Update Anomalies and Informal Design Guidelines

Table of contents

The four guidelines 1
Guideline 1 — one relation, one meaning 2
Guideline 2 — the three update anomalies 3
Guideline 3 — reduce nulls 5
Guideline 4 — join on keys 6
Decomposition as the fix 6
See also 6

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 anomalies**) that a bad schema produces. Introduced in 2026-04-27-database-design-guidelines-and-functional-dependencies[1].

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

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

#	Guideline	What it protects against
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[2] and normal-forms[3] 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[4]):

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

DEPARTMENT [Dname, Dnumber, Dmgr_ssn]
    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[5] does automatically when a schema is derived from an ER diagram: one relation per entity type.

[2] courses/infs1200/functional-dependencies.pdf
[3] courses/infs1200/normal-forms.pdf
[4] courses/infs1200/foreign-keys.pdf
[5] courses/infs1200/er-to-relational-mapping.pdf

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:

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[6], 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[7]).

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

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

Operation	Verdict
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[8] 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
Nulls **cannot be used for comparison**	Predicates and aggregates over the column stop behaving as expected — see sql-nulls[9]

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[10] leaves every short order carrying nulls; and the insertion anomaly above is really a null-in-the-primary-key problem, which integrity-constraints[11] forbids outright.

[8] courses/infs1200/functional-dependencies.pdf
[9] courses/infs1200/sql-nulls.pdf
[10] courses/infs1200/normal-forms.pdf
[11] courses/infs1200/integrity-constraints.pdf

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[12]), 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[13] 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[14].

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]
```

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[15] · normal-forms[16] · decomposition[17] · 3nf-synthesis[18]

[12] courses/infs1200/foreign-keys.pdf
[13] courses/infs1200/decomposition.pdf
[14] courses/infs1200/decomposition.pdf
[15] courses/infs1200/functional-dependencies.pdf
[16] courses/infs1200/normal-forms.pdf
[17] courses/infs1200/decomposition.pdf
[18] courses/infs1200/3nf-synthesis.pdf

- 2026-04-27-database-design-guidelines-and-functional-dependencies[19] — Module 4, part 1
- week10-tutorial-case-study-8-dirt-road-driving[20] — anomaly write-ups on a real schema

[19] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[20] courses/infs1200/week10-tutorial-case-study-8-dirt-road-driving.pdf