The D.E.A.T.H. Method: Tuning Indexes for Specific Queries

We're going to go a little out of order.

1.5 p1

D.E.A.: fast, affects many queries.

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

1.5 p2

The workflow

Deduplicating, **E**liminating indexes:
can be done in a matter of hours of focused work.

Adding indexes requires

- Requires more close examination of existing indexes
- Thinking about key order, selectivity
- Understanding how Clippy thinks, and remixing his ideas

Tuning indexes for specific queries requires:

- Finding the right queries to tune
- Testing adding an index, making sure it gets used
- Measuring the difference before & after
- Sometimes even tuning the query itself
- Realistically: this is 1-4 hours **per query.**

1.5 p3

This part is tricky:

- **Understanding how Clippy thinks, and remixing his ideas**

So before we do that, we're going to tune indexes for specific queries – and while you do it, you'll be able to see Clippy's recommendations.

In your next lab, you'll work with Clippy's ideas for this same workload, and you'll see how he goofs up.

1.5 p4

Let’s do this part first.

Tuning indexes for specific queries requires:

- Finding the right queries to tune
- Testing adding an index, making sure it gets used
- Measuring the difference before & after
- Sometimes even tuning the query itself
- Realistically: this is 1-4 hours per query.

Let's start with an easy one.

- dbo.Users
  - Columns
    - Id (PK, int, not null)
    - AboutMe (nvarchar(max), null)
    - Age (int, null)
    - CreationDate (datetime, not null)
    - DisplayName (nvarchar(40), not null)
    - DownVotes (int, not null)
    - EmailHash (nvarchar(40), null)
    - LastAccessDate (datetime, not null)
    - Location (nvarchar(100), null)
    - Reputation (int, not null)
    - UpVotes (int, not null)
    - Views (int, not null)
    - WebsiteUrl (nvarchar(200), null)
  - Keys
  - Constraints
  - Triggers
  - Indexes
    - PK_Users (Clustered)

```
SELECT Id
FROM dbo.Users
WHERE DisplayName = 'Brent Ozar'
  AND WebsiteUrl = 'https://www.brentozar.com';
```

Build the perfect index.

1.5 p6

Reminder from Fundamentals of Index Tuning

If all you have is equality searches, field order doesn't matter much.

Order starts to matter when:

- We have inequality searches: <, >, <>, IS NOT NULL, IN, etc.
- There's an ORDER BY
- When we join to other tables

1.5 p7

Next query: competitive Krock

http://data.stackexchange.com/stackoverflow/query/6925/newer-users-with-more-reputation-than-me

```sql
-- Newer Users with More Reputation Than Me
-- Find useres that have been members for a shorter time than me
-- but have more reputation points.


declare @UserId int = ##UserId##

select u.id as [User Link], u.Reputation, u.Reputation - me.Reputation as Difference
from users u cross join users me
where me.id = @UserId
and u.CreationDate > me.CreationDate
and u.Reputation > me.Reputation
```

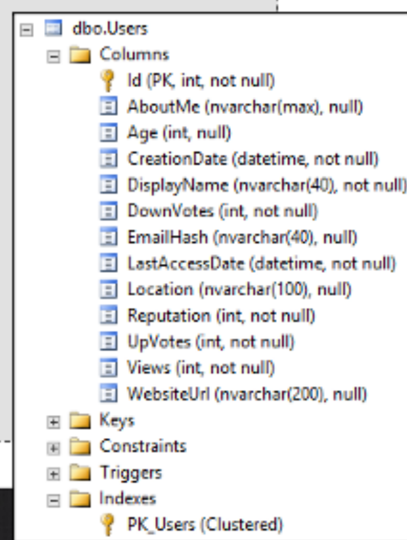
created **jul 11 10**

krock

1.5 p8

usp_Q6925 – cleaned up a little

```
SELECT
     u.Id as [User Link],
     u.Reputation,
     u.Reputation - me.Reputation
          as Difference
FROM dbo.Users me
JOIN dbo.Users u on
     u.CreationDate > me.CreationDate
     and u.Reputation > me.Reputation
WHERE me.Id = @UserId
ORDER BY u.Reputation DESC
GO
```



