Objective:

Perform Analysis on the Real Estate Data and Retrieve Meaningful Information from it.

Approach:

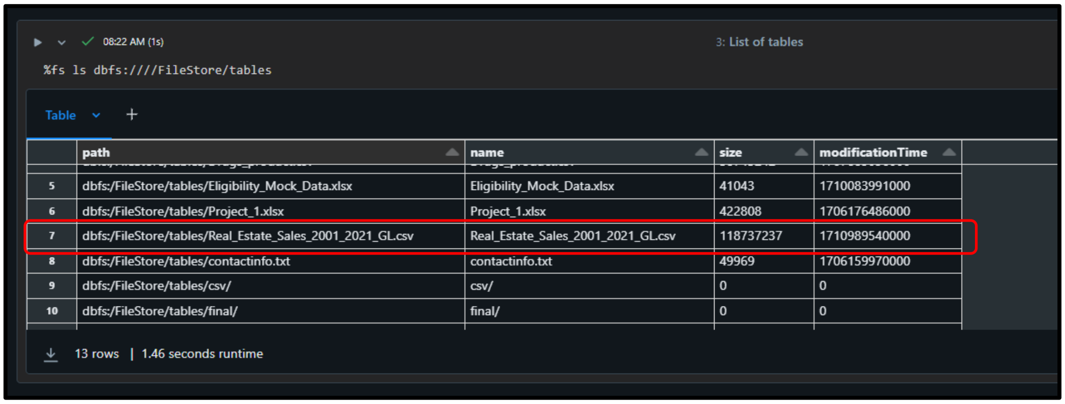
- Import the CSV file into Databricks.
- Analyze the data.
- Extract Information
- Create a Report

Goal:

Extract meaningful data from the records and create an appropriate record.

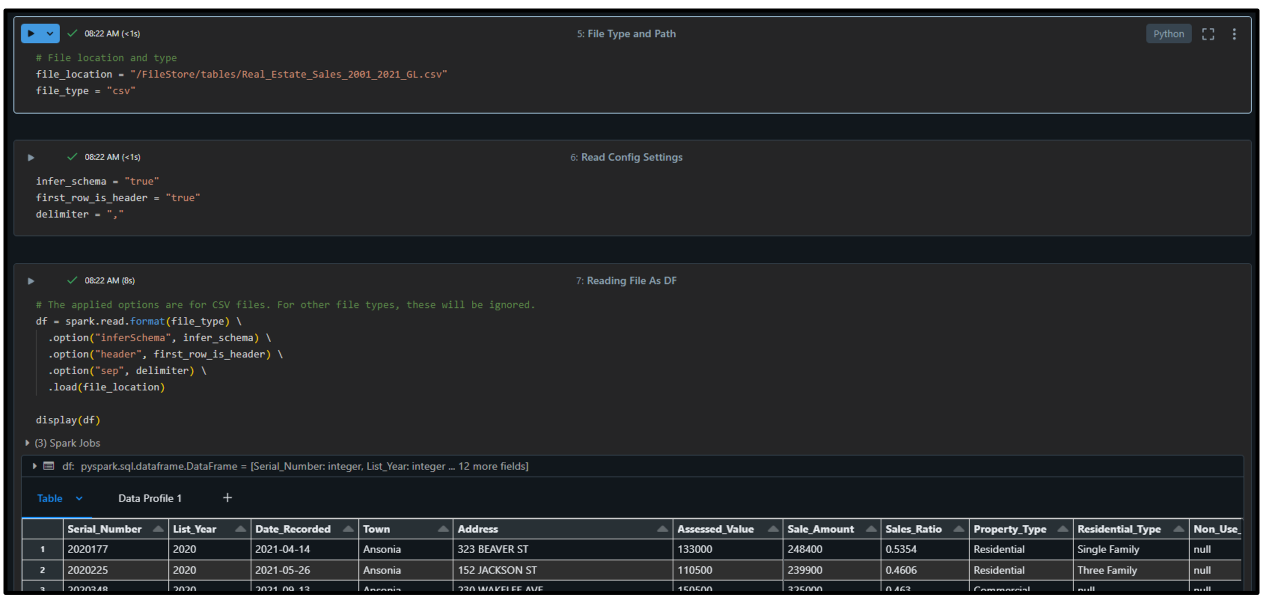
1. Uploading to Databricks

Firstly, the file is uploaded to Databricks to be analyzed. On doing so, we can see the file in the dbfs along with its file path.



2. Reading the CSV

The file is then read using **spark.read** into a dataframe. Required configurations to read the file are also applied.



3. Saving DF to Table

The dataframe is then saved as a table in Databricks.

8: Using Table for DF

08:22 AM (24s) Python

```python
table_name = "RealEstate_CSV"

spark.sql("DROP TABLE IF EXISTS " + table_name)

df.write.saveAsTable(table_name, mode="overwrite")

print(table_name)
```

(6) Spark Jobs

```
RealEstate_CSV
```

9: Selecting All Values From Table

08:22 AM (1s)

```sql
%sql
SELECT *
FROM RealEstate_CSV;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Serial_Number: integer, List_Year: integer ... 12 more fields]

Table

	Serial_Number	List_Year	Date_Recorded	Town	Address	Assessed_Value	Sale_Amount	Sales_Ratio	Property_Type	Residential_Type	Non_Use_
1	2020177	2020	2021-04-14	Ansonia	323 BEAVER ST	133000	248400	0.5354	Residential	Single Family	null
2	2020225	2020	2021-05-26	Ansonia	152 JACKSON ST	110500	239900	0.4606	Residential	Three Family	null
3	2020348	2020	2021-09-13	Ansonia	230 WAKELEE AVE	150500	325000	0.463	Commercial	null	null

4. Analyze Table Content

Firstly, column data types and amount of records are read.

11: Table Description

08:22 AM (<1s)

```sql
%sql
DESCRIBE TABLE RealEstate_CSV;
```

_sqldf: pyspark.sql.dataframe.DataFrame = [col_name: string, data_type: string ... 1 more field]

Table

	col_name	data_type	comment
1	Serial_Number	int	null
2	List_Year	int	null
3	Date_Recorded	date	null
4	Town	string	null
5	Address	string	null
6	Assessed_Value	double	null
7	Sale_Amount	double	null

14 rows | 0.36 seconds runtime Refreshed 5 hours ago

12: Data Count

08:22 AM (<1s)

