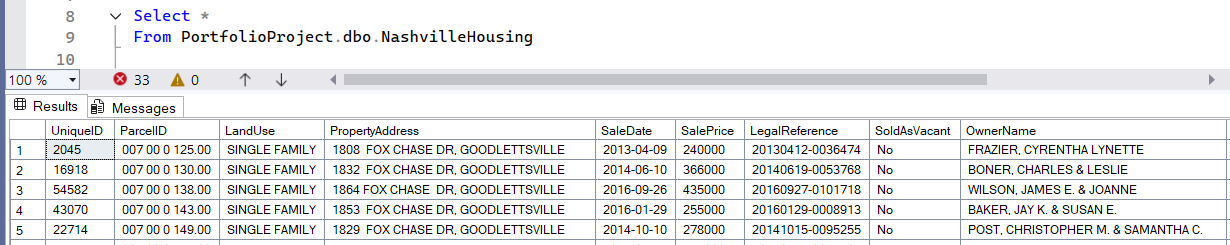
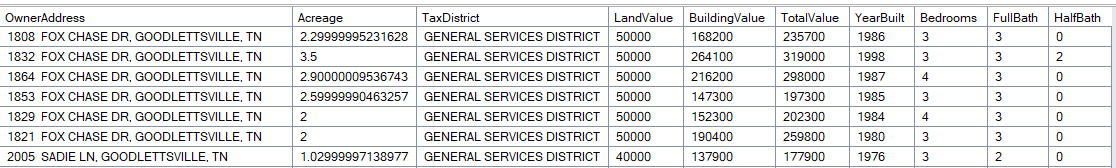
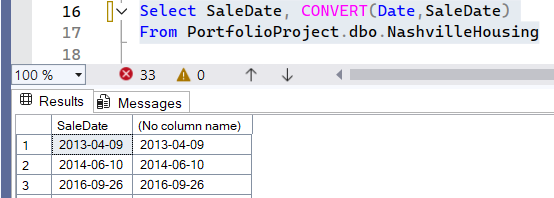
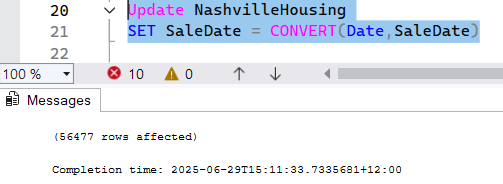
DATA CLEANING



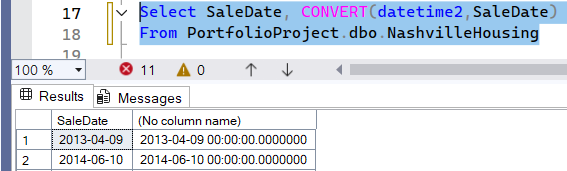
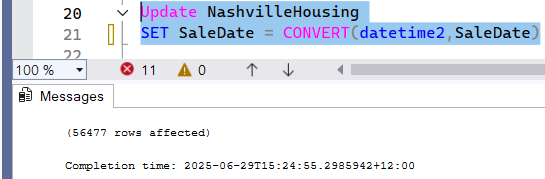


I. Standardizing date format

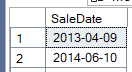
Before I’ve uploaded the .csv file I’ve standardized the format in excel already. It retained the format in SQL as you can see in the converted column. In that case it is safe or good practice to use an SQL query to standardize it. So to implement/apply it on the column “UPDATE” function is needed

For experiment, I’ve tried converting to different data type format

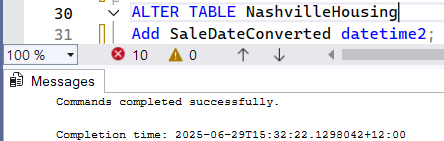
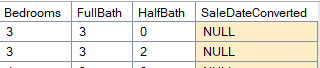
 

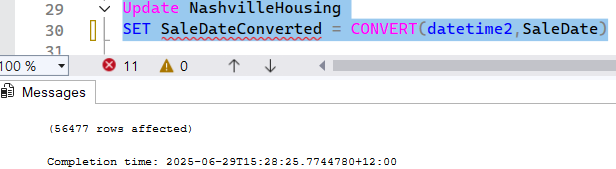
Upon checking, it remain unchanged. WHY? Because the data type of column “SaleDate” is datetime and you are changing the data type to datetime2. UPDATE and SET is used to update the values in the column NOT THE DATA TYPE.

II. Using ALTER TABLE to store the values that has new data type

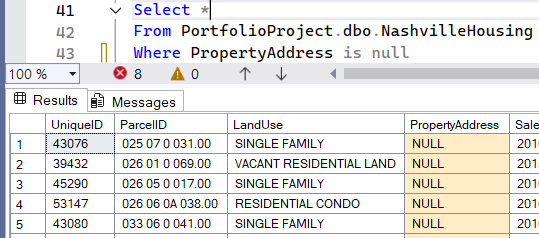
ALTER will add a column with the name and data type format. NOTE: the creation (ALTER) of table column, the data type format must be the same data type with the insertion (UPDATE) of data.

