DataScientest - com

Introduction to SQL

Introduction to tables and databases

This module presents an introduction to the **SQL language** through 5 notebooks.

- The first notebook deals the **general principles of tables and databases**, the main elements allowing the use of the SQL language.
- The second notebook will present the main **SQL queries**.
- The third notebook will present the **aggregation functions** and **Group by** queries, the **joins**.
- The fourth notebook will present the basic operations in modify tables or databases.
- The fifth notebook (optional) will introduce **SQLAlchemy**, a tool allowing to use SQL queries directly with Python.

Databases: DBMS and RDB

A **Database management system (DBMS)** is a software that allows users to access a database. The DBMS manages the access (identification, security) and interprets **queries** in order to **input, modify, retrieve or delete data**. There are two main families of SGBDs: the **relational** SGBDs and the **NoSQL** SGBDs.

DBMSs can be associated with development environments or frameworks software (IDEs) that contain text editors and SQL consoles that allow writing and executing SQL queries in a simplified way.

The most used DBMS currently are:

- **PostGreSQL**
- **Microsoft SQL Server**
- **MySQL**
- **Oracle Database**

These DBMS are used on IDEs such as **DBeaver** or **Microsoft SQL SERVER Management Studio**.

Relational databases (RDB) have been used since the 1970s and are now extremely popular. These databases use **tables** to store data. These tables follow a strict **scheme** which imposes constraints on the structure of the data.

To interact with these DBMS, we use generically use the **SQL** (Structured Query Language) language to make queries on the databases. SQL is very widespread and rather simple to use. It is therefore a **smart-language to the Data world**.

Diagrams and tables

Tables: excel in databases are sets of records (or observations) that form the rows of the table. Each of the variables (or a record) corresponds to an **attribute**. The attributes form the **columns** of the table.

We can compare a table to an excel table (or to a DataFrame): the records correspending to the rows and the attributes to the columns.

Tables are characterized by their **scheme**. The scheme of a table must contain the **attributes**, the **types** and **the constraints** on its over all tables. We can say that the scheme is a definition or a **"fingerprint "** of the table.

Let's look at the example of the **Films** table:

FilmID	Name	Director
1	Mission Impossible	Brian Di Palma
2	Avengers : Infinity War	Joe Russo et Anthony Russo
3	The Godfather	Frank Darabont

Each row in the table corresponds to a film. The columns of the table correspond to their characteristics:

- `FilmId` : the unique identifier of the film (an integer).
- `Name` : the name of the film (a string of characters).
- `Director` : the director (a string of characters).

By defining the name and type of each attribute we defined the scheme of the table.

We will often refer to a table with its **name followed by the name of its attributes in brackets**:

Films(`FilmId`, `Name`, `Director`)

This representation is called the **formal representation** of the table.

1. Types of variables

In SQL the attributes can take many types. Depending on the DBMS, the types may vary. The following are the main ones:

TypeVARCHAR	Type Integer
VARCHAR	Variable length string
INTEGER	Integer
FLOAT	Variable precision floating point
DOUBLE	Double precision floating point
DECIMAL	Number value (Precise Fixed)
DATE	Date
DATETIME	Date and Time
CHRONE	String of characters

- **(a)** Give the diagram and the formal representation of the **Sales** table described below:

SaleID	Item	Sale date	Quantity	Unit Price
1	pen	2020-01-10	2	0.50
2	eraser	2020-01-11	1	0.70
3	lamps	2020-01-13	5	2.50

In [ ]:
```
# Insert your code here
```

Hide solution

In [ ]:
```
'SaleID' : Unique identifier of the sale (INTEGER)
'Item' : Name of the item sold (VARCHAR)
'Sale Date' : Date of sale (DATE)
'Quantity' : Number of items sold (INTEGER)
'Unit Price' : Unit price of the item (FLOAT or DOUBLE)

The formal representation of the table is given by:
Sales(SaleID, Item, Sale date, Quantity, Unit Price)
'''
```

2. Transactions

A **transaction** is a sequence of operations requiring **reads** or **writes** access. This transaction is performed via a database management system

To guarantee the integrity of a database, the transactions must be **reliable**.