```python
display(spark.sql("SELECT count(*) FROM RealEstate_CSV"))
```

(2) Spark Jobs

Table

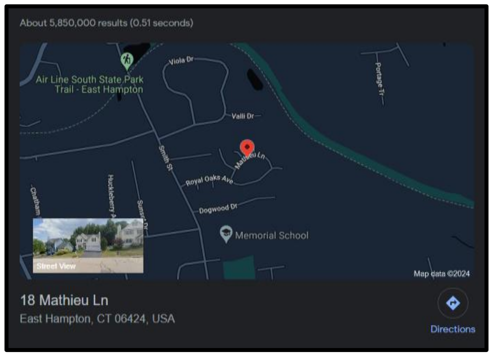
	count(1)
1	1054159

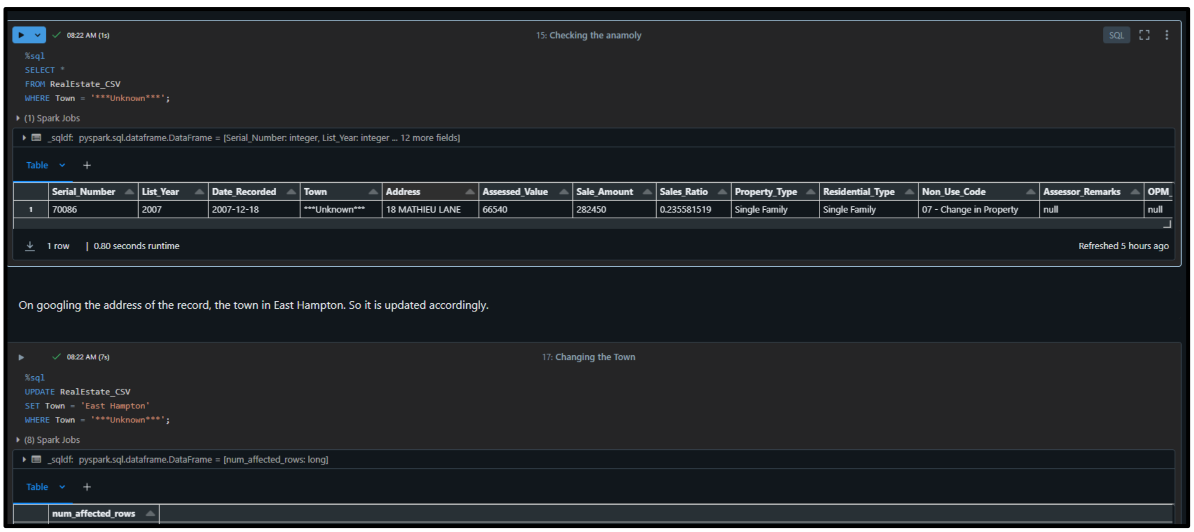
1 row | 0.50 seconds runtime Refreshed 5 hours ago

While checking number of records according to name, an anomaly is detected where the name of the town is ***Unknown***. The data is then further explored.

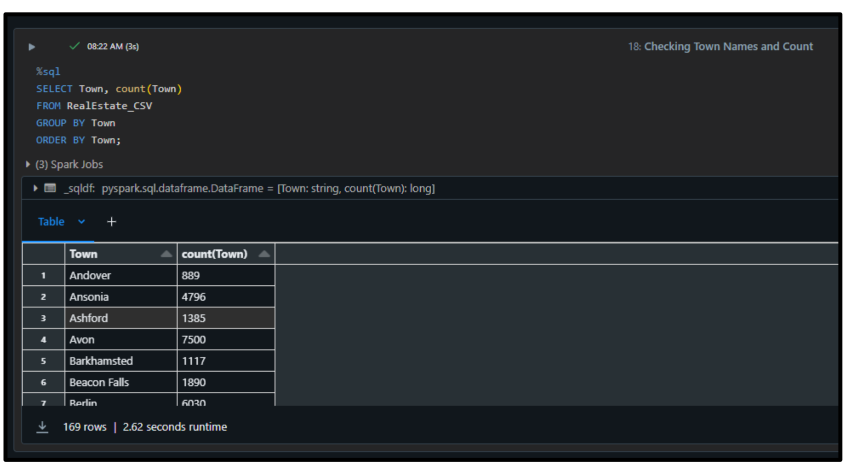
5. Town Name Anomaly

Since it is the only anomaly, the record is further explored. On doing so address can be retrieved. Googling the address resulted in the Town Name: ‘East Hampton’. The record in updated accordingly.



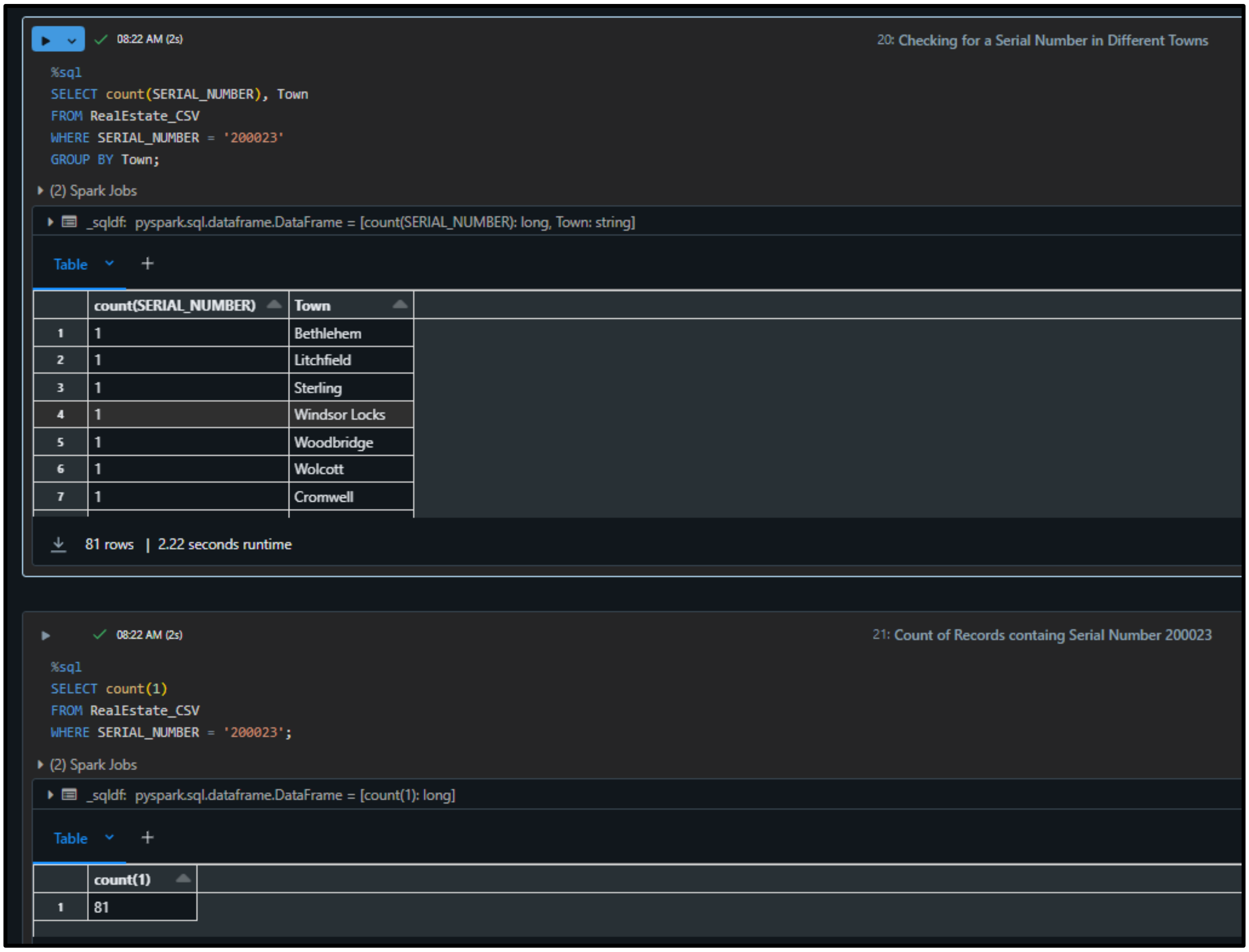


Upon checking the town names, we can now see the absence of the anomaly is the result,



6. Checking for Unique Identifier

On taking a glance of the data, Serial Number can be assumed as the Unique Identifier for records. A check is performed for unique identifier to see for presence of a random Serial Number through different Towns and its occurrence count. We can see it occurring 81 times. We can now assume the Serial Number is not a unique identifier for the records.



08:22 AM (2s) 20: Checking for a Serial Number in Different Towns

```
%sql
SELECT count(SERIAL_NUMBER), Town
FROM RealEstate_CSV
WHERE SERIAL_NUMBER = '200023'
GROUP BY Town;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [count(SERIAL_NUMBER): long, Town: string]

Table

	count(SERIAL_NUMBER)	Town
1	1	Bethlehem
2	1	Litchfield
3	1	Sterling
4	1	Windsor Locks
5	1	Woodbridge
6	1	Wolcott
7	1	Cromwell

81 rows | 2.22 seconds runtime

08:22 AM (2s) 21: Count of Records containg Serial Number 200023

```
%sql
SELECT count(1)
FROM RealEstate_CSV
WHERE SERIAL_NUMBER = '200023';
```

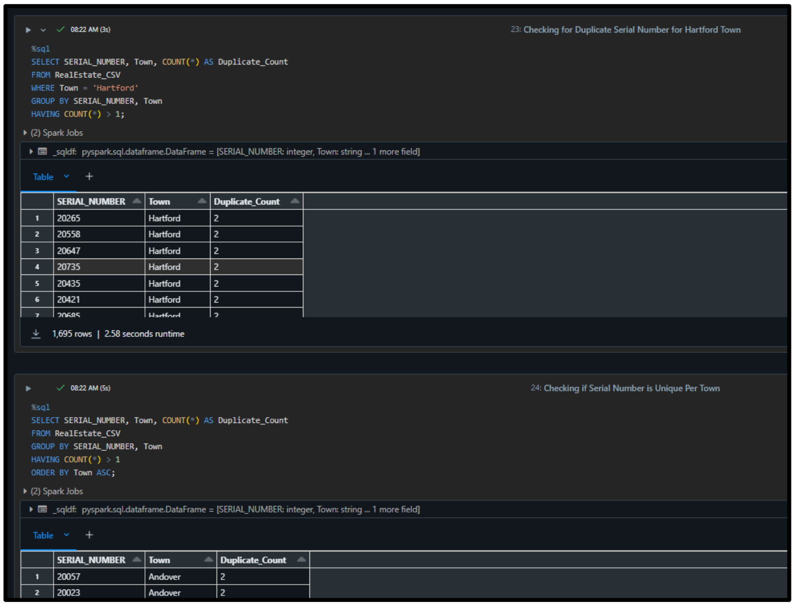
(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [count(1): long]

