Seminar 5

Performance Tuning in SQL Server

Query Tuning Methodology

- identify waits (bottleneck) at the server level
    - locks
    - transaction log
    - I/O
    - etc.
- correlate waits with queues
- drill down to database / file level
- drill down to process level
- tune problematic queries

* DMVs - dynamic management views

Identify Waits

- **`sys.dm_os_wait_stats`**
  - returned table
    - wait_type
      - resource waits (locks, latches, network, I/O), queue waits, external waits
    - waiting_tasks_count
    - wait_time_ms
    - max_wait_time_ms
    - signal_wait_time_ms
- reset DMV values
  - **`DBCC SQLPERF ('sys.dm_os_wait_stats', CLEAR);`**

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
  - ***object_name*** - the category of the counter
  - ***counter_name*** - the name of the counter
  - ***instance_name*** - the name of the specific instance of the counter; often contains the name of the database
  - ***cntr_value*** - the current value of the counter
  - ***cntr_type*** - the type of the counter (as defined by the Windows performance architecture)

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- > 500 counters: Access Methods, User Settable, Buffer Manager, Broker Statistics, SQL Errors, Latches, Buffer Partition, SQL Statistics, Locks, Buffer Node, Plan Cache, Cursor Manager by Type, Memory Manager, General Statistics, Databases, Catalog Metadata, Broker Activation, Broker/DBM Transport, Transactions, Cursor Manager Total, Exec Statistics, Wait Statistics, etc.
- ***cntr_type*** = 65792 → ***cntr_value*** contains the actual value

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- ***cntr_type*** = 537003264 → ***cntr_value*** contains real-time results, which are divided by a "base" to obtain the actual value; by themselves, they are useless
  - to get a ratio: divide by a "base" value
  - to get a percentage: multiply the result by 100.0

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- ***cntr_type*** = 272696576
  - time-based, cumulative counters
  - a secondary table can be used to log intermediate values

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- ***cntr_type*** = 1073874176 and ***cntr_type*** = 1073939712 → poll both the value (1073874176) and the base value (1073939712)
- poll both values again (e.g., after 15 seconds) ☺
- to obtain the desired result, compute:

$$\text{UnitsPerSec} = (cv2 - cv1) \,/\, (bv2 - bv1) \,/\, 15.0$$

Drill Down to Database / File Level

- **`sys.dm_io_virtual_file_stats`**
  - returns I/O information about *data files* and *log files*
- parameters
  - database_ID
    - NULL = all databases
    - useful function: DB_ID
  - file_ID
    - NULL = all files
    - useful function: FILE_IDEX

Drill Down to Database / File Level

- **`sys.dm_io_virtual_file_stats`**
  - returned table
    - database_ID
    - file_ID
    - sample_ms - # of milliseconds since the computer was started
    - num_of_reads - number of reads issued on the file
    - num_of_bytes_read - number of bytes read on the file
    - io_stall_read_ms – total time users waited for reads issued on the file

Drill Down to Database / File Level

- **`sys.dm_io_virtual_file_stats`**
  - returned table
    - num_of_writes - number of writes
    - num_of_bytes_written - total number of bytes written to the file
    - io_stall_write_ms - total time users waited for writes to be completed on the file
    - io_stall - total time users waited for the completion of I/O operations (ms)
    - file_handle

Drill Down to the Process Level

- a filter on duration / I/O only isolates individual processes (batch / proc / query)
- aggregate performance information by query pattern
  - patterns can be easily identified when using stored procedures
  - when one doesn't use stored procedures:
    - quick and dirty approach: LEFT(query string, n)
    - use a parser to identify the query pattern

Indexes

- one of the major factors influencing query performance
  - impact on: filtering, joins, sorting, grouping; blocking and deadlock avoidance, etc.
  - effect on modifications: <u>positive</u> effect (locating the rows); <u>negative</u> effect (cost of modifying the index)
- understanding indexes and their internal mechanisms
  - clustered/nonclustered, single/multicolumn, indexed views, indexes on computed columns, covering scenarios, intersection

Indexes

- one should carefully judge whether additional index maintenance costs are justified by improvements in query performance
  - take into account the environment and the ratio between SELECT queries and data modifications
- *multicolumn indexes*
  - tend to be more useful than single-column indexes
  - the query optimizer is more likely to use such indexes to cover a query

Indexes

- *indexed views* come with a higher maintenance cost than standard indexes
  - mandatory option
    - WITH SCHEMABINDING

Tools to Analyze Query Performance

- graphical execution plan
- STATISTICS IO - scan count, logical reads, physical reads, *read-ahead* reads
- STATISTICS TIME - duration and net CPU time
- SHOWPLAN_TEXT – SQL Server returns detailed information about how the statements are executed
- SHOWPLAN_ALL - SQL Server returns detailed information about how the statements are executed, provides estimates of the resource requirements

Tools to Analyze Query Performance

- STATISTICS PROFILE - a profile of the query execution
- STATISTICS XML – actual plan information in XML format
- SHOWPLAN_XML - estimated plan information in XML format

Query Optimization

- evaluating execution plans
  - sequences of physical/logical operations
- optimization - factors
  - search predicate
  - tables involved in joins
  - join conditions
  - result set size
  - list of indexes
- goal - avoid worst query plans
- SQL Server uses a *cost-based* query optimizer

STATISTICS IO and STATISTICS TIME

EXPRESSIA-1\E...SQLQuery1.sql* | Object Explorer Details

```sql
USE Adventureworks;
GO
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.Contact;
```

