Introducing My D.E.A.T.H. Method

Starting with Deduplicating and Eliminating indexes.

1.2 p1

What we'll cover

My 10 & 5 guideline

Demoing what happens at the extremes
(too many or too few indexes)

My D.E.A.T.H. Method to get closer to 10 & 5

Doing the D.E. parts first on paper

1.2 p2

Mo indexes, mo fields, mo problems

Longer delete/update/inserts (DUIs)

Slower storage response time (amplified writes)

The more blocking can become a problem

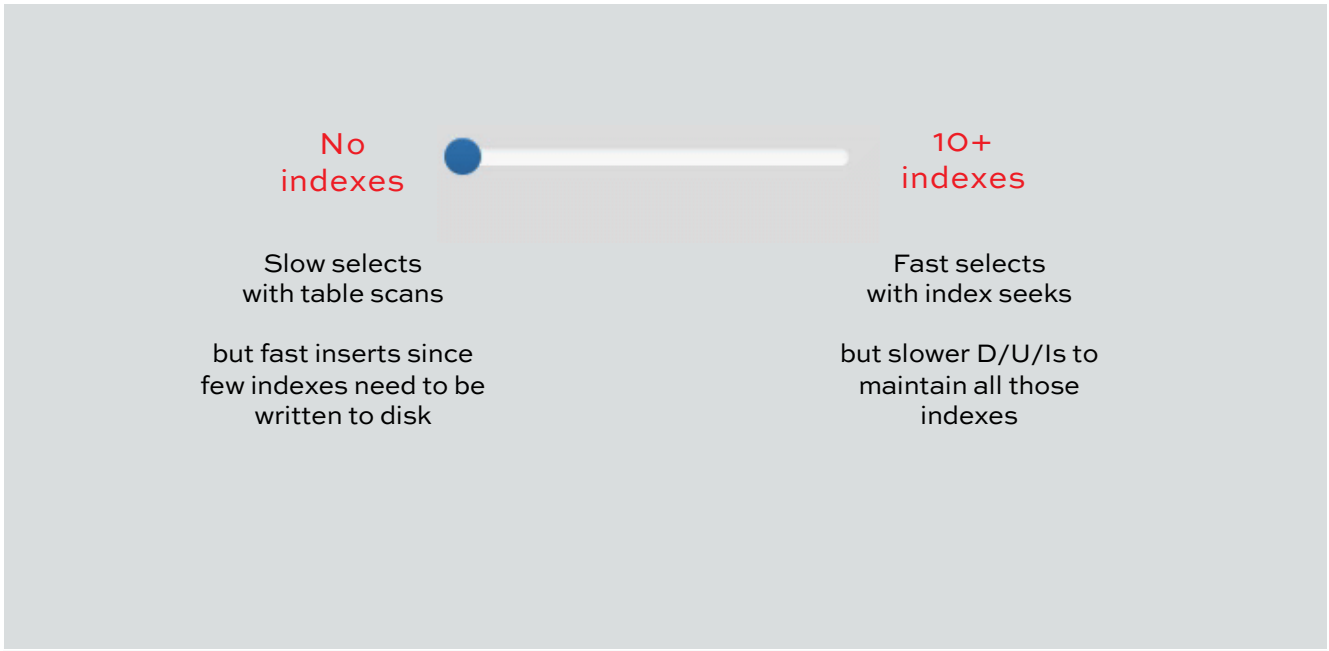
The longer your maintenance jobs take (backups, corruption checks, index rebuilds, stats updates)

The less effective memory you have
(because DUIs are done in memory)

1.2 p3

Goal:
the smallest number of indexes necessary to support your workload.

