Database and Data Mining

Project to build a Database using PostgreSQL and Golang

Domain: Transmission Projects in Europe

Lukas Schirren, Fridtjof Damm

During this Project we built a Database and performed queries with data from the Ten Year Network Development Plan (TYNDP). The ENTSO-E, the European Network of Transmission System Operators, published in 2018 a list of future transmission projects [1], which we used to build an ER-Model and extract result sets. For our Database Management System(DBMS) we used "PostgreSQL" and used Golang to define tables and simple functions. The SQL Queries were performed within the PgAdmin application.

Keywords Relational Database; Golang; PostgreSQL; Transmission Projects

1 Step by step

In the first step we analyzed the TYNDP-data from the Excel-Sheet and placed the columns in entities for our Entity-Relationship Model (ERM). The data from the Excel sheet was divided into eight parts for the respective entity to simplify the insertion of the data.

Then we wrote the code to define our database. For that, we used Golang and the pg-package from Github [2]. We included the "pg" package in a new package "db", where the eight tables and their attributes were defined. Unfortunately we had to define for each table (here implemented as structs) a "create table"-function, since Golang does not yet have generics [3]. We run the program with `go build` and `go run main.go`. The code can be found on Github [4].

After that we connected our Golang program to the PostgreSQL, which runs in a Docker-Container. The PostgreSQL, a relational DBMS, is used to store the data. With the command `pgcli -h localhost -p 5432 -U postgres`, an interface to access the database, could we confirm that the process was successful.

We worked on simple functions to insert projects and investments within the Golang program. But it is only possible to work with a single object. Since we later worked with the PgAdmin application, those functions weren't used intensively.

I tried to include the excel-data within the Golang code, but didn't manage to include it correctly. Since that is an important feature, this will be included in a future project.

2 Database Design

We had a clear definition of the data that we include in our database, since we used a static Excel sheet with 18 columns. We also knew what the purpose of the database was, since we defined questions to our data beforehand. With that, we could already start to divide the given data into tables and create an ERM. We had to make some changes to the data, since we had to include NULL values and we also included two new keys to point on entities.

Entity Relationship Model We decided to split the 18 attributes up into 8 entities. The most important entity is Investment with the primary key *investment_id*, which is associated to five other entities.

A Project can have one or many associated investments. The number of investments is represented in the attribute *Investments*. A relation between project and investment is "only one" to "one or many" [5], because an investment during its lifetime (it will be deleted when finished) can only be associated to exactly one project. A transmission line is built in different steps and the project is not finished until every investment is completed.

The primary key *ProjectId* from the entity project occurs as a foreign key in investments.The attribute Promoter can have multiple promoters in one cell, divided by a semicolon.

The entities FromTo and Country have a one to one relation to investment. The primary key for both of them is also a foreign key from investment. It's not an "only one" relation, since transmission lines can be build between the same countries or cities.

