Tennis Data Warehouse
Laboratory of Data Science 2021/22

Marianna Abbattista[1] and Fabio Michele Russo[2]

Contacts: [1]m.abbattista1@studenti.unipi.it, [2]f.russo11@studenti.unipi.it

December 30, 2021

Contents

1 Introduction

In this short report we will detail our steps in making the Data Warehouse required by the project assignment. We have 6 sections, an introduction and one section for every ETL area we have worked on, and we will refer the reader to specific parts of the code files with a refer block, such as this one:
`divide.py 143` would mean that the reader is invited to look at line 143 of the file divide.py.

1.1 Project organization

As we work collaboratively, we have to agree among ourselves on a project file structure so that it's always clear which files are unprocessed, which are processed, which are split and ready for **loading**. Therefore, we have divided the *data* folder into:

1. *data/inputs/* are the data files as given to us by the client, untouched;
2. *data/work/* are the work-in-progress data files, not ready for loading, but ready for the next processing steps;
3. *data/* contains only the csv files ready to be **loaded** onto the SQL Server.

1.2 How to execute the pipeline

We have a pipeline of **4 python files**. Again, this to allow for easier collaboration amongst ourselves; however, the 4 files could be collected into one if needed.
The data has to be put into `data/inputs/`. All the data is in there already, except for `tennis.csv`, for space reasons. Requirements, in the form of libraries, are listed in `requirements.txt`.
So to execute the pipeline, you must:

1. Add `tennis.csv` into `data/inputs/`;
2. In the terminal: `pip3 install -r requirements.txt` to install the required libraries;
3. To execute the actual project: `python3 preprocessing.py` (or `python`, depending on your configuration);
4. `python3 divide.py`;
5. `python3 postprocessing.py`;
6. `python3 load.py`.

2 Data Understanding and Pre-Processing

Starting to look at the csv files, we notice that there are some syntactic inconsistencies in some rows, so we decide to treat it in the file `preprocessing.py`.

- *countries.csv*: there are five countries names that are syntactically incorrect (**Urugay** for example). What we decide to do? Since for filling languages we have decided to use *countryinfo.tsv* we made a comparison between the corresponding values of the names and the five values of *countries.csv* and correct it. `preprocessing.py 19-20`
- *tennis.csv*: there are missing values in the post processing analysis, but we decide to fill this particular part here on tennis table so as not to have future problems. The columns *winner_rank*, *winner_rank_points*, *loser_rank* and *loser_rank_points* cover more than the 50% of the dataset so we decide to fix it, filling them with the mean **for that tourney level** for that *winner_rank*, for example. We do that also for *winner_rank_points*, *loser_rank* and *loser_rank_points*. In this way we no longer have missing values.

3 Splitting Tables

As already mentioned we start with a "big table" *tennis.csv* file and we need to partition it in several tables according to the required structure. However, we have made a couple of changes to it due to constraints we encountered while working, which we detail precisely in this chapter.
We plan to make, as output of this section, as many csv files as tables we will have in the server - so, **five**. The headers of said tables are manually defined by slicing the original *tennis.csv* header. `divide.py 11-25`

3.1 Table *Date*

We have in *tennis* a date for each tournament in the format **yyyymmdd**. Therefore, this can already be seen as unique for each date and therefore a key for a table *Date*, so we take this field as primary key of table *Date* and as required add columns for year, quarter, month, day by string-splitting the aforementioned key and creating **datetime** objects from it, that we then assign into the correct columns. `divide.py 289-327` The table *Tournament* will link to this table with a foreign key *date_id* that is exactly the old column *tourney_date* renamed.

3.2 Table *Geography*

We start with a few different inputs in this case: a *countries.csv* containing country codes, names and continents and a *countryinfo.tsv* containing for each country which languages (codes) are spoken there.

3.2.1 Error correction

There are a few countries in the *countries.csv* input which either have a wrong name, an *unknown* name or a name that does not correspond to the name of the country in *countryinfo.tsv*. We have corrected these errors `xxx xxx` so that to allow a simple join of languages from the tsv into table Countries. `divide.py 258-287`
We chose to leave the languages as language codes because languages are fairly hard to manage. The flavor of English spoken around the world can be very different, and that is why English is encoded as en-US, en-AU, en-UK, etc. This happens for many languages. The best solution that we could envision for the client is to make a separate table, *Languages*, and have in that a row for each language code, language name and language territory, like so:
en-US,**English**,**United States**
This would allow for **easy aggregation** on languages in the finished product. However, since this in our opinion goes a bit too far beyond the scope of the assignment, we left language **codes** for now.

