When you have a lot to do, you have to make a choice.

Do it all at once in a transaction

- Good: easy to code, gets done quickly
- Bad: lock escalation means no one else can work

Or separate it into small batches

- Good: lets other people work alongside you
- Bad: harder to code, takes longer to run

3.3 p5

Sooner or later, you're gonna have to do the latter.

Or separate it into small batches

- Good: lets other people work alongside you
- Bad: harder to code, takes longer to run

3.3 p6

Signs you need batching

A process takes hours to run

Other people need to keep querying while it runs

It can't be offloaded to another server,
and you can't temporarily upsize the server

It doesn't need transactional consistency

You're willing to spend some time coding T-SQL

3.3 p7

Our Stack Overflow scenario

Let's say we need to delete:

- All of the Users who live in London (where Location = 'London, United Kingdom')
- All of their rows in related tables: Badges, Comments, Posts, Votes
- All of those tables have a UserId or OwnerUserId column, and we can join on that

3.3 p8

The easy way: one big transaction

```
BEGIN TRAN;
DELETE d
    FROM dbo.Badges d
    INNER JOIN dbo.Users u ON d.UserId = u.Id
    WHERE u.Location = 'London, United Kingdom';

DELETE d
    FROM dbo.Comments d
    INNER JOIN dbo.Users u ON d.UserId = u.Id
    WHERE u.Location = 'London, United Kingdom';

DELETE d
    FROM dbo.Posts d
    INNER JOIN dbo.Users u ON d.OwnerUserId = u.Id
    WHERE u.Location = 'London, United Kingdom';

DELETE d
    FROM dbo.Votes d
    INNER JOIN dbo.Users u ON d.UserId = u.Id
    WHERE u.Location = 'London, United Kingdom';

/* No need for error checking here since it's in one tran: */
DELETE
    FROM dbo.Users
    WHERE Location = 'London, United Kingdom';

/* Commenting this out only so we can see the effects: */
--COMMIT;
```

I have indexes to help

```
CREATE INDEX Location ON dbo.Users(Location);
GO
CREATE INDEX UserId ON dbo.Badges(UserId);
GO
CREATE INDEX UserId ON dbo.Comments(UserId);
GO
CREATE INDEX OwnerUserId ON dbo.Posts(OwnerUserId);
GO
CREATE INDEX UserId ON dbo.Votes(UserId);
GO
```

3.3 p11

But one big transaction sucks.

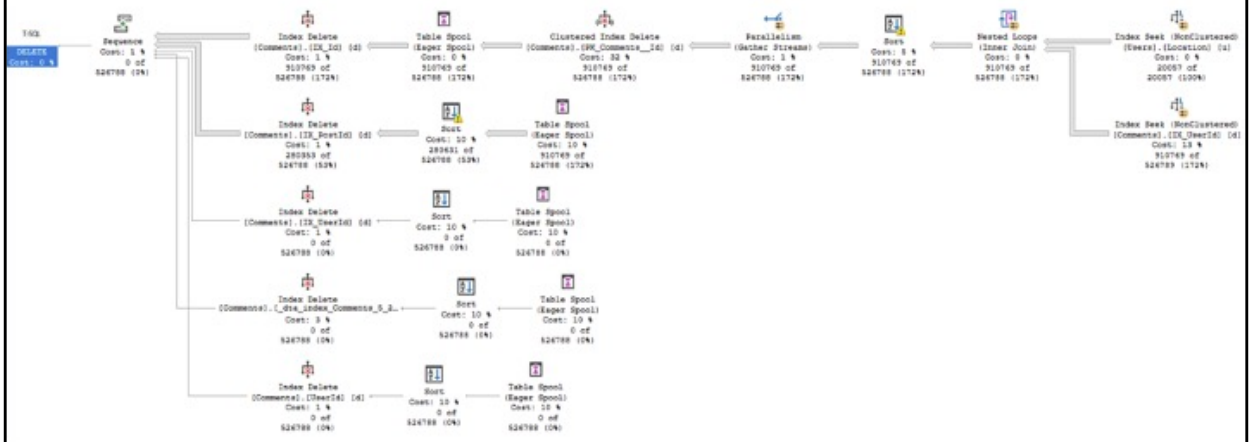
It takes a while to run.

While it's running:

- We're holding exclusive table-level locks on all the tables, not just one at a time
- The transaction log and the version store can grow out of control
- SQL Server deletes each index serially

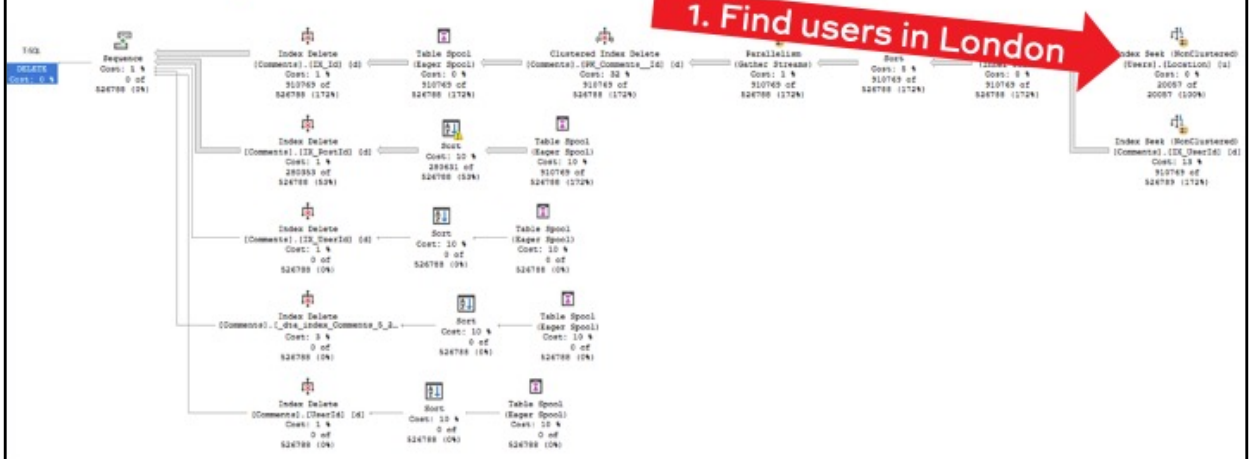
3.3 p12

As the Comments delete runs...



