**Home Sales Data Analysis**

**Overview**

This project involves analyzing home sales data using Apache Spark. The analysis includes calculating average home prices based on various attributes such as year sold, year built, number of bedrooms and bathrooms, square footage, and view ratings. The project also compares performance using different data storage and caching techniques.

Requirements:

1. **A Spark DataFrame is created from the dataset**: ✅created the home\_sales DataFrame from your dataset.
2. **A temporary table of the original DataFrame is created**: ✅ created a temporary table named home\_sales using createOrReplaceTempView.
3. **A query is written that returns the average price, rounded to two decimal places, for a four-bedroom house that was sold in each year**: ✅ wrote a query for this condition.
4. **A query is written that returns the average price, rounded to two decimal places, of a home that has three bedrooms and three bathrooms for each year the home was built**: ✅ This query was written and executed.
5. **A query is written that returns the average price of a home with three bedrooms, three bathrooms, two floors, and is greater than or equal to 2,000 square feet for each year the home was built rounded to two decimal places**: ✅ This query was also written and executed.
6. **A query is written that returns the average price of a home per "view" rating having an average home price greater than or equal to $350,000, rounded to two decimal places**: ✅ This query was written, and measured its runtime.
7. **A cache of the temporary "home\_sales" table is created and validated**: ✅ used CACHE TABLE home\_sales and checked if the table was cached using spark.catalog.isCached("home\_sales").
8. **The query from step 6 is run on the cached temporary table, and the runtime is computed**: ✅ executed the query on the cached table and measured the runtime.
9. **A partition of the home sales dataset by the "date\_built" field is created, and the formatted parquet data is read**: ✅ partitioned the data and read it into a DataFrame.
10. **A temporary table of the parquet data is created**: ✅ created a temporary table from the Parquet data.
11. **The query from step 6 is run on the parquet temporary table, and the runtime is computed**: ✅ ran the query on the parquet DataFrame and measured the runtime.
12. **The "home\_sales" temporary table is uncached and verified**: ✅ uncached the table using spark.catalog.clearCache() and confirmed it's no longer cached.

**Findings:**

# 3. What is the average price for a four bedroom house sold per year, rounded to two decimal places?

Here’s the summary:

* **2019:** $300,263.70
* **2020:** $298,353.78 (Slight decrease)
* **2021:** $301,819.44 (Increase)
* **2022:** $296,363.88 (Decrease)

# 4. What is the average price of a home for each year the home was built,# that have 3 bedrooms and 3 bathrooms, rounded to two decimal places?

These findings show that the average price of homes with **3 bedrooms and 3 bathrooms** fluctuates over the years without a clear increasing or decreasing trend:

* **2010:** $292,859.62
* **2011:** $291,117.47 (Decrease)
* **2012:** $293,683.19 (Increase)
* **2013:** $295,962.27 (Increase)
* **2014:** $290,852.27 (Decrease)
* **2015:** $288,770.30 (Decrease)
* **2016:** $290,555.07 (Increase)
* **2017:** $292,676.79 (Increase)

This indicates that while there are some fluctuations, the overall average price remains within a similar range.

# 5. What is the average price of a home for each year the home was built,

# that have 3 bedrooms, 3 bathrooms, with two floors,

# and are greater than or equal to 2,000 square feet, rounded to two decimal places?

These results highlight the **average home prices** for houses that meet the following criteria:

✅ **3 bedrooms**  
✅ **3 bathrooms**  
✅ **Two floors**  
✅ **At least 2,000 square feet**

The average prices per year:

* **2010:** $285,010.22
* **2011:** $276,553.81 (Decrease)
* **2012:** $307,539.97 (Increase)
* **2013:** $303,676.79 (Decrease)
* **2014:** $298,264.72 (Decrease)
* **2015:** $297,609.97 (Decrease)
* **2016:** $293,965.10 (Decrease)
* **2017:** $280,317.58 (Decrease)

**Observations:**

* The highest **average home price** was in **2012** at $307,539.97.
* The lowest **average home price** was in **2011** at $276,553.81.
* After **2012**, there was a steady **decline** in prices through **2017**.

# 6. What is the average price of a home per "view" rating, rounded to two decimal places,

# having an average home price greater than or equal to $350,000? Order by descending view rating.

# Although this is a small dataset, determine the run time for this query.

**Top View Ratings and Their Average Prices (Original Data):**

|  |  |
| --- | --- |
| **View Rating** | **Average Price ($)** |
| 100 | 1,026,669.50 |
| 99 | 1,061,201.42 |
| 98 | 1,053,739.33 |
| 97 | 1,129,040.15 |
| 96 | 1,017,815.92 |
| 95 | 1,054,325.60 |
| 94 | 1,033,536.20 |
| 93 | 1,026,006.06 |
| 92 | 970,402.55 |
| 91 | 1,137,372.73 |
| 90 | 1,062,654.16 |
| 89 | 1,107,839.15 |
| 88 | 1,031,719.35 |
| 87 | 1,072,285.20 |
| 86 | 1,070,444.25 |
| 85 | 1,056,336.74 |
| 84 | 1,117,233.13 |
| 83 | 1,033,965.93 |
| 82 | 1,063,498.00 |
| 81 | 1,053,472.79 |

**Observations:**

* The highest **view rating** of **100** corresponds to an **average price** of **$1,026,669.50**.
* Homes with a **view rating of 91** had the **highest average price** of **$1,137,372.73**.
* The **query runtime** for this dataset was **0.2689 seconds**, indicating a **fast** execution time.

# 9. Using the cached data, run the last query above, that calculates

# the average price of a home per "view" rating, rounded to two decimal places,

# having an average home price greater than or equal to $350,000.

# Determine the runtime and compare it to the uncached runtime.

**Comparison: Cached vs. Uncached Query Performance**

We ran the same query on **cached data** to compute the **average home price per "view" rating** for homes with an **average price ≥ $350,000**. Here's how it performed:

|  |  |
| --- | --- |
| **Query Type** | **Runtime (seconds)** |
| **Uncached** | 0.2689 sec |
| **Cached** | **0.0785 sec** ✅ **(~3.4x faster)** |

**Observations:**

* **Caching the data** significantly **improved query performance**, making it **~3.4 times faster** than the uncached version. 🚀
* The results are **sorted in descending order of "view" rating**, showing that higher-rated views generally correspond to **higher average prices**.

# 13. Using the parquet DataFrame, run the last query above, that calculates

# the average price of a home per "view" rating, rounded to two decimal places,

# having an average home price greater than or equal to $350,000.

# Determine the runtime and compare it to the cached runtime.

**Performance Comparison: Parquet vs. Cached vs. Uncached Queries**

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Runtime (seconds)** | **Performance Gain** |
| **Uncached Data** | 0.2689 sec | **Baseline** |
| **Parquet Data** | 0.1899 sec | **~1.4x faster** 🚀 |
| **Cached Data** | **0.0785 sec** ✅ | **~3.4x faster** 🔥 |

**Observations:**

* **Parquet format** improved performance **compared to the uncached query (~1.4x faster)** due to its **columnar storage and compression**.
* **Caching the DataFrame** was the **fastest approach (~3.4x faster than uncached)**, since Spark keeps the data **in memory**, avoiding disk reads.

**Conclusion:**

* If you **repeatedly** run a query, **caching** is the best option. 🏆
* For **one-time queries**, **Parquet** provides better performance than raw data. 🔥