CAT 2

Database Design

Alejandro Pérez Bueno

Jan 16, 2026

Table of Contents

Exercise 1

Class Diagram

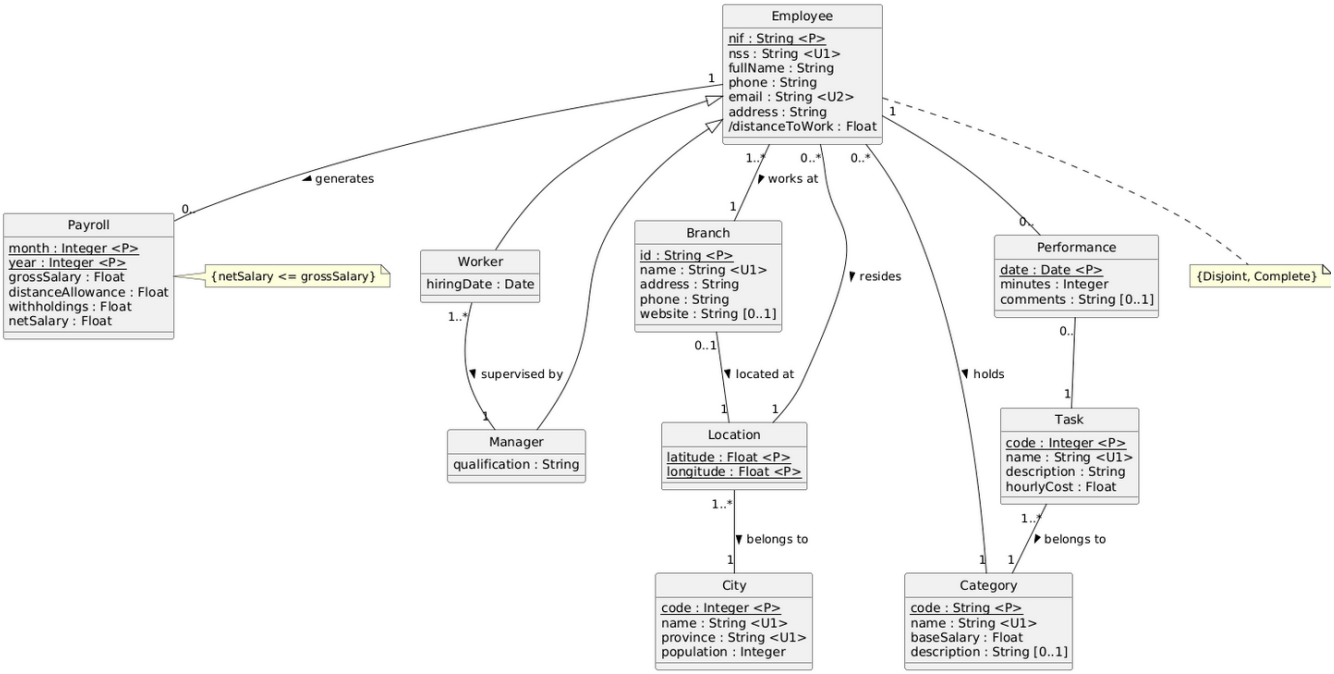

Figure 1: Conceptual Design

Constraints and Semantic Assumptions

- The calculation `netSalary = grossSalary + allowance - withholdings` cannot be enforced structurally, only the inequality constraint `netSalary <= grossSalary` can be calculated.
- The system cannot structurally ensure that "Although unusual, an employee may perform tasks that do not correspond to their category" is treated as an exception versus a rule. The model simply allows any employee to perform any task (Many-to-Many), which satisfies the requirement but does not force the preference for their own category.

Exercise 2

Relational Logical Model

> Notation
>
> Primary keys are underlined, and mandatory attributes (i.e., those that cannot be NULL) are shown in **bold**. Attributes that are unique (alternate keys) are highlighted.

> ⚠ Warning
>
> I could not find a reliable way to insert a dotted underline for alternate keys with the word processor I use, which is why I am using highlight instead.