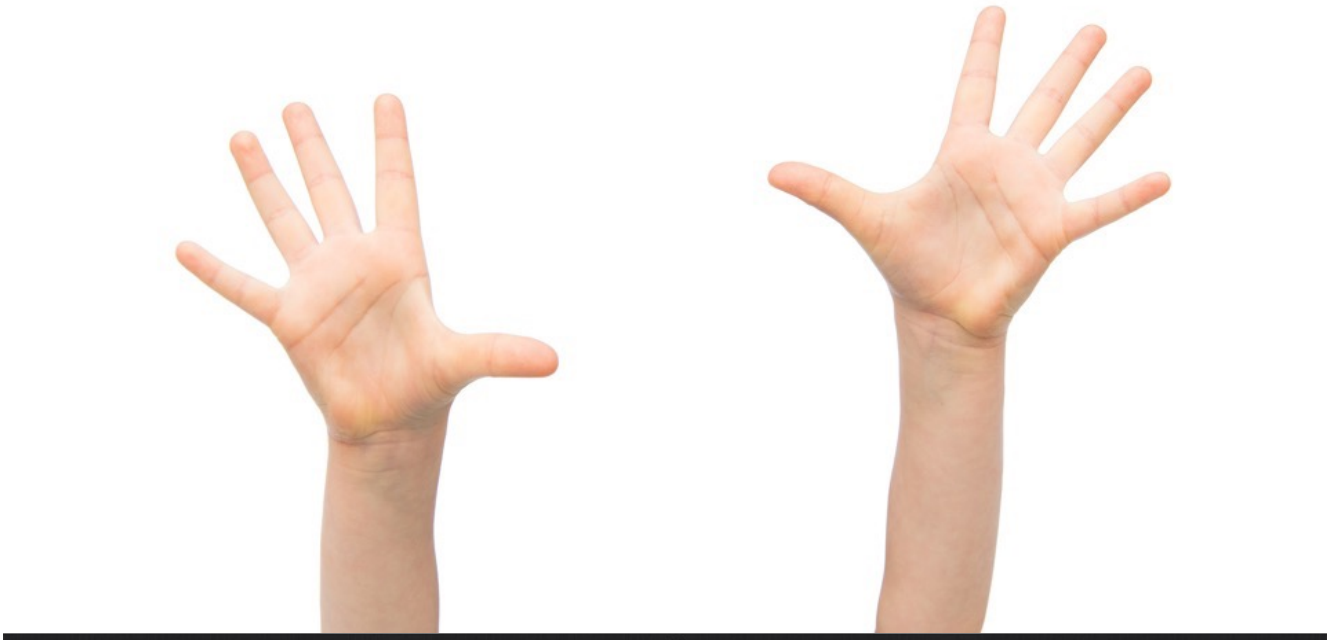
1.2 p4



1.2 p5

Generally, I aim for 5 or less indexes with 5 or less fields each

1.2 p6



1.2 p7

It's a starting point, not a rule.

More indexes can be fine when:

- Read-only (or read-biased) tables
- Very, very good hardware (memory, SSDs)
- When DUI speed doesn't matter

Less indexes may be required when:

- Ingestion speed is absolutely critical
- Read speed doesn't matter

1.2 p8

On decent hardware, 10 & 5.

Typical gaming PC:
12-16 CPU cores, 64GB RAM, 2TB SSD.

If your SQL Server is at least as fast
as a typical gaming PC,
aim for ~10 well-tuned indexes per table.

You'll still probably be able to avoid blocking.

I'll explain it with 6 indexes.

```
/* Create a few indexes: */
CREATE INDEX IX_LastAccessDate ON dbo.Users(LastAccessDate);
CREATE INDEX IX_Age ON dbo.Users(Age) INCLUDE (LastAccessDate);
CREATE INDEX IX_DisplayName ON dbo.Users(DisplayName) INCLUDE (LastAccessDate);
CREATE INDEX IX_DownVotes ON dbo.Users(DownVotes) INCLUDE (LastAccessDate);
CREATE INDEX IX_Location ON dbo.Users(Location) INCLUDE (LastAccessDate);
CREATE INDEX IX_Reputation ON dbo.Users(Reputation) INCLUDE (LastAccessDate);
GO
```

I'll cover:

- How they show up in DUI plans
- How they show up in performance metrics

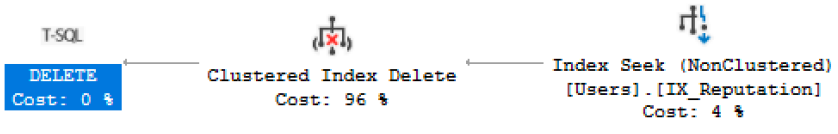
When we delete a few rows...

```
/* Get the estimated plans for these - don't actually run 'em: */
DELETE dbo.Users WHERE Reputation = 1000000;
```

150 %

Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
DELETE dbo.Users WHERE Reputation = 1000000
```

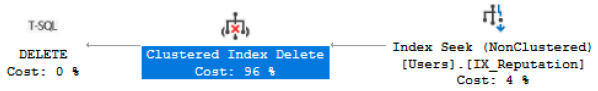


```
DELETE dbo.Users WHERE Reputation = 1000000;
```

150 %

Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
DELETE dbo.Users WHERE Reputation = 1000000
```



Clustered Index Delete

Delete rows from a clustered index.

Physical Operation	Clustered Index Delete
Logical Operation	Delete
Estimated Execution Mode	Row
Estimated Operator Cost	0.070007 (96%)
Estimated I/O Cost	0.07
Estimated Subtree Cost	0.0732901
Estimated CPU Cost	0.000007
Estimated Number of Executions	1
Estimated Number of Rows	1.00493
Estimated Row Size	9 B
Node ID	0

Object

[StackOverflow2013].[dbo].[Users].[PK_Users_Id],
[StackOverflow2013].[dbo].[Users].[IX_LastAccessDate],
[StackOverflow2013].[dbo].[Users].[IX_Age],
[StackOverflow2013].[dbo].[Users].[IX_DisplayName],
[StackOverflow2013].[dbo].[Users].[IX_DownVotes],
[StackOverflow2013].[dbo].[Users].[IX_Location],
[StackOverflow2013].[dbo].[Users].[IX_Reputation]

