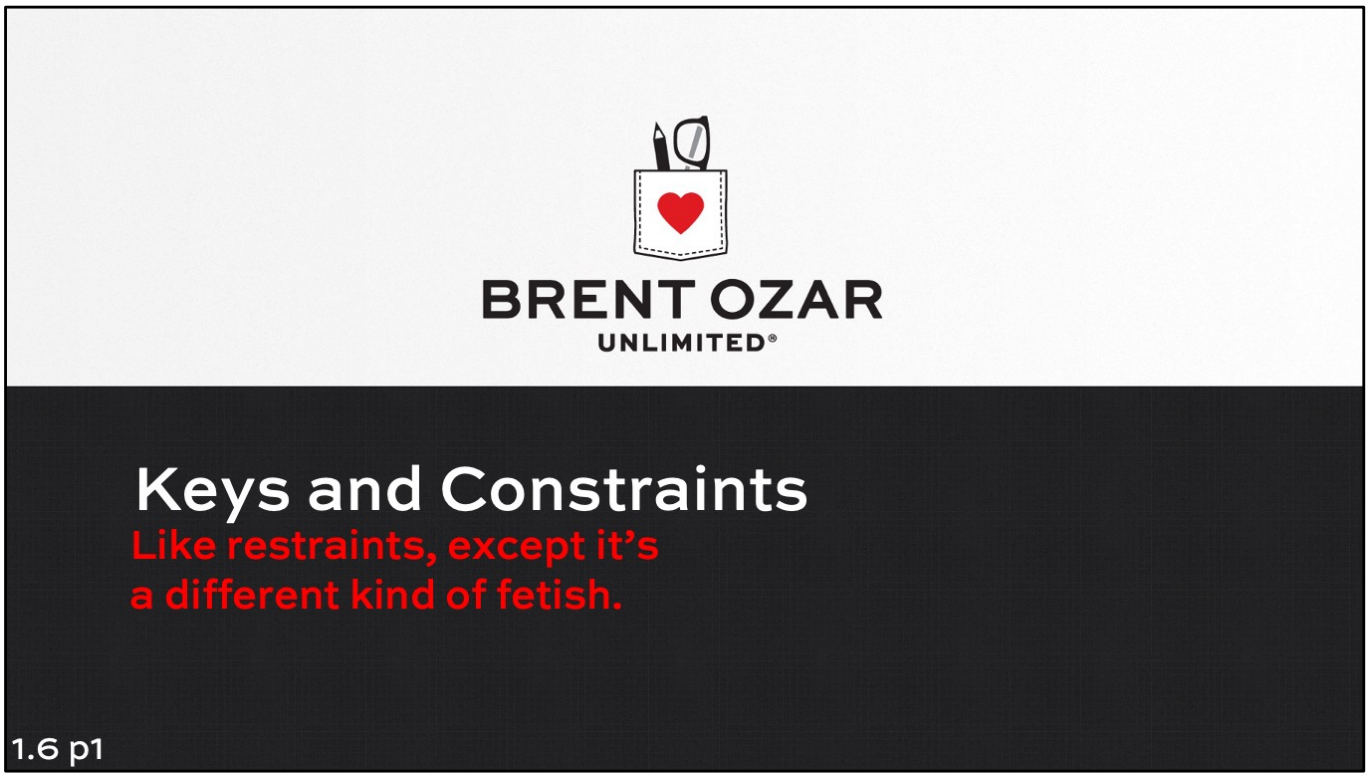


“Should I index my foreign keys?”

This question has two different answers:

- When you’re starting a new database from scratch, you should have foreign keys, and index them
- When you have an existing database, you might need to let go of this if you have blocking and you need to hit the 5 & 5.

