

Deliverable 1

Team 1 :

AJROUCHE Maël

DELAHAYE Adrien

LE GALL Paul

PHAN CONG Laurent

Table of contents

Context.....	2
Data dictionaries	3
Functional dependency	6
Conceptual data model (3rd normal form)	7
Logical data model (LDM).....	11
Physical Data Model (PDM).....	13
Algebraic trees	23
Conclusion	30

Context

Raymond F. Boyce discusses the requirements for information to be extracted from a database using queries. To meet these needs, it is necessary to design a conceptual data model (CDM) that represents the entities, relationships, and attributes of the database. The algebraic tree, on the other hand, translates queries formulated in natural language into a logical form that can be processed by the database management system (DBMS). By using algebraic trees to translate the queries formulated by the branch managers, it will be possible to easily retrieve the requested information from the database structured according to the conceptual data model. This will ensure a rigorous organisation of the data for better information retrieval.

Data dictionaries

We created our data dictionary to organise and standardise the information used in our project. In creating a data dictionary, we sought to clarify the definitions of specific technical terms and to establish a central reference for all the data used. We used an alphabetical approach to structure the dictionary and included terms, definitions, and detailed descriptions for each entry.

Our data dictionary consists of **10 tables**:

- **Agency**

Agency					
Meaning	Code	Type	Size	Constraint	Comment
Agency ID	ID_AGENCY	AUTO_INCREMENT	4	Primary Key	The agency's ID

We have as primary key **ID_AGENCY** to distinguish each agency. A **primary key** is an attribute or set of attributes that uniquely identifies each record in a table in a database.

It was decided to choose **auto-increment** as the type because it automatically generates a unique value for each new record, thus avoiding the need to manually specify a primary key for each insertion.

The size of the **auto-increment** in a data dictionary depends on the type of data used to generate the auto-incremental values and the amount of records expected in the table, we decided to choose a size of **4 bytes**.

- **Agent**

Agent					
Meaning	Code	Type	Size	Constraint	Comment
Employee ID	EMPLOYEE_ID	AUTO_INCREMENT	4	Primary Key	The employee's ID
Name	EMPLOYEE_NAME	VARCHAR	100	Not Null	Name of the employee
First name	EMPLOYEE_FIRST_NAME	VARCHAR	100	Not Null	First name of the employee
Birth date	EMPLOYEE_BIRTH_DATE	DATE	3	Not Null	Birth date of the employee
Start date	EMPLOYEE_START_DATE	DATE	3	Not Null	Start date of the employee
Street number	EMPLOYEE_STREET_NUMBER	INT	4	Not Null	Street number of the employee
Address	EMPLOYEE_ADDRESS	VARCHAR	100	Not Null	Address of the employee
Active	EMPLOYEE_ACTIVE	BOOLEAN	8	Not Null	Whether an employee is active or not

In the same way as for the Agency table, we decided to choose the auto increment type for our Employee ID. The **"Not Null"** is a constraint that can be applied to a column of a table to indicate that it cannot contain null values, hence its use.

For the name and address, we decided to use the **VARCHAR** because compared to other fixed data types like **CHAR**, which always occupy the same amount of space and it adds spaces to fill the storage, the **VARCHAR** type can save us storage space when the first and last names are of variable length. We have set a **size of 100** to ensure that even people with very long first and last names can be put in.

For the dates we used the **DATE** type, with a size of **3 bytes** in order to be able to put the whole date.

For the street number, we used the **INT** type because it is used to store integer numbers as in our case, where we just want to store street numbers.

For the activity of the employee, we decided to use the type **BOOLEAN**, in order to know if the employee is active or not.

- **Jobs types**

Jobs types					
Meaning	Code	Type	Size	Constraint	Comment
Jobs types	JOBS_TYPES	VARCHAR	30	Primary Key	The different types of employee jobs
Tasks	TASKS	Text	100	Not Null	The different tasks of the employee

For the primary key Jobs types, this time it's not a number, it's a word, that's why we decided to use the **VARCHAR** with a **size of 30** because the 3 different jobs in our case do not exceed 30 characters.

We also added the tasks of each employee, with the type **TEXT** which allows to put simply text.

- **City**

City					
Meaning	Code	Type	Size	Constraint	Comment
City ID	CITY_ID	AUTO_INCREMENT	4	Primary Key	The city's ID
City	CITY_NAME	VARCHAR	40	Not Null	City names
Postal code	PC	VARCHAR	10	Not Null	Postal codes

For the cities, we did the same way as the others. However, for the postcodes, they may also be letters and numbers, so we used **VARCHAR** with a size of 10 for the numbers and for the letters if necessary.

- **Region**

Region					
Meaning	Code	Type	Size	Constraint	Comment
Region ID	REGION_ID	AUTO_INCREMENT	4	Primary Key	The region's ID
Region	REGION_NAME	VARCHAR	50	Not Null	Region names

In this table, an ID (Region ID) and a region name have been created to avoid repetition.

- **Report**

Report					
Meaning	Code	Type	Size	Constraint	Comment
Report ID	REPORT_ID	AUTO_INCREMENT		Primary Key	The report's ID
Report date	REPORT_DATE	DATE	3	Not Null	The report's date

In this table, Report ID is our primary key. It represents the identifiers of the reports written by the administrative staff.

- **Sensor**

Sensor					
Meaning	Code	Type	Size	Constraint	Comment
Sensor ID	SENSOR_ID	AUTO_INCREMENT	4	Primary Key	The sensor's ID

For this table, Sensor Id is our primary key, it corresponds to the number of each sensor.

- **Technical record**

Technical Record					
Meaning	Code	Type	Size	Constraint	Comment
Record ID	RECORD_ID	AUTO_INCREMENT	4	Primary Key	The record's ID
Recorded data	RECORDED_DATA	FLOAT	4	Not Null	The report's data
Recorded date	RECORDED_DATE	DATE	3	Not Null	The recorded's date

For the data, we decided to use the **FLOAT** type, because it allows to take into account comma digits, in our case there are comma digits. And it has a size of **4 bytes** allowing us to store a huge number.

- **Gas**

Gas					
Meaning	Code	Type	Size	Constraint	Comment
Gas	GAS	VARCHAR	55	Primary Key	Gas names
Gas type	GAS_TYPE	VARCHAR	30	Not Null	Gas types

- **Sector activity**

Sector Activity					
Meaning	Code	Type	Size	Constraint	Comment
Sector Activity	SECTOR_ACTIVITY	VARCHAR	65	Primary Key	The sector of activity
PRG	PRG	INT	4	Not Null	The PRG

We decided to take a size of 65 for the sectors as some are long.

For all our tables, the types and sizes follow the same reasoning as explained for the first tables.

Functional dependency

We created our functional dependency matrix to visualise the relationships between the different entities and attributes in our system. By creating this matrix, we were able to identify the functional dependencies between entities and data dictionary attributes. We used this matrix to better understand how data are related to each other and to avoid potential data anomalies, such as redundancies and inconsistencies.

FUNCTIONAL DEPENDENCY MATRIX											
		1	2	3	4	5	6	7	8	9	10
		ID_AGENCY	EMPLOYEE_ID	JOBS_TYPES	CITY_ID	REPORT_ID	SENSOR_ID	ID_RECORD	REGION_ID	GAS	SECTOR_ACTIVITY
1	EMPLOYEE_NAME		1								
2	EMPLOYEE_FIRST_NAME		1								
3	EMPLOYEE_BIRTH_DATE		1								
4	EMPLOYEE_START_DATE		1								
5	EMPLOYEE_STREET_NUMBER		1								
6	EMPLOYEE_ADDRESS		1								
7	EMPLOYEE_ACTIVE		1								
8	TASKS			1							
9	CITY_NAME				1						
10	PC				1						
11	REPORT_DATE		(1)			1					
12	RECORDED_DATA						(1)	1			
13	RECORDED_DATE						(1)	1			
14	REGION_NAME				(1)				1		
15	GAS_TYPE						(1)			1	
16	PRG						(1)				1