Results | Messages

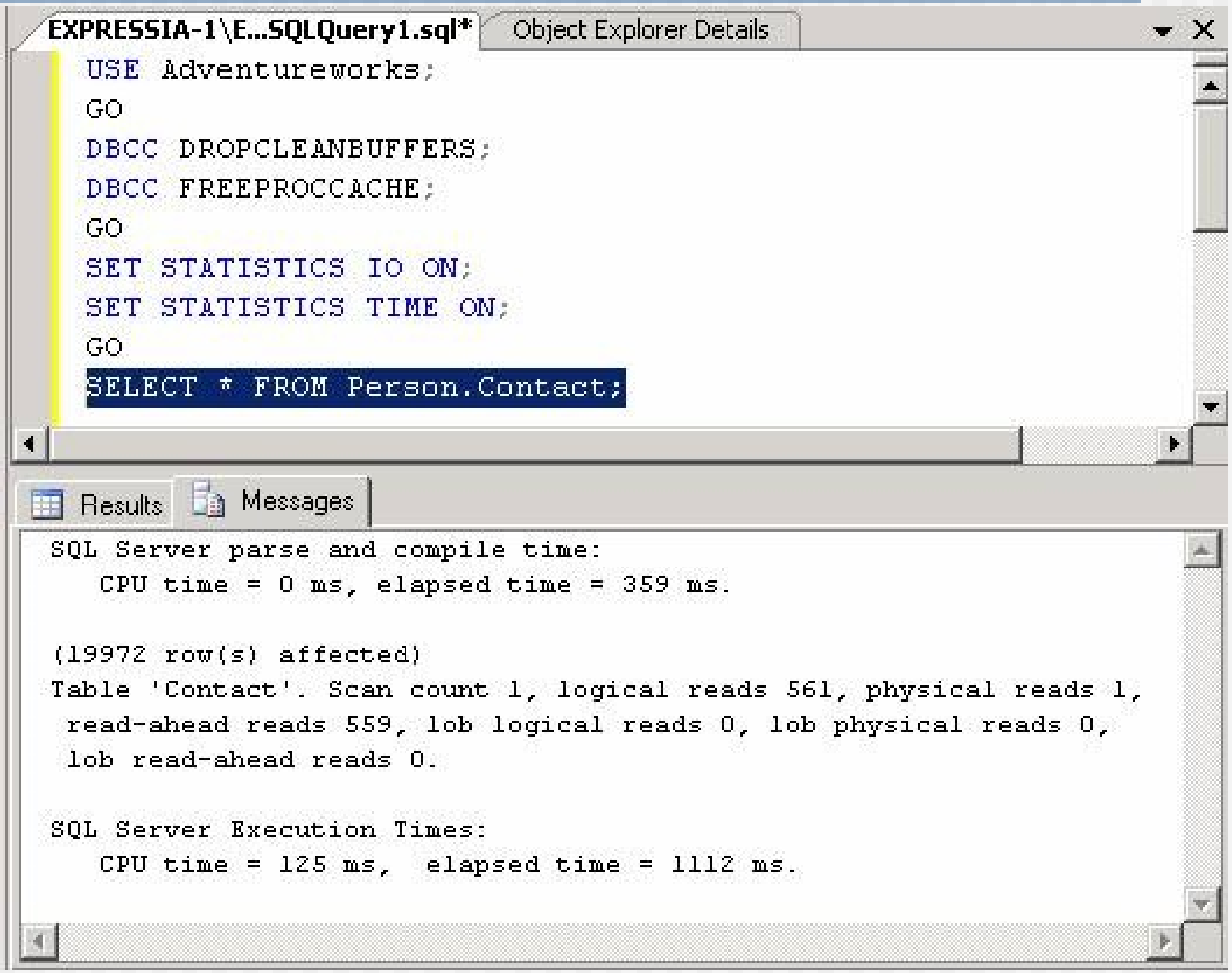
```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 359 ms.

(19972 row(s) affected)
Table 'Contact'. Scan count 1, logical reads 561, physical reads 1,
 read-ahead reads 559, lob logical reads 0, lob physical reads 0,
 lob read-ahead reads 0.

SQL Server Execution Times:
   CPU time = 125 ms,  elapsed time = 1112 ms.
```