In a perfect world

The more SQL Server knows about your data:

- The better execution plans you'll get
- The less bad data you'll get

So you can set up constraints on tables:

- Primary keys: data modeling tool to define uniqueness (may or may not be the clustered index)
- Foreign keys: modeling tool to define relationships between tables
- Check constraints: physical storage tool to bar bad data
- Unique constraints: physical storage tool to enforce uniqueness

- dbo.Posts
  - Columns
    - Id (PK, int, not null)
    - AcceptedAnswerId (int, null)
    - AnswerCount (int, null)
    - Body (nvarchar(max), not null)
    - ClosedDate (datetime, null)
    - CommentCount (int, null)
    - CommunityOwnedDate (datetime, null)
    - CreationDate (datetime, not null)
    - FavoriteCount (int, null)
    - LastActivityDate (datetime, not null)
    - LastEditDate (datetime, null)
    - LastEditorDisplayName (nvarchar(40), null)
    - LastEditorUserId (int, null)
    - OwnerUserId (int, null)
    - ParentId (int, null)
    - PostTypeId (int, not null)
    - Score (int, not null)
    - Tags (nvarchar(150), null)
    - Title (nvarchar(250), null)
    - ViewCount (int, not null)
  - Keys
    - PK_Posts__Id
  - Constraints
  - Triggers
  - Indexes
  - Statistics

Take the Posts table.

PostTypeId field: let's make sure all rows in Posts have a matching row in the PostTypes table.

```
SELECT * FROM dbo.PostTypes
```

150 %

Results | Messages

	Id	Type
1	1	Question
2	2	Answer
3	3	Wiki
4	4	TagWikiExerpt
5	5	TagWiki
6	6	ModeratorNomination
7	7	WikiPlaceholder
8	8	PrivilegeWiki

No foreign keys ship by default in StackOverflow.

1.6 p5

What are the most popular PostTypes?

Say we run this query to find out:

```
SELECT p.PostTypeId, COUNT(*) AS Posts
  FROM dbo.PostTypes pt
  INNER JOIN dbo.Posts p ON pt.Id = p.PostTypeId
  GROUP BY p.PostTypeId
  ORDER BY COUNT(*) DESC;
GO
```

150 %

Results | Messages | Execution plan

	PostTypeId	Posts
1	2	24676333
2	1	15930617
3	4	46606
4	5	46606
5	6	312
6	3	167
7	7	4
8	8	2

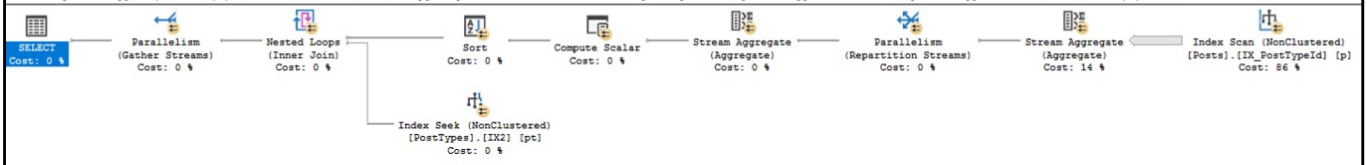
The plan hits both Posts and PostTypes.

```
SELECT p.PostTypeId, COUNT(*) AS Posts
  FROM dbo.PostTypes pt
  INNER JOIN dbo.Posts p ON pt.Id = p.PostTypeId
  GROUP BY p.PostTypeId
  ORDER BY COUNT(*) DESC;
GO
```

50 %

Results | Messages | Execution plan

Query 1: Query cost (relative to the batch): 100%
SELECT p.PostTypeId, COUNT(*) AS Posts FROM dbo.PostTypes pt INNER JOIN dbo.Posts p ON pt.Id = p.PostTypeId GROUP BY p.PostTypeId ORDER BY COUNT(*) DESC



1.6 p7

But what if we tell SQL they're related?

