



BRENT OZAR
UNLIMITED®

The D.E.A.T.H. Method: Adding Indexes with the DMVs

2.1 p1

Agenda

Learning to think like Clippy

Finding the queries Clippy's working on

Clippy in the cloud

Options I use when creating indexes

2.1 p2



In the last lab, we jumped to T.

Just once	Dedupe – reduce overlapping indexes Eliminate – unused indexes
Weekly for 1 month	Add – badly needed missing indexes
Do this only AFTER the easy stuff above	Tune – indexes for specific queries Heaps – usually need clustered indexes

2.1 p3



The A part relies on Clippy.

SQL Server gives us missing indexes in:

- sys.dm_db_missing_index_details
- Query plans and the plan cache

He doesn't care about:

- The size of the index
- The index's overhead on D/U/I operations
- Your other existing indexes
- A lot of operations in the plan (GROUP BY)



2.1 p4



Clippy's order of fields

1. Equality searches in the query
(ordered by the field order in the table)
2. Inequality searches
(ordered by the field order in the table)
3. Includes
(which might actually need to be sorted)



2.1 p5



Time to level up.

In Fundamentals of Index Tuning,
I gave you queries,
and had you come up with Clippy's field ordering.

Here in Mastering Index Tuning,
I'm going to give you Clippy's suggestion,
and I want you to reverse engineer the query.
(There are multiple answers for each index.)

2.1 p6



Quiz 1:

1 query at a time

2.1 p7



Missing index suggestion #1

```
Missing Index (Impact 99.992): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([DisplayName],[Location])
```

2.1 p8



Missing index suggestion #1

```
SELECT *
FROM dbo.Users
WHERE DisplayName = 'Brent Ozar'
AND Location = 'San Diego, CA, USA';
GO
```

Query 1: Query cost (relative to the batch): 100%

SELECT * FROM dbo.Users WHERE DisplayName = 'Brent Ozar' AND Location = 'San Diego, CA, USA'

Missing Index (Impact 99.992): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([<ColumnName>],[<ColumnName>]) INCLUDE ([<ColumnName>],[<ColumnName>])

Clustered Index Scan (Clustered)
(Users).[PK_Users_Id]
Cost: 100 %

In this case, key order really doesn't matter. (Even a single-column index on either field would work.)

2.1 p9



Missing index suggestion #2

Missing Index (Impact 99.599): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([Location],[DisplayName]) INCLUDE ([AboutMe],[Age],[C

```
<MissingIndexes>
<MissingIndexGroup Impact="99.599">
  <MissingIndex Database="[StackOverflow]" Schema="[dbo]" Table="[Users]">
    <ColumnGroup Usage="EQUALITY">
      <Column Name="[Location]" ColumnId="9" />
    </ColumnGroup>
    <ColumnGroup Usage="INEQUALITY">
      <Column Name="[DisplayName]" ColumnId="5" />
    </ColumnGroup>
    <ColumnGroup Usage="INCLUDE">
      <Column Name="[AboutMe]" ColumnId="2" />
      <Column Name="[Age]" ColumnId="3" />
      <Column Name="[CreationDate]" ColumnId="4" />
      <Column Name="[DownVotes]" ColumnId="6" />
      <Column Name="[EmailHash]" ColumnId="7" />
      <Column Name="[LastAccessDate]" ColumnId="8" />
      <Column Name="[Reputation]" ColumnId="10" />
      <Column Name="[UpVotes]" ColumnId="11" />
      <Column Name="[Views]" ColumnId="12" />
      <Column Name="[WebsiteUrl]" ColumnId="13" />
      <Column Name="[AccountId]" ColumnId="14" />
    </ColumnGroup>
  </MissingIndex>
</MissingIndexGroup>
```

Missing index suggestion #2

```
SELECT *
FROM dbo.Users
WHERE DisplayName <> 'Brent Ozar'
AND Location = 'San Diego, CA, USA';
GO
```

50 %

Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

SELECT * FROM dbo.Users WHERE DisplayName <> 'Brent Ozar' AND Location = 'San Diego, CA, USA'

Missing Index (Impact 99.599): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([Location],[DisplayName]) INCLUDE ([AboutMe],[Age],[C

SELECT
Cost: 0 %

Clustered Index Scan (Clustered)
[Users].[PK_Users_Id]
Cost: 100 %

Clippy puts equality fields first, then inequalities, all sorted by their field order in the table.

