate key, **or** `A` is a prime attribute
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.

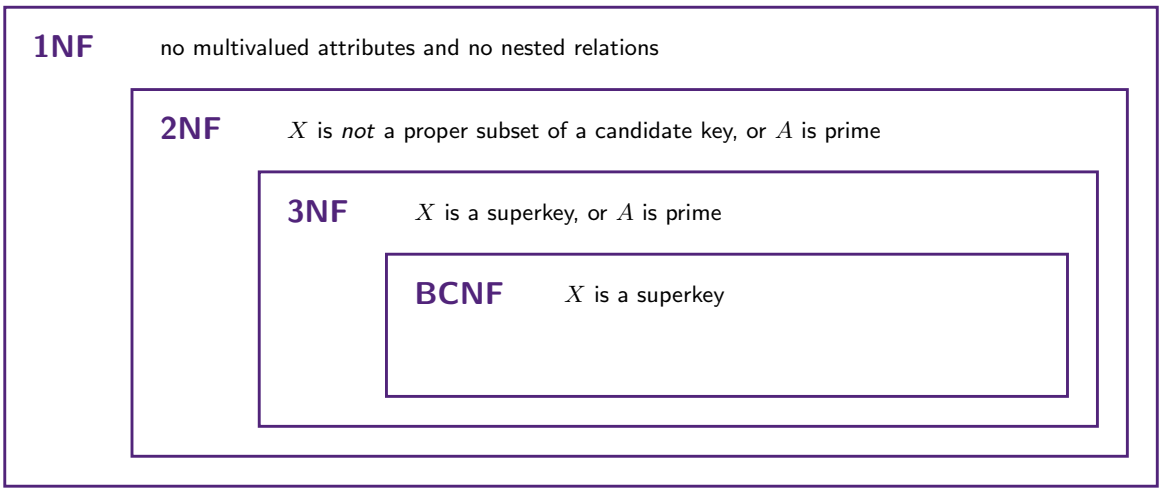


Figure 4: The four normal forms nest: each test is the previous one with an escape hatch removed, so a relation in an inner band is automatically in every band outside it. Each band carries its one-line test, and every test reads *for every non-trivial* `X → A` *in* `F`.

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] for dependency preservation and the BCNF algorithm, and 3nf-synthesis[63] for the algorithm that guarantees the 3NF side of the table.

Determining the highest normal form

1. Compute **all** candidate keys (attribute-closure[64]); 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/infs1200/3nf-synthesis.pdf
[64] 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[65]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.

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

See also

- functional-dependencies[66] · attribute-closure[67] · [decomposition] · 3nf-synthesis[68] · minimal-cover[69]
- update-anomalies[70] — the anomalies each normal form is removing
- 2026-05-04-normalisation-and-relational-database-schema-design[71] — Module 4, part 2
- week10-tutorial-applied-class-9-functional-dependencies[72] · week10-tutorial-case-study-8-dirt-road-driving[73] — highest-normal-form practice

NULLs in SQL

`NULL` is the special value a relational attribute takes when there is no ordinary value to store — it represents an **inapplicable, unknown or missing** value rather than zero or the empty string. Because a null is not a value in the attribute's domain, it needs its own handling everywhere it appears: in constraints, in `WHERE` conditions, and in the results a query produces. Defined in 2026-02-23-conceptual-database-design-and-the-er-model[74], constrained in 2026-03-09-the-relational-model-and-integrity-constraints[75], and handled in SQL from 2026-03-30-basic-sql-ddl-and-dml[76] onwards.

Which constraints permit a null

Declaration	Nulls allowed?
`PRIMARY KEY`	**No** — a primary key value is unique *and* not null; for a composite key, no part of it may be null (entity integrity)
`FOREIGN KEY`	**Yes** — a foreign key value must either be null, or match an existing row of the referenced relation (referential integrity)
`UNIQUE`	**Yes** — the value must be unique *or* null; this is the one difference from a primary key
`NOT NULL`	**No** — a value must be supplied

[66] courses/infs1200/functional-dependencies.pdf
[67] courses/infs1200/attribute-closure.pdf
[68] courses/infs1200/3nf-synthesis.pdf
[69] courses/infs1200/minimal-cover.pdf
[70] courses/infs1200/update-anomalies.pdf
[71] courses/infs1200/2026-05-04-normalisation-and-relational-database-schema-design.pdf
[72] courses/infs1200/week10-tutorial-applied-class-9-functional-dependencies.pdf
[73] courses/infs1200/week10-tutorial-case-study-8-dirt-road-driving.pdf
[74] courses/infs1200/2026-02-23-conceptual-database-design-and-the-er-model.pdf
[75] courses/infs1200/2026-03-09-the-relational-model-and-integrity-constraints.pdf
[76] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf

Declaration	Nulls allowed?