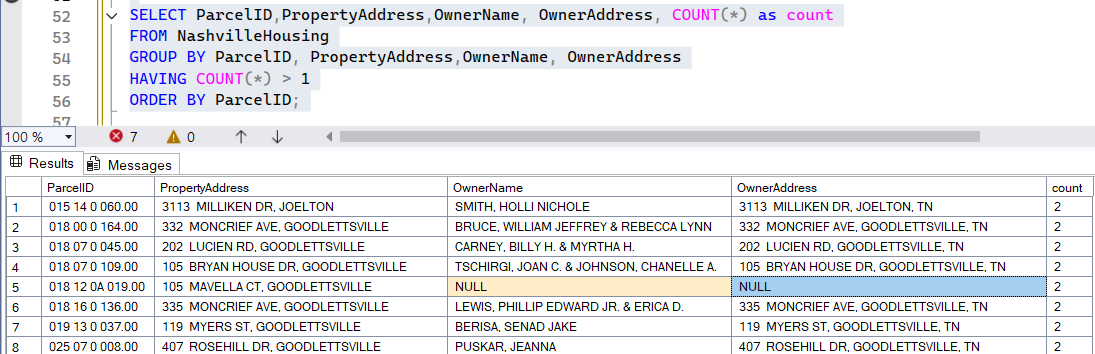
 

III. POPULATING NULL. If there’s a NULL in a certain column, you may want to check if that column has duplicates data. Then check if that duplicate data’s corresponding rows has also duplicates.

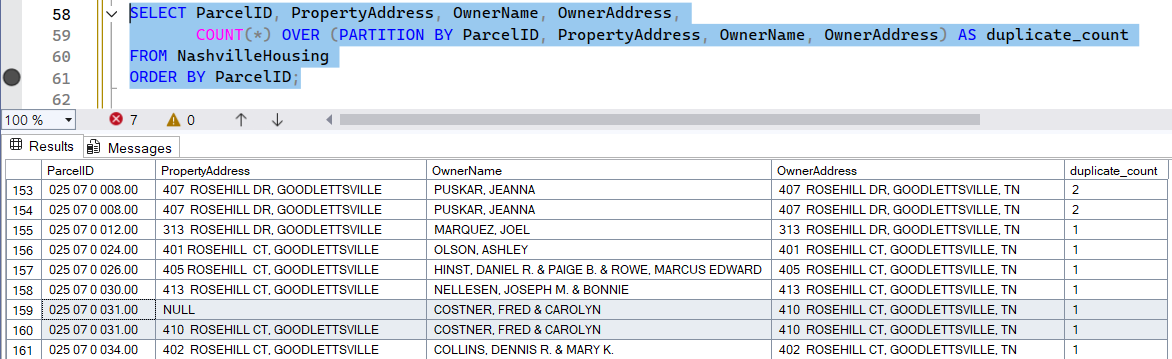
Checking all NULL property address. Does ‘PropertyAddress’ column have duplicate datas? And does that duplicate datas corresponding rows has also duplicate in other column?



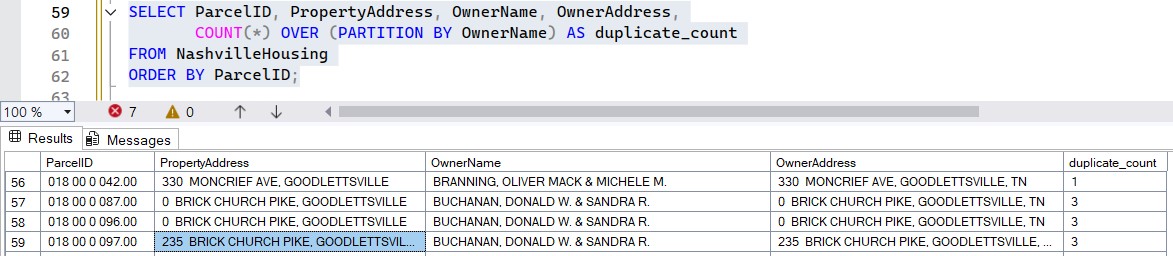
First check all the suspected column that has similarities in data using Group By and Having (can only be used with Group By)



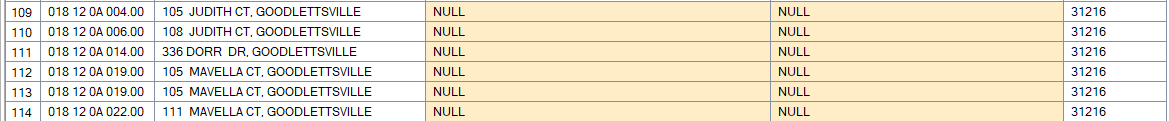
Now that you know there are columns that have similar rows of data, you can use ‘Partition By’ and ‘Over()” to see the ungrouped rows. As you can see on line 159 and 160, the OwnerName, OwnerAddress, and ParcelID are similar but line 159 , the PropertyAddress is missing. Now that you know that similar ParcelID means similar data with the column “PropertyAddres” and 3 more columns, you can easily copy the NULL rows in that specific column