There are drawbacks to doing this – but let's cover the positives first:

```
ALTER TABLE dbo.Posts
ADD CONSTRAINT fk_Posts_PostTypeId
    FOREIGN KEY (PostTypeId)
REFERENCES dbo.PostTypes(Id);
GO
```

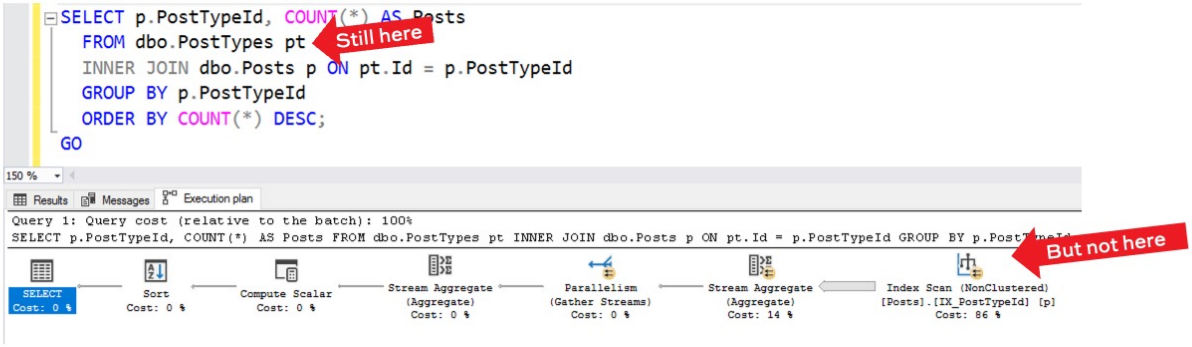
150 %

Messages

```
Commands completed successfully.
```


Now run the same query again...

And SQL Server no longer needs to touch the PostTypes table!



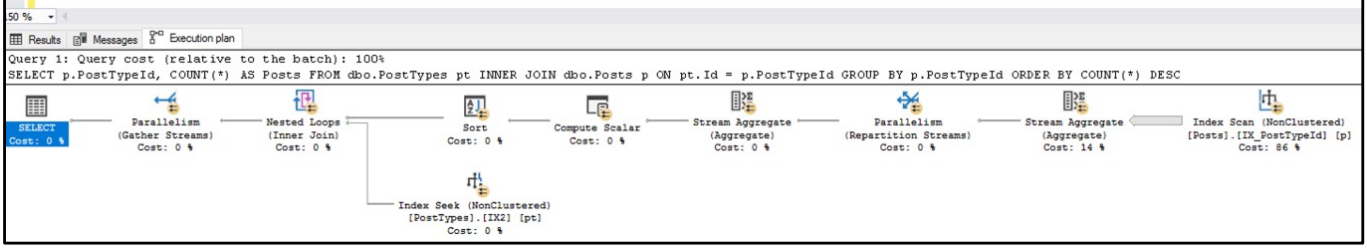
This is foreign key join elimination. Amazing, right?

Not really amazing here.

In the original plan, the PostTypes table was the least of our problems.

That's how it usually is with foreign keys: join elimination helps, but...

```
SELECT p.PostTypeId, COUNT(*) AS Posts
  FROM dbo.PostTypes pt
  INNER JOIN dbo.Posts p ON pt.Id = p.PostTypeId
  GROUP BY p.PostTypeId
  ORDER BY COUNT(*) DESC;
GO
```



Plus, that query is ridiculous.

Nobody writes queries like this:

```
SELECT p.PostTypeId, COUNT(*) AS Posts
  FROM dbo.PostTypes pt
  INNER JOIN dbo.Posts p ON pt.Id = p.PostTypeId
  GROUP BY p.PostTypeId
  ORDER BY COUNT(*) DESC;
GO
```

They want stuff from the FK table:

```
SELECT pt.Id, pt.Type, COUNT(*) AS Posts
  FROM dbo.PostTypes pt
  INNER JOIN dbo.Posts p ON pt.Id = p.PostTypeId
  GROUP BY pt.Id, pt.Type
  ORDER BY COUNT(*) DESC;
GO
```

