Red Flags in Your Query Plans

•••

Jay Rasch

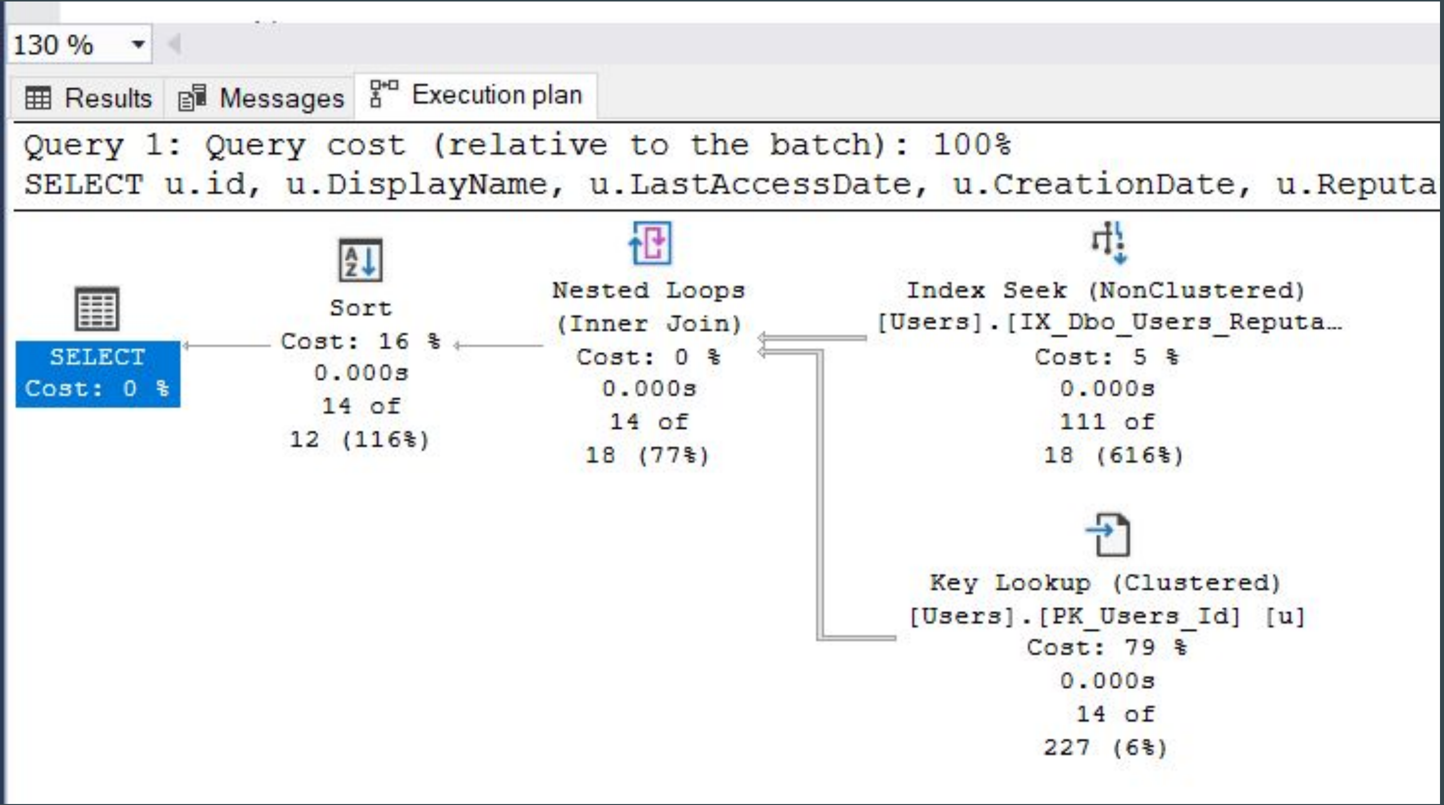
Who is Jay Rasch?

- Lead database developer on a project team for logistics and lead database developer on the database performance tuning team at Webstaurantstore.com, a restaurant supply e-commerce company.
- I have one wife and two cats and I’m quite fond of them.
- I don’t like pina coladas (they are too sweet) or getting caught in the rain (I’ll stay dry, thanks).
- For fun I like to hike and am an enthusiast in the United States Bartender’s Guild.
- My email is JamesMRasch@gmail.com

What Are Query Plans?

- Query plans are graphical representations of what the optimizer is doing to accomplish a query.
- Your query tells the optimizer what you want to be done and then based around the query, the database objects, statistics, and a bunch of heuristics, the optimizer will try to find an acceptable query plan in an acceptable amount of time.
- They contain a ton of information and it all centers around the order in which data was accessed, how it was accessed, and how the data is joined and transformed.

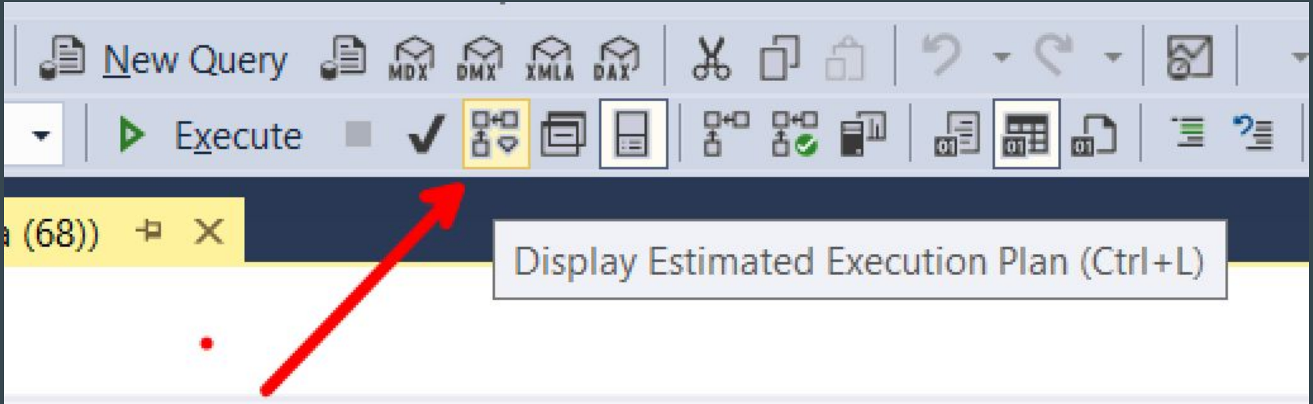
Behold, a query plan!



```
Query 1: Query cost (relative to the batch): 100%
SELECT u.id, u.DisplayName, u.LastAccessDate, u.CreationDate, u.Reputa
```

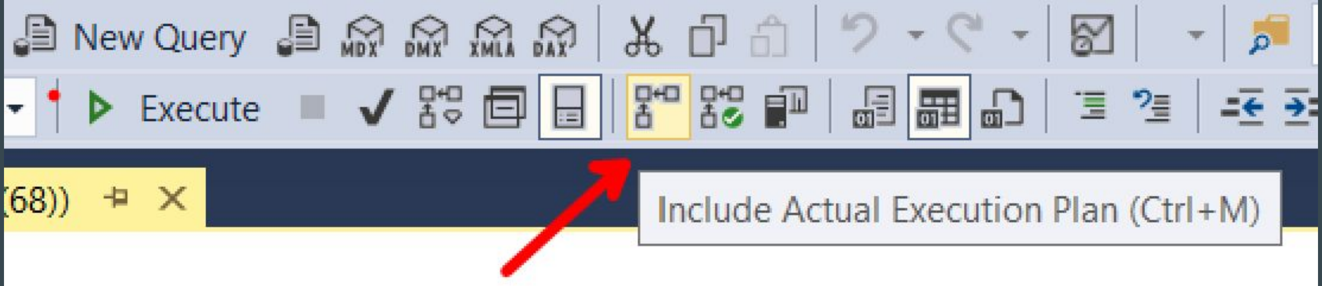
The Many Kinds of Query Plans

- Estimated Query Plan- also called the “query plan” by some very smart people.
  - This does not execute the query
    - It checks if the plan exists in the cache for that query. If it exists, that plan is returned.
    - If the plan does not exist, the optimizer complies a plan for the query and the plan is returned.



