Fundamentals of Index Tuning

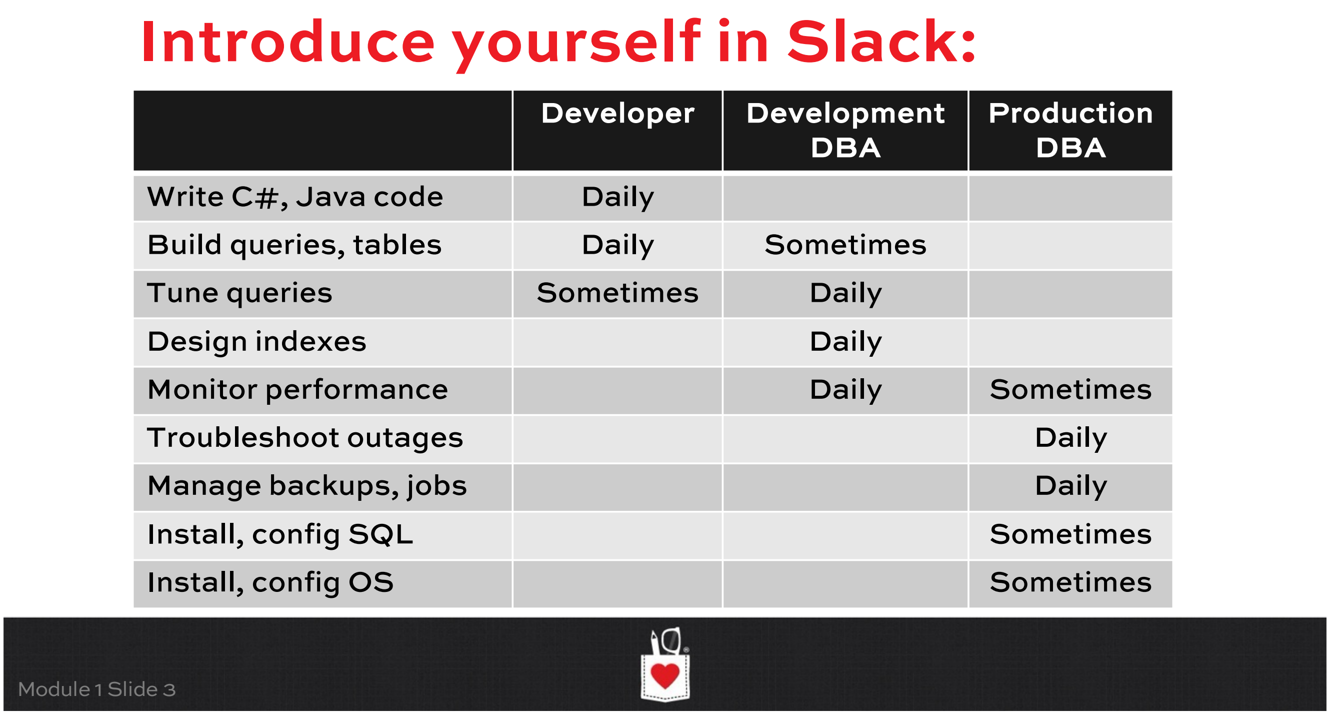
Up first: intros & logistics.

Module 1 Slide 1

99-05: dev, architect, DBA
05-08: DBA, VM, SAN admin
08-10: MCM, Quest Software
Since: consulting DBA

www.BrentOzar.com
Help@BrentOzar.com

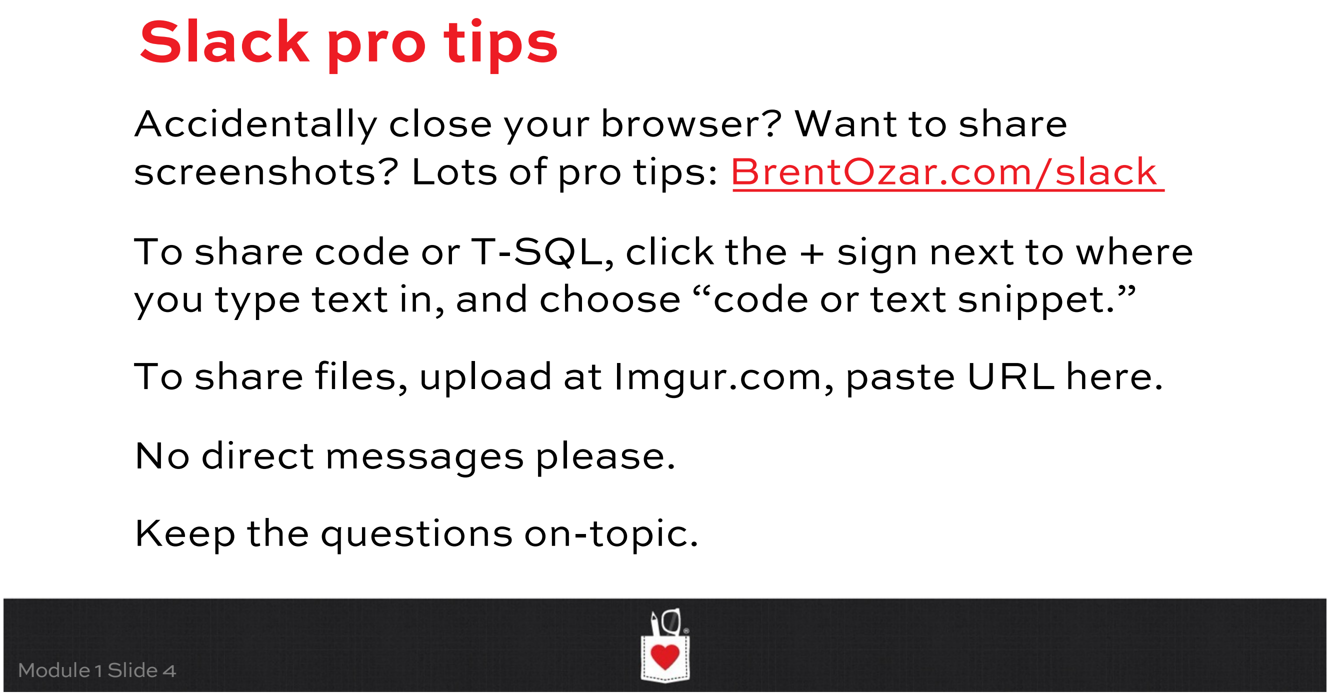
Module 1 Slide 2



Introduce yourself in Slack:

	Developer	Development DBA	Production DBA
Write C#, Java code	Daily		
Build queries, tables	Daily	Sometimes	
Tune queries	Sometimes	Daily	
Design indexes		Daily	
Monitor performance		Daily	Sometimes
Troubleshoot outages			Daily
Manage backups, jobs			Daily
Install, config SQL			Sometimes
Install, config OS			Sometimes

Module 1 Slide 3



Slack pro tips

Accidentally close your browser? Want to share screenshots? Lots of pro tips: BrentOzar.com/slack

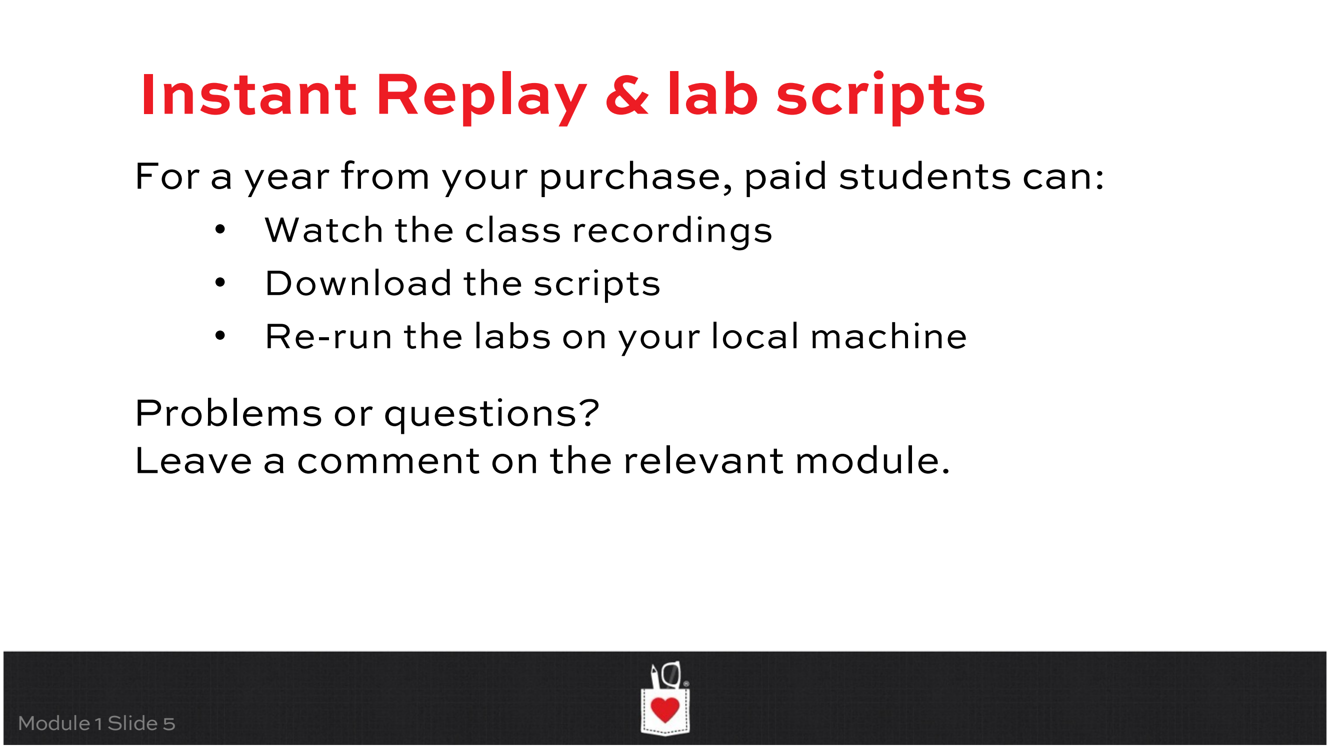
To share code or T-SQL, click the + sign next to where you type text in, and choose “code or text snippet.”

To share files, upload at Imgur.com, paste URL here.

No direct messages please.

Keep the questions on-topic.

Module 1 Slide 4

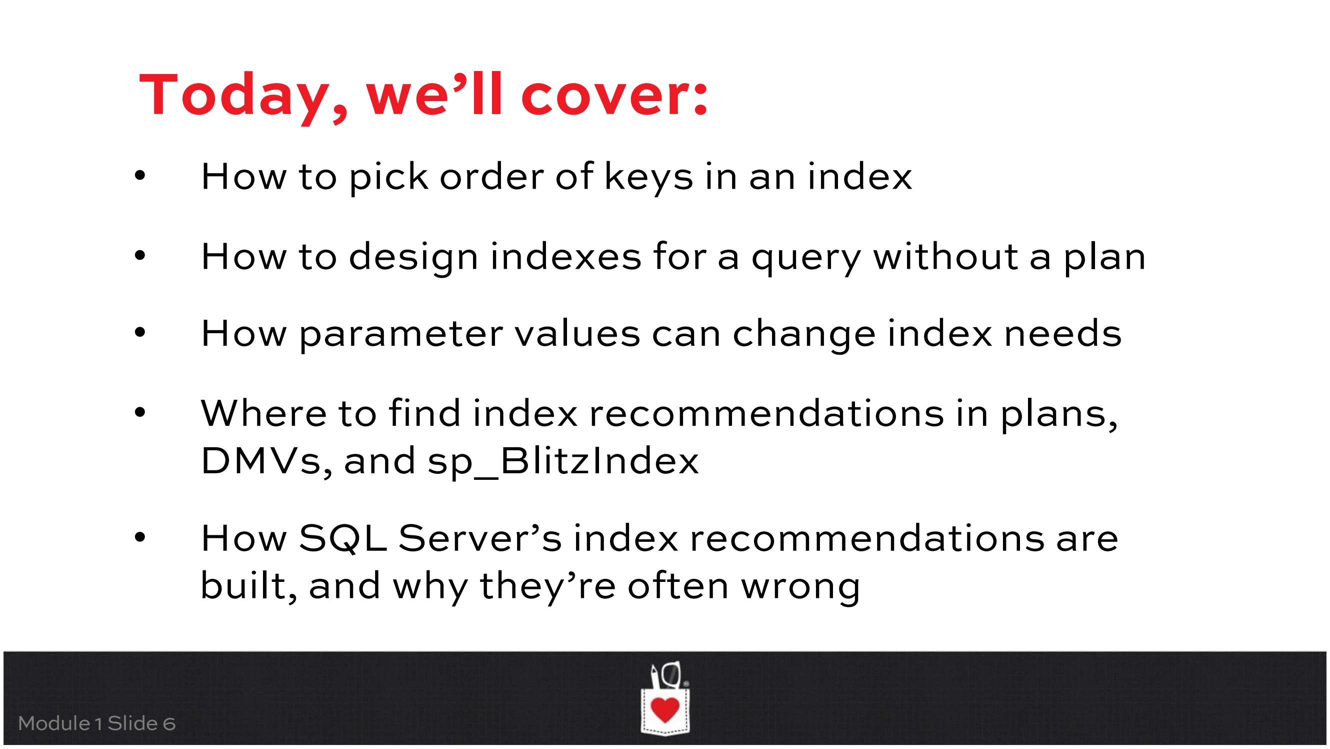


Instant Replay & lab scripts

For a year from your purchase, paid students can:

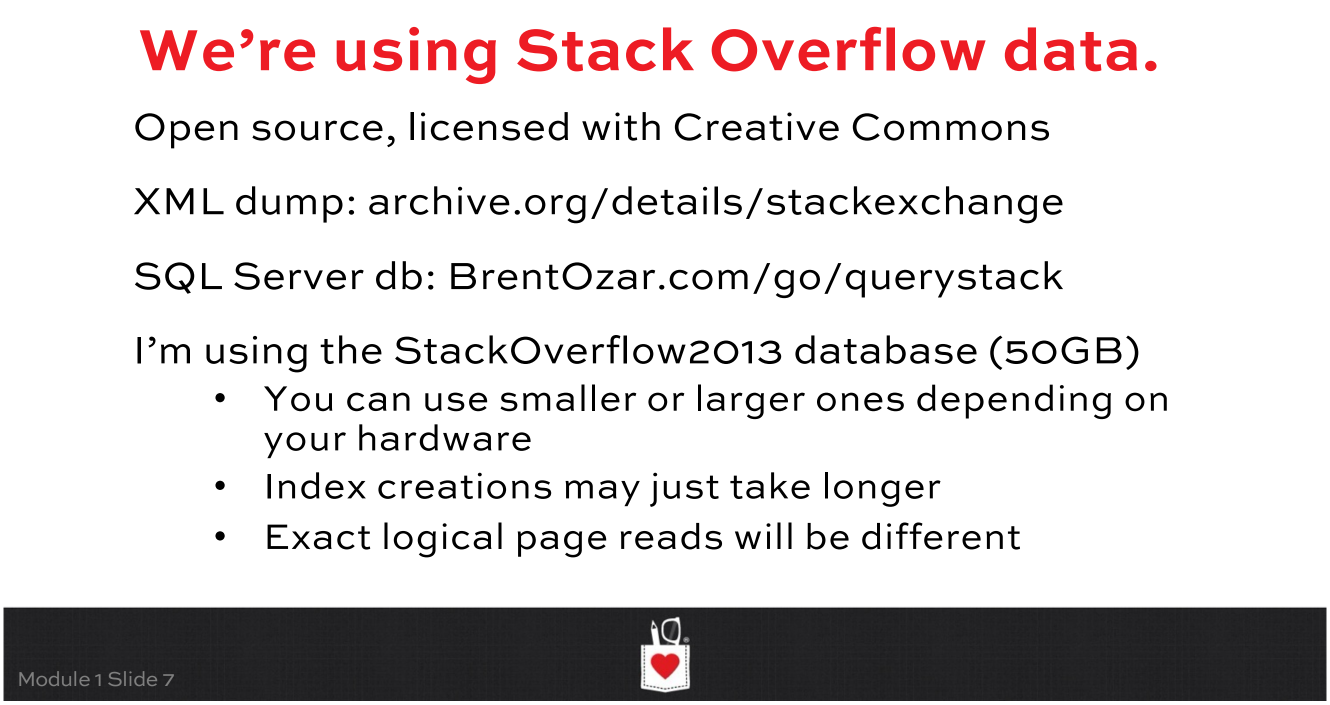
- Watch the class recordings
- Download the scripts
- Re-run the labs on your local machine

Problems or questions?
Leave a comment on the relevant module.



Today, we’ll cover:

- How to pick order of keys in an index
- How to design indexes for a query without a plan
- How parameter values can change index needs
- Where to find index recommendations in plans, DMVs, and sp_BlitzIndex
- How SQL Server’s index recommendations are built, and why they’re often wrong



We’re using Stack Overflow data.

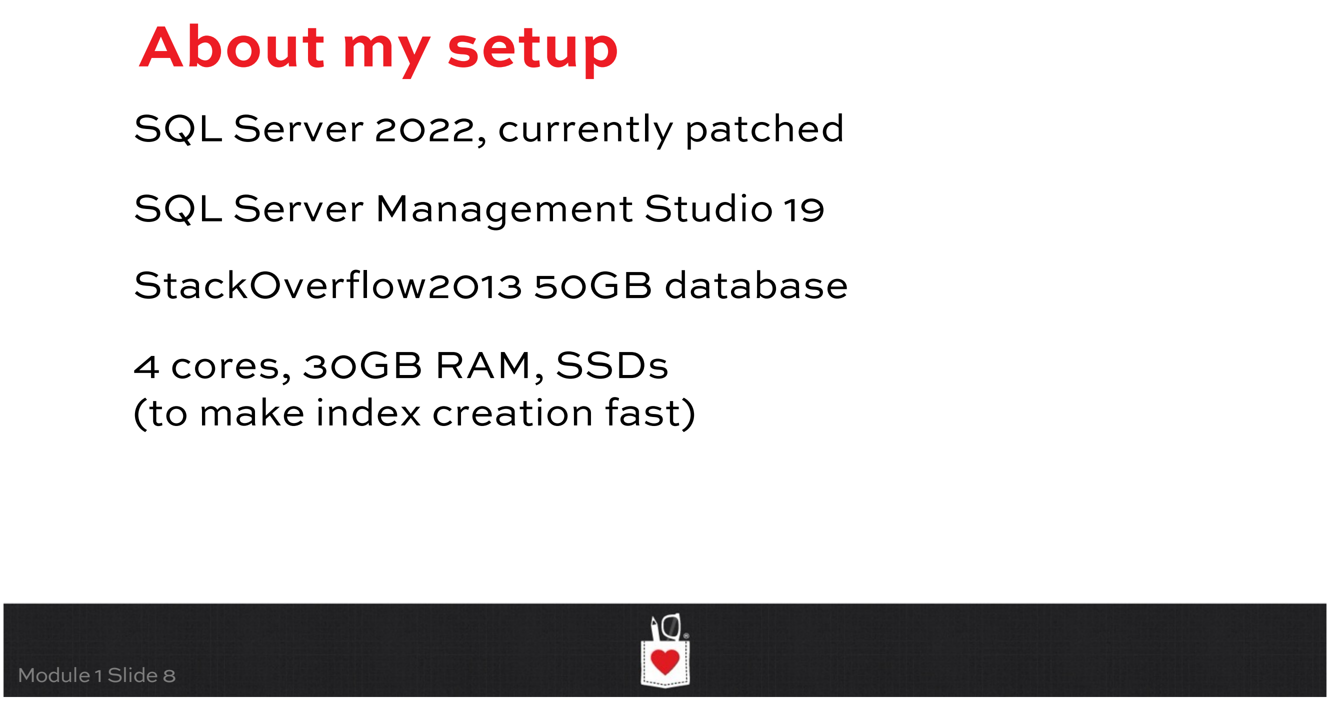
Open source, licensed with Creative Commons

XML dump: archive.org/details/stackexchange

SQL Server db: BrentOzar.com/go/querystack

I’m using the StackOverflow2013 database (50GB)

- You can use smaller or larger ones depending on your hardware
- Index creations may just take longer
- Exact logical page reads will be different



About my setup

SQL Server 2022, currently patched

SQL Server Management Studio 19

StackOverflow2013 50GB database

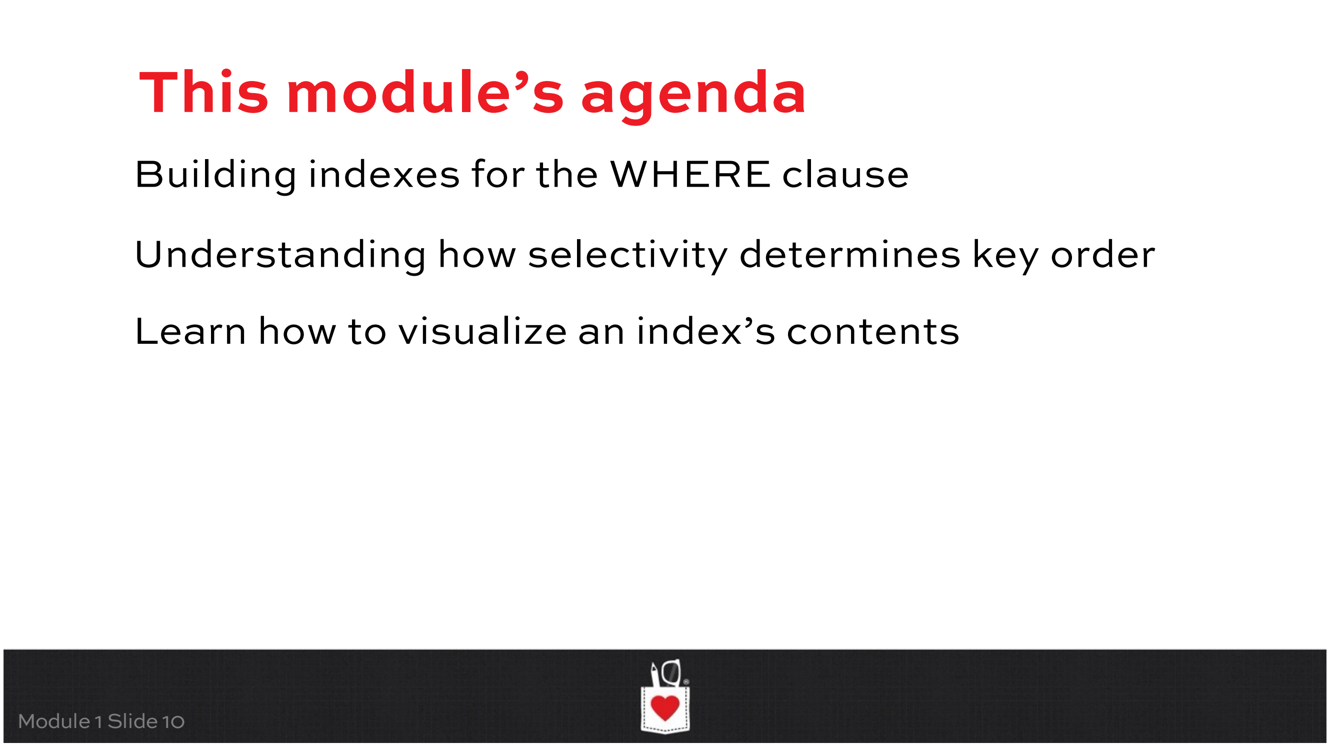
4 cores, 30GB RAM, SSDs
(to make index creation fast)

BRENT OZAR UNLIMITED®

Fundamentals of Index Tuning

Part 1: The WHERE Clause

Module 1 Slide 9



This module's agenda

Building indexes for the WHERE clause

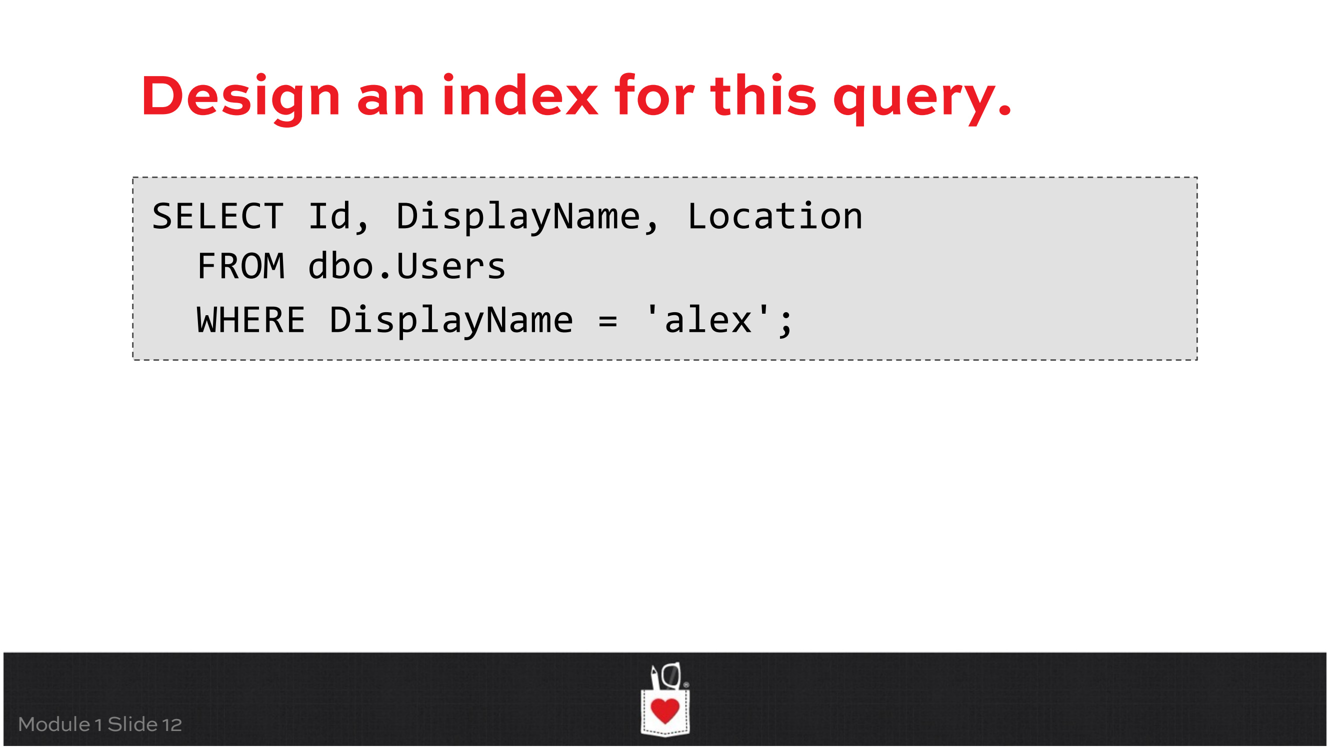
Understanding how selectivity determines key order

Learn how to visualize an index's contents

Module 1 Slide 10

WHERE
with 1 equality search

Module 1 Slide 11

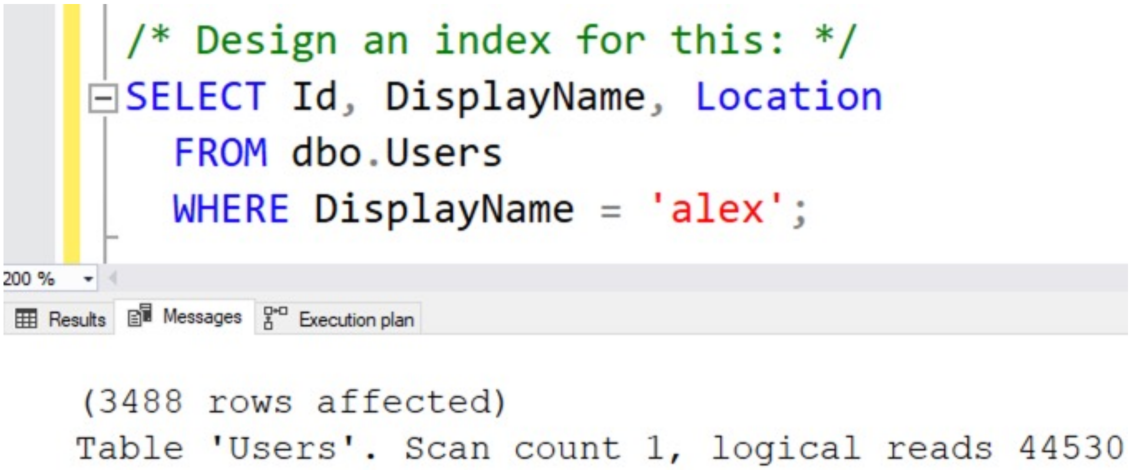


Design an index for this query.

