Project documentation

Construction of a data warehouse and OLAP model for analyzing the Scribus bug tracker

1. Introduction

This project was carried out as part of the 62-62 Data Exploitation module. Its objective was to build a comprehensive decision-making system capable of analyzing data from the Scribus bug tracker. We had to implement the various stages of a modern decision-making process:

1. automatic extraction and processing of weekly CSV snapshots,
2. design and feeding of a Data Warehouse based on a star schema,
3. management of historization (SCD Type 2) in the fact table,
4. creation of a Tabular OLAP model with SSAS,
5. development of analytical measures in DAX,
6. preparation of a model that can be used in tools such as Power BI or Excel.

---

2. Project objectives

The main objectives were as follows:

- Understand and structure the Scribus bug tracker data.
- Create a star schema adapted to analytical needs.
- Develop an automated ETL pipeline in Python.
- Manage dimensions in SCD Type 1 and the fact table in SCD Type 2.
- Build a Tabular model in SSAS based on the product DWH.
- enable detailed and multidimensional analysis of bug evolution.

The developed system had to be able to answer a set of analytical questions, including:

- evolution of the number of bugs over time,
- developer performance,
- quality of different software versions,
- distribution of bugs by module,
- analysis of reproducibility and execution environments.

---

3. Overall system architecture

The entire system is based on a classic Business Intelligence architecture consisting of three layers: Extraction, Transformation, and Loading. For this project, we developed an ETL pipeline in Python that automates the entire process. The source code is available in the `etl_pipeline` directory, along with the SQL DDL script for creating the Data Warehouse.

3.1 Extraction

Scribus snapshots are published as CSV files accessible from a web directory. Our pipeline automates their retrieval:

- it connects to the page listing the CSV files,
- identifies new files by comparing them with the local folder,
- downloads only the missing files,
- saves them in a dedicated directory.

To automate the process, we have configured a “cron” job on a Linux server to run the ETL script every week.

The structure of a cron is as follows:

```
- - - - - -
| | | | | |
| | | | +---- Day of week (0 - 7) (Sunday=0 ou 7)
| | | +------ Month (1 - 12)
| | +-------- Day (1 - 31)
| +---------- Hour (0 - 23)
+------------ Minutes (0 - 59)
```

Here is the cron job that runs the script every Monday at 8 a.m. on our server:

```
0 8 * * 1 /home/tottino/Documents/HESProjects/BI--CC/etl_pipeline/etl_pipeline.py
```

3.1.1 Downloading and reading files

`get_csv_from_url(URL, file_path='./data/')`

1. Download the HTML from the `URL` page.
2. Analyze the content with BeautifulSoup to find links to the bug tracker CSV files.
3. Filter the links to keep only files corresponding to Scribus dumps.
4. Compare the list of files found with the files already present in the local `file_path` folder.
5. Download only the missing files, with a progress bar for each file.

It acts as an automated extraction step, replacing the “flat file source + Foreach Loop” combination seen in SSIS.

3.1.2 Reading CSV files

`get_data_from_file(file_path)`

This function:

1. Loads a CSV file into a Pandas DataFrame.
2. Uses a regular expression to extract the snapshot date from the file name (format `YYYY-MM-DD`).
3. If no date is found, uses the current date.
4. Returns the DataFrame and the load date (snapshot).

The extracted date is then used as `SDC_StartDate` in the SCD2 logic of the fact table.

3.2 Transformation

The CSV files contain missing values, heterogeneous text data, and columns requiring type conversion. The following transformations were applied:

- normalization of text values (lowercase, cleaning),
- handling of missing values with standardized values,
- conversion of dates to appropriate types,
- removal of duplicates,
- renaming of columns to match the DWH schema,
- addition of an SDC_StartDate column representing the date of the loaded snapshot.

3.2.1 Data cleaning and preparation

`clean_data(data)`

This function applies the following transformations:

- Replacement of missing values in certain text columns with a standardized value (“Unknown”).
- Conversion of categorical columns (priority, severity, reproducibility, etc.) to lowercase to avoid logical duplicates due to case sensitivity.
- Conversion of date columns (Date Submitted, Updated) to datetime type.
- Removal of any duplicates in the data.

This corresponds to the cleaning and standardization steps presented in the courses (Derived Columns, Data Cleaning).

3.2.2 Preparation for loading

`prepare_data_for_staging(data, loaded_date)`

This function:

- Renames the DataFrame columns to match the DWH nomenclature (e.g., `Id` → `BugId`, `Project` → `ProjectName`, `Reporter` → `ReporterName`, etc.).
- Adds an `SDC_StartDate` column defined as the previously extracted snapshot date.

The idea is to prepare the data to be used for updating dimensions and loading the fact table, in a format consistent with the Data Warehouse schema.

3.3 Loading

Loading is performed in two separate steps: dimensions and facts.

3.3.1 Dimensions (SCD Type 1)

Dimensions are loaded using a mechanism that employs temporary SQL tables and MERGE commands. This approach allows:

- only new distinct values to be inserted,
- a single version per member to be retained,
- any historization in the dimensions to be avoided.

This is equivalent to the Slowly Changing Dimension component in SCD Type 1 mode in SSIS.

`_merge_simple_dimension(...)`

This utility function is used for all simple dimensions (Project, Priority, Severity, Reproducibility, Version, Category, Status, etc.). How it works:

1. Build a list of unique values (text) from a Pandas series, eliminating null values or `"Unknown"`.
2. Create a temporary SQL table (`#StageXXX`) with a single NVARCHAR column.
3. Insert all unique values into this temporary table.
4. Execute a `MERGE` command between the temporary table and the target dimension, inserting only values that are missing from the dimension.
5. Delete the temporary table.

This results in behavior equivalent to filling an SCD1 dimension in SSIS: no history, only new values are added.

`update_dimensions_staging(data, db_connector)`

This function:

- Creates an SQL cursor on the connection.
- Calls `_merge_simple_dimension` to populate the dimensions:
    - DimProject (ProjectName)
    - DimUser (ReporterName and AssigneeName)
    - DimPriority
    - DimSeverity
    - DimReproducibility
    - DimVersion (ProductVersionName and VersionFixedName)
    - DimCategory
    - DimStatus (ViewStatusName, StatusName, ResolutionName)
- Processes the DimOs dimension separately, which depends on several columns (Platform, OS, OSVersion), also using a temporary table and a MERGE.
- Updates DimCalendar by inserting all relevant dates (submission, update, SDC start, and SDC end dates). The dates are converted to a DateId key in `YYYYMMDD` format and inserted via a temporary table and a MERGE.

This step ensures that all dimensions required by FactBug are complete and up to date before the facts are loaded.

3.3.2 Fact table (SCD Type 2)

The FactBug table keeps a history of a bug's status over time.
For each snapshot loaded:

1. Existing rows corresponding to the same BugId and marked as IsCurrent are updated:
    - IsCurrent is set to 0,
    - SDC_EndDate is set to the day before loading,
2. A new version of the bug is inserted:
    - IsCurrent = 1,
    - SDC_StartDate corresponds to the snapshot date,
    - The surrogate keys of the dimensions are resolved via joins.