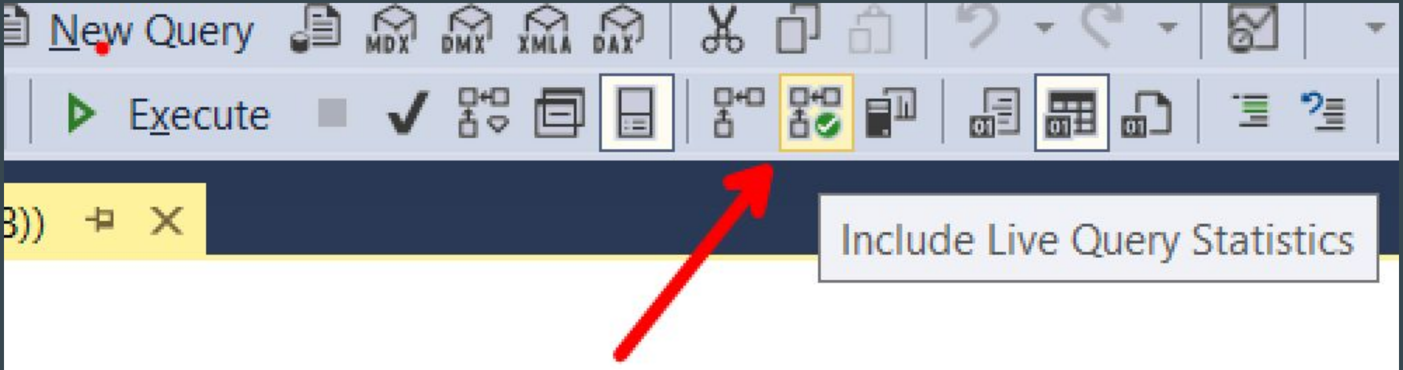
The Many Kinds of Query Plans

- Actual Query Plan- also called the "query plan with runtime statistics" by some very smart people.
  - The process starts out the same as the estimated query plan but the query runs and all sorts of information is returned.
  - This <u>should</u> give the same plan as you would have gotten from the estimated query plan but with all sorts of particulars about how your query ran.



The Many Kinds of Query Plans

- Include Live Query Statistics
  - This goes through the same process of checking the plan cache for the query and using the plan if it is found or compiling a plan if one does not exist, and then executing the query plan.
  - The distinguishing factor here is that as the query executes, it provides updates every second showing the progress of the query.



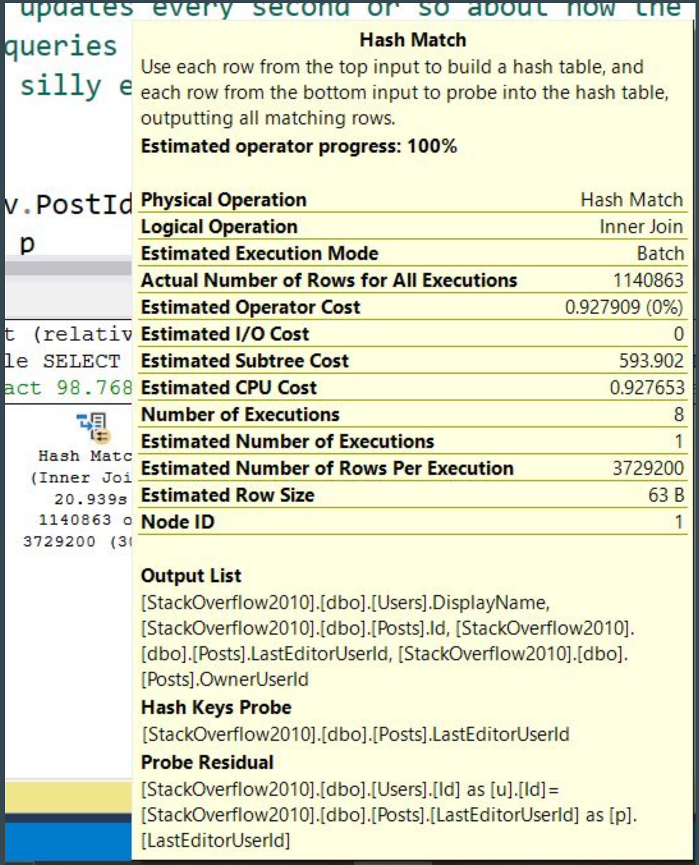
Can I Look At Query Plans Any Other Ways?

- You can look at the XML that the query plan is based on. It may not be especially easy to read but if you're looking for something specific it can be fast to search.
- SQL Sentry Plan Explorer makes viewing statistics from the query execution much easier to view than viewing the plan in SSMS directly.

Additional Information In Plans

- In addition to the query plan that directly appears, there are lots of additional pieces of information, I want to explicitly call out the tooltips and properties pane for operators.
- Tooltips appear when hovering over a particular operator and can give information like estimated and actual execution statistics and are readily accessible just with needing to hover over an operator.
- Properties require right clicking on an operator unless the properties pane is already pinned. These have much more information but take additional work to open up and view the data.

Tooltips (left) and Properties (right)



Hash Match

Use each row from the top input to build a hash table, and each row from the bottom input to probe into the hash table, outputting all matching rows.

Estimated operator progress: 100%

Physical Operation	Hash Match
Logical Operation	Inner Join
Estimated Execution Mode	Batch
Actual Number of Rows for All Executions	1140863
Estimated Operator Cost	0.927909 (0%)
Estimated I/O Cost	0
Estimated Subtree Cost	593.902
Estimated CPU Cost	0.927653
Number of Executions	8
Estimated Number of Executions	1
Estimated Number of Rows Per Execution	3729200
Estimated Row Size	63 B
Node ID	1