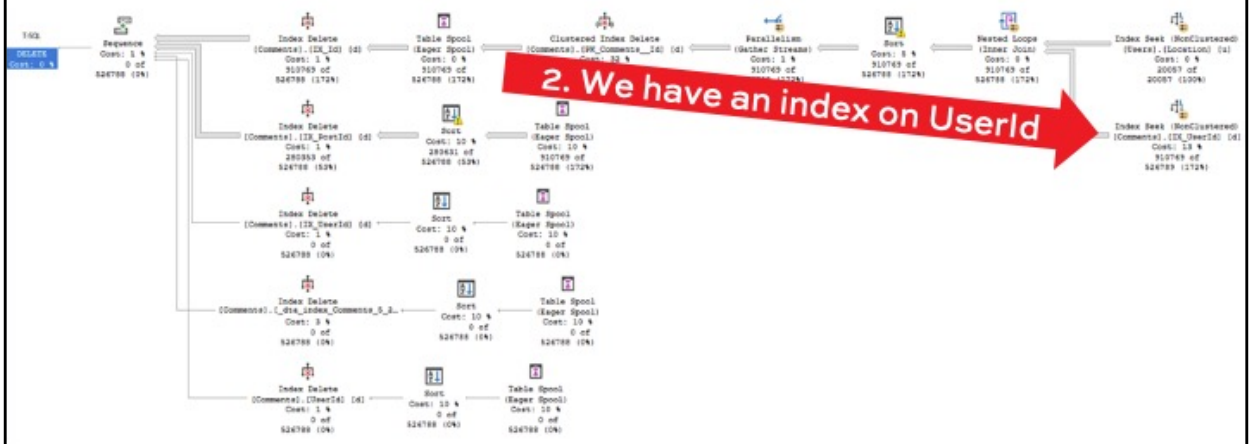
3.3 p13

Step 1: Find the Users in London



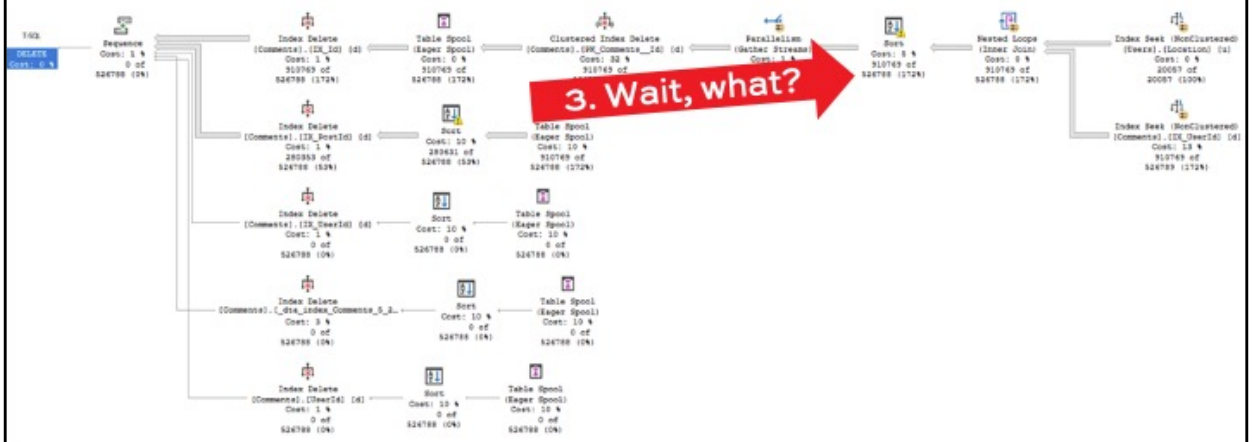
3.3 p14

Step 2: Find their Comments



3.3 p15

Step 3: Sort them by Comment Id



3.3 p16

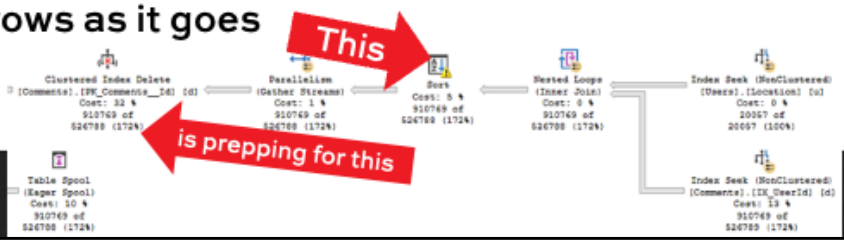
The delete is going to be hard, so

It's going to involve deleting a "lot" of Comments

The clustered index of Comments is Id (not UserId)

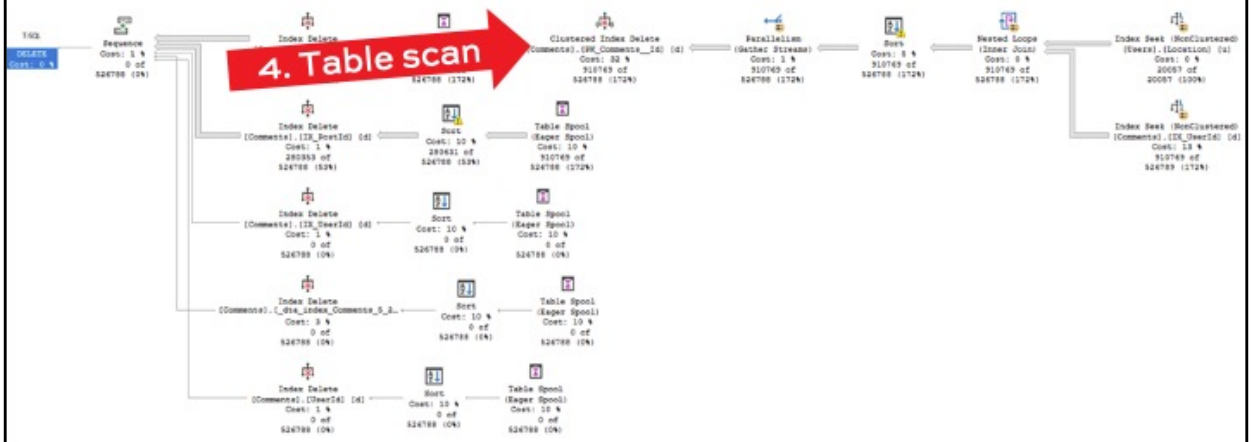
To efficiently find them, SQL Server wants to sort all of the comments from Londoners, by Comment Id