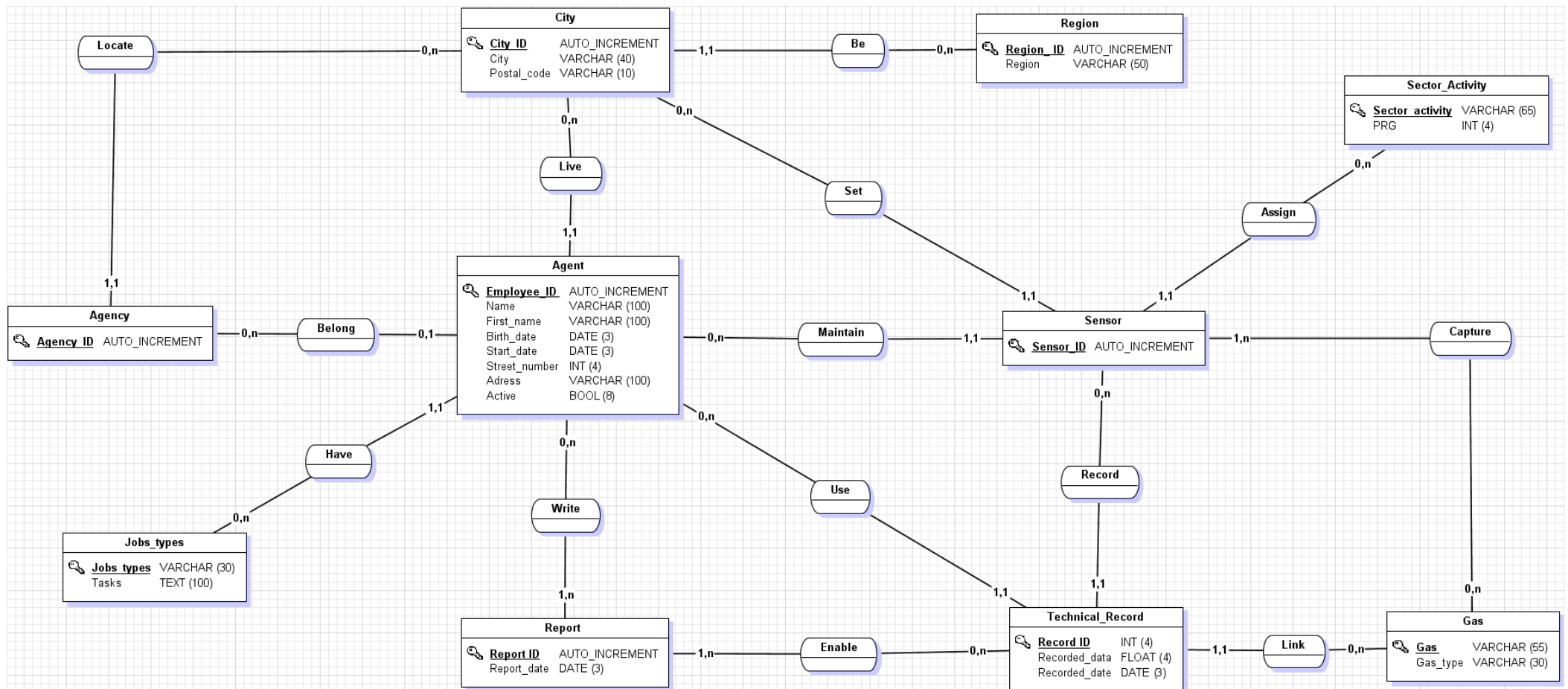
We have placed all of our primary keys in the top section to link them to the other elements.

The "1" in our functional dependency matrix refers to an attribute or entity that is dependent on another attribute or entity. This means that the value of one attribute or entity depends on the value of another attribute or entity in the context of a functional relationship.

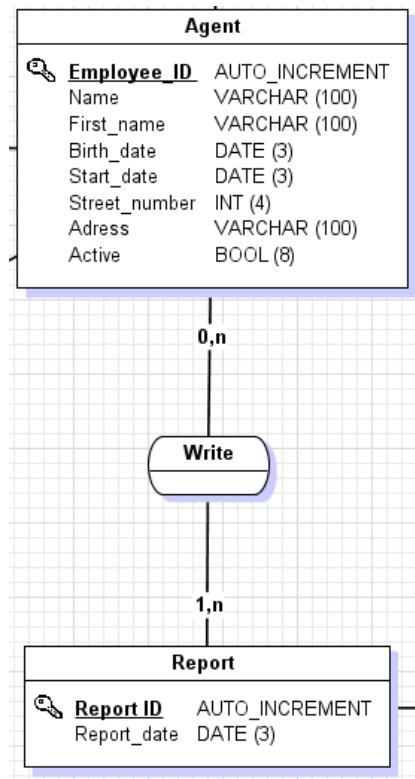
The "(1)" works in the same way but between attributes or entities in 2 different tables.

Conceptual data model (3rd normal form)

We created our conceptual data model in order to represent in a clear and structured way the different entities, attributes and relationships needed to meet our project requirements. This model allows us to visualise and plan the overall structure of our database, to define the relationships between entities, to identify primary and foreign keys, and to normalise the data to ensure its integrity and consistency.



We have applied the different cardinalities in a contextually consistent way, i.e. they refer to the number of relationships between two entities in a database. The different cardinalities are: **0,N; 1,N; 0,1; 1,1.**



For example, between the agent and report table:

- Zero, one or more agents can write a report (0,n)
- One or more reports can be written by an officer (administrative) (1,n)

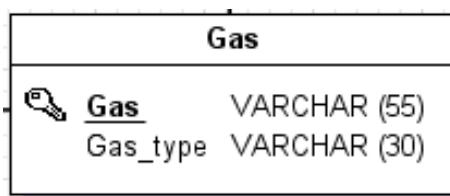
We have tried as much as possible with the help of the 3rd normal form to optimise our conceptual data model in order to avoid:

- Data redundancies and update anomalies in our relational database.

It ensures that each attribute in a relationship depends only on the primary key of that relationship.

The square figures represent the tables in our conceptual model, which are used to represent the entities and attributes (from our data dictionary) in our schema.

Like this :



We have **10 tables** that correspond well to the elements we presented in the data dictionary earlier:


Agent	
 Employee_ID	AUTO_INCREMENT
Name	VARCHAR (100)
First_name	VARCHAR (100)
Birth_date	DATE (3)
Start_date	DATE (3)
Street_number	INT (4)
Address	VARCHAR (100)
Active	BOOL (8)

Table : Agent


Agency	
 Agency_ID	AUTO_INCREMENT

Table : Agency


Jobs_types	
 Jobs_types	VARCHAR (30)
Tasks	TEXT (100)

Table : Jobs types


Region	
 Region_ID	AUTO_INCREMENT
Region	VARCHAR (50)

Table : Region


Sector_Activity	
 Sector activity	VARCHAR (65)
PRG	INT (4)

Table : Sector activity


Sensor	
 Sensor_ID	AUTO_INCREMENT

Table : Sensor


Technical_Record	
 Record ID	INT (4)
Recorded_data	FLOAT (4)
Recorded_date	DATE (3)

Table : Technical record


Gas	
 Gas	VARCHAR (55)
Gas_type	VARCHAR (30)

Table : Gas


Report	
 Report ID	AUTO_INCREMENT
Report_date	DATE (3)

Table : Report


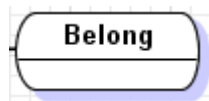
City	
 City ID	AUTO_INCREMENT
City	VARCHAR (40)
Postal_code	VARCHAR (10)

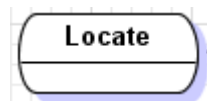
Table : City

In a conceptual data model, tables are linked using relations which are **infinitive verbs (ex : To have, to belong)**, they allow us to define the interactions and associations between the data stored in different tables. We have therefore created 14 relationships, each of which is positioned between **2 tables**:

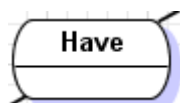
- Agent ----- Agency



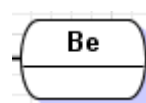
- Agency ----- City



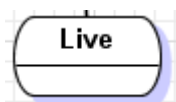
- Agent ----- Jobs_type



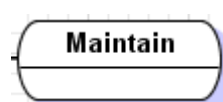
- City ----- Region



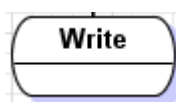
- Agent ----- City



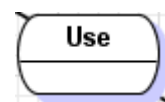
- Agent ----- Sensor



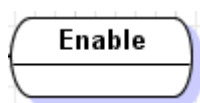
- Agent ----- Report



- Agent ----- Technical_record



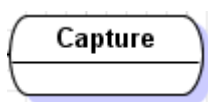
- Report ----- Technical_record



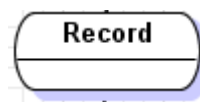
- Technical_record ----- Gas



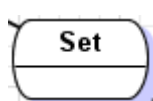
- Sensor ----- Gas



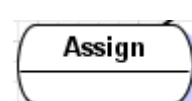
- Sensor ----- Technical_record



- City ----- Sensor

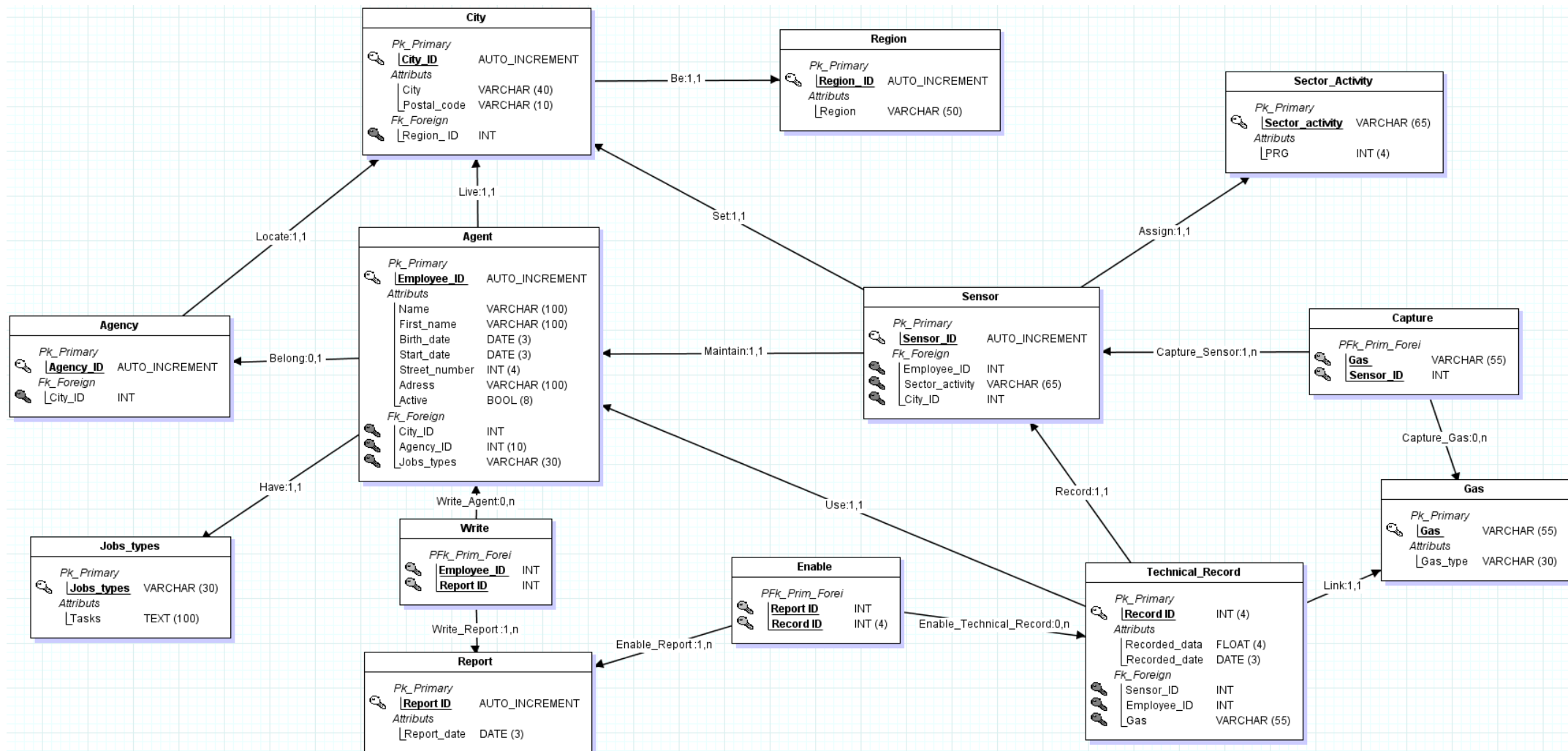


- Sensor ----- Sector_activity

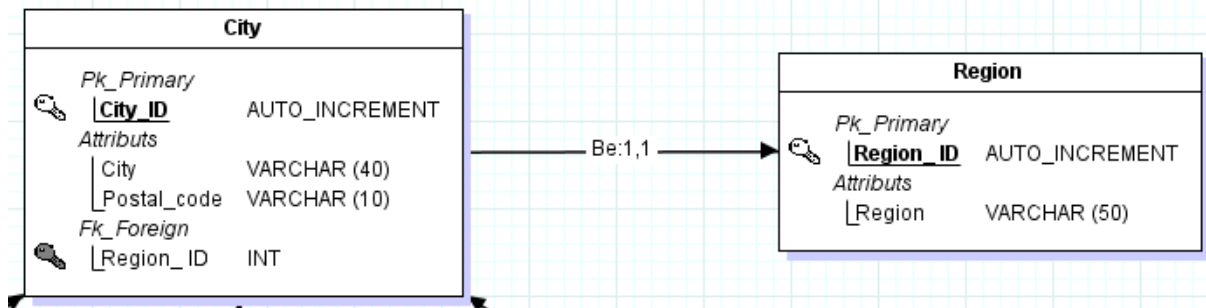


Logical data model (LDM)

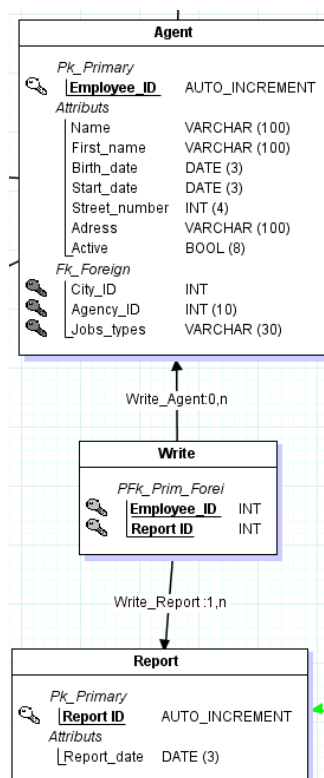
We have generated our Logical data model to provide a visual and understandable representation of the structure and organisation of the data. In this way, we can better understand how the tables are related but more importantly we can better visualise our DCM.



As we can see, we do have our primary keys and attributes in our tables, however the thing that the LDM allows to be displayed are the foreign keys that are used in the other tables:



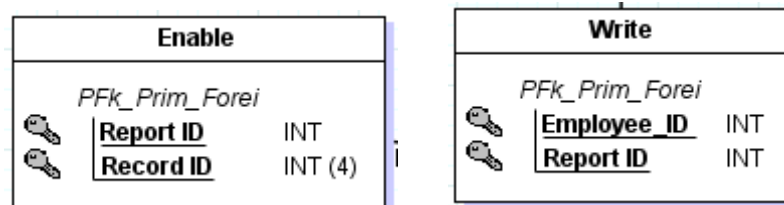
As can be seen, the "City" table now has a foreign key: "Region_ID".



There is another case, where between 2 tables we have multiple "n" in cardinalities, this will create a relationship table like this:

We have a cardinalities of **0,n** and **1,n**: which created a relationship table with our infinitive verb "Write" where we have a **composite key (PFK)** : which is a primary key that is made up of two or more fields in a table in a relational database.

We have exactly two of these:



Physical Data Model (PDM)

The physical data model is used to create the database tables and schemas using SQL as the query language. It forms the basis for creating the tables, adding constraints, indexes, and other database objects necessary for the efficient implementation of a relational database management system. We generated it using Jmerise software.

```
#-----  
# Table: Gas  
#-----  
  
CREATE TABLE Gas(  
    Gas      Varchar (55) NOT NULL ,  
    Gas_type Varchar (30) NOT NULL  
    ,CONSTRAINT Gas_PK PRIMARY KEY (Gas)  
)ENGINE=InnoDB;
```

The SQL code creates a table named "Gas" with the following columns:

- "Gas": A column of type VARCHAR with a maximum length of 55 characters, which cannot be NULL (i.e. it must have a value).
- "Gas_type": A column of type VARCHAR with a maximum length of 30 characters, which cannot be NULL.

