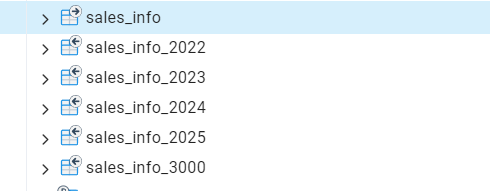
**Step 1: Create Parent Table**

* **Explanation**:  
  The SALES\_INFO table is the base table containing columns that will be shared across all partitioned child tables. This table includes the id, category, ischeck, and eventdate columns. The partitioning mechanism will ensure that data is spread across multiple child tables based on the eventdate.
* **Key Insight**:  
  The parent table holds the schema for partitioned tables and doesn't store any data itself. It is mainly used to define the structure and constraints.

**Step 2: Create Child Tables (Partitions)**

* **Explanation**:  
  Here, we create multiple child tables (sales\_info\_2021, sales\_info\_2022, etc.), each covering a specific year. These tables inherit from the SALES\_INFO parent table, but have a CHECK constraint to enforce data for specific years.
* **Key Insight**:  
  Partitioning by date allows for more efficient data management, where each partition holds records for a specific year. This can lead to faster query performance when filtering by date.

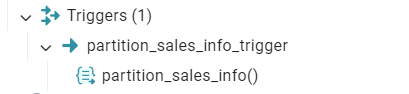


**Step 3: Create Partition Function**

* **Explanation**:  
  The function partition\_sales\_info() ensures that any data inserted into the SALES\_INFO table gets automatically placed into the correct child table based on the eventdate. Each INSERT operation checks the eventdate and routes the data to the appropriate partition.
* **Key Insight**:  
  Using a function like this allows for automation in partitioning, meaning you don't need to manually insert records into the correct partitions. This also ensures that the system is self-managing and scales with new data.