Output List

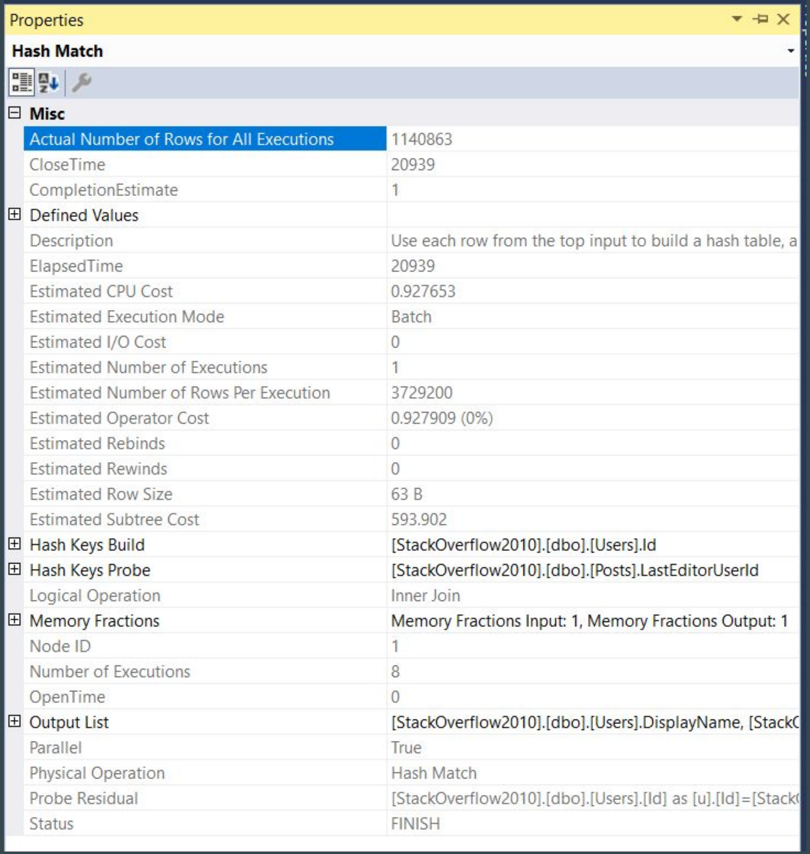
[StackOverflow2010].[dbo].[Users].DisplayName, [StackOverflow2010].[dbo].[Posts].Id, [StackOverflow2010].[dbo].[Posts].LastEditorUserId, [StackOverflow2010].[dbo].[Posts].OwnerUserId

Hash Keys Probe

[StackOverflow2010].[dbo].[Posts].LastEditorUserId

Probe Residual

[StackOverflow2010].[dbo].[Users].[Id] as [u].[Id]=[StackOverflow2010].[dbo].[Posts].[LastEditorUserId] as [p].[LastEditorUserId]



Misc

Actual Number of Rows for All Executions	1140863
CloseTime	20939
CompletionEstimate	1
⊞ Defined Values	
Description	Use each row from the top input to build a hash table, a
ElapsedTime	20939
Estimated CPU Cost	0.927653
Estimated Execution Mode	Batch
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows Per Execution	3729200
Estimated Operator Cost	0.927909 (0%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	63 B
Estimated Subtree Cost	593.902
⊞ Hash Keys Build	[StackOverflow2010].[dbo].[Users].Id
⊞ Hash Keys Probe	[StackOverflow2010].[dbo].[Posts].LastEditorUserId
Logical Operation	Inner Join
⊞ Memory Fractions	Memory Fractions Input: 1, Memory Fractions Output: 1
Node ID	1
Number of Executions	8
OpenTime	0
⊞ Output List	[StackOverflow2010].[dbo].[Users].DisplayName, [StackO
Parallel	True
Physical Operation	Hash Match
Probe Residual	[StackOverflow2010].[dbo].[Users].[Id] as [u].[Id]=[Stack
Status	FINISH

You’ve Fully Explained Everything, Right?

- But wait there’s more. You can find query plans other ways as well.
    - DMVs
    - Query store
    - Extended Events sessions
    - The monitoring tool of your choice.
- This session is based around the fact you know the query you want to tune, whether that was from your skills of proactive monitoring, user shouting, or the sudden realization a query is killing your server.

Common Operators

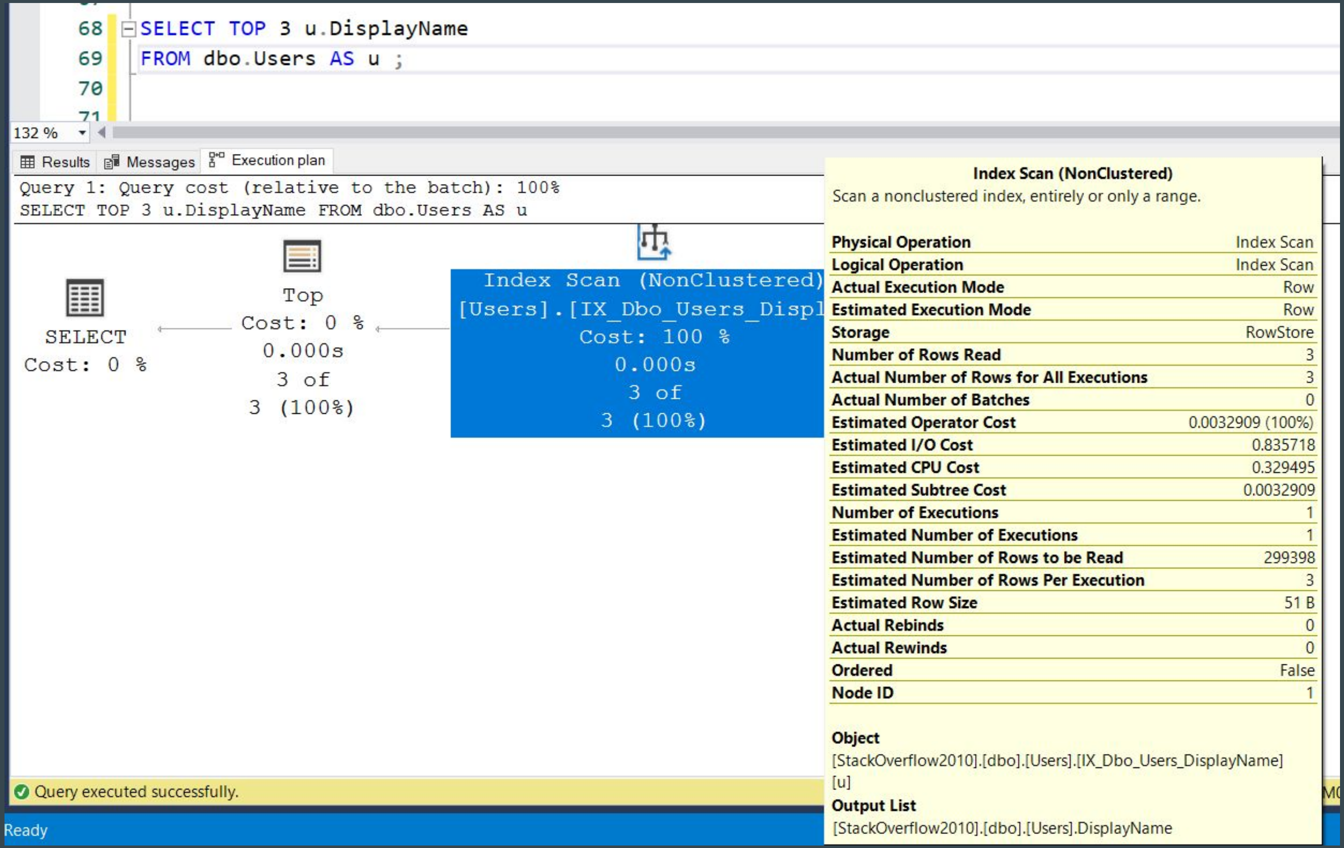
Why Are You Explaining Operators At Me?

- Since not everyone spends painful amounts of time looking at query plans, I’m going to do a crash course in some of the common operators.
- This will set the stage for looking at our query plans.
- This is far from comprehensive, go check out Hugo Kornelis’s SQLServerFast.com if you want comprehensive.

Index Scans

- As a general rule they start reading at one end of an index and continue until they have hit the end of the index
- These can also be used with TOP to make the end of data come when X number of rows have been returned.