The truth is down there

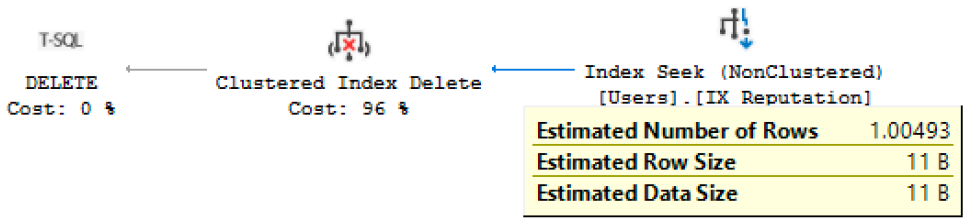
“Narrow” plan for small data

```
DELETE dbo.Users WHERE Reputation = 1000000;
```

150 %

Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
DELETE dbo.Users WHERE Reputation = 1000000
```



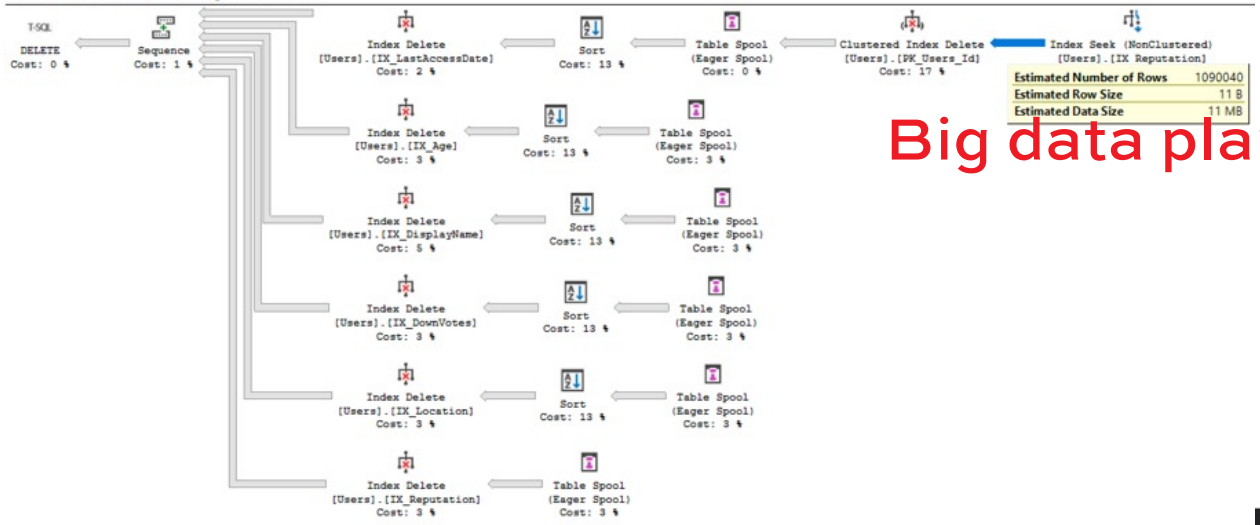
1.2 p13

```
DELETE dbo.Users WHERE Reputation = 1;
```

150 %

Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
DELETE dbo.Users WHERE Reputation = 1
```



Big data plan

1.2 p14

That's a "wide" plan

The Reputation = 1 finds a lot more rows to delete

- Rep = 1 estimated number of rows: millions
- Rep = 1,000,000 est number of rows: 1

If SQL Server is going to delete lots of rows,
it decides to use a wider plan, like re-sorting all of the data by index orders to make the deletes go quicker.

Indexes add overhead

SQL Server has to maintain them on inserts, updates deletes

Just because you don't see individual costs for them in all execution plans doesn't mean they are free!

That cost is just hidden from you most of the time (but we show it in sp_BlitzCache)

Results | Messages

	Database	Query Text	Query Type	Warnings
1	StackOverflow	INSERT dbo.Posts (Id, AcceptedAnswerId, AnswerCount, ...	Statement	Plan created last 4hrs, 11 Indexes Modified
2	StackOverflow	INSERT INTO [dbo].[Posts]([Id],[AcceptedAnswerId],[Answ...	Statement	Trivial Plans, Plan created last 4hrs, 11 Indexes Modified
3	StackOverflow	UPDATE [dbo].[Users] set [Reputation] = @1 WHERE [Id]...	Statement	Trivial Plans, Plan created last 4hrs
4	StackOverflow	DELETE TOP (250) FROM dbo.Posts WHERE PostTypeId...	Statement	Missing Indexes (1), Parallel, Plan created last 4hrs, 11 Indexes Modified
5	StackOverflow	UPDATE TOP (251) dbo.Posts SET AcceptedAnswerId = ...	Statement	Missing Indexes (1), Parallel, Multiple Plans, Plan created last 4hrs, 11 Indexes Modified
6	StackOverflow	UPDATE TOP (250) dbo.Posts SET AcceptedAnswerId = ...	Statement	Missing Indexes (1), Parallel, Multiple Plans, Plan created last 4hrs, 11 Indexes Modified