One way to ensure reliable transactions is to check **ACID** properties. And it's the acronym for:

- **Atomicity**: A transaction must be able to be rolled back in its entirety or it will be aborted. If one or single operation of the transaction is refused, all other operations of the transaction will be refused.
- **Consistency**: The initial and the final states of the database must be consistent. It is ensured by constraints of the database must be respected after a transaction.
- **Isolation**: A transaction must run independently of transactions that are running at the same time. The intermediate states of a transaction are hidden from the others.
- **Durability**: When a transaction is validated, its effects must be stored and therefore persist even in the cases of a system reset or power failure.

Relational databases check the ACID properties while NoSQL databases do not. Indeed, transactions are therefore less reliable, but they can perform better on large databases.

Keys and constraints

1. Primary key

Let's take the **Films** table defined earlier:

FilmID	Name	Director
1	Mission Impossible	Brian Di Palma
2	Avengers : Infinity War	Joe Russo et Anthony Russo
3	The Godfather	Frank Darabont

The `FilmId` attribute is an **identifier that uniquely defines each film**.

We can say that this attribute defines the **primary key** of the **Films** table.

The attributes `Name` and `Director` do not have to respect this constraint: movies can have the same name or the same director.

To indicate that the attribute `FilmId` is the primary key, we underline it in the formal representation of the table:

Films(<u>**FilmId**</u>, `Name`, `Director`)

What you need to know about primary keys:

- The primary key of a table is the element that allows **to uniquely identify a record of the table**.
- The primary key can be composed of one or more attributes (we will talk about this later when we will present the **composite keys**.
- The primary key must always be filled in and must be unique (two records cannot have the same primary key).

- **(b)** Give the primary keys of the following tables:

Sales :

SaleID	Item	Sale date	Quantity	Unit Price
1	pen	2020-01-10	2	0.50
2	eraser	2020-01-11	1	0.70
3	lamps	2020-01-13	5	2.50

Transactions :

TransID	Transac Date/Hr	Vendor ID	Timestamp
0000	2020-01-01	1247548	1578120000000000
0016.7	2020-01-01	1247548	1578120000000000
0000	2020-01-01	1247548	1578120000000000

In [ ]:
```
# Insert your code here
```

Hide solution

In [ ]:
```
'''
The primary key of the Sales table is: SaleID. This is a unique and systematic identifier.
The primary key of the Transactions table is: (TransactionID, Timestamp) or (TransID and a Vendor ID) can be listed
in several transactions. Also, depending on the precision of your timestamp, there could be several transactions
'''
```

2. Foreign key

In general, relational databases contain several tables. For example, you can find in the same database the table **Actors** :

ActorID	FirstName	LastName	CountryID
1	Brad	Pitt	1
2	Marion	Cotillard	2
3	Vince	Vaughn	1

and the table **Countries** :

CountryID	Name
1	USA
2	Australia
3	France

The formal representation of the **Countries** table is:

Countries(<u>**CountryID**</u>, `Name`)

We notice that the **Actors** table contains the attribute `CountryId`, which refers to the primary key of the **Countries** table. This attribute is a **foreign key** of the **Actors** table. The foreign key is represented by a of in the formal representation of the table :

Actors(<u>**ActorID**</u>, `FirstName`, `LastName`, `#CountryId`)

- A foreign key identifies a column (or a set of columns) of a table as a reference to a column of another table (the referenced table). The columns of the referenced table must be a primary key. Thus the foreign key ensures or guarantees that each of the values in the table exists in the referenced table.

For example, it is not possible to add a Spanish actor in the **Actors** table because the `CountryId` for Spain does not exist in the `Countries` table.

Here is the **Casting** table from the same database which associates the actors with the movies they played in:

ActorID	FilmID	Role
1	1	Ethan Hunt
2	2	Tony Stark
3	2	Thor

- The **Casting** table is an intermediate table. It allows to link the tables **Films** and **Actors** .

- **(c)** Give the representation of the **Casting** table.
- **(d)** Can we add the row (2, 4, Sherlock Holmes) to the **Casting** table? Why?