Design a great nonclustered index

```
SET STATISTICS IO, TIME ON;
EXEC usp_Q6925 @UserId = 26837
```

Deliverables:

- Measure the query before
- Write out the index definition
- Measure the query afterwards
- Bonus: try other parameters, see how they go

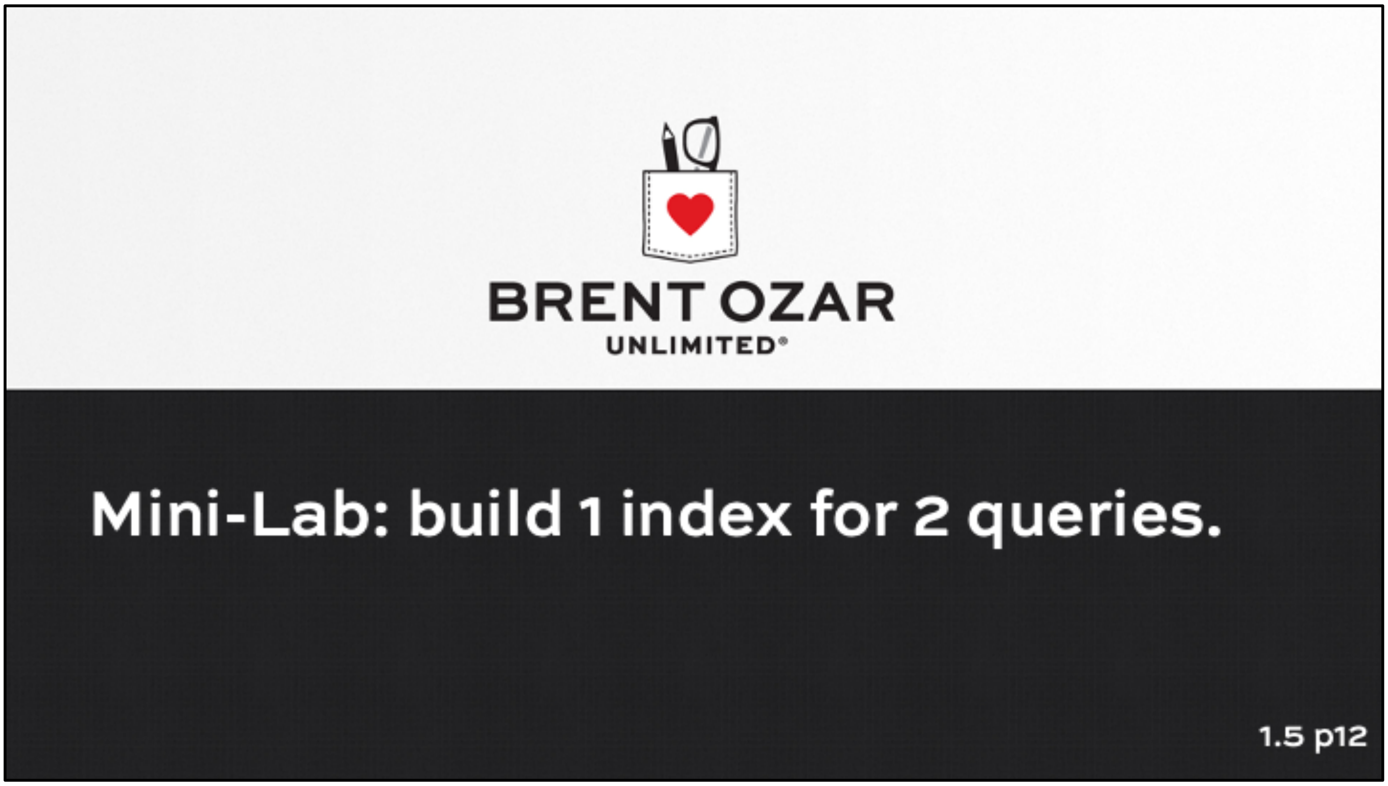
1.5 p10

Next takeaway

With range scans (inequality searches), field order starts to matter.