Now let's measure it at scale

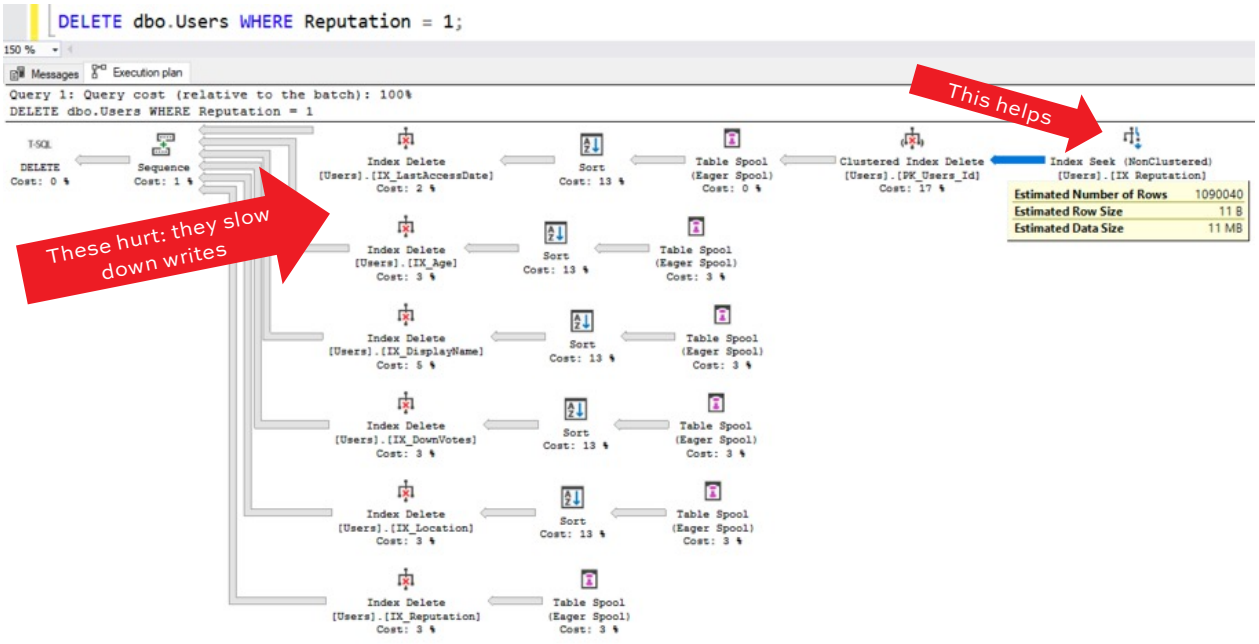
If I need to run this:

```
UPDATE dbo.Users
    SET LastAccessDate = GETDATE()
    WHERE Reputation = @Something;
```

Then:

- An index on Reputation helps me
- Every index with LastAccessDate hurts me

1.2 p17



1.2 p18

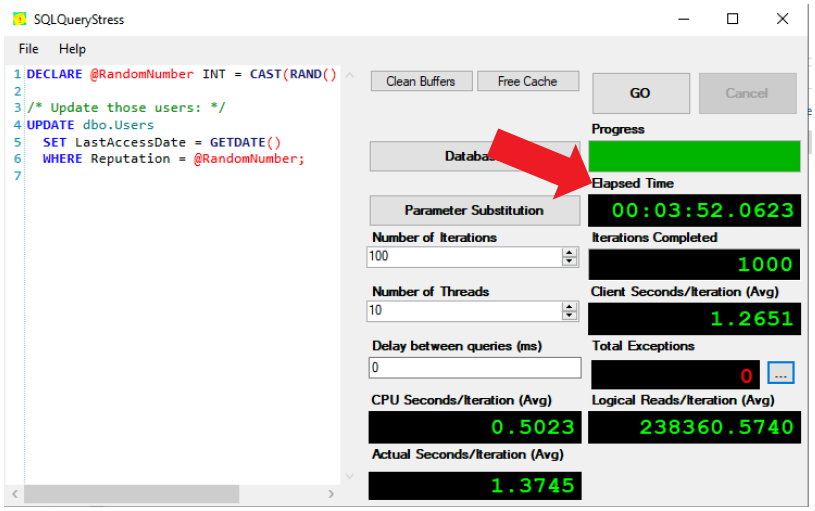
Demoing with SQLQueryStress

I'm going to run this across 10 concurrent threads:

```
/* Generate a random number: */
DECLARE @RandomNumber INT = CAST(RAND() * 100 AS INT);

/* Update those users: */
UPDATE dbo.Users
   SET LastAccessDate = GETDATE()
   WHERE Reputation = @RandomNumber;
```