2.1 p11



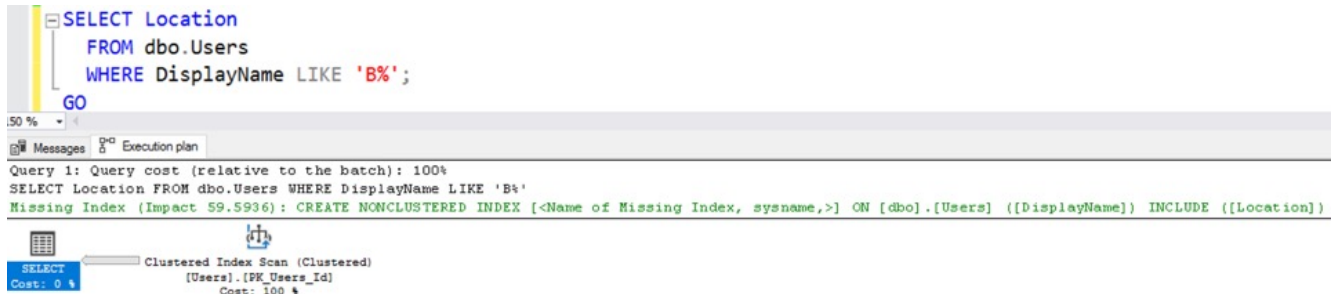
Missing index suggestion #3

Missing Index (Impact 59.5936): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([DisplayName]) INCLUDE ([Location])

2.1 p12



Missing index suggestion #3



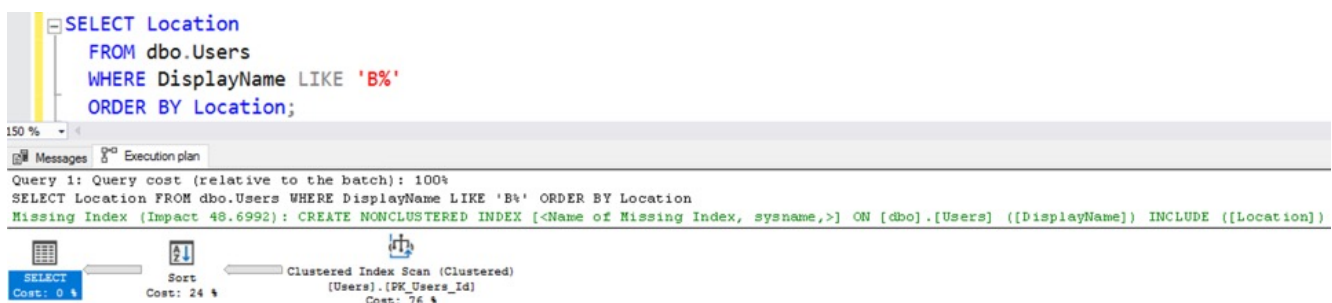
The screenshot shows a SQL query in the Messages pane: `SELECT Location FROM dbo.Users WHERE DisplayName LIKE 'B%';`. The Execution plan pane shows a single step: `SELECT` with a cost of 0%. The Missing Index pane shows a suggestion: `CREATE NONCLUSTERED INDEX [Name of Missing Index, sysname, >] ON [dbo].[Users] ([DisplayName]) INCLUDE ([Location])`. The query cost is 100%.

In this case, the Location doesn't need to be sorted.

2.1 p13



But this will also produce it...



The screenshot shows a SQL query in the Messages pane: `SELECT Location FROM dbo.Users WHERE DisplayName LIKE 'B%' ORDER BY Location;`. The Execution plan pane shows three steps: `SELECT` (cost 0%), `Sort` (cost 24%), and `Clustered Index Scan (Clustered)` (cost 76%). The Missing Index pane shows a suggestion: `CREATE NONCLUSTERED INDEX [Name of Missing Index, sysname, >] ON [dbo].[Users] ([DisplayName]) INCLUDE ([Location])`. The query cost is 100%.

Remember, Clippy has some blind spots like GROUP BY and ORDER BY. Generally, if you see a single-column index with includes, he probably needs one of the includes as a key, too.