See integrity-constraints[77] for the underlying entity and referential integrity rules, relational-keys[78] and foreign-keys[79] for what the keys themselves are, and sql-ddl[80] for how each is declared.

The nullable foreign key is what makes several mappings work at all: an employee with no manager, or a department with no manager assigned yet, is represented by a null foreign key rather than by a missing row. It is also why er-to-relational-mapping[81] prefers to hang a 1:1 foreign key off the *total*-participation side — the other side would be null for almost every row.

Testing for a null

`NULL` is tested with `IS`, never with `=`. `IS` sits alongside `LIKE`, `IN` and `BETWEEN` in the course's catalogue of things a `WHERE` search condition may contain (sql-select-statement[82]):

```
SELECT * FROM Employee WHERE dNum IS NULL;
SELECT * FROM Employee WHERE dNum IS NOT NULL;
```

This shows up inside subqueries as well as at the top level — here, restricting the result to employees whose department currently has a manager assigned:

```
SELECT name, dNum
FROM   Employee e
WHERE  EXISTS (
    SELECT * FROM Department d
    WHERE  d.dNumber = e.dNum AND d.mgrSSN IS NOT NULL
);
```

An explicit `IS NOT NULL` is also the usual guard before grouping or building a derived value out of several columns, so that rows with missing components are excluded rather than silently contributing:

[77] courses/infs1200/integrity-constraints.pdf
[78] courses/infs1200/relational-keys.pdf
[79] courses/infs1200/foreign-keys.pdf
[80] courses/infs1200/sql-ddl.pdf
[81] courses/infs1200/er-to-relational-mapping.pdf
[82] courses/infs1200/sql-select-statement.pdf

```sql
-- Last names shared by more than one employee, ignoring unknown names
SELECT   DISTINCT LastName FROM Employee
WHERE    LastName IS NOT NULL
GROUP BY LastName
HAVING   COUNT(FirstName) > 1;

-- Build FirstName-LastName-yyyy only where all three parts exist
SELECT CONCAT(FirstName, '-', LastName, '-', YEAR(DOB))
FROM   Employee
WHERE  FirstName IS NOT NULL
  AND  LastName  IS NOT NULL
  AND  DOB       IS NOT NULL;
```

The limits of what the course specifies

Module 3 treats nulls **operationally** rather than logically. It gives the `IS` operator, the constraint rules above, and the places a null can come from — and it stops there. The lectures do not develop a three-valued logic of `TRUE/FALSE/UNKNOWN`, do not state how an aggregate function treats the nulls inside a group, and do not cover what happens when a `NOT IN` subquery returns a null among its values. Those questions are simply not settled by the material, so don't reason from a rule the course never gave you.

What the course *does* teach is the defensive habit: write the `IS NOT NULL` guard explicitly wherever a null could reach a comparison, a grouping, or a derived value, rather than relying on the DBMS to do something sensible with it. Every example above follows that habit.

Where nulls come from

Not every null is data someone failed to enter — several are created by the schema or the query itself:

Source	Why
A genuinely inapplicable/unknown/missing value	The original meaning of a null
`ALTER TABLE … ADD <column>`	Every existing row gets a null in the new column — which is exactly why the new column cannot be declared `NOT NULL` (sql-ddl[83])

[83] courses/infs1200/sql-ddl.pdf

Source	Why
`ON DELETE SET NULL`	The referential action deliberately clears the link instead of deleting the dependent rows (update-operations-and-constraint-violations[84])
An **outer join**	Unmatched rows are padded with nulls in the columns of the table that had no match (sql-joins[85])
Access-control masking	A cell above a user's clearance is presented to them as null rather than withheld (database-access-control[86])

The last two matter when reading results: a null in the output of a `LEFT JOIN` means "no matching row existed", which is information about the *relationship*, not about the row's stored data.

```
-- Departments with no manager still appear, with NULL for the name
SELECT    D.dName, E.name
FROM      Department AS D
LEFT JOIN Employee AS E ON D.mgrSSN = E.ssn;
```

Nulls as a design smell

A schema that produces many nulls is usually telling you something. Reducing the number of null values in tuples is guideline 3 of the informal design guidelines behind normalisation (update-anomalies[87]) — an attribute that is null for most rows of a relation often belongs in a separate relation instead. See 2026-04-27-database-design-guidelines-and-functional-dependencies[88] and normal-forms[89].

Update Anomalies and Informal Design Guidelines

Before any of the formal machinery — functional dependencies, closures, normal forms — there are four *informal* measures of relational schema quality, and one recurring symptom (**update**

[84] courses/infs1200/update-operations-and-constraint-violations.pdf
[85] courses/infs1200/sql-joins.pdf
[86] courses/infs1200/database-access-control.pdf
[87] courses/infs1200/update-anomalies.pdf
[88] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[89] courses/infs1200/normal-forms.pdf

anomalies) that a bad schema produces. Introduced in 2026-04-27-database-design-guidelines-and-functional-dependencies[90].

