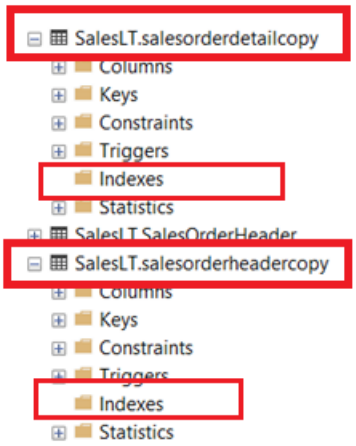
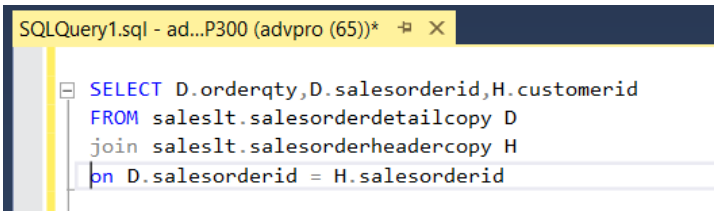
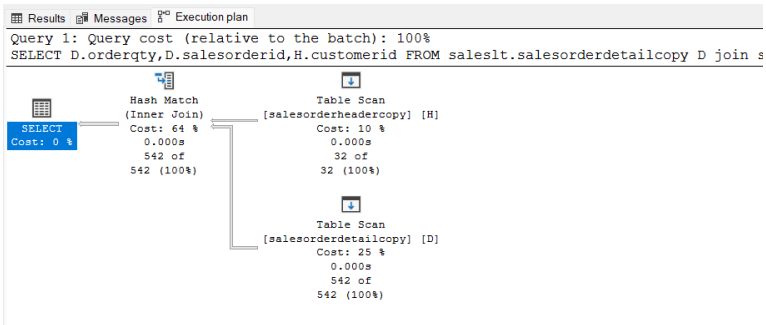
Recommend, Query construct modifications based on resource usage

we created the SalesOrderDetailCopy and SalesOrderHeaderCopy tables for previous usage. But the important thing about these is that they've got no indexes whatsoever.



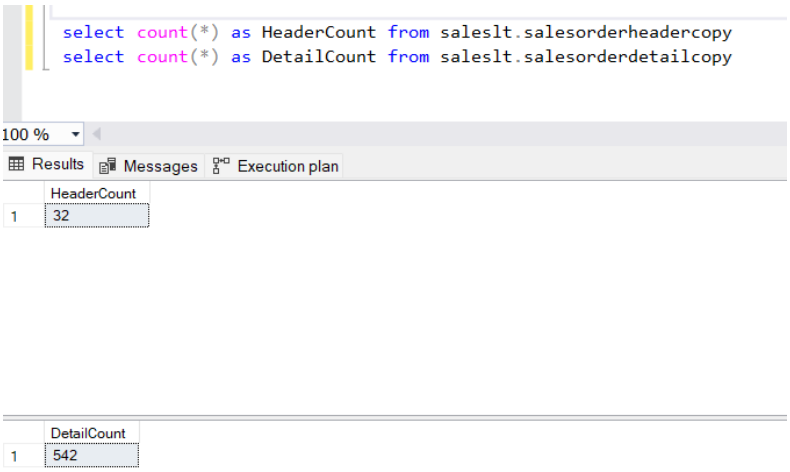
So, I created this query and ran it, and you can see users' table scans, because there's no WHERE clause, so you'll be using a scan. And it uses a Hash Match to create the results.





Now, the computer is not complaining about the lack of indexes, and there's a reason for this. If you have a look at how many rows we've got,

We've only got 32 rows.



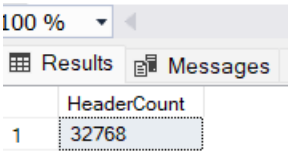
So really it doesn't make that much difference. So, when would it choose the indexes? Well, it uses it when getting the join here between SalesOrderID.