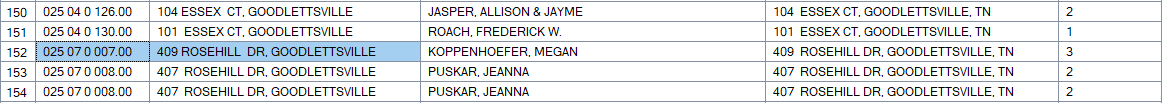
There are some outliers, example if you partition by OwnerName it will partition showing 3 similar names, with different parcelID. The third duplicate has different address.



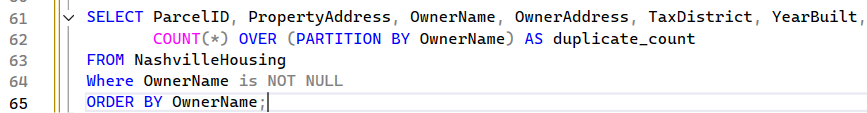
There are those with no names



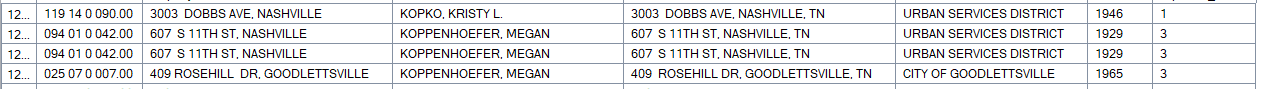
And there are this line 152 where it says duplicate count is 3 yet only showing one row because the query is ORDER BY ParcelID. That OwnerName has 3 different parcelID yet have 3 OwnerName in the database. the other two is somewhere where the ParcelID is arranged in ASC order



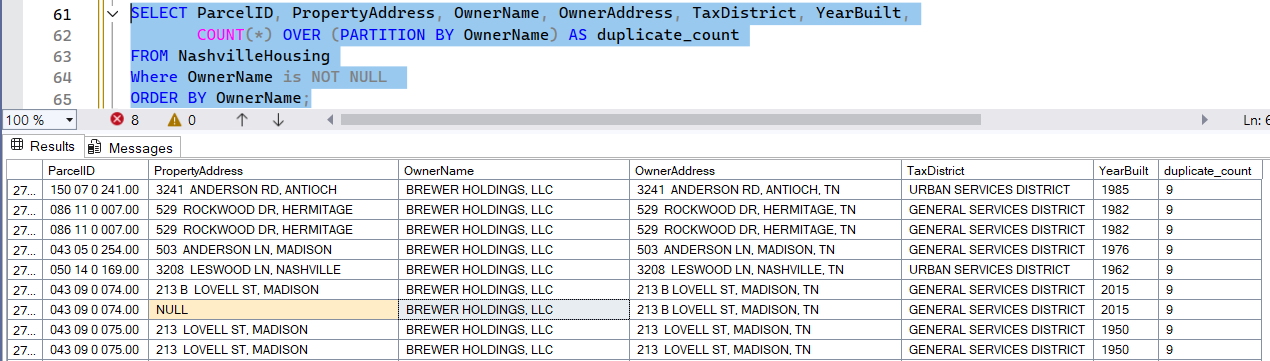
To rectify that, ORDER BY should be in the OwnerName



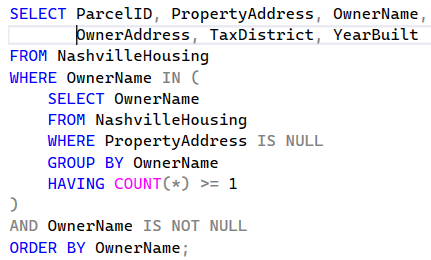
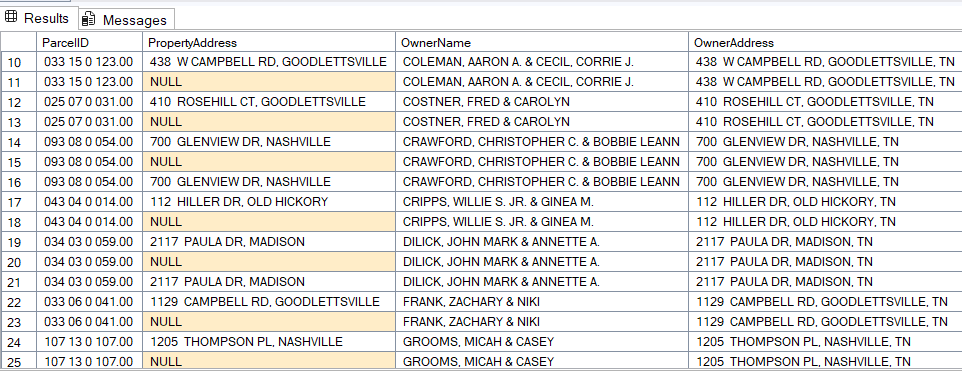
Having additional column line Tax District and Year Built can add to conclusion if there were error in the data. Here you can predict that the OwnerName might have a second property