Client (TIN, name, surname, address, phoneNumber)

BussinesAdvisor (TIN, phoneNumber, mail)

Contract (idContract, **registrationDate**, terminationDate, **TIN_Client**, TIN_Advisor)

- {TIN_Client} is Foreign Key to Client
- {TIN_Advisor} is Foreign Key to BussinesAdvisor

Effective (idContract, **isEffective**)

- {idContract} is Foreign Key to Contract

Service (idService, **service**, **serviceDate**, terminationDate, **idContract**)

- {idContract} is Foreign Key to Contract

BreakDowns24h (idService, **isHandyManServices**)

- {idService} is Foreign Key to Service

GasSupply (idService, oldCompany)

- {idService} is Foreign Key to Service

Breakdown (idService, failureDate, repairDate, startTime, endingTime)

- {idService} is Foreign Key to BreakDowns24h

Period (idService, month, year)

- {idService} is Foreign Key to GasSupply

Gasconsumption (idService, month, year, **gasReading**, **type**)

- {idService, month, year} is Foreign Key to Period

Requirements that cannot be reflected in the logical model

- It cannot be enforced that the month in `Period` is between 1 and 12 (it would require a CHECK constraint or application logic).
- The disjoint and complete constraint on the Service hierarchy (`GasSupply` and `BreakDowns24h`) cannot be enforced structurally in standard SQL without using triggers or specific constraints ehich means a Service must be either `GasSupply` or `BreakDowns24h` (not both) and not just a generic Service.

Exercise 3

a) Dependencies required for 2NF

To be in **Second Normal Form (2NF)**, the relation must be in 1NF furst and must not contain partial dependencies of non-key attributes on part of a composite candidate key.

In this case the candidate keys are `{attr1, attr2}` (primary key) and `{attr4}` (alternate key). The non-key attributes are `{attr3, attr5}`.

To satisfy 2NF, the key functional dependencies should include:

1. Full dependency on the composite primary key: both of the non-key attributes must depend on the *entire* composite key, not on `attr1` or `attr2` alone:
   - `{attr1, attr2} -> attr3`
   - `{attr1, attr2} -> attr5`
2. Dependency from the alternate key: because `attr4` is also a candidate key (it isunique), it determines all remaining attributes: `attr4 -> {attr1, attr2, attr3, attr5}`

b) Dependencies that would violate 3NF

A relation is in **Third Normal Form (3NF)** if it is already in 2NF and has no transitive dependencies where a non-key attribute determines another non-key attribute. `attr3` and `attr5` are non-key attributes, so any dependency that makes one determine the other breaks 3NF. Examples of this:

1. `attr3 ->attr5`
2. `attr5 -> attr3`
3. `{attr1, attr3} ->attr5`
4. `{attr1, attr5} -> attr3`
5. `...`

In each case, the determinant contains a non-key attribute (or is a non-key attribute by itself), which is exactly what 3NF is meant to prevent.

c) Converting the relation to BCNF

To reach Boyce–Codd Normal Form (BCNF), every functional dependency `X → Y` must have `X` as a superkey. The dependency `attr3 -> attr5` violates BCNF because `attr3` is *not* a candidate key, yet it determines another attribute.

A standard BCNF decomposition using that dependency would be:

1. Create a new relation for the violating dependency
   - `R2(attr3, attr5)`
   - Primary key: `attr3`
2. Remove the dependent attribute from the original relation

- `R1(attr1, attr2, attr3, attr4)`
- Primary key: `{attr1, attr2}`
- `attr3` remains to link to `R2` (so it acts as a foreign key referencing `R2(attr3)`)

BCNF relations

- `R1(attr1, attr2, attr3, attr4)` with PK `{attr1, attr2}`
- `R2(attr3, attr5)` with PK `attr3`

Exercise 4

Why do we use indexes and why is their use fundamental in databases?

Indexes are essential in databases for several reasons:

- They are specialized data structures that reduce the time it takes to retrieve rows when queries search, filter, or sort by specific values.
- When a table lacks an index, the database may need to examine every row to find matches, which becomes slow and inefficient as tables grow larger and as queries require joins or complex pperations.
- Because an index keeps values in an organized structure the database can often return results in the requested order with less work than sorting the full dataset.