Generally speaking, the more selective fields should go first, but test with STATS IO.

1.5 p11


BRENT OZAR
UNLIMITED®
Mini-Lab: build 1 index for 2 queries.
1.5 p12

Build 1 index to speed up both:

dbo.usp_PostsByCommentCount

- @PostTypeId = 2

dbo.usp_PostsByScore

- @PostTypeId = 2
- @CommentCountMinimum = 2

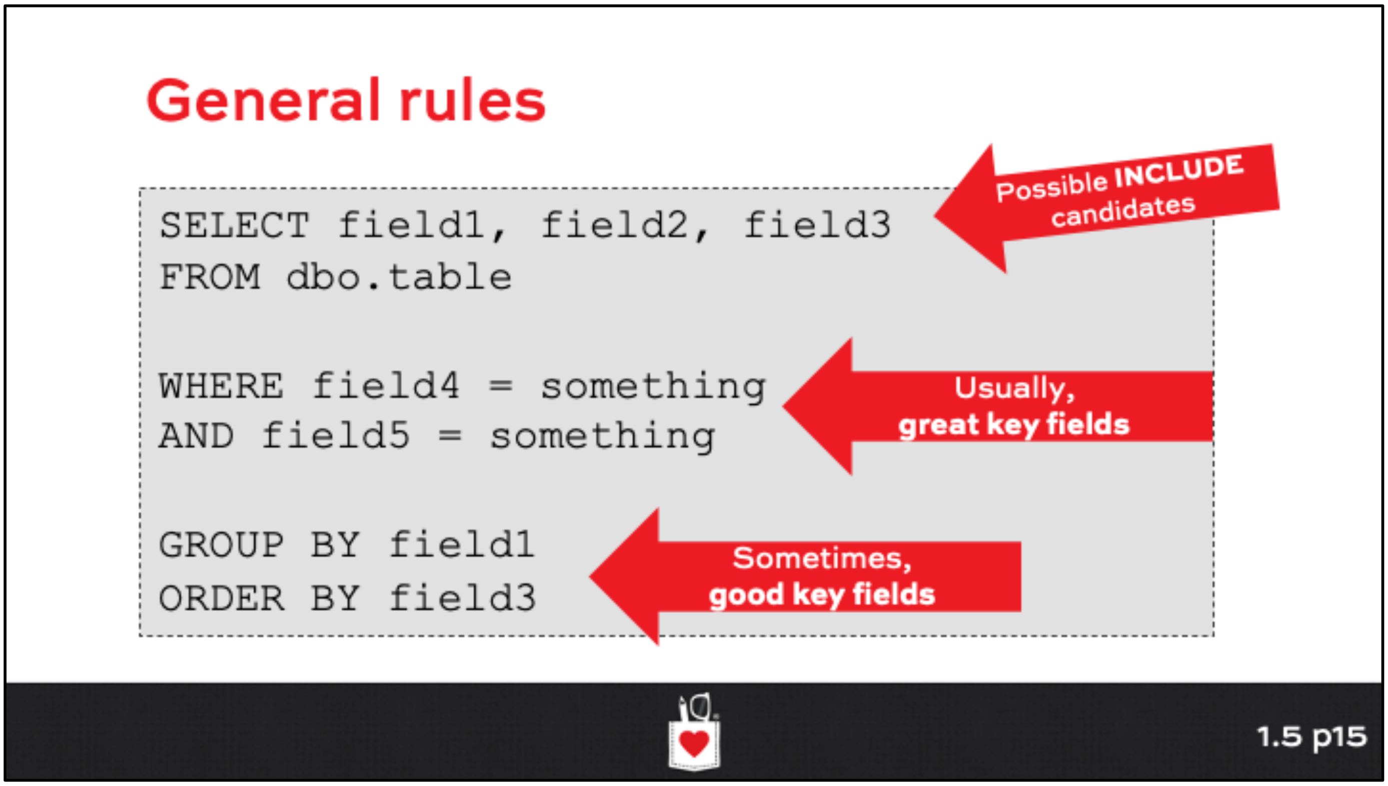
Don't worry about the number of indexes the tables already have: just build perfect indexes for each.