Now, what if this wasn't 32 rows?

What if this was a lot more rows?

So, let's double the number of rows. So, I'm inserting it into this table, so that it doubles the number of rows. So now there are 64 rows. So, if I execute this, we're still fine. But let's execute this a few more times. So 128, 256, 512. So now it is up to 1,024 rows. So, let's execute this. Look at the execution plan.

We're still fine. Now let's do it a few more times. So 2,048, 4,096, 8,192. Taking a bit longer. And 16,384.

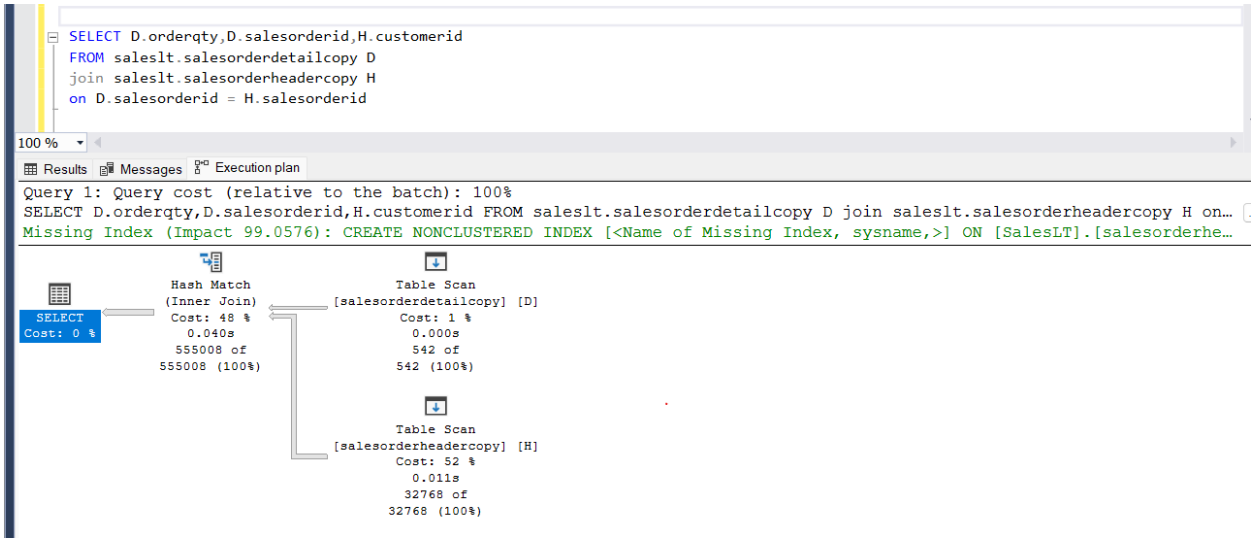


So now let's execute our original query. And if we now have a look at the execution plan, you can see that there is a missing index, and a suggestion of what we create.

Create non-clustered index, add a name on and it gives what we should be having.

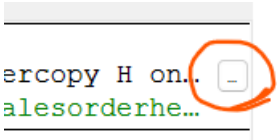
So, the SalesOrderID and the CustomerID as an include. So, the columns included would be in a separate part.

So, in other words, you would have the main index and then when you get to individual rows, there would then be a link to the separate parts.



So, it's a lot more efficient to be able to include columns rather than just having just one column and then having to get the information elsewhere.