That way it can scan the clustered index, deleting rows as it goes



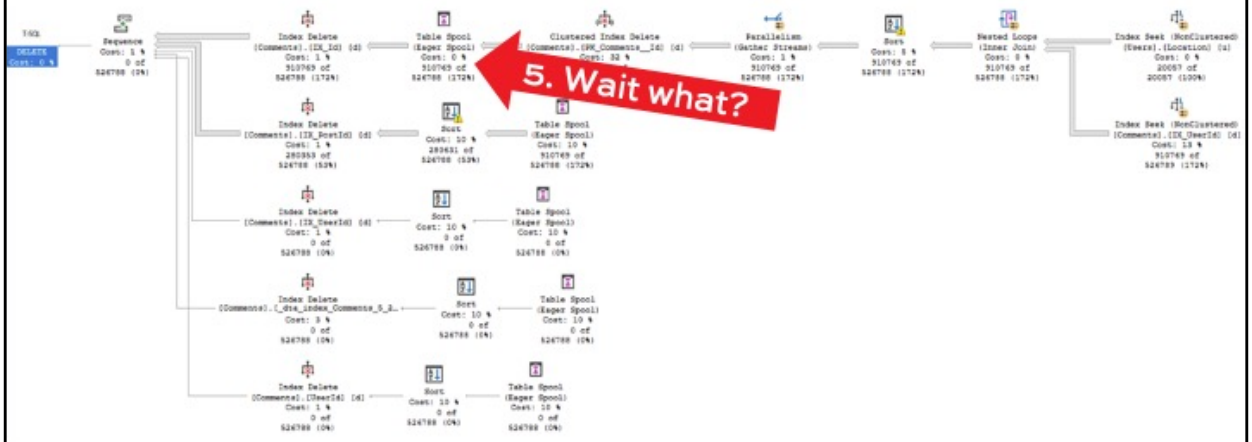
3.3 p17

Step 4: Scan the CX, delete rows



3.3 p18

Step 5: Sort all London rows again



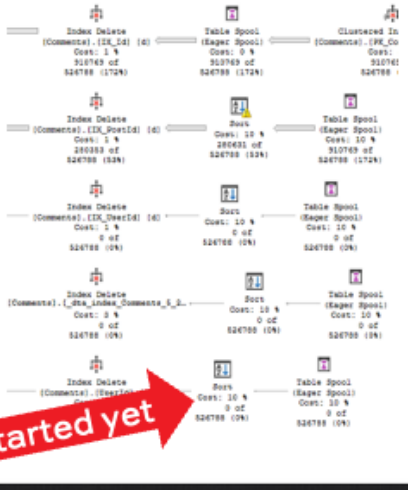
3.3 p19

Deleting each index is hard too

It's going to involve deleting a "lot" of rows

So each index that we need to delete, SQL Server sorts all of the to-be-deleted rows by the index's keys

This happens one index at a time, in order

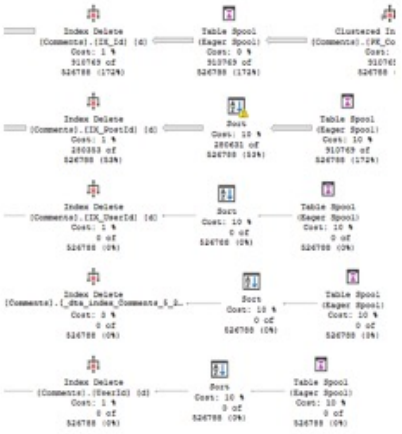


3.3 p20

All this work is single-threaded

Deletes, updates, and inserts are not normally parallelized:

https://www.brentozar.com/archive/2020/09/update-insert-and-delete-are-not-normally-processed-in-parallel/



3.3 p21

And we exclusively locked it all

As this query runs, it progressively grabs locks on more and more tables.

My favorite way to see it:

```
1  sp_WhoIsActive @get_locks = 1
```

150 %

Results | Messages

	dd hh:mm:ss.mss	session_id	sql_text	locks
1	00 00:08:19.653	53	<?query -- BEGIN TRAN; DELETE d FROM dbo.Bad...	<Database name="StackOverflow"><Locks><Lock requ...

3.3 p22


```
<Database name="StackOverflow">
  <Locks>
    <Lock request_mode="S" request_status="GRANT" request_count="1" />
  </Locks>
  <Objects>
    <Object name="(null)">
      <Locks>
        <Lock resource_type="EXTENT" page_type="*" request_mode="X" request_status="GRANT" request_count="803" />
      </Locks>
    </Object>
    <Object name="Badges" schema_name="dbo">
      <Locks>
        <Lock resource_type="OBJECT" request_mode="X" request_status="GRANT" request_count="1" />
      </Locks>
    </Object>
    <Object name="Comments" schema_name="dbo">
      <Locks>
        <Lock resource_type="OBJECT" request_mode="X" request_status="GRANT" request_count="1" />
      </Locks>
    </Object>
    <Object name="Posts" schema_name="dbo">
      <Locks>
        <Lock resource_type="OBJECT" request_mode="X" request_status="GRANT" request_count="1" />
        <Lock resource_type="PAGE" page_type="*" index_name="PK_Posts__Id" request_mode="X" request_status="GRANT"
      </Locks>
    </Object>
    <Object name="Users" schema_name="dbo">
      <Locks>
        <Lock resource_type="OBJECT" request_mode="X" request_status="GRANT" request_count="1" />
      </Locks>
    </Object>
    <Object name="Votes" schema_name="dbo">
      <Locks>
        <Lock resource_type="OBJECT" request_mode="X" request_status="GRANT" request_count="1" />
```

