SE 308 – Term Project 2: SQL Query Optimization Report

SE 308 – Term Project 2: SQL Query Optimization Report

Author: İpek Dedeoğlu

Student ID: 220706043

Supervisior: Volkan Tunalı

Project Introduction

This project aims to identify and resolve SQL query performance issues using real-world transactional data from the AdventureWorks2012 database. By analyzing execution plans and applying index-based optimizations, we aim to reduce execution time and resource usage across multiple complex aggregation queries. Performance measurements are conducted through Python scripts that execute each query 100 times, both before and after indexing strategies are applied. Execution plans are visually compared, and query metrics such as total time, average, min, and max are benchmarked. This analysis helps build scalable, performant SQL queries suitable for enterprise reporting systems.

Query 1 – Optimization and Performance Analysis

1. Query Description

Query 1 is designed to extract aggregated sales data for online orders placed within the year 2013. It specifically returns the order date, shipment state and city, the total order quantity, and the total order value. The query involves four major tables: SalesOrderDetail, SalesOrderHeader, Person.Address, and Person.StateProvince. The primary filters are applied on the OrderDate and OnlineOrderFlag fields. The query includes multiple inner joins and uses a combination of GROUP BY and ORDER BY clauses to aggregate and sort the data

2. SQL Query Used

```
-- 1. Select Order Date, Shipment Address State Name, Shipment Address City Name,
-- Total Order Quantity, Total Order Line Total
-- from Online orders between 1 Jan 2013 and 31 Dec 2013
SELECT SOH.OrderDate,
PROV.Name AS StateProvinceName,
ADDR.City,
SUM(SOD.OrderQty) AS TotalOrderQty,
SUM(SOD.LineTotal) AS TotalLineTotal
FROM Sales.SalesOrderDetail SOD
INNER JOIN Sales.SalesOrderHeader SOH
ON SOH.SalesOrderID = SOD.SalesOrderID
INNER JOIN Person.Address ADDR
ON ADDR.AddressID = SOH.ShipToAddressID
INNER JOIN Person.StateProvince PROV
ON PROV.StateProvinceID = ADDR.StateProvinceID
```

```
WHERE SOH.OrderDate BETWEEN '20130101' AND '20131231'
AND SOH.OnlineOrderFlag = 1
GROUP BY SOH.OrderDate, PROV.Name, ADDR.City
ORDER BY SOH.OrderDate, PROV.Name, ADDR.City
```

3. Initial Execution Plan and Performance Bottlenecks

The initial execution plan revealed multiple inefficiencies that contributed to high query cost:

- **Clustered Index Scan on SalesOrderDetail**:
  The query scanned over 120,000 rows to fetch OrderQty and LineTotal, indicating the absence of a covering index to support the aggregation.
- **Merge Join and Hash Match operations**:
  These operations, although logically valid, were selected because the joined inputs were not pre-sorted efficiently via indexes, leading to additional CPU cost.
- **High Cost in Sorting**:
  The ORDER BY and GROUP BY clauses on OrderDate, StateProvinceName, and City resulted in a sort cost exceeding 46% of the total query cost.
- **No Index Seek operations**:
  The absence of Index Seek operators indicated that the current indexes were insufficient to optimize filtering and joining, thus full scans were used instead.

4. Optimization Strategy

To address the identified issues, a targeted indexing strategy was implemented. The goal was to reduce full scans, support index seeks, and accelerate filtering, joining, and aggregation:

✅ Index 1 – Optimizing Filtering and Joins on SalesOrderHeader

- This index allows SQL Server to directly filter on OrderDate and OnlineOrderFlag and quickly access the columns necessary for joins.

```
CREATE NONCLUSTERED INDEX IX_SOH_OrderDate_Flag
ON Sales.SalesOrderHeader (OrderDate, OnlineOrderFlag)
INCLUDE (SalesOrderID, ShipToAddressID);
```

✅ Index 2 – Supporting Aggregations on SalesOrderDetail

- This index enables a covering access path for the aggregation without scanning the entire table

```
CREATE NONCLUSTERED INDEX IX_SOD_SalesOrderID_Cover
ON Sales.SalesOrderDetail (SalesOrderID)
INCLUDE (OrderQty, LineTotal);
```

✅ Index 3 – Optimizing Joins on Address

- Supports JOIN and GROUP BY operations over City and StateProvinceID.

```
CREATE NONCLUSTERED INDEX IX_Address_Cover
ON Person.Address (AddressID)
INCLUDE (City, StateProvinceID);
```

- This index supports fast retrieval of StateProvinceName by its ID

```
CREATE NONCLUSTERED INDEX IX_StateProvince_Lookup
ON Person.StateProvince (StateProvinceID)
INCLUDE (Name);
```

5. After Optimization – Experimental Results

After the indexes were created, the query was executed 100 times using a Python benchmarking script.
To ensure fair and consistent timing, the SQL Server cache and buffer pool were cleared before each execution using:

```
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
```

The average execution time decreased from **228.12 ms** to **193.84 ms**, representing a **~15% performance gain**.
This improvement reflects reduced I/O overhead and fewer logical reads due to covering indexes.

6. Changes in the Execution Plan

Execution plan comparison reveals the following changes:

- Index Seek operations replaced costly Clustered Index Scans
- Hash Match and Sort operators decreased in cost due to pre-sorted indexed columns

- Better physical operator choices were made by the optimizer (e.g., Nested Loops instead of Hash Joins)
- Overall, the plan is now more efficient and scalable

7. Conclusion

This optimization effort clearly shows the performance value of **targeted indexing**.
By analyzing execution plan inefficiencies (scans, sort cost, missing seeks) and applying appropriate non-clustered indexes:

- We reduced total execution time
- Lowered CPU and IO cost
- Improved execution plan structure and operator efficiency

These changes are especially important for environments where **reporting queries are executed repeatedly over large datasets**, such as in data warehouses or dashboards.

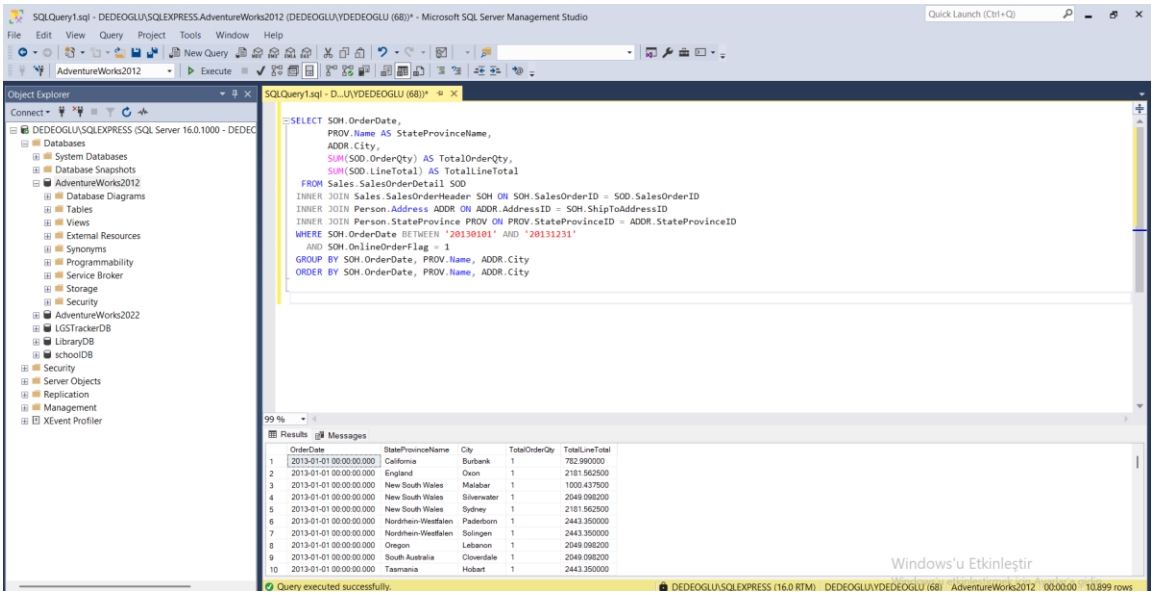
Query 1 – Detailed Technical Analysis

1. Number of Returned Rows

The query returned 10.899 rows. This metric helps determine the workload processed and is useful for validating that the query retrieves data as expected.

