

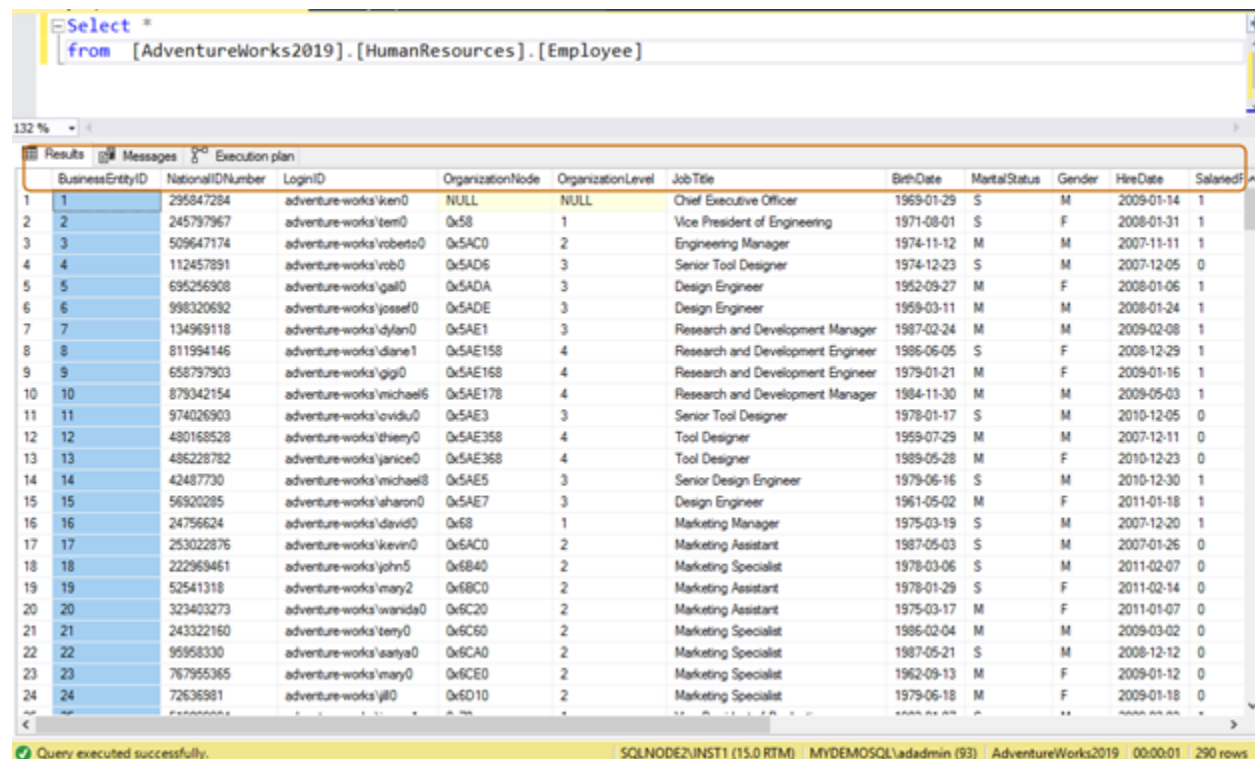
# SELECT \* vs SELECT column list

Usually, developers use the [SELECT \\* statement](#) to read data from a table. It reads all of the column's available data in the table. Suppose a table [AdventureWorks2019].[HumanResources].[Employee] stores data for 290 employees and you have a requirement to retrieve the following information:

- Employee National ID number
- DOB
- Gender
- Hire date

**Inefficient query:** If you use the SELECT \* statement, it returns all the column's data for all 290 employees.

Select \*  
from [AdventureWorks2019].[HumanResources].[Employee]



The screenshot shows a SQL Server query window with the following query:

```
Select *  
from [AdventureWorks2019].[HumanResources].[Employee]
```