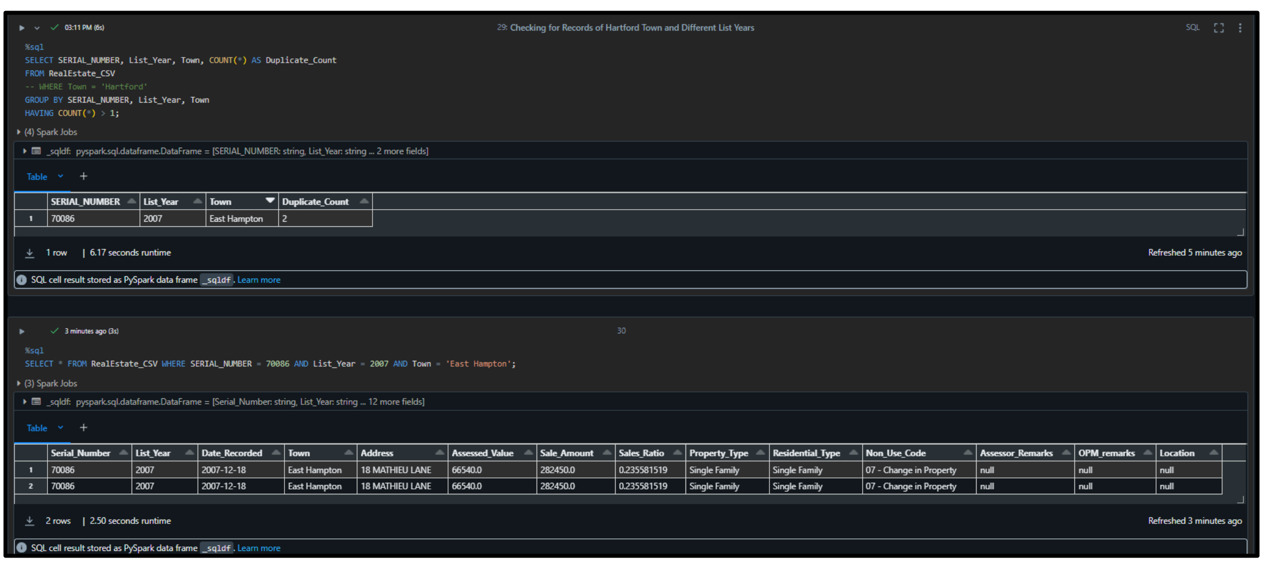
Table

	count(1)
1	81

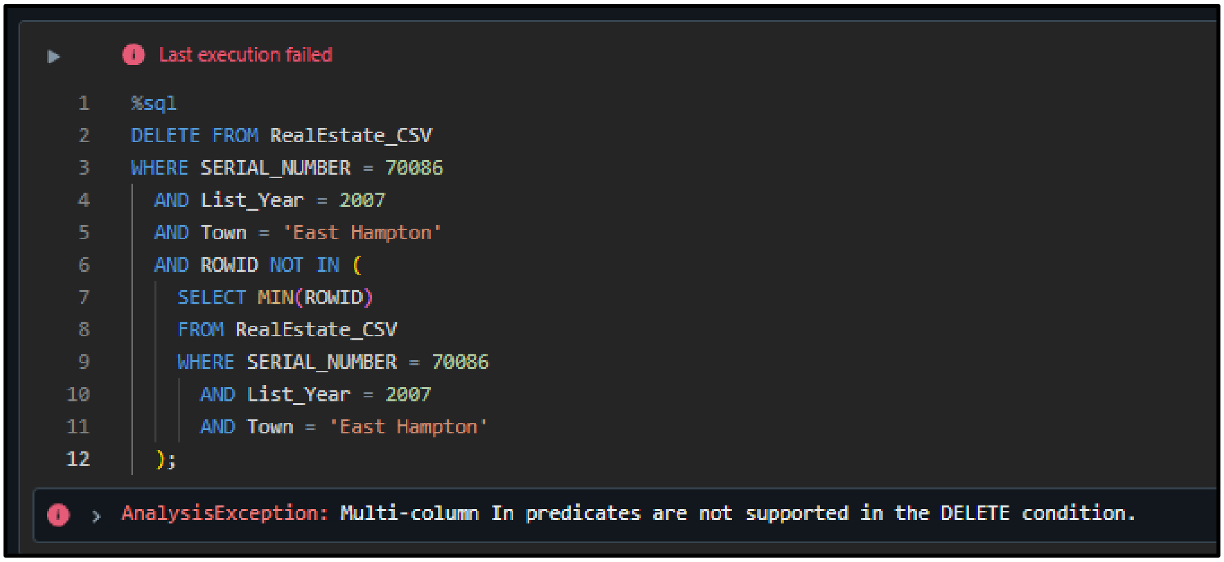
Now, a check is performed to see if Serial Number could be Unique for a particular town. On doing so, it results in multiple counts of the same Serial Number in the same Town. The Serial Number not being unique to a particular Town is also determined. It is also seen through all the towns in the data.

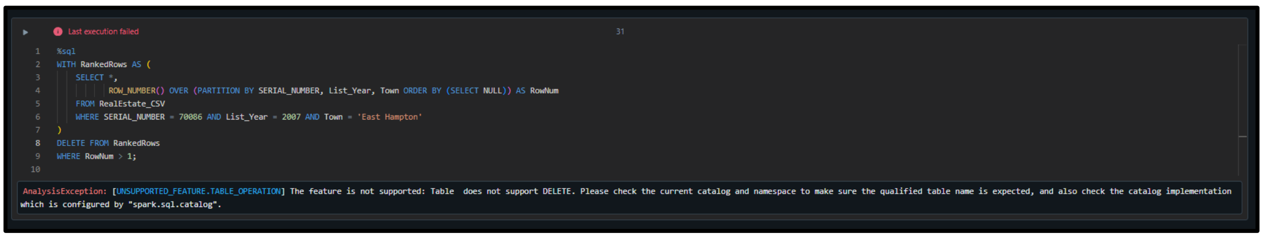


Now, a check is performed to see if Serial Number is Unique to a Particular Town for a Particular List Year. We can see it is unique except for 1 case. Upon further exploring it, the record is the exact duplicate.

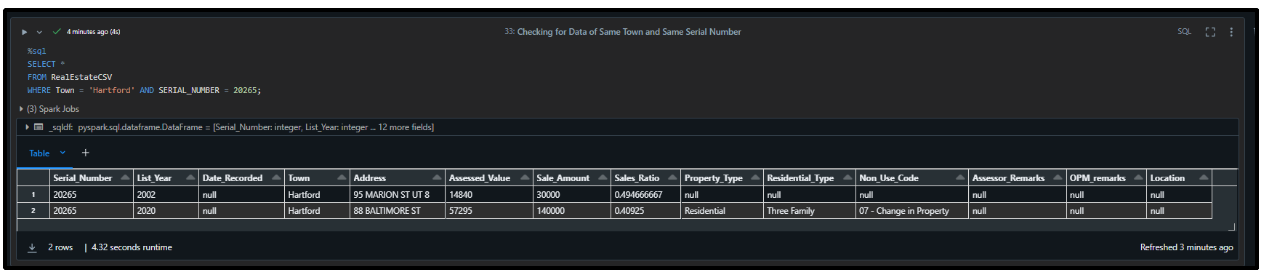


On trying to delete one of the records, it is not supported.



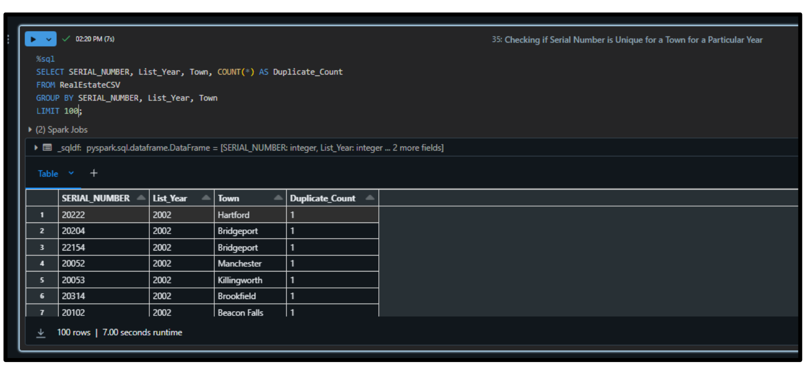


So, for now, it is left as is.



	Serial_Number	List_Year	Date_Recorded	Town	Address	Assessed_Value	Sale_Amount	Sales_Ratio	Property_Type	Residential_Type	Non_Use_Code	Assessor_Remarks	OPM_remarks	Location
1	20265	2002	null	Hartford	95 MARION ST UT 8	14840	30000	0.494666667	null	null	null	null	null	null
2	20265	2020	null	Hartford	88 BALTIMORE ST	57295	140000	0.40925	Residential	Three Family	07 - Change in Property	null	null	null

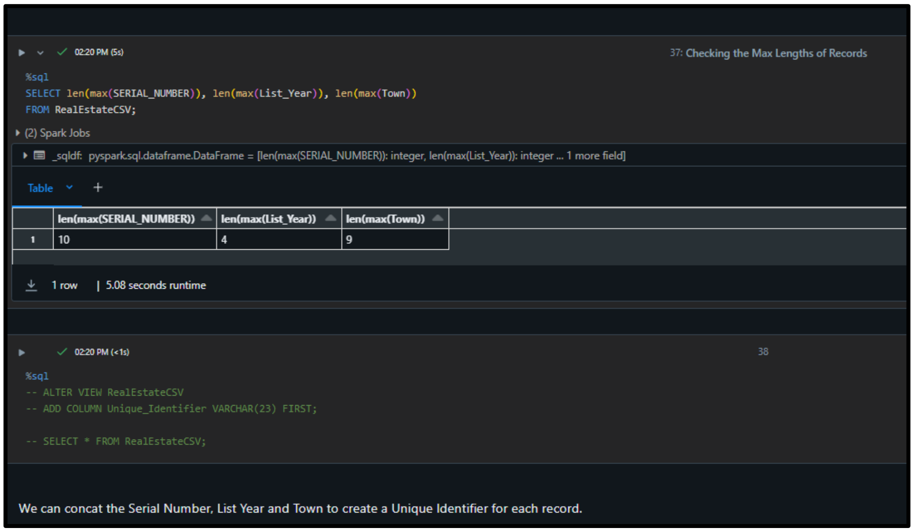
Except for this case, serial number can be a unique identifier for a property for a particular year.



	SERIAL_NUMBER	List_Year	Town	Duplicate_Count
1	20222	2002	Hartford	1
2	20204	2002	Bridgeport	1
3	22154	2002	Bridgeport	1
4	20052	2002	Manchester	1
5	20053	2002	Killingworth	1
6	20314	2002	Brookfield	1
7	20102	2002	Beacon Falls	1

7. Creating Unique Identifier

A unique Identifier for each record can be created by concatenating Serial_Number, List_Year and Town.


02:20 PM (5s) 37: Checking the Max Lengths of Records

```
%sql
SELECT len(max(SERIAL_NUMBER)), len(max(List_Year)), len(max(Town))
FROM RealEstateCSV;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [len(max(SERIAL_NUMBER)): integer, len(max(List_Year)): integer ... 1 more field]

Table

	len(max(SERIAL_NUMBER))	len(max(List_Year))	len(max(Town))
1	10	4	9

1 row | 5.08 seconds runtime

02:20 PM (<1s) 38

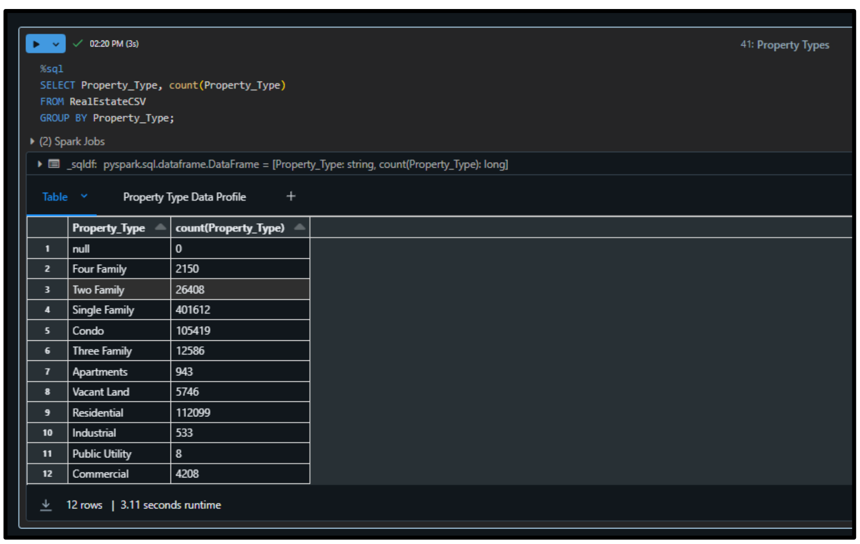
```
%sql
-- ALTER VIEW RealEstateCSV
-- ADD COLUMN Unique_Identifier VARCHAR(23) FIRST;