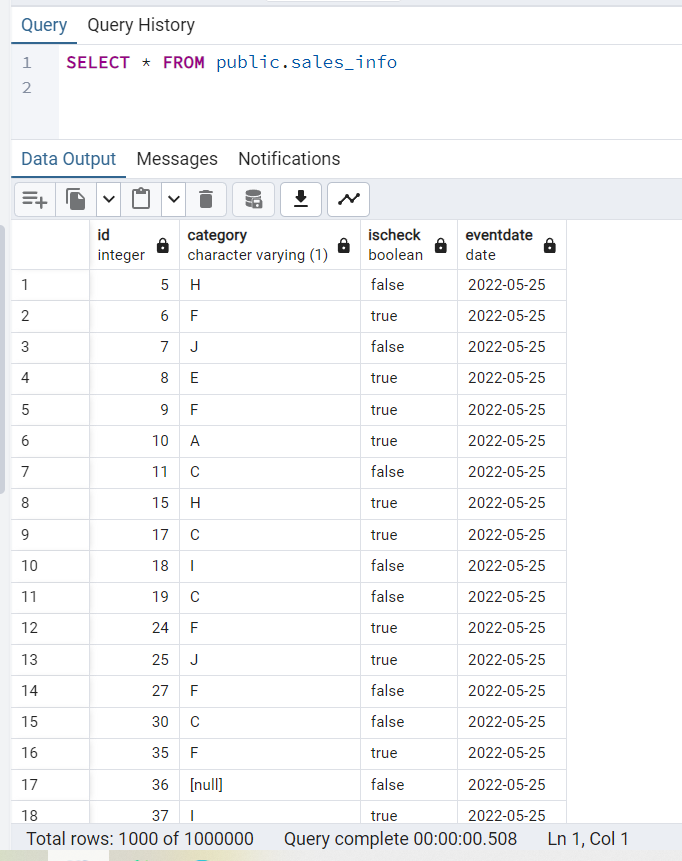
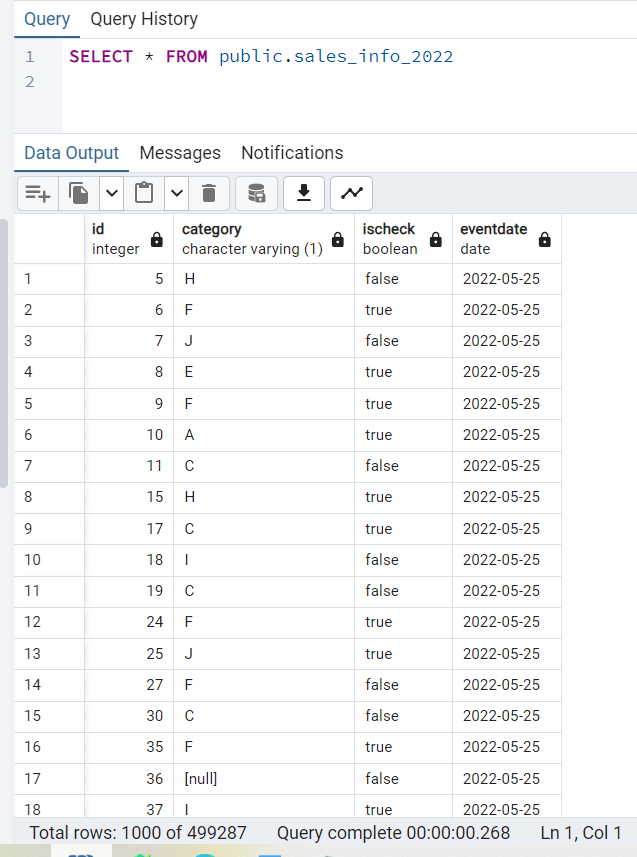
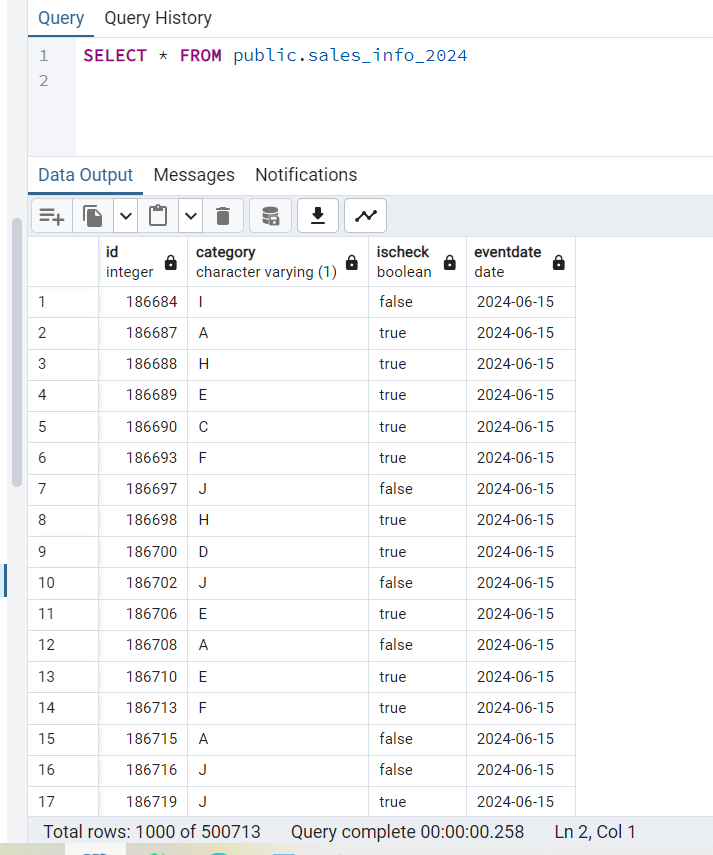
**Step 4: Create Trigger for Partitioning**

* **Explanation**:  
  The trigger partition\_sales\_info\_trigger calls the partition\_sales\_info() function before any insert operation on the SALES\_INFO table. This ensures that the data is routed to the correct partition before it is stored.
* **Key Insight**:  
  Triggers allow automatic actions when certain events (like inserts) happen. In this case, it ensures that partitioning happens without extra manual intervention.



**Step 5: Generate and Insert Test Data**

* **Explanation**:  
  This step generates random test data for SALES\_INFO using the generate\_series function. A million rows are inserted with random eventdate values, category, and ischeck flags.
* **Key Insight**:  
  Using synthetic data is useful for performance testing. The large volume of data will help in observing the behavior of the partitioned vs. non-partitioned tables. With partitioning, we expect inserts to be faster because each row is inserted into its corresponding partition instead of a single large table.