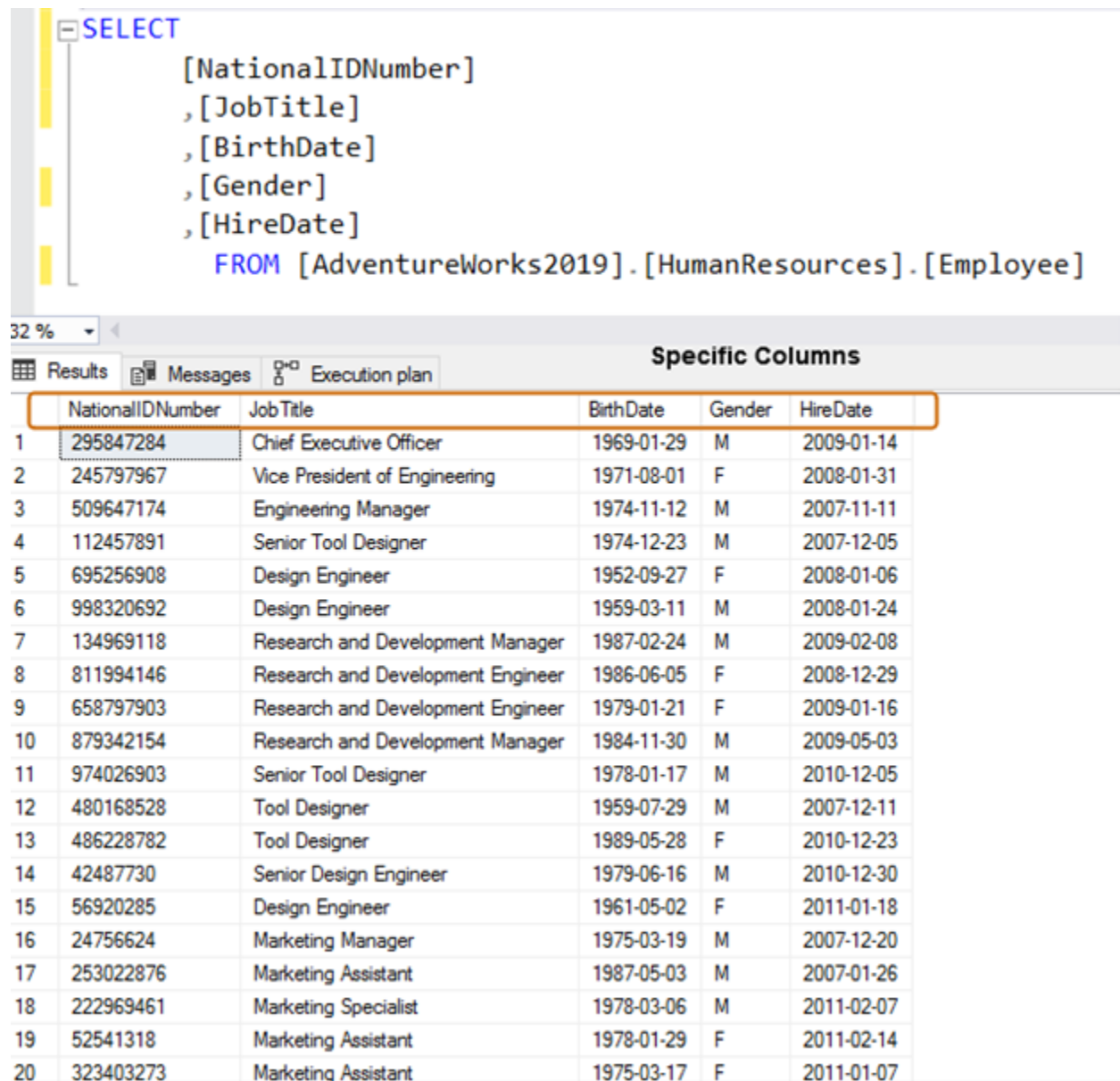
The query results are displayed in a table with the following columns: BusinessEntityID, NationalIDNumber, LoginID, OrganizationNode, OrganizationLevel, JobTitle, BirthDate, MaritalStatus, Gender, HireDate, and SalariedFlag. The table contains 290 rows of data, with the first 24 rows visible in the screenshot.

