Database Design Guidelines and Functional Dependencies

Table of contents

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>`

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

> **i** 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 2ⁿ 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

> **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.

[1] courses/infs1200/week10-tutorial-applied-class-9-functional-dependencies.pdf
[2] courses/infs1200/week10-tutorial-case-study-8-dirt-road-driving.pdf