`load_fact_snapshot_scd2_staging(data, db_connector)`

This function performs the core SCD2 logic of our ETL:

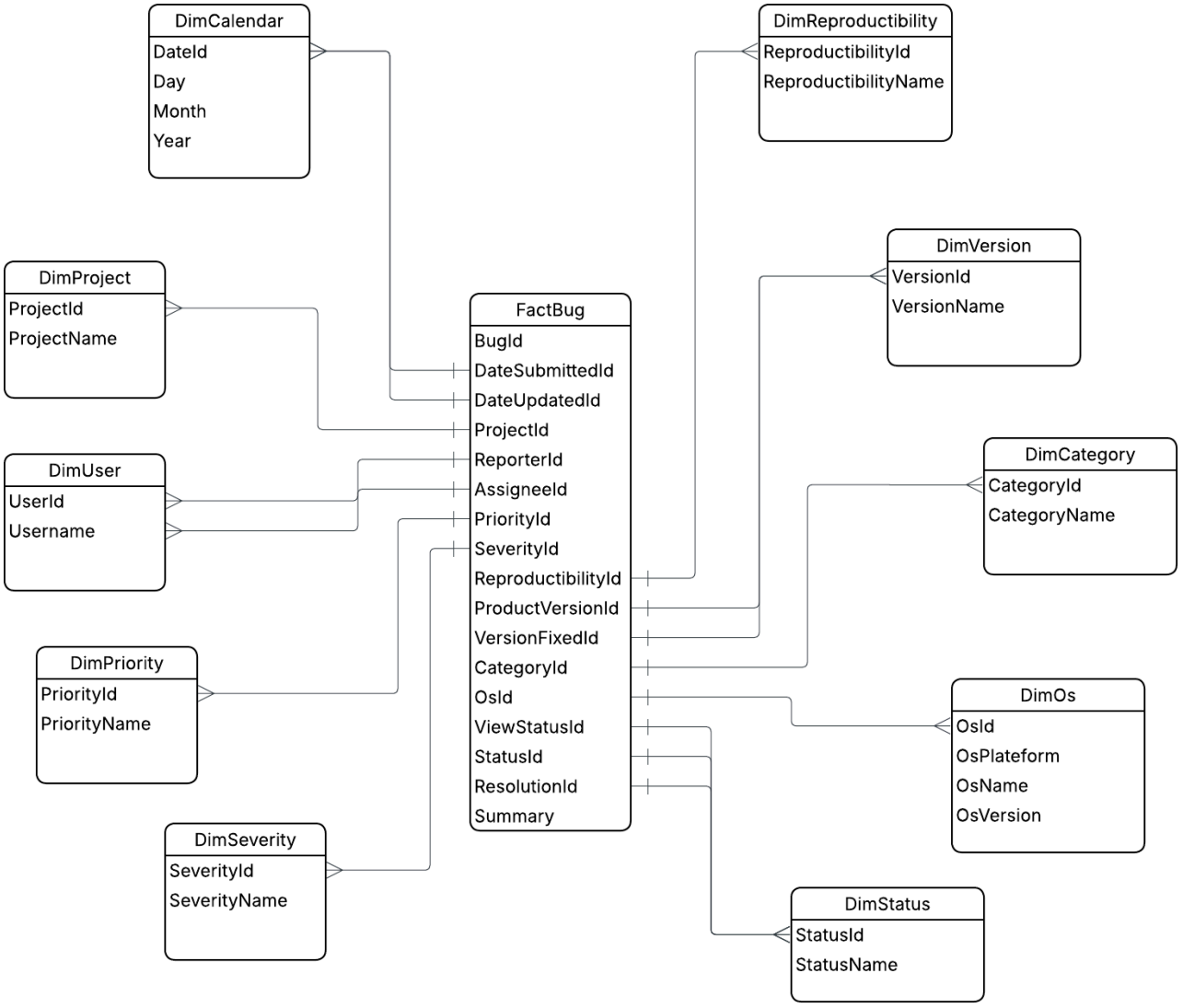
1. Data preparation:
    - Addition of date columns formatted for SQL (DateSubmitted_SQL, DateUpdated_SQL, SDC_StartDate_SQL).
    - Replacing missing values with `None` (SQL NULL).
2. Creating a temporary table `#StageFact` in SQL, with the necessary columns (BugId, text names, dates, etc.).
3. Inserting all rows from the DataFrame into `#StageFact`.
4. Closing old SCD2 versions:
    - Calculating `SDC_EndDate` as the day before `SDC_StartDate`.
    - Updating `FactBug` for all rows with the same BugId and `IsCurrent = 1`, setting `IsCurrent = 0` and filling in `SDC_EndDate`.
5. Inserting new rows:
    - `INSERT INTO FactBug` by selecting from `#StageFact` and joining all dimensions (DimProject, DimUser, DimPriority, DimSeverity, DimReproducibility, DimVersion, DimCategory, DimStatus, DimOs, DimCalendar).
    - The joins provide the necessary surrogate keys for the fact table.
    - The new rows have `IsCurrent = 1`, `SDC_StartDate` filled in, and `SDC_EndDate` set to NULL.
6. Delete the temporary table and commit the transaction.

The result is a historical fact table that keeps all successive versions of the same bug, consistent with the SCD Type 2 concept.

---

4. Data Warehouse Modeling

For the modeling part, we designed a star schema centered on the FactBug table.



4.1 Fact Table: FactBug

FactBug contains the historical facts and all keys to the dimensions.

4.2 Dimensions

The following dimensions were created and populated:

- DimProject
- DimUser
- DimPriority
- DimSeverity
- DimReproducibility
- DimVersion
- DimCategory
- DimStatus
- DimOs
- DimCalendar

5. Creating the analytical model (OLAP)

5.1 Choosing the Tabular model

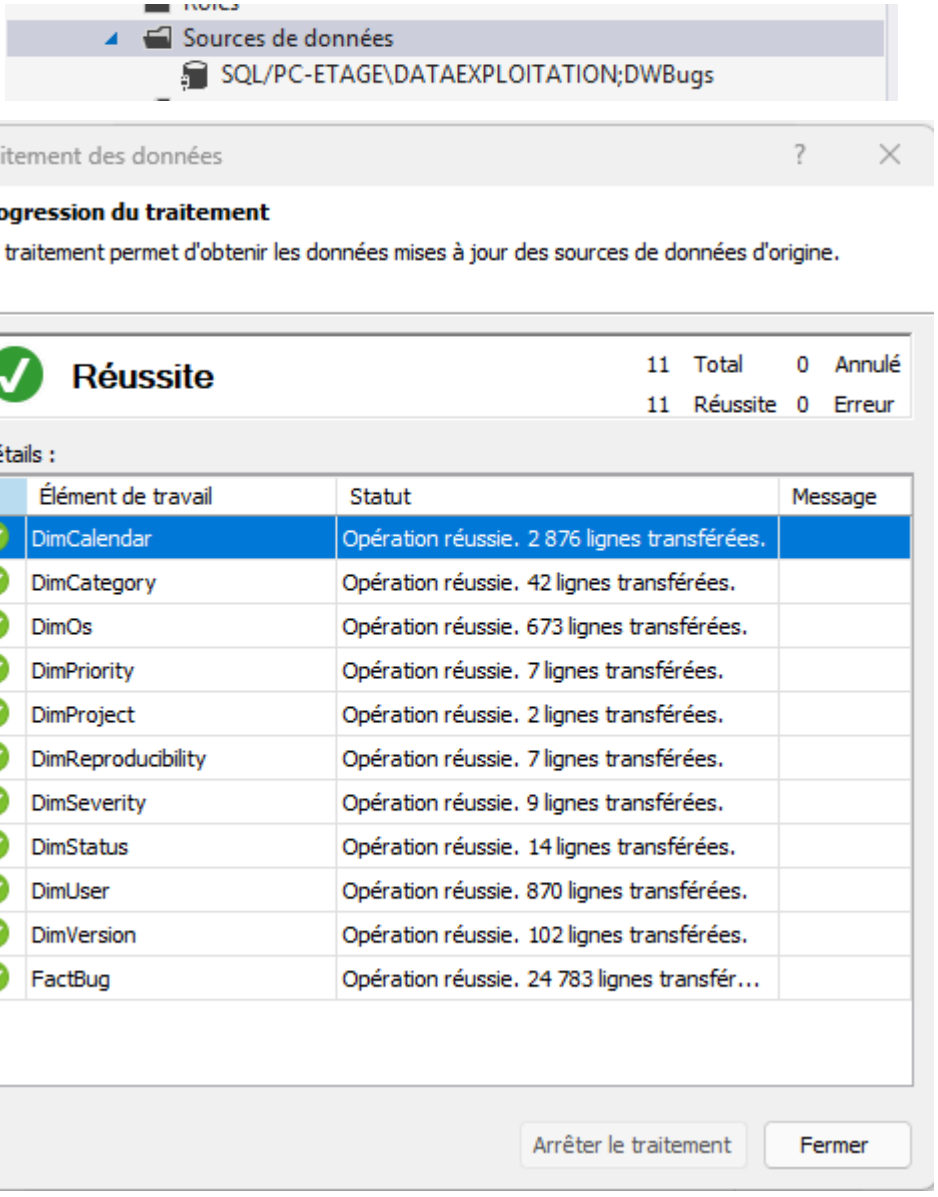
In SSAS (SQL Server Analysis Services), you have two main options for creating a model: Tabular and Multidimensional. For this project, we opted for the Tabular (DAX) because it is more modern, easier to use, and can be integrated with Power BI (which is not possible with Multidimensional).