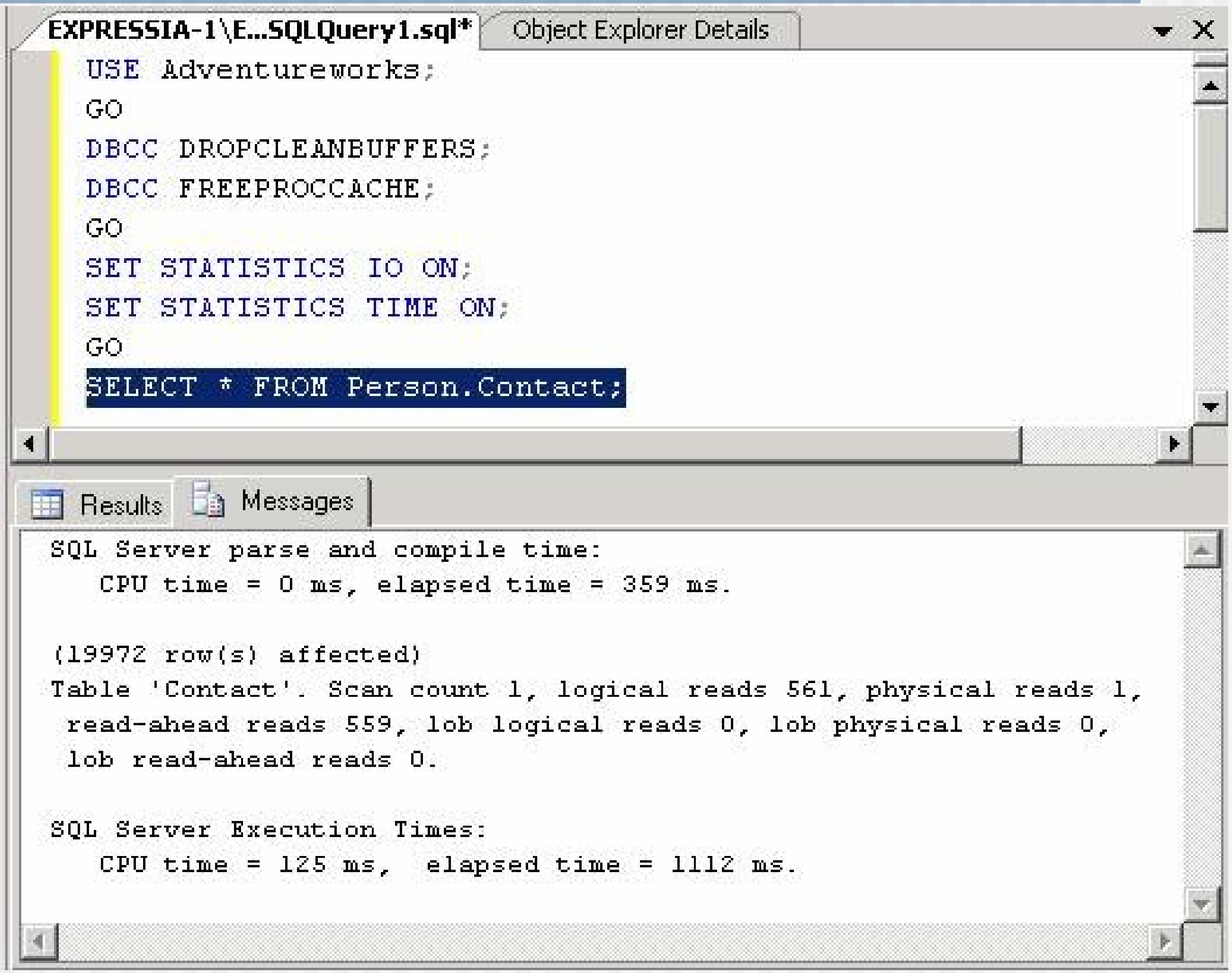
- DBCC DROPCLEANBUFFERS – clear data from the cache
- DBCC FREEPROCCACHE – clear execution plans from the cache

STATISTICS IO and STATISTICS TIME



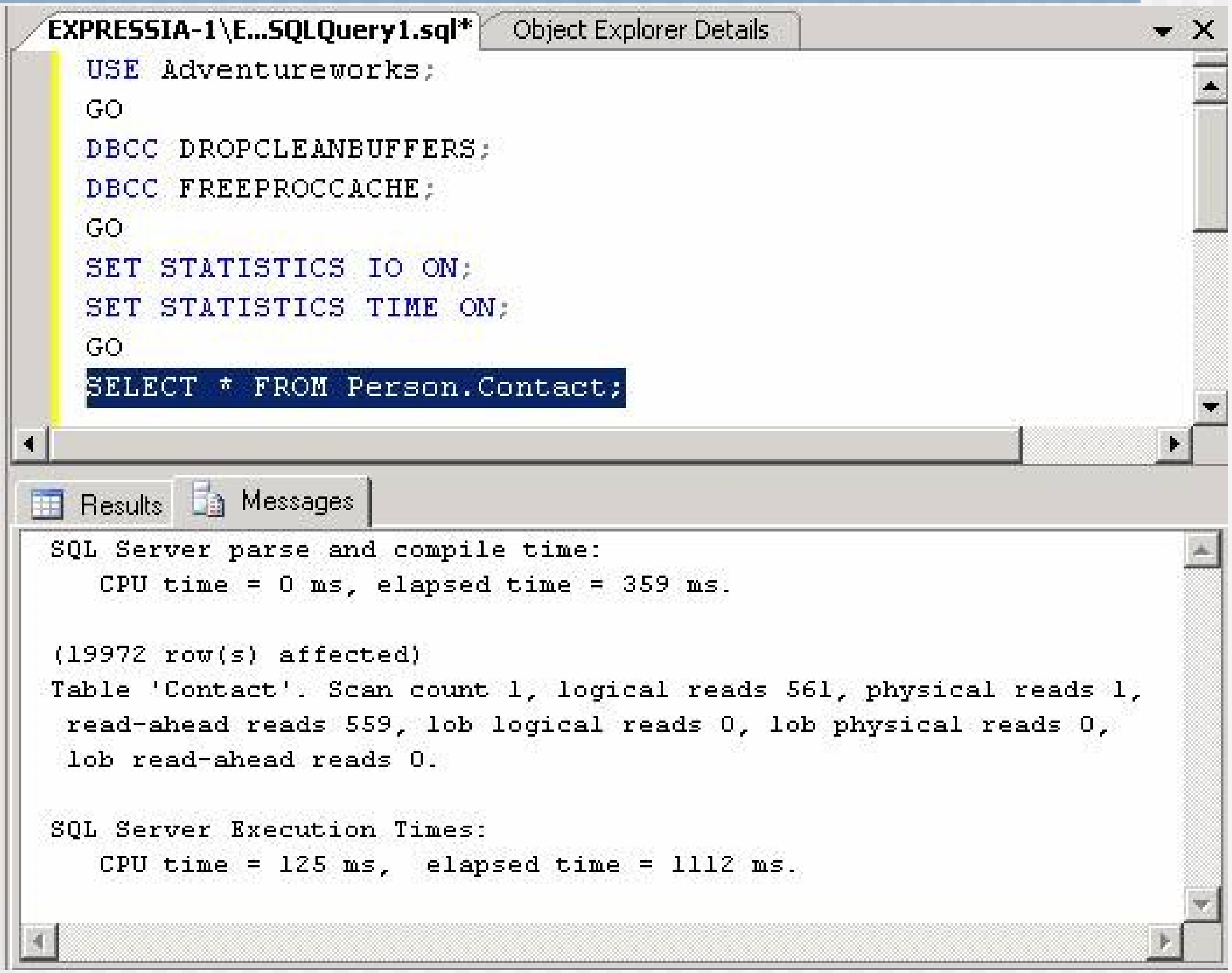
- *CPU time, elapsed time* – CPU time and elapsed time to parse and compile the query, and to execute it