-- SELECT * FROM RealEstateCSV;
```

We can concat the Serial Number, List Year and Town to create a Unique Identifier for each record.

8. Property Types

On checking the property types, we can see 12 property types including null with count 0. This is because it cannot be counted directly.


02:20 PM (3s) 41: Property Types

```
%sql
SELECT Property_Type, count(Property_Type)
FROM RealEstateCSV
GROUP BY Property_Type;
```

(2) Spark Jobs

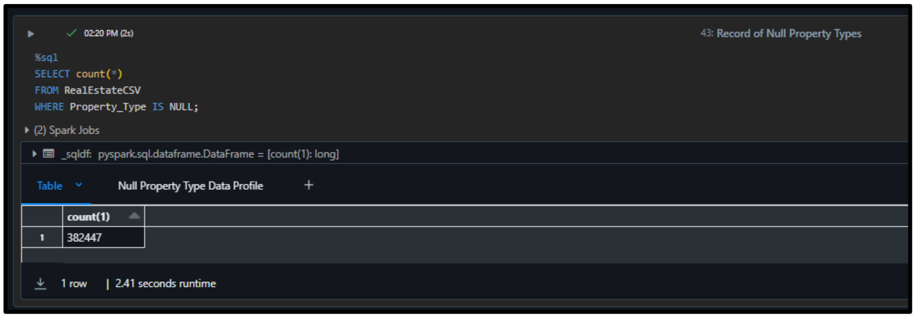
_sqldf: pyspark.sql.dataframe.DataFrame = [Property_Type: string, count(Property_Type): long]

Table Property Type Data Profile

	Property_Type	count(Property_Type)
1	null	0
2	Four Family	2150
3	Two Family	26408
4	Single Family	401612
5	Condo	105419
6	Three Family	12586
7	Apartments	943
8	Vacant Land	5746
9	Residential	112099
10	Industrial	533
11	Public Utility	8
12	Commercial	4208

12 rows | 3.11 seconds runtime

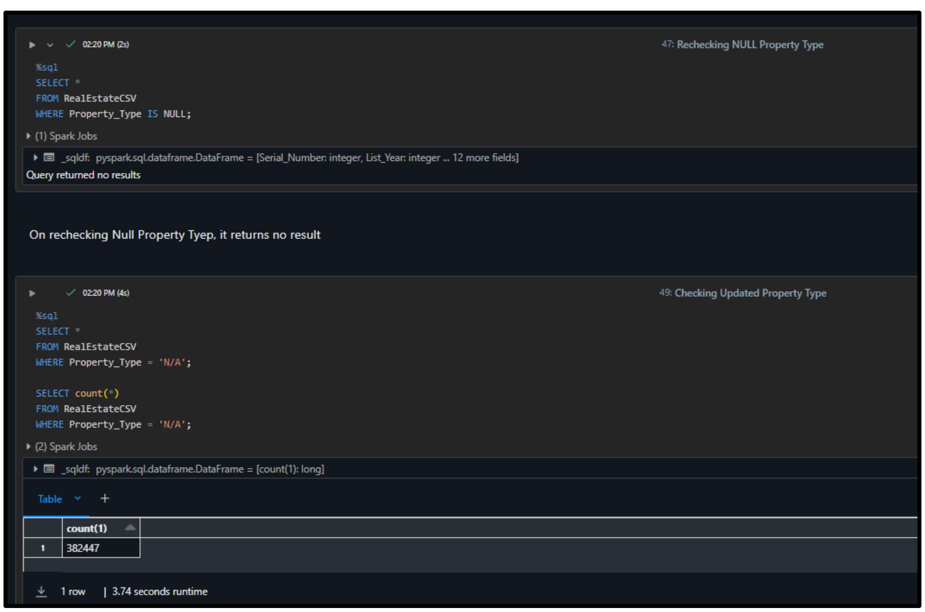
So only null property types are calculated which is more than 10000.



9. Replacing null Property Type

Null Property Types are replaced with string value ‘N/A’.

Null Property Types are not available anymore and data with N/A type is same as previous count.



10. Residential Property Types

On checking data for containing value in the Residential Type column, there was presence of the same data in property type and residential type.

Residential Types in Property Types

02:20 PM (2s) 52: Records Where PropertyType = ResidentialType

```sql
%sql
SELECT *
FROM RealEstateCSV
WHERE Property_Type = Residential_Type
LIMIT 100;
```

(1) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Serial_Number: integer, List_Year: integer ... 12 more fields]

	Serial_Number	List_Year	Date_Recorded	Town	Address	Assessed_Value	Sale_Amount	Sales_Ratio	Property_Type	Residential_Type	Non_Use_Code	Assessor_Remarks	OPM_remarks	Location
1	60108	2006	null	Greenwich	46 STANWICH RD	5130230	6000000	0.855038333	Single Family	Single Family	null	null	null	null
2	61113	2006	null	Fairfield	140 TUNXIS HL CUTOFF	298060	493500	0.603971631	Condo	Condo	null	null	null	null
3	60646	2006	null	Fairfield	68 SOUTHPORT WOODS DR	272720	372000	0.73311828	Condo	Condo	null	null	null	null
4	60274	2006	null	Clinton	123 LIBERTY ST	155400	211000	0.736492891	Condo	Condo	null	null	null	null
5	60346	2006	null	New Britain	224 BOOTH ST	97580	260000	0.375307692	Three Family	Three Family	null	null	null	null
6	60031	2006	null	Tolland	49 WEIGEL VLY DR UT 43	109900	314594	0.349339148	Condo	Condo	07 - Change in Property	null	null	null
7	60012	2006	null	Ridgefield	497 DANBURY RD SEGMENT X	340970	6000	56.82833333	Single Family	Single Family	06 - Portion of Property	null	null	POINT (-7

100 rows | 1.72 seconds runtime Refreshed 19 minutes ago

02:20 PM (3s) 54: PropertyType = ResidentialType Count

```sql
%sql
-- SELECT Property_Type, count(Property_Type)
-- FROM RealEstateCSV
-- WHERE Residential_Type IS NOT NULL
-- GROUP BY Property_Type;

SELECT Property_Type, count(Property_Type)
FROM RealEstateCSV
WHERE Property_Type = Residential_Type
GROUP BY Property_Type;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Property_Type: string, count(Property_Type): long]

	Property_Type	count(Property_Type)
1	Four Family	2150
2	Two Family	26408
3	Single Family	401612
4	Condo	105419
5	Three Family	12586

5 rows | 3.01 seconds runtime

Since, those contained records in the Residential Type column, we can assume that Property Type can be changed to Residential.

02:20 PM (22s) 56: Updating to Residential Property Type

```sql
%sql
UPDATE RealEstateCSV
SET Property_Type = 'Residential'
WHERE Property_Type = Residential_Type;
```

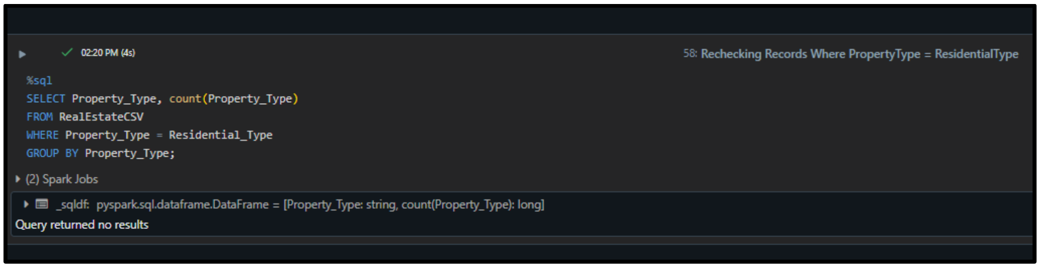
(8) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [num_affected_rows: long]

	num_affected_rows
1	548175

1 row | 22.27 seconds runtime

And, on re-checking the conditions, we can see that the property type is categorized properly.



02:20 PM (4s) 58: Rechecking Records Where PropertyType = ResidentialType

```
%sql
SELECT Property_Type, count(Property_Type)
FROM RealEstateCSV
WHERE Property_Type = Residential_Type
GROUP BY Property_Type;
```

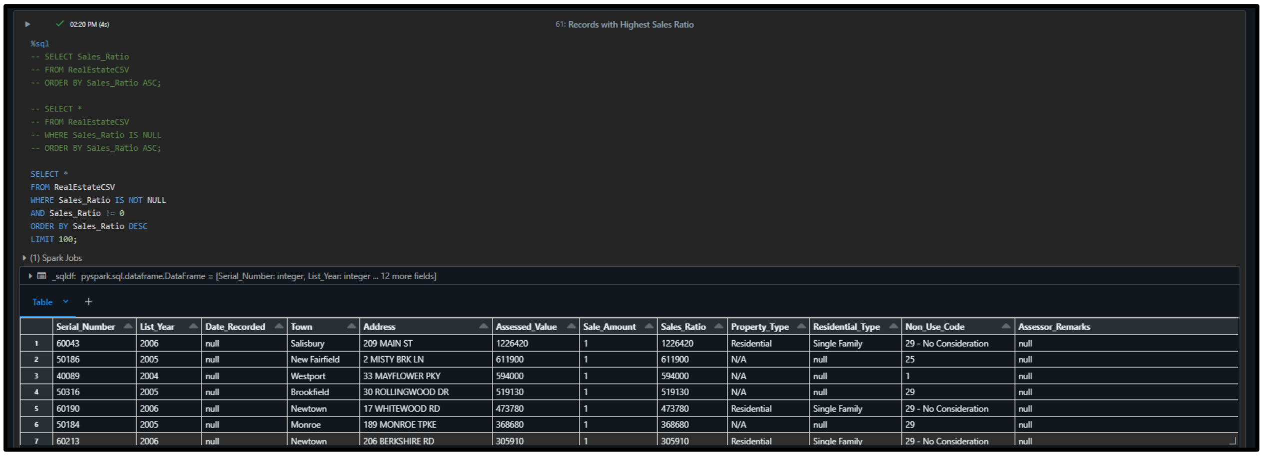
(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Property_Type: string, count(Property_Type): long]

Query returned no results

11. Average Sales Ratio

To get data for average sales ratio, first, we need to get data for the properties that have been sold. Then, the condition for average sales ratio across all towns can be calculated.