BusinessEntityID	NationalIDNumber	LoginID	OrganizationNode	OrganizationLevel	JobTitle	BirthDate	MaritalStatus	Gender	HireDate	SalariedFlag
1	295847284	adventure-works\ken0	NULL	NULL	Chief Executive Officer	1959-01-29	S	M	2009-01-14	1
2	245797967	adventure-works\ben0	0x58	1	Vice President of Engineering	1971-08-01	S	F	2008-01-31	1
3	509647174	adventure-works\roberto0	0x5AC0	2	Engineering Manager	1974-11-12	M	M	2007-11-11	1
4	112457891	adventure-works\rob0	0x5AD6	3	Senior Tool Designer	1974-12-23	S	M	2007-12-05	0
5	695256908	adventure-works\gail0	0x5ADA	3	Design Engineer	1952-09-27	M	F	2008-01-06	1
6	998320692	adventure-works\jossef0	0x5ADE	3	Design Engineer	1959-03-11	M	M	2008-01-24	1
7	134969118	adventure-works\dylan0	0x5AE1	3	Research and Development Manager	1987-02-24	M	M	2009-02-08	1
8	811994146	adventure-works\diane1	0x5AE158	4	Research and Development Engineer	1986-06-05	S	F	2008-12-29	1
9	658797903	adventure-works\gigi0	0x5AE168	4	Research and Development Engineer	1979-01-21	M	F	2009-01-16	1
10	879342154	adventure-works\michael6	0x5AE178	4	Research and Development Manager	1984-11-30	M	M	2009-05-03	1
11	974026903	adventure-works\ovidu0	0x5AE3	3	Senior Tool Designer	1978-01-17	S	M	2010-12-05	0
12	480168528	adventure-works\thieny0	0x5AE358	4	Tool Designer	1959-07-29	M	M	2007-12-11	0
13	486228782	adventure-works\janice0	0x5AE368	4	Tool Designer	1989-05-28	M	F	2010-12-23	0
14	42487730	adventure-works\michael8	0x5AE5	3	Senior Design Engineer	1979-06-16	S	M	2010-12-30	1
15	56920285	adventure-works\aharon0	0x5AE7	3	Design Engineer	1961-05-02	M	F	2011-01-18	1
16	24756624	adventure-works\david0	0x58	1	Marketing Manager	1975-03-19	S	M	2007-12-20	1
17	253022876	adventure-works\kevin0	0x5AC0	2	Marketing Assistant	1987-05-03	S	M	2007-01-26	0
18	222969461	adventure-works\john5	0x5B40	2	Marketing Specialist	1978-03-06	S	M	2011-02-07	0
19	52541318	adventure-works\mary2	0x5BC0	2	Marketing Assistant	1978-01-29	S	F	2011-02-14	0
20	323403273	adventure-works\wanida0	0x5C20	2	Marketing Assistant	1975-03-17	M	F	2011-01-07	0
21	243322160	adventure-works\terry0	0x5C60	2	Marketing Specialist	1986-02-04	M	M	2009-03-02	0
22	95958330	adventure-works\saitya0	0x5CA0	2	Marketing Specialist	1987-05-21	S	M	2008-12-12	0
23	767955365	adventure-works\mary0	0x5CE0	2	Marketing Specialist	1962-09-13	M	F	2009-01-12	0
24	72636981	adventure-works\jill0	0x5D10	2	Marketing Specialist	1979-06-18	M	F	2009-01-18	0

The status bar at the bottom indicates: Query executed successfully. SQLNODE2\INST1 (15.0 RTM) MYDEMOSQL\adadmin (93) AdventureWorks2019 00:00:01 290 rows

Instead, use specific column names for data retrieval.

```
SELECT
[NationalIDNumber]
,[JobTitle]
,[BirthDate]
,[Gender]
,[HireDate]
FROM [AdventureWorks2019].[HumanResources].[Employee]
```



The screenshot shows a SQL query window with the following text:

```
SELECT
    [NationalIDNumber]
    ,[JobTitle]
    ,[BirthDate]
    ,[Gender]
    ,[HireDate]
FROM [AdventureWorks2019].[HumanResources].[Employee]
```

Below the query window, the 'Results' tab is active, displaying a table with 20 rows of data. The table has the following columns: NationalIDNumber, JobTitle, BirthDate, Gender, and HireDate. The first row is highlighted with a blue background.

	NationalIDNumber	JobTitle	BirthDate	Gender	HireDate
1	295847284	Chief Executive Officer	1969-01-29	M	2009-01-14
2	245797967	Vice President of Engineering	1971-08-01	F	2008-01-31
3	509647174	Engineering Manager	1974-11-12	M	2007-11-11
4	112457891	Senior Tool Designer	1974-12-23	M	2007-12-05
5	695256908	Design Engineer	1952-09-27	F	2008-01-06
6	998320692	Design Engineer	1959-03-11	M	2008-01-24
7	134969118	Research and Development Manager	1987-02-24	M	2009-02-08
8	811994146	Research and Development Engineer	1986-06-05	F	2008-12-29
9	658797903	Research and Development Engineer	1979-01-21	F	2009-01-16
10	879342154	Research and Development Manager	1984-11-30	M	2009-05-03
11	974026903	Senior Tool Designer	1978-01-17	M	2010-12-05
12	480168528	Tool Designer	1959-07-29	M	2007-12-11
13	486228782	Tool Designer	1989-05-28	F	2010-12-23
14	42487730	Senior Design Engineer	1979-06-16	M	2010-12-30
15	56920285	Design Engineer	1961-05-02	F	2011-01-18
16	24756624	Marketing Manager	1975-03-19	M	2007-12-20
17	253022876	Marketing Assistant	1987-05-03	M	2007-01-26
18	222969461	Marketing Specialist	1978-03-06	M	2011-02-07
19	52541318	Marketing Assistant	1978-01-29	F	2011-02-14
20	323403273	Marketing Assistant	1975-03-17	F	2011-01-07

In the below execution plan, note the difference in the estimated row size for the same number of rows. You will notice a difference in CPU and IO for a large number of rows as well.