**Step 6: Update Some Rows in SALES\_INFO**

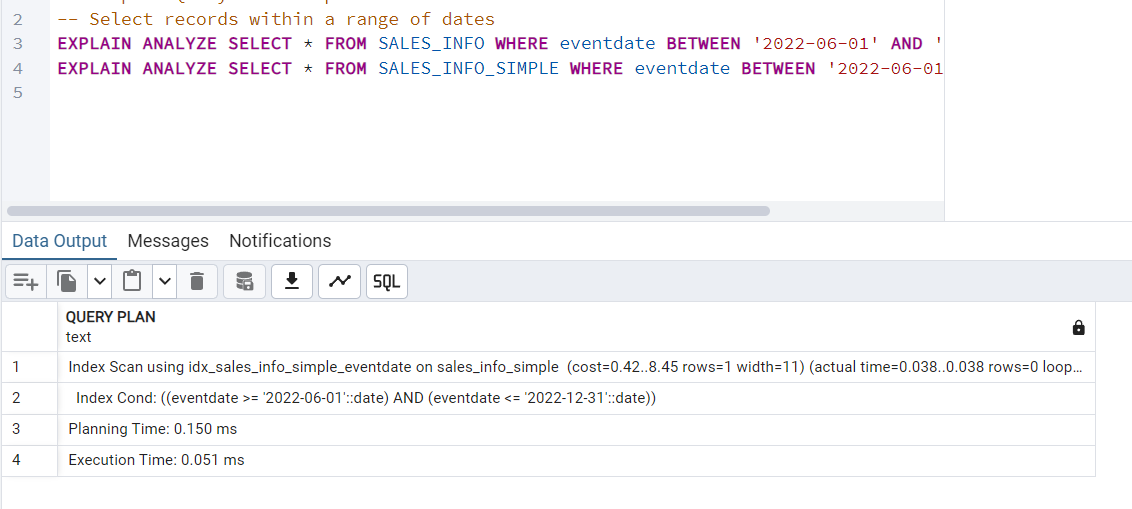
* **Explanation**:  
  Here, we delete some rows from SALES\_INFO where the eventdate is within 2025, and then we insert the same data with a new eventdate. This demonstrates how partitioning works with updates: the data will be deleted from one partition and inserted into another.
* **Key Insight**:  
  Partitioning can complicate updates because rows might need to be moved across partitions. This is an important consideration when deciding between partitioning schemes.