02:20 PM (4s) 61: Records with Highest Sales Ratio

```
%sql
-- SELECT Sales_Ratio
-- FROM RealEstateCSV
-- ORDER BY Sales_Ratio ASC;

-- SELECT *
-- FROM RealEstateCSV
-- WHERE Sales_Ratio IS NULL
-- ORDER BY Sales_Ratio ASC;

SELECT *
FROM RealEstateCSV
WHERE Sales_Ratio IS NOT NULL
AND Sales_Ratio != 0
ORDER BY Sales_Ratio DESC
LIMIT 100;
```

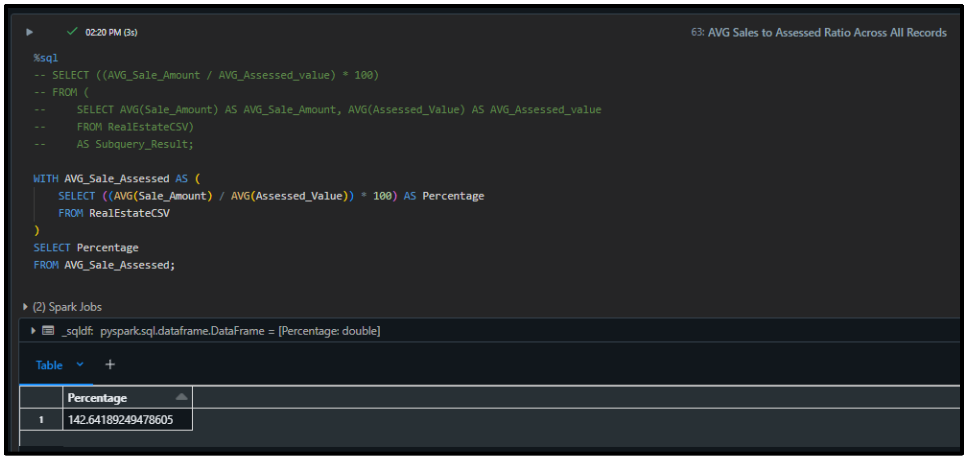
(1) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Serial_Number: integer, List_Year: integer ... 12 more fields]

Table

	Serial_Number	List_Year	Date_Recorded	Town	Address	Assessed_Value	Sale_Amount	Sales_Ratio	Property_Type	Residential_Type	Non_Use_Code	Assessor_Remarks
1	60043	2006	null	Salisbury	209 MAIN ST	1226420	1	1226420	Residential	Single Family	29 - No Consideration	null
2	50186	2005	null	New Fairfield	2 MISTY BRK LN	611900	1	611900	N/A	null	25	null
3	40089	2004	null	Westport	33 MAYFLOWER PKY	594000	1	594000	N/A	null	1	null
4	50316	2005	null	Brookfield	30 ROLLINGWOOD DR	519130	1	519130	N/A	null	29	null
5	60190	2006	null	Newtown	17 WHITEWOOD RD	473780	1	473780	Residential	Single Family	29 - No Consideration	null
6	50184	2005	null	Monroe	189 MONROE TPKE	368680	1	368680	N/A	null	29	null
7	60213	2006	null	Newtown	206 BERKSHIRE RD	305910	1	305910	Residential	Single Family	29 - No Consideration	null

We can see that the average sales are 42.64% above the average estimates value.



02:20 PM (3s) 63: AVG Sales to Assessed Ratio Across All Records

```
%sql
-- SELECT ((AVG_Sale_Amount / AVG_Assessed_value) * 100)
-- FROM (
--     SELECT AVG(Sale_Amount) AS AVG_Sale_Amount, AVG(Assessed_Value) AS AVG_Assessed_value
--     FROM RealEstateCSV)
--     AS Subquery_Result;

WITH AVG_Sale_Assessed AS (
    SELECT ((AVG(Sale_Amount) / AVG(Assessed_Value)) * 100) AS Percentage
    FROM RealEstateCSV
)
SELECT Percentage
FROM AVG_Sale_Assessed;
```

(2) Spark Jobs

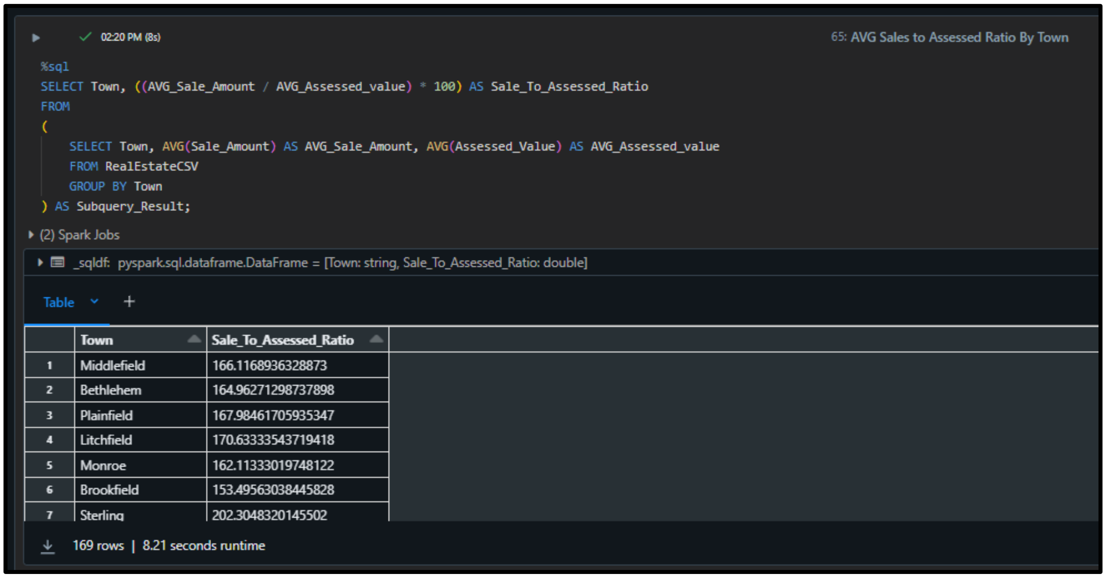
_sqldf: pyspark.sql.dataframe.DataFrame = [Percentage: double]

Table

	Percentage
1	142.64189249478605

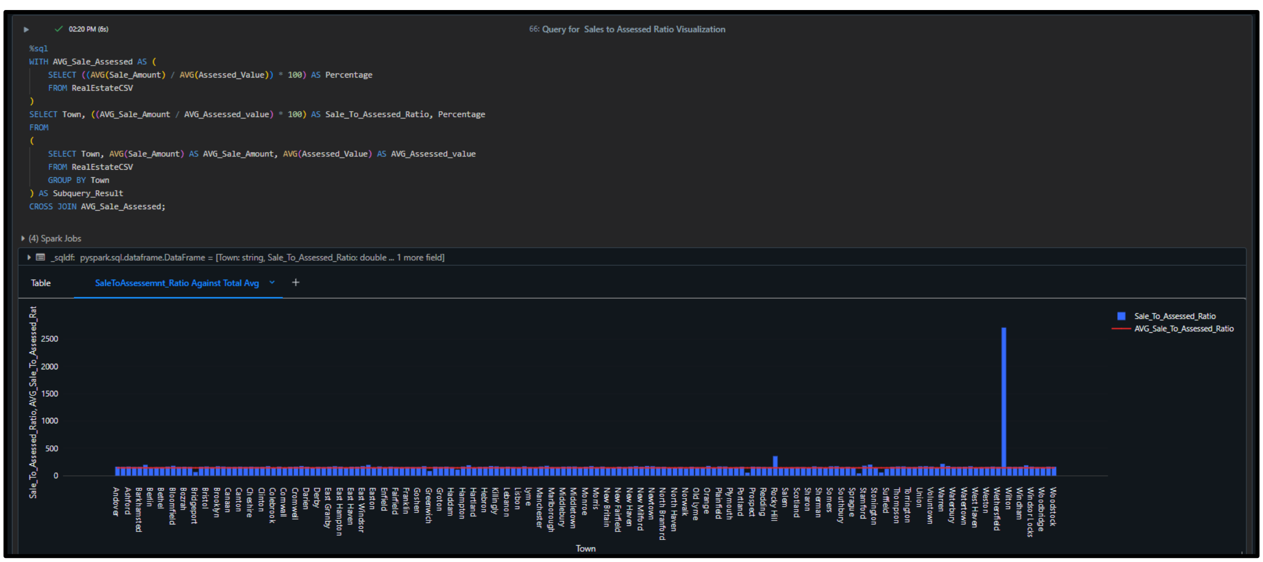
12. Sales to Assessment Ratio by Town

The same condition above can be applied to calculate the ratio by town.



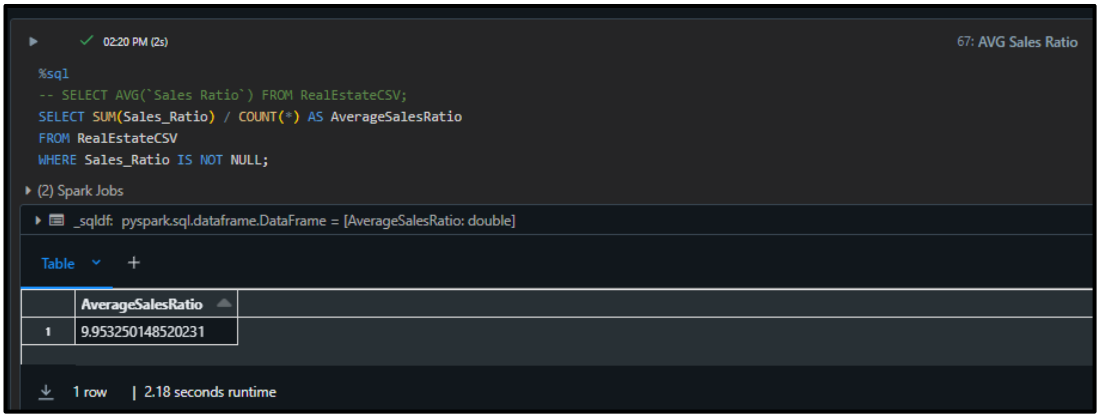
13. Average Sale-To-Assessment Ratio against Total Average

The records can be visualized using appropriate graphs. The graph below illustrates the average sale-to-assessment ratio to the overall sale-to-assessment ratio. This can also be used to indicate Towns with unusual increment in sale amount to assessed value.