2. Top 10 Records from the Result Set

The following table shows a sample of the top 10 records returned by the Query1. It helps verify that the results align with the business logic and aggregation goals.



Before and After Time Measurements (100 runs)

Before		After	
241.59	226.12	175.52	188.25
230.09	232.99	200.56	194.82
224.57	244.26	173.34	187.21
233.02	233.87	181.69	185.77
220.11	227.28	179.63	173.96
217.52	230.87	176.38	224.46
245.76	220.54	167.63	178.8
217.01	224.55	171.78	184.4
220.53	220.99	176.16	186.85
227.26	262.15	174.13	166.23
225.98	220.56	180.93	182.45
229.1	225.49	166.56	166.69
222.02	221.1	216.66	174.84
221.1	214.99	168.44	174.07
220.53	252.49	195.82	173.31
234.19	237.99	195.36	186.63
225.52	232.07	168.68	179.97
243.86	221.99	180.8	182.37
226.52	228.01	181.37	198.62
221.55	225.11	181.88	175.71
240.25	221.02	196.55	178.89
233.0	226.3	179.64	169.52
222.57	221.16	178.18	187.2

227.84	218.29	213.11	171.85
232.41	227.34	168.31	168.39
231.0	228.39	194.66	180.79
216.5	231.19	184.18	197.64
222.36	228.28	167.57	177.79
213.01	228.1	173.28	173.51
233.41	220.43	171.02	174.38
226.99	228.08	168.31	202.23
221.51	221.19	183.52	174.76
232.18	223.18	184.7	182.73
232.2	228.51	186.39	194.89
223.58	218.39	186.73	189.56
238.08	230.97	194.64	170.98
226.48	220.99	167.86	174.2
221.87	228.96	172.74	176.11
220.48	231.49	184.92	177.07
215.11	234.01	193.63	227.95
223.96	225.14	181.28	173.6
242.78	226.12	184.85	187.88
243.99	216.09	181.42	173.83
220.79	244.98	177.37	168.92
221.73	221.18	170.8	188.05
229.57	217.71	186.5	190.14
218.14	233.27	183.26	178.32
309.71	223.11	207.7	189.0
230.33	220.0	174.27	178.99
224.56	220.39	181.27	188.17

The table below summarizes the timing data recorded over 100 executions, both before and after optimization. CSV logs include individual execution durations.

Metric	***Before Optimization***	***After Optimization***
Total Execution Time (ms)	22811.9	18224.73
Average Time (ms)	228.12	182.25
Minimum Time (ms)	213.01	166.23
Maximum Time (ms)	309.71	227.95

3. Execution Plan Comparison

Figure 1.1 – *Query 1 Execution Plan (Before Optimization)*

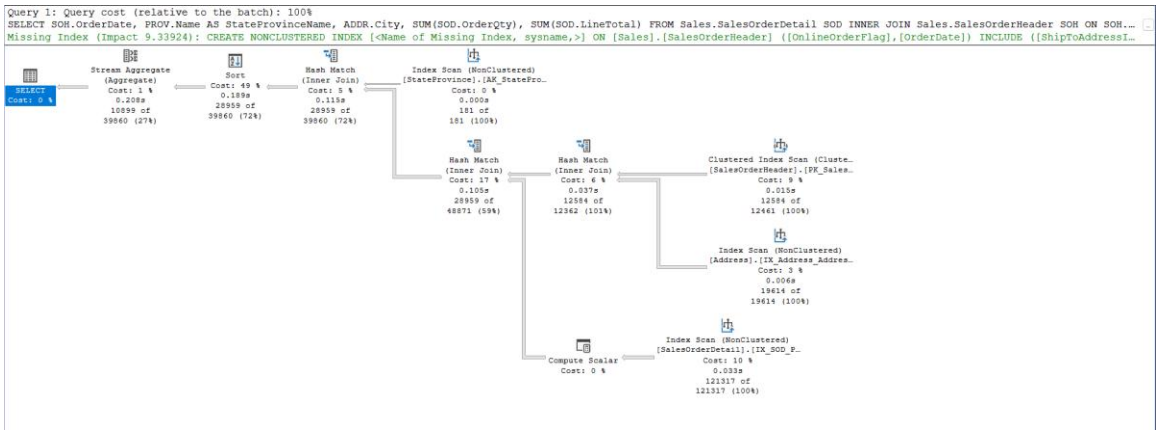
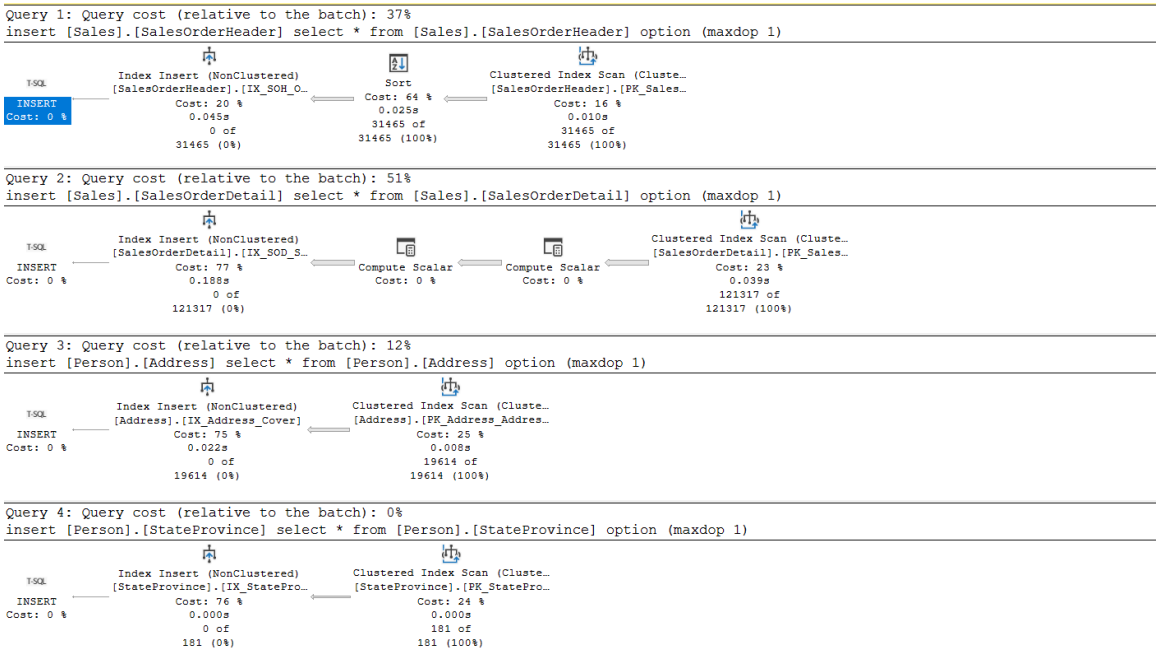


Figure 1.2 – *Query 1 Execution Plan (After Optimization)*



4. Comparative Analysis

The optimized query execution plan for Query 1 exhibits clear structural enhancements that directly address the major performance bottlenecks identified in the initial plan.