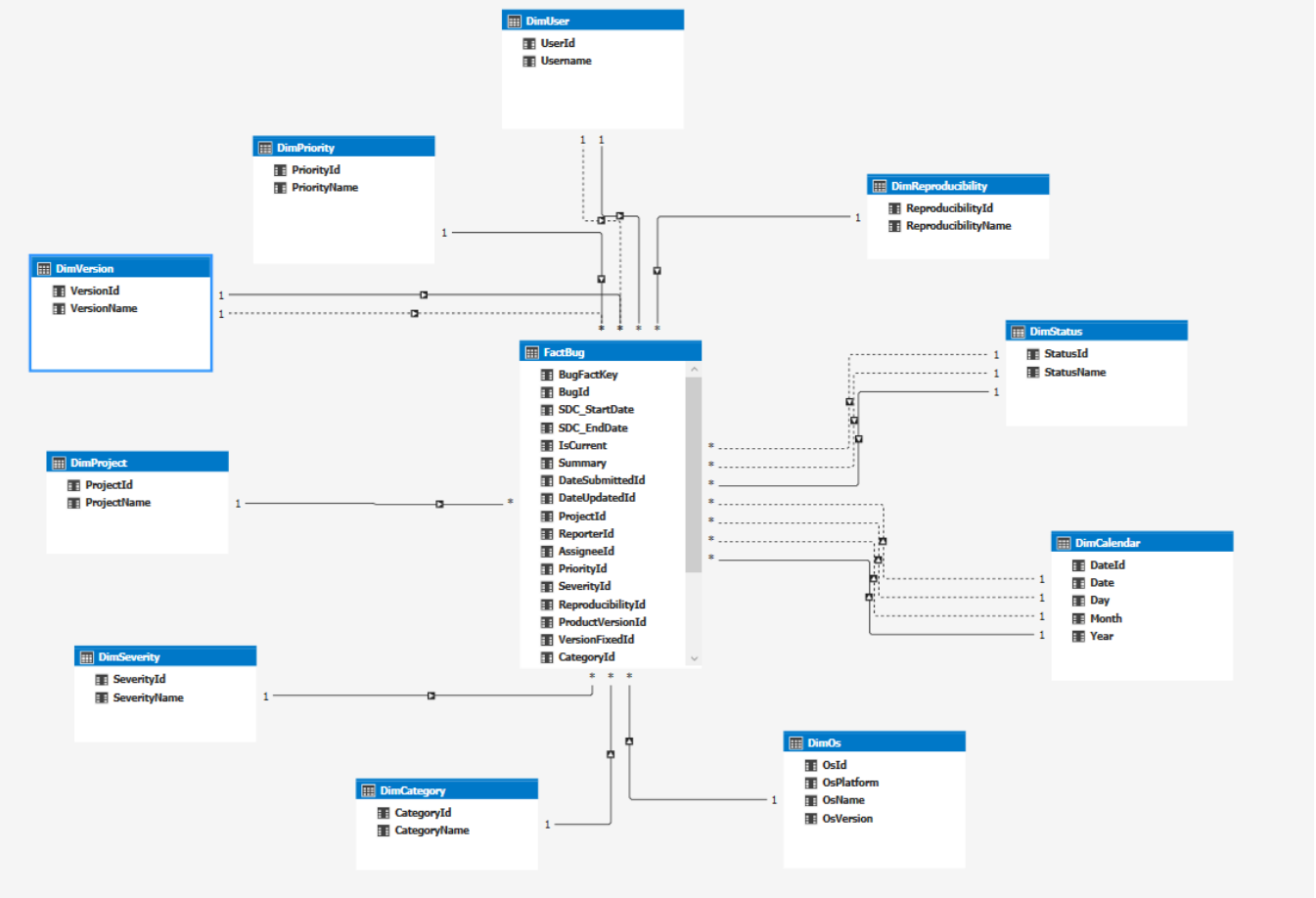
Tools used:

- Visual Studio 2022 with SQL Server Data Tools (SSDT) extension
- SQL Server Analysis Services (SSAS)

5.2 Building the model in Visual Studio

Importing tables, creating relationships, managing inactive relationships.

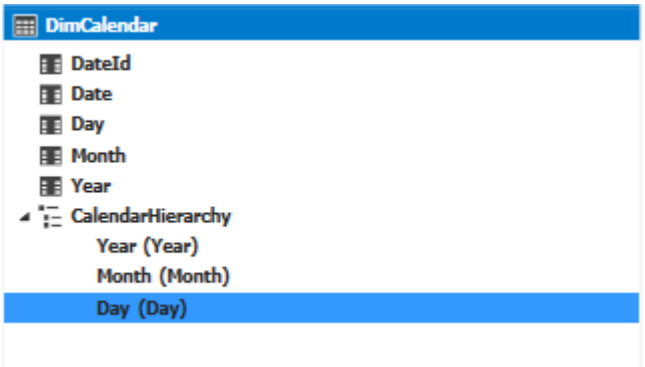


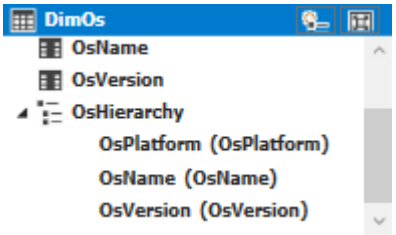


In order to calculate some measures, we had to modify some relationships to define which ones needed to be active or inactive.