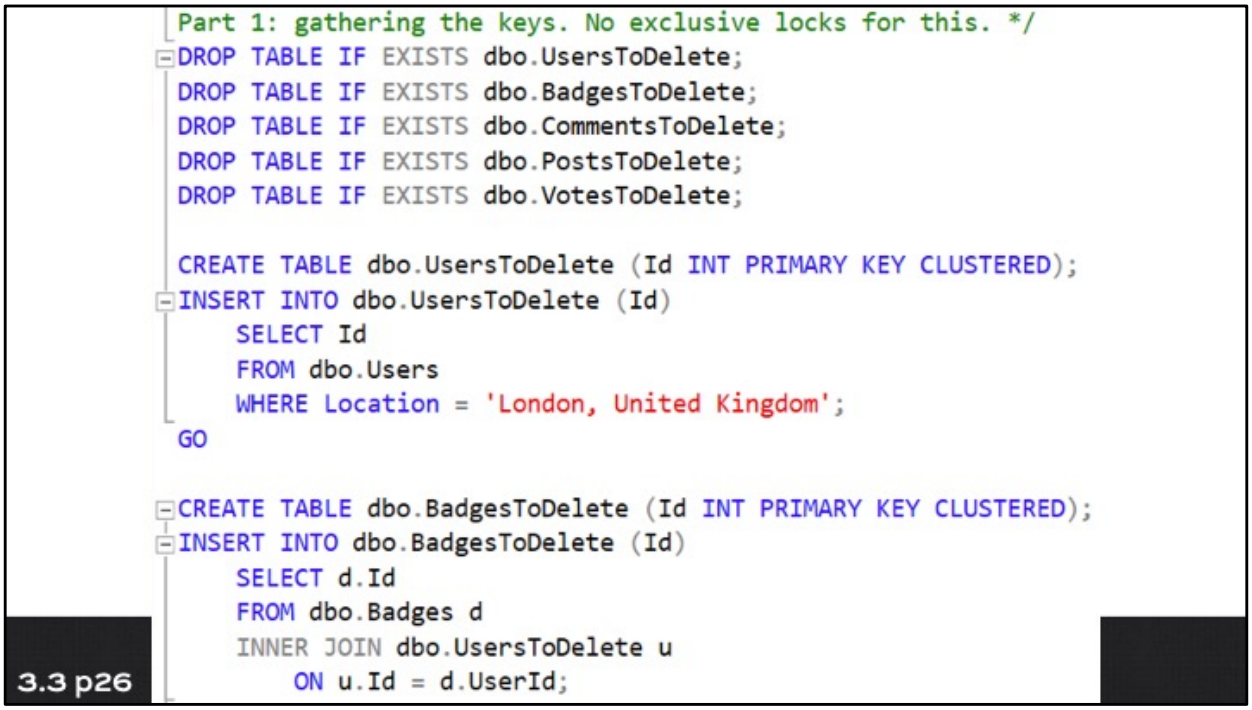
Pseudocode

Create a table with a list of UserIds to delete

For each child table:

- Create a temp table with a list of its Ids to delete
- While rows exist in that list, delete 1,000 of them (staying well below lock escalation thresholds)

3.3 p25



This is not a transaction.

This could be problematic if users are changing their locations while we work:

```
CREATE TABLE dbo.UsersToDelete (Id INT PRIMARY KEY CLUSTERED);
INSERT INTO dbo.UsersToDelete (Id)
    SELECT Id
    FROM dbo.Users
    WHERE Location = 'London, United Kingdom';
```

If users move into London, you could keep this table up to date with a trigger – but let's keep this simple.

Same thing with the child tables.

If London users are continuing to add Badges, they won't show up in this table.

```
CREATE TABLE dbo.BadgesToDelete (Id INT PRIMARY KEY CLUSTERED);
INSERT INTO dbo.BadgesToDelete (Id)
    SELECT d.Id
    FROM dbo.Badges d
    INNER JOIN dbo.UsersToDelete u
        ON u.Id = d.UserId;
```

Easy fix: just repeat the deletion process again at the end to catch any newly added Badges.

3.3 p28

Don’t use a temp table for this.

A real table will:

- Be query-able from other sessions to check progress on the deletes while they run
- Persist after reboots & failovers, helpful for really long-running jobs
- Allow you to parallelize it across many sessions

```
CREATE TABLE dbo.BadgesToDelete (Id INT PRIMARY KEY CLUSTERED);
INSERT INTO dbo.BadgesToDelete (Id)
    SELECT d.Id
    FROM dbo.Badges d
    INNER JOIN dbo.UsersToDelete u
        ON u.Id = d.UserId;
```