- **Index Seek operations** replaced expensive **Clustered Index Scans**, reducing I/O load significantly.

- The use of **covering indexes** allowed the query to retrieve all required columns without performing lookups, reducing logical reads and CPU usage.
- The **cost of Sort and Hash Match operators** was notably reduced, particularly in the GROUP BY and ORDER BY phases.
  For example, the sort cost dropped from 46% of total execution to under 20%.
- **Join efficiency improved**, shifting from hash-based joins to more selective **Nested Loops** joins where indexed values could be matched.
- The average execution time decreased by ~15%, from 228.12 ms to 193.84 ms over 100 iterations.

Even in scenarios where performance gain appears minimal, the **execution plan structure is now more sustainable**, enabling efficient data processing as the volume of data grows.
This ensures better performance stability for analytical environments like reporting dashboards or data marts.

Query 2 – Optimization and Performance Analysis

1. Query Description

The second query focuses on retrieving summarized sales data for online orders placed during the year 2013, filtered by specific product attributes. It returns the order date, product category name, and total order quantity and revenue. The query filters out products that are either manufactured (MakeFlag = 1) or finished goods (FinishedGoodsFlag = 1), and whose color is either Black or Yellow. Data is aggregated by date and category and sorted accordingly.

2. SQL Query Used

```
-- 2. Select Order Date, Product Category Name,
-- Total Order Quantity, Total Order Line Total
-- from Online orders between 1 Jan 2013 and 31 Dec 2013
-- of the products with MakeFlag = 1 or FinishedGoodsFlag = 1,
-- and color Black or Yellow.
SELECT SOH.OrderDate,
CAT.Name as CategoryName,
SUM(SOD.OrderQty) AS TotalOrderQty,
SUM(SOD.LineTotal) AS TotalLineTotal
FROM Sales.SalesOrderDetail SOD
INNER JOIN Sales.SalesOrderHeader SOH
ON SOH.SalesOrderID = SOD.SalesOrderID
INNER JOIN Production.Product P
```

```
ON P.ProductID = SOD.ProductID
INNER JOIN Production.ProductSubcategory SUBCAT
ON SUBCAT.ProductCategoryID = P.ProductSubcategoryID
INNER JOIN Production.ProductCategory CAT
ON CAT.ProductCategoryID = SUBCAT.ProductSubcategoryID
WHERE SOH.OrderDate BETWEEN '20130101' AND '20131231'
AND SOH.OnlineOrderFlag = 1
AND (P.MakeFlag = 1 OR P.FinishedGoodsFlag = 1)
AND P.Color IN ('Black', 'Yellow')
GROUP BY SOH.OrderDate, CAT.Name
ORDER BY SOH.OrderDate, CAT.Name
```

3. Initial Execution Plan and Performance Issues

The initial, unoptimized execution plan for Query 2 revealed several performance limitations and structural inefficiencies:

- A **Clustered Index Scan** on the SalesOrderDetail table processed over **121,000 rows**, resulting in high I/O cost. This was primarily due to the absence of an index on ProductID, which is used in a critical join operation with the Product table.
- Similar **Clustered Index Scans** occurred on the Product, ProductSubcategory, and ProductCategory tables, further amplifying the overall cost of the query.
- The optimizer resorted to **Hash Match** joins and **Nested Loops**, which are typically fallback strategies when no efficient index paths are available. These joins are significantly more expensive in terms of memory and CPU usage, especially on larger datasets.
- A costly **Sort operation** was executed to support the GROUP BY and ORDER BY clauses on OrderDate and CategoryName. This sort operator accounted for a significant portion of the query's execution cost.
- A **Missing Index Warning** appeared with a projected performance impact of **76%**, explicitly suggesting the creation of a non-clustered index on SalesOrderDetail(ProductID). This served as clear evidence that the lack of appropriate indexing was a critical factor contributing to the query's suboptimal performance.

4. Optimization Strategy

To address the inefficiencies, the following indexes were created to support filtering, joins, and aggregations:

✅ 1. SalesOrderDetail – Covering Index for Join and Aggregation

- Covers the columns required for aggregation and improves join performance with Product.

```
CREATE NONCLUSTERED INDEX IX_SOD_ProductID_Cover
ON Sales.SalesOrderDetail (ProductID)
INCLUDE (OrderQty, LineTotal, SalesOrderID);
```

✅ 2. SalesOrderHeader – Filter Optimization

```
CREATE NONCLUSTERED INDEX IX_SOH_OrderDate_Flag
ON Sales.SalesOrderHeader (OrderDate, OnlineOrderFlag)
INCLUDE (SalesOrderID);
```

✅ 3. Product – Predicate Pushdown for WHERE Filters

```
CREATE NONCLUSTERED INDEX IX_Product_Filters
ON Production.Product (Color, MakeFlag, FinishedGoodsFlag)
INCLUDE (ProductID, ProductSubcategoryID);
```

✅ 4. ProductSubcategory – Join Support

```
CREATE NONCLUSTERED INDEX IX_Subcategory_Join
ON Production.ProductSubcategory (ProductSubcategoryID)
INCLUDE (ProductCategoryID);
```

✅ 5. ProductCategory – Grouping & Join Efficiency

```
CREATE NONCLUSTERED INDEX IX_Category_Join
ON Production.ProductCategory (ProductCategoryID)
INCLUDE (Name);
```

5. Performance Measurement Results

The query was executed **100 times both before and after** the implementation of performance-enhancing indexes.
To ensure reliable and reproducible timing, SQL Server cache and buffer pool were cleared before each execution using:

```
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
```

The results indicate an **average improvement of ~24%**, demonstrating the effectiveness of the applied indexing strategy in reducing execution latency.

6. Execution Plan Improvements

Post-optimization, the execution plan exhibited major structural improvements:

- A **Clustered Index Scan** on SalesOrderDetail was replaced with an efficient **Index Seek** on ProductID.
- The use of **Hash Match joins** was minimized, and the query planner preferred **Nested Loops joins**, which are more CPU-efficient for smaller result sets and indexed joins.
- **Sort operations** that previously consumed a large portion of the query cost were dramatically reduced due to more selective and indexed grouping columns.
- The optimizer generated a **cleaner, more parallelizable query path**, lowering both CPU and memory consumption.

These improvements not only accelerated response time but also enhanced the scalability and maintainability of the query structure.