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```

Module 1 Slide 12

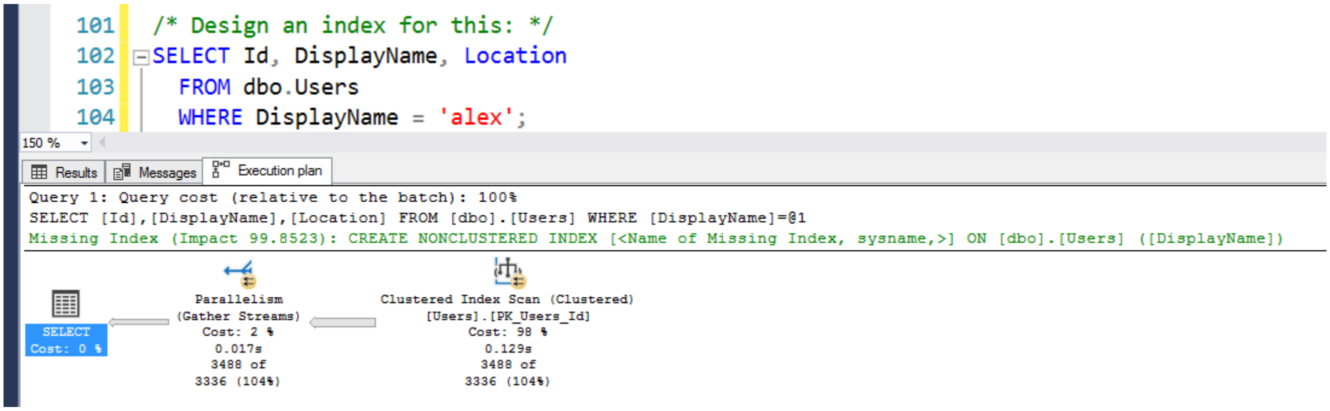
Check your logical reads before…



```
/* Design an index for this: */
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```

```
(3488 rows affected)
Table 'Users'. Scan count 1, logical reads 44530
```

Clippy has an idea:



```
/* Design an index for this: */
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT [Id],[DisplayName],[Location] FROM [dbo].[Users] WHERE [DisplayName]=@1
Missing Index (Impact 99.8523): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([DisplayName])
```

I'm not cutting that screenshot off, either:
Clippy didn't suggest that we include Location.

Clippy’s hints are a gift.

But they’re just a lucky byproduct of query plan optimization.

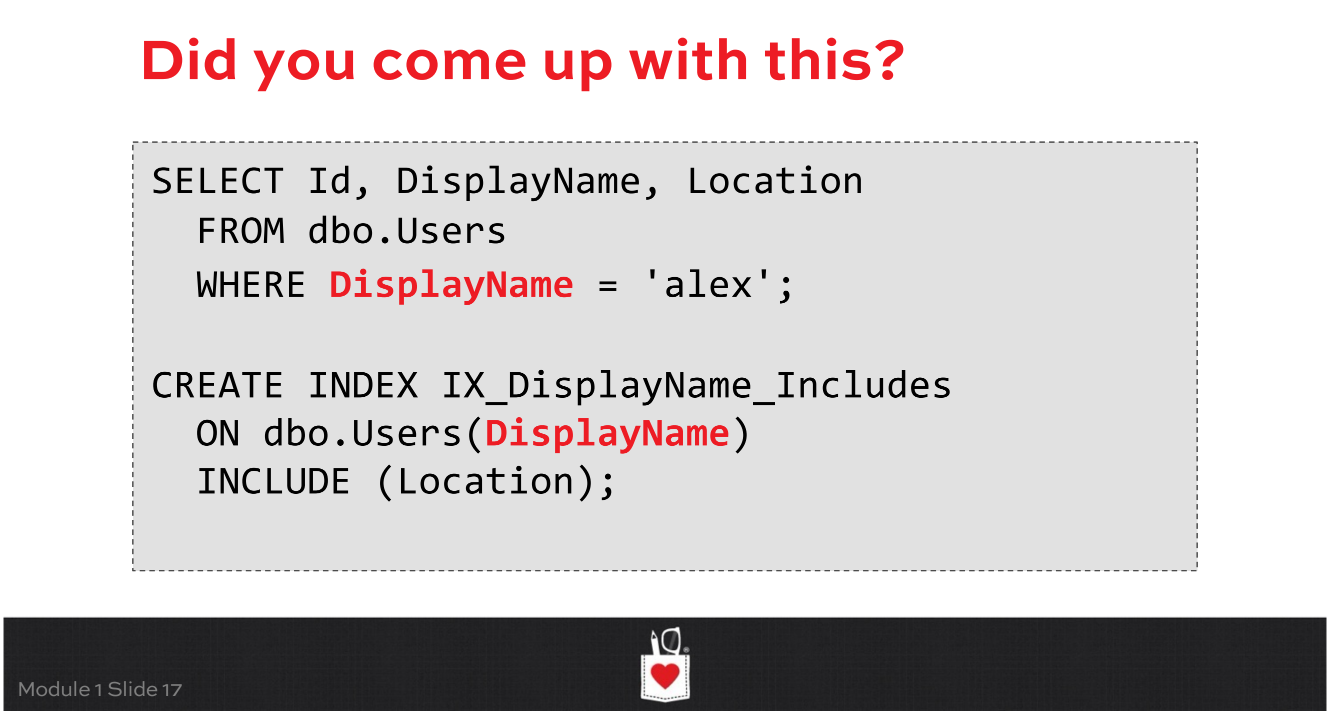
They’re not Clippy’s main job.

Later today, we’ll explain how he builds them and why they’re often wrong.

For now, focus on building your own.

I bet you could do better. Try it.

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```

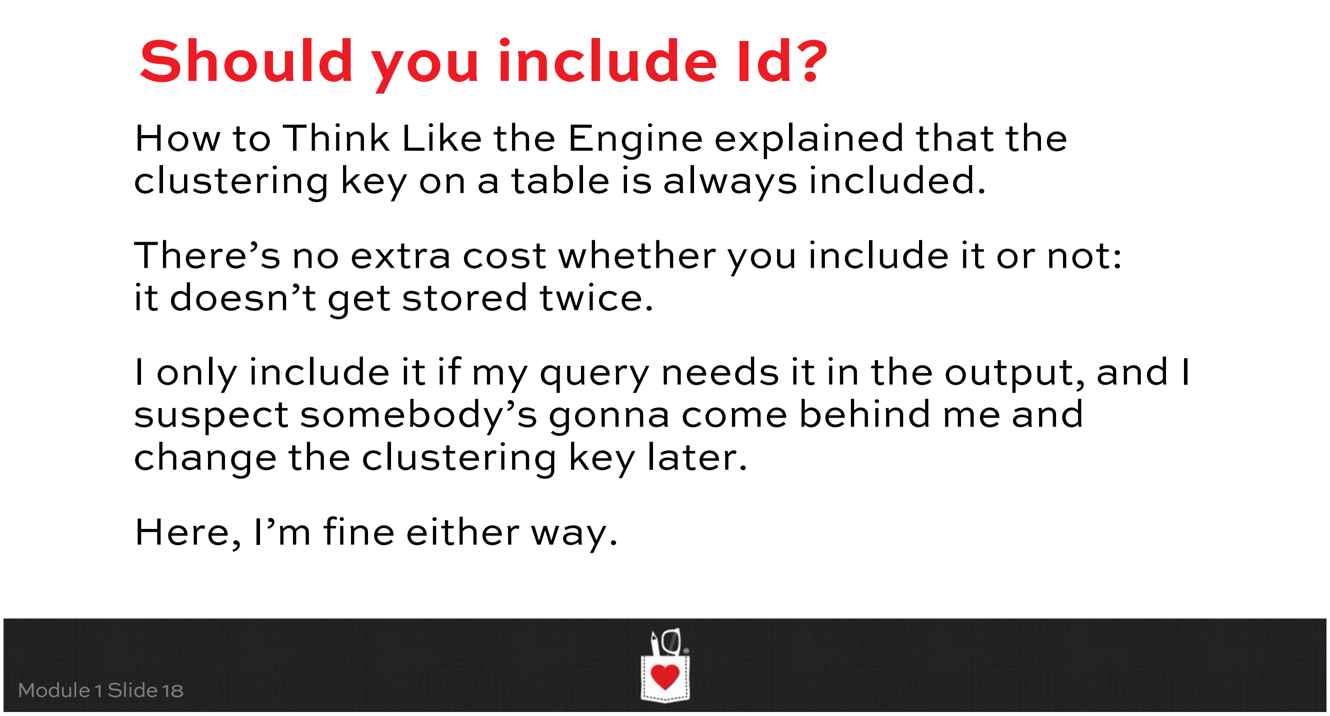


Did you come up with this?

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';

CREATE INDEX IX_DisplayName_Includes
  ON dbo.Users(DisplayName)
  INCLUDE (Location);
```

Module 1 Slide 17



Should you include Id?

How to Think Like the Engine explained that the clustering key on a table is always included.

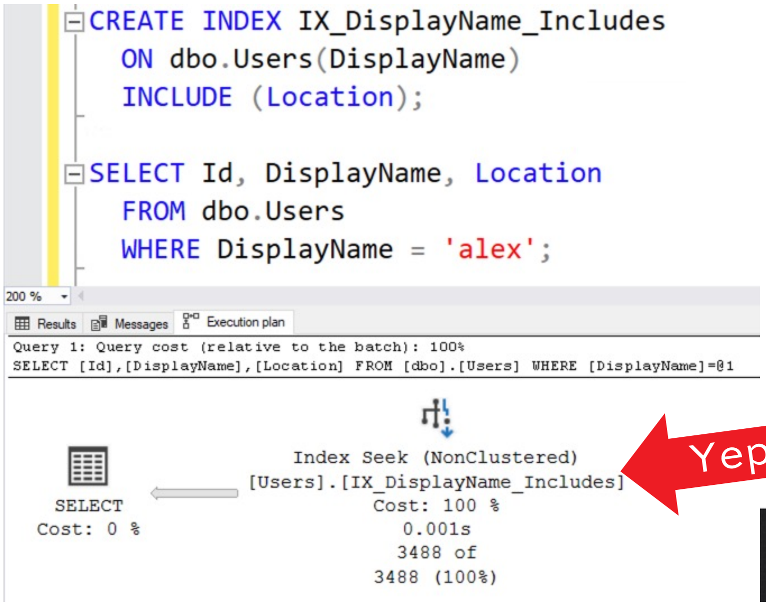
There's no extra cost whether you include it or not: it doesn't get stored twice.

I only include it if my query needs it in the output, and I suspect somebody's gonna come behind me and change the clustering key later.

Here, I'm fine either way.

Module 1 Slide 18

Create the index. Does it get used?



```
CREATE INDEX IX_DisplayName_Includes
  ON dbo.Users(DisplayName)
  INCLUDE (Location);

SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```

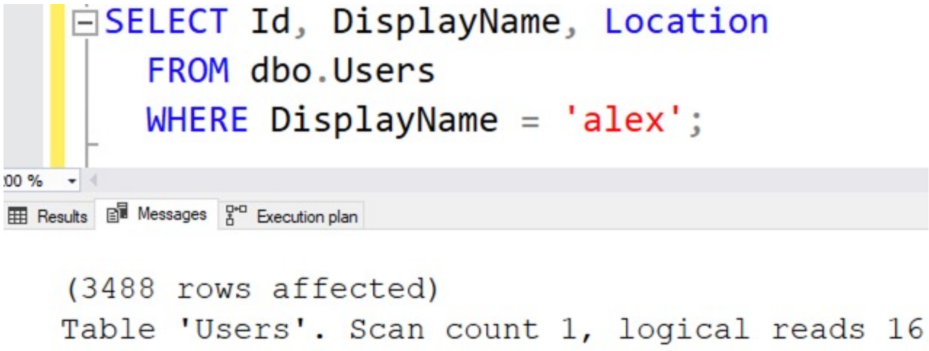
```
Query 1: Query cost (relative to the batch): 100%
SELECT [Id],[DisplayName],[Location] FROM [dbo].[Users] WHERE [DisplayName]=@1
```

Yep!

Does it do less logical reads?

Before: 44,530 logical page reads

Now, with the index:



```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```

```
(3488 rows affected)
Table 'Users'. Scan count 1, logical reads 16
```

Great!

Module 1 Slide 20

WHERE
with 2 equality searches

Module 1 Slide 21

Now try this query.

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location = 'Seattle, WA';
```

Module 1 Slide 22

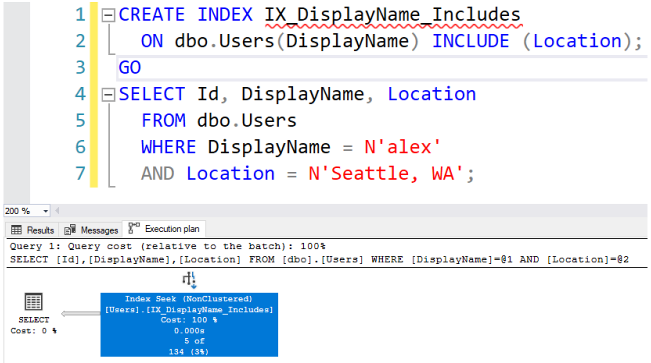
A couple of common solutions

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location = 'Seattle, WA';

CREATE INDEX IX_DisplayName_Location
  ON dbo.Users(DisplayName, Location);

CREATE INDEX IX_Location_DisplayName
  ON dbo.Users(Location, DisplayName);
```

But remember that last index?



```
CREATE INDEX IX_DisplayName_Includes
  ON dbo.Users(DisplayName) INCLUDE (Location);
GO
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = N'alex'
  AND Location = N'Seattle, WA';
```

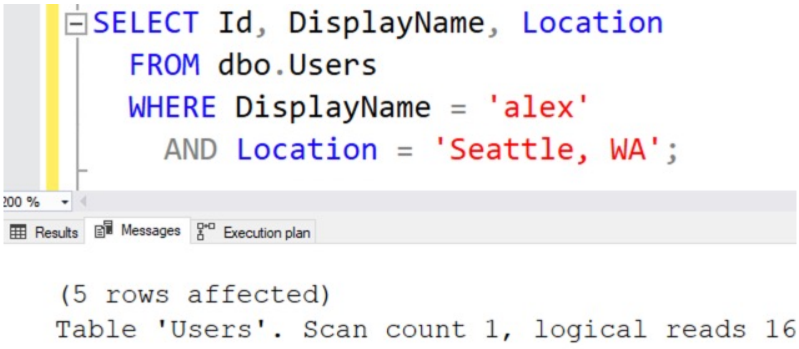
```
Query 1: Query cost (relative to the batch): 100%
SELECT [Id],[DisplayName],[Location] FROM [dbo].[Users] WHERE [DisplayName]=@1 AND [Location]=@2
```

SQL Server can use that last index we created.