2.1 p14



Quiz 2:

2 queries at a time

2.1 p15



Now it gets harder.

```
Missing Index (Impact 99.9968): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([LastAccessDate],[WebsiteUrl])
```

```
Missing Index (Impact 99.9956): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([DownVotes],[WebsiteUrl])
```

1. What queries might have built these hints?
2. Can one index satisfy both queries? Why?

2.1 p16




```

<MissingIndexes>
  <MissingIndexGroup Impact="99.9968">
    <MissingIndex Database="[StackOverflow]" Schema="[dbo]" Table="[Users]">
      <ColumnGroup Usage="EQUALITY">
        <Column Name="[LastAccessDate]" ColumnId="8" />
        <Column Name="[WebsiteUrl]" ColumnId="13" />
      </ColumnGroup>
    </MissingIndex>
  </MissingIndexGroup>
</MissingIndexes>

<MissingIndexes>
  <MissingIndexGroup Impact="99.9956">
    <MissingIndex Database="[StackOverflow]" Schema="[dbo]" Table="[Users]">
      <ColumnGroup Usage="EQUALITY">
        <Column Name="[DownVotes]" ColumnId="6" />
        <Column Name="[WebsiteUrl]" ColumnId="13" />
      </ColumnGroup>
    </MissingIndex>
  </MissingIndexGroup>
</MissingIndexes>

```

2.1 p17



150 %

```

SELECT Id
FROM dbo.Users
WHERE LastAccessDate = GETDATE()
AND WebsiteUrl = 'https://www.BrentOzar.com';

SELECT Id
FROM dbo.Users
WHERE DownVotes = 0
AND WebsiteUrl = 'https://www.BrentOzar.com';

```

Messages Execution plan

Query 1: Query cost (relative to the batch): 50%

SELECT Id FROM dbo.Users WHERE LastAccessDate = GETDATE() AND WebsiteUrl = 'https://www.BrentOzar.com'

Missing Index (Impact 99.9968): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ((LastAccessDate],[WebsiteUrl]))

SELECT
Cost: 0 %

Clustered Index Scan (Clustered)
[Users].[PK_Users_Id]
Cost: 100 %

Query 2: Query cost (relative to the batch): 50%

; SELECT Id FROM dbo.Users WHERE DownVotes = 0 AND WebsiteUrl = 'https://www.BrentOzar.com'

Missing Index (Impact 99.9956): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ((DownVotes],[WebsiteUrl]))

SELECT
Cost: 0 %

Clustered Index Scan (Clustered)
[Users].[PK_Users_Id]
Cost: 100 %

2.1 p18



Can one index help both?

Sure, as long as it leads with WebsiteUrl.

```
SELECT Id
FROM dbo.Users
WHERE LastAccessDate = GETDATE()
AND WebsiteUrl = 'https://www.BrentOzar.com';

SELECT Id
FROM dbo.Users
WHERE DownVotes = 0
AND WebsiteUrl = 'https://www.BrentOzar.com';
```

Any of these would help:

- (WebsiteUrl, DownVotes, LastAccessDate)
- (WebsiteUrl, LastAccessDate, DownVotes)
- (WebsiteUrl, DownVotes) INCLUDE (LastAccessDate)
- (WebsiteUrl, LastAccessDate) INCLUDE (DownVotes)
- Maybe even: (WebsiteUrl) INCLUDE (DownVotes, LastAccessDate)

2.1 p19



Perfect is the enemy of good.

Voltaire on index design

2.1 p20



Our job in index tuning

Look at the recommended indexes by:

- Equality fields
- Inequality fields
- Includes (which sometimes need to be promoted)

Look for things they have in common

Make guess based on our experience
with the queries and the data

Revisit it later in the T part of D.E.A.T.H.