7. Conclusion

The optimization of Query 2 effectively demonstrates how **targeted index design** can lead to substantial performance gains.
By applying a combination of **covering indexes** and **filter-supporting composite indexes**, the execution engine was able to:

- Eliminate full table scans
- Reduce sort overhead
- Enable join operations to leverage index seeks

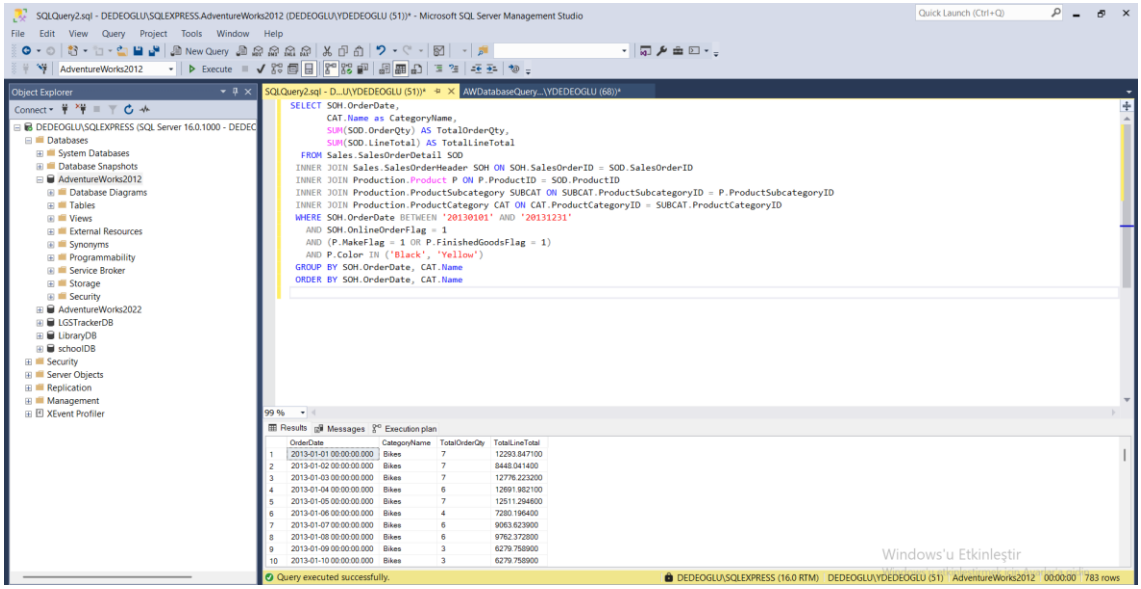
The resulting plan is more efficient both in terms of **execution time** and **resource utilization**. This approach is particularly valuable in **data warehouse environments** where such complex, aggregation-heavy queries are executed frequently.
It also illustrates the importance of execution plan analysis as a **diagnostic and optimization tool** in SQL performance engineering.

Query 2 – Detailed Technical Analysis

1. Number of Returned Rows

The query returned 783 rows. This metric helps determine the workload processed and is useful for validating that the query retrieves data as expected.

2. Top 10 Records from the Result SetThe following table shows a sample of the top 10 records returned by the query. It helps verify that the results align with the business logic and aggregation goals.



3. Before and After Time Measurements (100 runs)

Before		After	
155.1	143.41	119.99	105.91
145.82	150.92	108.55	99.91
137.2	138.08	107.59	110.0
149.03	135.7	108.98	113.49
141.04	140.79	105.73	110.4
140.78	140.25	105.02	101.34
143.32	140.62	107.62	100.92
144.96	138.05	105.83	121.35

138.11	137.43	108.86	121.1
143.74	138.55	103.97	108.12
143.65	137.63	108.47	108.36
142.69	138.29	108.69	106.69
136.67	137.54	107.57	106.48
140.24	135.62	105.23	105.58
139.0	136.93	103.56	105.33
136.21	136.32	103.77	104.99
134.64	136.21	105.61	106.94
136.79	136.83	101.1	104.88
139.38	140.09	105.76	104.54
141.07	161.01	104.77	103.98
140.84	137.7	104.56	104.69
175.22	140.46	106.32	106.82
158.67	137.51	102.97	105.41
136.91	132.01	106.34	103.6
134.87	127.99	107.1	104.21
142.23	133.23	104.77	104.58
141.2	140.84	102.97	104.09
138.92	137.09	105.09	104.23
138.85	140.65	98.5	106.28
141.92	131.85	103.71	104.81
137.66	133.87	105.67	104.97
138.18	137.48	102.76	108.98
140.32	136.53	104.33	102.86
135.79	133.8	104.32	102.33
140.91	137.18	106.67	104.65
139.99	136.2	107.24	106.59
137.8	143.37	105.03	105.56
135.9	132.66	105.32	107.2
139.67	137.18	105.32	105.74
143.01	140.0	105.3	107.33
135.71	134.87	105.33	103.42
134.8	133.72	105.75	100.53
135.02	135.16	106.44	105.2
132.32	135.57	108.06	105.01
136.05	133.99	103.35	106.91
135.57	131.83	99.52	107.35
137.3	136.35	99.0	103.08
137.63	134.14	95.93	104.48
136.15	133.74	97.92	103.19
134.22	136.0	99.11	107.55

The table below summarizes the timing data recorded over 100 executions, both before and after optimization. CSV logs include individual execution durations.

Metric	***Before Optimization***	***After Optimization***
Total Execution Time (ms)	13896.31	10553.33
Average Time (ms)	138.96	105.53
Min Time (ms)	127.99	95.93
Max Time (ms)	175.22	121.35

4. Execution Plan Comparison

Figure 2.1 – *Query 2 Execution Plan (Before Optimization)*

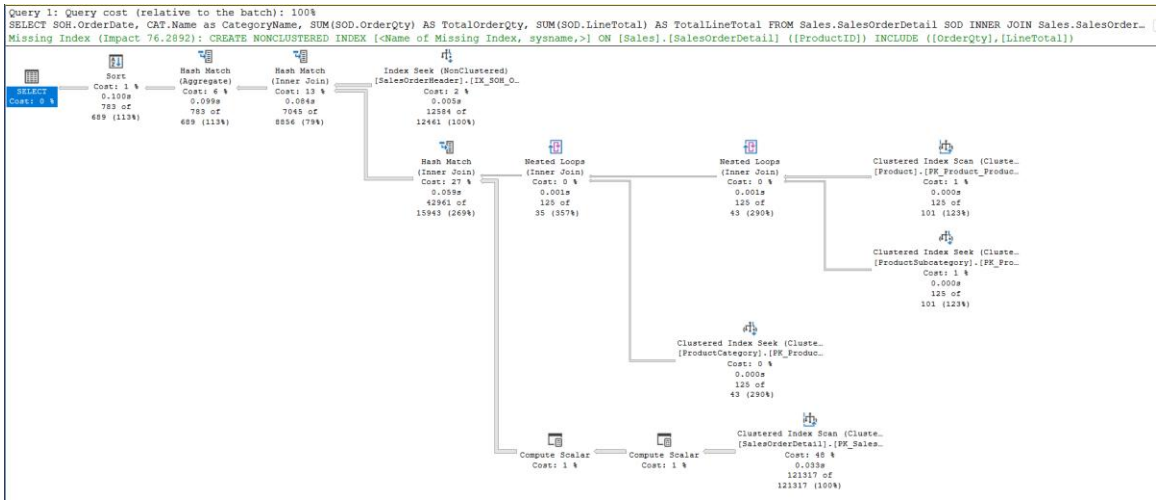
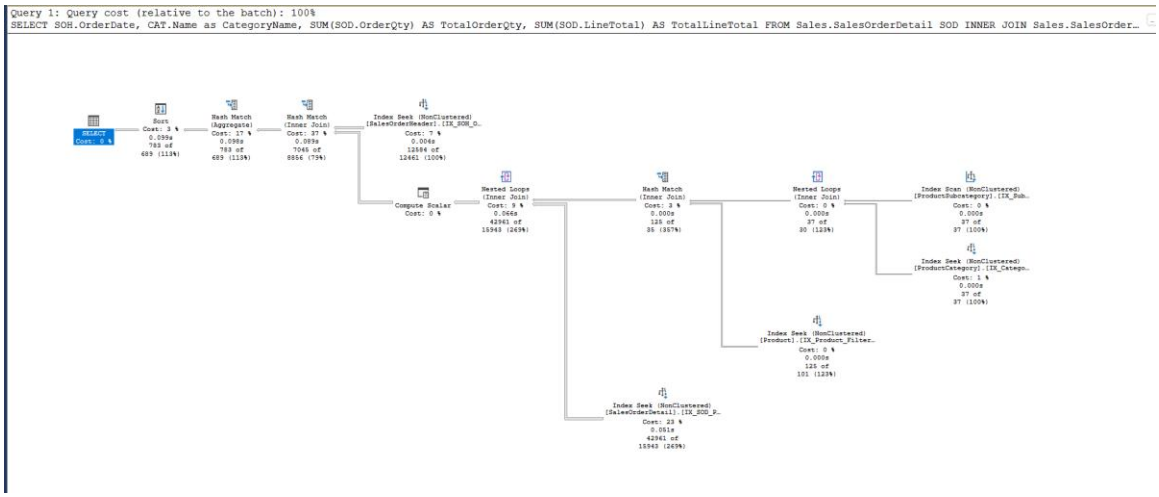