5.3 Hierarchies and perspectives

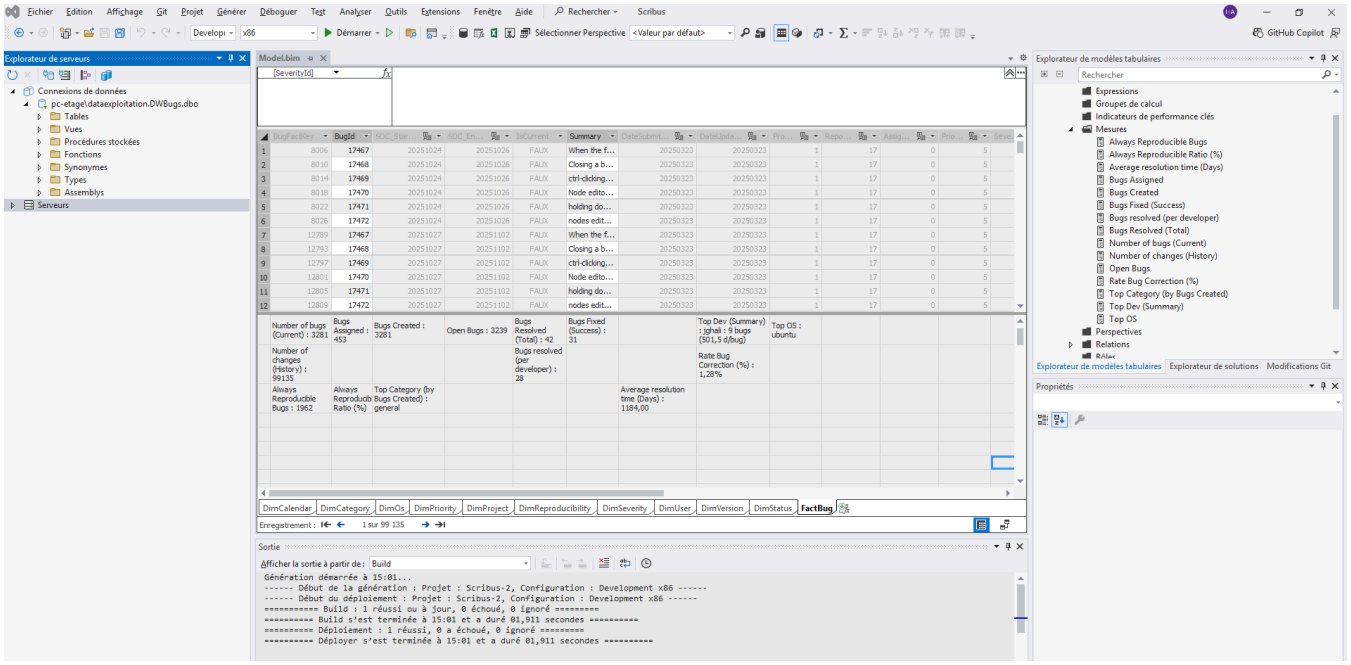
- calendar: Year > Month > Day
- operating system: Platform > OS > OSVersion





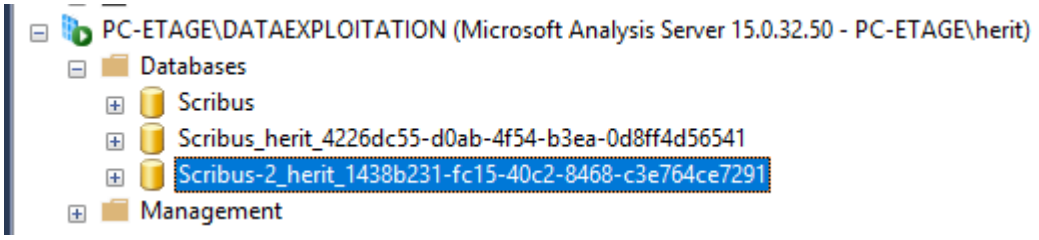
5.4 Deployment and DAX measures

Before deploying the model, we created several DAX measures to meet the analytical needs outlined in section 2.



5.5.1 Model deployment

Deploying the model to SSAS for use via Power BI, we can see the analytical base on the server:



5.5.2 Detailed DAX measures

```
Number of bugs (Current):=
CALCULATE(
    DISTINCTCOUNT(FactBug[BugId]),
    FactBug[IsCurrent] = TRUE()
)
```

```
Number of changes (History):= COUNTROWS(FactBug)
```

```
Bugs Assigned:=
CALCULATE(
    [Number of bugs (Current)],
    FactBug[IsCurrent] = TRUE(),
    FactBug[AssigneeId] <> 0
)
```

```
Bugs Created:=
CALCULATE(
    [Number of bugs (Current)],
    USERELATIONSHIP(FactBug[DateSubmittedId], DimCalendar[DateId])
)
```

```
Open Bugs:=
CALCULATE(
    [Number of bugs (Current)],
    USERELATIONSHIP(FactBug[ResolutionId], DimStatus[StatusId]),
    NOT(DimStatus[StatusName] IN {
        "fixed",
        "resolved",
        "won't fix",
        "unable to reproduce",
        "no change required"
    })
)
```

```
Bugs Resolved (Total):=
CALCULATE(
    [Number of bugs (Current)],
    USERELATIONSHIP(FactBug[ResolutionId], DimStatus[StatusId]),
    DimStatus[StatusName] IN {
        "fixed",
        "resolved",
        "won't fix",
        "unable to reproduce",
        "no change required"
    }
)
```

```
Bugs resolved (per developer):=
CALCULATE(
    [Bugs Resolved (Total)],
    FactBug[AssigneeId] <> 0
)
```

```
Bugs Fixed (Success):=
CALCULATE(
    [Number of bugs (Current)],
    USERELATIONSHIP(FactBug[ResolutionId], DimStatus[StatusId]),
    DimStatus[StatusName] IN {"fixed", "resolved"}
)
```

```
Average resolution time (Days):=
AVERAGEX(
    FILTER(
        FactBug,
```

```
        FactBug[IsCurrent] = TRUE() &&
        (
            VAR ResStatus = CALCULATE(
                SELECTEDVALUE(DimStatus[StatusName]),
                USERELATIONSHIP(FactBug[ResolutionId], DimStatus[StatusId])
            )
            RETURN ResStatus IN {"fixed", "resolved"}
        ) &&
        // --- SÉCURITÉ ANTI-1900 ---
        FactBug[DateSubmittedId] <> 0 &&
        FactBug[DateUpdatedId] <> 0 &&
        FactBug[DateUpdatedId] >= FactBug[DateSubmittedId] // Protection contre les dates
inversées
    ),

    // Calcul Mathématique (ID vers Date)
    VAR StartID = FactBug[DateSubmittedId]
    VAR EndID = FactBug[DateUpdatedId]

    VAR StartDate = DATE(INT(StartID / 10000), INT(MOD(StartID, 10000) / 100), MOD(StartID,
100))
    VAR EndDate =   DATE(INT(EndID / 10000),   INT(MOD(EndID, 10000) / 100),   MOD(EndID,
100))

    VAR Ecart = DATEDIFF(StartDate, EndDate, DAY)

    RETURN
    IF(Ecart = 0, 1, Ecart)
)
```

```
Top Dev (Summary):=
VAR DevStats =
    ADDCOLUMNS(
        VALUES(DimUser[UserId]),
        "UserName", CALCULATE(MAX(DimUser[Username])),
        "NbResolved", [Bugs resolved (per developer)],
        "AvgTime", [Average resolution time (Days)]
    )
VAR DevFiltered =
    FILTER(
        DevStats,
        [NbResolved] > 0 && NOT ISBLANK([AvgTime])
    )
VAR BestDev =
    TOPN(
        1,
        DevFiltered,
        [NbResolved], DESC, // Priorité au nombre de bugs
        [AvgTime], ASC      // Puis à la vitesse (le plus petit temps est le mieux)
    )
VAR TheName = MAXX(BestDev, [UserName])
VAR TheCount = MAXX(BestDev, [NbResolved])
```

```
VAR TheTime = MAXX(BestDev, [AvgTime])

RETURN
    IF(
        ISBLANK(TheName),
        "Aucune donnée",
        TheName & " : " & FORMAT(TheCount, "0") & " bugs (" & FORMAT(TheTime, "0.0") & "
d/bug)"
    )
```

```
Rate Bug Correction (%):=
DIVIDE(
    [Bugs Resolved (Total)],
    [Bugs Created],
    0
)
```

```
Top Category (by Bugs Created):=
VAR TopCat =
    TOPN(1, SUMMARIZE(FactBug, DimCategory[CategoryName], "Bugs", [Bugs Created]), [Bugs],
DESC)
RETURN
CONCATENATEX(TopCat, DimCategory[CategoryName], ", ")
```

```
Always Reproducible Ratio (%):=
DIVIDE(
[Always Reproducible Bugs],
[Number of bugs (Current)],
0)
```

```
Always Reproducible Bugs:=
CALCULATE(
    DISTINCTCOUNT(FactBug[BugId]),
    DimReproducibility[ReproducibilityName] = "always"
)
```

5.5.2 Deployment of the ETL script

The Python ETL script is automated via a cron job on a VPS server, executed weekly to load new snapshots.