Top Plan		Bottom Plan	
Clustered Index Scan (Clustered)		Clustered Index Scan (Clustered)	
Actual Execution Mode	Row	Actual Execution Mode	Row
Actual I/O Statistics		Actual I/O Statistics	
Actual Number of Batches	0	Actual Number of Batches	0
Actual Number of Rows for All Executions	290	Actual Number of Rows for All Executions	290
Actual Rebinds	0	Actual Rebinds	0
Actual Rewinds	0	Actual Rewinds	0
Actual Time Statistics		Actual Time Statistics	
Defined Values	[AdventureWorks2019].[HumanResources].[Employee].Bu	Defined Values	[AdventureWorks2019].[HumanResources].[Employee].N
Description	Scanning a clustered index, entirely or only a range.	Description	Scanning a clustered index, entirely or only a range.
Estimated CPU Cost	0.000476	Estimated CPU Cost	0.000476
Estimated Execution Mode	Row	Estimated Execution Mode	Row
Estimated I/O Cost	0.0075634	Estimated I/O Cost	0.0075634
Estimated Number of Executions	1	Estimated Number of Executions	1
Estimated Number of Rows Per Execution	290	Estimated Number of Rows Per Execution	290
Estimated Number of Rows to be Read	290	Estimated Number of Rows to be Read	290
Estimated Operator Cost	0.0080454 (99%)	Estimated Operator Cost	0.0080454 (100%)
Estimated Rebinds	0	Estimated Rebinds	0
Estimated Rewinds	0	Estimated Rewinds	0
Estimated Row Size	828 B	Estimated Row Size	85 B
Estimated Subtree Cost	0.0080454	Estimated Subtree Cost	0.0080454
Forced Index	False	Forced Index	False
ForceScan	False	ForceScan	False
Logical Operation	Clustered Index Scan	Logical Operation	Clustered Index Scan
Node ID	2	Node ID	0
NoExpandHint	False	NoExpandHint	False
Number of Executions	1	Number of Executions	1
Number of Rows Read	290	Number of Rows Read	290
Object	[AdventureWorks2019].[HumanResources].[Employee].[PK_E	Object	[AdventureWorks2019].[HumanResources].[Employee].[PK_E
Ordered	False	Ordered	False
Output List	[AdventureWorks2019].[HumanResources].[Employee].Bu	Output List	[AdventureWorks2019].[HumanResources].[Employee].N
Parallel	False	Parallel	False
Physical Operation	Clustered Index Scan	Physical Operation	Clustered Index Scan
Storage	RowStore	Storage	RowStore
TableCardinality	290	TableCardinality	290

## Use of COUNT() vs. EXISTS

Suppose you want to check if a specific record exists in the SQL table. Usually, we use COUNT (\*) to check the record, and it returns the number of records in the output.

However, we can use the IF EXISTS() function for this purpose. For the comparison, I enabled the statistics before executing the queries.

### The query for COUNT()

```
SET STATISTICS IO ON
```

```
Select count(*) from [AdventureWorks2019].[Sales].[SalesOrderDetail]
where [SalesOrderDetailID]=44824
```

```
SET STATISTICS IO OFF
```

### The query for IF EXISTS()

```
SET STATISTICS IO ON
```

```

IF EXISTS(Select [CarrierTrackingNumber] from
[AdventureWorks2019].[Sales].[SalesOrderDetail]
where [SalesOrderDetailID]=44824)
PRINT 'YES'
ELSE
PRINT 'NO'
SET STATISTICS IO OFF

```

I used [statisticsparser](#) for analyzing the statistics results of both queries. Look at the results below. The query with COUNT(\*) has 276 logical reads while the IF EXISTS() has 83 logical reads. You can even get a more significant reduction in logical reads with the IF EXISTS(). Therefore, you should use it to optimize SQL queries for better performance.

Row Num	Table	COUNT(*)	Scan Count	Logical Reads	Physical Reads	Read-Ahead Reads	LOB Logical Reads	LOB Physical Reads	LOB Read-Ahead Reads	% Logical Reads of Total Reads
	SalesOrderDetail		1	276	0	0	0	0	0	100.000
	Total		1	276	0	0	0	0	0	

Row Num	Table	IF EXISTS()	Scan Count	Logical Reads	Physical Reads	Read-Ahead Reads	LOB Logical Reads	LOB Physical Reads	LOB Read-Ahead Reads	% Logical Reads of Total Reads
	SalesOrderDetail		1	83	0	0	0	0	0	100.000
	Total		1	83	0	0	0	0	0	

## Avoid using SQL DISTINCT

Whenever we want unique records from the query, we habitually use the SQL DISTINCT clause. Suppose you joined two tables together, and in the output it returns the duplicate rows. A quick fix is to specify the DISTINCT operator that suppresses the duplicated row.