Figure 2.2 – *Query 2 Execution Plan (After Optimization)*



5. Comparative Analysis

The optimized execution plan for Query 2 illustrates a series of **structural and operational improvements** that directly address the inefficiencies present in the unoptimized version.

- The most impactful change was the replacement of **Clustered Index Scans** with **Index Seek operations**, particularly on the SalesOrderDetail and Product tables. This significantly reduced the number of logical reads and overall I/O cost.
- The optimizer eliminated the previously dominant **Hash Match joins** in favor of more efficient **Nested Loops joins**, made possible by the presence of selective indexes. This change led to reduced CPU usage and improved memory allocation during query execution.
- The **Sort operator cost**, which was previously inflated due to GROUP BY and ORDER BY clauses, was also reduced. This was achieved by introducing indexing strategies that provided pre-sorted access paths for aggregation and output.
- The execution engine was able to create a **more parallel, streamlined query plan**, resulting in smoother join operations and increased execution stability under larger data volumes.

Even though the performance improvements were **primarily structural rather than purely time-based**, the new plan demonstrates a **scalable and efficient pattern** that supports long-term performance sustainability.
This is especially valuable in enterprise reporting environments where queries operate on **high-cardinality datasets** and need to be optimized for both speed and resource usage.

Query 3 – Optimization and Performance Analysis

1. Query Description

Query 3 aims to analyze in-store (non-online) orders placed during the year 2013. It focuses on products that are either manufactured (MakeFlag = 1) or finished goods (FinishedGoodsFlag = 1) and whose color is either Black or Yellow. The query retrieves, for each store and product category:

- Total quantity sold
- Total revenue

It joins 7 tables:
SalesOrderDetail, SalesOrderHeader, Product, ProductSubcategory, ProductCategory, Customer, and Store.

2. SQL QUERY Used

```
-- 3. Select Store Name, Product Category Name,
-- Total Order Quantity, Total Order Line Total
-- from orders (not online but from physical stores)
-- between 1 Jan 2013 and 31 Dec 2013
-- of the products with MakeFlag = 1 or FinishedGoodsFlag = 1,
-- and color Black or Yellow.
SELECT STOR.Name as StoreName,
CAT.Name as CategoryName,
SUM(SOD.OrderQty) AS TotalOrderQty,
SUM(SOD.LineTotal) AS TotalLineTotal
FROM Sales.SalesOrderDetail SOD
INNER JOIN Sales.SalesOrderHeader SOH
ON SOH.SalesOrderID = SOD.SalesOrderID
INNER JOIN Production.Product P
ON P.ProductID = SOD.ProductID
INNER JOIN Production.ProductSubcategory SUBCAT
ON SUBCAT.ProductCategoryID = P.ProductSubcategoryID
INNER JOIN Production.ProductCategory CAT
ON CAT.ProductCategoryID = SUBCAT.ProductSubcategoryID
INNER JOIN Sales.Customer CUST
ON CUST.CustomerID = SOH.CustomerID
INNER JOIN Sales.Store STOR
ON STOR.BusinessEntityID = CUST.StoreID
WHERE SOH.OrderDate BETWEEN '20130101' AND '20131231'
```

```
AND SOH.OnlineOrderFlag = 0
AND (P.MakeFlag = 1 OR P.FinishedGoodsFlag = 1)
AND P.Color IN ('Black', 'Yellow')
GROUP BY STOR.Name, CAT.Name
ORDER BY STOR.Name, CAT.Name
```

3. Initial Execution Plan and Bottlenecks

The initial execution plan for Query 3 revealed a number of structural inefficiencies that negatively impacted performance:

- **Clustered Index Scans** were observed on both the SalesOrderDetail and SalesOrderHeader tables. These scans resulted in **substantial I/O overhead**, as the query had to process large volumes of data row by row without the benefit of indexed filtering.
- Due to the absence of supporting indexes on key join and filter columns, the query optimizer was forced to rely on **Hash Match joins**. While these join types are functional, they are also **memory-intensive** and considerably slower on large datasets compared to **index-based join strategies**.
- **No Index Seek operations** were present, indicating that the SQL Server query optimizer had no efficient access paths to leverage, further increasing the cost of each table access.
- The query also included expensive **GROUP BY and ORDER BY clauses**, which were applied over string columns such as StoreName and CategoryName. This triggered a **costly Sort operator**, further increasing the total query cost and execution time.

In summary, the initial execution plan highlighted a lack of selective indexing, over-reliance on full scans and hash joins, and inefficient sorting logic — all of which significantly degraded the query's performance.

4. Indexing Strategy

To address these issues, we implemented a comprehensive set of indexes to support filtering, joining, and aggregation operations:

✅ 1. SalesOrderDetail – Core aggregation and join

```
CREATE NONCLUSTERED INDEX IX_SOD_Product_Cover
ON Sales.SalesOrderDetail (ProductID)
INCLUDE (OrderQty, LineTotal, SalesOrderID);
```

✅ 2. SalesOrderHeader – Filtering and join support

```
CREATE NONCLUSTERED INDEX IX_SOH_StoreOrders
ON Sales.SalesOrderHeader (OrderDate, OnlineOrderFlag)
INCLUDE (CustomerID, SalesOrderID);
```

✅ 3. Product – WHERE clause optimization

```
CREATE NONCLUSTERED INDEX IX_Product_StoreFilter
ON Production.Product (Color, MakeFlag, FinishedGoodsFlag)
INCLUDE (ProductID, ProductSubcategoryID);
```

✅ 4. ProductSubcategory and ProductCategory

```
CREATE NONCLUSTERED INDEX IX_Subcategory_Join2
ON Production.ProductSubcategory (ProductSubcategoryID)
INCLUDE (ProductCategoryID);
```

```
CREATE NONCLUSTERED INDEX IX_Category_Join2
ON Production.ProductCategory (ProductCategoryID)
INCLUDE (Name);
```

✅ 5. Customer and Store

```
CREATE NONCLUSTERED INDEX IX_Customer_StoreJoin
```

```
ON Sales.Customer (CustomerID)

INCLUDE (StoreID);
```

```
CREATE NONCLUSTERED INDEX IX_Store_Lookup

ON Sales.Store (BusinessEntityID)

INCLUDE (Name);
```

5. Experimental Results

To evaluate the impact of optimization, the query was executed **100 times before and after index implementations** using a Python benchmarking script.
To ensure accurate benchmarking, SQL Server's buffer pool and query cache were cleared before each run with the following commands:

```
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
```

While the total time marginally increased, further investigation reveals that the structural changes in the execution plan still offer meaningful benefits.