3.3 p29

Back to the big picture

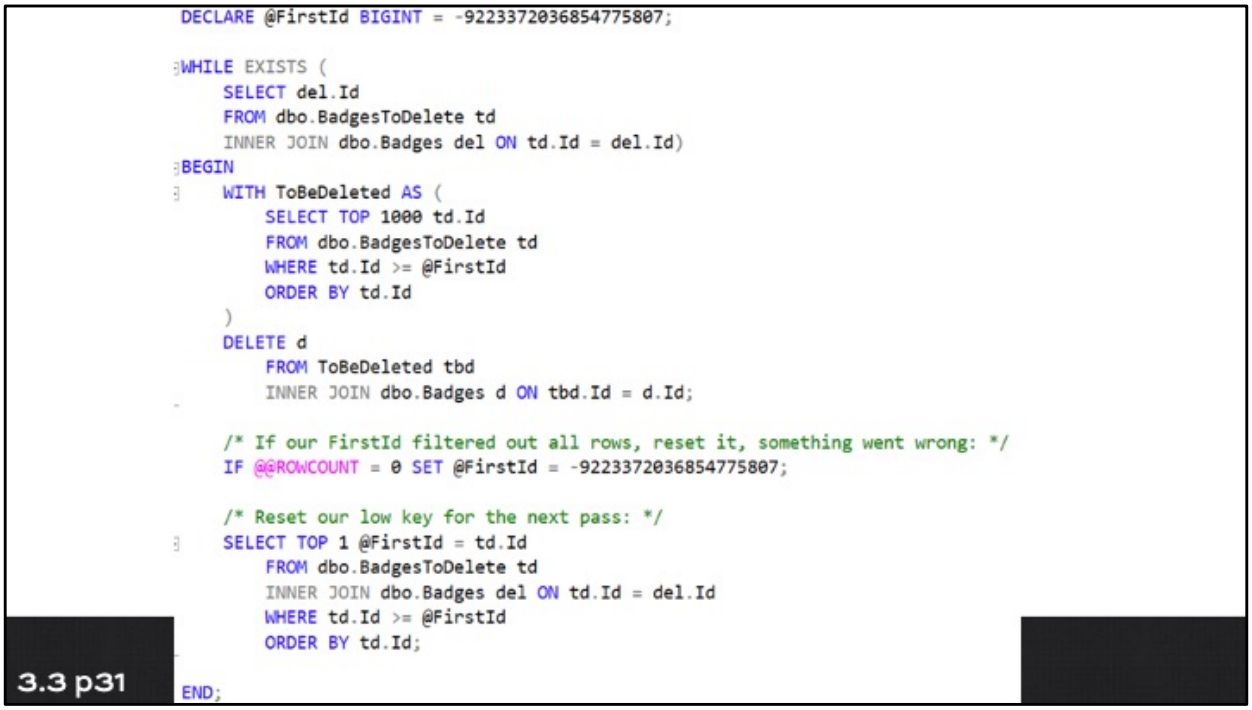
Done Create a table with a list of UserIds to delete

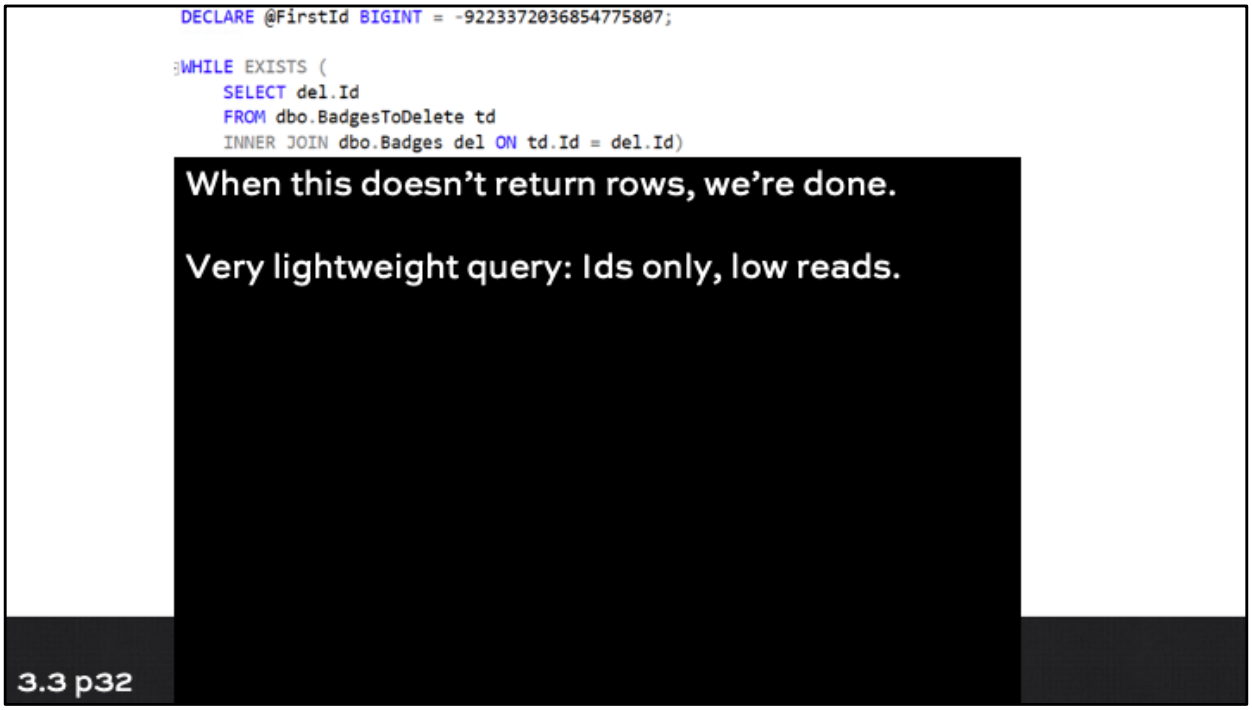
For each child table:

Done Create a temp table with a list of its Ids to delete

- Next While rows exist in that list, delete 1,000 of them (staying well below lock escalation thresholds)

3.3 p30





Now the locking is about to start, so we have to be careful about getting few rows, and touching as few indexes as possible.

```
WITH ToBeDeleted AS (
    SELECT TOP 1000 td.Id
    FROM dbo.BadgesToDelete td
    WHERE td.Id >= @FirstId
    ORDER BY td.Id
)
```

The TOP 1000 with an ORDER BY means SQL Server knows exactly how many rows will come out, so it helps avoid lock escalation.

We don't have to join it to the Badges table to figure out which rows are left to delete.