The four guidelines

#	Guideline	What it protects against
1	Make sure the **semantics of the attributes are clear** in the schema	A relation whose meaning can't be stated in one sentence
2	**Reduce redundant values** in tuples	Insertion, deletion and modification anomalies
3	**Reduce null values** in tuples	Wasted space, ambiguous meaning, broken entity integrity
4	**Disallow spurious tuples** — don't allow lossy joins	Information silently *invented* by a join

They're informal because they're qualitative: they tell you a schema *looks* wrong, but they don't tell you which normal form it satisfies or how to fix it. That's what functional-dependencies[91] and normal-forms[92] add.

Guideline 1 — one relation, one meaning

Design each relation so that it is easy to explain its meaning. Two halves: **do** use meaningful names, and **don't** combine attributes from multiple entity types and relationship types into a single relation — that makes the meaning of an entity type confusing, and it leads to redundancy.

Take a schema with one relation per entity type, joined by a foreign key each way (see foreign-keys[93]):

```
EMPLOYEE [Ename, Ssn, Bdate, Address, Dnumber]
    P.K. Ssn
    F.K. Dnumber references DEPARTMENT.Dnumber

DEPARTMENT [Dname, Dnumber, Dmgr_ssn]
```

[90] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[91] courses/infs1200/functional-dependencies.pdf
[92] courses/infs1200/normal-forms.pdf
[93] courses/infs1200/foreign-keys.pdf

```
P.K. Dnumber
F.K. Dmgr_ssn references EMPLOYEE.Ssn
```

versus lumping both entity types into one relation:

```
EMP_DEPT [Ename, Ssn, Bdate, Address, Dnumber, Dname, Dmgr_ssn]
```

`EMP_DEPT` can't be described without the word "and" — it's about employees *and* the departments they work for — and every employee of a department repeats that department's name and manager. Guideline 1 is the informal statement of what er-to-relational-mapping[94] does automatically when a schema is derived from an ER diagram: one relation per entity type.

Guideline 2 — the three update anomalies

Design the base relation schemas so that **no insertion, deletion or modification anomalies occur** in the relations. Where an anomaly can't be designed out, the fallback is procedural: every application that touches the database must update the relations in a way that doesn't compromise its integrity — which is a far weaker guarantee than a schema that makes the anomaly impossible.

The underlying design goal is to minimise the storage space base relations occupy, and how attributes are grouped into relation schemas has a large effect on that. An incorrect grouping costs more than space, though: it causes update anomalies, and those produce inconsistent data or outright loss of data.

Take `Employee [id, name, level, salary]`, where salary is fixed per level in the real world (Developer = 60,000, Manager = 700,000, Driver = 50,000, Administration = 50,000):

id	name	level	salary
1	Paris	Developer	60,000
2	Anna	Manager	700,000
3	Ben	Manager	700,000
4	Rose	Driver	50,000
5	Jack	Developer	60,000
6	Charlie	Administration	50,000

The level→salary fact is stored once per *employee* rather than once per *level*, and that single design mistake produces three distinct failure modes:

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

Anomaly	What goes wrong	Example
Modification	The same fact is stored in several tuples, so updating one copy leaves the others contradicting it	Raising Paris's salary makes the two Developer rows disagree about what a Developer earns
Deletion	Deleting a tuple destroys an unrelated fact that happened to be stored only there	Deleting Charlie loses the fact that Administration pays 50,000 — he was the only one
Insertion	A fact can't be recorded until some *other*, unrelated entity exists to hang it on; or inserting one introduces a contradiction	A Cook's salary can't be recorded until someone is actually employed as a Cook; and inserting a new Developer with a different salary contradicts the existing ones

These aren't only a textbook concern. A widely reported 2021 issue with Australia's vaccine certificate system came from essentially this class of problem — state vaccination staff recorded details in a way that didn't precisely match federal records, so contradicting datasets failed to reconcile automatically.

Arguing an anomaly concretely

Exam and case-study answers want the *mechanism*, not the definition restated. The pattern that works: name the tuple, name the FD it duplicates, and name the other tuple(s) that now disagree. From week10-tutorial-case-study-8-dirt-road-driving[95], on `AssetUse [employeeID, assetID, timestamp, useDuration, assetType, purchaseDate, insuranceValue]` with `assetID → assetType, purchaseDate`:

- **Modification** — changing the `purchaseDate` of the row `<2014, 1200, 20-12-2019 11:04:14, ...>` requires changing the *other* row with `assetID = 1200` too, or the table now disagrees about when asset 1200 was bought.
- **Deletion** — the row for asset `7901` is the only one recording that asset, so deleting that *use event* also destroys the asset's type and purchase date, which are facts about the asset, not the event.
- **Insertion** — recording an asset nobody has used yet means inserting `<null, 2200, null, null, "Power Tools", 29-11-2019, 1,000>`; since `employeeID` and `timestamp` are part of the primary key, that's a null in the key and violates entity integrity (see integrity-constraints[96]).