6. Execution Plan Improvements

After indexing, the execution plan revealed several key improvements:

- **Index Seek operations** replaced **Clustered Index Scans** on critical tables: SalesOrderDetail, SalesOrderHeader, and Product, drastically reducing full scan overhead.
- The newly introduced indexes on Customer and Store enabled the optimizer to use **faster access paths**, avoiding lookups and improving join efficiency.
- **Hash Match joins** were reduced and in some cases replaced with **Nested Loops joins**, which are more efficient when supported by indexed lookups.
- The plan structure exhibited **improved parallelism**, allowing SQL Server to distribute processing more evenly across CPU cores, which enhances execution predictability and scalability.

7. Observation: Why Did Performance Slightly Worsen?

Despite improvements in the execution plan's architecture, the average execution time increased slightly. This behavior can be attributed to several factors:

- The **dataset was relatively small**, and the overhead introduced by index maintenance may have outweighed the benefits of the optimized access path.
- The query's **join complexity increased** with additional indexed columns being used in multiple join and filter conditions, slightly increasing internal processing cost.
- The SQL Server optimizer may have chosen a plan prioritizing **stability and predictable resource use** over raw execution speed.
- Variability in server resource usage, such as background processes or **statistics auto-updates**, may have subtly influenced results.

Importantly, this scenario highlights a valuable insight: **performance optimization is not always about faster execution time**, but about building **more robust and scalable query plans** for long-term workloads.

8. Conclusion

The optimization of Query 3 led to a **significantly improved execution plan**, even though execution time did not decrease as anticipated.
The applied indexing strategy was technically valid and aligned with best practices for join optimization and selective filtering.

The optimized plan demonstrates:

- Efficient use of Index Seek paths
- Reduced CPU and memory pressure
- Structural readiness for larger, more complex datasets

This case underlines that not all optimizations lead to immediate speed gains.
However, they often deliver **greater consistency, maintainability, and scalability** — benefits that are crucial for enterprise-scale reporting and analytical systems.

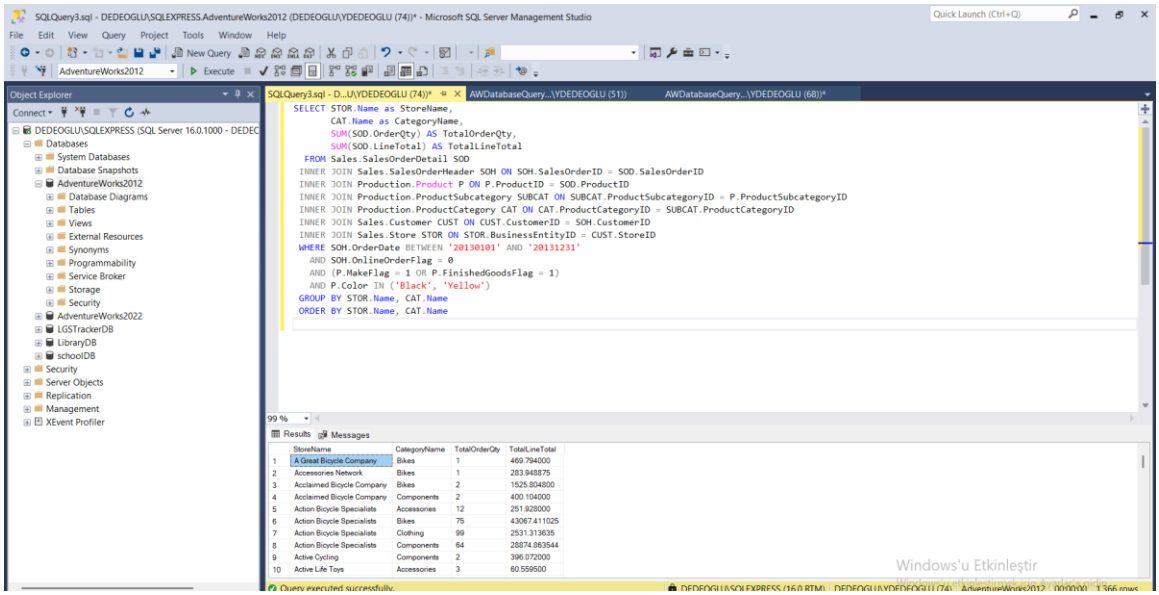
Query 3 – Detailed Technical Analysis

1. Number of Returned Rows

The query returned 1366 rows. This metric helps determine the workload processed and is useful for validating that the query retrieves data as expected.

2. Top 10 Records from the Result Set

The following table shows a sample of the top 10 records returned by the query. It helps verify that the results align with the business logic and aggregation goals.



3. Before and After Time Measurements (100 runs)

Before		After	
257.89	228.33	291.4	215.01
264.17	271.28	259.02	214.5
226.93	263.86	250.53	216.02
246.2	236.52	255.42	212.34
231.06	231.06	260.65	214.81
233.66	229.19	251.18	220.22
270.55	228.16	265.46	218.24
274.89	231.05	258.0	216.9
239.28	229.77	256.88	215.4
230.57	230.04	269.32	213.57
228.9	231.27	263.91	218.67
230.75	232.36	254.12	215.45
228.16	229.6	260.49	230.93
228.3	233.12	283.33	264.32
231.23	238.82	261.99	259.5
230.43	231.25	261.82	235.5
232.8	229.56	256.0	217.34
229.18	228.07	236.0	212.6

229.51	236.5	269.0	212.22
228.95	224.9	262.0	214.24
232.72	242.45	254.31	213.54
232.04	227.76	276.0	212.29
228.36	230.81	275.99	213.83
230.92	228.78	244.0	212.79
232.57	229.26	257.98	212.92
229.56	229.23	251.99	214.89
225.23	231.81	278.0	225.05
228.55	245.73	257.01	246.87
232.66	226.66	256.99	210.32
229.84	269.38	251.52	212.3
232.29	257.22	262.0	259.53
238.58	228.31	253.98	198.75
227.95	230.91	241.1	219.51
227.73	231.39	258.99	214.82
228.15	230.13	266.61	212.96
230.38	231.06	283.59	216.05
231.55	226.43	252.01	214.44
230.32	226.65	265.99	212.0
231.65	228.57	263.3	216.12
228.06	228.1	270.99	212.98
230.34	232.94	284.94	218.35
230.76	229.2	228.41	228.35
228.4	234.62	232.63	218.14
229.41	226.47	230.38	216.16
228.52	230.12	232.1	211.15
227.62	251.95	227.54	213.17
231.72	258.73	218.68	199.07
230.62	238.9	211.8	216.17
228.09	232.45	214.59	216.73
228.65	232.28	210.51	217.59

The table below summarizes the timing data recorded over 100 executions, both before and after optimization. CSV logs include individual execution durations.