The table also has a primary key constraint defined on the "Gas" column, indicated as "Gas_PK". This means that the "Gas" column will uniquely identify each row in the table and cannot contain duplicate values.

```
#-----  
# Table: Report  
#-----  
  
CREATE TABLE Report(  
    Report_ID Int Auto_increment NOT NULL ,  
    Report_date Date NOT NULL  
    ,CONSTRAINT Report_PK PRIMARY KEY (Report_ID)  
)ENGINE=InnoDB;
```

The SQL code creates a table named "Report" with the following columns:

- "Report_ID": A column of type INT that is automatically incremented (Auto_increment), meaning that it automatically fills with a unique value each time a new row is inserted. The column cannot be NULL (i.e. it must have a value).
- "Report_date": A column of type DATE which represents a date, and which cannot be NULL.

The table also has a primary key constraint defined on the "Report_ID" column, specified as "Report_PK". This means that the "Report_ID" column will uniquely identify each row in the table and cannot contain duplicate values.

```
#-----  
# Table: Jobs_types  
#-----  
  
CREATE TABLE Jobs_types(  
    Jobs_types Varchar (30) NOT NULL ,  
    Tasks Text NOT NULL  
    ,CONSTRAINT Jobs_types_PK PRIMARY KEY (Jobs_types)  
)ENGINE=InnoDB;
```

The SQL code creates a table named "Jobs_types" with the following columns:

- "Jobs_types": A column of type VARCHAR with a maximum length of 30 characters, which cannot be NULL (i.e. it must have a value).
- "Tasks": A column of type TEXT which can contain text of variable length, and which cannot be NULL.

The table also has a primary key constraint defined on the "Jobs_types" column, indicated as "Jobs_types_PK". This means that the "Jobs_types" column will uniquely identify each row in the table and cannot contain duplicate values.

```
#-----  
# Table: Region  
#-----  
  
CREATE TABLE Region(  
    Region__ID Int Auto_increment NOT NULL ,  
    Region Varchar (50)  
    ,CONSTRAINT Region_PK PRIMARY KEY (Region__ID)  
)ENGINE=InnoDB;
```

The SQL code creates a table named "Region" with the following columns:

- "Region__ID": A column of type INT that is automatically incremented (Auto_increment), meaning that it automatically fills with a unique value each time a new row is inserted. The column cannot be NULL (i.e. it must have a value).
- "Region": A column of type VARCHAR with a maximum length of 50 characters, which can contain the name of a region.

The table also has a primary key constraint defined on the "Region__ID" column, indicated as "Region_PK". This means that the "Region__ID" column will uniquely identify each row in the table and cannot contain duplicate values.

```
#-----
# Table: City
#-----

CREATE TABLE City(
  City_ID      Int Auto_increment NOT NULL ,
  City         Varchar (40) NOT NULL ,
  Postal_code  Varchar (10) NOT NULL ,
  Region__ID   Int NOT NULL
  ,CONSTRAINT City_PK PRIMARY KEY (City_ID)

  ,CONSTRAINT City_Region_FK FOREIGN KEY (Region__ID) REFERENCES Region(Region__ID)
)ENGINE=InnoDB;
```

The SQL code creates a table named "City" with the following columns:

- "City_ID": A column of type INT that is automatically incremented (Auto_increment), meaning that it automatically fills with a unique value each time a new row is inserted. The column cannot be NULL (i.e. it must have a value). "City": A column of type VARCHAR with a maximum length of 40 characters, which represents the name of a city and cannot be NULL.
- "Postal_code": A column of type VARCHAR with a maximum length of 10 characters, representing the postal code of a city, which cannot be NULL.
- "Region__ID": A column of type INT which represents the identifier of the region to which the city belongs and which cannot be NULL.

The table also has a primary key constraint defined on the "City_ID" column, indicated as "City_PK". This means that the "City_ID" column will uniquely identify each row in the table and cannot contain duplicate values.

The table also has a foreign key constraint defined on the "Region__ID" column, specified as "City_Region_FK", which refers to the "Region__ID" column in the "Region" table. This means that the value of the "Region__ID" column in the "City" table must match an existing value in the "Region__ID" column of the "Region" table, thus ensuring referential integrity between the two tables.

```
#-----
# Table: Sector_Activity
#-----

CREATE TABLE Sector_Activity(
  Sector_activity Varchar (65) NOT NULL ,
  PRG            Int NOT NULL
  ,CONSTRAINT Sector_Activity_PK PRIMARY KEY (Sector_activity)
)ENGINE=InnoDB;
```

The SQL code creates a table named "Sector Activity" with the following columns:

- "Sector_activity": A column of type VARCHAR with a maximum length of 65 characters, which represents a sector activity, and which cannot be NULL.
- "PRG": A column of type INT which represents a PRG (Plan de Redressement et de Gestion) code and which cannot be NULL.

The table also has a primary key constraint defined on the "Sector_activity" column, indicated as "Sector_Activity_PK". This means that the "Sector_activity" column will uniquely identify each row in the table and cannot contain duplicate values.

```
#-----  
# Table: Agency  
#-----  
  
CREATE TABLE Agency(  
    Agency_ID Int Auto_increment NOT NULL ,  
    City_ID Int NOT NULL  
    ,CONSTRAINT Agency_PK PRIMARY KEY (Agency_ID)  
  
    ,CONSTRAINT Agency_City_FK FOREIGN KEY (City_ID) REFERENCES City(City_ID)  
)ENGINE=InnoDB;
```

The SQL code creates a table named "Agency" with the following columns:

- "Agency_ID": A column of type INT that is automatically incremented (Auto_increment), meaning that it automatically fills with a unique value each time a new row is inserted. The column cannot be NULL (i.e. it must have a value).
- "City_ID": A column of type INT which represents the identifier of the city to which the agency belongs and which cannot be NULL.

The table also has a primary key constraint defined on the "Agency_ID" column, indicated as "Agency_PK". This means that the "Agency_ID" column will uniquely identify each row in the table and cannot contain duplicate values.

The table also has a foreign key constraint defined on the "City_ID" column, specified as "Agency_City_FK", which refers to the "City_ID" column of the "City" table. This means that the value of the "City_ID" column in the "Agency" table must match an existing value in the "City_ID" column of the "City" table, thus ensuring referential integrity between the two tables.


```

#-----
# Table: Agent
#-----

CREATE TABLE Agent(
  Employee_ID    Int Auto_increment NOT NULL ,
  Name           Varchar (100) NOT NULL ,
  First_name     Varchar (100) NOT NULL ,
  Birth_date     Date NOT NULL ,
  Start_date     Date NOT NULL ,
  Street_number  Int NOT NULL ,
  Address        Varchar (100) NOT NULL ,
  Active         Bool NOT NULL ,
  City_ID        Int NOT NULL ,
  Agency_ID      Int ,
  Jobs_types     Varchar (30) NOT NULL
  ,CONSTRAINT Agent_PK PRIMARY KEY (Employee_ID)

  ,CONSTRAINT Agent_City_FK FOREIGN KEY (City_ID) REFERENCES City(City_ID)
  ,CONSTRAINT Agent_Agency0_FK FOREIGN KEY (Agency_ID) REFERENCES Agency(Agency_ID)
  ,CONSTRAINT Agent_Jobs_types1_FK FOREIGN KEY (Jobs_types) REFERENCES Jobs_types(Jobs_types)
)ENGINE=InnoDB;

```

The SQL code creates a table named "Agent" with the following columns:

- "Employee_ID": A column of type INT that is automatically incremented (Auto_increment), meaning that it automatically fills with a unique value each time a new row is inserted. The column cannot be NULL (i.e. it must have a value).
- "Name": A column of type VARCHAR(100) which represents the name of the agent and which cannot be NULL.
- "First_name": A column of type VARCHAR(100) which represents the agent's first name and cannot be NULL.
- "Birth_date": A column of type DATE which represents the agent's date of birth and cannot be NULL.
- "Start_date": A DATE column representing the agent's start date and cannot be NULL.
- "Street_number": An INT column representing the street number of the agent's address and cannot be NULL.
- "Address": A column of type VARCHAR(100) that represents the agent's address and cannot be NULL.
- "Active": A BOOL (boolean) column representing whether the agent is active or not, and cannot be NULL.
- "City_ID": A column of type INT which represents the identifier of the city where the agent is based and which cannot be NULL.
- "Agency_ID": A column of type INT which represents the identifier of the agency to which the agent is attached. This column can be NULL, meaning that an agent may not be attached to an agency.
- "Jobs_types": A column of type VARCHAR(30) which represents the agent's job type and which cannot be NULL.