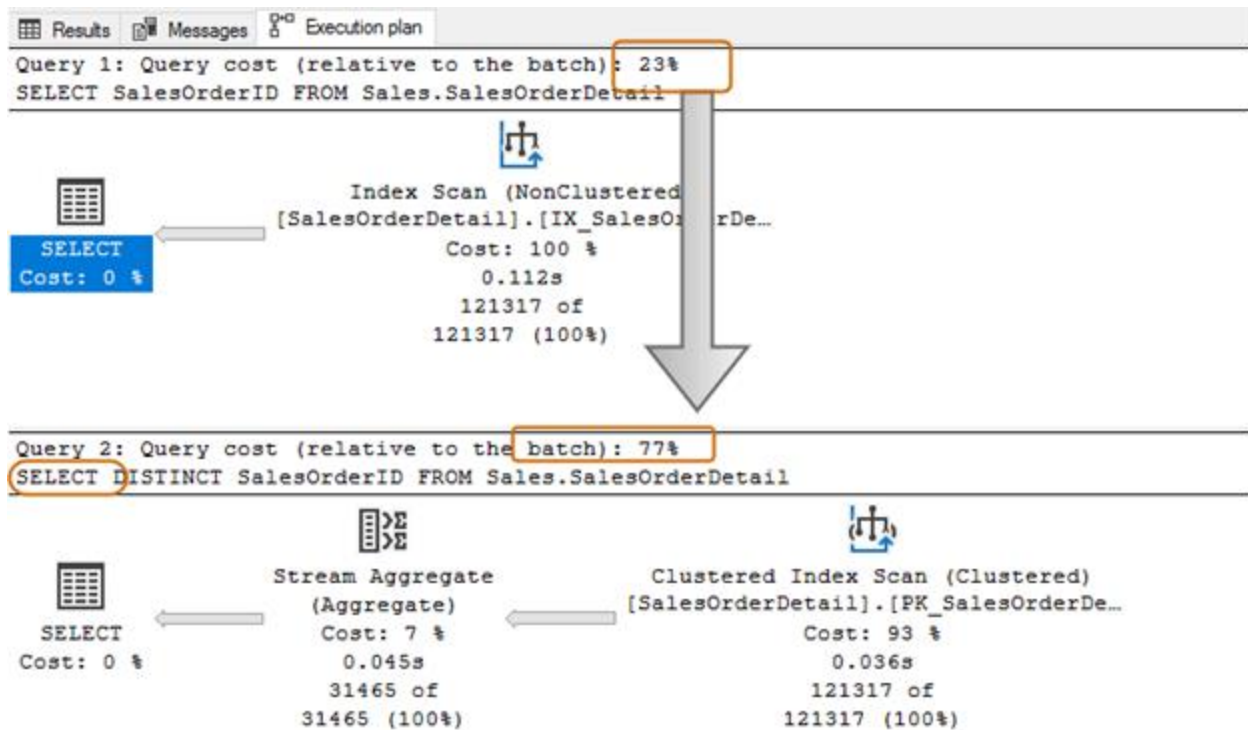
Let's look at the simple SELECT statements and compare the execution plans. The only difference between both queries is a DISTINCT operator.

```

SELECT SalesOrderID FROM Sales.SalesOrderDetail
Go
SELECT DISTINCT SalesOrderID FROM Sales.SalesOrderDetail
Go

```

With the DISTINCT operator, the query cost is 77%, while the earlier query (without DISTINCT) has only 23% batch cost.



You can use GROUP BY, CTE or a subquery for writing efficient SQL code instead of using DISTINCT for getting distinct values from the result set. Additionally, you can retrieve additional columns for a distinct result set.

```
SELECT SalesOrderID FROM Sales.SalesOrderDetail Group by SalesOrderID
```

## Wildcards usage in the SQL query

Suppose you want to search for the specific records containing names starting with the specified string. Developers use a [wildcard](#) to search for the matching records.

In the below query, it searches for the string Ken in the first name column. This query retrieves the expected results of **Kendra** and **Kenneth**. But, it also provides unexpected results as well, for example, **Mackenzie** and **Nkenge**.