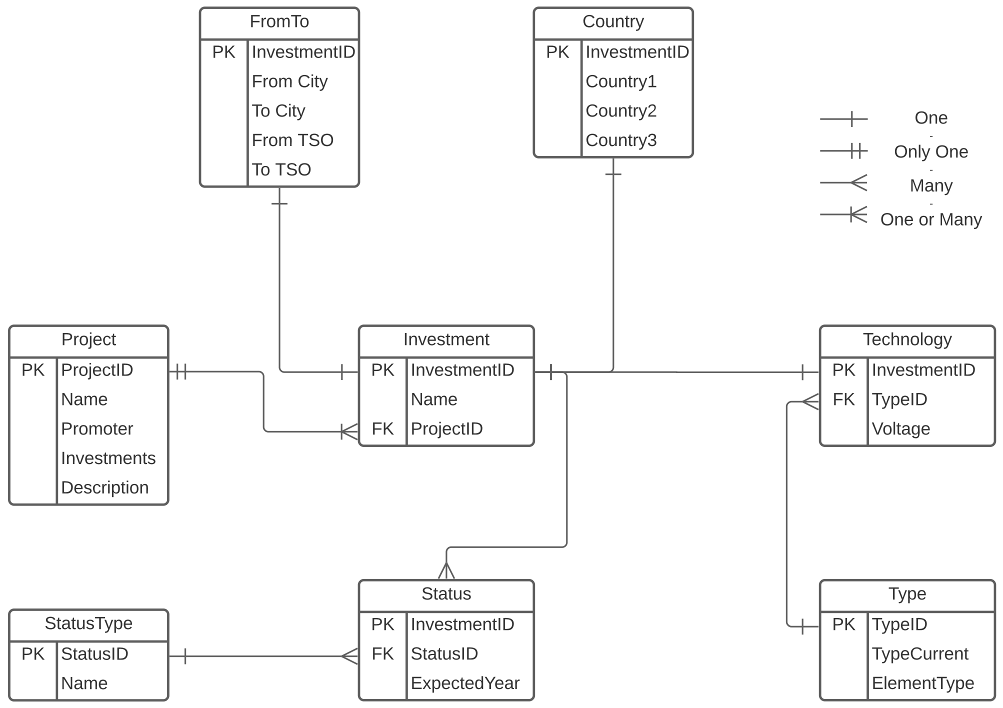

FIGURE 1: Entity Relationship Model

The entity Technology defines *ElementType* (type of transmission line), *TypeCurrent* (AC or DC) and how much *Voltage* is used. The element type and current type are related, for example an sub-sea cable is always DC and an overhead line AC. Due to that we decided to build an extra entity Type. We implemented it by using in the technology entity a *TypeID* (FK) to connect it to the Type entity.

A similar implementation is used for the Status. The *StatusID* (FK) is used to identify the current status. This was implemented, because there wasn't a standardized way to define the current status of an investment and multiple variants of one status existed. With that variant we set a domain for the current status (commissioned, permitted, under consideration etc.), here *Name* in StatusType.

3 Performed Queries

Before we started the project, we formulated several questions for the data.

1. **Find all investments with a higher Voltage level of 400 which are built with the „Elements Type" of sub-sea cable**

   ```
   SELECT investment_id FROM technologies WHERE type_id = 5 AND voltage > 400
   ```

   Relational Algebra: $\Pi_{investment\_id}(\sigma_{type\_id} = 5ANDvoltage > 400(technologies))$

   We select all technologies, who have the *type_id* equal to 5 (sub-sea cable) and the voltage higher than 400, then we project them on the *investment_id*. In this query, we use only constraints.

   There was another way, where we use `INTERSECT`.

   ```
   SELECT investment\_id FROM technologies WHERE type\_id = 5
   INTERSECT
   SELECT investment\_id FROM technologies WHERE voltage > 400
   ```

   Relational Algebra: $\Pi_{investment\_id}(\sigma_{type\_id} = 5(technologies))type_id = 5(technologies)$ $\cap\ \Pi_{investment\_id}(\sigma_{voltage} > 400(technologies))$

   In Figure 2 we see the two sub-queries who constrain the entity Technology. Then the tables append, select all instances and copy them into one set. The hashed-intersect statement discards all *investment_id*'s which do not exist in both select queries.

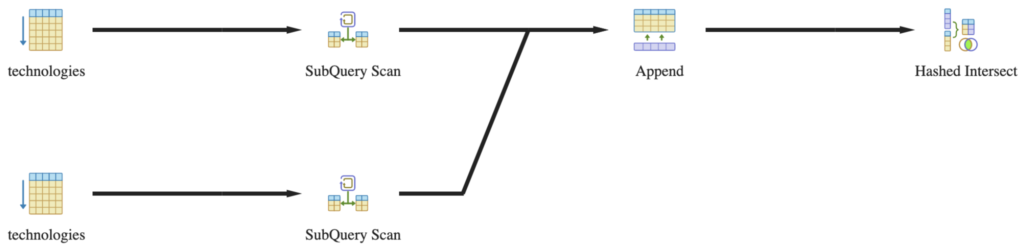

Figure 2: Graphical presentation of the first question

2. **List of Promoters and the number of associated projects**

```
SELECT DISTINCT promoter, COUNT(A) AS occurrence
FROM
(SELECT split_part(promoter::text,',',1) as promoter FROM projects
UNION ALL
SELECT split_part(promoter::text,',',2) as promoter FROM projects
UNION ALL
SELECT split_part(promoter::text,',',3) as promoter FROM projects) A
WHERE promoter <> ''
GROUP BY promoter ORDER BY occurrence DESC
```

Since some instances in the attribute *promoter* have up to three promoters, we split them apart, and then combine them with the `UNION ALL` statement. Then we count the occurrence of each promoter and show a distinct table.

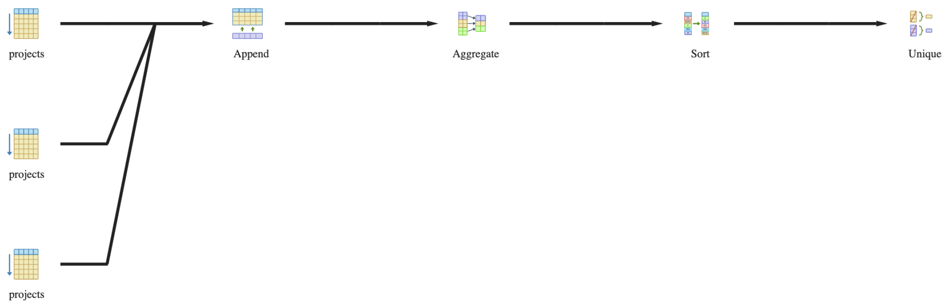

FIGURE 3: Graphical presentation of the second question

The split is used on the table promoter. In the append statement they result in one column, due to `UNION ALL`. With the `GROUP BY promoter` they get aggregated over each promoter and return the number of the occurrence, done with `COUNT()`. Then we sort the data for the biggest number, `ORDER BY occurrence DESC`. As a last step, we exclude the empty cells.