A Non-Clustered Index Scan



```
SELECT TOP 3 u.DisplayName
FROM dbo.Users AS u ;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 3 u.DisplayName FROM dbo.Users AS u
```

Index Scan (NonClustered)

Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	3
Actual Number of Rows for All Executions	3
Actual Number of Batches	0
Estimated Operator Cost	0.0032909 (100%)
Estimated I/O Cost	0.835718
Estimated CPU Cost	0.329495
Estimated Subtree Cost	0.0032909
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows to be Read	299398
Estimated Number of Rows Per Execution	3
Estimated Row Size	51 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

Object

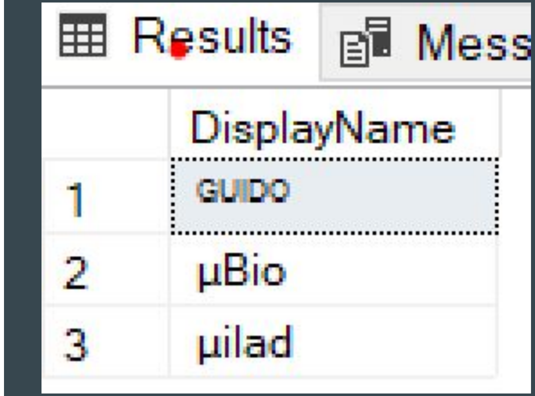
[StackOverflow2010].[dbo].[Users].[IX_Dbo_Users_DisplayName]
[u]

Output List

[StackOverflow2010].[dbo].[Users].DisplayName

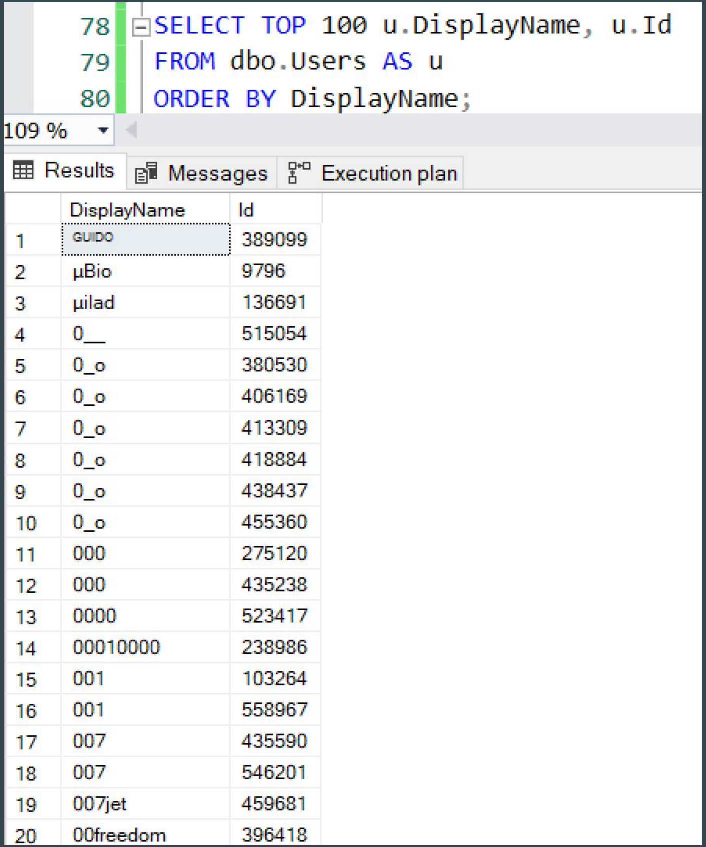
Query executed successfully.

Ready



	DisplayName
1	GUIDO
2	µBio
3	µilad

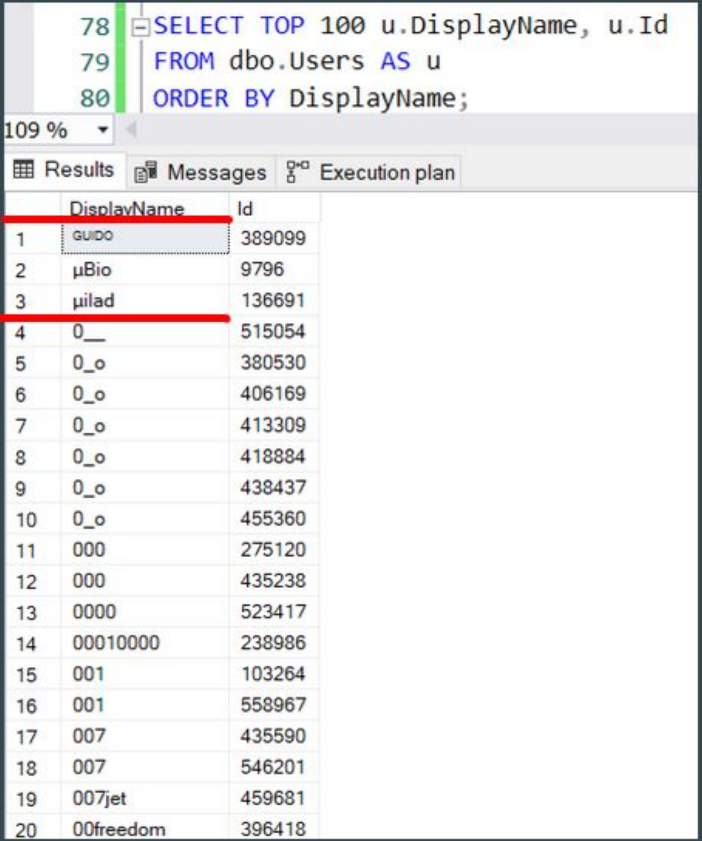
Visualizing The Index



```
SELECT TOP 100 u.DisplayName, u.Id
FROM dbo.Users AS u
ORDER BY DisplayName;
```

	DisplayName	Id
1	GUIDO	389099
2	µBio	9796
3	µilad	136691
4	0__	515054
5	0_o	380530
6	0_o	406169
7	0_o	413309
8	0_o	418884
9	0_o	438437
10	0_o	455360
11	000	275120
12	000	435238
13	0000	523417
14	00010000	238986
15	001	103264
16	001	558967
17	007	435590
18	007	546201
19	007jet	459681
20	00freedom	396418

Visualizing The Scan



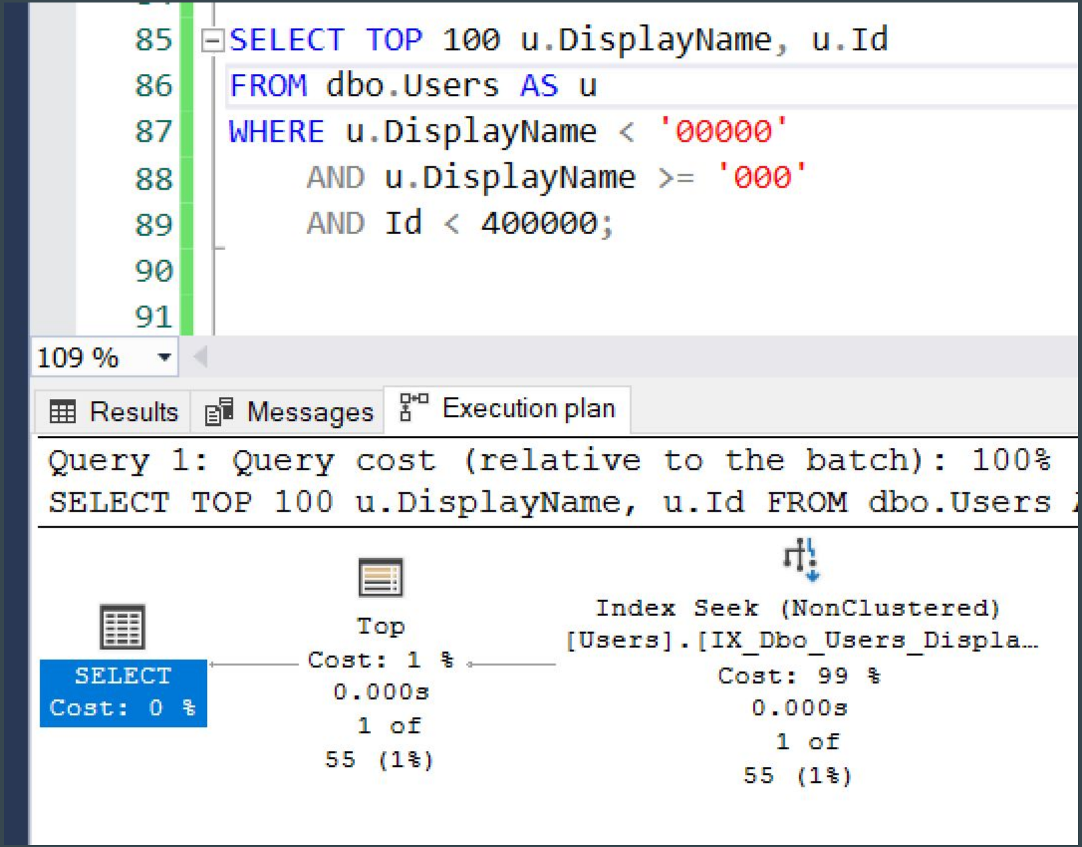
```
SELECT TOP 100 u.DisplayName, u.Id
FROM dbo.Users AS u
ORDER BY DisplayName;
```

