INFS1200 — Week 10 Notes

Table of contents

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

ID	Name	Level	Salary
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 `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

> **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].

[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

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).

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	?

> **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}.

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

i 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.

i 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) — this is a **superkey**.
- {B, C} = {A, B, C, D, E, F} (via {B,C}→A, then A→D, C→F, F→E) — this is a **composite (candidate) key**: {B, C, A} is a superkey but not minimal (since {B, C} alone already suffices), and {B, C} is the minimal set achieving the same closure.

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.

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.

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.

i 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.

i 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.

i 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[4]. 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.

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[5]). 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
```

[4] courses/infs1200/2026-04-27-database-design-guidelines-and-functional-dependencies.pdf
[5] courses/infs1200/week11-tutorial-case-study-9-payroll-system.pdf

```
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, advise
TravelInsuranceHistory.tripName references TripExpenseAllocations.tripName
```

Sample data (relevant excerpts):

```
AssetUse
employeeID  assetID  timestamp             useDuration  assetType     purchaseDate  insurance
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

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

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

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.