2.1 p21



Quiz 3: 3 queries at a time

2.1 p22



We've been looking at plans...

```
Missing Index (Impact 99.9977): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Votes] ([PostId]) INCLUDE ([VoteTypeId])

Missing Index (Impact 99.9978): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Votes] ([PostId]) INCLUDE ([UserId],[Bounty

Missing Index (Impact 99.9996): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Votes] ([PostId],[VoteTypeId]) INCLUDE ([Us
```

2.1 p23



But let's switch to sp_BlitzIndex

sp_BlitzIndex

Priority	Finding	Details: schema.table.index(indexid)	Definition: [Property] ColumnName (datatype maxbytes)	Usage
-1	sp_BlitzIndex(TM) v7.5 - April 27, 2019: Da...	http://FirstResponderKit.org	Server: SQL2017LAB1 Days Uptime: 0.25	
50	Indexaphobia: High value missing index	[StackOverflow].[dbo].[Votes] Est. benefit per day: 1,045,472	EQUALITY: [PostId] INEQUALITY: [VoteTypeId] INCLUDES: [UserId], [BountyAmount], [CreationDate]	3 uses: Impact: 100.0%; Avg query cost: 871.2268
50	Indexaphobia: High value missing index	[StackOverflow].[dbo].[Votes] Est. benefit per day: 900,733	EQUALITY: [PostId] INCLUDES: [VoteTypeId]	3 uses: Impact: 100.0%; Avg query cost: 750.6115
50	Indexaphobia: High value missing index	[StackOverflow].[dbo].[Votes] Est. benefit per day: 900,733	EQUALITY: [PostId] INCLUDES: [UserId], [BountyAmount], [VoteTypeId], [CreationDate]	3 uses: Impact: 100.0%; Avg query cost: 750.6114

1. What queries might have built these hints?
2. What indexes would you create to help as many of the queries as practical?

2.1 p24



Guessing at it

```
EQUALITY: [PostId] INEQUALITY: [VoteTypeId] INCLUDES: [UserId], [BountyAmount], [CreationDate]  
EQUALITY: [PostId] INCLUDES: [VoteTypeId]  
EQUALITY: [PostId] INCLUDES: [UserId], [BountyAmount], [VoteTypeId], [CreationDate]
```

PostId, VoteTypeId would satisfy the top two

Depending on the uniqueness of PostId, it might even satisfy the 3rd one (but not cover it)

2.1 p25



How'd your indexes do with these?

```
SELECT VoteTypeId, COUNT(*) AS TotalVotes  
FROM dbo.Votes  
WHERE PostId = 12345  
GROUP BY VoteTypeId;
```

```
SELECT *  
FROM dbo.Votes  
WHERE PostId = 12345  
ORDER BY CreationDate;
```

```
SELECT *  
FROM dbo.Votes  
WHERE PostId = 12345  
AND VoteTypeId IN (2, 3);
```

2.1 p26



What everyone wants to know:

“But how do I find the real queries?”

2.1 p27



Fastest: SQL Server 2019+ only

SQL Server 2019 added a new DMV:
`sys.dm_db_missing_index_group_stats_query`.

`sp_BlitzIndex @TableName = 'mytable'`
Look in the far right of the missing indexes section

`sp_BlitzIndex @Mode = 3`
(lists missing index requests)

2.1 p28



Azure SQL DB? Log it regularly.

Since your database can (and will) restart without warning, log this data weekly so it's there when you want to do index analysis, and use the most recent:

```
sp_BlitzIndex @Mode = 3,  
@OutputDatabaseName = 'MyDB',  
@OutputSchemaName = 'dbo',  
@OutputTableName = 'BlitzIndex_Mode3'
```

2 = inventory, 3 = missing indexes

2.1 p29



On 2017 & prior, I don't find the queries in the "A" phase.

Just
once

Dedupe – reduce overlapping indexes

Eliminate – unused indexes

Weekly
for 1
month

Add – badly needed missing indexes

Do this only
AFTER the easy
stuff above

Tune – indexes for specific queries

Heaps – usually need clustered indexes

2.1 p30



Fast, but less accurate: top reads

```
sp_BlitzCache @SortOrder = 'reads',  
@DatabaseName = 'mydb'
```

- Lists the top resource-intensive queries
- They probably need an index or two
- Probably not the specific index you're looking for
- Doesn't catch queries that originate in other dbs, queries that aren't in the plan cache now

2.1 p31



Slow: query the entire cache

sp_BlitzCache queries just top readers, so it's fast.

This query checks the entire plan cache, looking at the XML to see if there's a missing index on a specific table you're looking for:

<http://www.sqlnuggets.com/blog/sql-scripts-find-queries-that-have-missing-index-requests/>

Doesn't catch queries that originate in other dbs, queries that aren't in the plan cache now

2.1 p32



Slow: Query Store

Query Store is just like a persisted version of the plan cache, and you can query the plans in it.

