TASK #2: Potential issues

Prevent inserting data in poor quality

Our first line of defence from bad data is to use restrictions on the front-end. Having data checks on that level will prevent and guide the users on data entry and ensure that the database will contain good and structured data and that there will be no need to clean up the mess later. Checks can include data type, duplicate entry, length of strings or numbers and mandatory fields.

From a back-end perspective there are still barriers or notifications that can enable us to be aware of data quality being compromised, which will be discussed in the following points.

# Not Null usage

When creating our tables, we can ensure to add not null constraint for columns which is critical for reading data correctly and naturally for primary and foreign keys. However, while ensuring that data can be queried in a meaningful way, we want to ensure that the database isn't too rigid. Say if an employee haven't got a Latvian personal-code yet or filed a travel request with an open-ended return date (endDate). In such cases, it should be considered whether data is useful if null or whether it is considered an incomplete entry.

## Triggers

Adding triggers will allow for an automated check when data is created or updated. Triggers will guarantee that inserted and updated data is valid, that it follows business rules, are unique records and can generate log/audit changes in the database. Triggers can be used before or after the command (trigger\_time) and it will then function as either a warning or an error.

While triggers comes with benefits mentioned, they can cause issues when developers are debugging as they occur 'behind the scene'. Triggers can initiate other triggers and can cause confusion for employees who are not aware of them – deleting one place can insert a new row in another place.

Maintaining data quality

It is not easy to completely prevent bad data to enter a database. Perhaps it is allowed to intake data sometimes with varied quality – for the purpose of getting enough data into a database. Therefore, the ability to maintain data quality is key and will allow for a flexible intake of data combined with a way to identify compromised records. The following examples will go through some of the consideration for the EMPLOYEE and related tables.

# Assertion queries

To verify our data, we can run queries daily or on a regular basis to ensure that data follows certain business rules. Checking for null values, unique entries using WHERE clause to query unwanted entries such as

* column\_name is NULL
* len(personalCode) > 12
* NOT process IN (“approved”, “rejected”)
* etc.

# Former employees

As the workforce in Nordigen is dynamic and talented people will join and leave the company, it should be considered how to handle such cases from a database perspective.

## Cascade delete

When considering how to handle employees leaving the company, an option is to use cascade delete. This will enable to delete an employee and the records related to this employee – when the id is used as a foreign key in other tables.

While allowing to delete rows in multiple tables gives certain flexibilities, consideration must be taken to think about issue that might come with it. TRAVELS table might be compromised if an employee is removed that is added as byEmployee as it will remove entries for other employees.

## Add active/inactive column

Adding a new status column to the EMPLOYEE table will allow keeping a reference to employees who has left the company. Previous employees id used as a foreign key in other tables will be kept while enabling queries to distinguish between current and previous workforce.