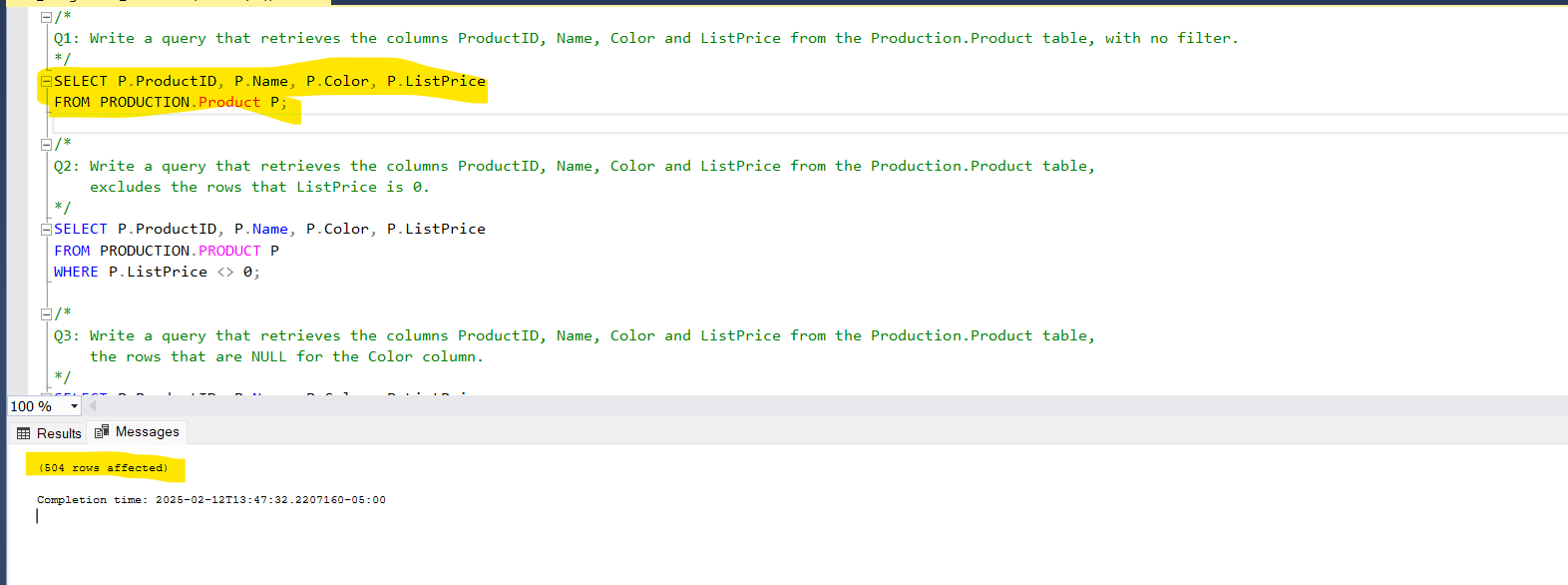
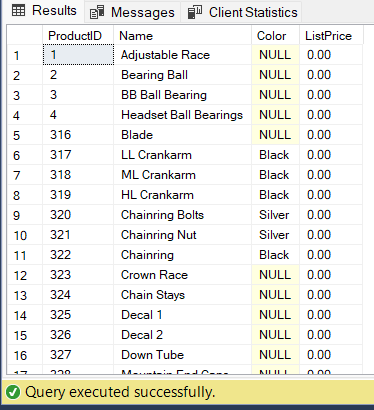
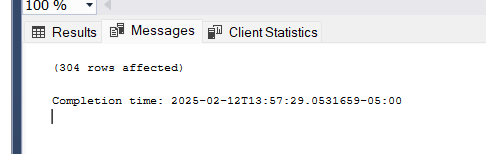
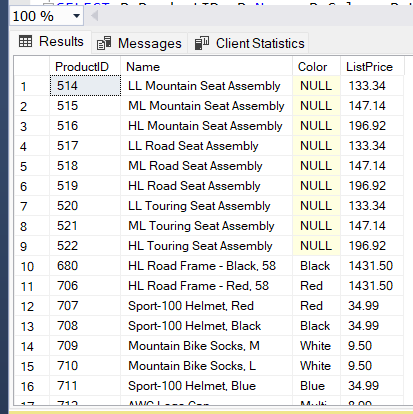
1. 