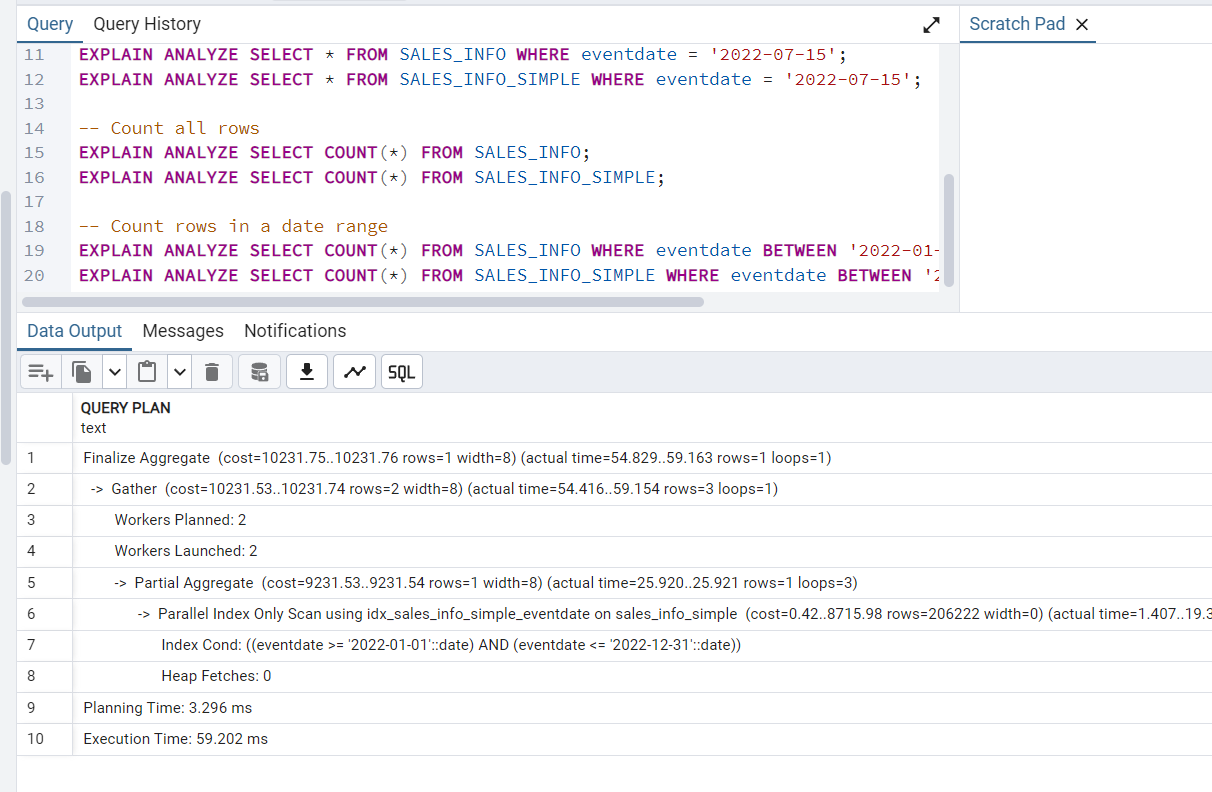
**Step 7: Query Plan Comparisons**

* **Explanation**:  
  Here, we compare the query execution plans of partitioned (SALES\_INFO) vs. unpartitioned (SALES\_INFO\_SIMPLE) tables using EXPLAIN ANALYZE. The goal is to compare how queries are optimized when running on partitioned vs. non-partitioned data.
* **Key Insights**:
  + **Partitioned Tables**: Queries that filter by eventdate (e.g., for a specific year or range) benefit from partition pruning, meaning only the relevant partitions are scanned. This can significantly reduce the number of rows processed.
  + **Unpartitioned Tables**: Queries need to scan the entire table, which becomes inefficient as the table size grows. Partitioning allows for targeted scans and can speed up queries dramatically.

**Example of Performance Impact**:

* + **Query for a Date Range**: A query filtering by eventdate BETWEEN '2022-06-01' AND '2022-12-31' will only scan the sales\_info\_2022 partition when partitioning is in place. Without partitioning, the query scans the entire SALES\_INFO\_SIMPLE table.





**Step 8: Delete Old Partition and Add New Partition**

* **Explanation**:  
  In this step, we remove the old partition (sales\_info\_2021) and add a new partition for data beyond 2025 (sales\_info\_3000). This shows how easy it is to add or remove partitions when needed.
* **Key Insight**:  
  Partition management is flexible; you can easily add or remove partitions as data grows or becomes outdated. This flexibility ensures that the database can scale effectively over time.

**Step 9: Declarative Partitioning with Range and List**

* **Explanation**:  
  This step demonstrates declarative partitioning by creating range partitions based on eventdate and further subdividing them into list partitions based on category. This creates multiple layers of partitioning, making the data more organized.
* **Key Insight**:  
  Declarative partitioning, where partitions are defined using PARTITION BY RANGE and PARTITION BY LIST, allows for finer control over how the data is distributed. This can lead to performance improvements for complex queries filtering both on date and category.