Also by that order, by SCANNING ALL ROWS ONE BY ONE, user can see the NULL values in the PropertyAddress

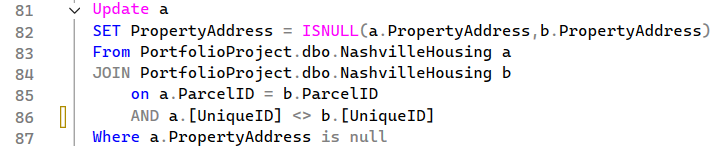


Using Subquery to show the ‘OwnerName’ that has duplicates AND have a NULL row in ‘PropertyAddress’ in realtion to ParcelID, OwnerAddress, and other selected column for double checking that the NULL column is really related to the OwnerName.

Now that the relationship is confirmed, we can proceed on updating the NULL columns

We use this query to update the column



**What this code does:**

You're using a **self-join** (joining the table to itself) to **fill in missing PropertyAddress values** for rows where it’s NULL, using values from other rows that share the same ParcelID.

**🔍 Why join the same table?**

Because you’re saying:

"If one row has a NULL in PropertyAddress, but another row with the same ParcelID has a valid address, copy that address over."

Since all this data lives in the **same table**, you have to **join the table to itself** to compare the rows.

**🧠 Let’s break down the key parts:**

**1. Self join:**

FROM NashvilleHousing a

JOIN NashvilleHousing b

ON a.ParcelID = b.ParcelID

AND a.UniqueID <> b.UniqueID

* This says: match row a with row b **if they share the same ParcelID** but are **not the same row** (UniqueID <> UniqueID ensures that).
* You're looking for "another" row that can provide the missing address.

**2. Filter for rows with missing address:**

WHERE a.PropertyAddress IS NULL

* Only fix rows where the PropertyAddress is missing in a.
  1. **ISNULL():**

ISNULL(a.PropertyAddress, b.PropertyAddress)

* If a.PropertyAddress is null, replace it with b.PropertyAddress.
* If not null, leave it as-is (in this case, you’re only targeting the null ones anyway).

**🔧 Then the UPDATE query:**

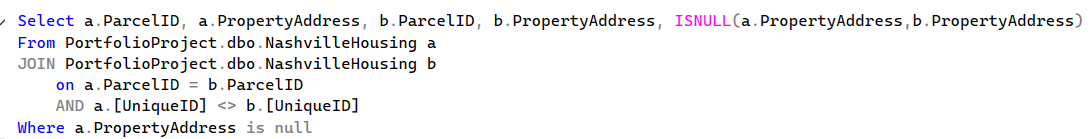
UPDATE a

SET PropertyAddress = ISNULL(a.PropertyAddress, b.PropertyAddress)

...

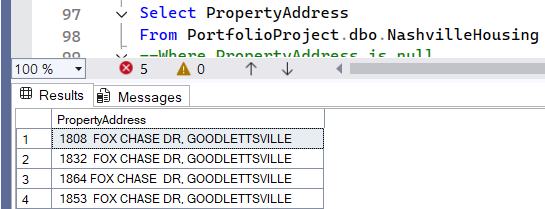
This updates each a row's PropertyAddress **using the value from b**, wherever it was null.

This code below has the same function as using the subquery to view the rows that has duplicates



IV. BREAKING OUT ADDRESS INTO INDIVIDUAL COLUMNS (Address, City, State)

This is the dataset column that will be divided. You can see that there is comma, a form of delimiter that divides two sets of data we can use that to divide this column



Introduction on Delimiter

In SQL, a **delimiter** is a **character or symbol that separates pieces of information** — like values, commands, or fields — so the database knows how to parse or interpret them.

It depends on the context, but here are the most common uses:

**🔹 a. Column Delimiter in CSV/Flat Files**

When importing or exporting data (e.g., from a CSV file), a **delimiter** separates fields in each row.

Example: a comma-delimited file (CSV):

Name,Age,City

Alice,30,New York

Bob,25,Chicago

* Here, the **comma,** is the **delimiter** that separates columns.
* You might also see other delimiters like ; , tabs (\t), or pipes (|) depending on the file format.

**🔹 b. Statement Delimiters**

In SQL scripts, a **delimiter** separates individual SQL statements.