3.3 p33

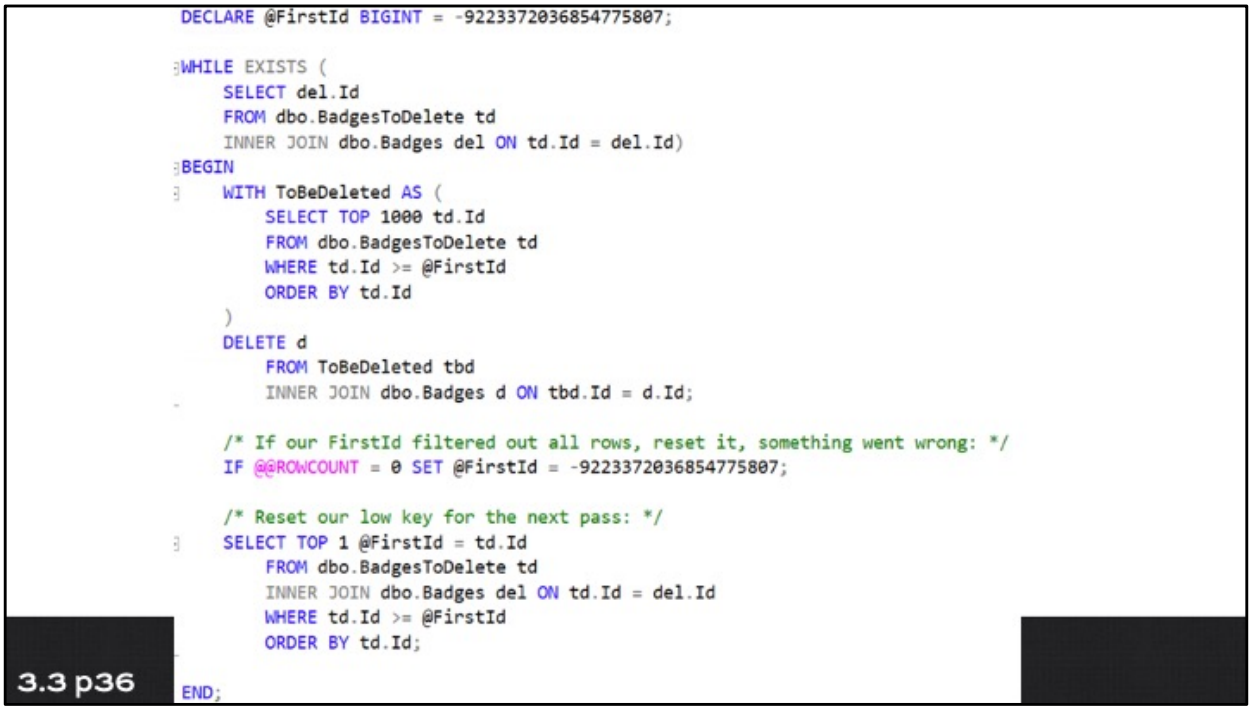
We take the 1,000 rows from the CTE:

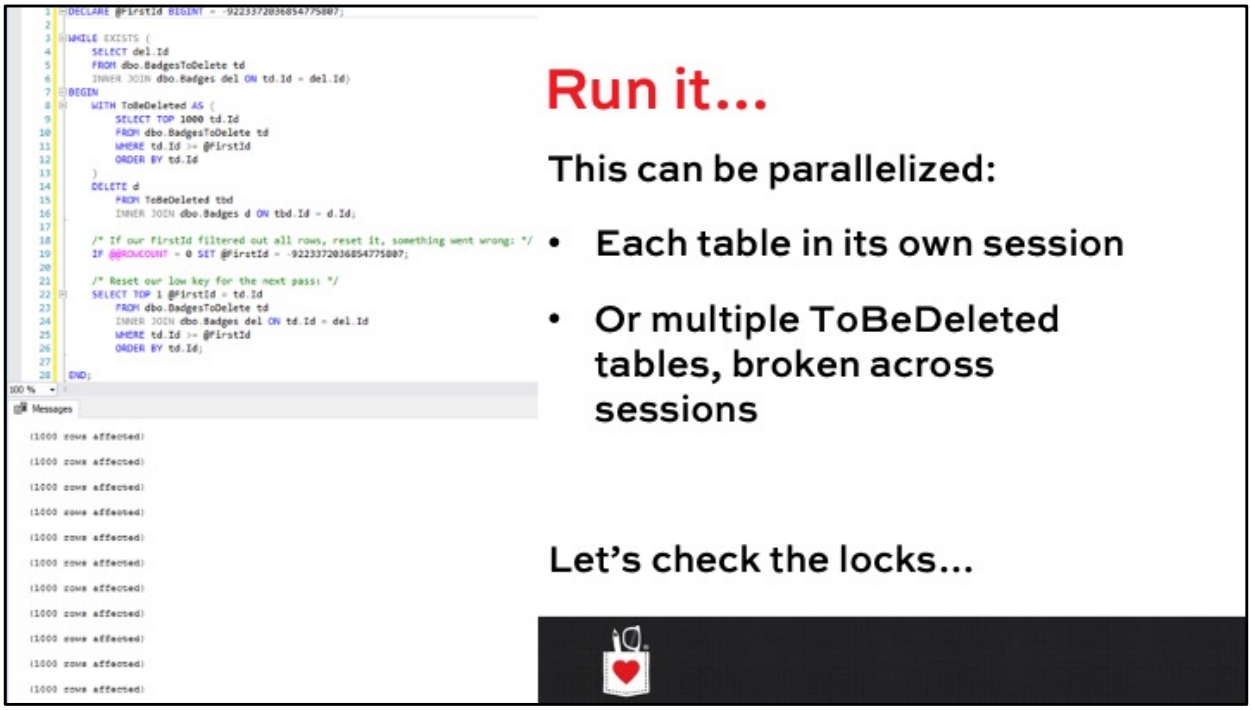
```
WITH ToBeDeleted AS (
    SELECT TOP 1000 td.Id
    FROM dbo.BadgesToDelete td
    WHERE td.Id >= @FirstId
    ORDER BY td.Id
)
DELETE d
    FROM ToBeDeleted tbd
    INNER JOIN dbo.Badges d ON tbd.Id = d.Id;
```

And join directly to Badges using the Id.