The table also has a primary key constraint defined on the "Employee_ID" column, specified as "Agent_PK". This means that the "Employee_ID" column will uniquely identify each row in the table and cannot contain duplicate values.

The table also has three foreign key constraints defined:

- "Agent_City_FK": which refers to the "City_ID" column of the "City" table. This means that the value of the "City_ID" column in the "Agent" table must match an existing value in the "City_ID" column of the "City" table, thus ensuring referential integrity between the two tables.
- "Agent_Agency0_FK": which refers to the "Agency_ID" column of the "Agency" table. This means that the value of the "Agency_ID" column in the "Agent" table must match an existing value in the "Agency_ID" column of the "Agency" table, thus ensuring referential integrity between the two tables. This constraint allows an agent to be linked to an agency.
- "Agent_Jobs_types1_FK": which refers to the column.

```
#-----
# Table: Sensor
#-----

CREATE TABLE Sensor(
  Sensor_ID      Int Auto_increment NOT NULL ,
  Employee_ID    Int NOT NULL ,
  Sector_activity Varchar (65) NOT NULL ,
  City_ID        Int NOT NULL
  ,CONSTRAINT Sensor_PK PRIMARY KEY (Sensor_ID)
  ,CONSTRAINT Sensor_Agent_FK FOREIGN KEY (Employee_ID) REFERENCES Agent(Employee_ID)
  ,CONSTRAINT Sensor_Sector_Activity0_FK FOREIGN KEY (Sector_activity) REFERENCES Sector_Activity(Sector_activity)
  ,CONSTRAINT Sensor_City1_FK FOREIGN KEY (City_ID) REFERENCES City(City_ID)
)ENGINE=InnoDB;
```

The SQL code creates a table named "Sensor" with the following columns:

- "Sensor_ID": A column of type INT that is automatically incremented (Auto_increment), meaning that it automatically fills with a unique value each time a new row is inserted. The column cannot be NULL (i.e. it must have a value).
- "Employee_ID": A column of type INT which represents the identifier of the agent associated with the sensor. This column cannot be NULL.
- "Sector_activity" : A column of type VARCHAR(65) which represents the activity of the sector associated with the sensor. This column cannot be NULL.
- "City_ID": A column of type INT which represents the identifier of the city where the sensor is located. This column cannot be NULL.
- The table also has a primary key constraint defined on the "Sensor_ID" column, indicated as "Sensor_PK". This means that the "Sensor_ID" column will uniquely identify each row in the table and cannot contain duplicate values.

The table also has three foreign key constraints defined:

- "Sensor_Agent_FK": which refers to the "Employee_ID" column in the "Agent" table. This means that the value of the "Employee_ID" column in the "Sensor" table must match an existing value in the "Employee_ID" column of the "Agent" table, thus ensuring referential integrity between the two tables. This constraint allows a sensor to be linked to an agent.
- "Sensor_Sector_Activity0_FK": which refers to the "Sector_activity" column of the "Sector_Activity" table. This means that the value of the "Sector_activity" column in the "Sensor" table must match an existing value in the "Sector_activity" column of the "Sector_Activity" table, thus ensuring referential integrity between the two tables.
- "Sensor_City1_FK": which refers to the "City_ID" column of the "City" table. This means that the value of the "City_ID" column in the "Sensor" table must match an existing value in the "City_ID" column of the "City" table, thus ensuring referential integrity between the two tables.

```
#-----
# Table: Technical_Record
#-----

CREATE TABLE Technical_Record(
  Record_ID      Int NOT NULL ,
  Recorded_data   Float NOT NULL ,
  Recorded_date   Date NOT NULL ,
  Sensor_ID       Int NOT NULL ,
  Employee_ID     Int NOT NULL ,
  Gas             Varchar (55) NOT NULL
  ,CONSTRAINT Technical_Record_PK PRIMARY KEY (Record_ID)

  ,CONSTRAINT Technical_Record_Sensor_FK FOREIGN KEY (Sensor_ID) REFERENCES Sensor(Sensor_ID)
  ,CONSTRAINT Technical_Record_Agent0_FK FOREIGN KEY (Employee_ID) REFERENCES Agent(Employee_ID)
  ,CONSTRAINT Technical_Record_Gas1_FK FOREIGN KEY (Gas) REFERENCES Gas(Gas)
)ENGINE=InnoDB;
```

The SQL code creates a table named "Technical_Record" with the following columns:

- "Record_ID": A column of type INT that represents the identifier of the technical record. This column cannot be NULL.
- "Recorded_data": A column of type FLOAT which represents the recorded data. This column cannot be NULL.
- "Recorded_date": A column of type DATE which represents the date of the record. This column cannot be NULL.
- "Sensor_ID": An INT column representing the sensor ID associated with this record. This column cannot be NULL.
- "Employee_ID": A column of type INT that represents the agent ID associated with this record. This column cannot be NULL.
- "Gas": A column of type VARCHAR(55) representing the type of gas being recorded. This column cannot be NULL.
- The table also has a primary key constraint defined on the "Record_ID" column, indicated as "Technical_Record_PK". This means that the "Record_ID" column will uniquely identify each row in the table and cannot contain duplicate values.

The table also has three foreign key constraints defined:

- "Technical_Record_Sensor_FK": which refers to the "Sensor_ID" column of the "Sensor" table. This means that the value of the "Sensor_ID" column in the "Technical_Record" table must match an existing value in the "Sensor_ID" column of the "Sensor" table, thus ensuring referential integrity between the two tables.
- "Technical_Record_Agent0_FK": which refers to the "Employee_ID" column of the "Agent" table. This means that the value of the "Employee_ID" column in the "Technical_Record" table must match an existing value in the "Employee_ID" column of the "Agent" table, thus ensuring referential integrity between the two tables.
- "Technical_Record_Gas1_FK": which refers to the "Gas" column of the "Gas" table. This means that the value of the "Gas" column in the "Technical_Record" table must correspond to an existing value in the "Gas" column of the "Gas" table, thus ensuring referential integrity between the two tables.

```
#-----
# Table: Capture
#-----

CREATE TABLE Capture(
  Gas      Varchar (55) NOT NULL ,
  Sensor_ID Int NOT NULL
  ,CONSTRAINT Capture_PK PRIMARY KEY (Gas,Sensor_ID)

  ,CONSTRAINT Capture_Gas_FK FOREIGN KEY (Gas) REFERENCES Gas(Gas)
  ,CONSTRAINT Capture_Sensor0_FK FOREIGN KEY (Sensor_ID) REFERENCES Sensor(Sensor_ID)
)ENGINE=InnoDB;
```

The SQL code creates a table named "Capture" with the following columns:

- "Gas": A column of type VARCHAR(55) that represents the type of gas captured. This column cannot be NULL.
- "Sensor_ID": A column of type INT that represents the sensor ID associated with the capture. This column cannot be NULL.

The table also has a primary key constraint defined on the "Gas" and "Sensor_ID" columns, indicated as "Capture_PK". This means that the combination of values in these two columns will uniquely identify each row in the table and cannot contain duplicate values.

The table also has two foreign key constraints defined:

- "Capture_Gas_FK": which refers to the "Gas" column of the "Gas" table. This means that the value of the "Gas" column in the "Capture" table must match an existing value in the "Gas" column of the "Gas" table, thus ensuring referential integrity between the two tables.
- "Capture_Sensor0_FK": which refers to the "Sensor_ID" column of the "Sensor" table. This means that the value of the "Sensor_ID" column in the "Capture" table must match an existing value in the "Sensor_ID" column of the "Sensor" table, thus ensuring referential integrity between the two tables.

```
#-----
# Table: Write
#-----

CREATE TABLE Write(
  Employee_ID Int NOT NULL ,
  Report_ID Int NOT NULL
  ,CONSTRAINT Write_PK PRIMARY KEY (Employee_ID,Report_ID)

  ,CONSTRAINT Write_Agent_FK FOREIGN KEY (Employee_ID) REFERENCES Agent(Employee_ID)
  ,CONSTRAINT Write_Report0_FK FOREIGN KEY (Report_ID) REFERENCES Report(Report_ID)
)ENGINE=InnoDB;
```

The SQL code creates a table named "Write" with the following columns:

- "Employee_ID": A column of type INT that represents the identifier of the employee writing the report. This column cannot be NULL.
- "Report_ID": A column of type INT which represents the identifier of the report being written. This column cannot be NULL.