	DisplayName	Id
1	GUIDO	389099
2	µBio	9796
3	µilad	136691
4	0__	515054
5	0_o	380530
6	0_o	406169
7	0_o	413309
8	0_o	418884
9	0_o	438437
10	0_o	455360
11	000	275120
12	000	435238
13	0000	523417
14	00010000	238986
15	001	103264
16	001	558967
17	007	435590
18	007	546201
19	007jet	459681
20	00freedom	396418

Index Seeks

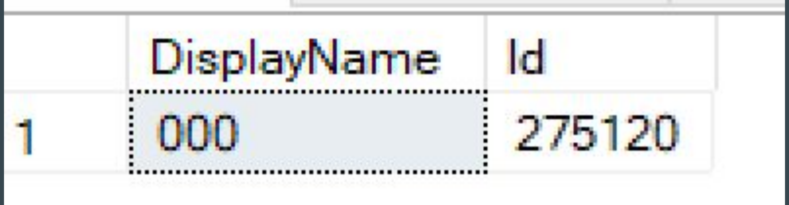
- Seeks effectively find single records or narrow ranges of data in an index.
- The optimizer navigates through the index down to the specific record or range of records needed.
- This is often used in conjunction with key lookups.
- If you do too many seeks it becomes less efficient than having just scanned an entire index once.

A Non-Clustered Index Seek



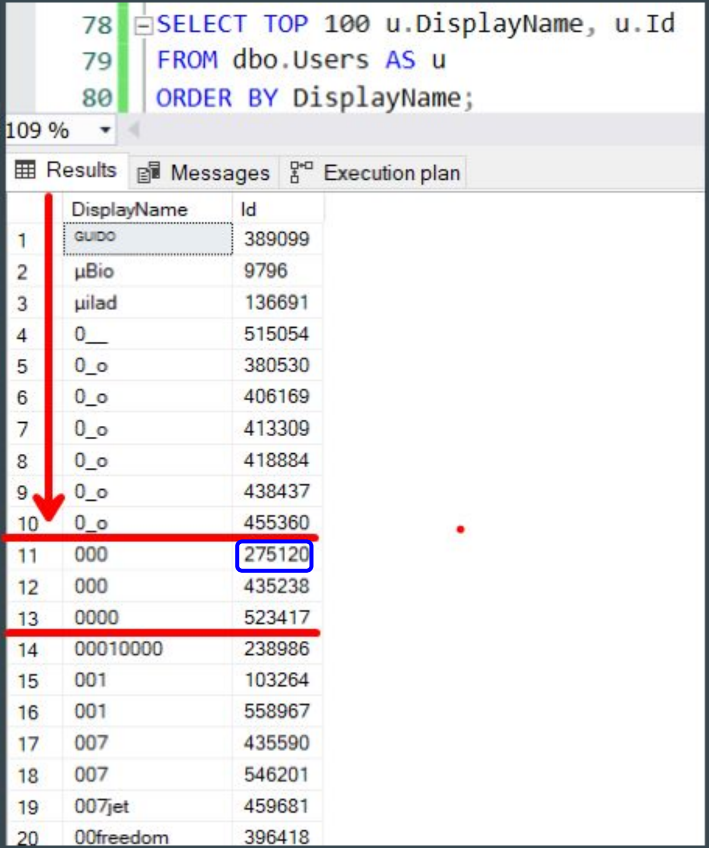
```sql
SELECT TOP 100 u.DisplayName, u.Id
FROM dbo.Users AS u
WHERE u.DisplayName < '00000'
    AND u.DisplayName >= '000'
    AND Id < 400000;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 u.DisplayName, u.Id FROM dbo.Users A
```



	DisplayName	Id
1	000	275120

Visualizing The Seek



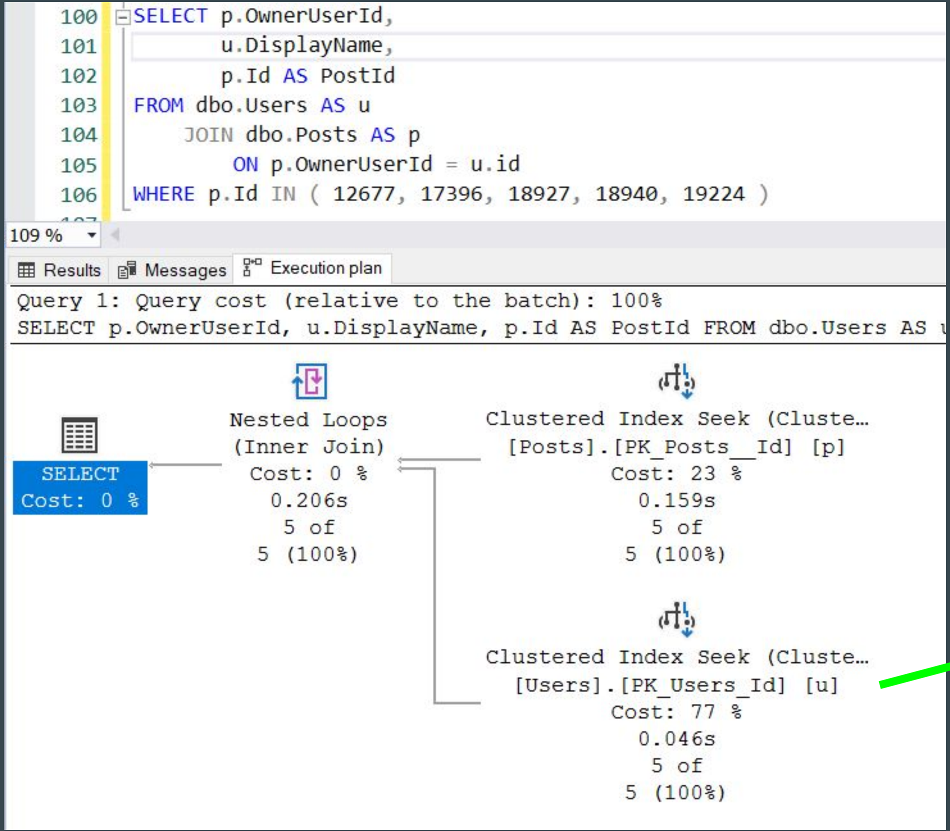
```sql
SELECT TOP 100 u.DisplayName, u.Id
FROM dbo.Users AS u
ORDER BY DisplayName;
```

	DisplayName	Id
1	GUIDO	389099
2	µBio	9796
3	µilad	136691
4	0__	515054
5	0_o	380530
6	0_o	406169
7	0_o	413309
8	0_o	418884
9	0_o	438437
10	0_o	455360
11	000	275120
12	000	435238
13	0000	523417
14	00010000	238986
15	001	103264
16	001	558967
17	007	435590
18	007	546201
19	007jet	459681
20	00freedom	396418

Nested Loop Joins

- Looks up the information for a join one row at a time.
- The nested loop join takes a row from the outer operator and then goes to the inner portion of the loop join and executes the operation(s) for the row passed in.
- This works best when there are relatively few rows from the outer operator and the inner operation has a low cost.
- Performance degrades when you have lots of rows to process or expensive inner operations.