STATISTICS IO and STATISTICS TIME



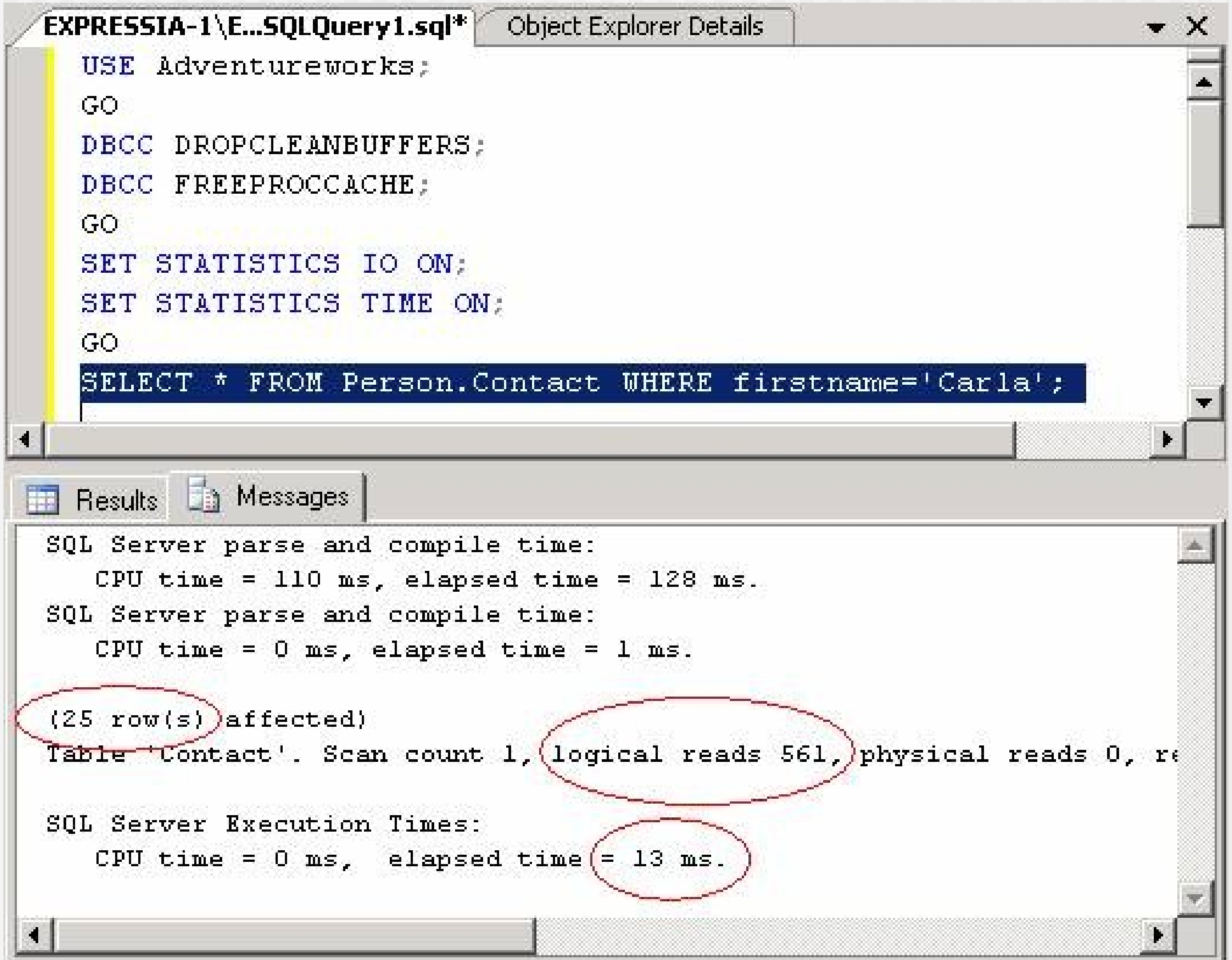
- *physical reads* - number of pages read from the disk
- *read-ahead reads* - number of pages placed in the cache for the query