3.3 Table *Player*

We have to take into account that players appear as either **winners** or **losers** of each match. Therefore, we have to extract them from both groups into a separate csv and being careful not to add duplicate entities to it. Therefore, we scan *tennis.csv* and add into a python **set** each new player ID while writing the corresponding player into the new csv. `divide.py 90-133`
We test our solution `divide.py 135-149` by using the pandas library (only for testing), that we have as many lines into the csv as many unique players we found and that we actually found **all unique players**.

3.3.1 How to do duplicate removal

Something of note is that when building *Player*, we took the first instance of each *player_id* we found in either winners or losers and copied those values to the Player table. However, by doing this we might miss some fields for certain players: let's imagine that the same player appears twice with the same ID but, in the first instance it only has the Name, and in the second it also has all other fields. With our procedure, we would take the first of these two rows and therefore lose all other relevant fields that appear in the second of the two instances. So, we checked whether this happened often: it turns out that it happens very rarely. We calculated that by implementing this kind of check on duplicate removal we would "recuperate" only about 30 values of height and just a few of other fields. Considering that we actually delete the column *height* because of its abysmal coverage and considering also that this kind of check would complicate heavily the data preparation process, we chose to skip it.

3.4 Tables *Tournament* and *Match*

The *Tournament* table is extracted by renaming *tourney_date* to *date_id* and getting all appropriate attributes, while for matches we aimed to a one-to-one correspondence with rows of input *tennis*, with the issue of which primary key to use. `divide.py 150-153, 28-71`

3.4.1 The problem of *Match*'s primary key

The original instruction was to make a *match_id* by concatenating *tourney_id* and *match_num*. However, this does not make a unique key for this table as there would only be 158393 unique rows out of 186073.
We revised the concatenation to add *tourney_level* as well, but it still was not sufficient for a unique key (only 161817 rows). We considered that concatenating more and more attributes would have made a fairly long and complex primary key without enough of an upside. Therefore, we chose to keep all original attributes in the table (*match_num*, *tourney_level* etc.) and to make a simple **sequential primary key 0,1,...,n_rows-1**. This in our view is the most straightforward way to ensure the uniqueness of the key and that it would not need to be changed in the future.

4 Data Post-Processing

After the splitting part we decide to check all our tables, and observe if there are or not some type of missing value or inconsistency that could create problems in the construction of the Data warehouse and in the subse-

quent management of these. So we decided to look at the tables individually and at the end of this paragraph you can observe all the single table with Column, data type and NValue.

- *Date*: since it is custom made, doesn't require any additional treatment.
- *Geography*: like we said in the pre-processing part we fill the country name written wrong, but we notice that there's another country with *country_ code*, that is our primary key, **POC** that doesn't have any value, so we fill it manually searching what state for, and we discover that is Pacific Oceania, so we fill it `postprocessing.py 43-45`. After that we notice that the Kosovo *country_ code* was written wrong, so we fixed it. After that we decide to add another row that we call "Unknown", with *country_ code* **UNK**, (`postprocessing.py 52-53`) because we notice that in *players.csv* there are some players that don't have a *country_ code* that match with the value in *country_ code* in *countries.csv*. (In the *players.csv* part we explain better this part.)
- *Tournament*: we notice that there is only one column to be treated ,*surface* has 1.3% of missing values. So we decide to fill it making the mode of the values (that are categorical) and insert **Hard** into all the rows that doesn't contains a value `postprocessing.py 87-89`.
- *Player*: we first decide to do some check in the columns *country_ name* and control if there are name that starts with a formalism like 'mr,miss,master, Mr,...'. from analyzing which are the total values percentage of the single columns that have missing values we discover that:
  - *ht* : has 94.8 % of missing values so we decide to delete it.
  - *gender* : has 0.3% of missing values, that is 30 values out of 10074 that is the number of unique *player_ id* . So we decide to fill it manually, looking at the single player and inserting **M** or **F** `postprocessing.py 119-123`.
  - *hand* : has 0.3% of missing , so we do the mode of the hand value(that is a categorical variable and could be **L**,**R** or **U**) , we insert **U** in that 33 values , that stays for **unknown'**`postprocessing.py 128-131`.
  - *yob* : has the 20.9% , that is 2108 values out of 10074, so we fill it with the value **Unknown** because there's not a reasonable way to fill it manually.