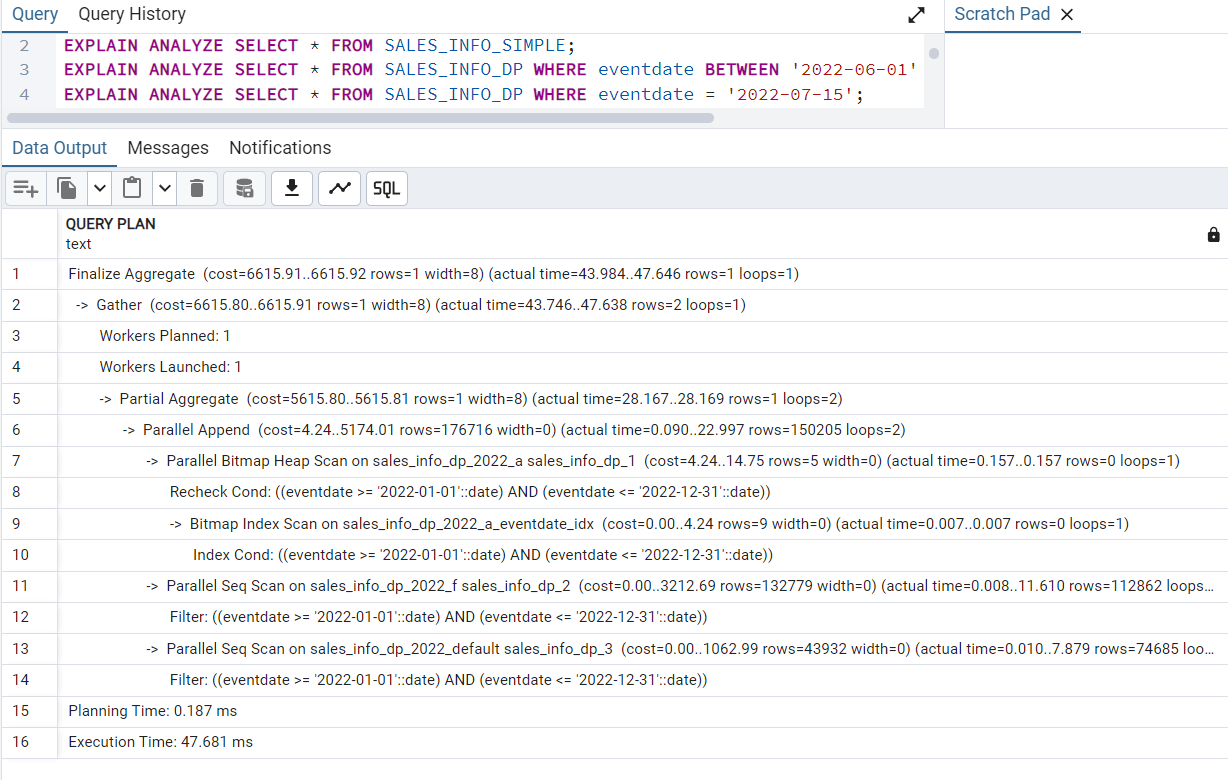


**Step 10: Insert Data into Declarative Partitions**

* **Explanation**:  
  Data is inserted into the SALES\_INFO\_DP table with declarative partitioning. Each row is routed based on both the eventdate (for range partitioning) and category (for list partitioning).
* **Key Insight**:  
  Declarative partitioning gives better control and optimization over how data is distributed. By combining both range and list partitioning, queries can be optimized for both time and specific categories.

**Step 11: Query Performance Comparisons for Declarative Partitioning**

* **Explanation**:  
  Similar to Step 7, we now compare the performance of queries run on SALES\_INFO\_DP (declarative partitioning) with SALES\_INFO\_SIMPLE (unpartitioned). The goal is to see how partitioning improves performance when filtering by both eventdate and category.
* **Key Insight**:  
  Partitioning not only improves query speed for date-based filtering, but also for category-based filtering. As queries become more specific (e.g., filtering by both date and category), partitioning helps reduce the number of rows scanned, leading to better performance.



**Step 14: SET max\_parallel\_workers\_per\_gather = 4;**

**Explanation:**

This command configures PostgreSQL to use parallel workers for queries that can benefit from parallel execution. Specifically, it sets the maximum number of parallel workers that can be used in a Gather or Gather Merge operation, which is involved in parallel query execution. max\_parallel\_workers\_per\_gather defines the number of parallel workers allowed to be used per query. By setting this to 4, you are instructing PostgreSQL to use up to 4 parallel workers for a single query, provided the query planner determines that parallelism is beneficial for execution. This can help improve the performance of large queries by distributing the work across multiple CPU cores.

**Key Insight:**

Parallel queries are particularly beneficial when scanning large datasets. By using multiple workers, the database can perform operations (like sorting, filtering, etc.) faster, especially with partitioned tables where the data is spread across multiple partitions.

**Step 15: Indexing and Parallel Query Performance. Creating Indexes**

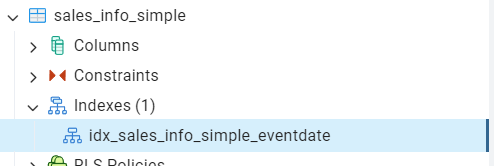
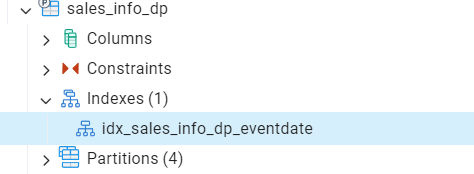
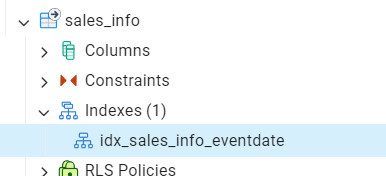
**Explanation:**

CREATE INDEX idx\_sales\_info\_eventdate ON SALES\_INFO(eventdate);

CREATE INDEX idx\_sales\_info\_dp\_eventdate ON SALES\_INFO\_DP(eventdate);

CREATE INDEX idx\_sales\_info\_simple\_eventdate ON SALES\_INFO\_SIMPLE(eventdate);

These indexes are created on the eventdate column in each table. Indexing improves query performance, especially for queries that filter by eventdate. When combined with partitioning, indexes can speed up searches by reducing the need for full table scans.

**Conclusion**

During this homework, I learned how to perform different types of partitioning such as inheritance and declartive. Also I tried a parallel quering for comparing execution plans for SALES\_INFO (inheritance-based partitioning), SALES\_INFO\_DP (declarative partitioning), and SALES\_INFO\_SIMPLE (unpartitioned). In conclusion, I firgured out that Partitioning + Indexing + Parallel Queries gives the best performance.