So, if I click on the dot dot dot



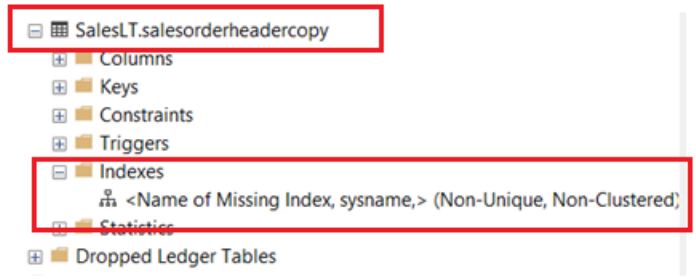
you can see this is the query text

```
/*
This query text was retrieved from showplan XML, and may be truncated.
*/

SELECT D.orderqty,D.salesorderid,H.customerid
 FROM saleslt.salesorderdetailcopy D
 join saleslt.salesorderheadercopy H
 on D.salesorderid = H.salesorderid
```

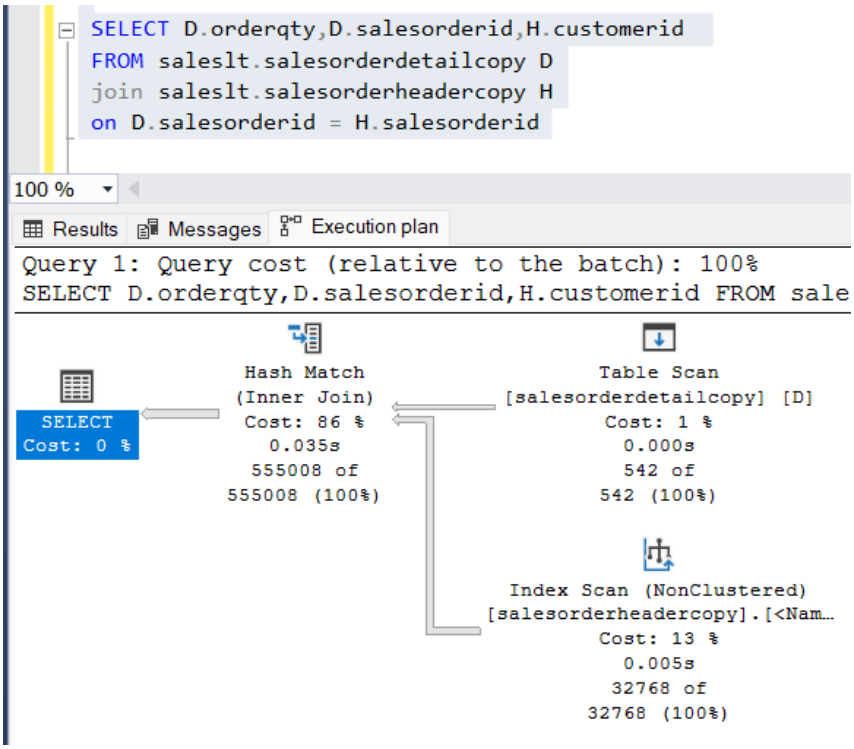
and if I right-hand click → missing index details and there is my code that I would need

```
/*
Missing Index Details from SQLQuery1.sql - advpro.database.windows.net.DP300 (advpro (65))
The Query Processor estimates that implementing the following index could improve the query cost by 99.0576%.
*/

/*
USE [DP300]
GO
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [SalesLT].[salesorderheadercopy] ([SalesOrderID])
INCLUDE ([CustomerID])
GO
*/
```

Now the index is created,



So, let's run that code and have a look at this query again. And now we can see users in index scan non-clustered of the SalesOrderHeaderCopy.