* In most SQL systems (like SQL Server, PostgreSQL, or SQLite), the delimiter is typically a **semicolon ;**

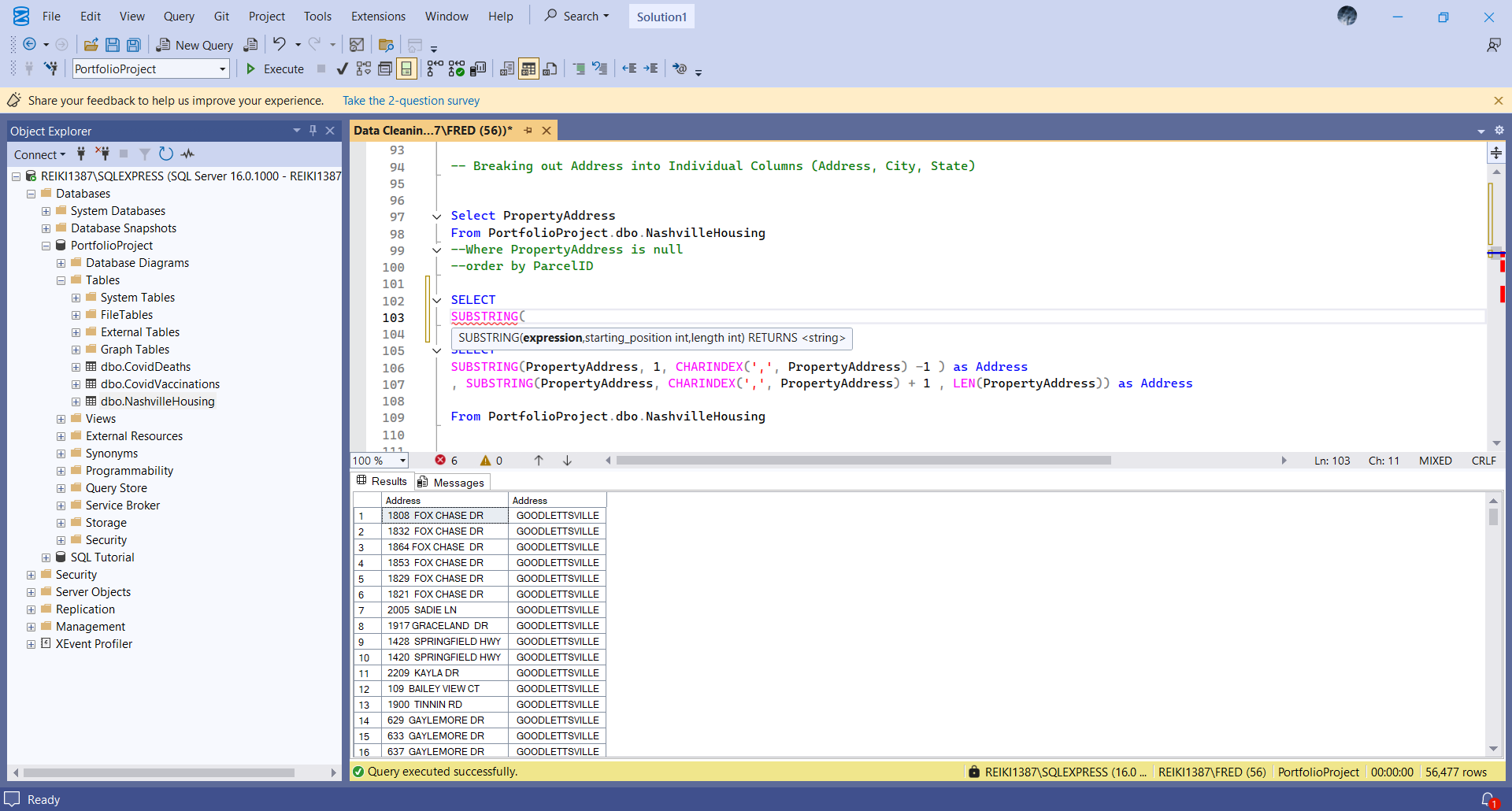
Example:

SELECT \* FROM Employees;

UPDATE Employees SET Salary = Salary \* 1.05 WHERE Department = 'Sales';

Each ; tells the SQL engine where one command ends and the next begins.

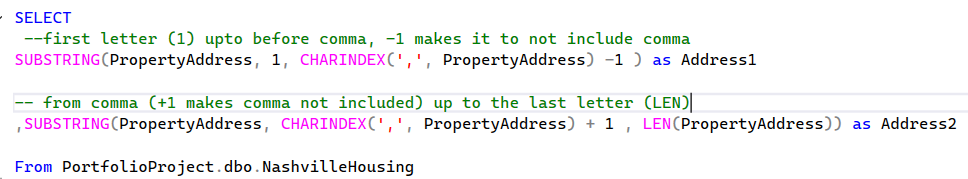
**SUBSTRING function:**

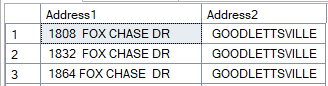


**CHARINDEX(substring, string, start)**

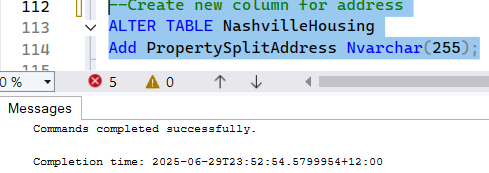
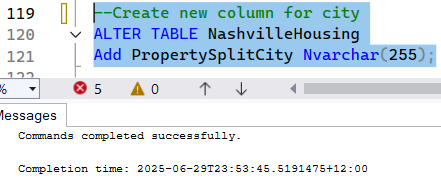
The CHARINDEX() function searches for a substring in a string, and returns the position.