Sure, Location isn’t sorted in order, but we can still:

1. Seek to Alex
2. Scan through them, checking their locations

How many reads does it do?



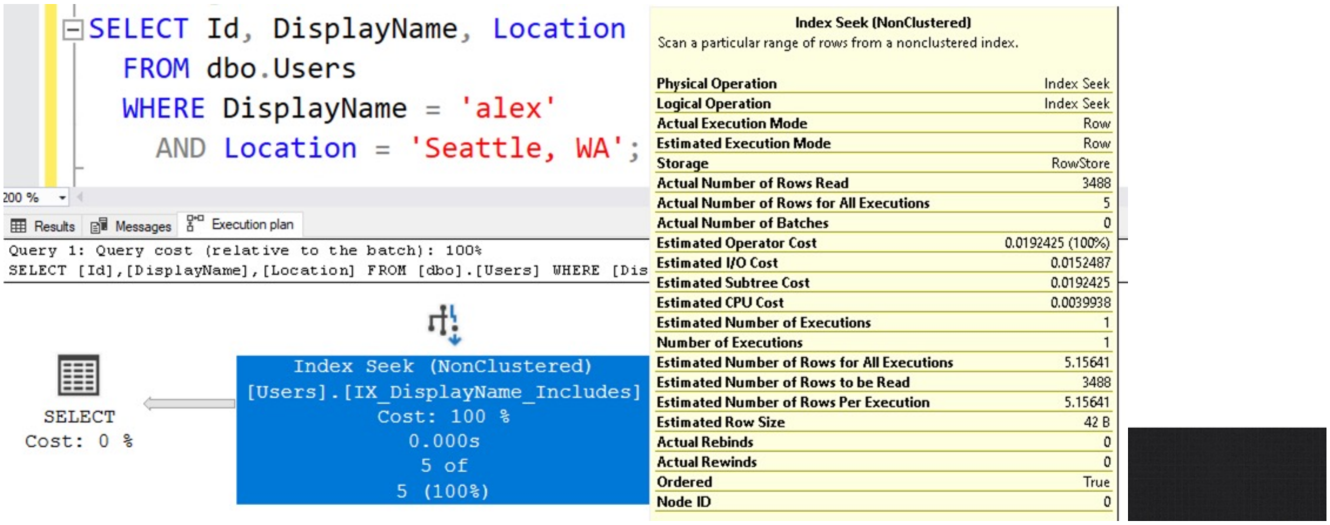
```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location = 'Seattle, WA';
```

```
(5 rows affected)
Table 'Users'. Scan count 1, logical reads 16
```

Just 16 8KB pages! That’s pretty good.

Could it be even better? Sure.

Hover your mouse over the Index Seek in this plan:



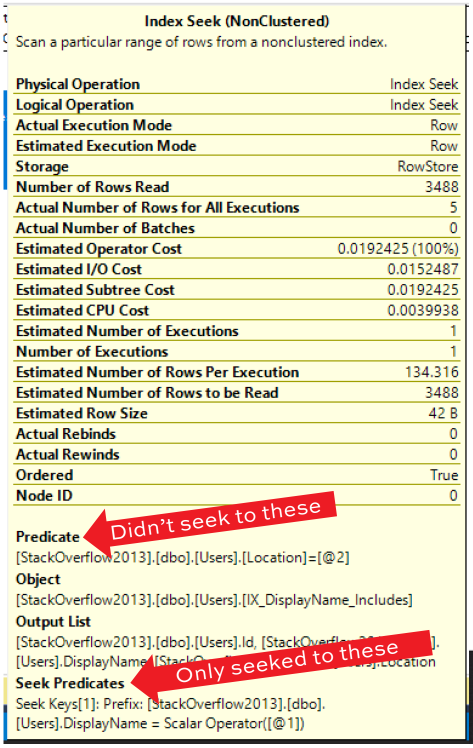
```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location = 'Seattle, WA';
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT [Id],[DisplayName],[Location] FROM [dbo].[Users] WHERE [Dis
```

Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows Read	3488
Actual Number of Rows for All Executions	5
Actual Number of Batches	0
Estimated Operator Cost	0.0192425 (100%)
Estimated I/O Cost	0.0152487
Estimated Subtree Cost	0.0192425
Estimated CPU Cost	0.0039938
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	5.15641
Estimated Number of Rows to be Read	3488
Estimated Number of Rows Per Execution	5.15641
Estimated Row Size	42 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	3488
Actual Number of Rows for All Executions	5
Actual Number of Batches	0
Estimated Operator Cost	0.0192425 (100%)
Estimated I/O Cost	0.0152487
Estimated Subtree Cost	0.0192425
Estimated CPU Cost	0.0039938
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows Per Execution	134.316
Estimated Number of Rows to be Read	3488
Estimated Row Size	42 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0

Predicate

Didn't seek to these

[StackOverflow2013].[dbo].[Users].[Location]=[@2]

Object

[StackOverflow2013].[dbo].[Users].[IX_DisplayName_Includes]

Output List

[StackOverflow2013].[dbo].[Users].Id, [StackOverflo].
[Users].DisplayName, [StackO].Location

Only seeked to these

Seek Predicates

Seek Keys[1]: Prefix: [StackOverflow2013].[dbo].
[Users].DisplayName = Scalar Operator([@1])

Decoding the popup

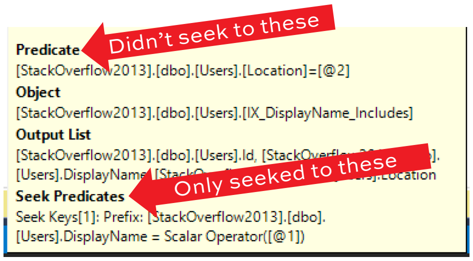
Seek predicate on DisplayName: we're able to jump straight to the Alexes.

Predicate on Location: but note that it doesn't have the word "seek" in front of it. This is called a residual predicate.

What "index seek" really means

SQL Server seeked to a specific value
on the index's first column.

It doesn't necessarily mean we seeked
on subsequent columns: we may have scanned them.



Predicate

Didn't seek to these

[StackOverflow2013].[dbo].[Users].[Location]=[@2]

Object

[StackOverflow2013].[dbo].[Users].[IX_DisplayName_Includes]

Output List

[StackOverflow2013].[dbo].[Users].Id, [StackOverflo].
[Users].DisplayName, [StackO].Location

Only seeked to these

Seek Predicates

Seek Keys[1]: Prefix: [StackOverflow2013].[dbo].
[Users].DisplayName = Scalar Operator([@1])

To find out why, visualize the index

Write a query to exactly show what's inside this index:

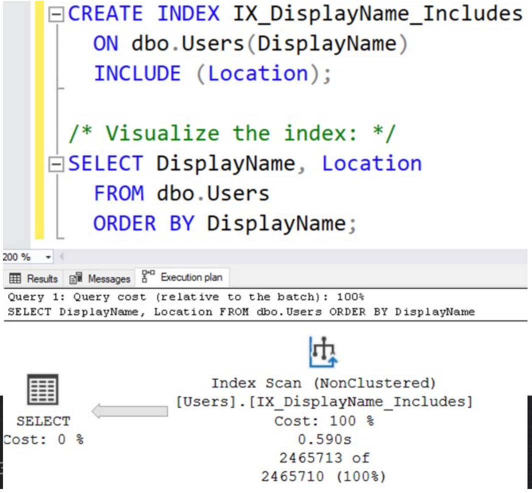
- How it's ordered: ORDER BY
- What columns are on it: INCLUDE

```sql
CREATE INDEX IX_DisplayName_Includes
  ON dbo.Users(DisplayName)
  INCLUDE (Location);

SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```

If the visualization query is right

Then the visualization query's plan will ONLY have an index scan on that index – no other indexes, no sorts:



```sql
CREATE INDEX IX_DisplayName_Includes
  ON dbo.Users(DisplayName)
  INCLUDE (Location);

/* Visualize the index: */
SELECT DisplayName, Location
  FROM dbo.Users
  ORDER BY DisplayName;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT DisplayName, Location FROM dbo.Users ORDER BY DisplayName
```

Gotchas with this technique

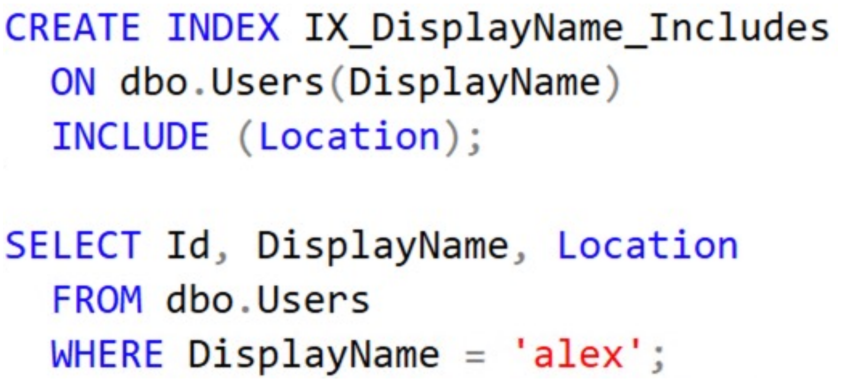
You may have to use an estimated plan:
if your table is big, a select w/o a where can take days

You can use a WHERE to get results faster,
but only on the first column in the index!

If you use a WHERE on the first column,
your plan will be an index seek,
but only a seek: no sorts. If you see sorts,
the visualization query isn't right.

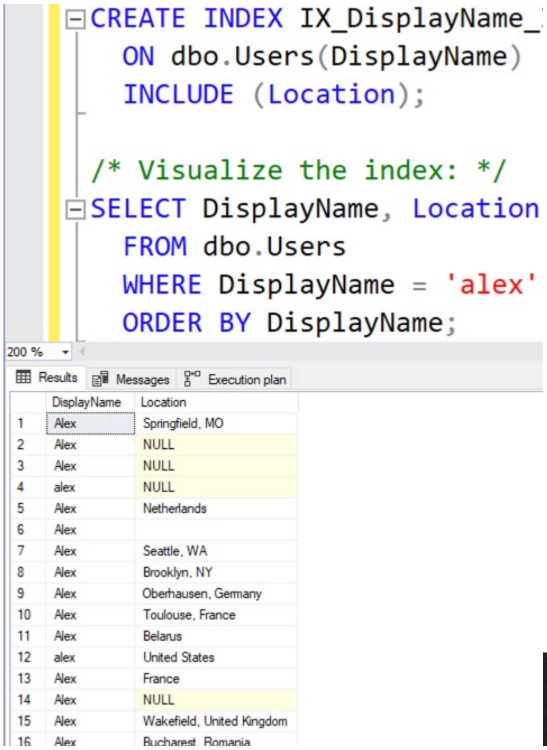
Testing our query & index

We're trying to see why this query has a residual
(scan) predicate with our index:



```
CREATE INDEX IX_DisplayName_Includes
  ON dbo.Users(DisplayName)
  INCLUDE (Location);

SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```



```
CREATE INDEX IX_DisplayName_
  ON dbo.Users(DisplayName)
  INCLUDE (Location);

/* Visualize the index: */
SELECT DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
  ORDER BY DisplayName;
```

	DisplayName	Location
1	Alex	Springfield, MO
2	Alex	NULL
3	Alex	NULL
4	alex	NULL
5	Alex	Netherlands
6	Alex	
7	Alex	Seattle, WA
8	Alex	Brooklyn, NY
9	Alex	Oberhausen, Germany
10	Alex	Toulouse, France
11	Alex	Belarus
12	alex	United States
13	Alex	France
14	Alex	NULL
15	Alex	Wakefield, United Kingdom
16	Alex	Bucharest, Romania

Visualize the index

The index is sorted on DisplayName, and Location is only included.

It’s not sorted by Location.

So we can seek to Alex, but then we have to read all Alexes in all locations.

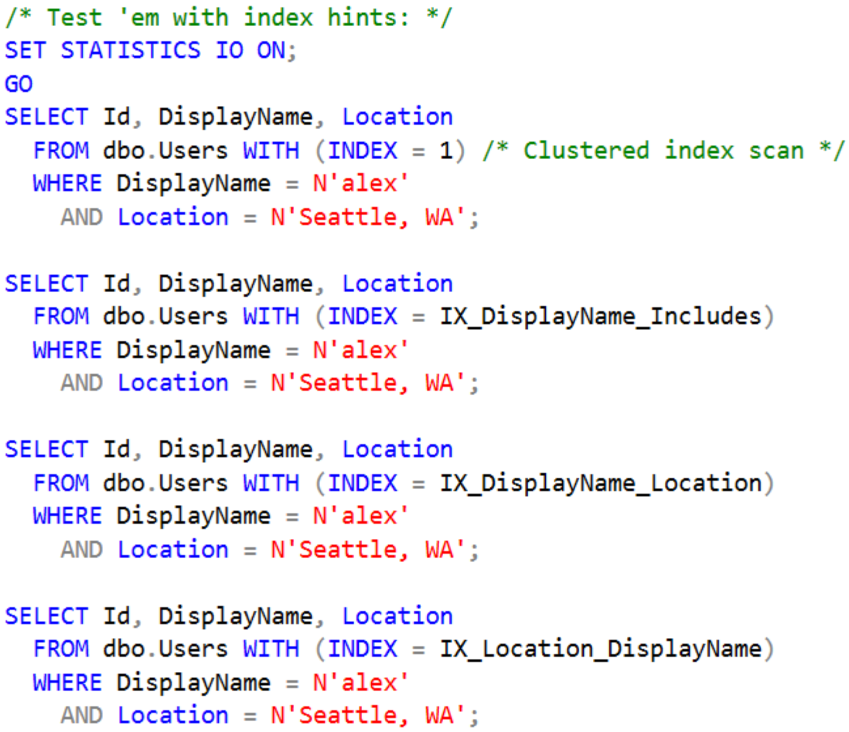
That’s why we had 16 reads even with few Alexes in Seattle.