In [ ]:
```
# Insert your code here
```

Hide solution

In [ ]:
```
'''
c) Casting(#ActorId, #FilmId, Role)
d) The answer is no. The FilmId values of the Casting table is a foreign key referring to the primary key
of the Films table. The value 4 does not exist in the Films table. We must first add the film in the Films table and then row in the Casting table.
You could first have to add the row (4, Sherlock Holmes, Spy Thriller) in the Movies table.
'''
```

3. Composite key

A **composite key** is a key on several attributes. There are **composite primary keys**, for example an address for it to be unique you have to specify the number, the street name, the city name, the postal code and the country. We could store all this information in a single variable, but we would lose the possibility of making queries by city or street for example.

Coming back to the **Casting** table, the primary key is neither `FilmId` nor `ActorId` (which uniquely defines a single in this table. Then we have :

Casting(<u>**#ActorId**</u>, <u>**#FilmId**</u>, Role)

The **Books** table lists books written by Sir Arthur Conan Doyle and published in several languages.

ISBN13Id	Language	Title
1a	en	Sherlock Holmes: A Study in Scarlet
1b	en	The Sign of the Four: Sherlock Holmes
2a	en	The Hound of the Baskervilles: Sherlock Holmes
2b	en	Le Chien des Baskerville

- **(e)** What is the primary key of the **Books** table?

In [ ]:
```
# Insert your code here
```

Hide solution

In [ ]:
```
'''
The ISBN13Id attribute is not unique in the Books table: in fact translations of the same story must have the same ISBN13Id.
The title is not unique either because the title can be the same for two books in different languages.
You only may be uniquely define a book in interface the couple (ISBN13Id, Language).
'''
```

4. Constraints

To modify a table, you must respect the constraints imposed on it. There are 4 types of constraints:

- **Domain constraint**: It defines what a data or attribute can take. I.e. the data type and restrictions on this type.
- **Key constraint**: The uses of a table cannot have the same primary key.
- **Entity integrity constraint**: an attribute cannot take on a value or a value that is not a primary key. When creating a table, we add a constraint on the values that the attribute can take.
- **Referential integrity constraint**: an attribute cannot take on a value or a value that is not a primary key or not in another relationship. The Referential integrity ensures and therefore prohibits the existence of a tuple that refers to a tuple that does not exist in another table.

Relational diagrams

As we have seen with the foreign keys, the tables can be connected. We is called a **relational diagram**. The connection can be a column existing in both tables, or 2 distinct columns linked by another table (attribution table).

The rows of a table can therefore correspond to one rows in other tables. We can have three types of correspondence:

- **One to One**: a row corresponds to another row in the other table
- **One to Many**: a row corresponds to several rows in the other table.
- **Many to Many**: a row of a table can correspond to several rows of the other table and inversely.

Keeping the same database as before, we can build the **Synopsis** table which gives the summary of each film in the **Films** table.

SynopsisId	FilmId	Synopsis
1	1	[illegible]
2	2	[illegible]
3	3	[illegible]

- **(f)** What is the relationship between the **Films** and **Actors** tables?
- **(g)** What is the relationship between the **Films** and **Synopsis** tables?
- **(h)** What is the relationship between the **Films** and **Casting** tables?

In [ ]:
```
# Insert your code here
```

Hide solution

In [ ]:
```
'''
f) Several actors play in the same movie and an actor can play in different movies.
It is thus a Many to Many relationship.
g) Each movie corresponds to a synopsis and vice versa (we can notice that FilmId is a primary key of Synopsis).
It is thus a One to One relation.
h) Each movie can correspond to several rows of the Casting table (all rows corresponding to the movie).
On the other hand, a row of the Casting table corresponds to only one movie. It is thus a One to Many relation.
'''
```

Conclusion

You learned what a relational database is and what are its main characteristics.

Here is what you must to remember from this notebook before moving on to the **SQL Queries** course:

- The main characteristics of relational databases (diagrams, and tables, tuples, ACID properties).
- The **primary keys** to identify a tuple of a table (a unique key).
- **Foreign keys** are used to link a table to another table.
- **Composite keys**.
- The main types of diagrams between tables.

Validate