1.5 p13

Parameter sniffing affects missing index requests.

Gotta make sure you're tuning the query that you really want to go faster.

1.5 p14


General rules
SELECT field1, field2, field3
FROM dbo.table
WHERE field4 = something
AND field5 = something
GROUP BY field1
ORDER BY field3
Possible INCLUDE candidates
Usually, great key fields
Sometimes, good key fields
1.5 p15

Query 1: look for key columns

```
SELECT TOP 10 CommentCount,Score,ViewCount
FROM dbo.Posts
WHERE PostTypeId=@PostTypeId
ORDER BY CommentCount DESC;
```

An equality comparison

1.5 p16

$int is a parameterized value that can be any valid value– not something for a filtered index

Query 1: look for more key columns

```
SELECT TOP 10 CommentCount,Score,ViewCount
FROM dbo.Posts
WHERE PostTypeId=@PostTypeId
ORDER BY CommentCount DESC;
```

TOP / ORDER BY

1.5 p17

Query 1: look for possible includes

```
SELECT TOP 10 CommentCount,Score,ViewCount
FROM dbo.Posts
WHERE PostTypeId=@PostTypeId
ORDER BY CommentCount DESC;
```

1.5 p18

Query 1 – scratch paper

Possible keys (order still negotiable)

- PostTypeId – an equality predicate
- CommentCount – TOP 10 ordered by this DESC

Possible includes

- Score
- ViewCount

1.5 p19

Query 2: look for key columns

```
SELECT TOP 10 Id, CommentCount, Score
FROM dbo.Posts
WHERE CommentCount >= @CommentCountMin
AND PostTypeId=@PostTypeId
ORDER BY Score DESC;
```

An equality comparison

1.5 p20

Query 2: look for key columns

```
SELECT TOP 10 Id, CommentCount, Score
FROM dbo.Posts
WHERE CommentCount >= @CommentCountMin
AND PostTypeId=@PostTypeId
ORDER BY Score DESC;
```

An inequality comparison

1.5 p21

Query 2: look for even more keys

```
SELECT TOP 10 Id, CommentCount, Score
FROM dbo.Posts
WHERE CommentCount >= @CommentCountMin
AND PostTypeId=@PostTypeId
ORDER BY Score DESC;
```

TOP / ORDER BY

1.5 p22

Query 2: look for possible includes

```
SELECT TOP 10 Id, CommentCount, Score
FROM dbo.Posts
WHERE CommentCount >= @CommentCountMin
AND PostTypeId=@PostTypeId
ORDER BY Score DESC;
```

1.5 p23

Query 2 – scratch paper

Possible keys (order still negotiable)

- PostTypeId – an equality predicate
- Score – top 10 ordered by this DESC
  - This is probably most efficient as the second column based on selectivity – a lot of low score things may get comments over whatever threshold is passed in
- CommentCount – an inequality >=

Possible includes

- Id
- (the other possible includes are already key cols)

1.5 p24

Scratch paper

A great index for query 1:

- KEY (PostTypeId, CommentCount DESC)
- INCLUDE (Score, ViewCount)