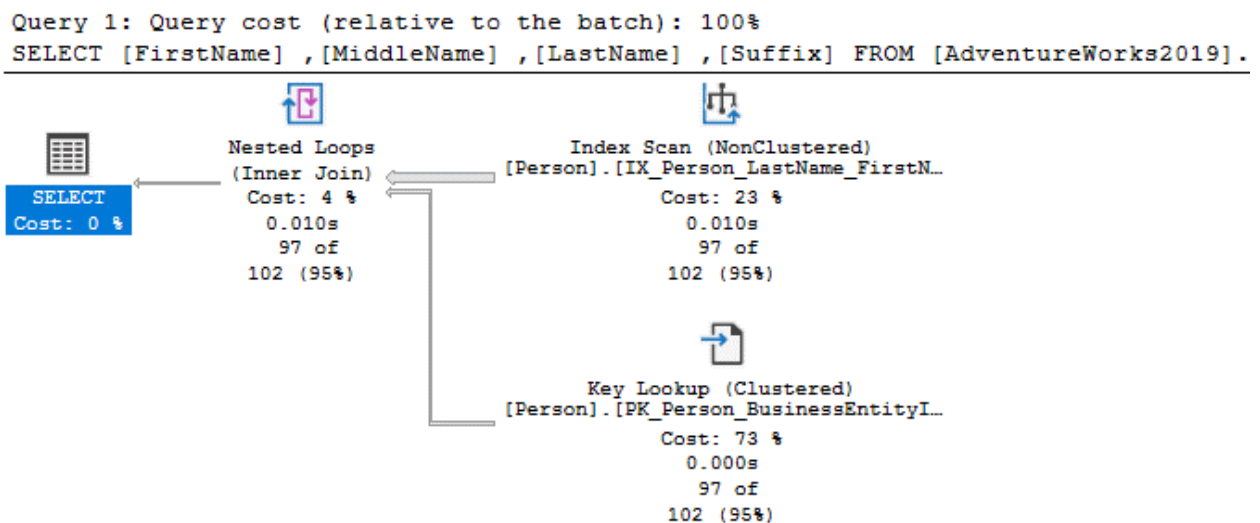
```

SELECT top 10
    [FirstName]
    , [MiddleName]
    , [LastName]
    , [Suffix]
FROM [AdventureWorks2019].[Person].[Person]
where firstname like '%Ken%'

```

FirstName	MiddleName	LastName	Suffix
Mackenzie	A	Adams	NULL
Mackenzie	NULL	Allen	NULL
Kendra	C	Alonso	NULL
Kendra	R	Alvarez	NULL
Kenneth	L	Anand	NULL
Kenneth	NULL	Andersen	NULL
Mackenzie	J	Bailey	NULL
Mackenzie	NULL	Baker	NULL
Kenneth	NULL	Becker	NULL
Kendra	NULL	Blanco	NULL

In the execution plan, you see the index scan and the key lookup for the above query.



You can avoid the unexpected result using the wildcard character at the end of the string.

```

SELECT Top 10
[FirstName]
,[MiddleName]
,[LastName]
,[Suffix]
FROM [AdventureWorks2019].[Person].[Person]
Where firstname like 'Ken%'

```

Now, you get the filtered result based on your requirements.

The screenshot displays a SQL query in the query editor and its results in the Results tab. The query is as follows:

```

SET STATISTICS IO ON
SELECT
    [FirstName]
    ,[MiddleName]
    ,[LastName]
    ,[Suffix]
FROM [AdventureWorks2019].[Person].[Person]
where firstname like 'Ken%'
SET STATISTICS IO OFF

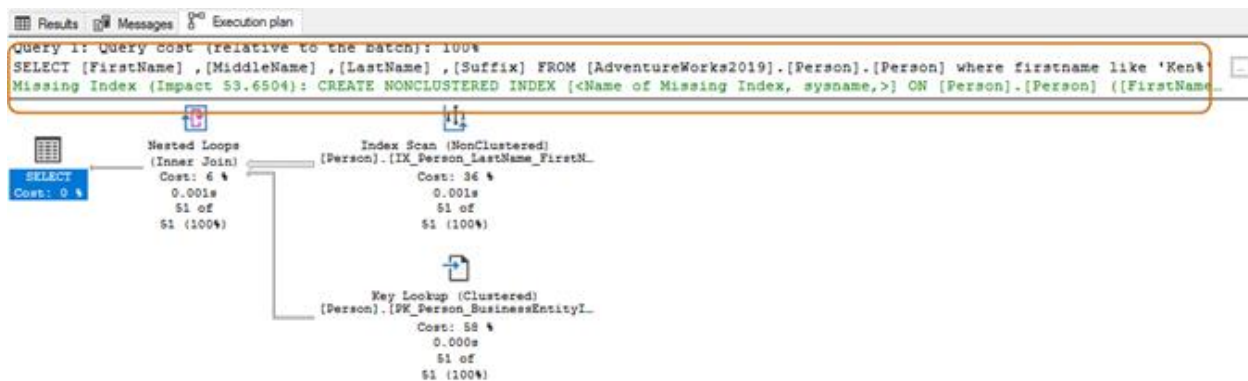
```

The Results tab shows a table with 9 rows and 5 columns: FirstName, MiddleName, LastName, and Suffix. The first row is highlighted with a blue selection box.