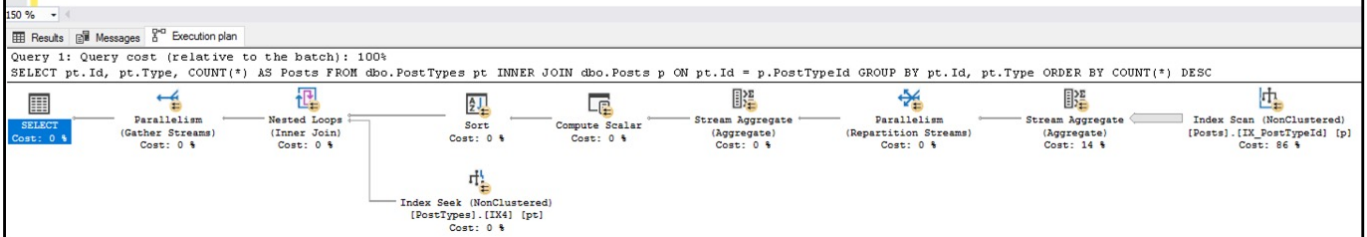
150 %

Results | Messages | Execution plan

	Id	Type	Posts
1	2	Answer	24676333
2	1	Question	15930617
3	4	TagWikiExerpt	46606
4	5	TagWiki	46606
5	6	ModeratorNomination	312
6	3	Wiki	167
7	7	WikiPlaceholder	4
8	8	PrivilegeWiki	2

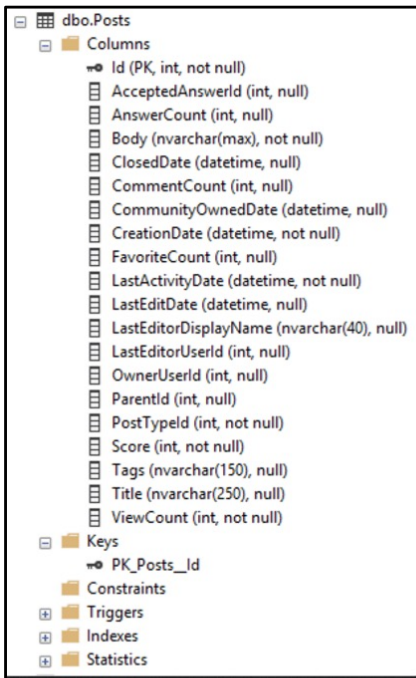
Then we're right back where we started.

```
SELECT pt.Id, pt.Type, COUNT(*) AS Posts
  FROM dbo.PostTypes pt
  INNER JOIN dbo.Posts p ON pt.Id = p.PostTypeId
  GROUP BY pt.Id, pt.Type
  ORDER BY COUNT(*) DESC;
GO
```



The PostTypes table is back in the plan.

We can't eliminate it when we need data from it.



Back to the Posts table...

There's a OwnerUserId field.

It relates to the user who asked the question or posted the answer.

Let's make sure that relationship is good.

SadTrombone.com

```
ALTER TABLE dbo.Posts
ADD CONSTRAINT fk_Posts_OwnerUserId
    FOREIGN KEY (OwnerUserId)
REFERENCES dbo.Users(Id);
GO
```

150 %

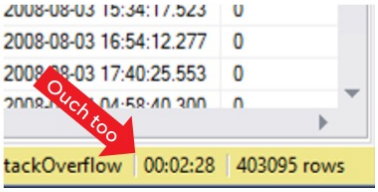
Messages

```
Msg 547, Level 16, State 0, Line 24
The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "fk_Posts_OwnerUserId".
The conflict occurred in database "StackOverflow", table "dbo.Users", column 'Id'.
```

You can't add a foreign key if some of the data doesn't match.

So now we have to go find the bad data...

And this isn't quick.



Implementing (and enforcing) constraints isn't easy:

- You need object locks to enable/disable
- You need supporting indexes on both tables
- In real-world operations, you may need to create the indexes ahead of time before the FK implementation starts, just to ease that pain
- You may already be way off the 5 & 5 rule

You could fake it, but:

```
ALTER TABLE dbo.Posts WITH NOCHECK
ADD CONSTRAINT fk_Posts_OwnerUserId
    FOREIGN KEY (OwnerUserId)
REFERENCES dbo.Users(Id)
GO
```

150 %

Messages

Commands completed successfully.