3. **Show already finished investments and projects**

Since the data is from 2018, projects that end in 2021 are outdated. Here, we need to find the finished investments and check if all investments inside a project are finished. Only then, the project and associated investments can be removed. As a first step, we looked at the investments and applied a constraint to show all *investment_id*'s before 2022, our finished investments:

```
SELECT investment_id FROM statuses WHERE expected_year < 2022
```

Relational Algebra: $\Pi_{investment\_id}(\sigma_{expected\_year} < 2022(statuses)$

Then we counted the finished investments and put them next to their *project_id*, done via a join:

```
SELECT project_id, COUNT(E.investment_id) as finished_investments
FROM investments E JOIN statuses F ON E.investment_id = F.investment_id
WHERE expected_year < 2022
GROUP BY project_id ORDER BY project_id
```

At this point we didn't know how to proceed, since performing two counts within one query struck me as complicated and an idea from Stack Overflow didn't work in our case [6].

Another way were two joins. We get all *investment_id*'s, who are finished at the end of 2021. Possible with a join over the attribute *expected_year* from Status. Then we wanted to check, if the number of associated investments from a project is equal to finished investments.

```
SELECT DISTINCT P.project_id FROM investments AS I
JOIN projects AS P ON P.investments = SUM(case when ???? then 1 else 0 end)
JOIN statuses AS S ON S.investment_id=I.investment_id
WHERE S.expected_year < 2022
GROUP BY project_id
```

The query has four question-marks at the `SUM()`, because we didn't find a key to compare the entity project and investment. The *project_id* does not help, since we work with the investments.

Eventually we removed the attribute *expected_year* from Status and inserted it in the Investment entity (Appendix A). Due to that it was possible to perform the following query:

```
SELECT a.project_id FROM projects a
WHERE a.investments = (SELECT COUNT(investment_id) FROM investments g
WHERE expected_year<2022 AND a.project_id = g.project_id) ORDER BY a.project_id
```

In Figure 4 we can see, that the query performs a sup-plan for the `WHERE` statement. This is due to brackets we set around the `(SELECT COUNT()...)` statement. Then we select the *project_id*'s, where the count of the finished investments is equal to the attribute *investments*. The result set contains the finished projects.

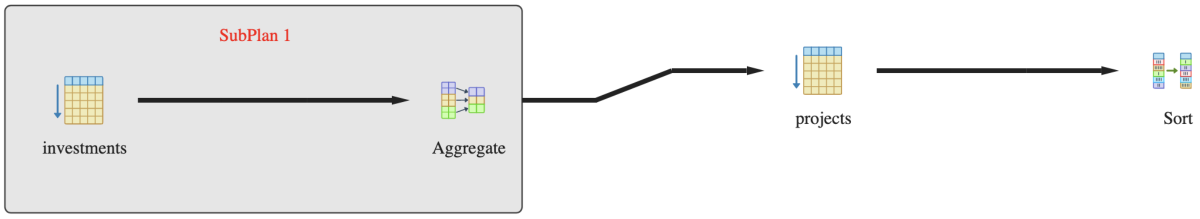

Figure 4: Graphical presentation of the third question

4. **Count the associated investments in each country**

```
SELECT DISTINCT country,
COUNT(a) as occurrence FROM
(SELECT country1 as country FROM countries WHERE country1!='NULL'
UNION ALL SELECT country2 as country FROM countries WHERE country2!='NULL'
UNION ALL SELECT country3 as country FROM countries WHERE country3!='NULL') a
GROUP BY country
```

Here we counted the occurrence of investments in each country. First we exclude all cells with the value "NULL" and perform a `UNION` of all countries. This is seen in figure 5 and results in one column. Then the occurrence for each country is calculated with `COUNT()` and `GROUP BY country`.