The table also has a primary key constraint defined on the "Employee_ID" and "Report_ID" columns, indicated as "Write_PK". This means that the combination of values in these two columns will uniquely identify each row in the table and cannot contain duplicate values.

The table also has two foreign key constraints defined:

- "Write_Agent_FK": which refers to the "Employee_ID" column of the "Agent" table. This means that the value of the "Employee_ID" column in the "Write" table must match an existing value in the "Employee_ID" column of the "Agent" table, thus ensuring referential integrity between the two tables.
- "Write_Report0_FK": which refers to the "Report_ID" column of the "Report" table. This means that the value of the "Report_ID" column in the "Write" table must match an existing value in the "Report_ID" column of the "Report" table, thus ensuring referential integrity between the two tables.

```
#-----
# Table: Enable
#-----

CREATE TABLE Enable(
  Report_ID Int NOT NULL ,
  Record_ID Int NOT NULL
  ,CONSTRAINT Enable_PK PRIMARY KEY (Report_ID,Record_ID)

  ,CONSTRAINT Enable_Report_FK FOREIGN KEY (Report_ID) REFERENCES Report(Report_ID)
  ,CONSTRAINT Enable_Technical_Record0_FK FOREIGN KEY (Record_ID) REFERENCES Technical_Record(Record_ID)
)ENGINE=InnoDB;
```

The SQL code creates a table named "Enable" with the following columns:

- "Report_ID": A column of type INT that represents the identifier of the enabled report. This column cannot be NULL.
- "Record_ID": A column of type INT which represents the identifier of the activated technical record. This column cannot be NULL.

The table also has a primary key constraint defined on the "Report_ID" and "Record_ID" columns, indicated as "Enable_PK". This means that the combination of values in these two columns will uniquely identify each row in the table and cannot contain duplicate values.

The table also has two foreign key constraints defined:

- "Enable_Report_FK": which refers to the "Report_ID" column of the "Report" table. This means that the value of the "Report_ID" column in the "Enable" table must match an existing value in the "Report_ID" column of the "Report" table, thus ensuring referential integrity between the two tables.
- "Enable_Technical_Record0_FK": which refers to the "Record_ID" column of the "Technical_Record" table. This means that the value of the "Record_ID" column in the "Enable" table must match an existing value in the "Record_ID" column of the "Technical_Record" table, thus ensuring referential integrity between the two tables.

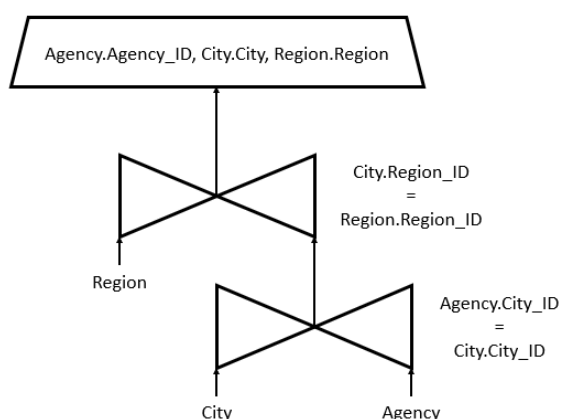
Algebraic trees

After creating our conceptual, logical, and physical data models, we can now work on algebraic trees since the main objective of this database is to gather all the information from the country's agencies and to be able to access it easily.

The government's technical lead has explicitly requested certain functionalities in the form of a list of information that should be easily retrievable from the database through queries:

1. List all agencies.
2. List all technical staff from the Bordeaux agency.
3. Provide the total number of deployed sensors.
4. List the reports published between 2018 and 2022.
5. Calculate the total greenhouse gas emissions by region in 2020.
6. Display the most polluting sector of activity in Ile de France.
7. Rank the reports concerning NH3 emissions in chronological order.
8. Provide the names of technical agents maintaining sensors related to acidifying pollutants.
9. For each gas, provide the sum of its emissions (in tons) in the Ile-de-France region in 2020.
10. Provide the productivity rate of administrative agents from the Toulouse agency (based on the number of written reports and their seniority in the position).
11. For a given gas, list the reports containing data concerning it (the name of the gas must be provided as a parameter).
12. List the regions where there are fewer sensors than agencies.

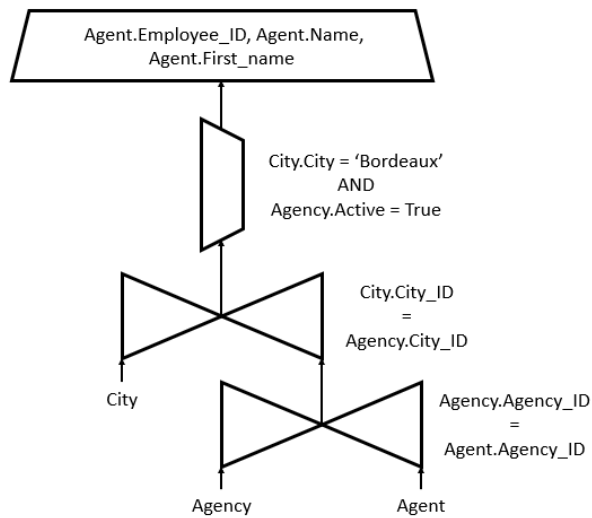
1. List all agencies:



To list all the agencies of the database, we just have to select all rows from the Agency table.

We added the city and the region where the agency is located is situated for a more comprehensive view of the output, by joining Agency table with the City table and then joining with the Region table.

2. List all technical staff from the Bordeaux agency:

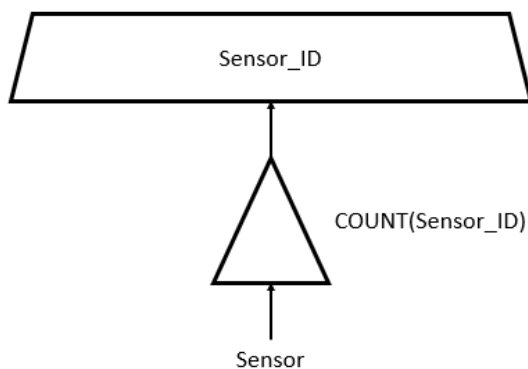


It is somewhat like the previous query, but this time we want to list all the agents and only those from Bordeaux.

We can't directly join the table **Agent** to the table **City** because an individual may live in one city and work in another. Therefore, we joined the table **Agent** and the table **Agency** then we joined with the table **City**.

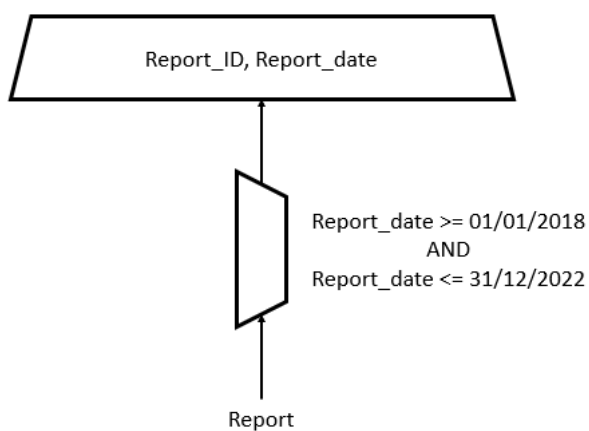
This allows us to filter the data to only get the data where $\text{City} = \text{'Bordeaux'}$.