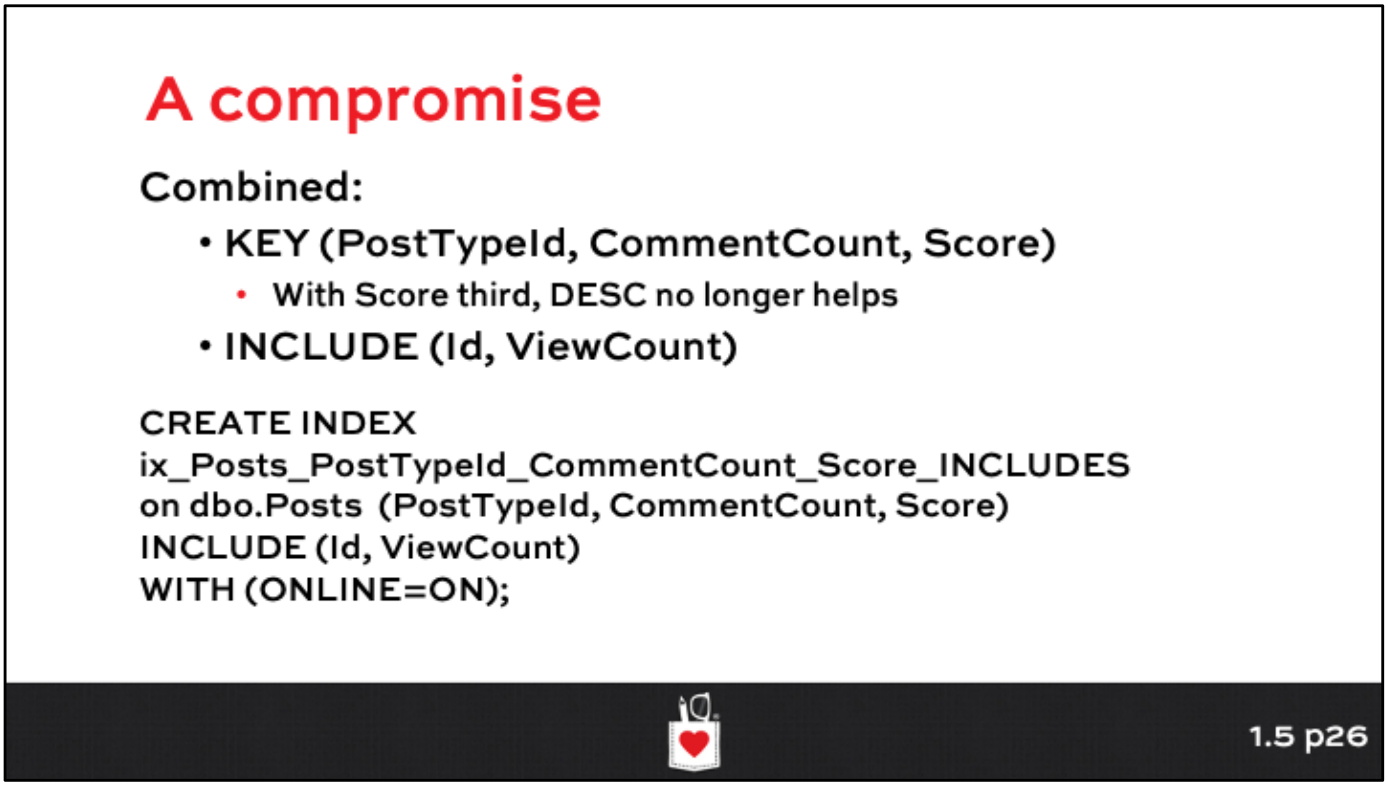
A great index for query 2:

- KEY (PostTypeId, Score DESC, CommentCount)
- INCLUDE (Id)

These queries have different ‘ideal’ indexes

But if every query gets an ideal index, we’ve got other problems

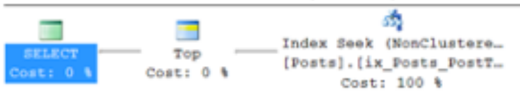
1.5 p25


A compromise
Combined:
• KEY (PostTypeId, CommentCount, Score)
• With Score third, DESC no longer helps
• INCLUDE (Id, ViewCount)
CREATE INDEX
ix_Posts_PostTypeId_CommentCount_Score_INCLUDES
on dbo.Posts (PostTypeId, CommentCount, Score)
INCLUDE (Id, ViewCount)
WITH (ONLINE=ON);
1.5 p26

Example query plans

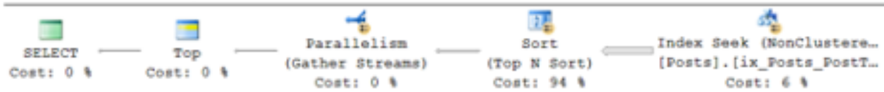
KEY (PostTypeId, CommentCount, Score) INCLUDE (Id, ViewCount)