Nested Loop Joins



```sql
SELECT p.OwnerUserId,
       u.DisplayName,
       p.Id AS PostId
FROM dbo.Users AS u
    JOIN dbo.Posts AS p
        ON p.OwnerUserId = u.id
WHERE p.Id IN ( 12677, 17396, 18927, 18940, 19224 )
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT p.OwnerUserId, u.DisplayName, p.Id AS PostId FROM dbo.Users AS u
```

SELECT
Cost: 0 %

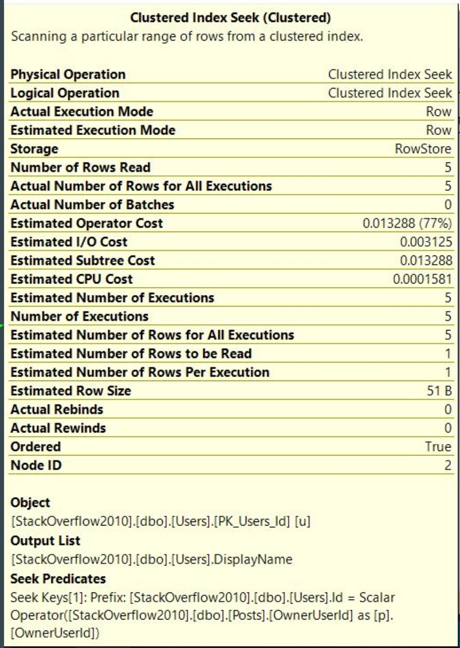
Nested Loops
(Inner Join)
Cost: 0 %
0.206s
5 of
5 (100%)

Clustered Index Seek (Cluste...
[Posts].[PK_Posts__Id] [p]
Cost: 23 %
0.159s
5 of
5 (100%)

Clustered Index Seek (Cluste...
[Users].[PK_Users_Id] [u]
Cost: 77 %
0.046s
5 of
5 (100%)

Outer Reference

Inner Reference



Clustered Index Seek (Clustered)

Scanning a particular range of rows from a clustered index.

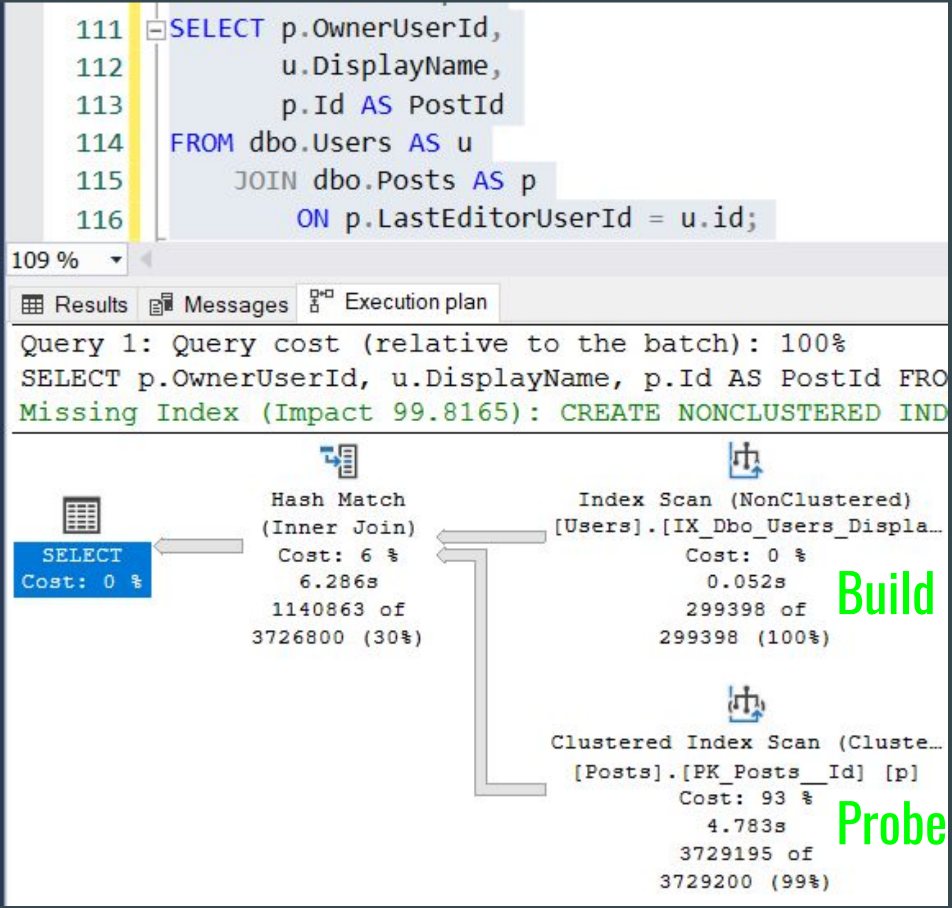
Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	5
Actual Number of Rows for All Executions	5
Actual Number of Batches	0
Estimated Operator Cost	0.013288 (77%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.013288
Estimated CPU Cost	0.0001581
Estimated Number of Executions	5
Number of Executions	5
Estimated Number of Rows for All Executions	5
Estimated Number of Rows to be Read	1
Estimated Number of Rows Per Execution	1
Estimated Row Size	51 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	2

Object
[StackOverflow2010].[dbo].[Users].[PK_Users_Id] [u]
Output List
[StackOverflow2010].[dbo].[Users].DisplayName
Seek Predicates
Seek Keys[1]: Prefix: [StackOverflow2010].[dbo].[Users].Id = Scalar Operator([StackOverflow2010].[dbo].[Posts].[OwnerUserId] as [p].[OwnerUserId])

Hash Joins

- These join together large amounts of unsorted data.
- They execute each of the operators for the join only once and the optimizer creates a table of hash values in memory based on the smaller set of data (the build input), creating buckets for each input row
- The optimizer then starts reading the other set of data (the probe input), converting the probe keys to hash values, and associating each record with the correct bucket in the hash table.
- These have the ability to write to tempdb when they run out of memory which means they'll just keep on trucking until they finish, it just may take forever.