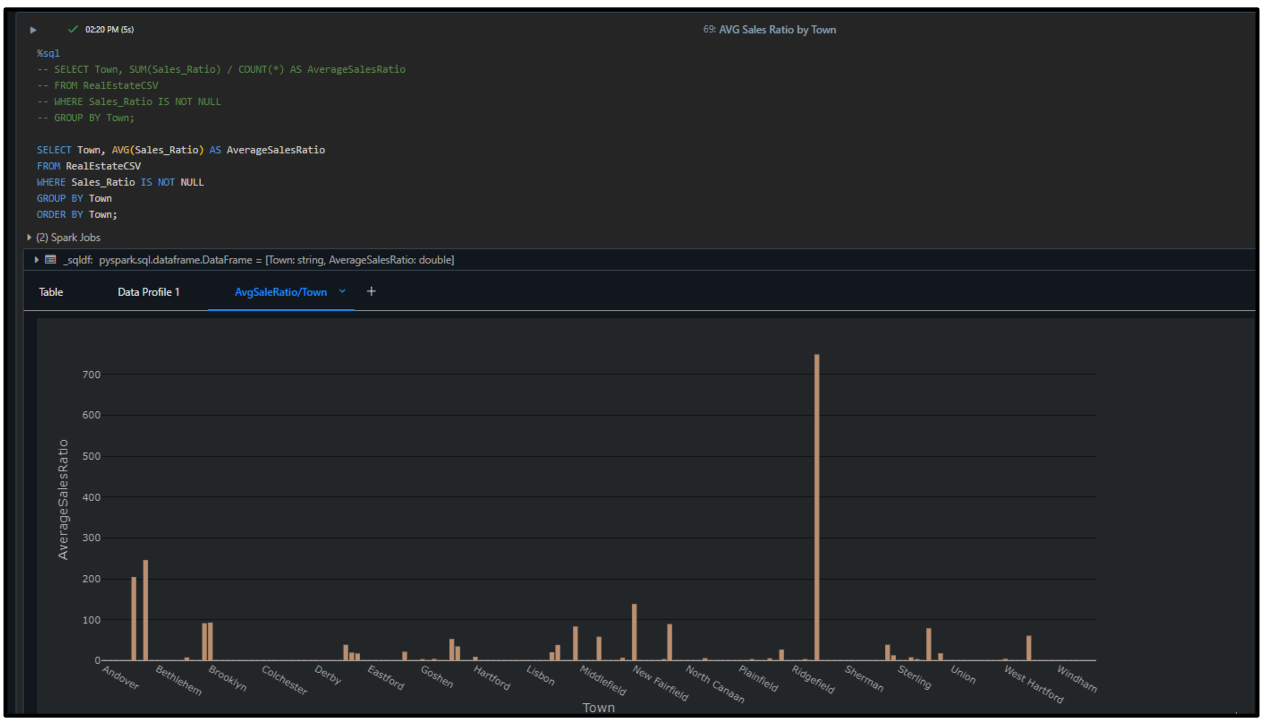


14. Average Sales Ratio

We can also get the average sales ratio.

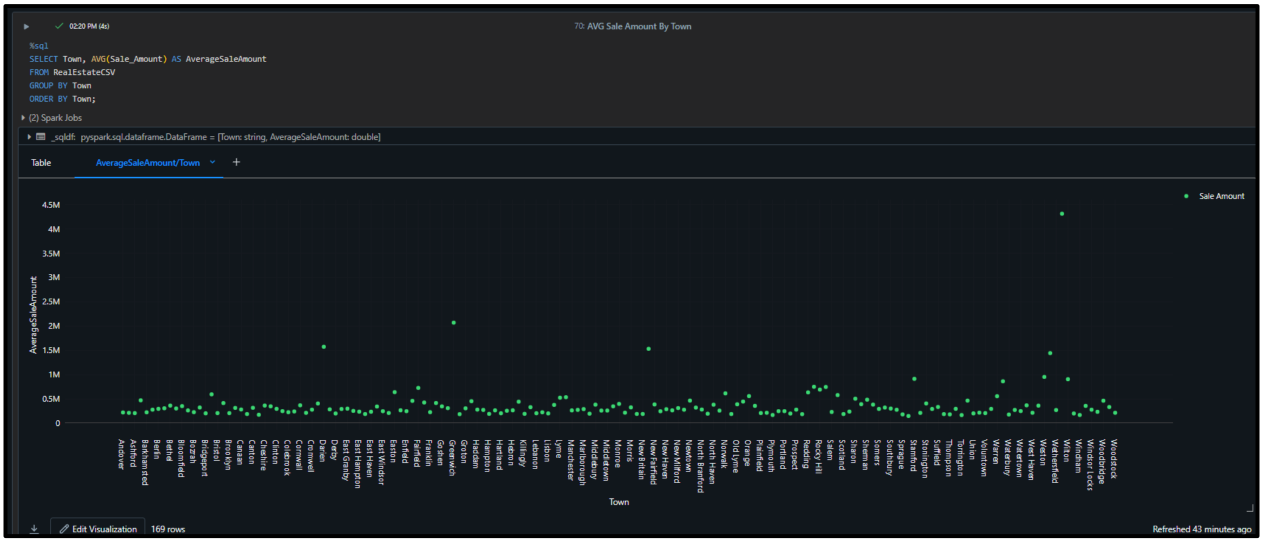


It can also be organized by Towns. It visualizes the Towns with the most rapid increase in the property valuations.



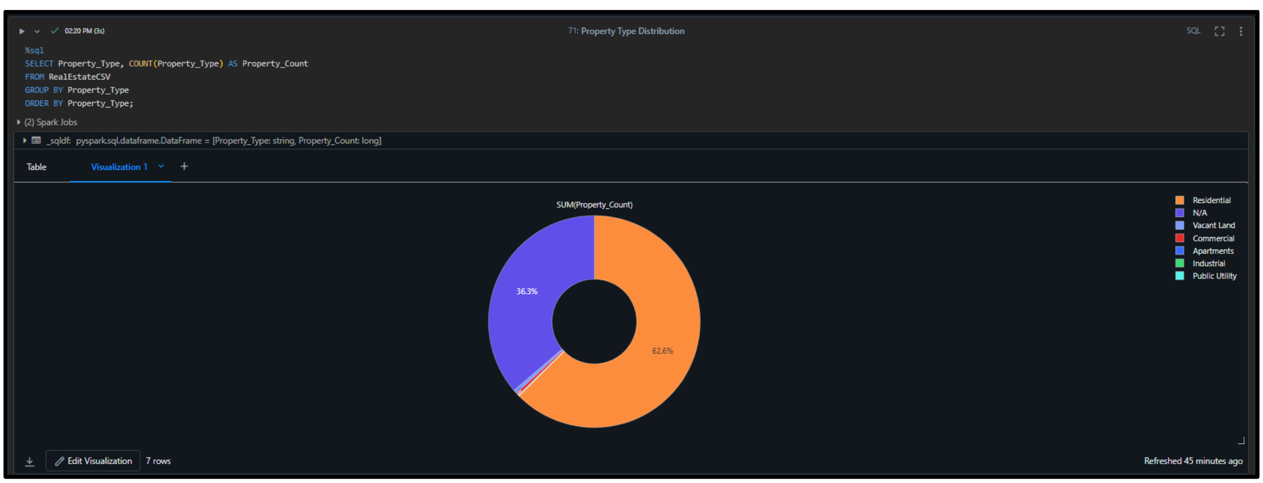
15. Average Sales Amount by Town

The average Sales Amount by Town can also be visualized to see where in average the highest sale amount of any property by town.

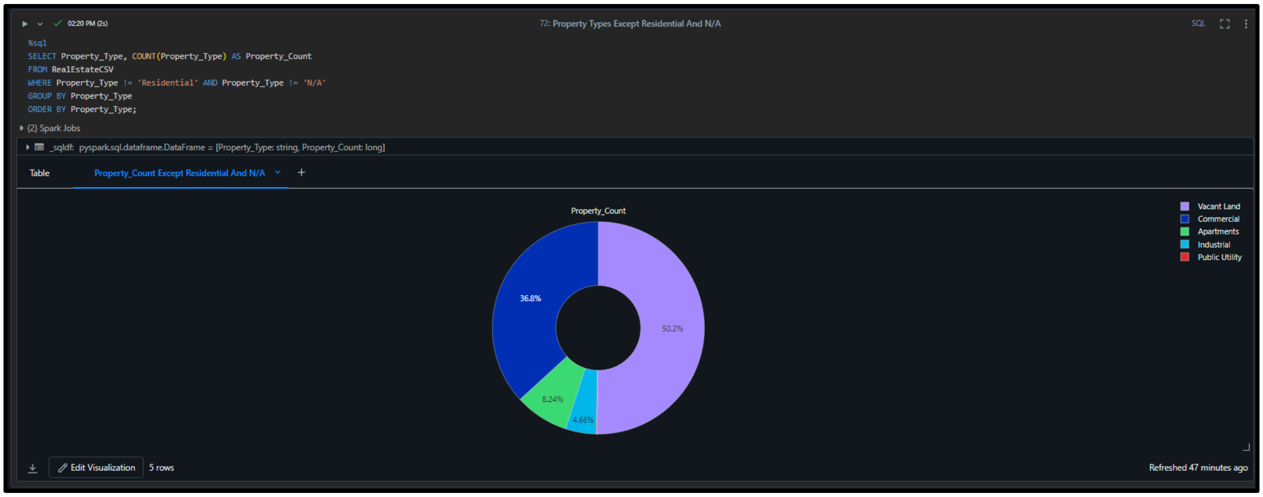


16. Property Type Distribution

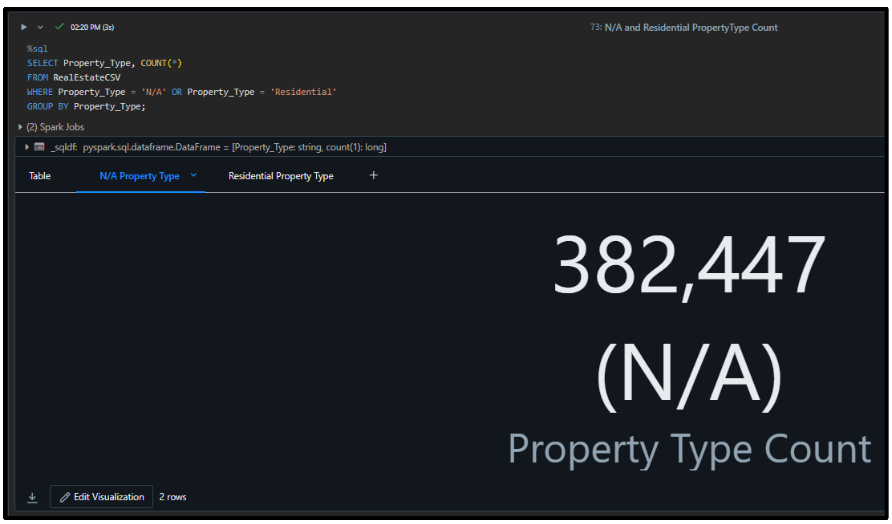
We can visualize the Property Type Distribution in the available data.