Query 1: Query cost (relative to the batch): 0%
SELECT TOP 10 CommentCount, Score, ViewCount FROM dbo.Posts WHERE PostTypeId=2 ORDER BY CommentCount DESC



logical reads 3

Query 2: Query cost (relative to the batch): 100%
SELECT TOP 10 Id, CommentCount, Score FROM dbo.Posts WHERE CommentCount >= 2 AND PostTypeId=2 ORDER BY Score DESC

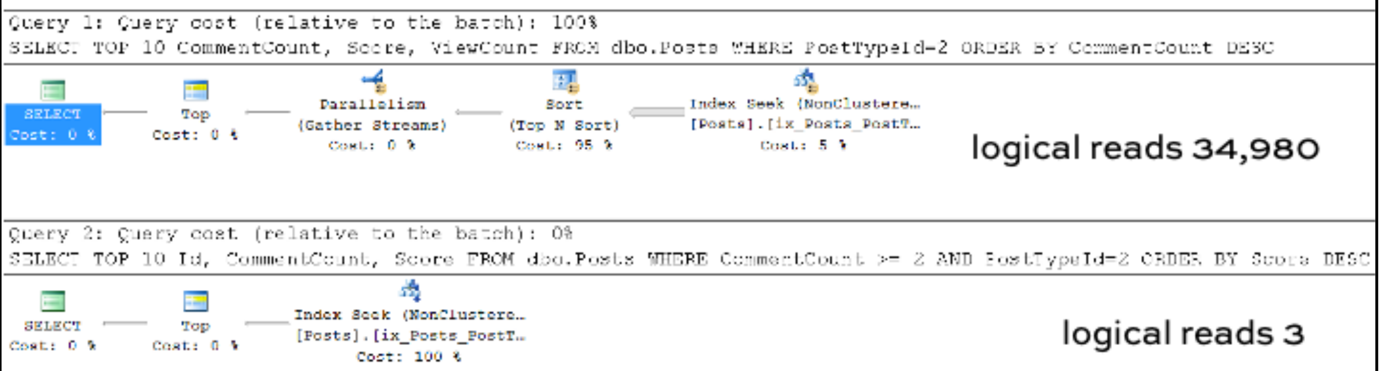


logical reads 10,793

1.5 p27

What if the order was different?

KEY (PostTypeId, Score, CommentCount) INCLUDE (Id, ViewCount)

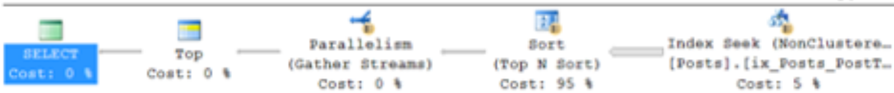


1.5 p28

What if we tried this?

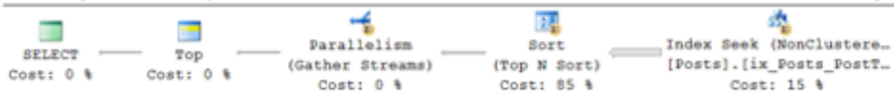
KEY (PostTypeId) INCLUDE (Score, CommentCount, Id, ViewCount)

```
Query 1: Query cost (relative to the batch): 74%
SELECT TOP 10 CommentCount, Score, ViewCount FROM dbo.Posts WHERE PostTypeId=2 ORDER BY CommentCount DESC
```



logical reads 34,754

```
Query 2: Query cost (relative to the batch): 26%
SELECT TOP 10 Id, CommentCount, Score FROM dbo.Posts WHERE CommentCount >= 2 AND PostTypeId=2 ORDER BY Score DESC
Missing Index (Impact 13.3575): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Posts] ([
```



logical reads 34,830

1.5 p29

This requires a decision

Things to research

- Will one of these queries run more often than the other?
- Does one of these queries need to be faster?
- Can either of these queries be cached in the app?
- How many other read and write queries run?