3 minutes, 52 seconds



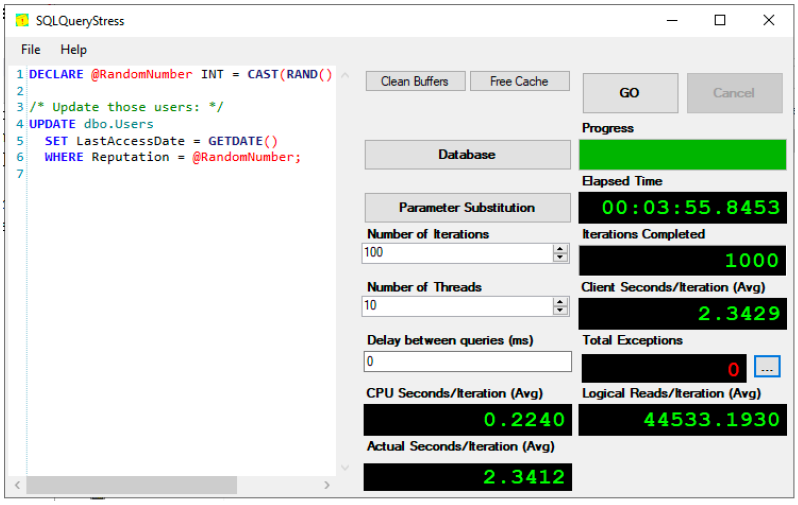
Stack Overflow 2013 (50GB)

All solid state storage

Database cached in RAM

Bottleneck: blocking

Drop all indexes, try again:

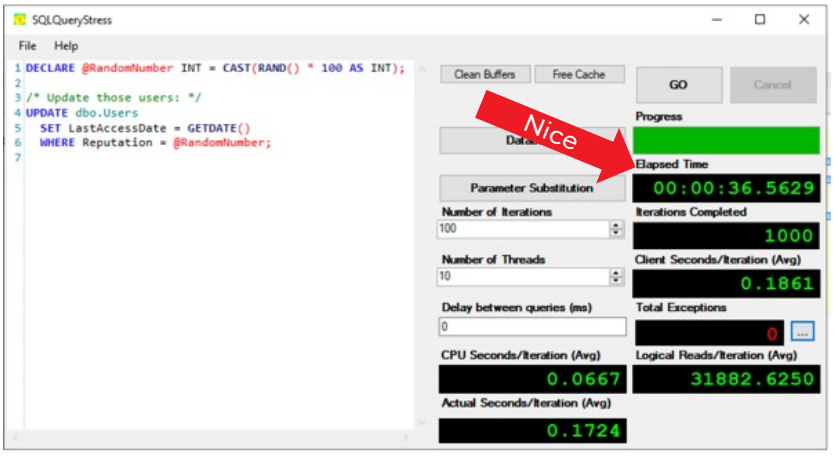


This time, to find the users to update, we have to scan the clustered index

Bottleneck: blocking

This extreme sucks too.

Add just one index on Reputation



Now, we can seek into Reputation to find the users we want to update.

We only have to update LastAccessDate on the clustered index.

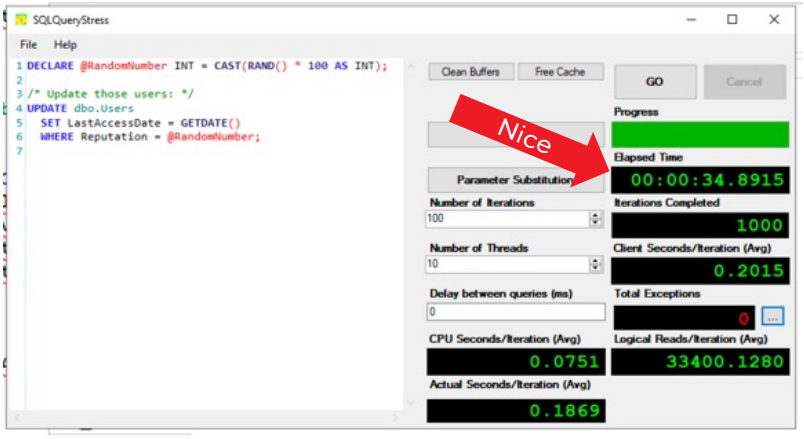
It's not included anywhere.

Add in the rest of our indexes

```
/* All our indexes, but WITHOUT the LastAccessDate include: */
EXEC DropIndexes;
GO
CREATE INDEX IX_Age ON dbo.Users(Age);
CREATE INDEX IX_DisplayName ON dbo.Users(DisplayName);
CREATE INDEX IX_DownVotes ON dbo.Users(DownVotes);
CREATE INDEX IX_Location ON dbo.Users(Location);
CREATE INDEX IX_Reputation ON dbo.Users(Reputation);
GO
```

But don't include LastAccessDate, nor do we add an index on that column as a key either.