Hash Joins

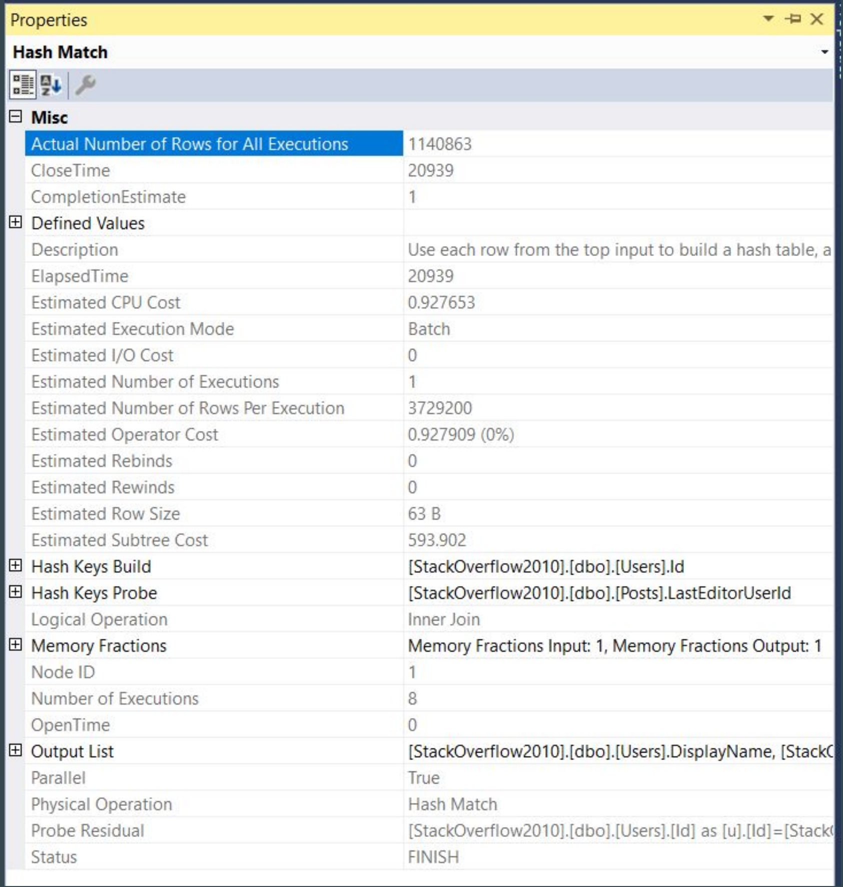


```sql
SELECT p.OwnerUserId,
       u.DisplayName,
       p.Id AS PostId
FROM dbo.Users AS u
    JOIN dbo.Posts AS p
      ON p.LastEditorUserId = u.id;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT p.OwnerUserId, u.DisplayName, p.Id AS PostId FRO
Missing Index (Impact 99.8165): CREATE NONCLUSTERED IND
```

Build

Probe

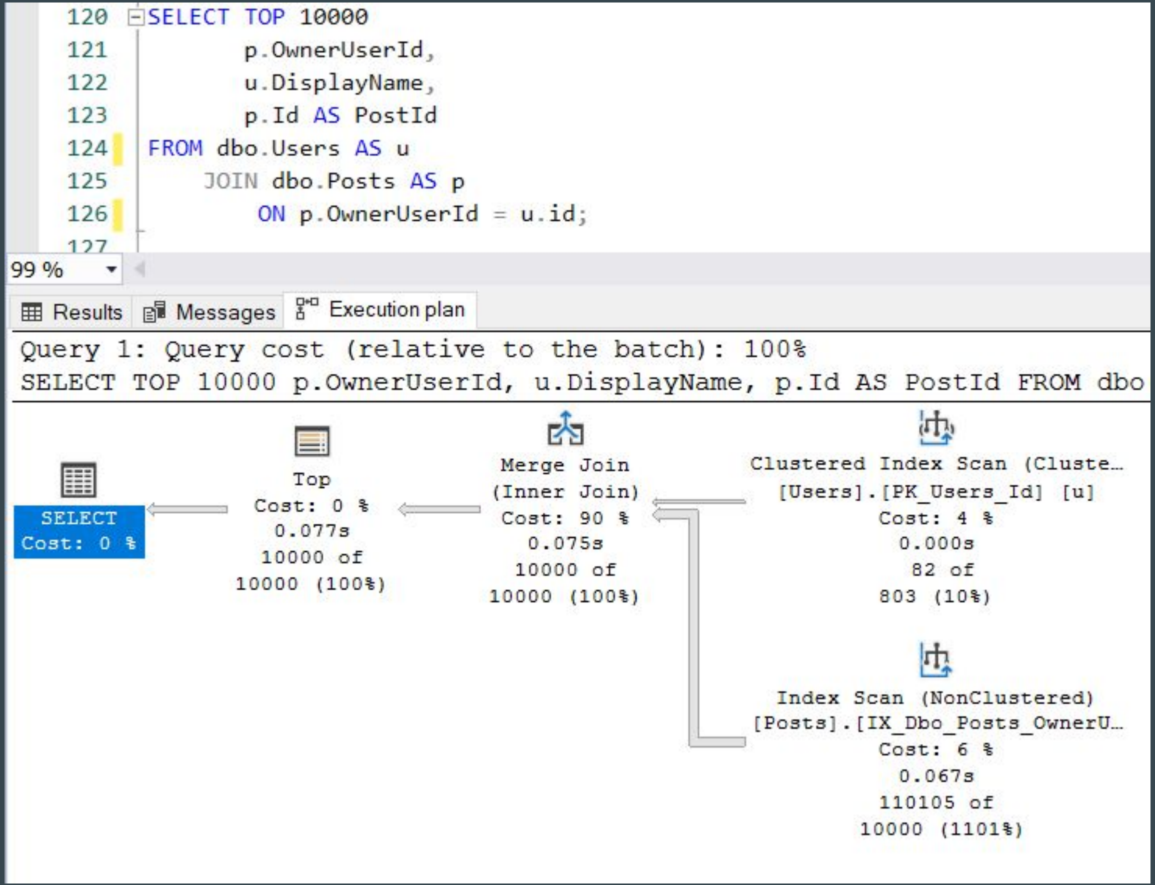


Misc	
Actual Number of Rows for All Executions	1140863
CloseTime	20939
CompletionEstimate	1
Defined Values	
Description	Use each row from the top input to build a hash table, a
ElapsedTime	20939
Estimated CPU Cost	0.927653
Estimated Execution Mode	Batch
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows Per Execution	3729200
Estimated Operator Cost	0.927909 (0%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	63 B
Estimated Subtree Cost	593.902
Hash Keys Build	[StackOverflow2010].[dbo].[Users].Id
Hash Keys Probe	[StackOverflow2010].[dbo].[Posts].LastEditorUserId
Logical Operation	Inner Join
Memory Fractions	Memory Fractions Input: 1, Memory Fractions Output: 1
Node ID	1
Number of Executions	8
OpenTime	0
Output List	[StackOverflow2010].[dbo].[Users].DisplayName, [StackO
Parallel	True
Physical Operation	Hash Match
Probe Residual	[StackOverflow2010].[dbo].[Users].[Id] as [u].[Id]=[Stack
Status	FINISH

Merge Joins

- These join together data that is (or can be) sorted for an equality join predicate.
- Both operators that feed in are executed only once.
- They read 1 row of data from each input and compare them, if they match then they are returned, if they do not match the smaller value is discarded and the join advances to the next value on that side.
- For a quick visualization, let's throw it over to Data with Bert (thanks Bert!)