In general, one index that is great for one query
and pretty good for the other is usually best

- KEY (PostTypeId, CommentCount, Score)
  INCLUDE (Id, ViewCount)

1.5 p30

Field order guidelines (not rules)

Fields you use the most often should go first

When doing range scans, selectivity matters

Compromise involves:

- Prioritizing reads vs writes
- Prioritizing which queries need to be fastest
- Caching data in the application
- Spending more money on hardware

The better you know your workload,
the better your decisions become.

1.5 p31

So how do we get the workload?

Ask the users

Ask your gut

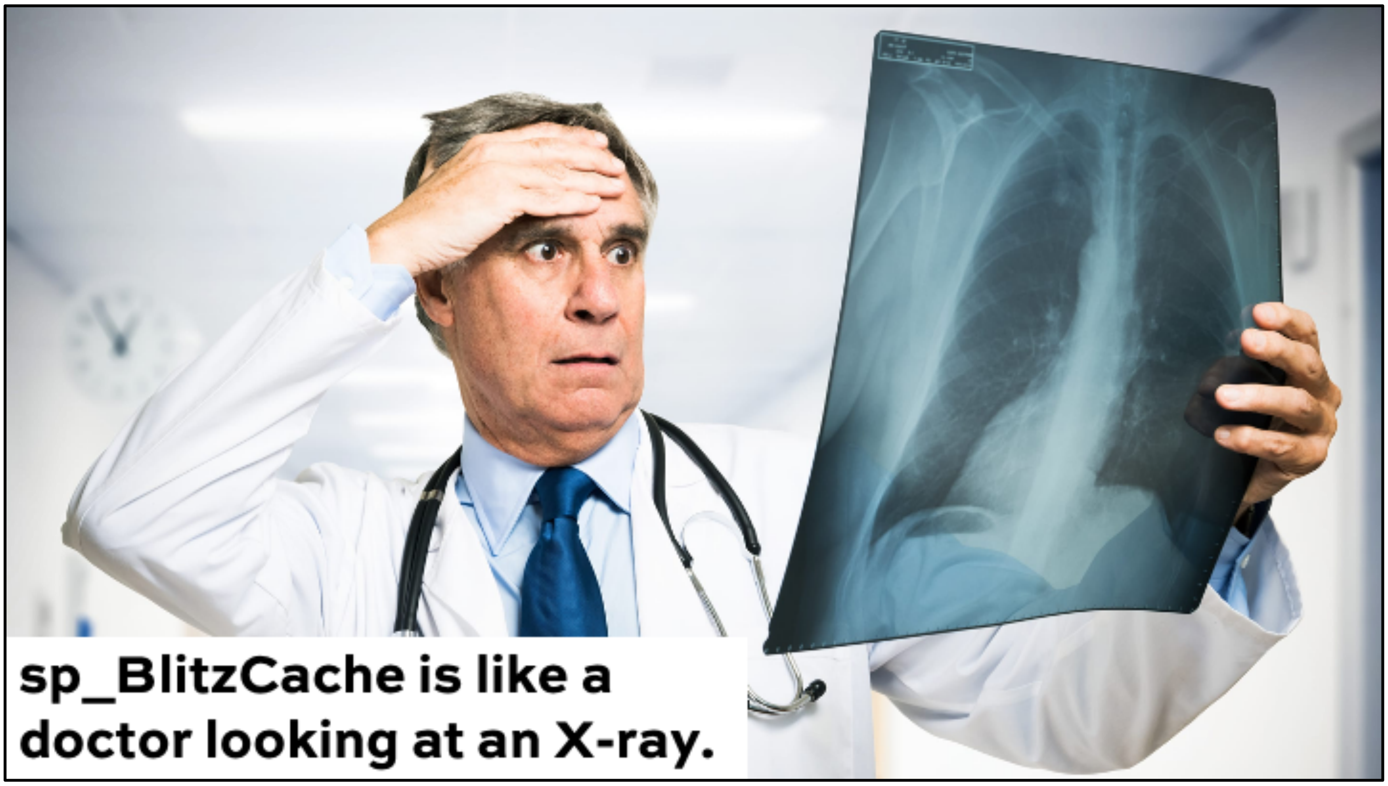
Capture a Profiler or Extended Events trace

Enable Query Store and build reporting queries

Use a monitoring tool
(Idera SQL DM, Quest Spotlight, SentryOne, etc)

My favorite: read the plan cache with sp_BlitzCache

1.5 p32


sp_BlitzCache is like a doctor looking at an X-ray.