SQL Server knows exactly how many rows will be involved, so this avoids lock escalation.

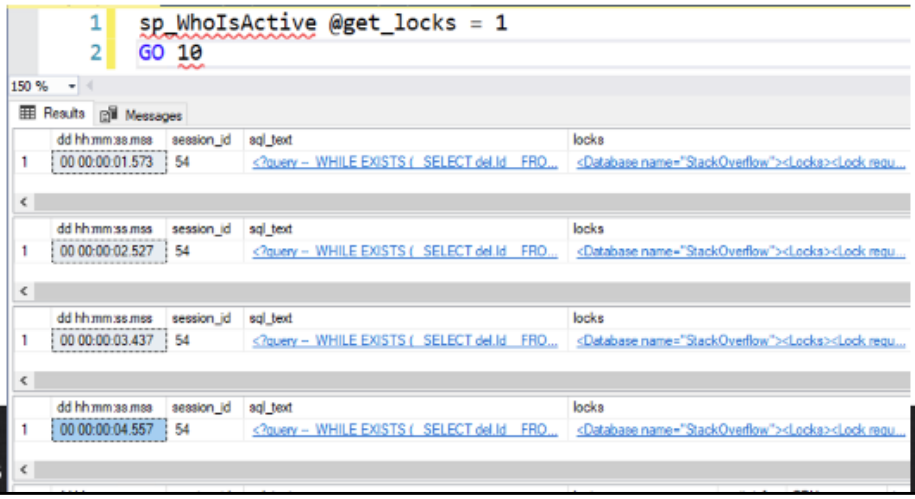
3.3 p34





Run sp_WhoIsActive repeatedly

The locks will be changing every few milliseconds:



3.3 p38

If we did our job right...

There won't be any eXclusive table-level locks, only row-level or page-level locks:

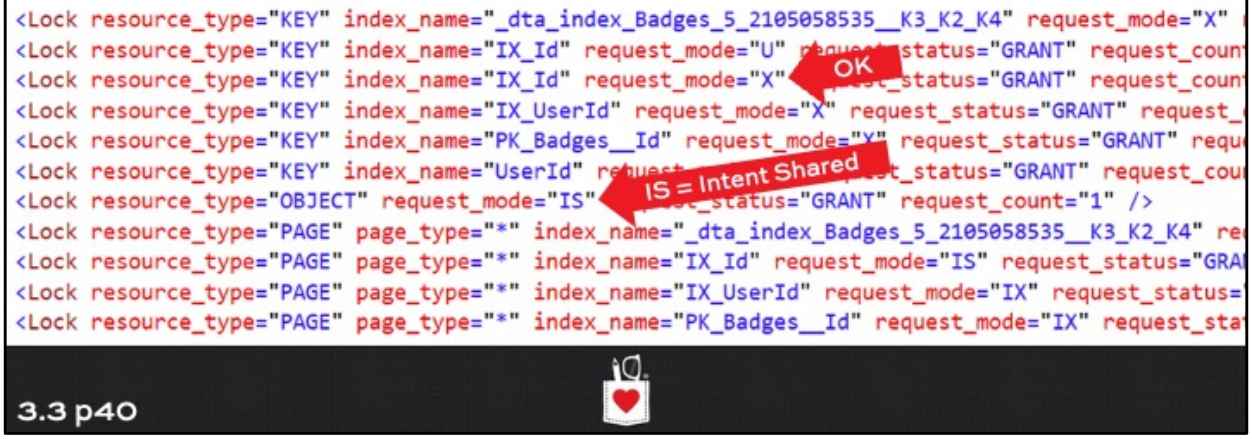
```
<Database name="StackOverflow">
  <Locks>
    <Lock request_mode="S" request_status="GRANT" request_count="1" />
  </Locks>
  <Objects>
    <Object name="Badges" schema_name="dbo">
      <Locks>
        <Lock resource_type="KEY" index_name="_dta_index_Badges_5_2105058535__K3_K2_K4" request_mode="X" request_status="GRANT" request_count="52" />
        <Lock resource_type="KEY" index_name="IX_Id" request_mode="U" request_status="GRANT" request_count="69" />
        <Lock resource_type="KEY" index_name="IX_Id" request_mode="X" request_status="GRANT" request_count="109" />
        <Lock resource_type="KEY" index_name="IX_UserId" request_mode="X" request_status="GRANT" request_count="72" />
        <Lock resource_type="KEY" index_name="PK_Badges__Id" request_mode="X" request_status="GRANT" request_count="179" />
        <Lock resource_type="KEY" index_name="UserId" request_mode="X" request_status="GRANT" request_count="19" />
        <Lock resource_type="OBJECT" request_mode="IS" request_status="GRANT" request_count="1" />
        <Lock resource_type="PAGE" page_type="*" index_name="_dta_index_Badges_5_2105058535__K3_K2_K4" request_mode="IX" request_status="GRANT" request_cou
        <Lock resource_type="PAGE" page_type="*" index_name="IX_Id" request_mode="IS" request_status="GRANT" request_count="1" />
        <Lock resource_type="PAGE" page_type="*" index_name="IX_UserId" request_mode="IX" request_status="GRANT" request_count="7" />
        <Lock resource_type="PAGE" page_type="*" index_name="PK_Badges__Id" request_mode="IX" request_status="GRANT" request_count="7" />
      </Locks>
    </Object>
    <Object name="BadgesToDelete" schema_name="dbo">
      <Locks>
        <Lock resource_type="OBJECT" request_mode="IS" request_status="GRANT" request_count="1" />
        <Lock resource_type="PAGE" page_type="*" index_name="PK__BadgesTo__3214EC07B41C00C1" request_mode="IS" request_status="GRANT" request_count="1" />
      </Locks>
    </Object>
  </Objects>
</Database>
```

3.3 p39

ENHANCE

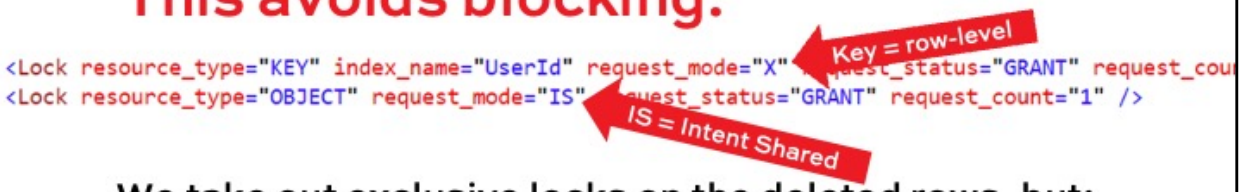
X = exclusive locks

X = OK on keys, but not okay on entire objects.