Merge Joins



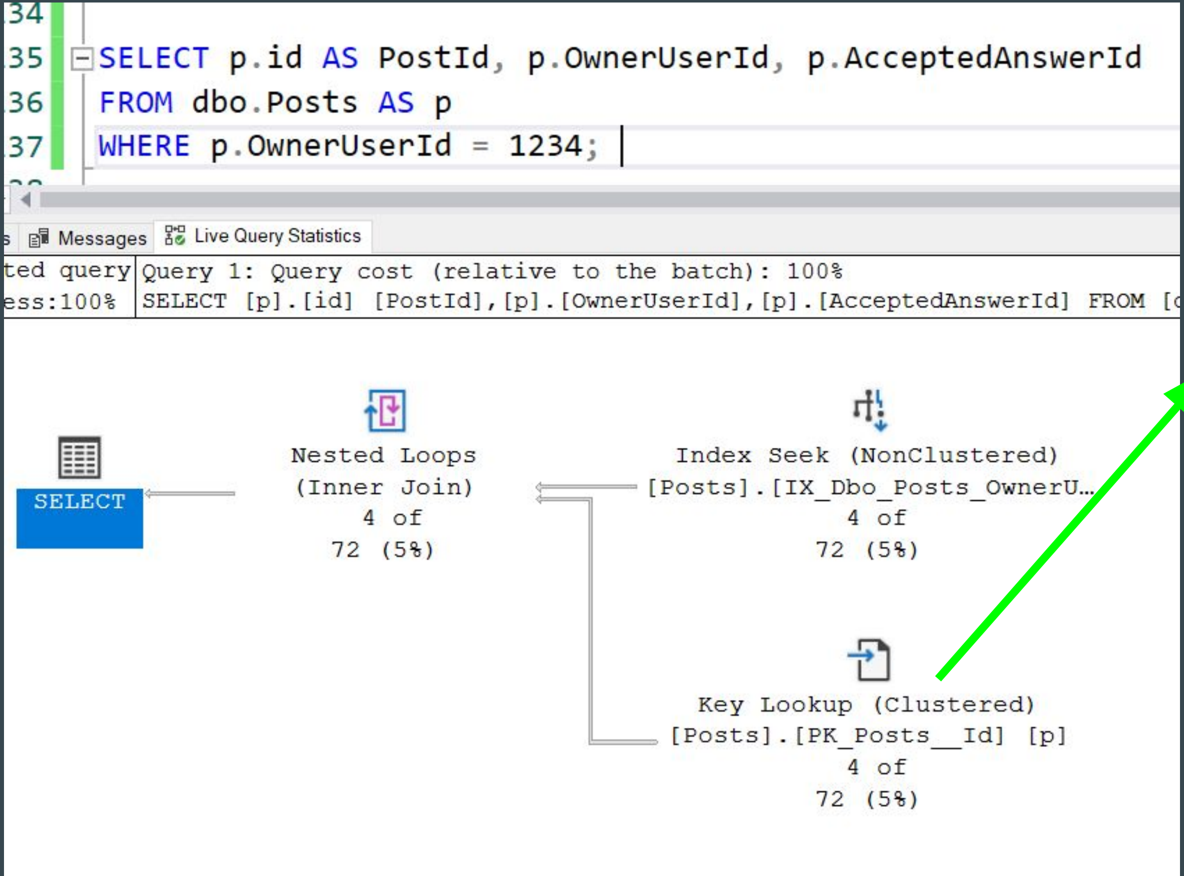
```sql
SELECT TOP 10000
       p.OwnerUserId,
       u.DisplayName,
       p.Id AS PostId
FROM dbo.Users AS u
    JOIN dbo.Posts AS p
        ON p.OwnerUserId = u.id;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP 10000 p.OwnerUserId, u.DisplayName, p.Id AS PostId FROM dbo
```

Key Lookups

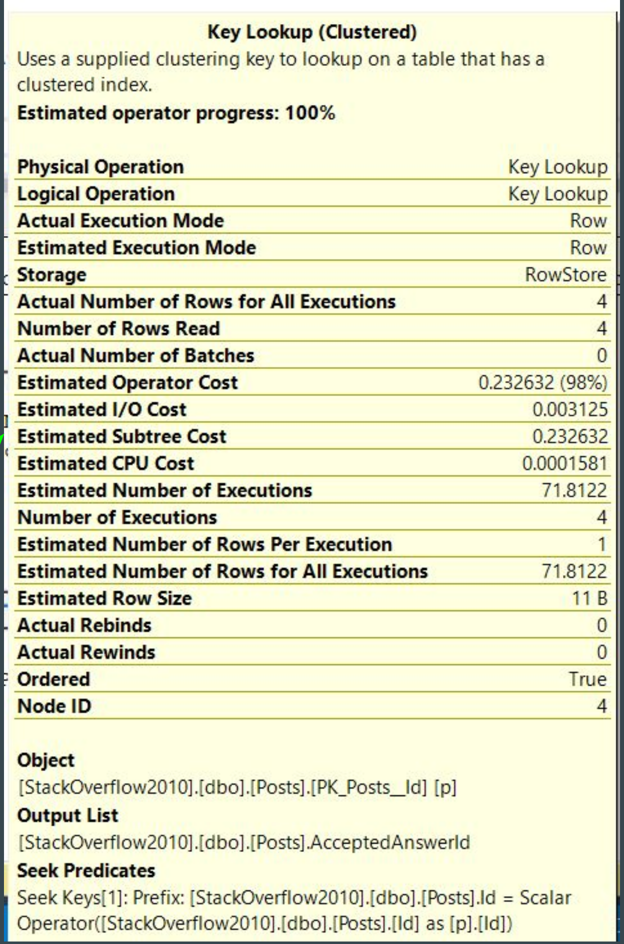
- Key lookups happen when SQL needs to look up fields that were not in a non-clustered index.
- Non-clustered indexes include a table’s clustering key (or a heap’s Row Id)
- Key lookups are effectively CI seeks, one for each row. These happen inside of a nested loop join.
- These can perform poorly when they execute a whole lot or when a filtering column can only be found in the key lookup.

Key Lookups



```sql
SELECT p.id AS PostId, p.OwnerUserId, p.AcceptedAnswerId
FROM dbo.Posts AS p
WHERE p.OwnerUserId = 1234;
```

```
ted query  Query 1: Query cost (relative to the batch): 100%
ess:100%   SELECT [p].[id] [PostId],[p].[OwnerUserId],[p].[AcceptedAnswerId] FROM [c
```



Key Lookup (Clustered)

Uses a supplied clustering key to lookup on a table that has a clustered index.

Estimated operator progress: 100%

Physical Operation	Key Lookup
Logical Operation	Key Lookup
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows for All Executions	4
Number of Rows Read	4
Actual Number of Batches	0
Estimated Operator Cost	0.232632 (98%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.232632
Estimated CPU Cost	0.0001581
Estimated Number of Executions	71.8122
Number of Executions	4
Estimated Number of Rows Per Execution	1
Estimated Number of Rows for All Executions	71.8122
Estimated Row Size	11 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	4

Object

[StackOverflow2010].[dbo].[Posts].[PK_Posts__Id] [p]

Output List

[StackOverflow2010].[dbo].[Posts].AcceptedAnswerId

Seek Predicates

Seek Keys[1]: Prefix: [StackOverflow2010].[dbo].[Posts].Id = Scalar Operator([StackOverflow2010].[dbo].[Posts].[Id] as [p].[Id])

Demo Setup

- All the demos are based around the 2010 copy of the StackOverflow database from Brent Ozar's site.
- SQL Server 2019
- Cost Threshold For Parallelism = 50
- All my examples are as simple as possible, what you find in the wild is likely to be much uglier.
- The majority of these were inspired by actual issues I have seen in production.

Example Time!

Thanks!

- To cope with the terrible things you'll see in tuning, try to remember people were doing the best they could with the knowledge and time they had.
- Slides and scripts can be found at https://github.com/JamesMRasch/RedFlags
- I can be found at JamesMRasch@gmail.com