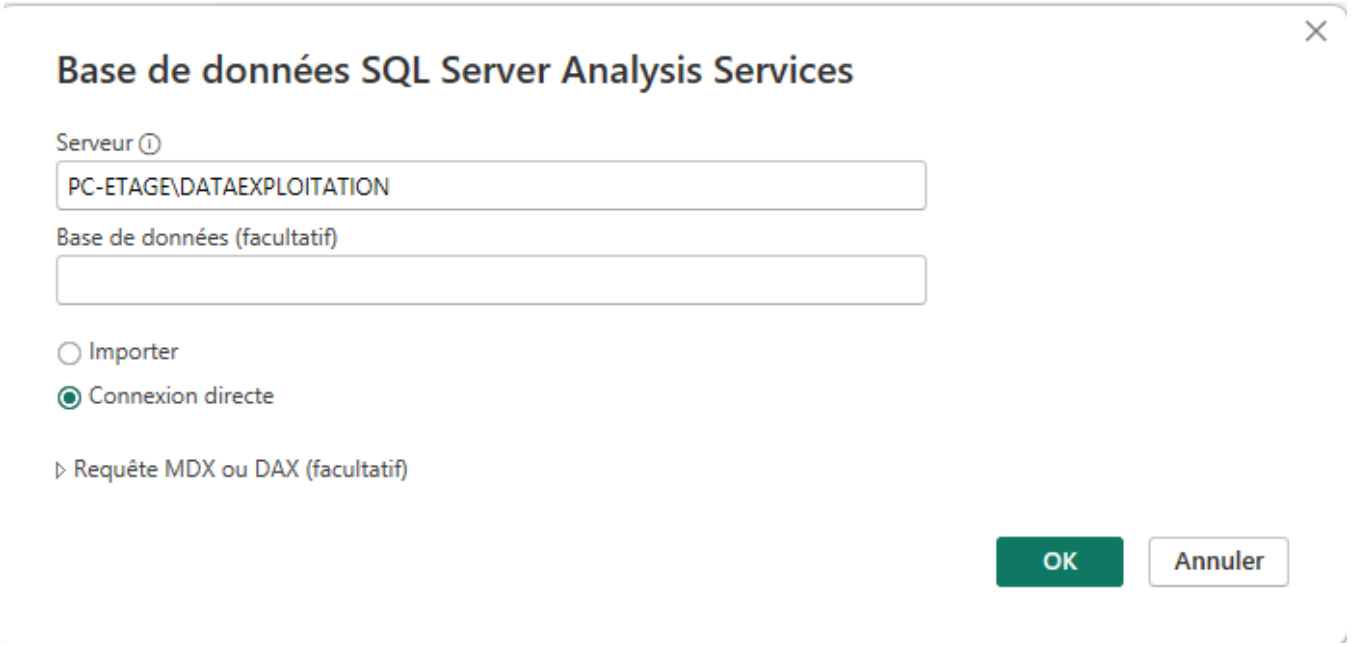
Since files can only be retrieved from the school network, it is currently automated on a local workstation.