Last things we do with *player.csv* table is to substitute the *country_ code* that doesn't match with the column *country_ code* in the *countries.csv* table. We discover that there are 68 *country_ code* of this type, and we replace it with the foreign key **UNK** `postprocessing.py 150-162`.

- *Match* has a lot of missing value to be treated:
  - *minutes* : have the 56.1 % of values so we decide to fill the remaining value with **-1** because we felt that it was not correct to calculate its value using the average or something else to retrieve the values for each of them.
  - *score* : have the 0.1% of missing (175 out of 186073) so we fill it with the value **Unknown** because it is impossible to retrieve it consistently.

Excluding this two columns we have 18 columns that regards the statistic of each player, 9 for winner and 9 for loser, that because we have a double foreign key *loser_ id* and *winner_ id* that referees to the *player_ id* key in the column. For each of them there are 55.8% of values, so we decide not to delete it but to replace the corresponding missing value with the value **-1**.
And at the end we have the tables:

Date				
	date_id	month	quarter	year
Dtype	int	int	int	int
				NValue 375

Figure 1: Date.csv

Geography				
	country_code	country_name	continent	language
DType	string	string	string	string
				NValue 125

Figure 2: Geography.csv

Tournament								
	tourney_id	tourney_name	surface	draw_size	tourney_level	date_id	tourney_spectators	tourney_revenue
DType	string	string	string	int	string	int	int	float
								NValue 4853

Figure 3: Tournaments.csv

Player						
	player_id	name	hand	country_code	gender	yob
DType	int	string	string	string	string	string
						NValue 10074

Figure 4: Player.csv

Match																	
	match_id	tourney_id	match_num	winner_id	loser_id	score	best_of	round	minutes	w_ace	...	l_ace	...	winner_ rank	winner_ rank_ points	loser_ rank	loser_ rank_ points
DType	int	string	int	int	int	string	int	string	int	int	int	int	int	int	int	int	int
																	NValue 186073

Figure 5: Match.csv

5 Loading Tables

We read tables ready to be loaded from the various csv files we've built so far. Then we load them by executing a standard *insert* query `load.py 104`, row by row, and executing the commit every 100 rows `load.py 111` and again, once, at the end for any "leftover" rows `load.py 116`. We execute a commit every 100 rows because there's a lot of data in the *Match* table and we want to ensure that even if the connection fails for any reason (i.e. timeout or others) we have only lost at the most the loading of the last 100 rows.
We're loading first the dimensions tables (*Date*, *Geography*, then *Tournament* and *Player*) and the fact table last (*Match*) so as not to have failing constraints on the foreign keys.
We need to remember to insert quotation marks ' where the data type is string (this is not done automatically by Python) and to correctly escape any quotation mark that should be already in the data (like the name **O'Connell**).
`load.py 96`

6 Multidimensional Data Analysis

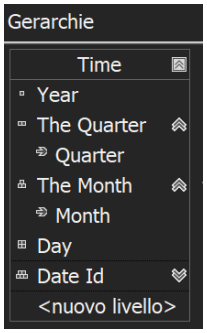
6.1 Creation of the SASS project

For the creation of the Cube part, we first select from Visual Studio our database that was called **Group 4 DB.ds** and on that we create a View.
As a first step we decided to insert some **calculated field** inside the **Date** table:

- *TheQuarter*: because in the table the attribute *quarter* was only a number and we wanted to add the ***Q*** first of the value.
- *TheMonth*: for translation the number of the month into the first three letters of that month, for example, the number ***4*** is ***Apr***.

We also want that these 2 'new' attributes are added in place of *quarter* and *month*.

6.1.1 Creation of the Dimension Tournament



For this first dimension it was selected all the attributes that are present into the Database and we associate the **PK(tourney_id) → FK(tourney_id)** inside the our fact table **Match**. The hierarchy present into this dimension is **Time** ,the attribute *date_id* a FK into the **Tournament** table and a PK into the **Date** table. So we create **Time** with the relation *tourney_id → date_id → year → TheQuarter → TheMonth → day*. Of course like we said before we substitute the two attribute *quarter* and *month*. So we create a functional dependencies between *TheQuarter → quarter* and *TheMonth → month* and we order by **AttributeKey** and the attribute selected was for *TheQuarter, quarter* and for *TheMonth, month*. With this method we have the hierarchy ordered correctly with a more explicit content.