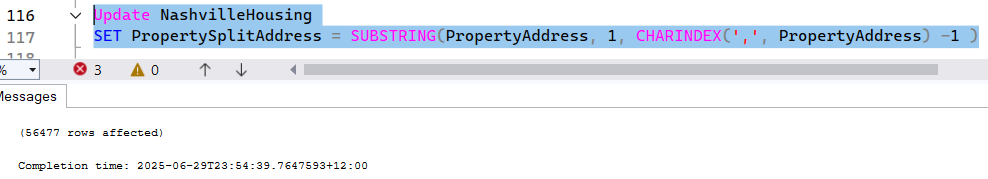
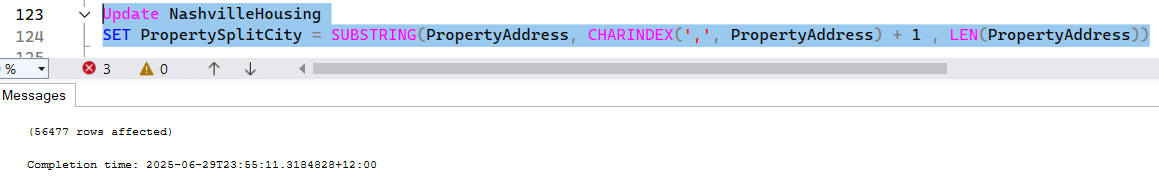
If the substring is not found, this function returns 0.

QUERY TO SHOW SEPARATING ADDRESS BY THE DE-LIMITER(COMMA) 

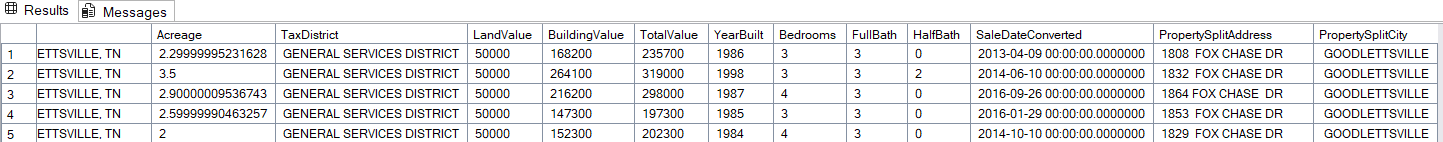


QUERY TO SPLIT THE ADDRESS

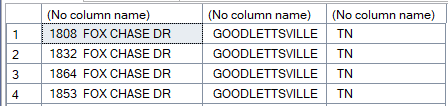
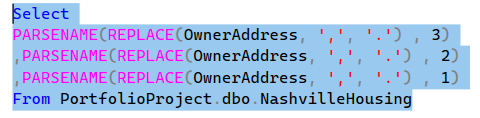
 



New column are always added at the end



**THERE IS MUCH EASIER WAY TO SEPARATE DATAS WITH DELIMITER- USING PARSENAME**



**✅ What is PARSENAME?**

PARSENAME() is a SQL Server function that **extracts parts of a string** separated by dots (.).

It was originally made to break apart **SQL object names** like Database.Schema.Table.Column, but you can also use it creatively on other strings **as long as they’re separated by dots**.

**🔍 General Syntax:**

PARSENAME('part1.part2.part3.part4', part\_number)

* part\_number is **1 for the last part**, **2 for the second-last**, and so on — up to 4 parts.
* Example:
* SELECT PARSENAME('db.schema.table.column', 1) -- returns 'column'
* SELECT PARSENAME('db.schema.table.column', 2) -- returns 'table'

**🧠 What the code is doing:**

SELECT

PARSENAME(REPLACE(OwnerAddress, ',', '.'), 3),

PARSENAME(REPLACE(OwnerAddress, ',', '.'), 2),

PARSENAME(REPLACE(OwnerAddress, ',', '.'), 1)

FROM PortfolioProject.dbo.NashvilleHousing

Here’s what it’s doing, line by line:

**1. REPLACE(OwnerAddress, ',', '.')**

* The original OwnerAddress might look like:  
  '123 Main St, Nashville, TN'
* That becomes:  
  '123 Main St. Nashville. TN'

NOTE: So you're converting **commas** to **dots**, because PARSENAME() only works with dots.