Maybe your indexes are better. Let’s create ‘em and try ‘em.

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location = 'Seattle, WA';

CREATE INDEX IX_DisplayName_Location
  ON dbo.Users(DisplayName, Location);

CREATE INDEX IX_Location_DisplayName
  ON dbo.Users(Location, DisplayName);
```



```
/* Test 'em with index hints: */
SET STATISTICS IO ON;
GO
SELECT Id, DisplayName, Location
  FROM dbo.Users WITH (INDEX = 1) /* Clustered index scan */
  WHERE DisplayName = N'alex'
    AND Location = N'Seattle, WA';

SELECT Id, DisplayName, Location
  FROM dbo.Users WITH (INDEX = IX_DisplayName_Includes)
  WHERE DisplayName = N'alex'
    AND Location = N'Seattle, WA';

SELECT Id, DisplayName, Location
  FROM dbo.Users WITH (INDEX = IX_DisplayName_Location)
  WHERE DisplayName = N'alex'
    AND Location = N'Seattle, WA';

SELECT Id, DisplayName, Location
  FROM dbo.Users WITH (INDEX = IX_Location_DisplayName)
  WHERE DisplayName = N'alex'
    AND Location = N'Seattle, WA';
```

Test ‘em

I don’t like index hints for long-term usage because your query will simply fail if the index disappears or is renamed. Hints are great for checking logical reads though.

[redacted]

Survey says…

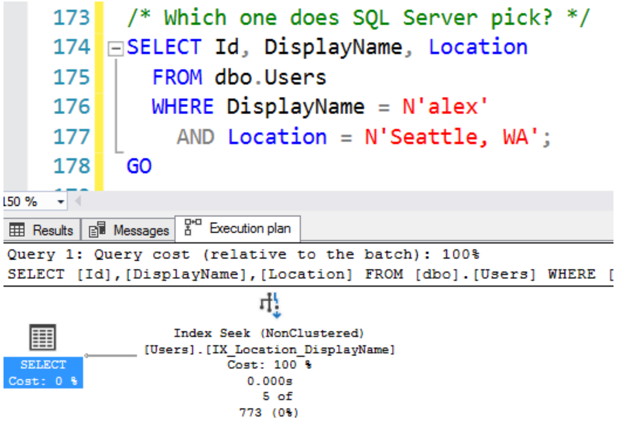
Index	Logical Reads
Clustered index (white pages)	45,184
IX_DisplayName_Includes	16
IX_DisplayName_Location	4
IX_Location_DisplayName	5

But don’t quibble over a handful of logical reads.

All of the indexes are pretty good!

Module 1 Slide 36

Without a hint, what gets used?



```
/* Which one does SQL Server pick? */
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = N'alex'
    AND Location = N'Seattle, WA';
GO
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT [Id],[DisplayName],[Location] FROM [dbo].[Users] WHERE [
```

SQL Server uses the Location, DisplayName index.

Don’t read too much into that yet though: it doesn’t mean it’s the one you should keep.

Before you pick a winner…

You rarely see SQL Servers that only run one query.

What if this query still runs sometimes?

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```

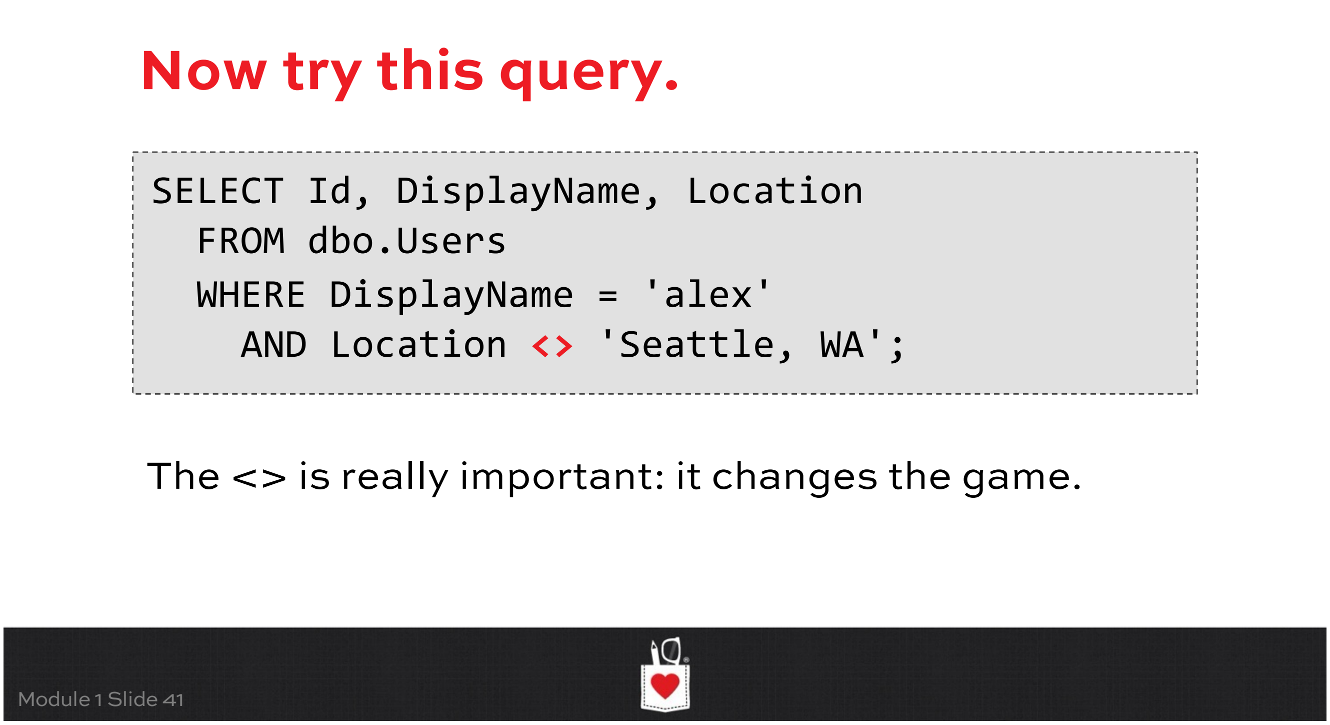
You wouldn’t want one index for this query, and a different index for Location, DisplayName.

WHERE

with both equality and inequality searches

What SQL Server calls equality

Equality searches	Inequality searches
=	<>, >, <, >=, <=
IS NULL	IS NOT NULL
IN (one value)	IN (two or more values)
	LIKE, NOT LIKE

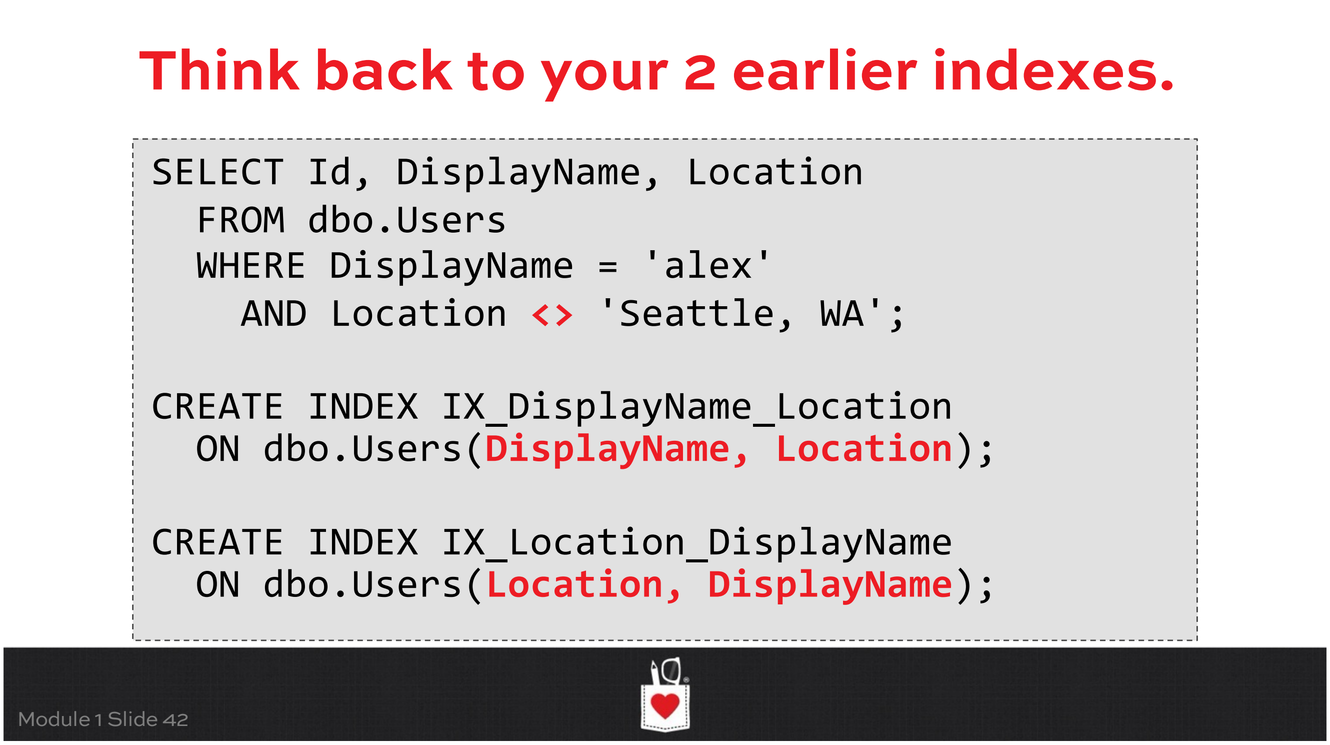


Now try this query.

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location <> 'Seattle, WA';
```

The <> is really important: it changes the game.

Module 1 Slide 41



Think back to your 2 earlier indexes.

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location <> 'Seattle, WA';

CREATE INDEX IX_DisplayName_Location
  ON dbo.Users(DisplayName, Location);

CREATE INDEX IX_Location_DisplayName
  ON dbo.Users(Location, DisplayName);
```

Module 1 Slide 42

Your execution plan…

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location <> 'Seattle, WA';

CREATE INDEX IX_DisplayName_Location
  ON dbo.Users(DisplayName, Location);
```

"I'll seek to DisplayName = Alex,
then read through them, looking at their locations,
returning all the ones who aren't in Seattle."

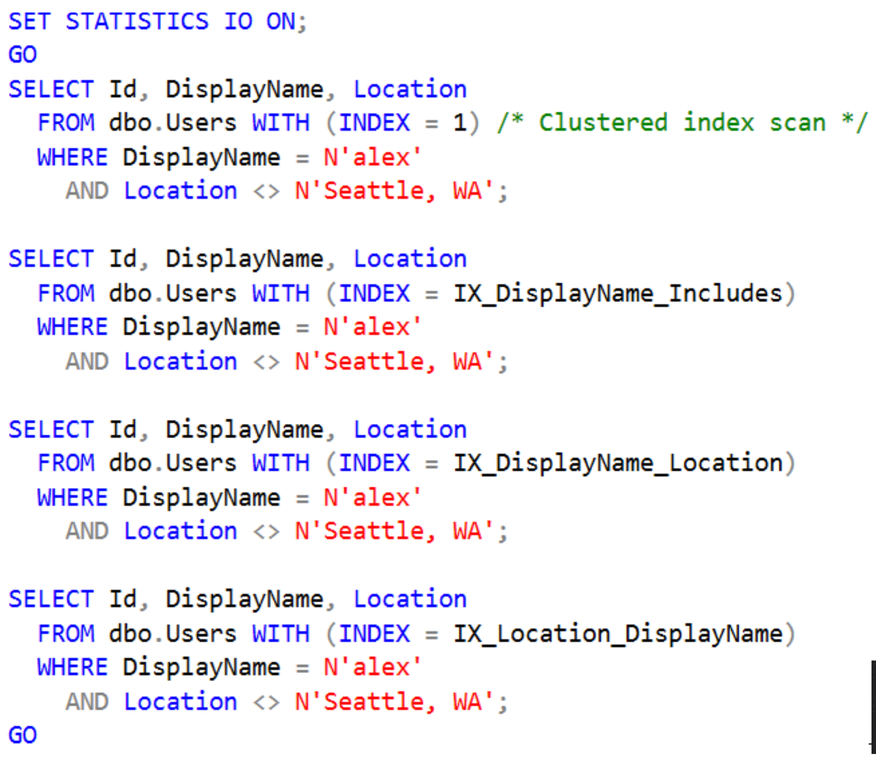
But if the index starts with Location?

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location <> 'Seattle, WA';

CREATE INDEX IX_Location_DisplayName
  ON dbo.Users(Location, DisplayName);
```

"I can't just jump to the Alexes."

"I'm gonna have to scan most of the index."