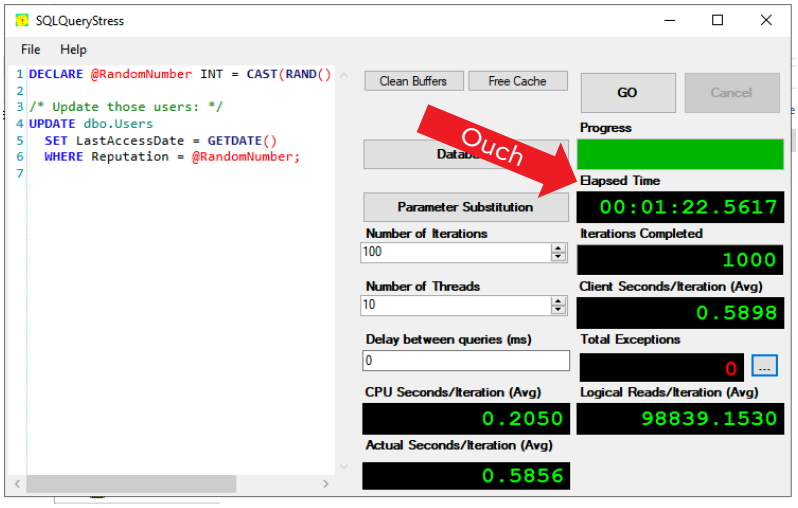
Add in the rest of our indexes



But don't include LastAccessDate, nor do we add an index on that column as a key either.

Still really fast.

Add an index on LastAccessDate



So now we have to update LastAccessDate in 2 places: the clustered index, and a new index on LastAccessDate.

Instantly >2x slower!

Scorecard

	Test time
No nonclustered indexes at all	3m:56sec
One index on Reputation (to help us find the rows to update)	0:37
Lots of indexes (but none have LastAccessDate as a key or an included column)	0:35
Add one index on LastAccessDate	1:23
All indexes include LastAccessDate	3:52

Sweet spot

No indexes

10+ indexes

Slow selects with table scans

but fast inserts since few indexes need to be written to disk

Fast selects with index seeks

but slower D/U/Is to maintain all those indexes

1.2 p27

"But I have a 1TB table with 250 columns, and I only have 32GB RAM, and cloud storage!"

1.2 p28

You might be starting higher.

Some of you inherited an existing database where:

- Everybody just added missing indexes from plans, scripts, and dynamic management views
- Everybody ran the Database Tuning Advisor (which doesn't care about slowing down DUIs)
- Everybody guessed at what indexes they needed
- Everybody indexed every foreign key

1.2 p29

5 & 5 is just a guideline.

Even just one index on the wrong field might be a showstopper for your workloads.

The ~5 fields per index guideline includes includes: includes take up space too and have to be updated.

The faster you want to go, the more you have to:

- Understand which fields are hot
- Avoid including those fields in indexes
- Run experiments to measure impact

Now, let's figure out how to make an existing table better.

1.2 p30

My D.E.A.T.H. index method

Dedupe – reduce overlapping indexes

Eliminate – unused indexes

Add – badly needed missing indexes

Tune – indexes for specific queries

Heaps – usually need clustered indexes

1.2 p31

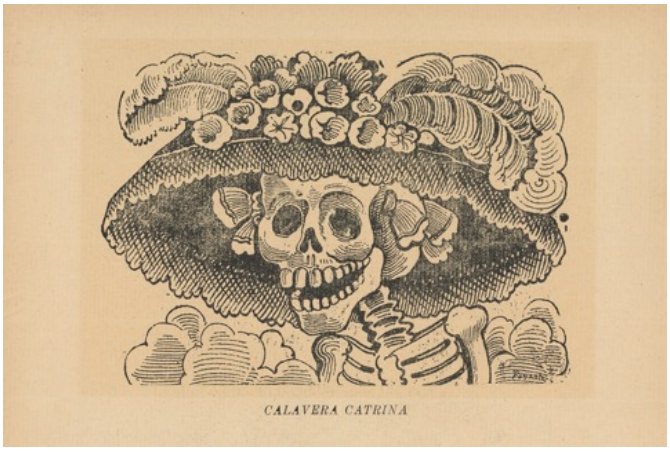
Dia de los Muertos (Day of the Dead)