6.1.2 Creation of the Dimension Player

The second dimension that was created is **Player**, in this case we have to create the hierarchy **Geography**, here we have the FK *country_id* related to the PK *country_id* into the table **Geography**. So we built it with the functional dependencies that refers to the table **Geography** and we create the relationship *player_id → country_id → Continent → CountryName → Language*. In the picture you can see how it was created.

6.1.3 Creation of the Cube

For the creation of the cube we first select the Fact table where we want that the measures are, so **Match** is our fact table. The measures requested are:

- Winner Rank
- Winner Rank Points
- Loser Rank
- Loser Rank Points

but we also decided to add:

- Count_match_id, the counting of the single row in the fact table
- Count_winner_id ,the counting of unique values of winner_id

That have been created for completeness and with a view to potential future utility. Not all of them have necessarily been used in subsequent tasks.
So at the end our cube is composed by:

- 3 dimensions **Winner**, **Loser** and **Tournament**, we don't have the **Player** dimension because in the fact table **Match** it was associated 2 FK of the *PK(player_id)* that are *winner_id* and *loser_id*, and for that we have 2 different dimensions in the Cube.
- 6 measures that was explained before.

The name of the Analysis Services database in which it exists and from which you can import what is up to now discussed is *Group 4 DB*.

6.2 Query MDX

6.2.1 Query 1

Show the total winners for each country and the grand total with respect to the continent.

```
select {[Measures].[Count-winner-id]} on columns,
nonempty(([Winner].[Geography].children, [Winner].[Country Name].members)) on rows
from [Group 4 DB];
```

We select the measure that we are interested in, the count of the winner player on columns, and with **nonempty** we select, on rows, first the members of country name regarding the winner dimension and after that the geography hierarchies but with the children function for taking the grand total respect to the Continent.

6.2.2 Query 2

Show the total winner rank points for each year and the running yearly winner rank points for European players.

```
with member eur_p as
[Measures].[Winner Rank Points] + ([Tournament].[Time].prevmember,eur_p )

select {[Measures].[Winner Rank Points], eur_p} on columns,
nonempty(([Tournament].[Time].[Year])) on rows
from [Group 4 DB]
where [Winner].[Continent].&[Europe];
```

First of all we decide to select the winner rank points measures on columns and to take, using the hierarchy Time, the Year attribute, but with the restriction on the Continent Europe, for the European players. With that we have the first part of the Query, so the total Winner rank points, year after year, for European players. The second part of the query, it was resolved with the help of the member *eur_p* where it was calculated the running yearly winner rank points summing the winner rank points of that year with the points from the Time hierarchy year previous member, so the preceding year, and the previous member of *eur_p* for increment the counting of the points. Of curse the first row is equal to the total winner rank points, and the other one are incremented.

6.2.3 Query 3

Show the ratio between the total winner rank points of each year w.r.t the previous year.