[95] courses/infs1200/week10-tutorial-case-study-8-dirt-road-driving.pdf
[96] courses/infs1200/integrity-constraints.pdf

Spotting anomalies on an abstract relation

The drill version gives a relation, its FDs and an instance, and asks which of several operations is **not** an anomaly. Take `R [A, B, C, D]` with primary key `B` and `D → {A, C}`:

A	B	C	D
1	4	2	5
2	3	4	3
1	1	2	5

Operation	Verdict
Delete `<2, 3, 4, 3>`	**Deletion anomaly** — the only row with `D = 3`, so the fact `3 → {2, 4}` is lost
Insert `<3, 5, 3, 3>`	**Insertion anomaly** — asserts `3 → {3, 3}` while the table already says `3 → {2, 4}`
Modify `<1, 1, 2, 5>` to `<1, 2, 2, 5>`	**No anomaly** — only the key value changes; `5 → {1, 2}` still holds everywhere
Insert `<1, null, 2, 4>`	**Insertion anomaly** — null in the primary key `B`
Modify `<1, 1, 2, 5>` to `<1, 2, 3, 5>`	**Modification anomaly** — now `5 → {1, 2}` and `5 → {1, 3}`

Every verdict is the same one test: does the operation leave two tuples that agree on an FD's LHS but disagree on its RHS (or put a null in the primary key)? Anomalies are FD violations waiting to happen, which is why functional-dependencies[97] come next in the module.

Guideline 3 — reduce nulls

As far as possible, avoid placing attributes in a base relation whose values may be null. There are three reasons, worth keeping distinct because they fail in different ways:

Problem	Consequence
Nulls **waste storage space**	Every row pays for a column most rows can't use
Nulls **introduce ambiguity**	The reader can't tell "not applicable", "unknown" and "not recorded" apart

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

Problem	Consequence
Nulls **cannot be used for comparison**	Predicates and aggregates over the column stop behaving as expected — see sql-nulls[98]

Where nulls are unavoidable, the requirement is that they apply in **exceptional cases only** — a column that is null for most rows is the signal that those attributes belong in a separate relation.

Two concrete cases: flattening a repeating group into fixed columns (`item1`, `item2`, `item3`, …) to reach 1NF[99] leaves every short order carrying nulls; and the insertion anomaly above is really a null-in-the-primary-key problem, which integrity-constraints[100] forbids outright.

Guideline 4 — join on keys

Design relation schemas so they can be (relationally) joined with equality conditions on attributes that are either **primary keys or foreign keys** (foreign-keys[101]), in a way that guarantees no **spurious tuples** are generated. Joining on a non-key attribute lets a join manufacture rows that were never in the original relation — see [decomposition] for the worked lossy-join example, the lossless-join property, and how to test a proposed decomposition for it.

Decomposition as the fix

The cure for all three anomalies is to split the relation so each fact is stored once — a **decomposition**, defined and formalised in [decomposition].

For the `Employee` example, the anomalies come from `level → salary` being stored per-employee. Splitting on exactly that dependency:

```
Employee [id, name, level]
Employee.level references LevelSalary.level

LevelSalary [level, salary]
```

[98] courses/infs1200/sql-nulls.pdf
[99] courses/infs1200/normal-forms.pdf
[100] courses/infs1200/integrity-constraints.pdf
[101] courses/infs1200/foreign-keys.pdf

removes all three anomaly types at once — modifying a level's salary touches one row; deleting an employee can't lose a level's salary; a new level's salary can be recorded before anyone holds that role.

Not every decomposition works. Splitting the same relation into `[id, name, salary]` and `[salary, level]` keeps all the attributes but fixes nothing: `salary` isn't a valid link back to `level` (two levels share 50,000), so the pieces can't be rejoined faithfully. Which split is correct isn't a matter of taste — it's determined by the FDs, which is why the rest of Module 4 exists.

See also

- functional-dependencies[102] · normal-forms[103] · [decomposition] · 3nf-synthesis[104]
- 2026-04-27-database-design-guidelines-and-functional-dependencies[105] — Module 4, part 1
- week10-tutorial-case-study-8-dirt-road-driving[106] — anomaly write-ups on a real schema

[102] courses/infs1200/functional-dependencies.pdf
[103] courses/infs1200/normal-forms.pdf
[104] courses/infs1200/3nf-synthesis.pdf
[105] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[106] courses/infs1200/week10-tutorial-case-study-8-dirt-road-driving.pdf