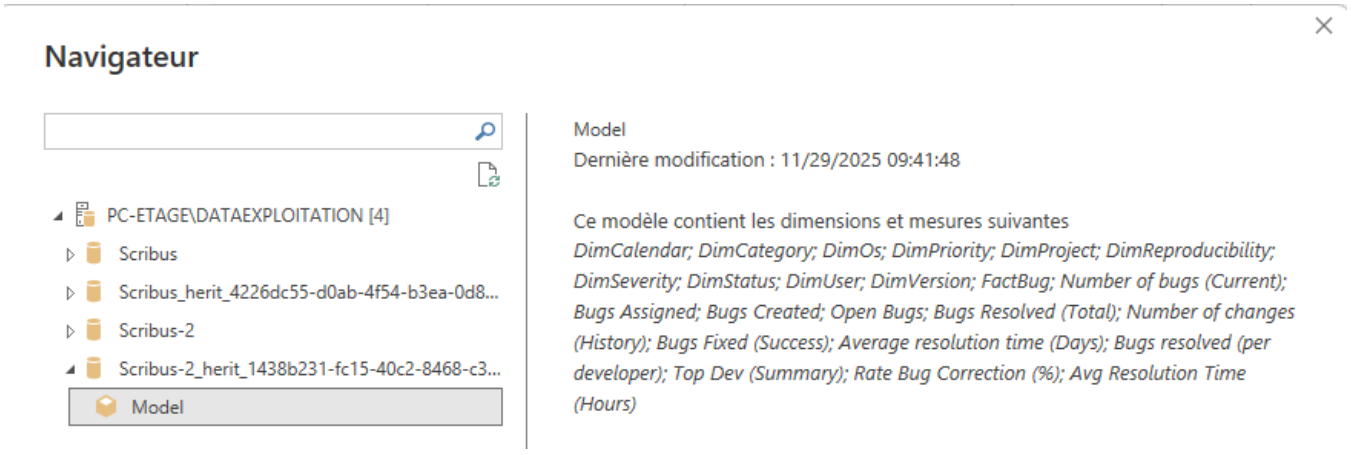
---

6. Analysis and exploitation

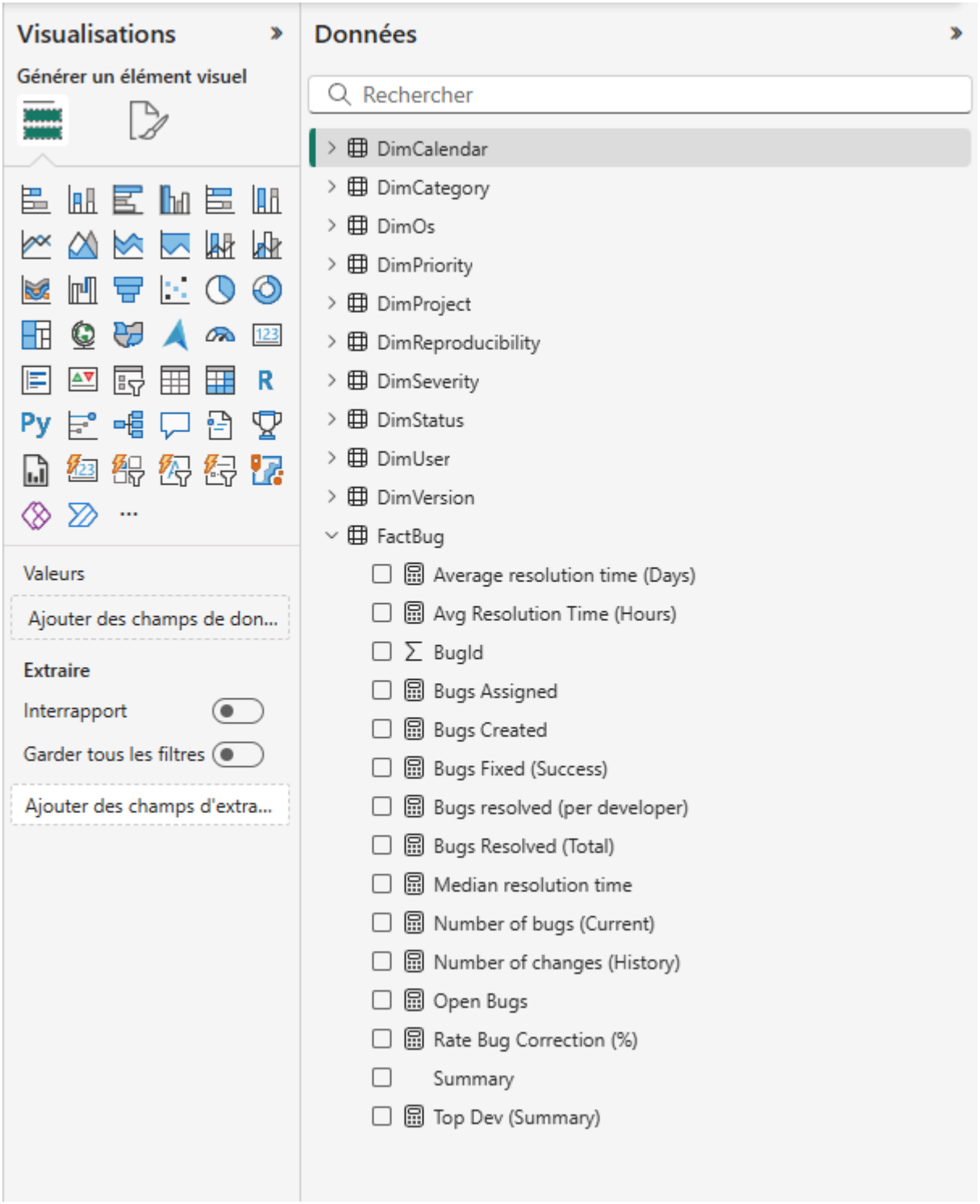
6.1 Connection with Power BI

Once the model was deployed on SSAS, we were able to connect with Power BI to create analytical reports.

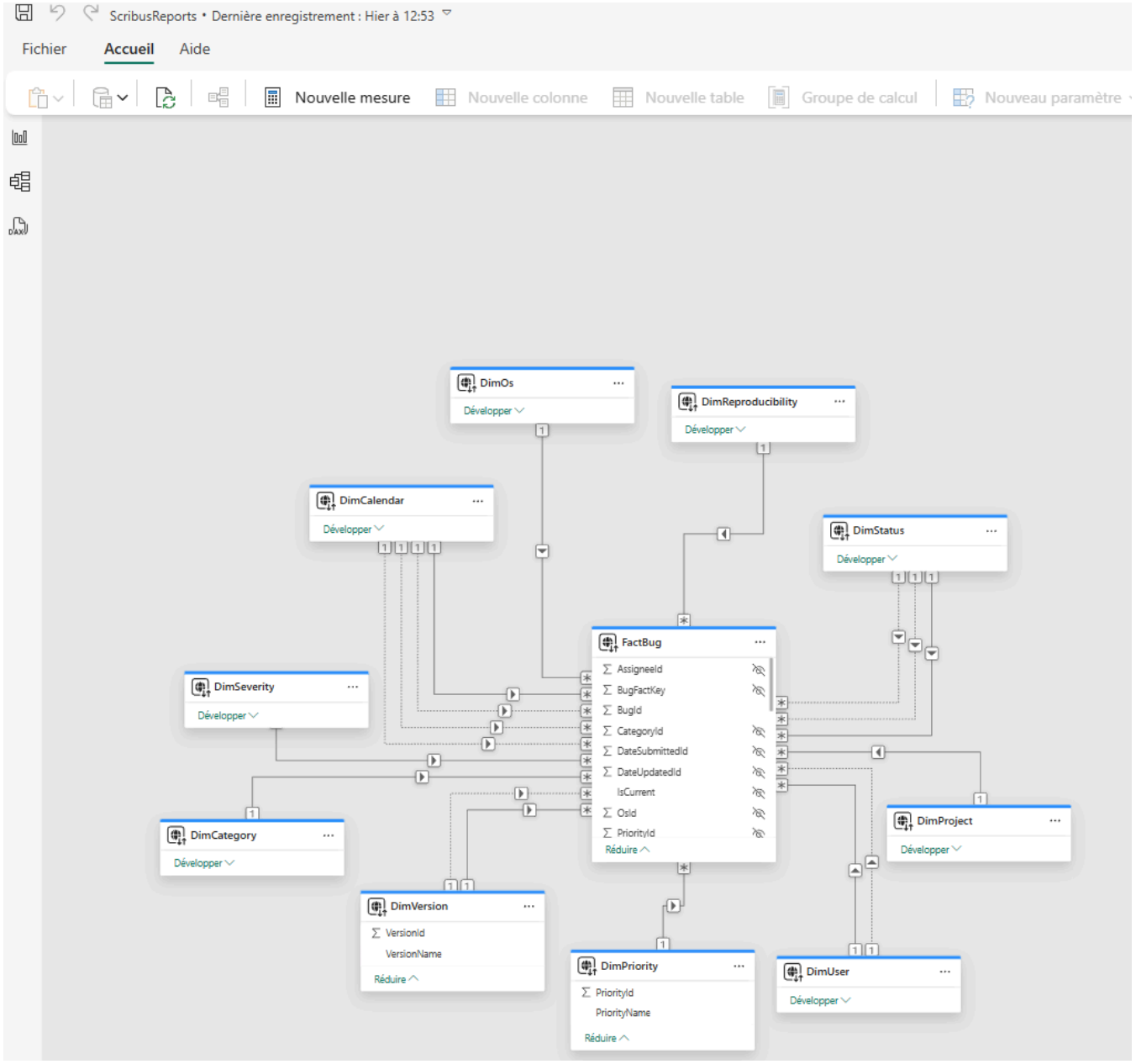




We can see that all DAX tables and measures are available for analysis:



And we can find the model view also :