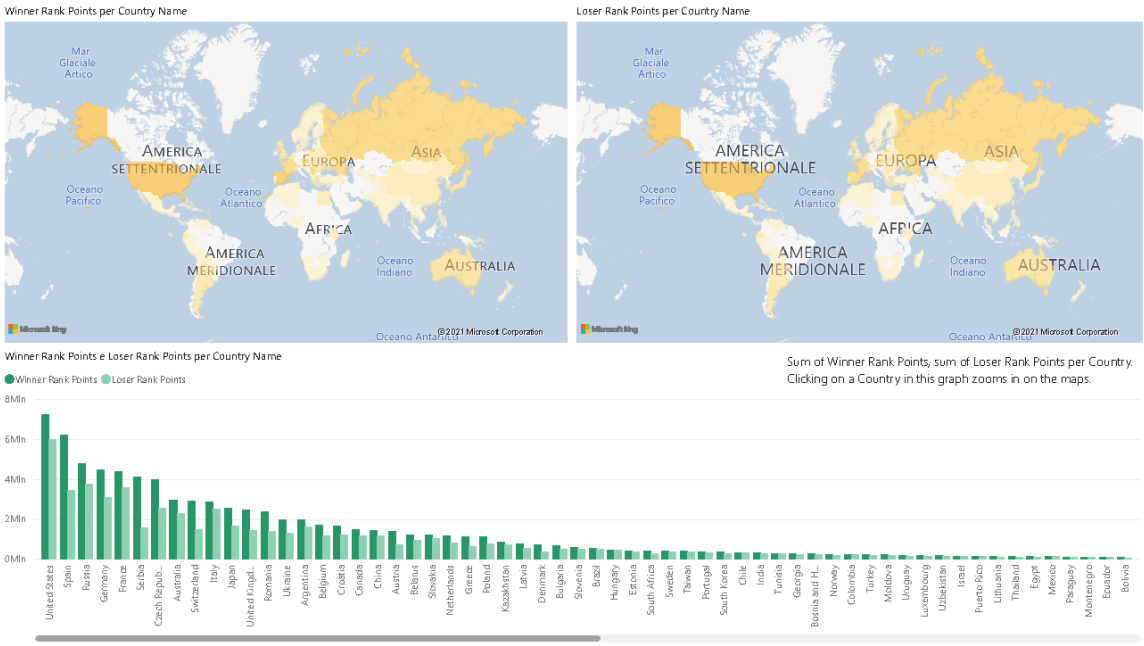
```
with member ratio as
iif(([Tournament].[Time].prevmember, [Measures].[Winner Rank Points]) = 0, null,
[Measures].[Winner Rank Points] / ([Tournament].[Time].prevmember, [Measures].[Winner Rank Points])),
format_string = 'percent'

select {[Measures].[Winner Rank Points], ratio} on columns,
nonempty([Tournament].[Time].[Year]) on rows
from [Group 4 DB];
```

The first part of the query is just selecting the total winner rank points of that year with the hierarchy Time. After that we define the member *ratio*, first of all we define an iff function for setting the *null* value where there isn't the previous year, so for instance, the count of the year starts from 2016, so for that year we have (null). Instead for the calculus of the ratio we divide the total points over the points of the previous year and we assign to it the associated percentage of the ratio.

6.3 Power BI Dashboards

6.3.1 Geographical distribution of Winner Rank Points and Loser Rank Points



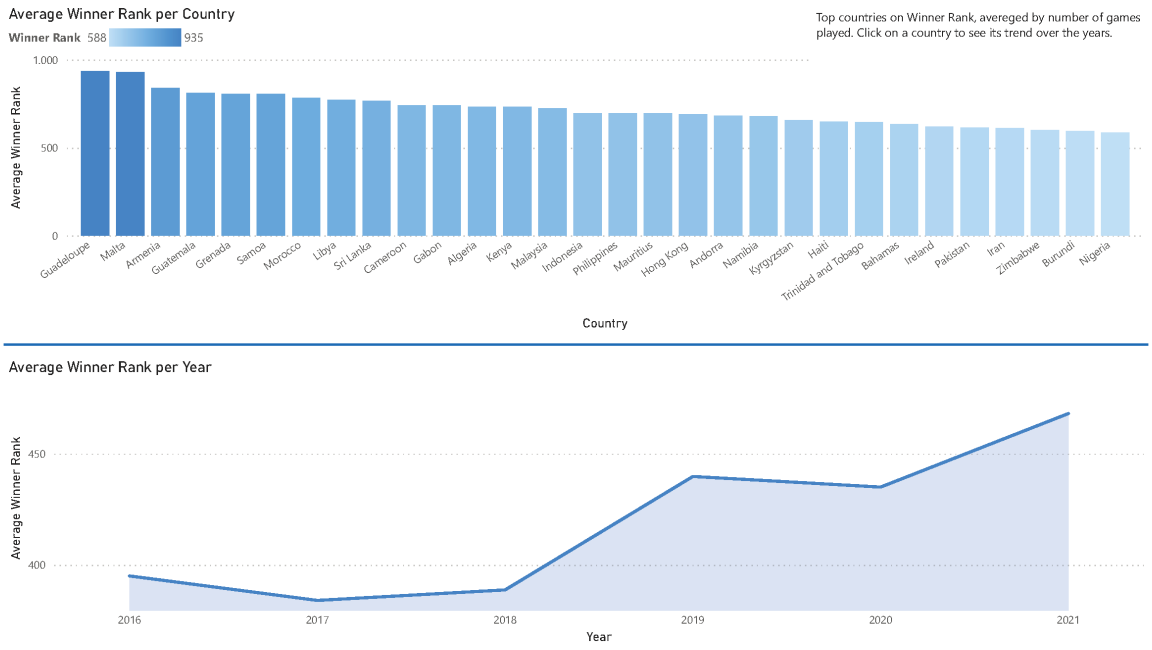
We employed two maps and a graph bar. In the **left map** we show visually the distribution of `SUM(Winner Rank Points)` over each country by coloring the countries in various gradients of yellow/orange according to where the value of the metric for that country falls into its range, which is [41 - 7.218.420].
We do the same for the right map for `SUM(Loser Rank Points)`. In the bar graph, however, we show all countries, ordered by Sum of Winner Rank Points. Clicking on a country in the bar graph focuses on that country on the maps.
It is clear that the most successful countries worldwide according to the **sum** of *Winner Rank Points* are USA, Spain, Russia, Germany, which may be due to a very high number of athletes, correlated with a high population of these countries. We wanted to dig a little deeper into the success in tennis of various countries. We don't necessarily want to associate success with an aggregate sum of the points gained by each athlete. In trying to figure this out is where *"our choice"* dashboard comes in.