```
SET STATISTICS IO ON;
GO
SELECT Id, DisplayName, Location
  FROM dbo.Users WITH (INDEX = 1) /* Clustered index scan */
  WHERE DisplayName = N'alex'
    AND Location <> N'Seattle, WA';

SELECT Id, DisplayName, Location
  FROM dbo.Users WITH (INDEX = IX_DisplayName_Includes)
  WHERE DisplayName = N'alex'
    AND Location <> N'Seattle, WA';

SELECT Id, DisplayName, Location
  FROM dbo.Users WITH (INDEX = IX_DisplayName_Location)
  WHERE DisplayName = N'alex'
    AND Location <> N'Seattle, WA';

SELECT Id, DisplayName, Location
  FROM dbo.Users WITH (INDEX = IX_Location_DisplayName)
  WHERE DisplayName = N'alex'
    AND Location <> N'Seattle, WA';
GO
```

Test ‘em

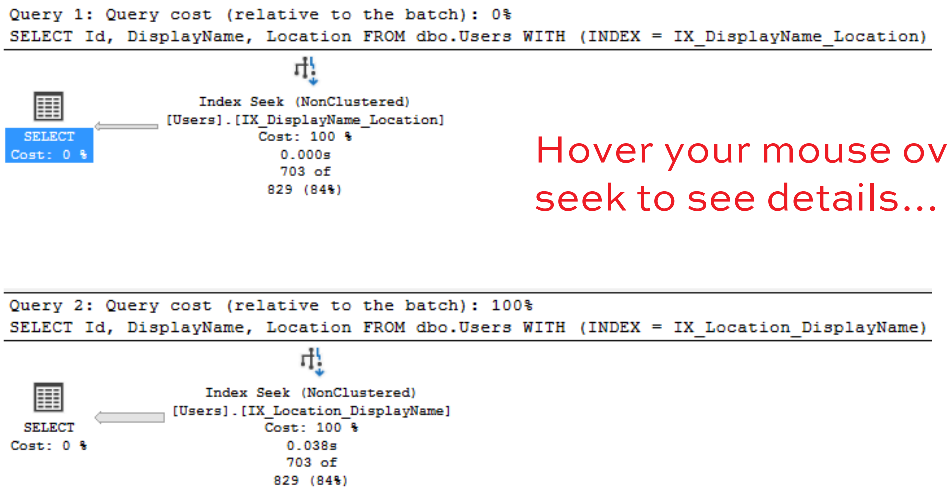
Note the <>.

Survey says…

Index	Logical Reads	Total Pages in the Index
Clustered index (white pages)	45,184	45,184
IX_DisplayName_Includes	16	12,577
IX_DisplayName_Location	13	12,701
IX_Location_DisplayName	4,566	13,183

Let’s compare the actual execution plans for the bottom two.

Remember, "seek" just means we seeked on the first column



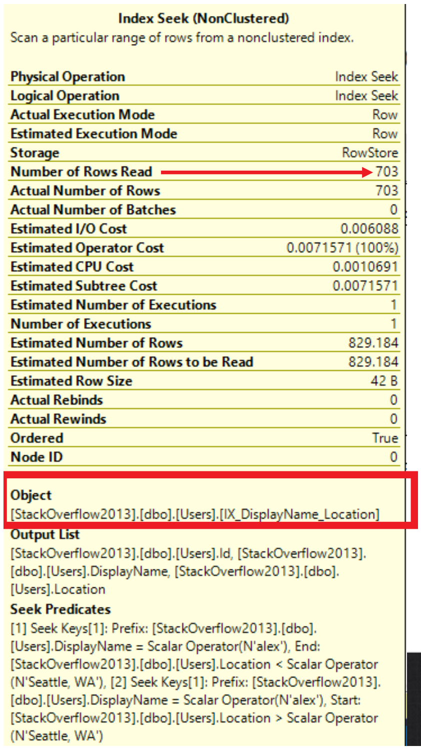
```
Query 1: Query cost (relative to the batch): 0%
SELECT Id, DisplayName, Location FROM dbo.Users WITH (INDEX = IX_DisplayName_Location)
```

SELECT
Cost: 0 %
Index Seek (NonClustered)
[Users].[IX_DisplayName_Location]
Cost: 100 %
0.000s
703 of
829 (84%)

Hover your mouse over each seek to see details…

```
Query 2: Query cost (relative to the batch): 100%
SELECT Id, DisplayName, Location FROM dbo.Users WITH (INDEX = IX_Location_DisplayName)
```

SELECT
Cost: 0 %
Index Seek (NonClustered)
[Users].[IX_Location_DisplayName]
Cost: 100 %
0.038s
703 of
829 (84%)



Index Seek (NonClustered)

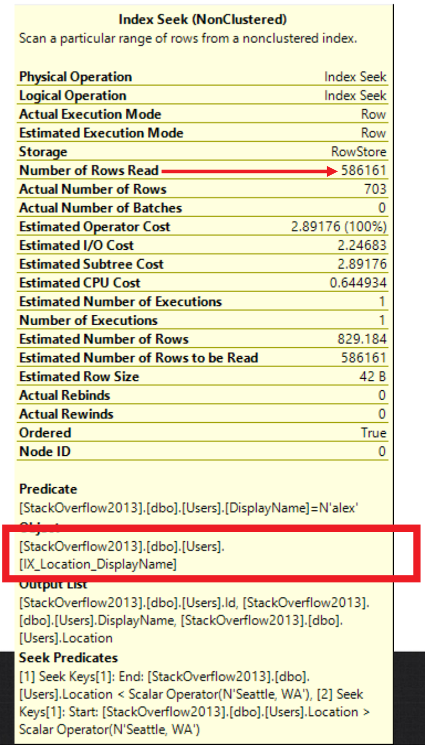
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	703
Actual Number of Rows	703
Actual Number of Batches	0
Estimated I/O Cost	0.006088
Estimated Operator Cost	0.0071571 (100%)
Estimated CPU Cost	0.0010691
Estimated Subtree Cost	0.0071571
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	829.184
Estimated Number of Rows to be Read	829.184
Estimated Row Size	42 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0

Object
[StackOverflow2013].[dbo].[Users].[IX_DisplayName_Location]

Output List
[StackOverflow2013].[dbo].[Users].Id, [StackOverflow2013].[dbo].[Users].DisplayName, [StackOverflow2013].[dbo].[Users].Location

Seek Predicates
[1] Seek Keys[1]: Prefix: [StackOverflow2013].[dbo].[Users].DisplayName = Scalar Operator(N'alex'), End: [StackOverflow2013].[dbo].[Users].Location < Scalar Operator(N'Seattle, WA'), [2] Seek Keys[1]: Prefix: [StackOverflow2013].[dbo].[Users].DisplayName = Scalar Operator(N'alex'), Start: [StackOverflow2013].[dbo].[Users].Location > Scalar Operator(N'Seattle, WA')



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	586161
Actual Number of Rows	703
Actual Number of Batches	0
Estimated Operator Cost	2.89176 (100%)
Estimated I/O Cost	2.24683
Estimated Subtree Cost	2.89176
Estimated CPU Cost	0.644934
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	829.184
Estimated Number of Rows to be Read	586161
Estimated Row Size	42 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0

Predicate
[StackOverflow2013].[dbo].[Users].[DisplayName]=N'alex'

Object
[StackOverflow2013].[dbo].[Users].[IX_Location_DisplayName]

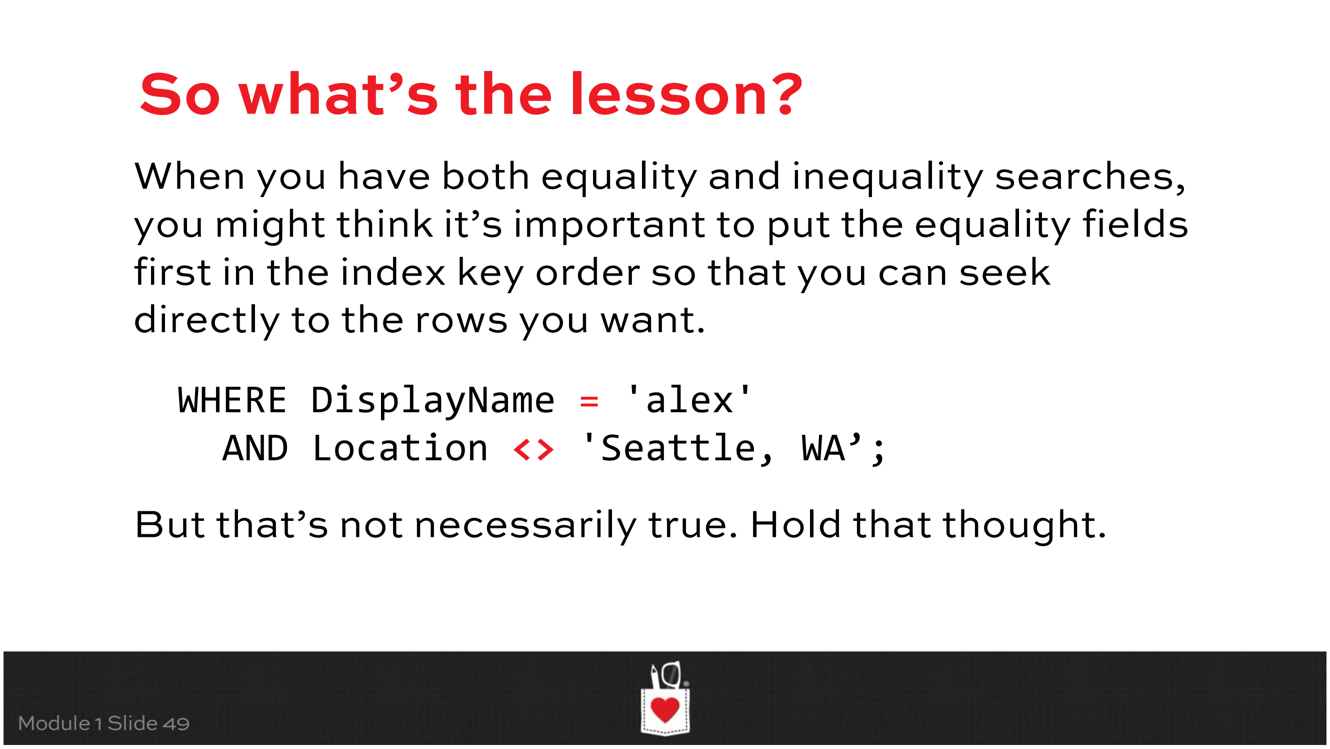
Output List
[StackOverflow2013].[dbo].[Users].Id, [StackOverflow2013].[dbo].[Users].DisplayName, [StackOverflow2013].[dbo].[Users].Location

Seek Predicates
[1] Seek Keys[1]: End: [StackOverflow2013].[dbo].[Users].Location < Scalar Operator(N'Seattle, WA'), [2] Seek Keys[1]: Start: [StackOverflow2013].[dbo].[Users].Location > Scalar Operator(N'Seattle, WA')

Different:

# of rows read

The one on the right has a residual predicate (Alex) that we can't seek to.



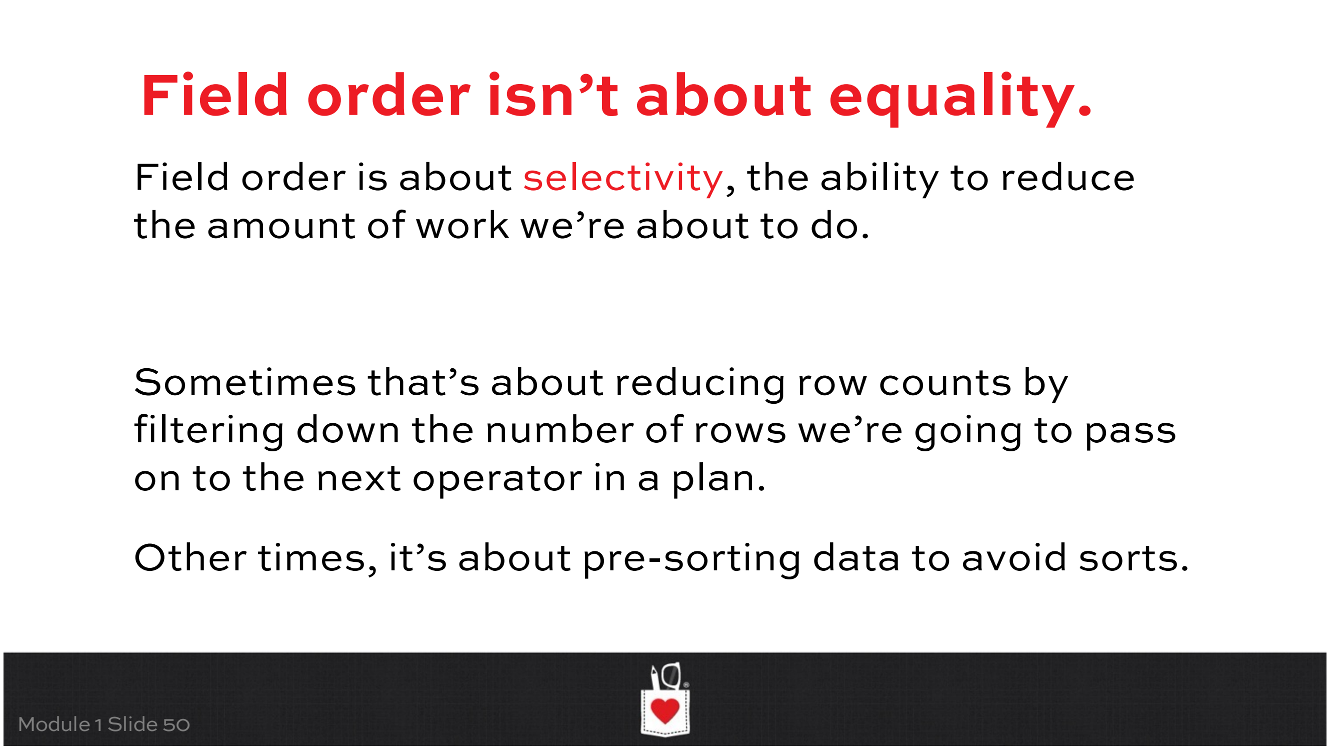
So what’s the lesson?

When you have both equality and inequality searches, you might think it’s important to put the equality fields first in the index key order so that you can seek directly to the rows you want.