**2. PARSENAME(..., 3) → returns the first part**

* This would be: '123 Main St'

**3. PARSENAME(..., 2) → returns the second part**

* This would be: 'Nashville'

**4. PARSENAME(..., 1) → returns the third part**

* This would be: 'TN'

**⚠️ Limitation:**

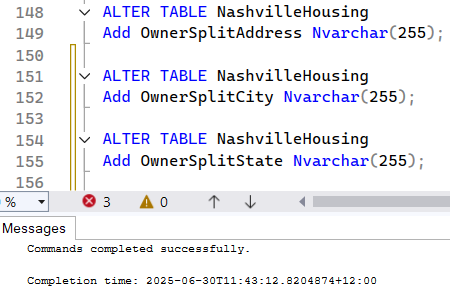
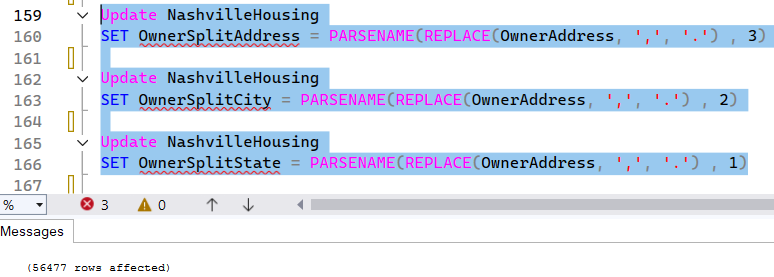
* PARSENAME() only supports up to **4 parts**.
* If your address has more than 3 commas (i.e., more than 4 parts), it may give unexpected results or NULL.

**✅ Why use this?**

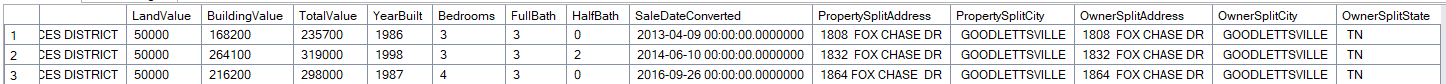
It’s a **clever trick** in SQL Server to split a string into multiple parts — like splitting an address into:

* Street
* City
* State

CREATION OF TABLE CAN BE DONE ALTOGETHER, AS WELL AS UPDATING

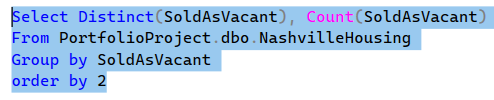
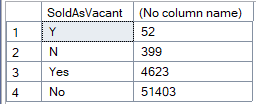
 

The Output is always added on the end

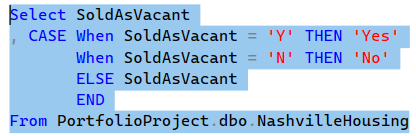
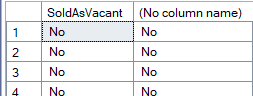


V. Changing column data using CASE Statement

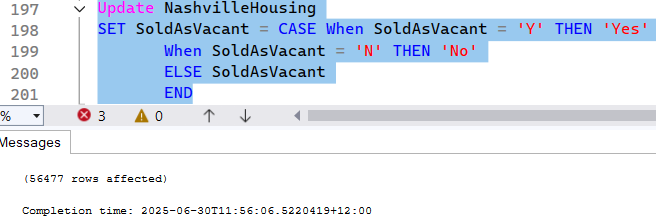
a. Checking data.

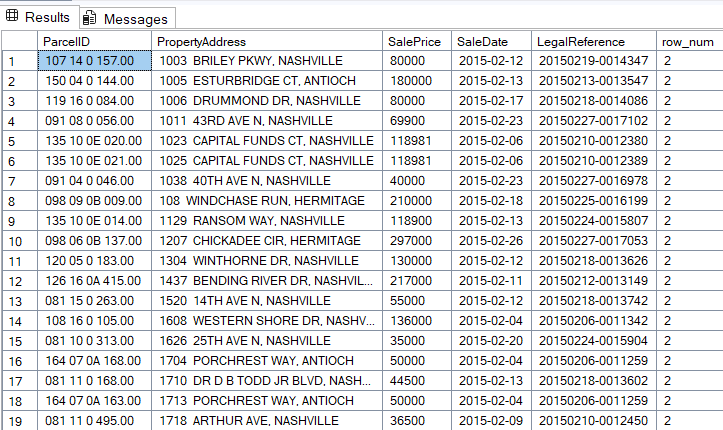
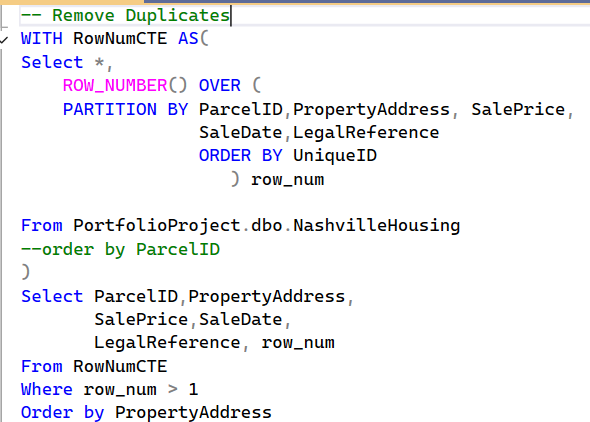
b. This SELECT statement is **just for previewing** what will happen before you actually run the UPDATE. CASE is a derived column so it doesn’t really modify the actual column. You **don’t need** this SELECT statement in order for the UPDATE to work.