STATISTICS IO and STATISTICS TIME



- *scan count* - number of seeks or scans after reaching the leaves
- *logical reads* - number of pages read from the data cache

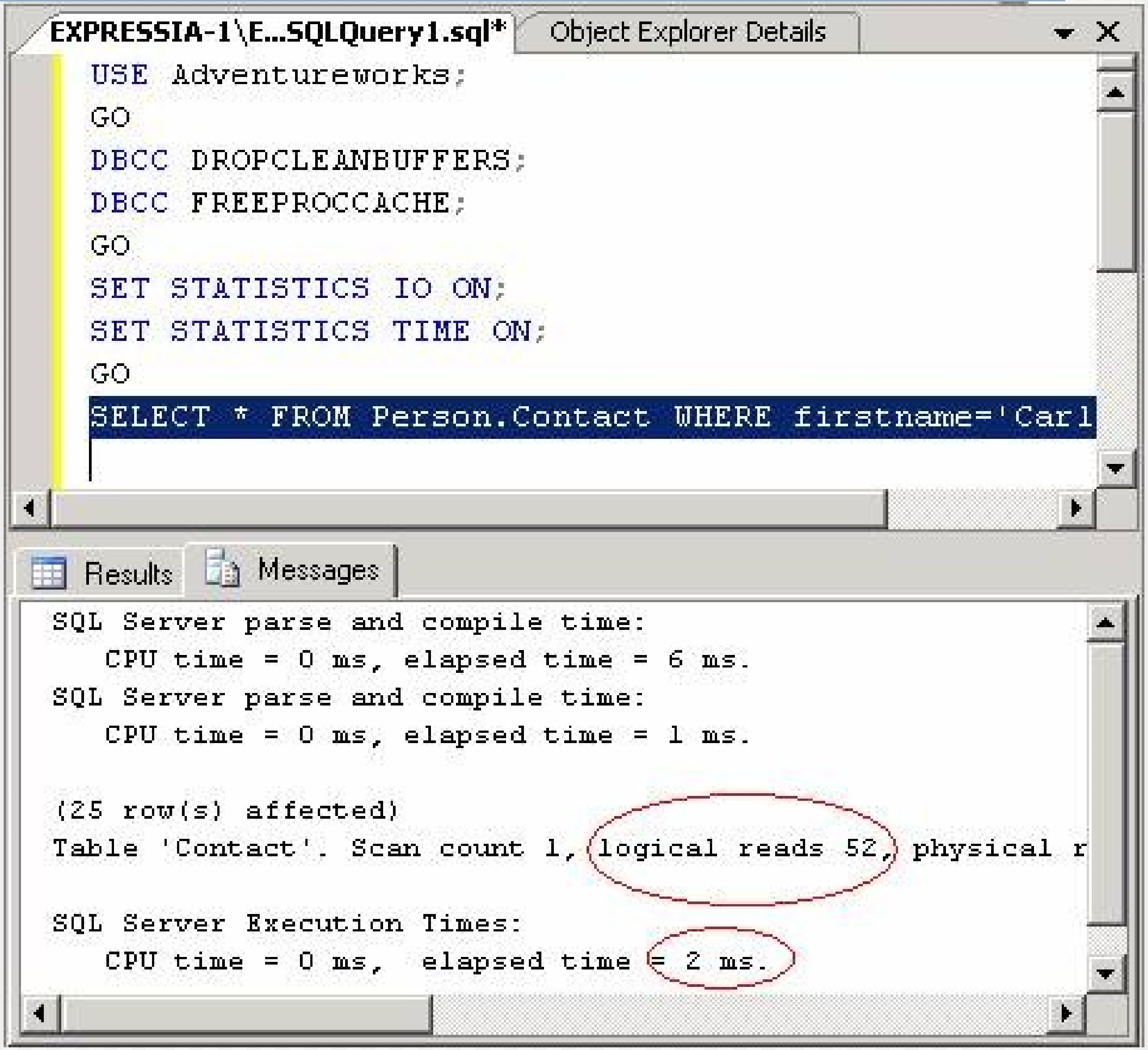
STATISTICS IO and STATISTICS TIME



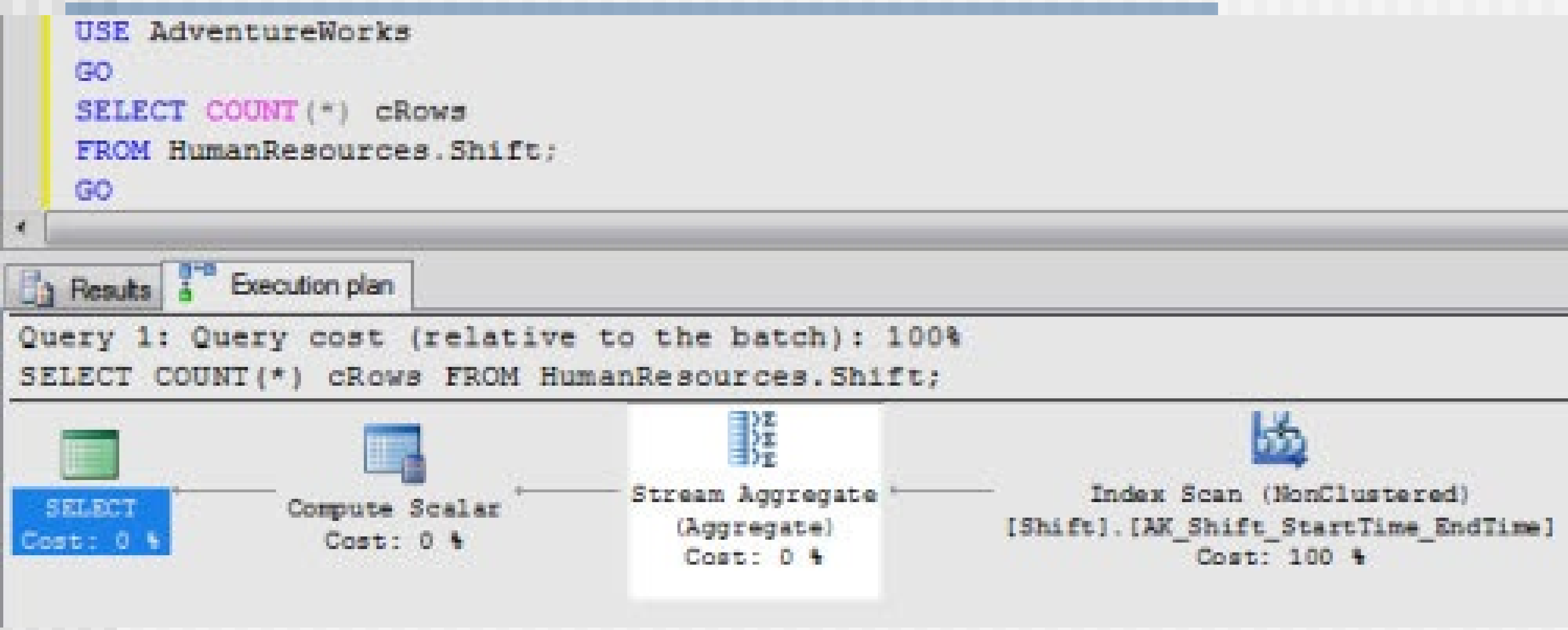
STATISTICS IO and STATISTICS TIME

```
USE AdventureWorks
GO
CREATE NONCLUSTERED INDEX IDX_FirstName
  ON Person.Contact(FirstName ASC)
GO
```

STATISTICS IO and STATISTICS TIME



Graphical Execution Plan



SHOWPLAN_ALL

```
SET SHOWPLAN_ALL ON;