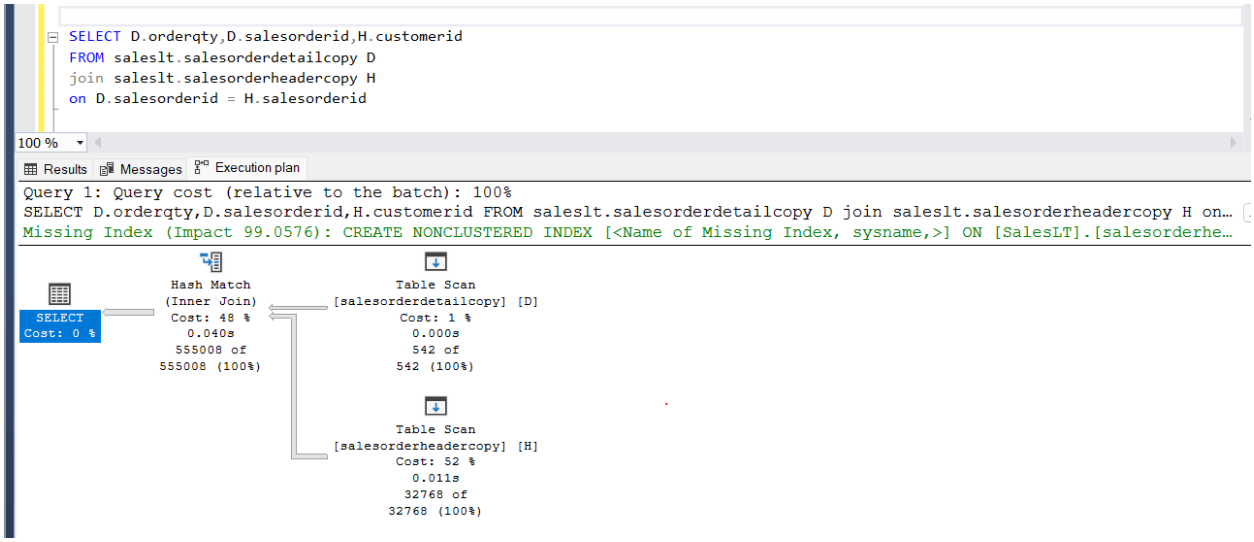
Right now, let's just drop that index, so I'll get rid of the words CREATE and non-clustered temporarily.

So just drop this index.

```
USE [DP300]
GO
drop INDEX [<Name of Missing Index, sysname,>]
ON [SalesLT].[salesorderheadercopy] ([SalesOrderID])
INCLUDE ([CustomerID])
GO
```

I didn't give it a proper name; I'm just using the default name. So now it's dropped

So run this again, and we'll have that problem again.

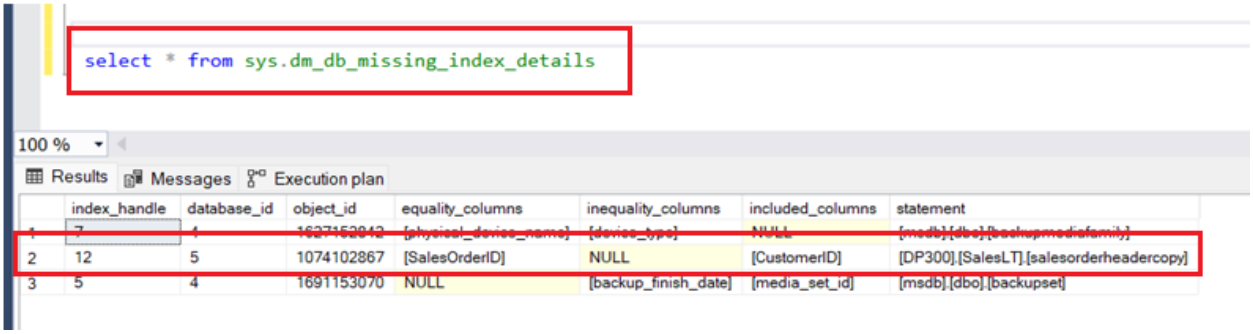


So, this is one way of detecting missing indexes,

but is there a better way to do it for the entire database?

The answer is, we can use a DMV.

And this DMV is sys.dm_db_missing_index_details, plural.



So, if we have a look at this, because we are missing at this current index, it comes up. So, you can see it is in database ID 5, object ID And you can see that we've got an equality column of SalesOrderID, no inequality columns, and then some included columns.

What are equality and inequality?

```
SELECT D.orderqty,D.salesorderid,H.customerid
FROM saleslt.salesorderdetailcopy D
join saleslt.salesorderheadercopy H
on D.salesorderid = H.salesorderid
```

So, we go back to the query, and you can see that the query has got this equal. So, that is equality. So, if somewhere like in the where clause we would have no equals to or greater than or anything apart from equality, then that would be inequality.