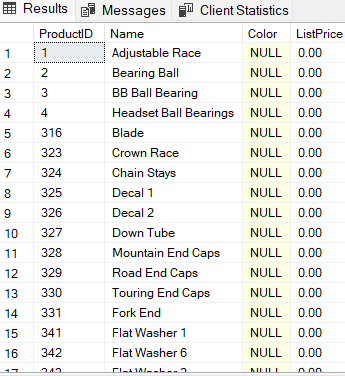


2.

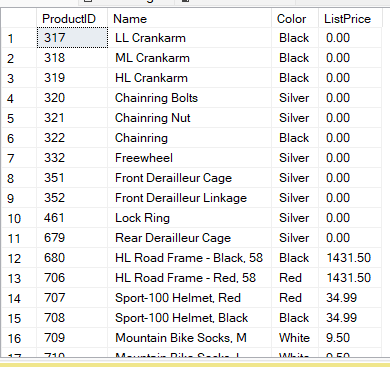


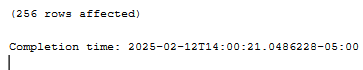


3.

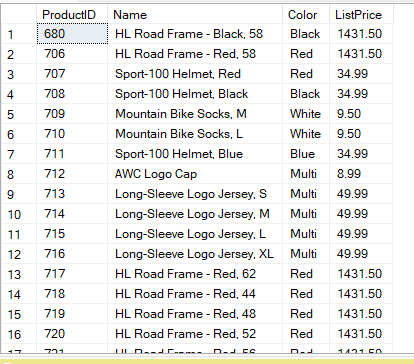
 

4.





5.



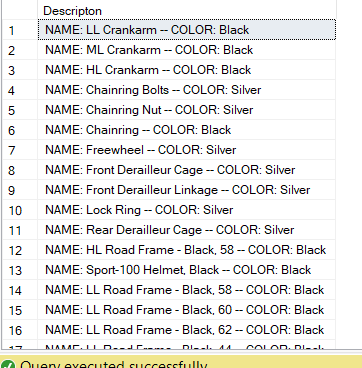


6.



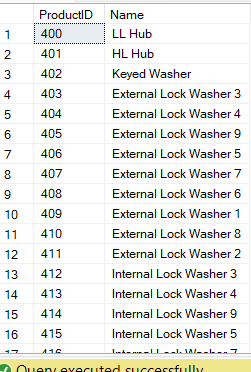


7.



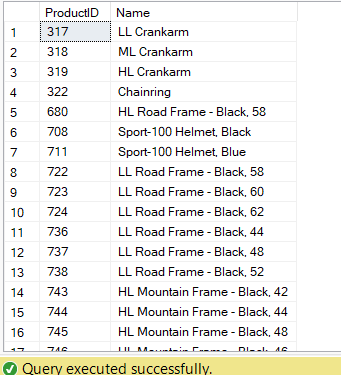


8.



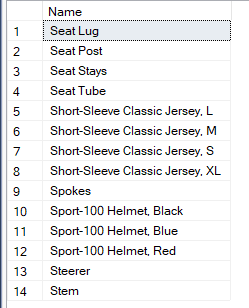


9.



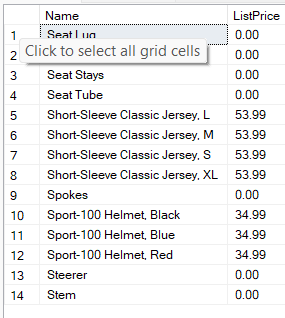


10.



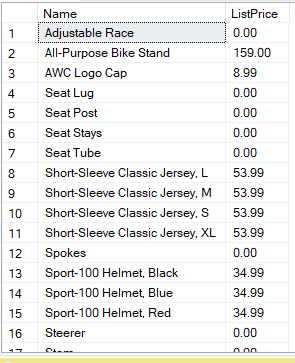


11.