X-rays have some hard truths.

For example, there's a strange object inside you.

They just can't tell how the object got there,
or how the doctor's going to remove it.

sp_BlitzCache's hard truths

Query level metrics by total and average:

Reads, writes

Executions, executions per minute

CPU

Duration

Newer SQL versions add memory grants, spills

1.5 p35

Chloroform: stuff that disappears

Per-execution statistics like:

- Variances between estimated & actual rows
- Spills per execution
- Parameters used to execute (not compile) a query

Queries that hide:

- OPTION (RECOMPILE)
- Dynamic SQL
- Unparameterized SQL
- Servers with memory pressure (or resource_semaphore)

We cover those in Mastering Query Tuning, Server Tuning.

1.5 p36

It shows the top 10 queries.

It's up to you to pick the sort order.

By default: @SortOrder = 'cpu'

But when you're doing index tuning,
use @SortOrder = 'reads' instead.

1.5 p37

Other params: pick a database

```
EXEC dbo.sp_BlitzCache @DatabaseName = 'StackOverflow'
```

Results | Messages

Database	Cost	Query Text	Query Type
StackOverflow	10335.5	CREATE PROCEDURE dbo.ExpensiveSort (@I...	Procedure or Function: ExpensiveSort
StackOverflow	10335.5	SELECT TOP 2147483647 u.DisplayName INT...	Statement (parent [dbo].[ExpensiveSort])
StackOverflow	16.9095	CREATE PROCEDURE dbo.ExpensiveKeyLookup ...	Procedure or Function: ExpensiveKeyLookup
StackOverflow	16.9095	SELECT * FROM dbo.Users AS u WHER...	Statement (parent [dbo].[ExpensiveKeyLookup])
StackOverflow	0.19619	SELECT db_id() AS database_id, c.system_typ...	Statement
StackOverflow	0.0792935	SELECT db_id() AS database_id, o.[type] as ModuleT...	Statement
StackOverflow	0.135987	SELECT db_id() AS database_id, o.[type] AS o...	Statement
StackOverflow	0.0860988	SELECT clmns.name AS [Name], clmns.column_id AS...	Statement
StackOverflow	0.167497	WITH TablesAndViews AS (SELECT object_id, ta...	Statement
StackOverflow	0.175316	SELECT clmns.name AS [Name], clmns.column_id AS...	Statement

1.5 p38

Stored proc name

Note: don't prefix schema here

```
EXEC dbo.sp_BlitzCache @StoredProcName = 'GetUserCrap'
```

Results | Messages

Database	Cost	Query Text	Query Type
StackOverflow	0.00669186	CREATE PROCEDURE [dbo].[GetUserCrap] @UserId INT...	Procedure or Function: GetUserCrap
StackOverflow	0.00335548	SELECT p.OwnerUserId, p.LastActivityDate FROM dbo.P...	Statement (parent [dbo].[GetUserCrap])
StackOverflow	0.00333638	SELECT p.OwnerUserId, p.LastActivityDate FROM dbo.P...	Statement (parent [dbo].[GetUserCrap])

1.5 p39

sp_BlitzCache finds

Missing indexes, but also:

Implicit conversion

Forced serialization

Table variables

Expensive sorts

Expensive key lookups

Columnstore indexes not in batch mode

1.5 p40

Recap

1.5 p41

The T part of D.E.A.T.H. is:

1. Find your top resource-consuming queries with sp_BlitzCache @SortOrder = 'reads'
2. Acknowledge Clippy's index recommendations, but build your own
3. Generally, equality fields go first, then inequality fields
4. ORDER BY, joins can come into play too
5. Try to build as few indexes as practical to satisfy many queries: that's where knowing your workload helps

1.5 p42