GO
SELECT COUNT(*) cRows
FROM HumanResources.Shift;
GO
SET SHOWPLAN_ALL OFF;
GO
```

Results

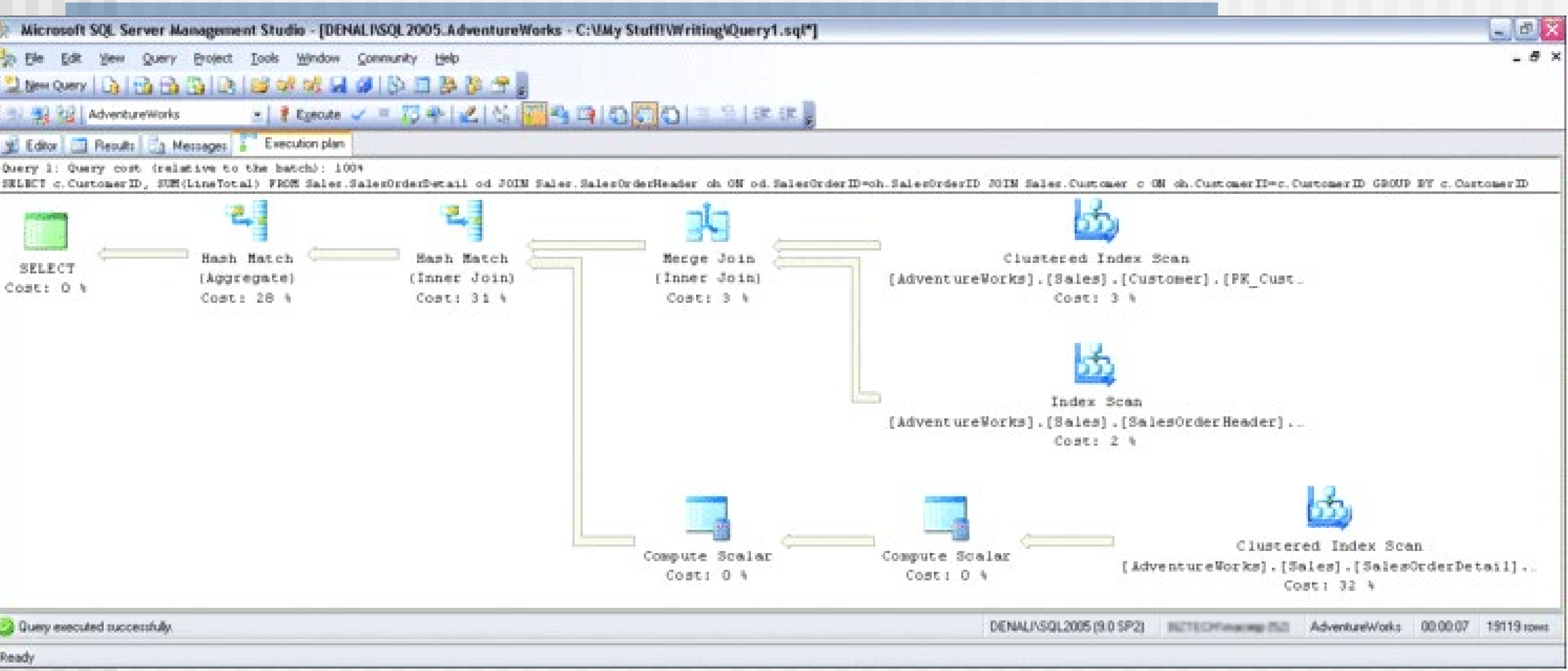
```
StmtText
--------------------------------------------------------------------------------------------------
SELECT COUNT(*) cRows
FROM HumanResources.Shift;
  |--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1004],0)))
       |--Stream Aggregate(DEFINE:([Expr1004]=Count(*)))
            |--Index Scan(OBJECT:([master].[HumanResources].[Shift].[AK_Shif

(4 row(s) affected)
```