3. Provide the total number of deployed sensors:



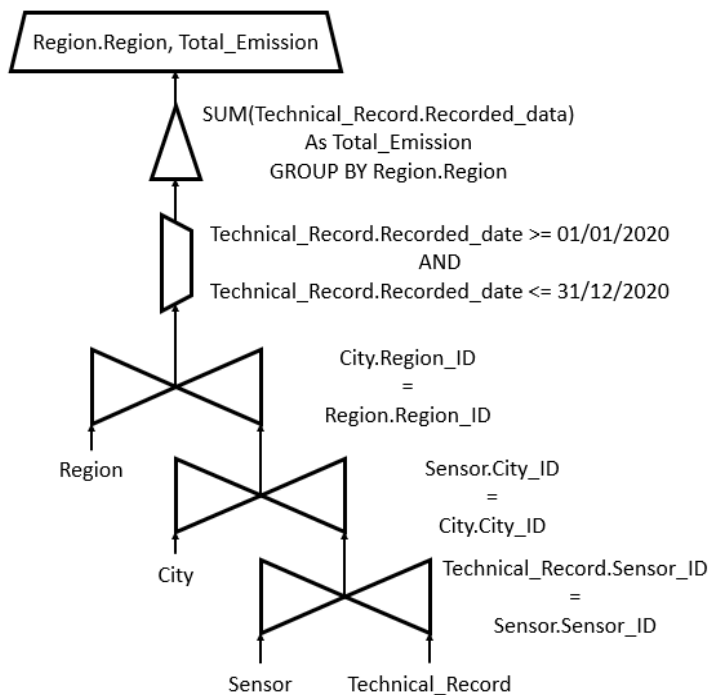
To obtain this total number, we need to process the sum of all the deployed sensors by using **COUNT**.

4. List the reports published between 2018 and 2022:



We just need to filter the data from the **Report** table to get the reports published from 2018 through 2022.

5. Calculate the total greenhouse gas emissions by region in 2020:



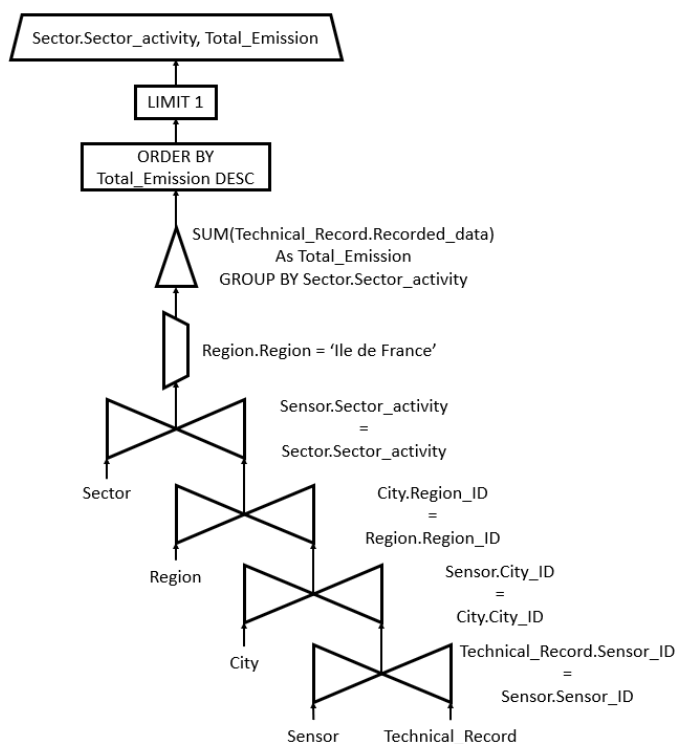
We need to calculate the total greenhouse gas emissions from a certain period.

Firstly, we gather all the relevant information by joining the Technical_Record table with the Sensor table, then joining this with the City table, and finally joining with the Region table.

Secondly, we want to focus on the data from 2020 so we remove all the data that is not within this time frame.

To finish, we sum up all the data by region and then display the results.

6. Display the most polluting sector of activity in Ile de France:

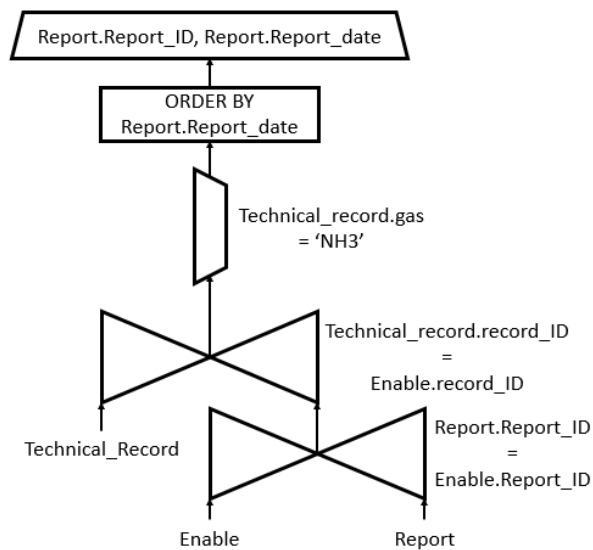


To display the most polluting sector in Ile de France, we need to rank all the polluting sector in descending order and then show the top one.

We join several tables to gather all the necessary data and then we use a filter to focus on the data related to Ile de France.

After that, we aggregate the technical records by sector of activity to generate the initially mentioned list.

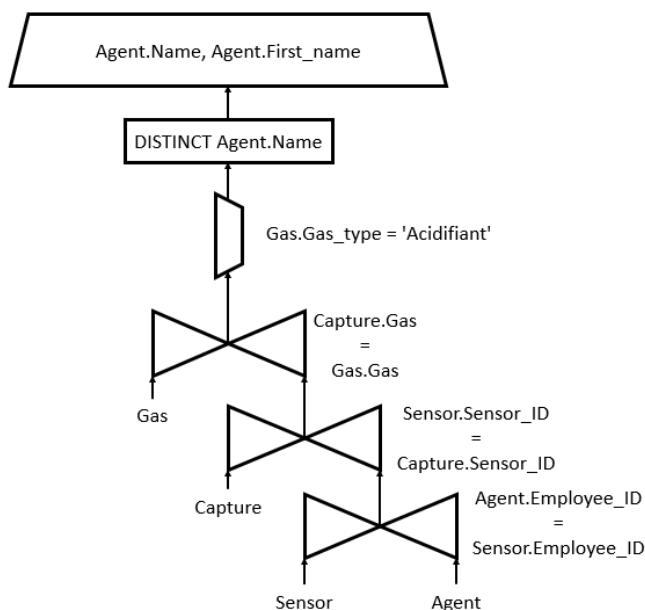
7. Rank the reports concerning NH3 emissions in chronological order:



The reports are created using technical records. So, to list all the reports that mentioned NH3, we need to join the table `Report` and the table `Technical_Record` with the intermediate `Enable` table.

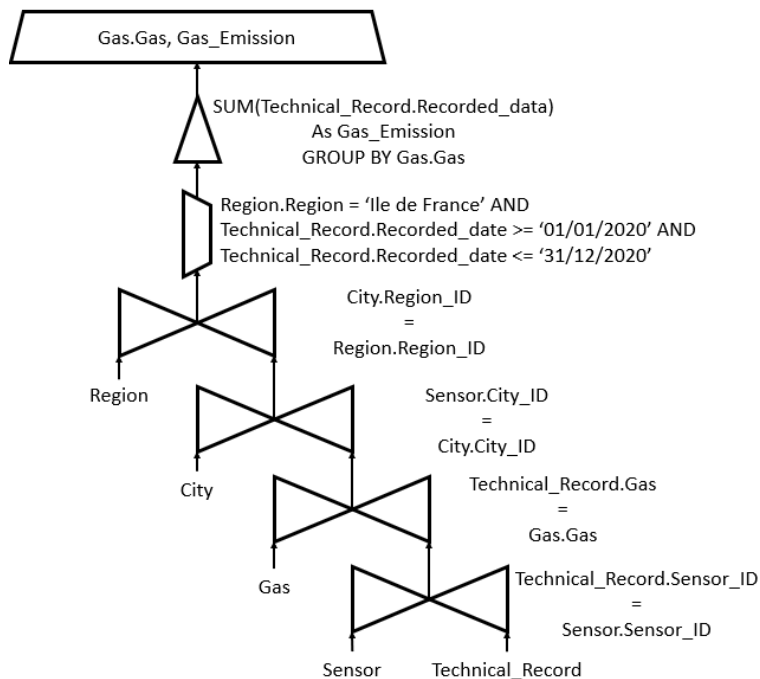
Then we just filter all the report and sort them in chronological order.

8. Provide the names of technical agents maintaining sensors related to acidifying pollutants:



If we just display the agent names after filtering the joined tables, we will see several times the same agents. To take away those duplicates, we use `DISTINCT` to get our list of technical agents maintaining sensors related to acidifying pollutants.