This query finds ALL missing index requests, so it's going to be slow and unfiltered:

<https://www.scarydba.com/2019/03/11/missing-indexes-in-the-query-store/>

Drawbacks:

- Requires Query Store (which has many drawbacks)
- Only catches plans that made it to Query Store

2.1 p33



Cloud Clippy

2.1 p34



“Azure does it for me!”

Microsoft’s marketing implies self-tuning databases.

The reality is very different:

<https://learn.microsoft.com/en-us/azure/azure-sql/database/automatic-tuning-overview?view=azuresql>

2.1 p35



Automatic tuning options

The automatic tuning options available in Azure SQL Database and Azure SQL Managed Instance are:

Automatic tuning option	Description	Single database and pooled database support	Instance database support
CREATE INDEX	Identifies indexes that may improve performance of your workload, creates indexes, and automatically verifies that performance of queries has improved. When recommending a new index, the system considers space available in the database. If index addition is estimated to increase space utilization to over 90% toward maximum data size, index recommendation is not generated. Once the system identifies a period of low utilization and starts to create an index, it will not pause or cancel this operation even if resource utilization unexpectedly increases. If index creation fails, it will be retried during a future period of low utilization. Index recommendations are not provided for tables where the clustered index or heap is larger than 10 GB.	Yes	No
DROP INDEX	Drops unused (over the last 90 days) and duplicate indexes. Unique indexes, including indexes supporting primary key and unique constraints, are never dropped. This option may be automatically disabled when queries with index hints are present in the workload, or when the workload performs partition switching. On Premium and Business Critical service tiers, this option will never drop unused indexes, but will drop duplicate indexes, if any.	Yes	No

Automatic tuning options

The automatic tuning options available in Azure SQL Database and Azure SQL Managed Instance are:

Automatic tuning option	Description	Single database and pooled database support	Instance database support
CREATE INDEX	<p>index, the system considers space available in the database. If index addition is estimated to increase space utilization to over 90% toward maximum data size, index recommendation is not generated.</p>	Yes	No
DROP INDEX		Yes	No

Automatic tuning options

The automatic tuning options available in Azure SQL Database and Azure SQL Managed Instance are:

Automatic tuning option	Description	Single database and pooled database support	Instance database support
CREATE INDEX	<p>Index recommendations are not provided for tables where the clustered index or heap is larger than 10 GB.</p>	Yes	No
DROP INDEX		Yes	No

Automatic tuning options

The automatic tuning options available in Azure SQL Database and Azure SQL Managed Instance are:

Automatic tuning option	Description	Single database and pooled database support	Instance database support
CREATE INDEX		Yes	No
DROP INDEX	Drops unused (over the last 90 days) and duplicate indexes.	Yes	No

Automatic tuning options

The automatic tuning options available in Azure SQL Database and Azure SQL Managed Instance are:

Automatic tuning option	Description	Single database and pooled database support	Instance database support
CREATE INDEX		Yes	No
DROP INDEX	<div>On Premium and Business Critical service tiers, this option will never drop unused indexes, but will drop duplicate indexes, if any.</div>	Yes	No

This is a really timid feature.

And even when it does kick in, here's how it works:

https://www.microsoft.com/en-us/research/uploads/prod/2019/02/autoindexing_azuredb.pdf

The index recommender analyzes query execution statistics to automatically identify inputs, such as a workload to tune. To generate high quality index recommendations, we adapt and extend two building blocks for index recommendations in SQL Server: (a) Missing Indexes [34]; and (b) Database Engine Tuning Advisor (DTA) [2, 10]. These

2.1 p41



Tweak's Clippy's logic a little

We perform the following high-level steps to generate the final index recommendations. *First*, we define a non-clustered (secondary) index candidate with information in the MI DMV. The EQUALITY columns are selected as keys, INCLUDE columns are the included columns. SQL Server storage engine can seek a B+ tree index with multiple equality predicates but only one inequality predicate. Hence, we pick one of the INEQUALITY columns as a key (ordered after EQUALITY columns), the remaining are included columns.