November 1 & 2

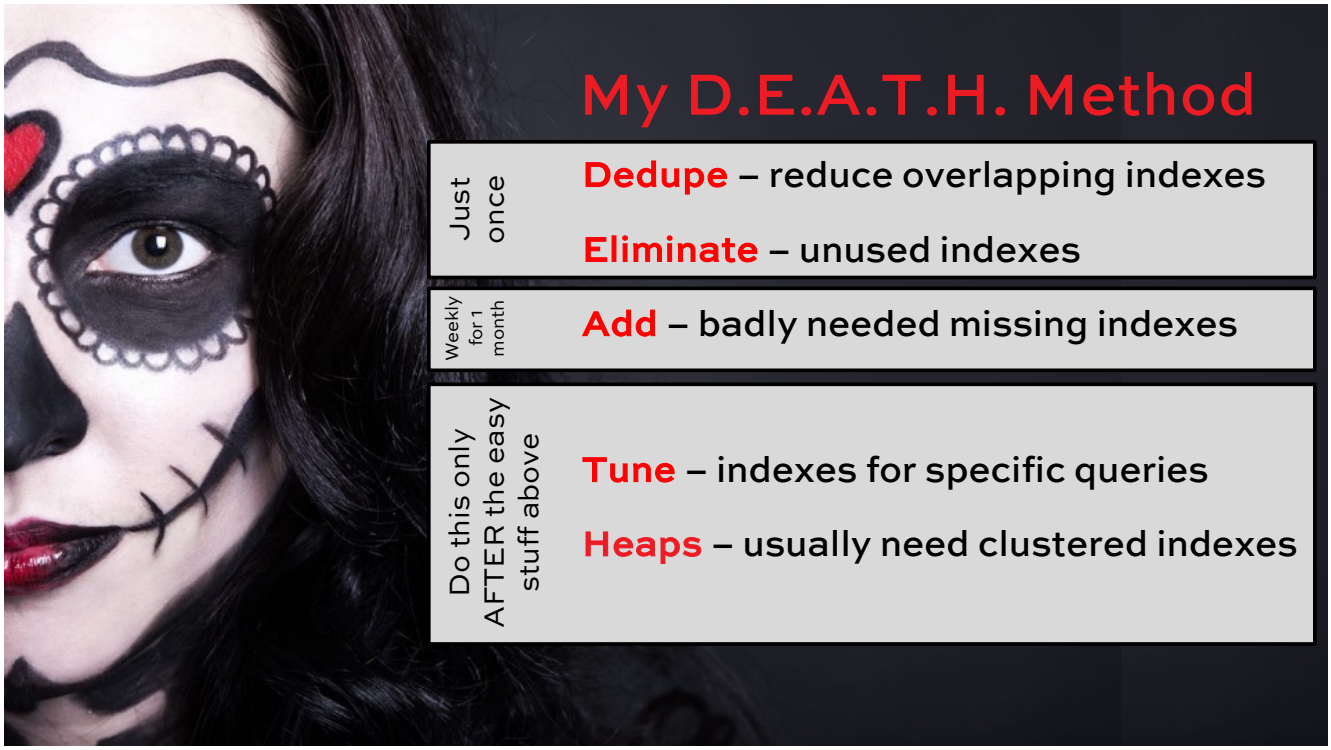
Praying for families and friends who passed

Celebration, not mourning

Key figure:
La Calavera Catrina



1.2 p32



Dedupe/eliminate: easy, safe* one-time step.

Quick: you can do this across lots of tables in an hour.

Safe: as long as you're diligent, it's hard to mess up.

Not necessarily a huge bang for the buck:
users may not notice a big speed-up.

We're trying to make it easier to add indexes next.

**Nothing is ever totally safe.*

Adding indexes is a little harder.

The missing index DMVs have a lot of gotchas.

You can add indexes that make queries slower.

1.2 p35

Tuning queries is less easy.

- Identify the query that needs to be tuned
- Test to make sure you're getting the right plan (and not a victim of parameter sniffing)
- Read the execution plan, decide what to change
- Change it, make sure it's faster while producing exactly the same results
- Deploy it to production (peer review, source control, change the app, etc)

All this can take hours/days for just one query.

1.2 p36

Heaps can be easy to fix, but...

Sometimes the table doesn't have a good candidate for clustering, and we have to add one

Sometimes it just doesn't need a clustered index

Changing the clustered index can be invasive

If you can fix these quickly, great.
Otherwise, focus on the low-hanging fruit first.

1.2 p37

So first, D/E.

Dedupe – reduce overlapping indexes

Eliminate – unused indexes

The deduplication process

1. Identical indexes: eliminate all but one of 'em
2. Borderline identical: see if you can merge 'em:
    1. Identify the key superset (key order matters)
    2. Combine the included columns
    3. Review to see if it's a monster:
       we're aiming for the 5/5 rule
    4. Create the new index
    5. Drop the rest

1.2 p39

Should you drop or disable?

Disabled indexes:

- Easy to re-enable: just ALTER INDEX ... REBUILD
- Definitions are still there, easy to script out
- But anyone's rebuild can enable it again,
  including index maintenance plans & scripts

Dropped indexes are gone for good. I usually do that.

1.2 p40

Dedupe & eliminate dbo.Users.

ID	Key Columns	Included Columns	Reads	Writes
1	Id (clustered index)	(all of 'em)	4,932,101	695,023
2	LastAccessDate	Id	2,100,399	695,023
3	LastAccessDate, DisplayName, Age	Reputation	5,011,022	695,023
4	LastAccessDate, DisplayName	Location, CreationDate	1,302,230	695,023
5	DisplayName	LastAccessDate	571,609	1,503
6	WebsiteUrl	Id	0	105,411

1.2 p41

This guy's only slowing us down.