To create or not to create an index

a)

Creating an index on `email` is a good idea, because queries are expected to filter by this column, and email values are usually unique, so an index will be highly effective at narrowing results quickly. Since the data is rarely updated, the extra maintenance cost an index adds to inserts or updates should be low. With several thousand users, an index should also outperform a full table scan for typical lookups.

b)

Creating an index on `availablePlaces` is not advisable. This value changes frequently, so the database would have to continuously maintain the index, thus adding overhead to updates that is likely to outweigh any benefit for reads. For `totalPlaces`, even though it barely changes, an index is only worth it if queries commonly filter on specific totals, otherwise it may not help much, especially if many rows share the same total value (which reduces selectivity and makes the index less useful).

Index types

a) Bicycle table with queries on the `status` column (Available, In Use, Unloaded, Broken)

A `bitmap index` is appropriate because the `status` column has low cardinality with only a few possible values, and bitmap structures are efficient for filtering and for combining conditions using logical operations like AND, OR, and

NOT.

b) User table with queries on `balance` (searching for values less than or equal to a threshold)

The `B+ Tree index` is the best fit because it supports range queries efficiently, letting the database traverse to the correct point in the sorted index and then scan through the linked leaf level for all balances up to the given value.

c) Person table with queries filtering by `phone` number

Using a `hash index` is suitable because phone lookups are typically exact-match searches, and hashing maps a specific phone value directly to its stored location for very fast retrieval, though a B-Tree is often used in practice due to broader versatility.

B+ Tree maximum order

> Data
>
> - Key (`code`): VARCHAR(16)
> - Page Size (S): 8 KB = 8192 bytes
> - Pointer Size (p): 3 bytes
> - RID Size (r): 4 bytes
> - Total Rows (N): 10,000.

a) Calculate the maximum order (d) of a B+ tree

We need to distinguish between internal nodes and leaf nodes to find the order, but usually, the order d is defined by the capacity of the internal nodes (or the stricter constraint between the two).

1. Internal Node Equation:

An internal node of order d contains at most $2d$ keys and $2d + 1$ pointers to children. The formula for the space occupied by an internal node is:

$$\text{Space} = (2d \times \text{Key Size}) + ((2d + 1) \times \text{Pointer Size}) \leq \text{Page Size}$$

Substitute the values:

$$(2d \times 16) + ((2d + 1) \times 3) \leq 8192$$

$$32d + 6d + 3 \leq 8192$$

$$38d \leq 8189$$

$$d \leq 8189/38$$

$$d \leq 215.5$$

Since d must be an integer, $d = 215$ for internal nodes.

2. Leaf Node Equation:

Leaf nodes contain index entries consisting of a key and a Record ID. It also has a pointer to the next leaf page. Here is how we can calculate how many entries (n) fit in a leaf:

$$n \times (\text{Key Size} + \text{RID Size}) + \text{Pointer Size (next leaf)} \leq \text{Page Size}$$

$$n \times (16 + 4) + 3 \leq 8192$$

$$20n \leq 8189$$

$$n \leq 409.45$$

> Note
>
> A leaf can hold 409 entries.

> Result
>
> The maximum order d is 215.

b) Calculate how many rows can be indexed if the height of the tree is h = 1

In a B+ Tree, height usually refers to the levels *above* the leaf level,as follows:

- **h = 0:** Just the root node.
- **h = 1:** Root node + 1 level of Leaf nodes.

Structure at h=1:

1. **Root Node:** It is an internal node. At maximum occupancy, it has $(2d + 1)$ children.

- Max children $= (2 \times 215) + 1 = 431$ children.
- These children are Leaf Nodes.

2. **Leaf Nodes:** We calculated in step (a) that a leaf node can hold $n = 409$ entries (rows).

Calculation:

$$\text{Total Rows} = (\text{Max Children of Root}) \times (\text{Max Entries per Leaf})$$

$$\text{Total Rows} = 431 \times 409$$

$$\text{Total Rows} = 176,279$$

> Result
>
> 176279 rows can be indexed