2.1 p42



It probably won't come on-prem:

Since index changes can have significant performance impact, such experimentation is not possible on a database's primary replica. We create *B-instances*, which are independent copies of the database seeded from a snapshot of the primary copy where the real application's workload is forwarded and replayed in parallel to the primary copy. We make physical design changes or enable new features on these B-instances without impacting the primary workload, allowing us to experiment with different index configurations and measure their impact on execution (Section 7). While such experimen-

2.1 p43



Overall, my experience

Automatic index tuning:

- Only kicks in for small objects (<10GB)
- Won't drop indexes on production servers

Good for SaaS/ISV apps with one db per client:

- Ship with a core set of critical indexes
- Let Azure add indexes automatically based on each client's particular usage

2.1 p44



CREATE INDEX

bonus options

2.1 p45



What you'll usually see me use

WITH (ONLINE = OFF, MAXDOP = 0)

ONLINE = OFF is FASTER = ON.

Even when no other activity is happening on the server,
ONLINE = ON is slower. Use ON only when you have to.

MAXDOP = 0 overrides server-level MAXDOP limits,
using all the CPU cores to knock it out quickly.

2.1 p46



Other options I like

DROP_EXISTING = ON: helps you keep existing indexes in place, avoiding problems with index hints.

SORT_IN_TEMPDB = ON: useful when:

- Your data files are on really slow network storage (especially in the cloud)
- TempDB is on fast local SSD (ephemeral)

2.1 p47



Options I'm ambivalent about

DATA_COMPRESSION = ROW, PAGE, NONE

Only works for on-row data:
things that stay on the 8KB page.

Doesn't work for big MAX, JSON, XML data.

Data warehouse on 2016+?
Consider columnstore indexes instead.

2.1 p48



Signs columnstore will work well

- Tables with 100M+ rows, 100GB+, dozens of columns
- Loaded in batches (like 100k+ rows at a time)
- Not usually updated or deleted
- SQL Server 2016 or newer, 64GB+ RAM
- Data is highly repetitive (compressible)
- Learn more: Columnscore.com, my columnstore classes

2.1 p49



Loading data fast

“Should I drop my indexes before loading?”

Data Loading Performance Guide (2009):

[https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008/dd425070\(v=sql.100\)](https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008/dd425070(v=sql.100))

Doesn't cover:

- Columnstore indexes
- Database mirroring, AGs
- Azure SQL DB and its variants

Even more guidance:
Google for Paul White minimal logging

Summarizing Minimal Logging Conditions

To assist you in understanding which bulk load operations will be minimally logged and which will not, the following table lists the possible combinations.

Table Indexes	Rows in table	Hints	Without TF 610	With TF 610	Concurrent possible
Heap	Any	TABLOCK	Minimal	Minimal	Yes
Heap	Any	None	Full	Full	Yes
Heap + Index	Any	TABLOCK	Full	Depends (3)	No
Cluster	Empty	TABLOCK, ORDER (1)	Minimal	Minimal	No
Cluster	Empty	None	Full	Minimal	Yes (2)
Cluster	Any	None	Full	Minimal	Yes (2)
Cluster	Any	TABLOCK	Full	Minimal	No
Cluster + Index	Any	None	Full	Depends (3)	Yes (2)
Cluster + Index	Any	TABLOCK	Full	Depends (3)	No

2.1 p50



When indexing temp tables

In Fundamentals of TempDB, you learned:

- If a table is modified after it's created, then its execution plans get recompiled
- Bad if a query runs frequently
- Good if a query rarely runs, and needs new execution plans each time

So that affects how you create indexes.

2.1 p51



How that affects CREATE INDEX

If you want recompiles, add indexes in a separate statement, after the CREATE TABLE.

If you want fixed plans, create indexes inline:

```
CREATE TABLE #Users
(
    Id INT PRIMARY KEY CLUSTERED,
    DisplayName NVARCHAR(40) INDEX IX_DisplayName,
    Location NVARCHAR(100),
    INDEX IX_Location_DisplayName NONCLUSTERED (Location, DisplayName));
```

2.1 p52





The A in the D.E.A.T.H. Method

Use Clippy to rapidly go from 0 to ~5 indexes.

Don't try digging into which query plans are involved yet (unless you're on SQL 2019+.)

Too many columns? Try without the includes first.

Selectivity involves the query's search space.
Try to guess what the query filters on.