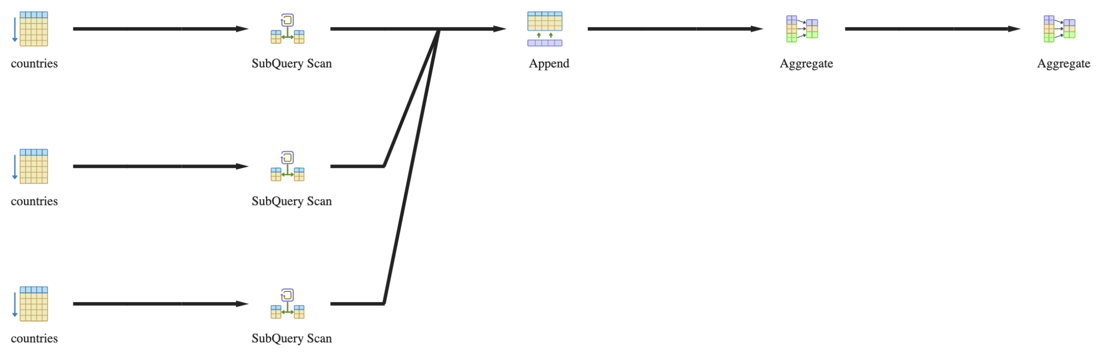

Figure 5: Graphical presentation of the third question

4 Transactions

In our database there are three important transactions:

- **Insert a new project with associated investments** It is important that each investment has an associated project. Due to that it makes sense to insert a new project first and build a relation with the investments afterwards. When that is finished, the attribute *investments*, number of investments, can be set. We could set here the atomicity property, that all projects and all investments must be included, otherwise the transaction is cancelled.
- **Update the current status** is not as critical as an insert or delete. It must be ensured that there are not two users writing at the same time, so that the value of one cell changes appropriately (consistency and isolation). To change the state we only need to change the attribute, the rest stays.
- **Delete a project and the associated investments when it is finished** can be done in the opposite way of an insert. First we delete all leaves and go back to the entity investment, then the associated project can be deleted.

For example, we could update the status of an *investment_id* in the entity Status:

```
UPDATE statuses SET status_id=1 WHERE investment_id = 4
```

Here we updated in the entity Status the status from "in permitting" to "commissioning" from the investment_id 4.

References

[1] Ten year network development plan. Available at `https://tyndp.entsoe.eu/maps-data/`.

[2] Github package "go-pg". Available at `https://github.com/go-pg/pg`.

[3] Generics in go. Available at `https://go.dev/blog/generics-proposal`.

[4] Github repository. Available at `https://github.com/lukasschirren/DBM_GoLang/tree/master`.

[5] Symbols and notations for er diagram. Available at `https://www.lucidchart.com/pages/ER-diagram-symbols-and-meaning`.

[6] Multiple counts in one query. Available at `https://stackoverflow.com/questions/12789396/how-to-get-multiple-counts-with-one-sql-query`.

Appendices

A New Entity-Relationship Model

Since we include the attribute *expected_year* in the entity Investments we could include the status name directly in the investment entity as *status_id* and delete the whole entity Status. Then we relate Investment and status_type directly as seen in figure 6. This would simplify the third SQL query and the ERM due to less entities.

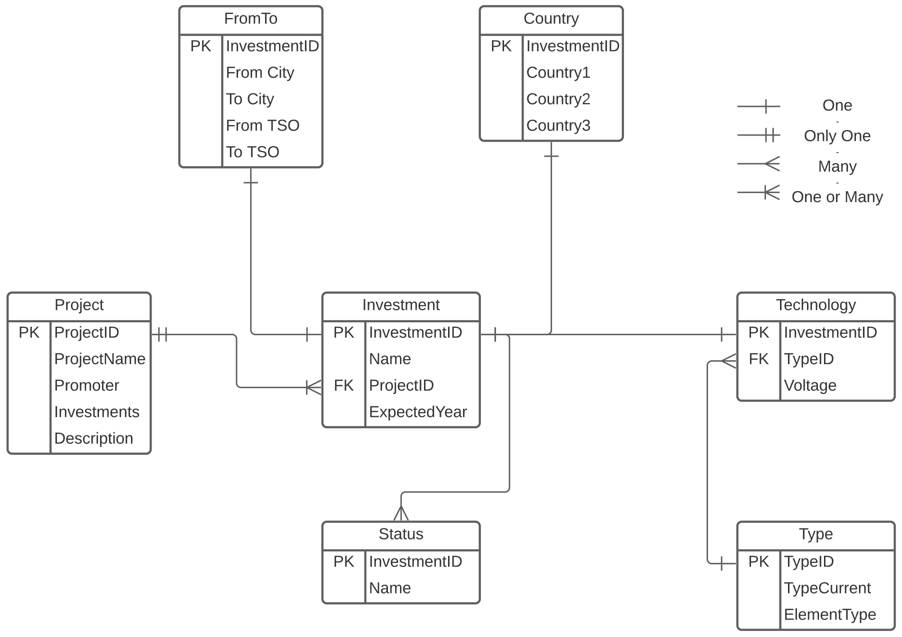

Figure 6: New Entity Relationship Model