ID	Key Columns	Included Columns	Reads	Writes
1	Id (clustered index)	(all of 'em)	4,932,101	695,023
2	LastAccessDate	Id	2,100,399	695,023
3	LastAccessDate, DisplayName, Age	Reputation	5,011,022	695,023
4	LastAccessDate, DisplayName	Location, CreationDate	1,302,230	695,023
5	DisplayName	LastAccessDate	571,609	1,503
6	WebsiteUrl	Id	0	105,411

1.2 p42

The rest are all getting read.

ID	Key Columns	Included Columns	Reads	Writes
1	Id (clustered index)	(all of 'em)	4,932,101	695,023
2	LastAccessDate	Id	2,100,399	695,023
3	LastAccessDate, DisplayName, Age	Reputation	5,011,022	695,023
4	LastAccessDate, DisplayName	Location, CreationDate	1,302,230	695,023
5	DisplayName	LastAccessDate	571,609	1,503

1.2 p43

#2-4 are suspiciously similar.

ID	Key Columns	Included Columns	Reads	Writes
1	Id (clustered index)	(all of 'em)	4,932,101	695,023
2	LastAccessDate	Id	2,100,399	695,023
3	LastAccessDate, DisplayName, Age	Reputation	5,011,022	695,023
4	LastAccessDate, DisplayName	Location, CreationDate	1,302,230	695,023
5	DisplayName	LastAccessDate	571,609	1,503

1.2 p44

One index to merge #2/#3/#4

KEY

LastAccessDate, DisplayName, Age

INCLUDED COLUMNS

Reputation, Location, CreationDate

1.2 p45

The format I like to get from DBAs

```
/* SQLPROD1 Index Changes
Brent Ozar, 2021-11-21

Drop these indexes because they're not getting used: */
DROP INDEX WebsiteUrl ON dbo.Users;
GO

/* Undo script:
CREATE INDEX WebsiteUrl ON dbo.Users(WebsiteUrl) INCLUDE(Id);

Merge these together because they're overlapping:
*/
CREATE INDEX LastAccessDate_DisplayName_Age ON dbo.Users
    (LastAccessDate, DisplayName, Age) INCLUDE (Reputation, Location, CreationDate);

DROP INDEX LastAccessDate ON dbo.Users;
DROP INDEX IX_LastAccessDate_DisplayName_Age ON dbo.Users;
DROP INDEX LastAccessDate_DisplayName ON dbo.Users;
GO
/* Undo script:
DROP INDEX LastAccessDate_DisplayName_Age ON dbo.Users;
CREATE INDEX LastAccessDate ON dbo.Users(LastAccessDate) INCLUDE (Id);
CREATE INDEX IX_LastAccessDate_DisplayName_Age ON dbo.Users
    (LastAccessDate, DisplayName, Age) INCLUDE (Reputation);
CREATE INDEX LastAccessDate_DisplayName ON dbo.Users
```

1.2 p46

Now I'm down to these.

ID	Key Columns	Included Columns	Reads	Writes
1	Id (clustered index)	(all of 'em)	4,932,101	695,023
2	LastAccessDate, DisplayName, Age	Reputation, Location, CreationDate	2,100,399	695,023
5	DisplayName	LastAccessDate	571,609	1,503

Are they the same thing?

Can I get rid of #5?

1.2 p47

Which index will this query pick?

```
SELECT DisplayName FROM dbo.Users
WHERE LastAccessDate = GETDATE()
```

ID	Key Columns	Included Columns	Reads	Writes
1	Id (clustered index)	(all of 'em)	4,932,101	695,023
2	LastAccessDate, DisplayName, Age	Reputation, Location, CreationDate	2,100,399	695,023
5	DisplayName	LastAccessDate	571,609	1,503

1.2 p48

Which index will this query pick?

```
SELECT LastAccessDate FROM dbo.Users
WHERE DisplayName = 'Brent Ozar'
```

ID	Key Columns	Included Columns	Reads	Writes
1	Id (clustered index)	(all of 'em)	4,932,101	695,023
2	LastAccessDate, DisplayName, Age	Reputation, Location, CreationDate	2,100,399	695,023
5	DisplayName	LastAccessDate	571,609	1,503

1.2 p49

Indexes are only identical if they have the same leading field.

ID	Key Columns	Included Columns	Reads	Writes
1	Id (clustered index)	(all of 'em)	4,932,101	695,023
2	LastAccessDate, DisplayName, Age	Reputation, Location, CreationDate	2,100,399	695,023
5	DisplayName	LastAccessDate	571,609	1,503

You have to be really careful when dropping an index that's reportedly getting used. Let's talk about why.

1.2 p50

What we learned

Extremes are bad:

- Too many indexes on too many fields means too many writes.
- No indexes at all means table scans, which can be slow for writes too.

We have to find the sweet spot:

- Enough indexes to make queries fast, on just the right fields
- Not so many indexes & fields that changes are slow

What we learned

Start by aiming for:

- ~5 indexes per table
  (~10 on gaming PC quality HW)
- ~5 or less fields per index

Use the D.E.A.T.H. Method to get there, starting with Deduping and Eliminating first.

Next, let's learn the drawbacks of the DMVs that give us this data.

1.2 p53