	FirstName	MiddleName	LastName	Suffix
1	Kendra	C	Alonso	NULL
2	Kendra	R	Alvarez	NULL
3	Kenneth	L	Anand	NULL
4	Kenneth	NULL	Andersen	NULL
5	Kenneth	NULL	Becker	NULL
6	Kendra	NULL	Blanco	NULL
7	Kendra	NULL	Carlson	NULL
8	Kenneth	NULL	Deng	NULL
9	Kendra	NULL	Diaz	NULL

In using the wildcard character at the beginning, the query optimizer might not be able to use the suitable index. As shown in the below screenshot, with a trailing wild character, query optimizer suggests a missing index as well.





Here, you'll want to evaluate your application requirements. You should try to avoid using a wildcard character in the search strings, as it might force query optimizer to use a table scan. If the table is enormous, it would require higher system resources for IO, CPU and memory, and can cause performance issues for your SQL query.

## Use of the WHERE and HAVING clauses

The WHERE and HAVING clauses are used as data row filters. The WHERE clause filters the data before applying the grouping logic, while the HAVING clause filters rows after the aggregate calculations.

For example, in the below query, we use a data filter in the HAVING clause without a WHERE clause.

```
Select SalesOrderID,
SUM(UnitPrice* OrderQty) as OrderTotal
From Sales.salesOrderDetail
GROUP BY SalesOrderID
HAVING SalesOrderID>30000 and SalesOrderID<55555 and SUM(UnitPrice* OrderQty)>1
Go
```

The following query filters the data first in the WHERE clause and then uses the HAVING clause for the aggregate data filter.

```
Select SalesOrderID,
SUM(UnitPrice* OrderQty) as OrderTotal
From Sales.salesOrderDetail
where SalesOrderID>30000 and SalesOrderID<55555
GROUP BY SalesOrderID
HAVING SUM(UnitPrice* OrderQty)>1000
Go
```



I recommend using the WHERE clause for data filtering and the HAVING clause for your aggregate data filter as a best practice.

## Usage of the IN and EXISTS clauses

You should avoid using the IN-operator clause for your SQL queries. For example, in the below query, first, we found the product id from the [Production].[TransactionHistory] table and then looked for the corresponding records in the [Production].[Product] table.

```
Select * from [Production].[Product] p
where productid IN
(select productid from [AdventureWorks2019].[Production].[TransactionHistory]);
Go
```

In the below query, we replaced the IN clause with an EXISTS clause.

```
Select * from [Production].[Product] p
where EXISTS
(select productid from [AdventureWorks2019].[Production].[TransactionHistory])
```

Now, let's compare the statistics after executing both queries.

The IN clause uses 504 scans, while the EXISTS clause uses 1 scan for the [Production].[TransactionHistory] table].

Row Num	Table	Scan Count	Logical Reads	Physical Reads	Read-Ahead Reads	LOB Logical Reads	LOB Physical Reads	LOB Read-Ahead Reads	% Logical Reads of Total Reads
	TransactionHistory	504	1,101	0	0	0	0	0	98.656
	Product	1	15	0	0	0	0	0	1.344
	Total	505	1,116	0	0	0	0	0	

(1 row affected)

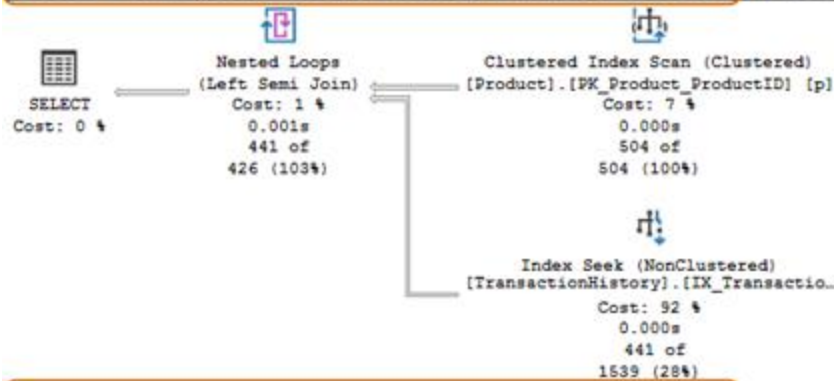
(504 rows affected)

Row Num	Table	Scan Count	Logical Reads	Physical Reads	Read-Ahead Reads	LOB Logical Reads	LOB Physical Reads	LOB Read-Ahead Reads	% Logical Reads of Total Reads
	TransactionHistory	1	1,512	0	0	0	0	0	99.018
	Product	1	15	0	0	0	0	0	0.982
	Total	2	1,527	0	0	0	0	0	

The IN clause query batch costs 74%, whereas the EXISTS clause cost is 24%. Therefore, you should avoid the IN clause especially if the subquery returns a large dataset.