3.3 p40

This avoids blocking.



We take out exclusive locks on the deleted rows, but:

- Only those 1,000 rows (not hitting lock escalation to lock the entire table)
- We're releasing those 1,000 locks as soon as our delete finishes

3.3 p41

This is easy to get wrong.

If you try to do all this in a transaction,
you'll escalate to table-level locking.

If you do this with row-level lock hints,
your SQL Server can run out of memory.

If you don't have the supporting indexes,
you'll hit a table scan and lock escalation.

Start with my code, modify it as needed,
and test it in a dev server with prod scale data.

3.3 p42

How to test it

Run the batches just like you plan to run in prod.

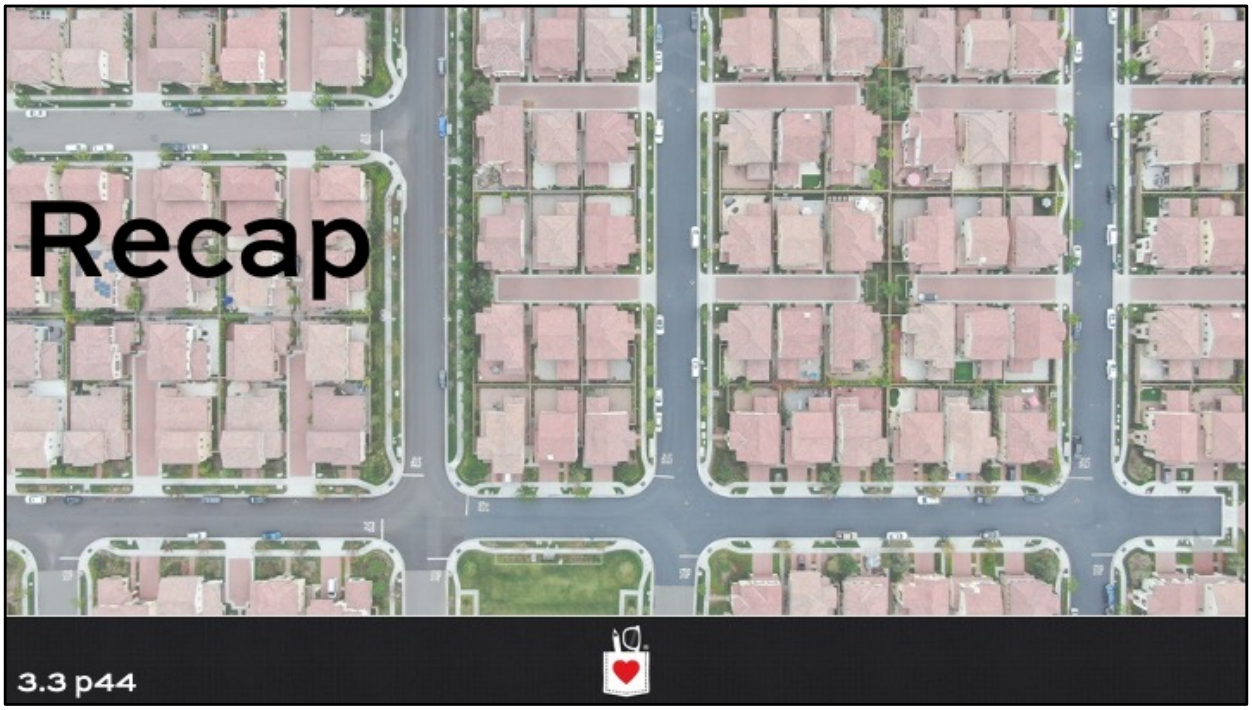
You don't need the same server horsepower:
your code should behave the same on any server size.

While it runs, run sp_WhoIsActive @get_locks GO 10.

Examine the Locks column, and if you see this, you're hitting lock escalation – tweak the code/indexes:

```
<Object name="Badges" schema_name="dbo">
  <Locks>
    <Lock resource_type="OBJECT" request_mode="X"
```

X = BAD



You have the choice.
The answer isn't always batching.

Do it all at once in a transaction

- Good: easy to code, gets done quickly
- Bad: lock escalation means no one else can work

Or separate it into small batches

- Good: lets other people work alongside you
- Bad: harder to code, takes longer to run

Batching isn't easier: it's hard. It's one of your tools.

3.3 p45

My scenario was deletes.

But this same thought process holds true for:

- Trickle loading data: track which data needs to be inserted in a table
- Mass updates: track which rows need to be updated, and what changes need to be made

3.3 p46

When you do need to do batching:

Avoid the ~5,000 row lock escalation threshold:
https://www.littlekendra.com/2017/04/03/which-locks-count-toward-lock-escalation/

Design your batching code to run on the clustered indexes of the tables involved.

Use the fast-ordered-delete technique of CTEs or views so SQL Server understands how many rows will be involved:
https://www.brentozar.com/archive/2018/04/how-to-delete-just-some-rows-from-a-really-big-table/

3.3 p47

Code is yours, and there's no lab.

The full demo scripts are in this video's page.

There's no test on this in the labs because:

- This is about processes that take a long time
- You're probably not going to need to jump into this technique next week
- When you *do* need it, you'll need to customize it based on your own database schema anyway

3.3 p48