9. For each gas, provide the sum of its emissions in the Ile-de-France region in 2020:

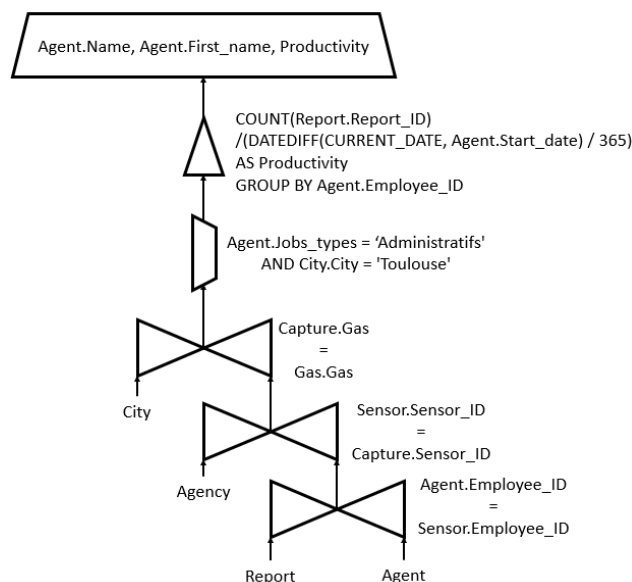


To get the list of all the emissions of those gases in Ile-de-France, we have to join the table Technical_Record with the table Region.

After the joint, we filter the data to only have the Record from Ile-de-France in 2020.

To finish, we sum up all the recorded data from the technical records by region.

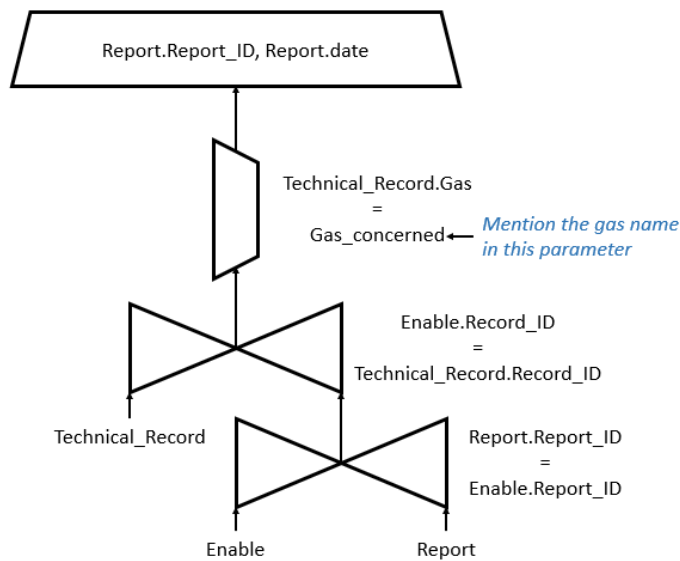
10. Provide the productivity rate of administrative agents from the Toulouse agency (based on the number of written reports and their seniority in the position):



To process this rate, we need the amount of report the agent made and how long an agent is in the company.

We also must filter the data to only get the administrative officers from Toulouse.

11. For a given gas, list the reports containing data concerning it (the name of the gas must be provided as a parameter):

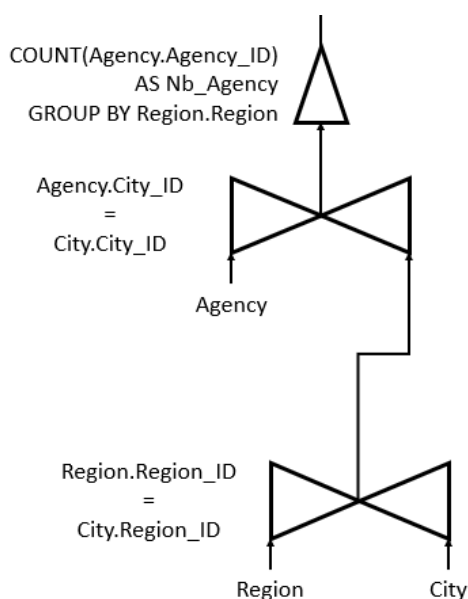


This one is like the 7th without the ordering, but we need to use a parameter in the filter.

12. List the regions where there are fewer sensors than agencies.

This task is more challenging than the others, and we must divide the tree into three parts to accomplish it:

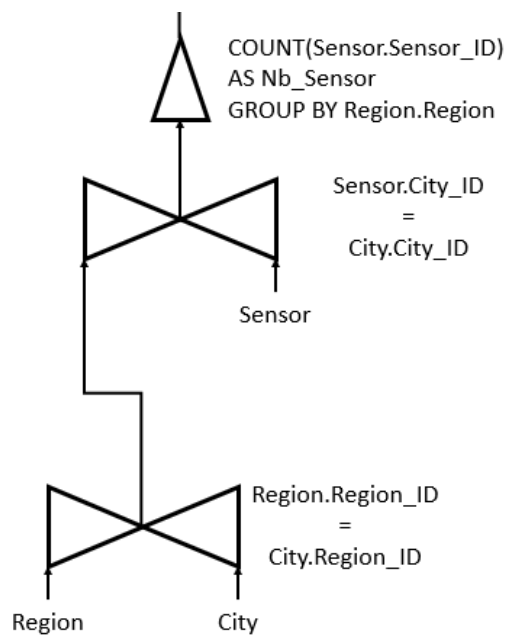
- Get the number of agencies per region.
- Get the number of sensors per region.
- Retain the regions where there are more agencies than sensors.



To begin, we need to gather information on the agencies and their locations.

To do this, we join the table Region and the table City then join this with the table Agency.

Afterward, we can count the number of agencies per regions and store it in a column named Nb_Agency.

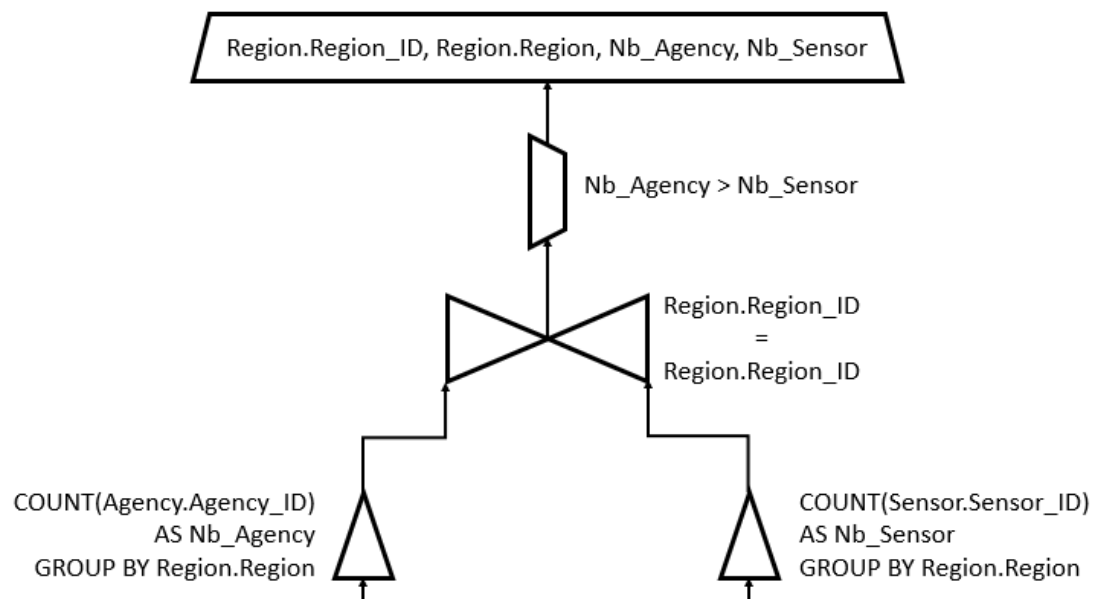


In the other “branch”, we have to do the same stuff as we did for the agencies.

We join the table **Region** and the table **City** then join this with the table **Sensor**.

Next, we count the number of sensors per regions and put this in a column called **Nb_Sensor**.

Taking everything into account, we can join these results and then compared them.



We join the 2 branches and then display only the regions where there are more agencies than sensors.

Conclusion

By adopting these approaches, we have structured and optimized our database, making information more accessible while facilitating access to information. We've developed data models and identified functional dependencies to maintain data coherence and reliability.

These achievements will contribute to an efficient implementation and population of the database, enable high-performance queries, and ensure the sustainability and adaptability of the project to anticipate potential developments and facilitate future updates.