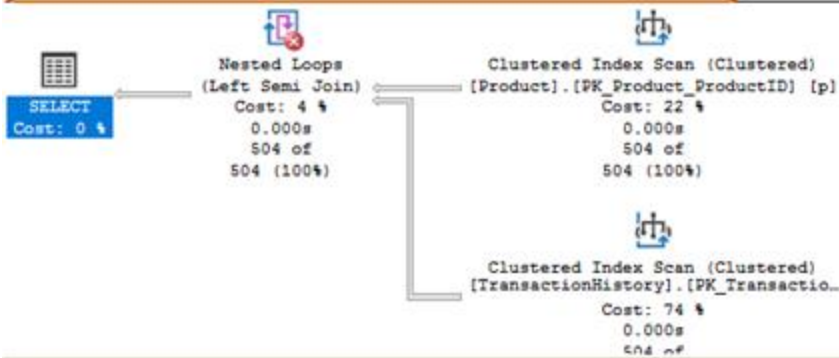
Query 1: Query cost (relative to the batch): 76%

Select \* from [Production].[Product] p where productid IN (select productid from [AdventureWorks



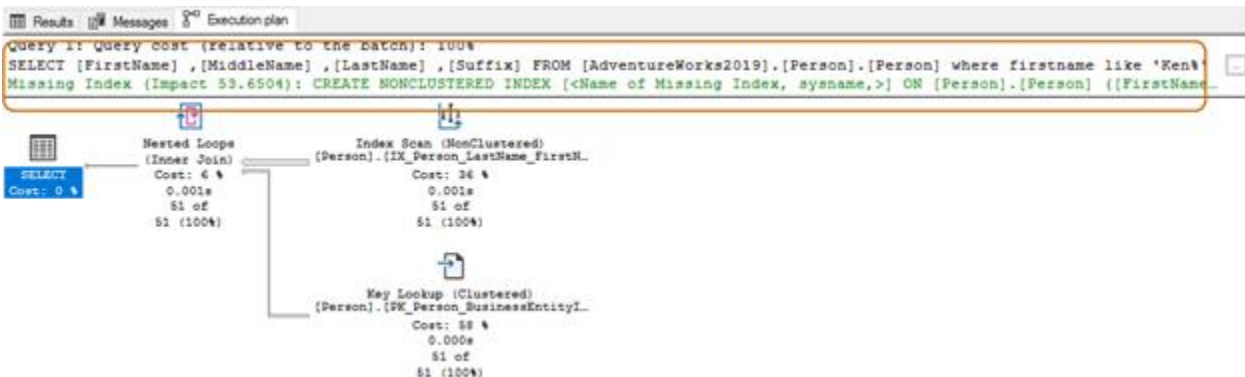
Query 2: Query cost (relative to the batch): 24%

Select \* from [Production].[Product] p where EXISTS (select productid from [AdventureWorks2019].



## Missing indexes

Sometimes, when we execute a SQL query and look for the actual [execution plan](#) in SSMS, you get a suggestion about an index that might improve your SQL query.



Alternatively, you can use the dynamic management views to check the details of missing indexes in your environment.

- [sys.dm\\_db\\_missing\\_index\\_details](#)
- [sys.dm\\_db\\_missing\\_index\\_group\\_stats](#)
- [sys.dm\\_db\\_missing\\_index\\_groups](#)
- [sys.dm\\_db\\_missing\\_index\\_columns](#)

Usually, DBAs follow the advice from SSMS and create the indexes. It might improve query performance for the moment. However, you should not create the index directly based on those recommendations. It might affect other query performances and slow down your INSERT and UPDATE statements.

- First, review the existing indexes for your SQL table.
- Note, over-indexing and under-indexing are both bad for query performance.
- Apply the missing index recommendations with the highest impact after reviewing your existing indexes and implement it on your lower environment. If your workload works well after implementing the new missing index, it's worth adding it.

I suggest you refer to this article for detailed indexing best practices: [11 SQL Server Index Best Practices for Improved Performance Tuning](#).

## Query hints

Developers specify the query hints explicitly in their t-SQL statements. These query hints override query optimizer behavior and force it to [prepare an execution plan](#) based on your query hint. Frequently used query hints are NOLOCK, Optimize For and Recompile Merge/Hash/Loop. They are short-term fixes for your queries. However, you should work on analyzing your query, indexes, statistics and execution plan for a permanent solution.

As per best practices, you should minimize the usage of any query hint. You want to use the query hints in the SQL query after first understanding the implications of it, and do not use it unnecessarily.

## SQL query optimization reminders

As we discussed, SQL query optimization is an open-ended road. You can apply best practices and small fixes that can greatly improve performance. Consider the following tips for better query development:

- Always look at system resources (disks, CPU, memory) allocations

- Review your startup trace flags, indexes and database maintenance tasks
- Analyze your workload using extended events, profiler or third-party [database monitoring tools](#)
- Always implement any solution (even if you are 100% confident) on the test environment first and analyze its impact; once you are satisfied, plan for production implementations