When you are creating an index, you put equality first and then the inequality columns. Both should be in the key and then you've got the included columns in the INCLUDE section of the index.

Examples of the full thing that you can do with these sorts of things. So, this is a much wider query, and it uses three different DMVs, but it gives basically the same results.

```
SELECT
CONVERT (varchar, getdate(), 126) AS runtime
, mig.index_group_handle
, mid.index_handle
, CONVERT (decimal (28,1), migs.avg_total_user_cost * migs.avg_user_impact *
(migs.user_seeks + migs.user_scans)) AS improvement_measure
, 'CREATE INDEX missing_index_' + CONVERT (varchar, mig.index_group_handle) + '_' +
CONVERT (varchar, mid.index_handle) + ' ON ' + mid.statement + '
(' + ISNULL (mid.equality_columns,'')
+ CASE WHEN mid.equality_columns IS NOT NULL
AND mid.inequality_columns IS NOT NULL
THEN ',' ELSE '' END + ISNULL (mid.inequality_columns, '') + ')'
+ ISNULL (' INCLUDE (' + mid.included_columns + ')', '') AS create_index_statement
, migs.*
, mid.database_id
, mid.[object_id]
FROM sys.dm_db_missing_index_groups AS mig
INNER JOIN sys.dm_db_missing_index_group_stats AS migs
ON migs.group_handle = mig.index_group_handle
INNER JOIN sys.dm_db_missing_index_details AS mid
ON mig.index_handle = mid.index_handle
ORDER BY migs.avg_total_user_cost * migs.avg_user_impact * (migs.user_seeks + migs.user_scans) DESC
```

100 %

Results | Messages

	runtime	index_group_handle	index_handle	improvement_measure	create_index_statement	group_handle	unique_compiles	user_seek
1	2025-02-24T08:54:38.137	13	12	143.7	CREATE INDEX missing_index_13_12 ON [DP300].[SalesLT].[salesorderhe...	13	1	1

It's just that now you have the correct index statement there. So, you just need to copy and paste it. So, there's my entire index statement, including a good name for the index.

Going further on, we've got things like *avg_user_impact*. So, that's the average percentage benefit that user queries could experience if this missing index group was implemented.

You've got *avg_total_user_cost*, so that's the average cost that could be reduced by the index. So, you must go okay, this might be a big saving, 98%. But if we're saving not that much, is it worth it?

But this is when a query like this could help, because it orders by those indexes which are most beneficial. So, a quick look at what all these DMVs are.

So, they all start dm_db_missing_index. So, give you the details. Groups that return information of our indexes that are missing from a specific index group. And then we have group_stats. So, this is summary information about groups of missing indexes.

And you've also got dm_db_missing_index_group_stats query, which returns information about queries that are missing in index.

So, this is probably the biggest query construct modification I can think of.

Create indexes as and when they're needed. Of course, don't create too many, because as soon as you insert additional information update or delete or merge, then the indexes need to be updated,

so that you could grind your system to a halt. In terms of actual changes that could be needed to the query, well, we've already gone through everything that we need to be sargable.

So, we need to ensure, for instance, we're not using functions when we can avoid it.

So don't use the year function when you can use between two sets of dates.

don't use left when you can use like.

don't use the is-no function,

So, if there is no certain field, then give me this, unless you've got a specific need for it. Instead, you could say if my field is no, or my field is equal to whatever.

Yes, it adds a bit more to the words,

but it makes it sargable, which means that you can use any index that are available. So, you don't necessarily need to know all these specifics of how to change a query to make a sargable query.

You just need to know, for instance, you shouldn't use functions when you can avoid it if there is a better alternative that can use an index. So, the greatest thing you can do to speed up your queries is to have appropriate indexes, and you can do that by using sys.dm_db_missing index details and similar DMVs.