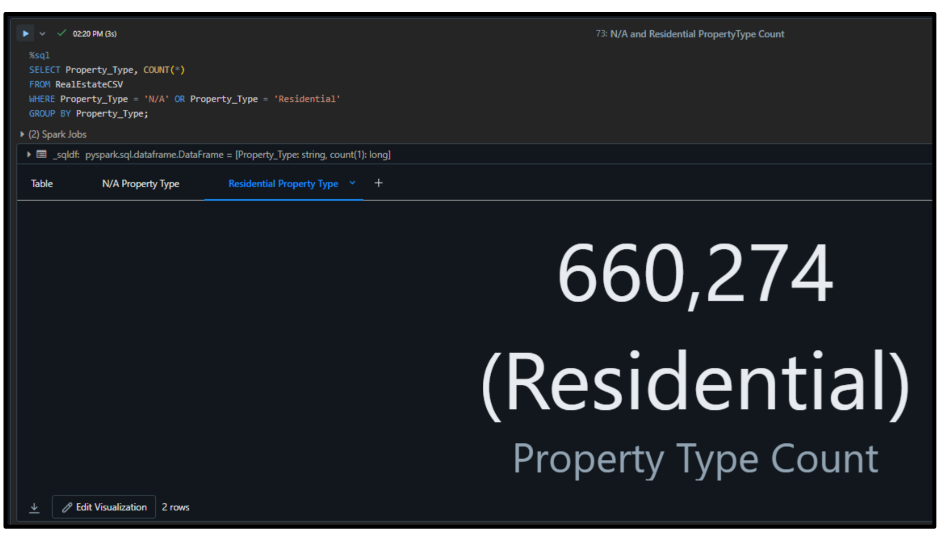


We can see that most of the property type is Residential, and the next largest data is not available. The rest of the properties cannot be visualized properly. So, property types except Residential and N/A are visualized separately.



Also, the count of Residential and N/A Property Type is also visualized.





Also, property types except Residential and N/A are counted by Town to be visualized.

02:20 PM (3s) 74: Propert Type Distibution By Town Except Residential and N/A

```sql
%sql
SELECT Town, Property_Type, COUNT(Property_Type) AS Property_Count
FROM RealEstateCSV
WHERE Property_Type != 'Residential' AND Property_Type != 'N/A'
GROUP BY Town, Property_Type
ORDER BY Town, Property_Type;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Town: string, Property_Type: string ... 1 more field]

Table | Property Distribution by Town Except N/A and Residential

Town	Property_Type	Property_Count
Andover	Vacant Land	7
Ansonia	Apartments	3
Ansonia	Commercial	16
Ansonia	Industrial	7
Ansonia	Vacant Land	22
Ashford	Apartments	4
Ashford	Commercial	4

Edit Visualization 509 rows

The same is also done with Residential and N/A property types.

02:20 PM (4s) 75: Residential anf N/A Property Type Count

```sql
%sql
SELECT Town, Property_Type, COUNT(Property_Type) AS Property_Count
FROM RealEstateCSV
WHERE Property_Type = 'Residential' OR Property_Type = 'N/A'
GROUP BY Town, Property_Type
ORDER BY Town, Property_Type;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Town: string, Property_Type: string ... 1 more field]

Table | Residential and N/A PropertyType Count

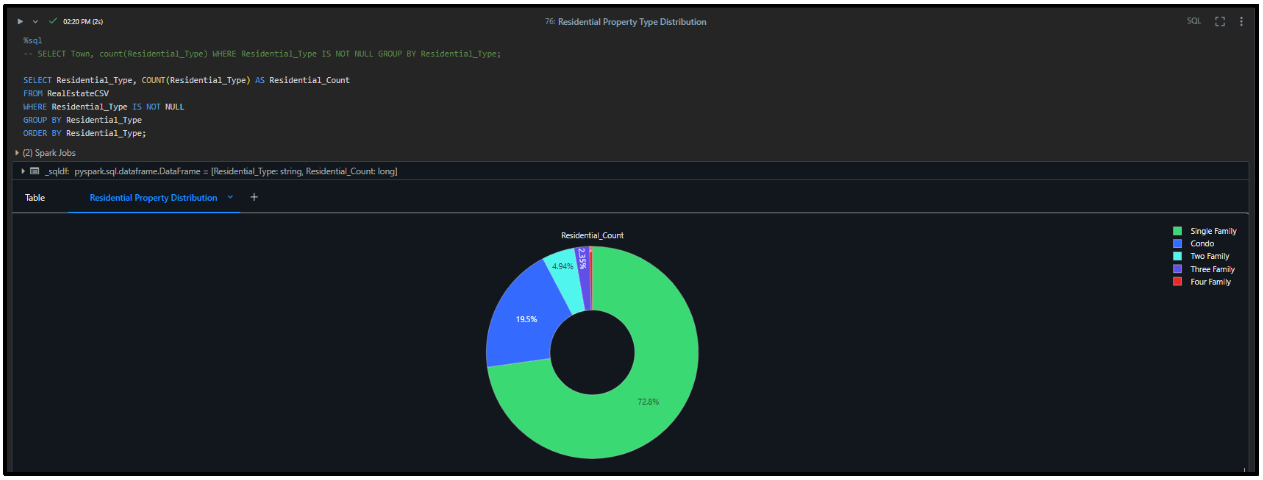
Town	Property_Type	Property_Count
Andover	N/A	366
Andover	Residential	516
Ansonia	N/A	1551
Ansonia	Residential	3197
Ashford	N/A	637
Ashford	Residential	729
Avon	N/A	2533
Avon	Residential	4922
Barkhamsted	N/A	439
Barkhamsted	Residential	627
Beacon Falls	N/A	668
Beacon Falls	Residential	1213

Edit Visualization 338 rows

This is done so that the number of different properties across the towns can be visualized by the count and distribution across the records.

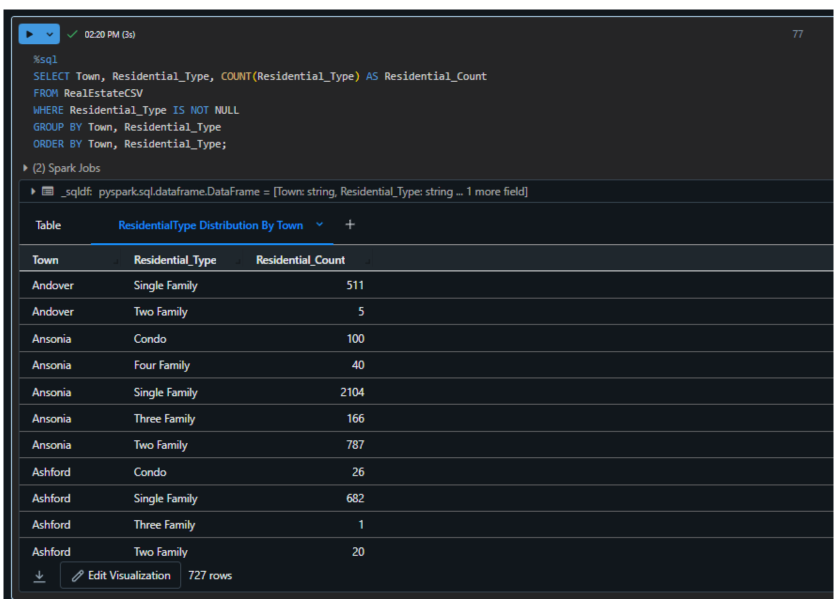
17. Residential Property Type Distribution

Residential Property Type Distribution can be visualized. This shows the types and distribution across the records.



18. Residential Property Type Count

Residential Property Type Count can also be visualized across all the towns.