```
WHERE DisplayName = 'alex'
  AND Location <> 'Seattle, WA’;
```

But that’s not necessarily true. Hold that thought.

Module 1 Slide 49



Field order isn’t about equality.

Field order is about selectivity, the ability to reduce the amount of work we’re about to do.

Sometimes that’s about reducing row counts by filtering down the number of rows we’re going to pass on to the next operator in a plan.

Other times, it’s about pre-sorting data to avoid sorts.

Module 1 Slide 50

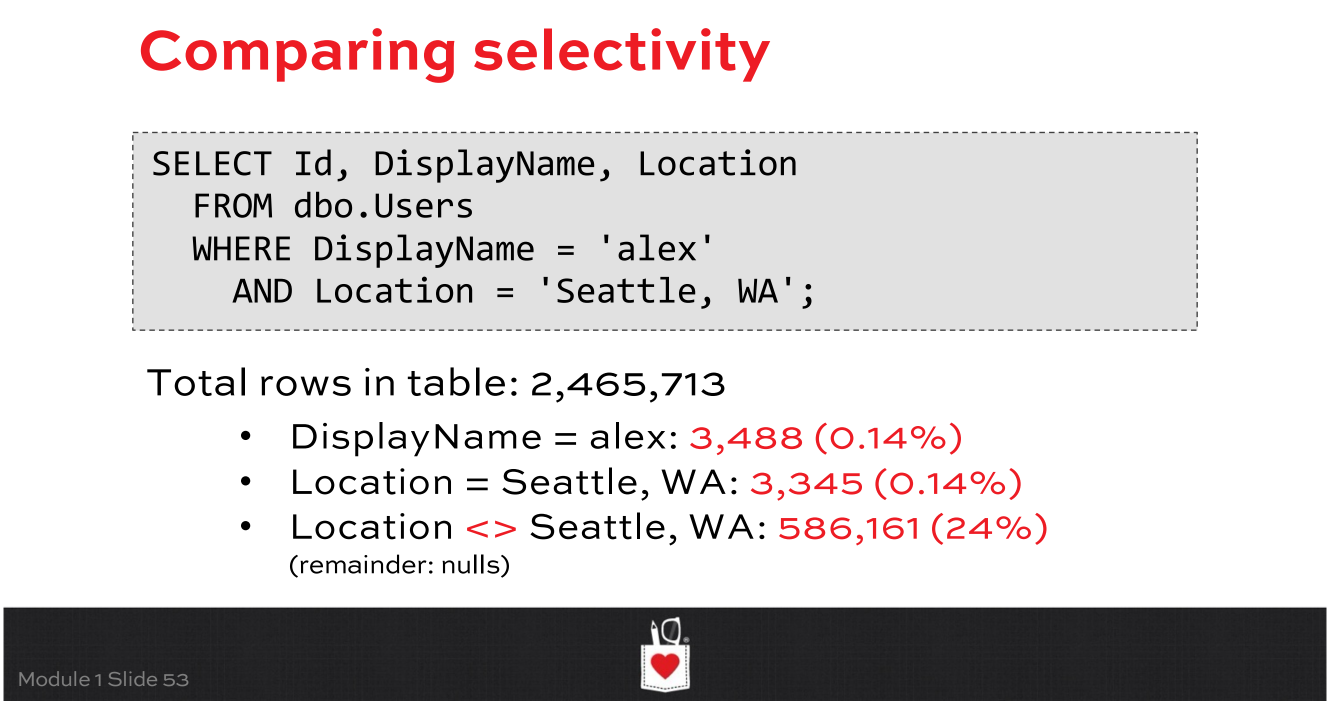
Selectivity

How selective is Alex?

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex';
```

Total rows in table: 2,465,713

Rows where DisplayName = alex: 3,488 (0.14%)



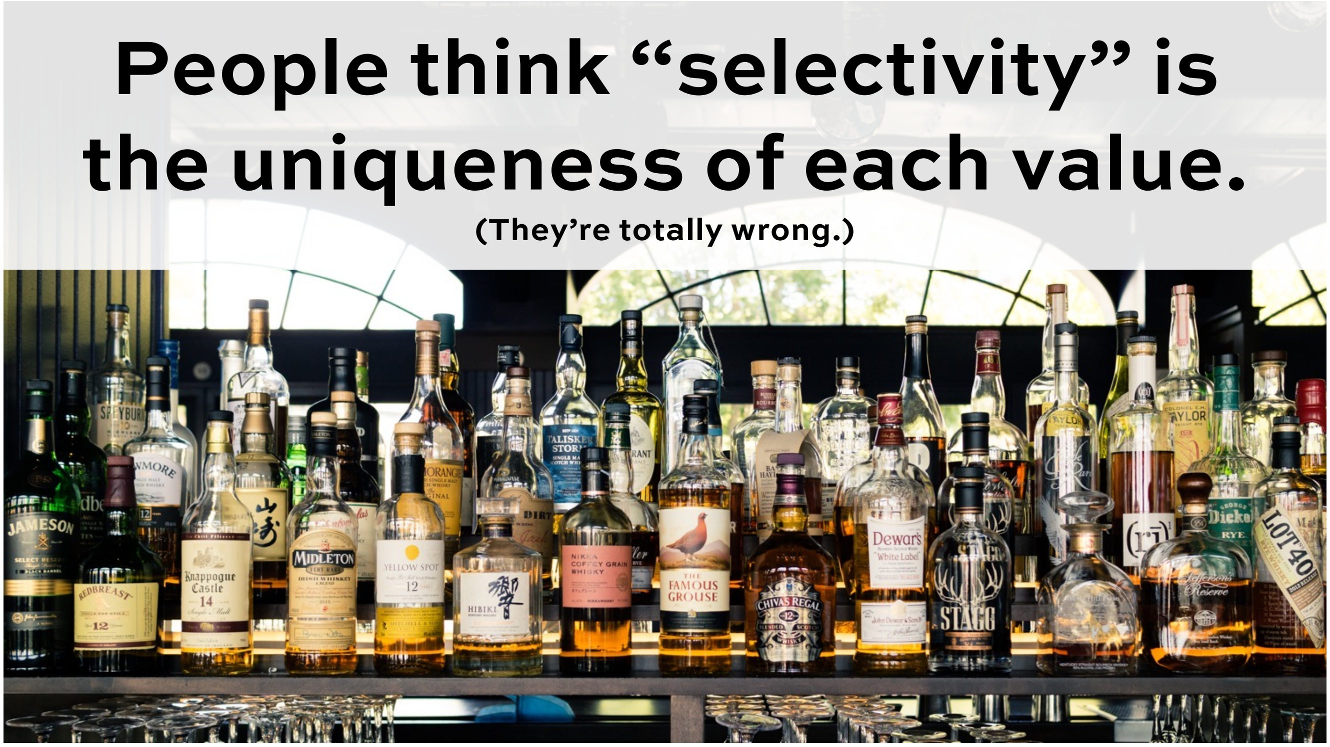
Comparing selectivity

```
SELECT Id, DisplayName, Location
  FROM dbo.Users
  WHERE DisplayName = 'alex'
    AND Location = 'Seattle, WA';
```

Total rows in table: 2,465,713

- DisplayName = alex: 3,488 (0.14%)
- Location = Seattle, WA: 3,345 (0.14%)
- Location <> Seattle, WA: 586,161 (24%)
  (remainder: nulls)

Module 1 Slide 53



People think “selectivity” is the uniqueness of each value.

(They’re totally wrong.)

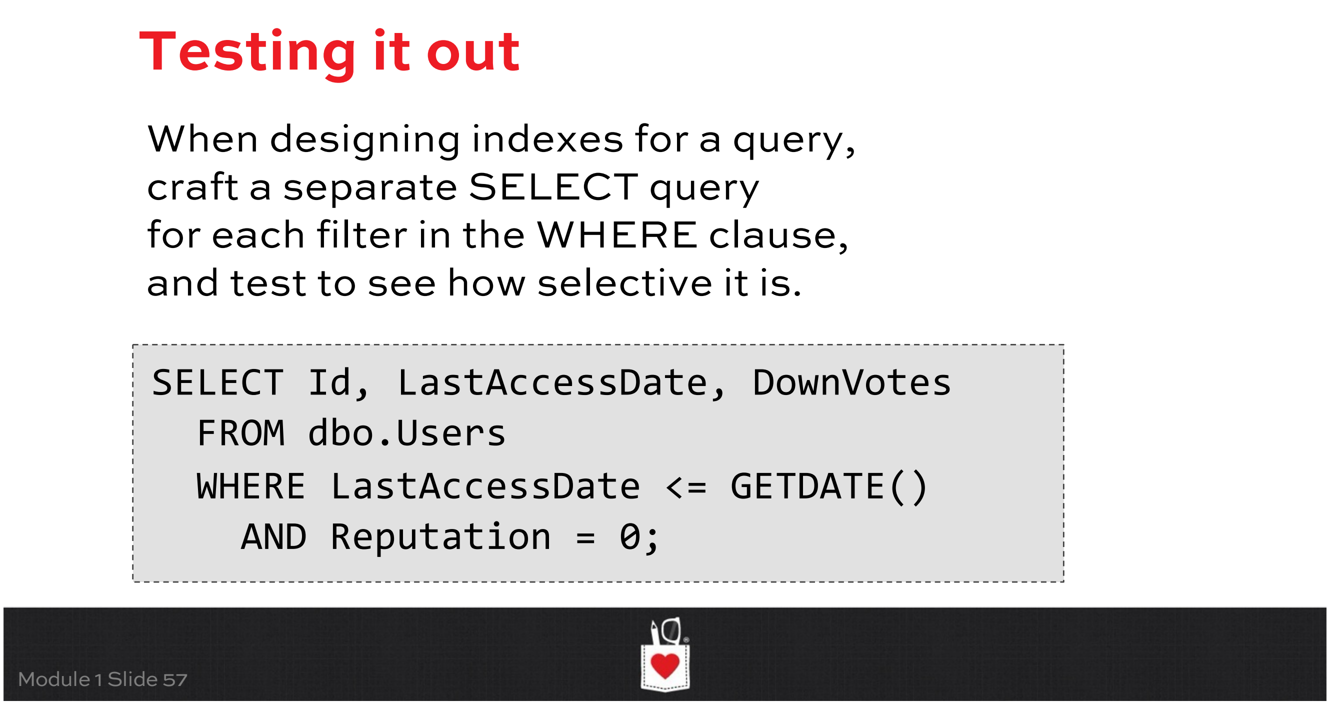
```
SELECT * FROM dbo.Booze
WHERE BrandName LIKE '%e%';
```

What selectivity really means

It’s not about how unique each row is by itself.

It’s about your query,
and how small a percentage of the table
you’re searching for.

When evaluating column order for indexes,
don’t think about how unique each column is.
Think about what percentage you’re searching for.



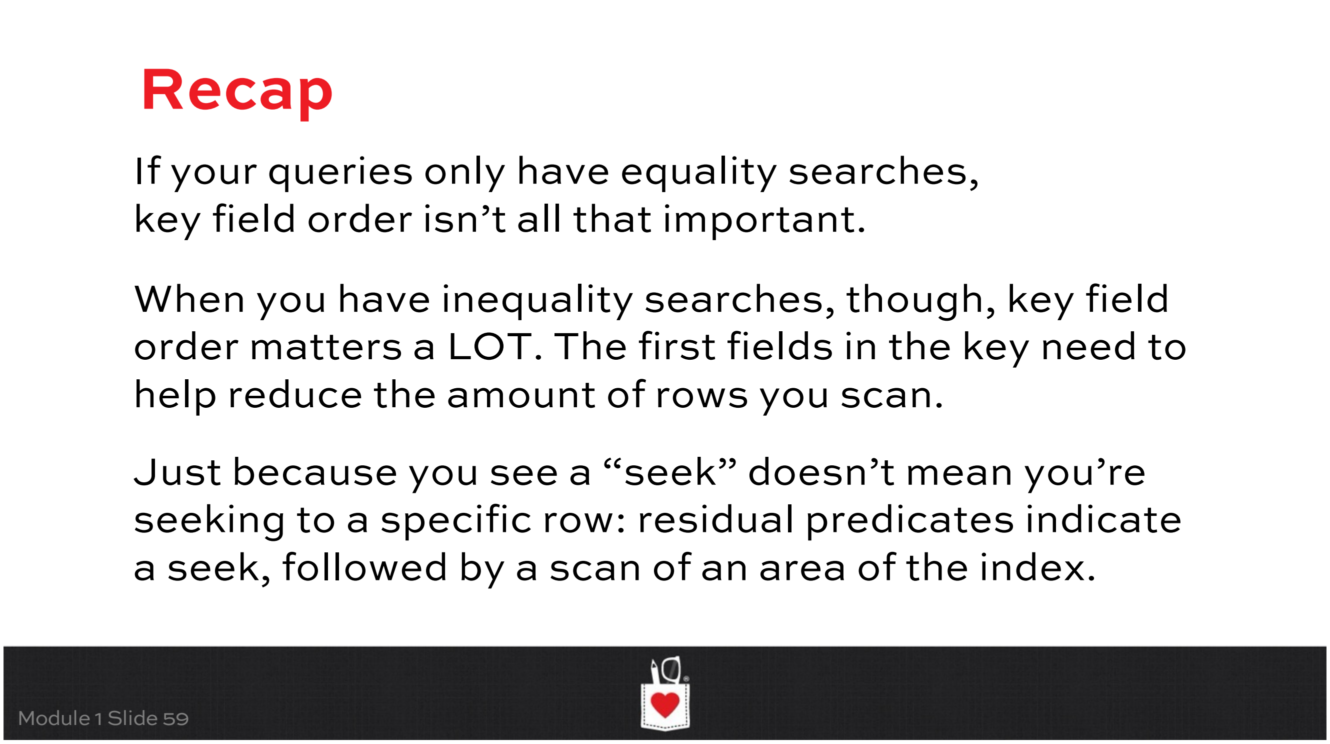
Testing it out

When designing indexes for a query,
craft a separate SELECT query
for each filter in the WHERE clause,
and test to see how selective it is.

```
SELECT Id, LastAccessDate, DownVotes
  FROM dbo.Users
  WHERE LastAccessDate <= GETDATE()
    AND Reputation = 0;
```