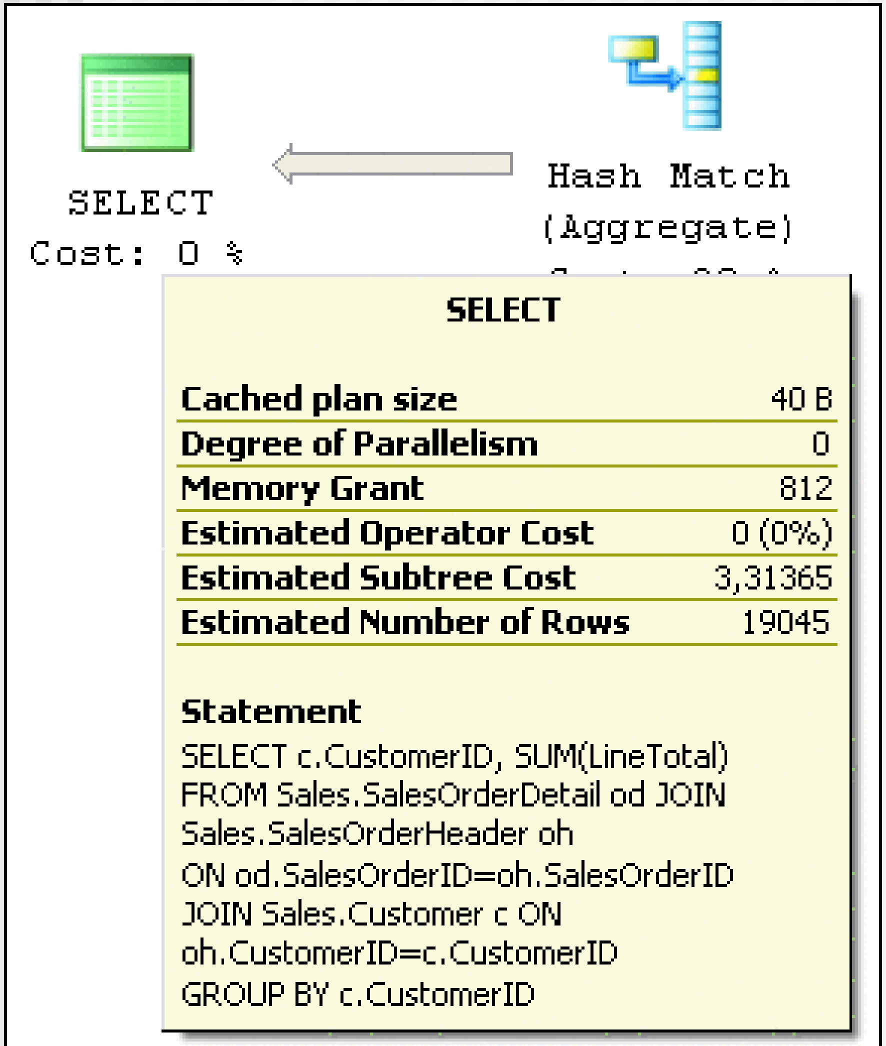
Graphical Execution Plan

```
SELECT c.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail od
  JOIN Sales.SalesOrderHeader oh ON
    od.SalesOrderID = oh.SalesOrderID
  JOIN Sales.Customer c ON
    oh.CustomerID = c.CustomerID
GROUP BY c.CustomerID
```