02:20 PM (3s)

```
%sql
SELECT Town, Residential_Type, COUNT(Residential_Type) AS Residential_Count
FROM RealEstateCSV
WHERE Residential_Type IS NOT NULL
GROUP BY Town, Residential_Type
ORDER BY Town, Residential_Type;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Town: string, Residential_Type: string ... 1 more field]

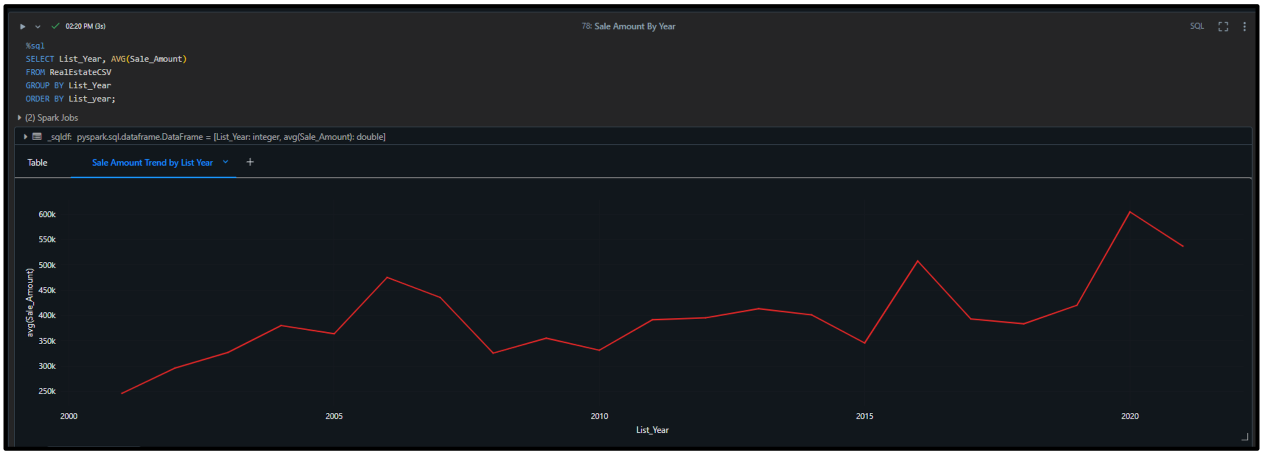
Table | ResidentialType Distribution By Town

Town	Residential_Type	Residential_Count
Andover	Single Family	511
Andover	Two Family	5
Ansonia	Condo	100
Ansonia	Four Family	40
Ansonia	Single Family	2104
Ansonia	Three Family	166
Ansonia	Two Family	787
Ashford	Condo	26
Ashford	Single Family	682
Ashford	Three Family	1
Ashford	Two Family	20

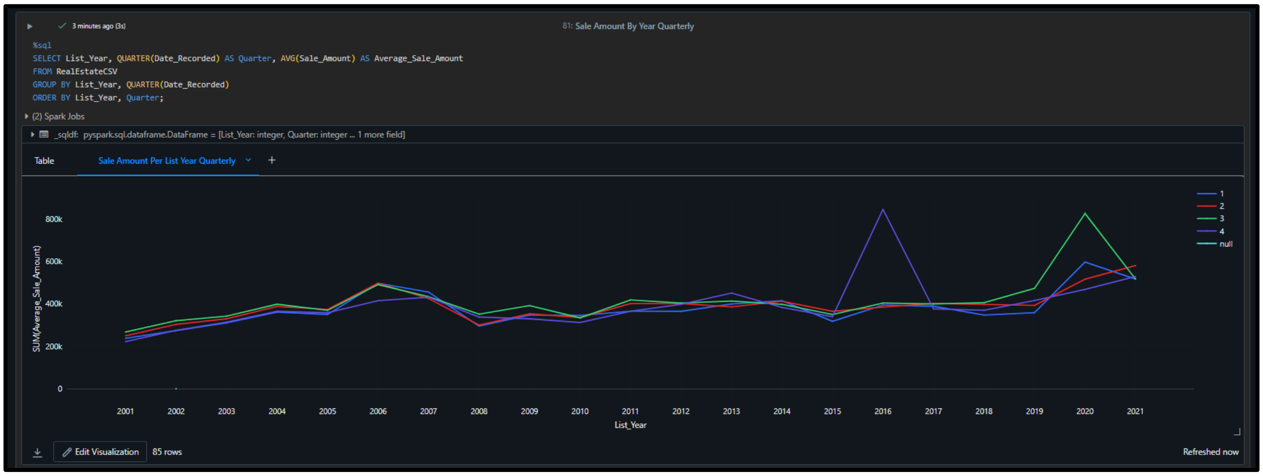
Edit Visualization 727 rows

19. Sale Amount to List Year

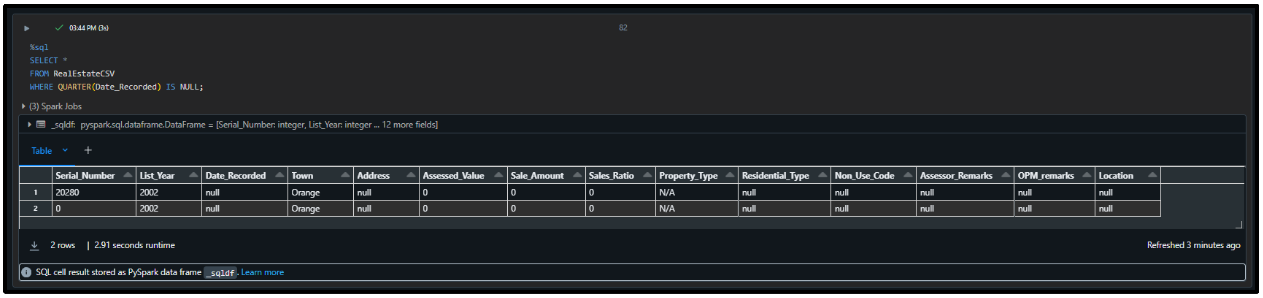
We can also visualize the average sale mount of properties according to different list years. We can see a massive rise in 2006 with a massive fall in 2008. That is when the housing market crashed. Soon after there was a gradual rise to 2014. Then a dip occurred in 2015 and decrease in 2016. Again, a massive spike occurred since then up to 2020 which again fell during the covid years.



We can also see the sale amount distribution across the years quarterly. This can help indicate quarterly changes in sale amount.



We can see the presence of a null record, on investigating, we can see the presence of an incomplete data.


03:44 PM (3s) 82

```
%sql
SELECT *
FROM RealEstateCSV
WHERE QUARTER(Date_Recorded) IS NULL;
```

(3) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Serial_Number: integer, List_Year: integer ... 12 more fields]

Table +

	Serial_Number	List_Year	Date_Recorded	Town	Address	Assessed_Value	Sale_Amount	Sales_Ratio	Property_Type	Residential_Type	Non_Use_Code	Assessor_Remarks	OPM_remarks	Location
1	20280	2002	null	Orange	null	0	0	0	N/A	null	null	null	null	null
2	0	2002	null	Orange	null	0	0	0	N/A	null	null	null	null	null

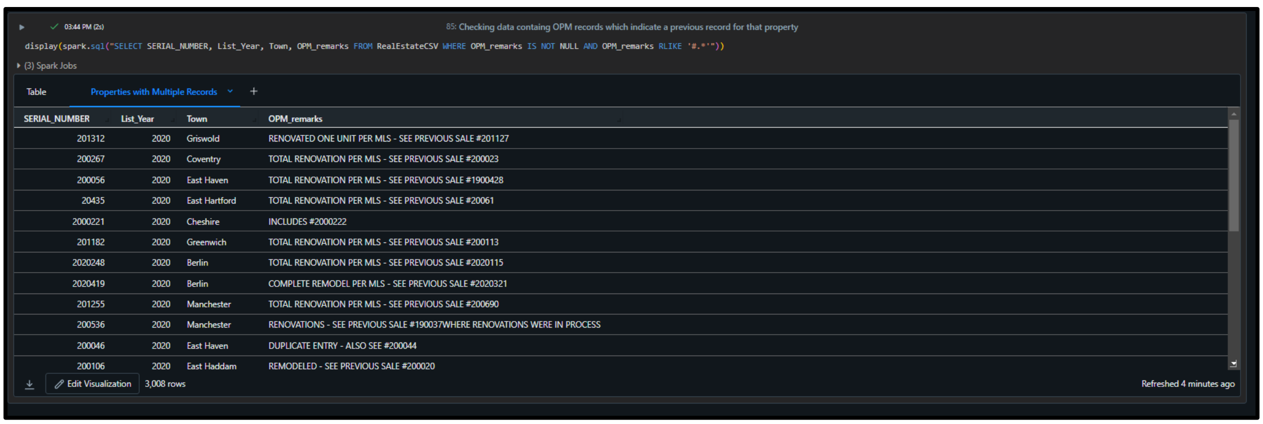
2 rows | 2.91 seconds runtime Refreshed 3 minutes ago

SQL cell result stored as PySpark data frame _sqldf. Learn more

It can be fixed simply by deleting the record.

20. Chain Records

During data exploration, records indicating to a serial number is indicated. This indicated towards chained records. This can be used further for determining things like changes to assessment ratio when changes are made to the property itself. For now, the records are simply visualized on a table.


03:44 PM (2s) 85: Checking data containg OPM records which indicate a previous record for that property

```
display(spark.sql("SELECT SERIAL_NUMBER, List_Year, Town, OPM_remarks FROM RealEstateCSV WHERE OPM_remarks IS NOT NULL AND OPM_remarks RLIKE '#.*'"))
```

(3) Spark Jobs

Table Properties with Multiple Records +

SERIAL_NUMBER	List_Year	Town	OPM_remarks
201312	2020	Griswold	RENOVATED ONE UNIT PER MLS - SEE PREVIOUS SALE #201127
200267	2020	Coventry	TOTAL RENOVATION PER MLS - SEE PREVIOUS SALE #200023
200056	2020	East Haven	TOTAL RENOVATION PER MLS - SEE PREVIOUS SALE #1900428
20435	2020	East Hartford	TOTAL RENOVATION PER MLS - SEE PREVIOUS SALE #20061
2000221	2020	Cheshire	INCLUDES #2000222
201182	2020	Greenwich	TOTAL RENOVATION PER MLS - SEE PREVIOUS SALE #200113
2020248	2020	Berlin	TOTAL RENOVATION PER MLS - SEE PREVIOUS SALE #2020115
2020419	2020	Berlin	COMPLETE REMODEL PER MLS - SEE PREVIOUS SALE #2020321
201255	2020	Manchester	TOTAL RENOVATION PER MLS - SEE PREVIOUS SALE #200690
200536	2020	Manchester	RENOVATIONS - SEE PREVIOUS SALE #190037WHERE RENOVATIONS WERE IN PROCESS
200046	2020	East Haven	DUPLICATE ENTRY - ALSO SEE #200044
200106	2020	East Haddam	REMODELED - SEE PREVIOUS SALE #200020

Edit Visualization 3,008 rows Refreshed 4 minutes ago

21. Dashboards

From the above 3 dashboards can be created.

- AVG Dashboard
  https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/6332182203914549/2743257718242605/1006725448119626/latest.html

- Property Dashboard
  https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/6332182203914549/2743257718242605/1006725448119626/latest.html

- Sale/Year Dashboard
  https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/6332182203914549/2743257718242605/1006725448119626/latest.html

22. Notebook

The notebook is the following:

https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/6332182203914549/2743257718242605/1006725448119626/latest.html