Module 1 Slide 57

Re-cap



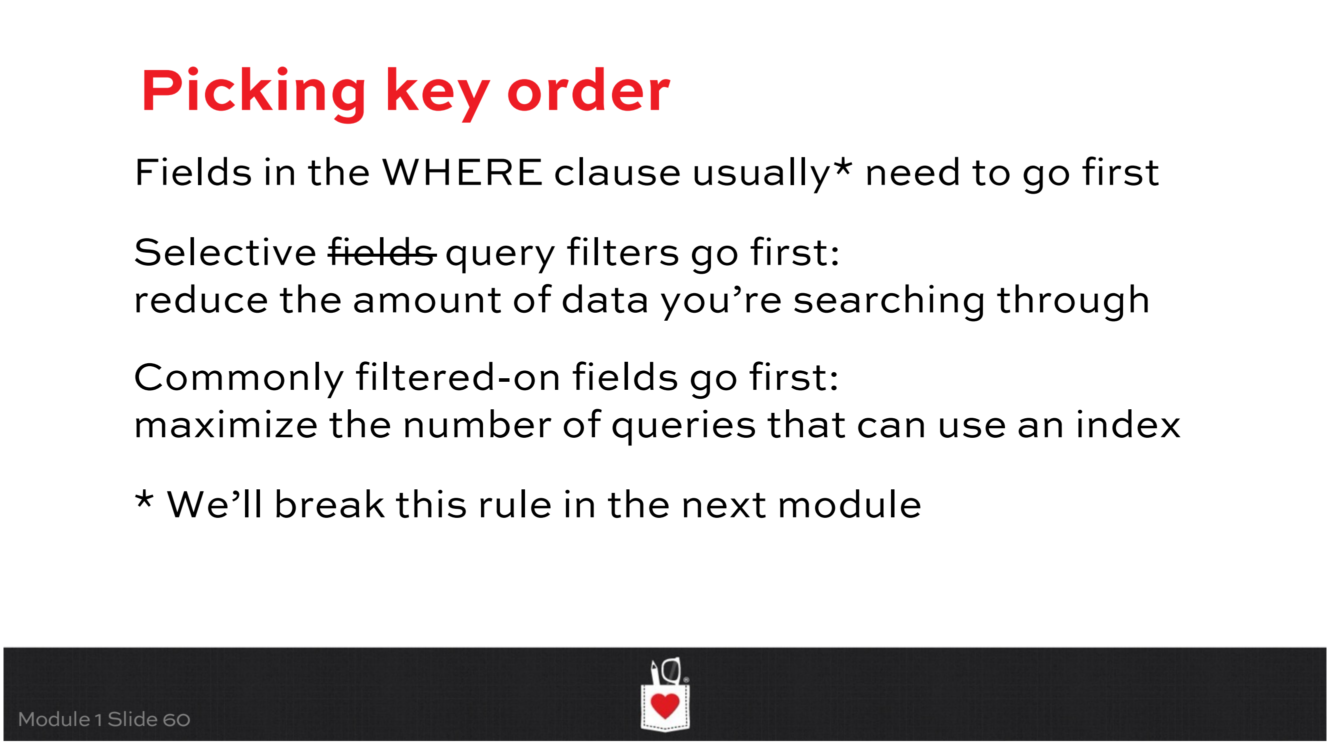
Recap

If your queries only have equality searches,
key field order isn't all that important.

When you have inequality searches, though, key field order matters a LOT. The first fields in the key need to help reduce the amount of rows you scan.

Just because you see a "seek" doesn't mean you're seeking to a specific row: residual predicates indicate a seek, followed by a scan of an area of the index.

Module 1 Slide 59



Picking key order

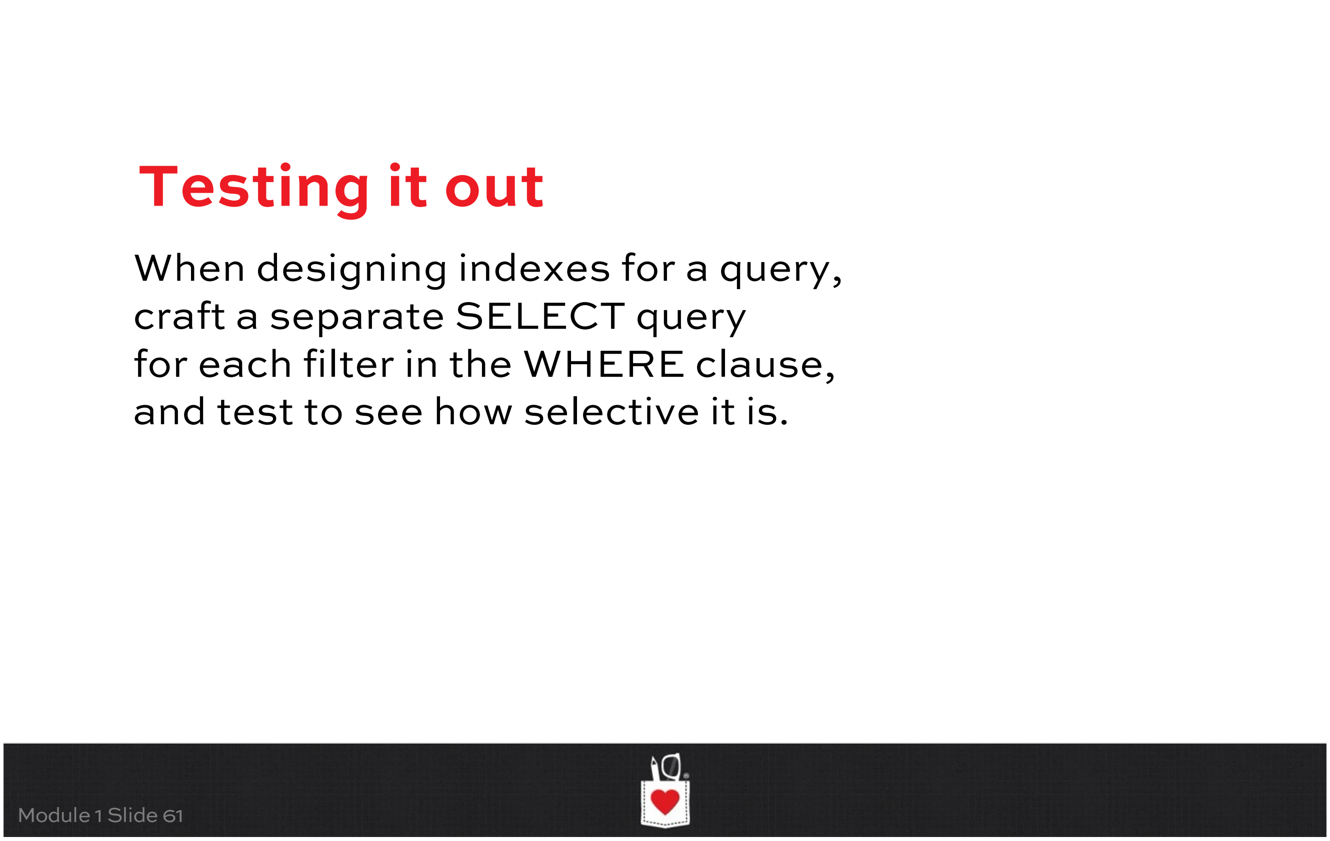
Fields in the WHERE clause usually* need to go first

Selective ~~fields~~ query filters go first:
reduce the amount of data you're searching through

Commonly filtered-on fields go first:
maximize the number of queries that can use an index

* We'll break this rule in the next module

Module 1 Slide 60



Testing it out

When designing indexes for a query,
craft a separate SELECT query
for each filter in the WHERE clause,
and test to see how selective it is.

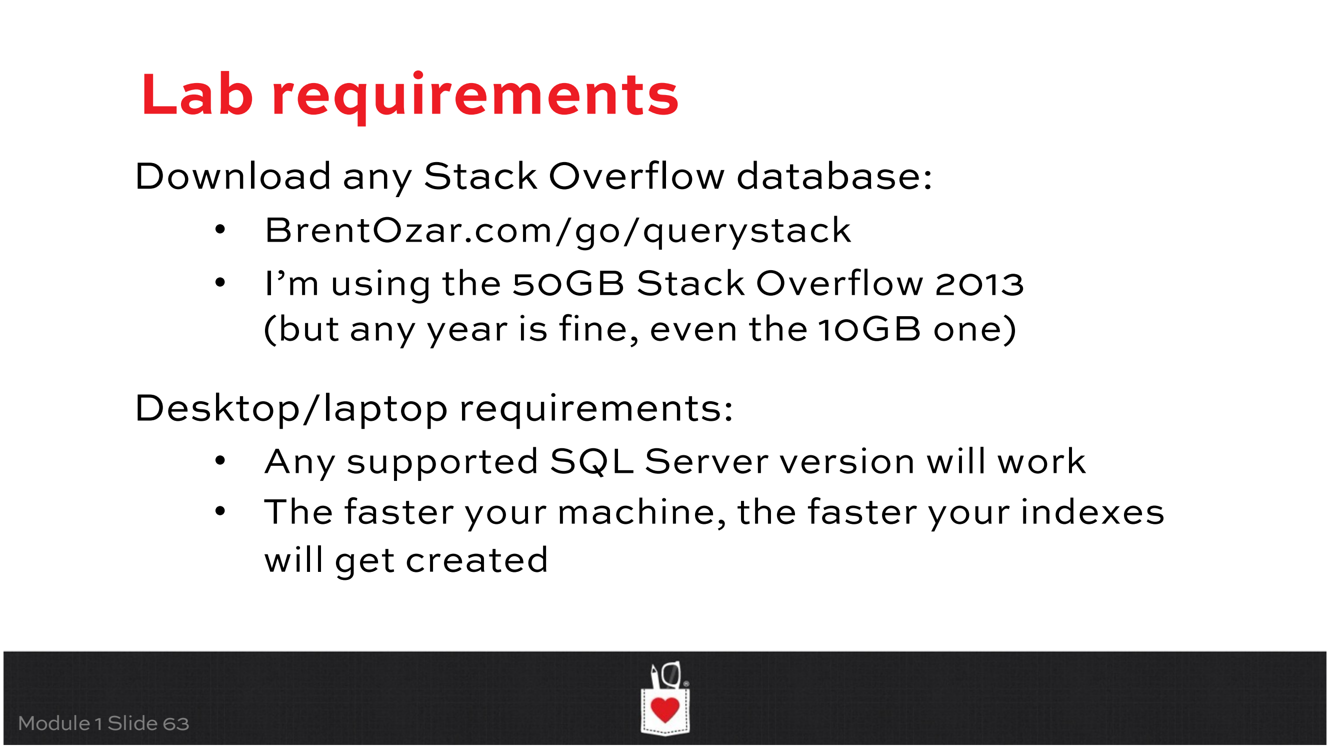
Module 1 Slide 61

BRENT OZAR
UNLIMITED®

Fundamentals of Index Tuning

Lab 1: let’s see what you learned about WHERE filters.

Module 1 Slide 62



Lab requirements

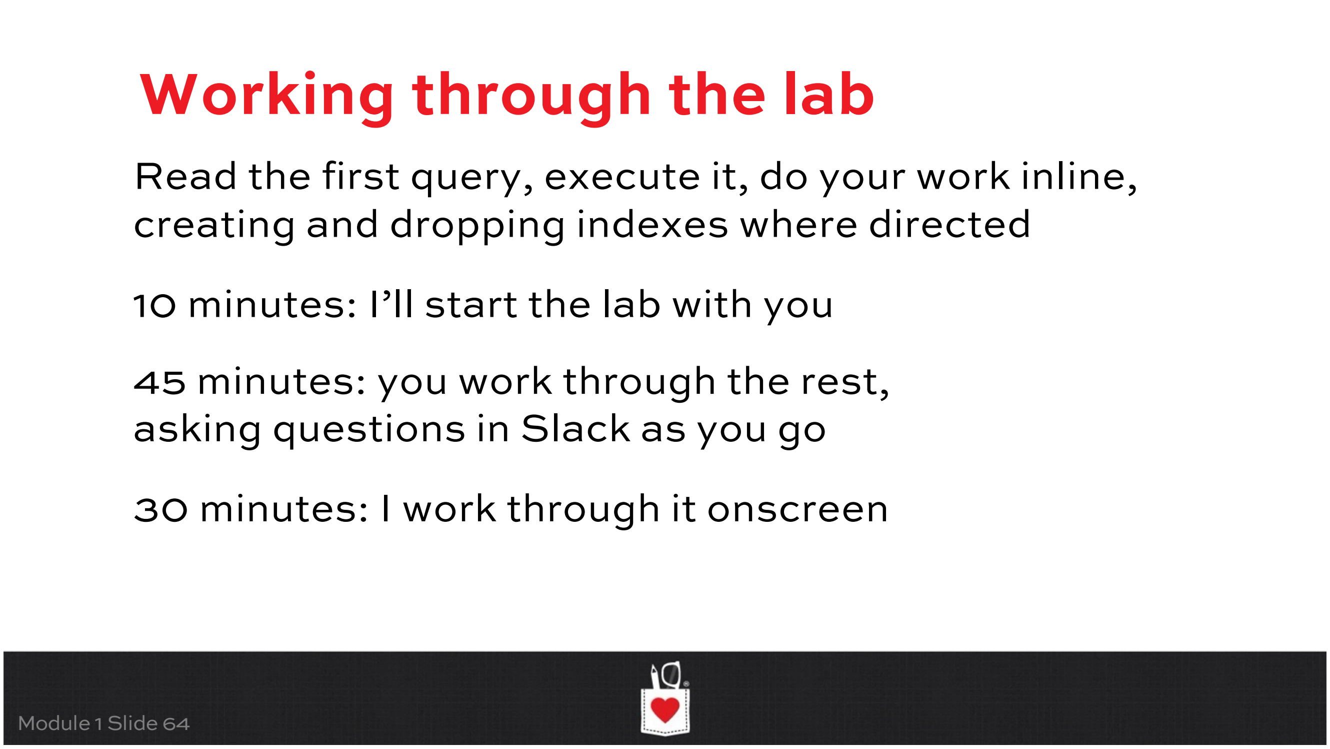
Download any Stack Overflow database:

- BrentOzar.com/go/querystack
- I'm using the 50GB Stack Overflow 2013 (but any year is fine, even the 10GB one)

Desktop/laptop requirements:

- Any supported SQL Server version will work
- The faster your machine, the faster your indexes will get created

Module 1 Slide 63



Working through the lab

Read the first query, execute it, do your work inline, creating and dropping indexes where directed

10 minutes: I'll start the lab with you

45 minutes: you work through the rest, asking questions in Slack as you go

30 minutes: I work through it onscreen

Module 1 Slide 64