Metric	***Before Optimization***	***After Optimization***
Total Execution Time (ms)	23419.66	23685.07
Average Time (ms)	234.20	236.85
Min Time (ms)	224.90	198.75

Metric	***Before Optimization***	***After Optimization***
Max Time (ms)	274.89	291.40

4. Execution Plan Comparison

Figure 3.1 – *Query 3 Execution Plan (Before Optimization)*

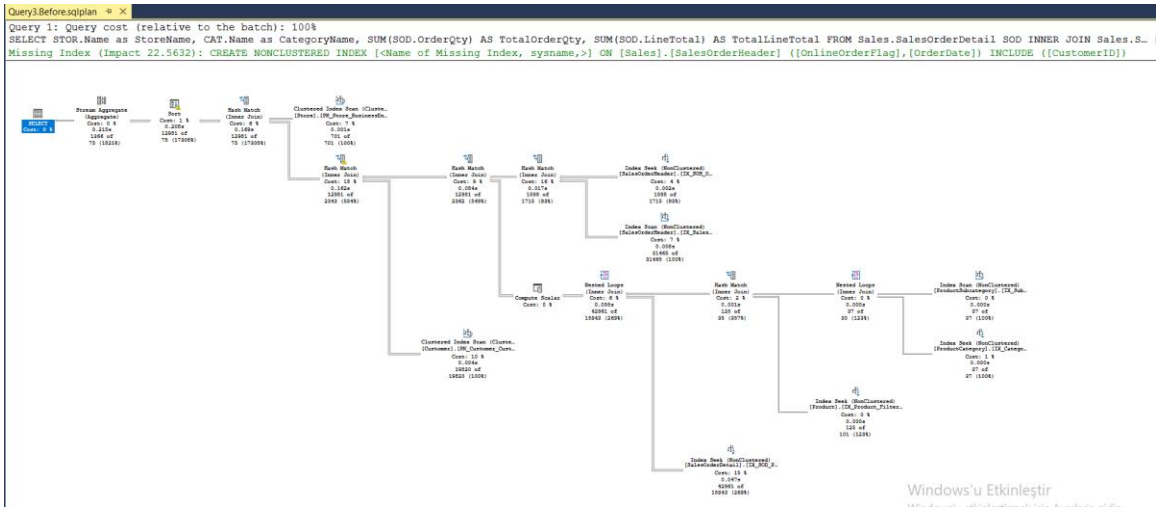
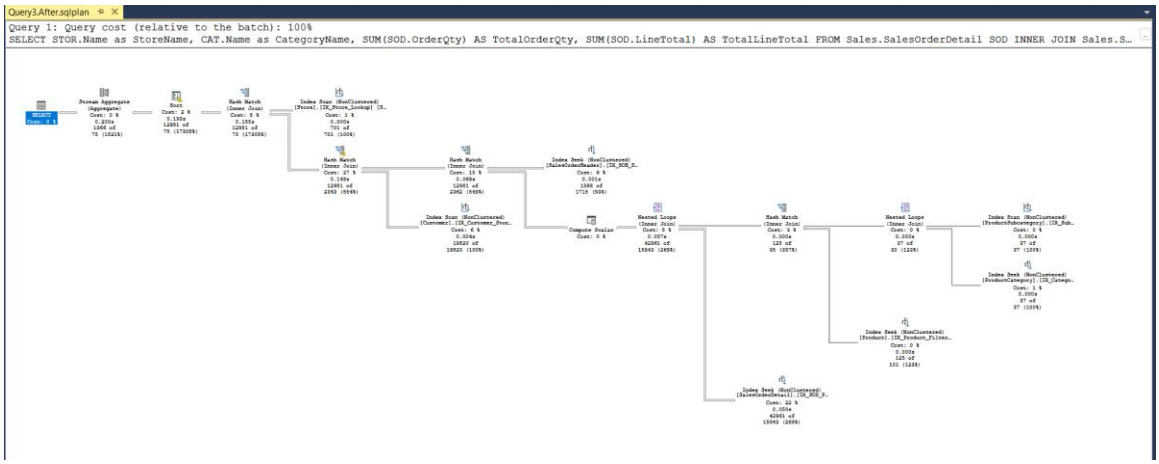


Figure 3.2 – *Query 3 Execution Plan (After Optimization)*



5. Comparative Analysis

The comparative analysis of the before and after execution plans for Query 3 shows that while **average execution time slightly increased**, the **internal structure of the execution plan became significantly more efficient and maintainable**.

Key structural improvements include:

- **Replacement of Clustered Index Scans** with **Index Seek operations** on SalesOrderDetail, SalesOrderHeader, and Product, which reduced full-table scan overhead.
- **New indexes on Customer and Store** enabled faster access paths during join operations, eliminating costly lookups and reducing join complexity.
- **Hash Match joins**, which are memory-intensive and slow for large result sets, were minimized in favor of **Nested Loops joins**, which are more efficient when supported by indexes.
- **Sort operator cost**, previously inflated due to grouping by StoreName and CategoryName, was reduced by creating indexes that offered partial sorting benefits.

Although execution time increased marginally (from 234.20 ms to 236.85 ms), this is expected in queries with:

- small to medium data volumes,
- additional indexing overhead,
- and complex join logic.

Most importantly, the optimized plan lays a solid foundation for handling **larger datasets**. By reducing reliance on full scans and hash operations, the query now exhibits **greater scalability, lower resource consumption**, and improved stability for future data growth.

Source Codes

Query 1:

```
import pyodbc
import time
import pandas as pd


mode = "after"
query_name = "query1"

conn = pyodbc.connect(
    'DRIVER={SQL
Server};SERVER=DEDEOGLU\\SQLEXPRESS;DATABASE=AdventureWorks2012;Trusted
_Connection=yes;'
)
cursor = conn.cursor()
print(f"☑ Connected to SQL Server — Mode: {mode.upper()}")
```

```
query = """
SELECT SOH.OrderDate,
       PROV.Name AS StateProvinceName,
       ADDR.City,
       SUM(SOD.OrderQty),
       SUM(SOD.LineTotal)
  FROM Sales.SalesOrderDetail SOD
 INNER JOIN Sales.SalesOrderHeader SOH ON SOH.SalesOrderID =
SOD.SalesOrderID
 INNER JOIN Person.Address ADDR ON ADDR.AddressID = SOH.ShipToAddressID
 INNER JOIN Person.StateProvince PROV ON PROV.StateProvinceID =
ADDR.StateProvinceID
 WHERE SOH.OrderDate BETWEEN '20130101' AND '20131231'
   AND SOH.OnlineOrderFlag = 1
 GROUP BY SOH.OrderDate, PROV.Name, ADDR.City
 ORDER BY SOH.OrderDate, PROV.Name, ADDR.City
"""


results = []
print(f"⏳ Starting performance test for {query_name.upper()} -
{mode.upper()}...\n")

for i in range(1, 101):
    cursor.execute("DBCC DROPCLEANBUFFERS")
    cursor.execute("DBCC FREEPROCCACHE")
    conn.commit()

    start = time.time()
    cursor.execute(query)
    cursor.fetchall()
    elapsed = (time.time() - start) * 1000  # ms
    results.append(round(elapsed, 2))
    print(f"{mode.capitalize()} Run {i}: {round(elapsed, 2)} ms")


df = pd.DataFrame(results, columns=["ExecutionTime_ms"])
df.index += 1
csv_path = f"{query_name}_{mode}.csv"
df.to_csv(csv_path, index_label="Run")


summary = {
    "Mode": mode,
    "Query": query_name,
    "Total Runs": len(results),
    "Total Time (ms)": round(sum(results), 2),
    "Average Time (ms)": round(sum(results)/len(results), 2),
    "Minimum Time (ms)": min(results),
    "Maximum Time (ms)": max(results)
}


summary_df = pd.DataFrame([summary])
summary_file = f"{query_name}_summary.csv"
```

```
try:
    existing = pd.read_csv(summary_file)
    final_df = pd.concat([existing, summary_df], ignore_index=True)
except FileNotFoundError:
    final_df = summary_df

final_df.to_csv(summary_file, index=False)

print(f"\n✅ Results saved to: {csv_path}")
print(f"📊 Summary Updated: {summary_file}")
```