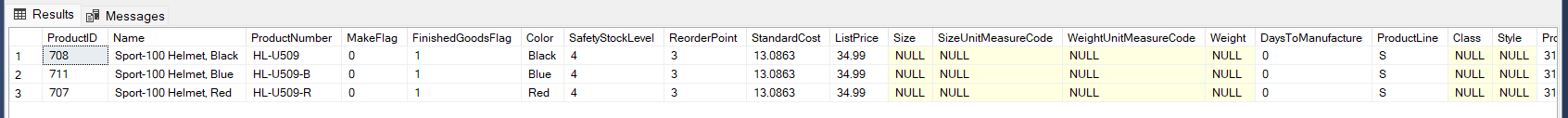


12.





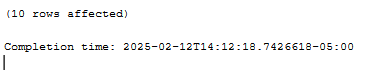
13.



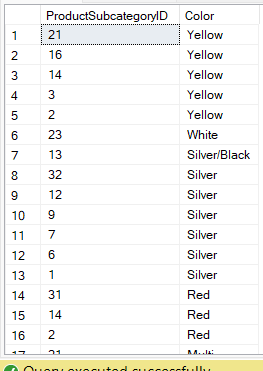


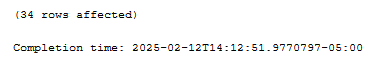
14.





15.





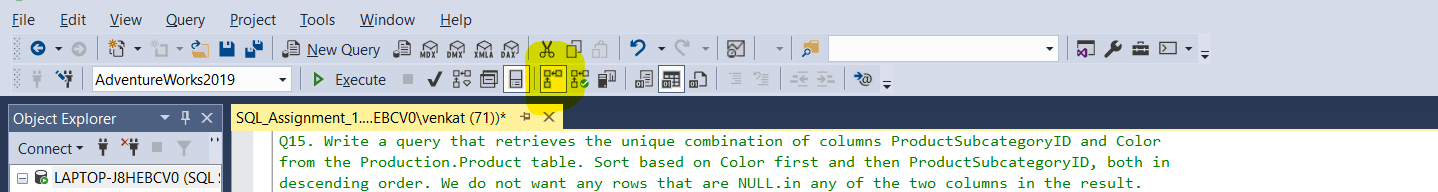
16. UNION and UNION ALL combine results from multiple SELECT statements, but they differ in handling duplicates. UNION removes duplicate rows, performing an implicit DISTINCT operation, which slows performance due to sorting and deduplication. In contrast, UNION ALL retains all rows, including duplicates, making it faster. If uniqueness is required, UNION is preferred; otherwise, UNION ALL is more efficient when duplicates are acceptable.

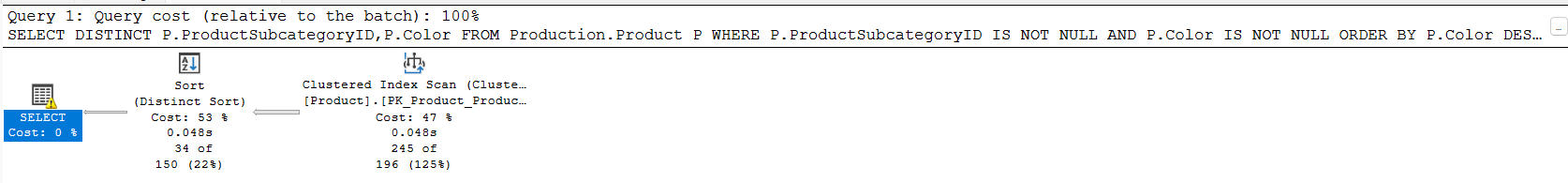
For example, SELECT city FROM customers UNION SELECT city FROM suppliers; returns unique cities, whereas replacing UNION with UNION ALL includes all occurrences, even if repeated. Choosing the right one depends on the use case.

17. The WHERE and HAVING clauses both filter records in SQL but differ in how and when they are applied.

* WHERE filters rows **before** aggregation (GROUP BY). It applies conditions to individual rows in a table and is used with standard comparisons (=, >, <, LIKE, etc.).
* HAVING filters groups **after** aggregation. It is used with aggregate functions (SUM(), AVG(), COUNT(), etc.) to filter grouped records.