6.3.2 Custom dashboard

We'd like to **extract insights on the success of players from each nationality irrespective of the amount of players of that nationality**.
One way - not the only - to accomplish this is to look at *Winner Rank* (the ELO-style ranks for tennis players) and aggregate them with the **average** (summing them would not make sense), but averaging on the **number of winning matches played by all players of each country**. An example would be: if players from Guatemala have played 3 winning matches in the database, with Winner Rank of 600, 550, 700, the value of our measure would be $(600+550+700)/3 = 616.67$.
Here is the Power BI syntax of our calculated field that we show as **Average Winner Rank** measure:

```
IF(Count('Group 4 DB'[Count_winner_id]) > 1,
SUM('Group 4 DB'[Winner Rank])/ Count('Group 4 DB'[Count_winner_id]))
```

We therefore show in our dashboard a bar chart and a line chart. In the bar is shown, ordered, the average *Winner Rank* of players from each of the countries. We only show the top 30 countries for this metric. Hovering the mouse over a country we can read the numeric value of our metric. By doing this, we take out the effect of the amount of players from the information and focus on sport success (by country).



In the line graph we also add the temporal dimension. We show an Average Winner Rank per Year, for the whole dataset. However, if clicking on a country in the bar chart, we show the temporal trend of the metric over years for that country only.
We can drill up in the bar chart, show continents, then drill down on a specific continent to show its countries.
Something more: with this dashboard we also demonstrate the usage of the connection to the data with an **extract**, instead of the previous assignment, in which we use the **live connection** to Analysis Services.
One last touch is that we include the `IF` statement, because we want to analyse this information with at least two winning matches played. If, on the other hand, there should be only one winning match played of any nationality in a database of 186k rows, we believe there's a moderate chance for it to be an outlier value. This condition eliminates the countries of ***Cambodia*** and ***Namibia*** because they only have 1 match won by an athlete.

6.3.2.1 Intended audience and main insights gained

As we can see from the dashboard, the most successful countries are much smaller. Among them: ***Guadeloupe***, ***Malta***, ***Armenia***.
We can think as possible audience of this visualization:

- **financial stakeholders**: investors, advertisers, public entities;
- **sport stakeholders**: high level tennis coaches and other personnel.

For all of these people, it might be very informative to look at countries that have high success levels but perhaps don't have as much of a tennis presence in terms of players. It might be indicative of a particular local tennis passion, cultural elements, or a few strong champions. If I'm an investor, I might want to invest there, in infrastructure and personnel or commercial partnerships. If I'm that country's government, I might invest in local tennis trying to encourage those good results. If I'm a coach looking for high level work, I might look further into those countries and any opportunity they can offer.
For those decision makers that would prefer to look at this online instead of in the Power BI application, there's also a link available to view it in the browser[1].

✍

[1] `<https://app.powerbi.com/groups/c2e57d24-e6be-4f7f-8dc2-7b64e018331a/reports/7c677d98-49cf-4d21-a581-715f83feb65b/ReportSectiondbe617f54b8e5374c63b>`