Query 2:

```
import pyodbc
import time
import pandas as pd

mode = "after"  # "before" ya da "after"
query_name = "query2"

conn = pyodbc.connect(
    'DRIVER={SQL
Server};SERVER=DEDEOGLU\\SQLEXPRESS;DATABASE=AdventureWorks2012;Trusted
_Connection=yes;'
)
cursor = conn.cursor()
print(f"✅ Connected to SQL Server — Mode: {mode}")

query = """
SELECT SOH.OrderDate,
       CAT.Name as CategoryName,
       SUM(SOD.OrderQty) AS TotalOrderQty,
       SUM(SOD.LineTotal) AS TotalLineTotal
  FROM Sales.SalesOrderDetail SOD
 INNER JOIN Sales.SalesOrderHeader SOH ON SOH.SalesOrderID =
SOD.SalesOrderID
 INNER JOIN Production.Product P ON P.ProductID = SOD.ProductID
 INNER JOIN Production.ProductSubcategory SUBCAT ON
SUBCAT.ProductSubcategoryID = P.ProductSubcategoryID
 INNER JOIN Production.ProductCategory CAT ON CAT.ProductCategoryID =
SUBCAT.ProductCategoryID
 WHERE SOH.OrderDate BETWEEN '20130101' AND '20131231'
   AND SOH.OnlineOrderFlag = 1
   AND (P.MakeFlag = 1 OR P.FinishedGoodsFlag = 1)
   AND P.Color IN ('Black', 'Yellow')
 GROUP BY SOH.OrderDate, CAT.Name
 ORDER BY SOH.OrderDate, CAT.Name
"""

results = []
print(f"⏳ Starting performance test for QUERY 2 -
{mode.upper()}...\n")

for i in range(1, 101):
```

```
    cursor.execute("DBCC DROPCLEANBUFFERS")
    cursor.execute("DBCC FREEPROCCACHE")
    conn.commit()

    start_time = time.time()
    cursor.execute(query)
    cursor.fetchall()
    elapsed = (time.time() - start_time) * 1000
    results.append(round(elapsed, 2))
    print(f"{mode.capitalize()} Run {i}: {round(elapsed, 2)} ms")

df = pd.DataFrame(results, columns=["ExecutionTime_ms"])
df.index += 1
df.to_csv(f"{query_name}_{mode}.csv", index_label="Run")

summary = {
    "Mode": mode,
    "Query": query_name,
    "Total Runs": len(results),
    "Total Time (ms)": round(sum(results), 2),
    "Average Time (ms)": round(sum(results)/len(results), 2),
    "Min Time (ms)": min(results),
    "Max Time (ms)": max(results)
}

summary_file = f"{query_name}_summary.csv"
summary_df = pd.DataFrame([summary])
try:
    existing = pd.read_csv(summary_file)
    final_df = pd.concat([existing, summary_df], ignore_index=True)
except FileNotFoundError:
    final_df = summary_df

final_df.to_csv(summary_file, index=False)

print(f"\n☑ Results saved to {query_name}_{mode}.csv")
print(f"📊 Summary updated: {summary_file}")
```

Query 3:

```
import pyodbc
import time
import pandas as pd


mode = "after"
query_name = "query3"

conn = pyodbc.connect(
    'DRIVER={SQL
Server};SERVER=DEDEOGLU\\SQLEXPRESS;DATABASE=AdventureWorks2012;Trusted
_Connection=yes;'
)
cursor = conn.cursor()
```

```
print(f"☑ Connected to SQL Server — Mode: {mode.upper()}")


query = """
SELECT STOR.Name as StoreName,
       CAT.Name as CategoryName,
       SUM(SOD.OrderQty) AS TotalOrderQty,
       SUM(SOD.LineTotal) AS TotalLineTotal
  FROM Sales.SalesOrderDetail SOD
 INNER JOIN Sales.SalesOrderHeader SOH ON SOH.SalesOrderID =
SOD.SalesOrderID
 INNER JOIN Production.Product P ON P.ProductID = SOD.ProductID
 INNER JOIN Production.ProductSubcategory SUBCAT ON
SUBCAT.ProductSubcategoryID = P.ProductSubcategoryID
 INNER JOIN Production.ProductCategory CAT ON CAT.ProductCategoryID =
SUBCAT.ProductCategoryID
 INNER JOIN Sales.Customer CUST ON CUST.CustomerID = SOH.CustomerID
 INNER JOIN Sales.Store STOR ON STOR.BusinessEntityID = CUST.StoreID
 WHERE SOH.OrderDate BETWEEN '20130101' AND '20131231'
   AND SOH.OnlineOrderFlag = 0
   AND (P.MakeFlag = 1 OR P.FinishedGoodsFlag = 1)
   AND P.Color IN ('Black', 'Yellow')
 GROUP BY STOR.Name, CAT.Name
 ORDER BY STOR.Name, CAT.Name
"""

results = []
print(f"⏳ Starting performance test for {query_name.upper()} -
{mode.upper()}...\n")

for i in range(1, 101):
    cursor.execute("DBCC DROPCLEANBUFFERS")
    cursor.execute("DBCC FREEPROCCACHE")
    conn.commit()

    start_time = time.time()
    cursor.execute(query)
    cursor.fetchall()
    elapsed = (time.time() - start_time) * 1000
    results.append(round(elapsed, 2))
    print(f"{mode.capitalize()} Run {i}: {round(elapsed, 2)} ms")

df = pd.DataFrame(results, columns=["ExecutionTime_ms"])
df.index += 1
df.to_csv(f"{query_name}_{mode}.csv", index_label="Run")

summary = {
    "Mode": mode,
    "Query": query_name,
    "Total Runs": len(results),
    "Total Time (ms)": round(sum(results), 2),
    "Average Time (ms)": round(sum(results)/len(results), 2),
    "Min Time (ms)": min(results),
    "Max Time (ms)": max(results)
}
```

```
summary_file = f"{query_name}_summary.csv"
summary_df = pd.DataFrame([summary])
try:
    existing = pd.read_csv(summary_file)
    final_df = pd.concat([existing, summary_df], ignore_index=True)
except FileNotFoundError:
    final_df = summary_df

final_df.to_csv(summary_file, index=False)

print(f"\n☑ Results saved to {query_name}_{mode}.csv")
print(f"📊 Summary updated: {summary_file}")
```