18. To be able to execute queries, the SQL Server Database Engine must analyze the statement to determine an efficient way to access the required data and process it. This analysis is handled by a component called the Query Optimizer. The input to the Query Optimizer consists of the query, the database schema (table and index definitions), and the database statistics. The Query Optimizer builds one or more *query execution plans*, sometimes referred to as *query plans* or *execution plans*. The Query Optimizer chooses a query plan using a set of heuristics to balance compilation time and plan optimality in order to find a good query plan.





19. Joins in SQL are used to combine data from multiple tables based on a related column.

1. **INNER JOIN**
   * Returns only the matching rows from both tables.
   * Example:

SELECT users.user\_id, users.username, training\_details.training\_id

FROM users

INNER JOIN training\_details ON users.user\_id = training\_details.user\_id;

* + Output includes only users who have training records.

1. **LEFT JOIN (LEFT OUTER JOIN)**
   * Returns all rows from the left table (users), and matching rows from the right (training\_details). If no match, NULLs are returned for the right table.
   * Example:

SELECT users.user\_id, users.username, training\_details.training\_id

FROM users

LEFT JOIN training\_details ON users.user\_id = training\_details.user\_id;

* + Includes all users, even those without training.

1. **RIGHT JOIN (RIGHT OUTER JOIN)**
   * Returns all rows from the right table (training\_details), and matching rows from the left (users). If no match, NULLs are returned for the left table.
   * Example:

SELECT users.user\_id, users.username, training\_details.training\_id

FROM users

RIGHT JOIN training\_details ON users.user\_id = training\_details.user\_id;

* + Ensures all training details appear, even if no user is linked.

1. **FULL JOIN (FULL OUTER JOIN)**
   * Returns all rows from both tables, with NULLs where there is no match.
   * Example:

SELECT users.user\_id, users.username, training\_details.training\_id

FROM users

FULL JOIN training\_details ON users.user\_id = training\_details.user\_id;

* + Ensures no data is excluded.

1. **CROSS JOIN**
   * Returns the Cartesian product of both tables (each row from the first table pairs with every row from the second).
   * Example:

SELECT users.user\_id, users.username, training\_details.training\_id

FROM users

CROSS JOIN training\_details;

* + Useful for combinations but can be large in size.

20. SELECT \* FROM users

WHERE user\_id % 2 = 0;

21. SELECT TOP 100 user\_id, username

FROM users

WHERE user\_id % 2 = 1

ORDER BY user\_id;

22. 1. Using left join and with is null

SELECT u.user\_id FROM dbo.users u LEFT JOIN dbo.users2 u2 ON u.user\_id = u2.user\_id WHERE u2.user\_id IS NULL;

2. Using not in

SELECT user\_id FROM dbo.users WHERE user\_id NOT IN (SELECT user\_id FROM dbo.users2);

23. SELECT DISTINCT user\_id FROM training\_details WHERE training\_date IN ('2015-08-02', '2015-08-03') INTERSECT SELECT DISTINCT user\_id FROM training\_details WHERE training\_date = '2015-08-04';

24. SQL Server Profiler is a graphical tool used for monitoring, capturing, and analyzing database events in Microsoft SQL Server. It helps database administrators (DBAs) and developers troubleshoot issues, optimize queries, and audit activities.

If a query is running slow, SQL Profiler can **capture execution details**, helping pinpoint issues like:

* Missing indexes
* Locking/blocking problems
* High CPU/memory usage

25. In SQL Server, a local variable is declared using the DECLARE statement and is limited to the batch or stored procedure where it is defined.

26. 1. master – Stores system-wide configuration, login accounts, and database metadata. Essential for SQL Server startup.

2. model – Acts as a template for new databases, defining default settings like recovery model and collation.

3. msdb – Manages SQL Server Agent jobs, backups, alerts, and automation tasks.

4. tempdb – Handles temporary tables, sorting, indexing, and cursor operations. Recreated on every SQL Server restart.

27. **DDL (Data Definition Language) Statements** – CREATE, ALTER, DROP, TRUNCATE, RENAME (Used for defining or modifying database structures).

**DML (Data Manipulation Language) Statements** – SELECT, INSERT, UPDATE, DELETE, MERGE (Used for modifying and retrieving data in tables).