Graphical Execution Plan



Graphical Execution Plan

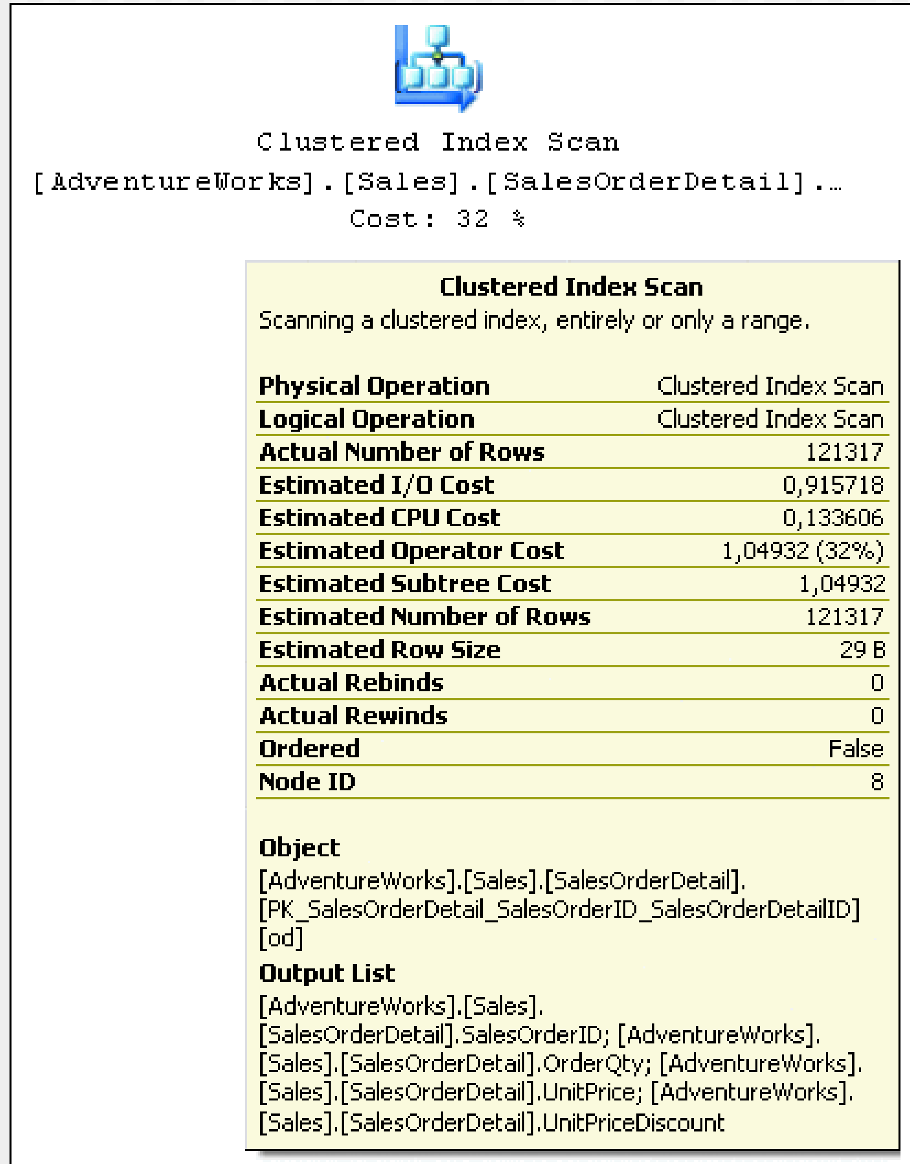


SELECT

Cached plan size	40 B
Degree of Parallelism	0
Memory Grant	812
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	3,31365
Estimated Number of Rows	19045

Statement

SELECT c.CustomerID, SUM(LineTotal) FROM Sales.SalesOrderDetail od JOIN Sales.SalesOrderHeader oh ON od.SalesOrderID=oh.SalesOrderID JOIN Sales.Customer c ON oh.CustomerID=c.CustomerID GROUP BY c.CustomerID



Clustered Index Scan

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	121317
Estimated I/O Cost	0,915718
Estimated CPU Cost	0,133606
Estimated Operator Cost	1,04932 (32%)
Estimated Subtree Cost	1,04932
Estimated Number of Rows	121317
Estimated Row Size	29 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	8

Object

[AdventureWorks].[Sales].[SalesOrderDetail].[PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID] [od]

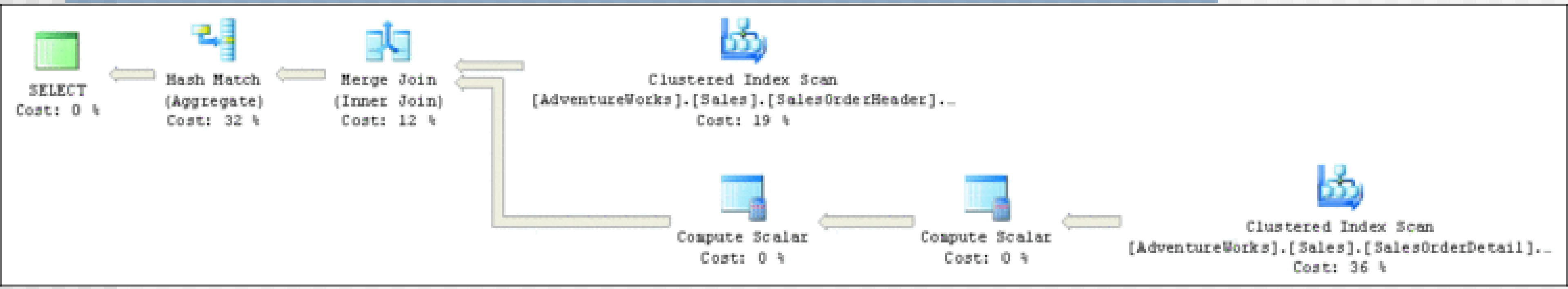
Output List

[AdventureWorks].[Sales].[SalesOrderDetail].SalesOrderID; [AdventureWorks].[Sales].[SalesOrderDetail].OrderQty; [AdventureWorks].[Sales].[SalesOrderDetail].UnitPrice; [AdventureWorks].[Sales].[SalesOrderDetail].UnitPriceDiscount

Graphical Execution Plan

```
SELECT oh.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail od
  JOIN Sales.SalesOrderHeader oh ON
    od.SalesOrderID=oh.SalesOrderID
GROUP BY oh.CustomerID
```

Graphical Execution Plan



Graphical Execution Plan

```
CREATE INDEX IDX_OrderDetail_OrderID_TotalLine
  ON Sales.SalesOrderDetail (SalesOrderID)
  INCLUDE (LineTotal)
```