c. UPDATING using CASE Statement



VI. Removing duplicates



Explanation of the SQL Code

This SQL query is using a Common Table Expression (CTE) named RowNumCTE to identify duplicate records in the NashvilleHousing table. Let me break it down:

ROW\_NUMBER() Function

ROW\_NUMBER() is a window function that assigns a sequential integer to each row within a partition of a result set. The numbering starts at 1 for the first row in each partition.

How It Works:

PARTITION BY ParcelID, PropertyAddress, SalePrice, SaleDate, LegalReference

→ Groups rows where all these columns match exactly.

→ Each unique combination of these values forms a separate partition.

ROW\_NUMBER()

→ Assigns sequential numbers (1, 2, 3, ...) within each partition.

→ The numbering restarts at 1 for each new partition.

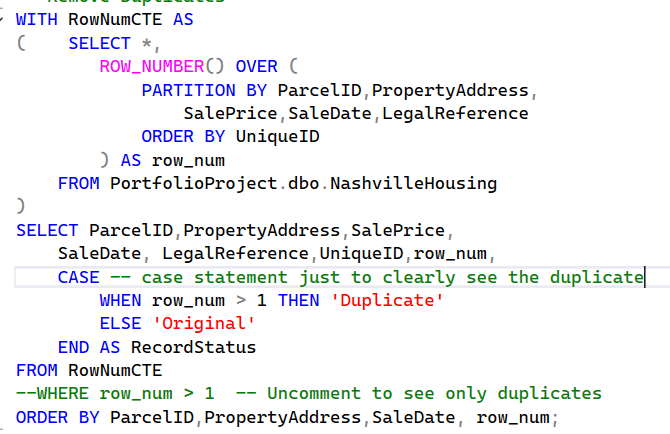
The CTE and Query

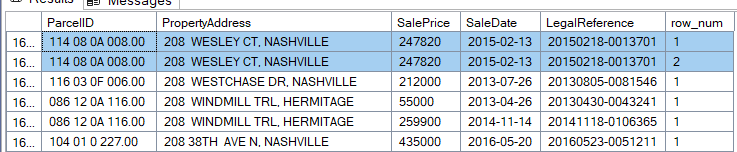
1. The CTE RowNumCTE selects all columns from NashvilleHousing plus adds the row\_num column
2. The main query then selects all rows from the CTE where row\_num > 1, which identifies duplicate records (since identical records in the same partition would get numbered 1, 2, 3, etc.)
3. Results are ordered by PropertyAddress

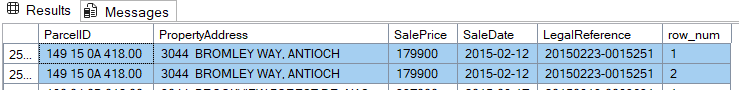
Key Difference:

* Without PARTITION BY, ROW\_NUMBER() would number all rows in the entire table sequentially (1, 2, 3,... to the end)
* With PARTITION BY, it numbers within each paritition (1, 2, 3 in group A; then 1, 2, 3 in group B; etc.)

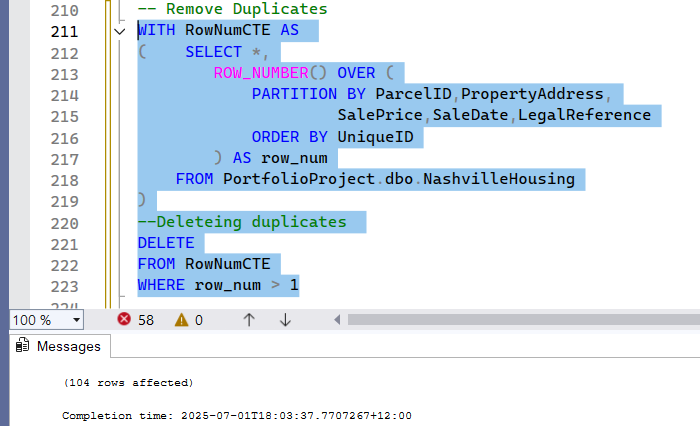
NOTE: If the WHERE statement is commented out we can see the numbering that ROW\_NUMBER() has made. The CASE statement is just additional to see the duplicate column properly



‘Partition by’ groups data by the given column name. So each ‘partition by’ will normally have only one row of data, BUT if there are rows with the same data then it will be added to the partition. ROW\_NUMBER() counts all thee row in that paritition



NOTE: Always remember that CTE always needs to be executed along with the query. If you only execute line 221-223, it will not work



DELETING UNUSED COLUMN