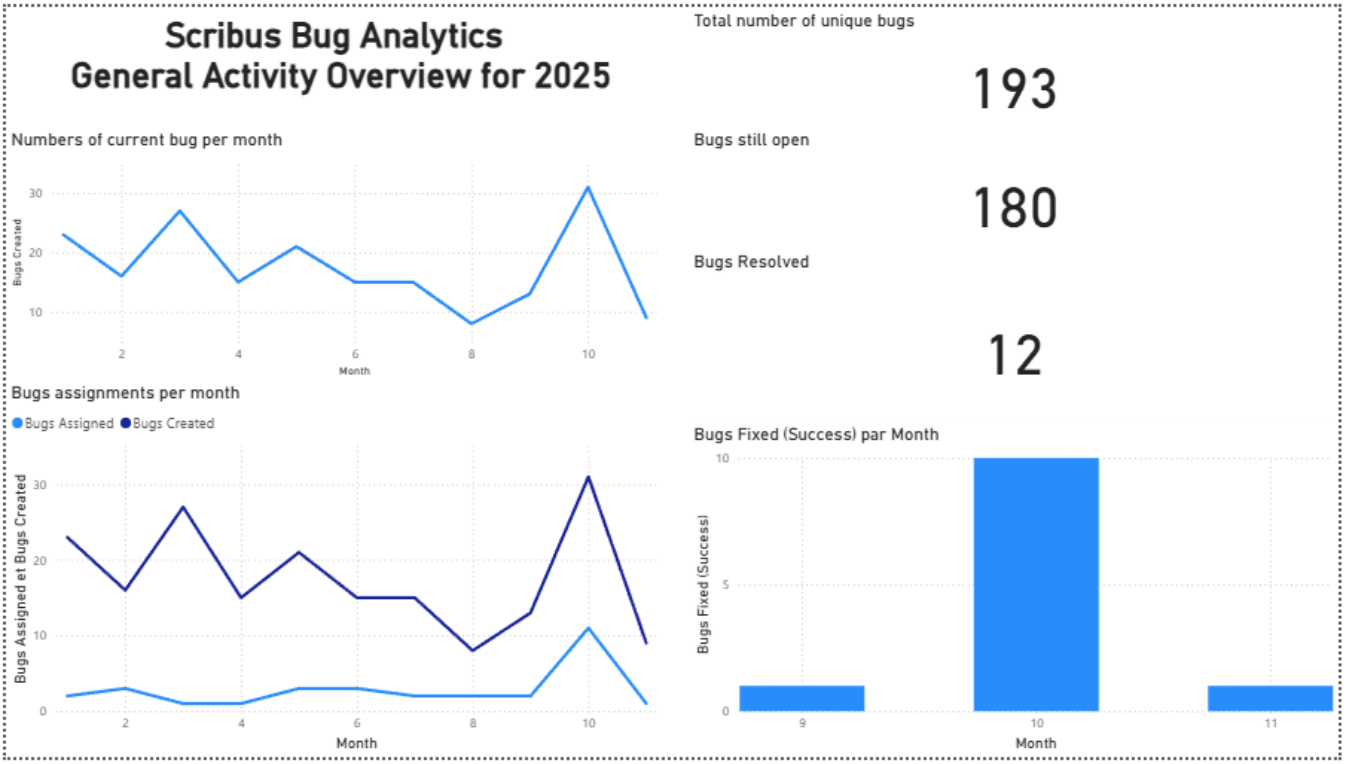


6.2 Creating analytical reports

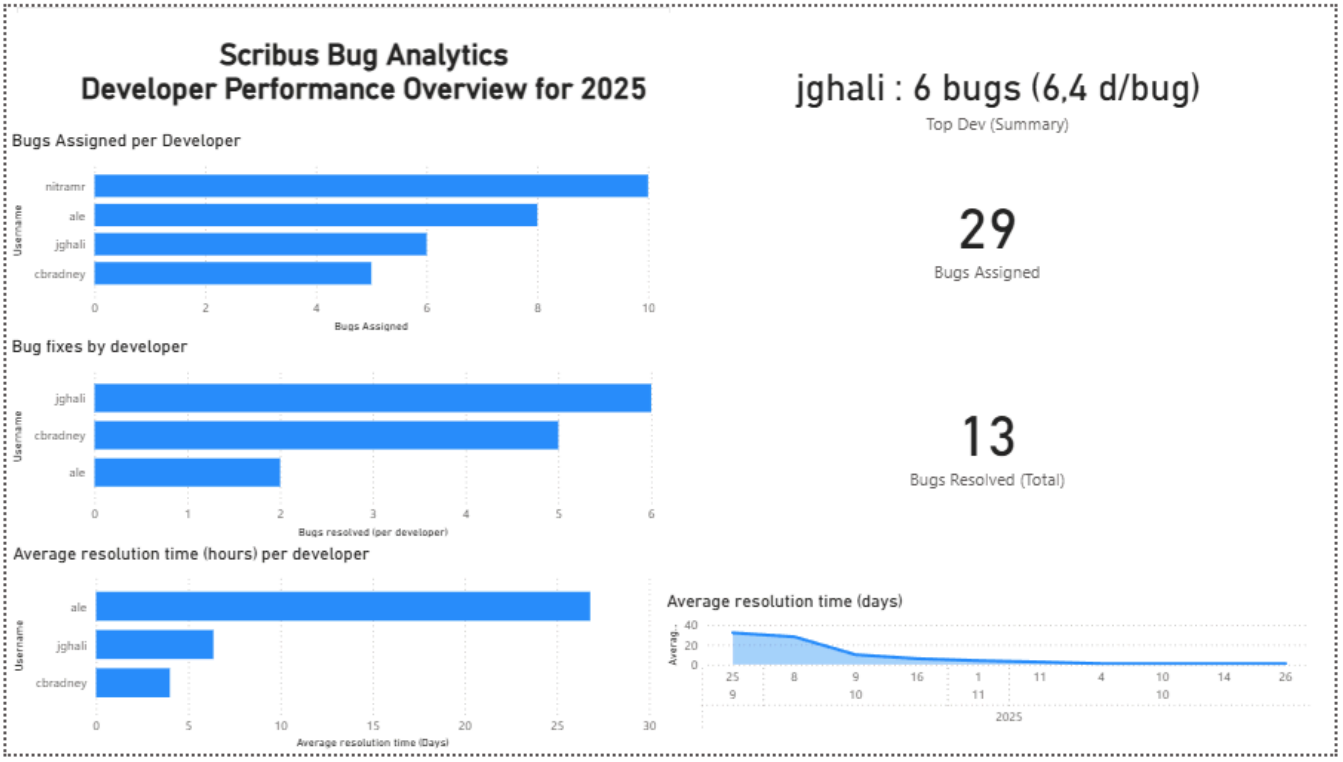
Using DAX measures and model dimensions, we created several reports to analyze data from the Scribus bug tracker.

Since the data for previous years is not very relevant (no resolutions, no assignments, etc.) and since updates have been coming in since this year, we have created reports for the current year (2025):

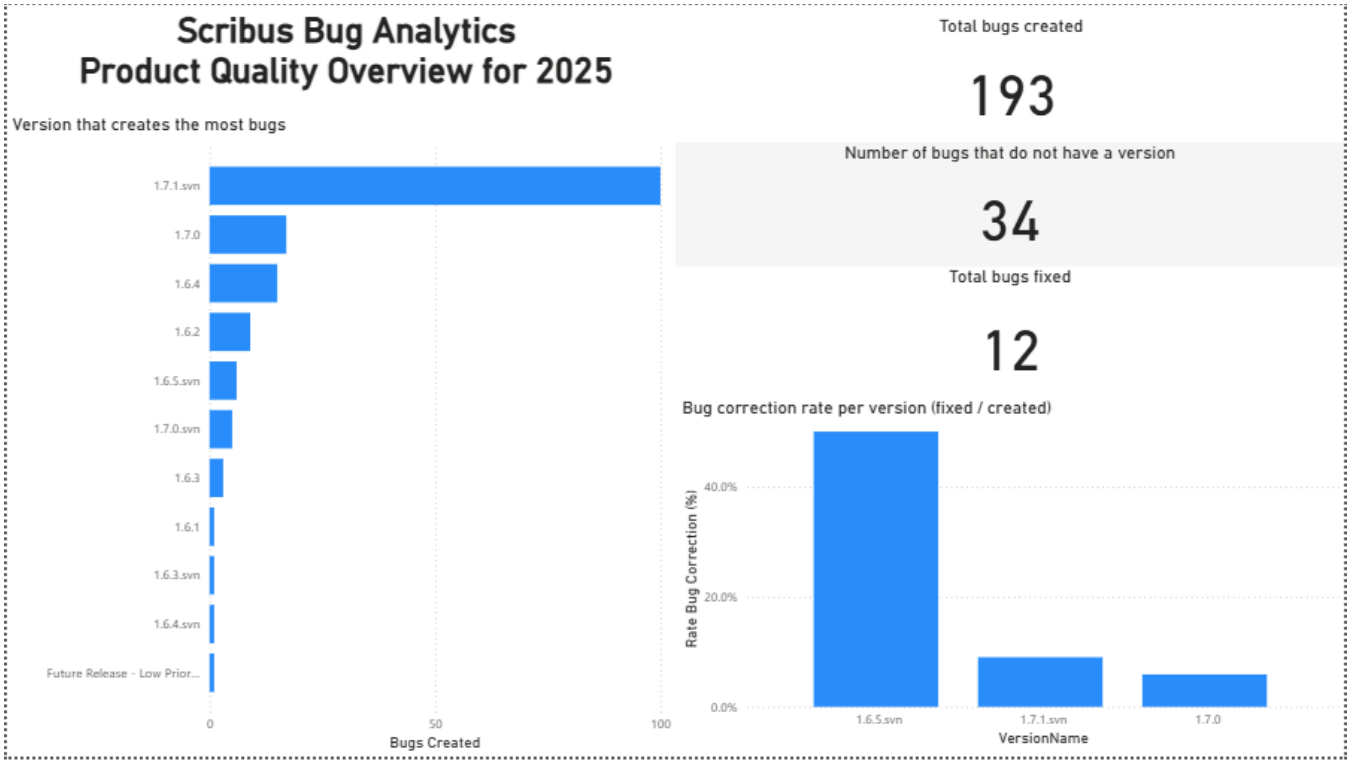
- General activity:



- Developer performance:

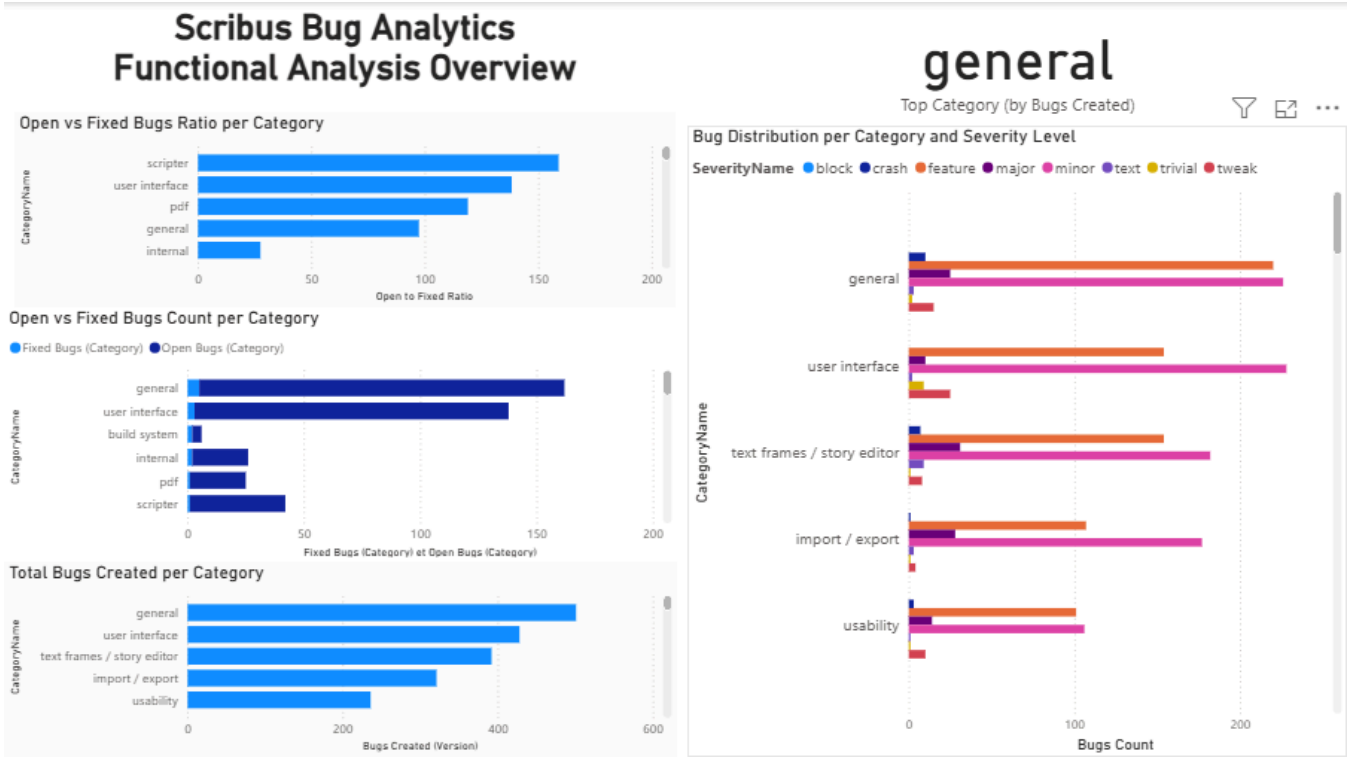


- Version quality:

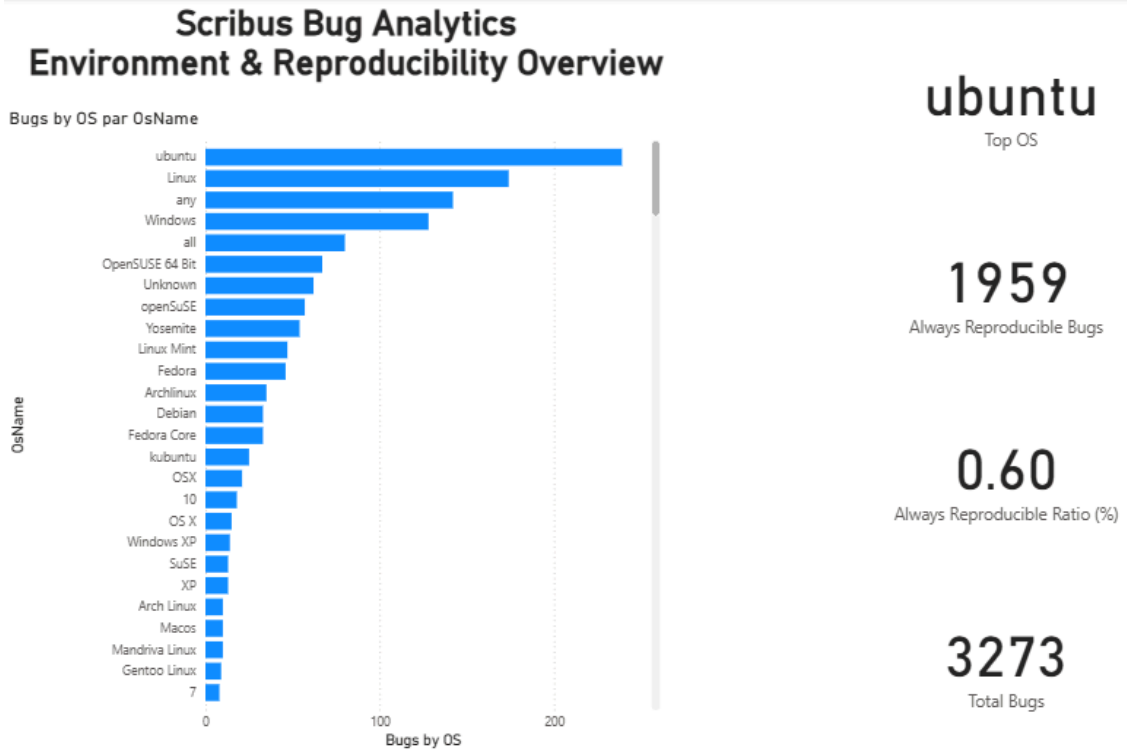


It is very easy to adapt these graphs to analyze other years; it's simply a matter of filters.

- Functional Analysis:



- Environment & Reproductibility:



7. Conclusion

The project implements the entire decision-making chain: ETL, SCD, DWH, OLAP Tabular, and analytical measures.

This project demonstrates the successful implementation of a complete Business Intelligence pipeline, from raw data extraction to analytical reporting. Starting from unstructured CSV snapshots published by the Scribus project, we designed and implemented a robust ETL process capable of cleaning, transforming, and historizing bug data through an SCD Type 2 fact table.

The resulting Data Warehouse, modeled using a star schema, provides a stable and consistent foundation for analytical workloads. The construction of a Tabular OLAP model in SSAS further enabled the creation of advanced DAX measures tailored to real analytical needs such as developer performance, product quality, bug resolution trends, and environment-specific issues.

The integration with Power BI allowed us to produce clear, interactive dashboards that answer the project's key analytical questions. These dashboards highlight important insights, such as the evolution of bug creation over time, the performance of Scribus developers, and the stability of specific software versions and operating systems.

Beyond fulfilling the academic requirements, this project provided hands-on experience with essential BI concepts: ETL design, SCD management, dimensional modeling, DAX development, and OLAP cube deployment. It also demonstrated how structured analytical systems can transform raw operational data into meaningful information that supports decision-making.

Overall, the project successfully delivers a complete, end-to-end BI solution that is maintainable, scalable, and ready to support further analytical extensions in the future.