You can create the constraint without checking the data.

This does work – and finishes instantly…

But SQL Server doesn't trust the contents, so it won't use this to do join elimination.

We warn you about that in sp_Blitz:

```
EXEC sp_Blitz;
```

150 %

Results | Messages

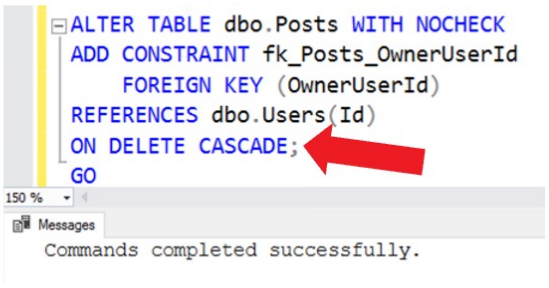
	Priority	FindingsGroup	Finding	DatabaseName	URL	Details
38	100	Performance	Stored Procedure WITH RECOMPILE	StackOverflow_SAN	https://BrentOzar.com/go/recompile	[StackOverflow_SAN].[dbo].[usp_ServerLab2_CPU] has WITH RECOMPILE in the s
39	100	Performance	Stored Procedure WITH RECOMPILE	StackOverflow_SAN	https://BrentOzar.com/go/recompile	[StackOverflow_SAN].[dbo].[usp_ServerLab3] has WITH RECOMPILE in the stored
40	100	Performance	Stored Procedure WITH RECOMPILE	StackOverflow_SAN	https://BrentOzar.com/go/recompile	[StackOverflow_SAN].[dbo].[usp_ServerLab4] has WITH RECOMPILE in the stored
41	150	Performance	Foreign Keys Not Trusted	StackOverflow	https://BrentOzar.com/go/trust	The [StackOverflow] database has foreign keys that were probably disabled, data wa
42	200	Backup	MSDB Backup History Not Purged	msdb	https://BrentOzar.com/go/history	Database backup history retained back to Feb 9 2019 8:38PM

Cascadingly worse.

Cascading updates/deletes: allow queries to delete the parent, and deletes flow down to the child tables.

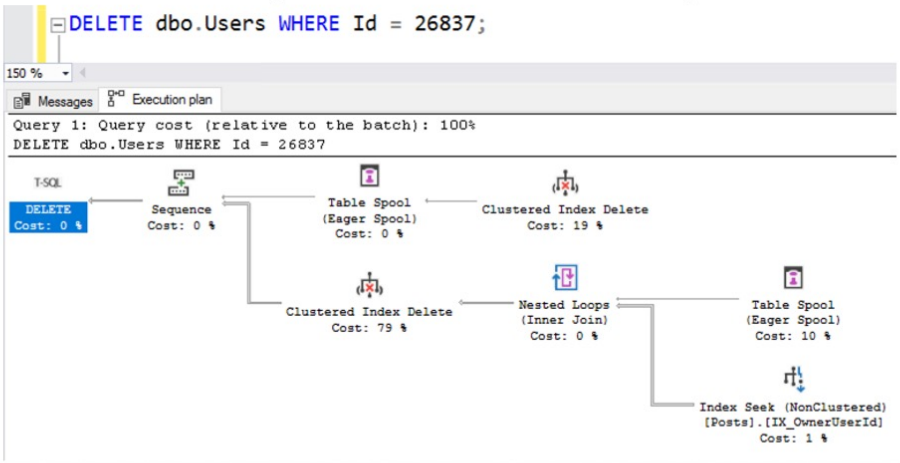
Example: delete a user, and delete all the Posts/Comments they own.

It sounds good in theory – it might prevent the bad data we're seeing in the StackOverflow database, right?



And it does work...

Delete a user, and it deletes their posts.



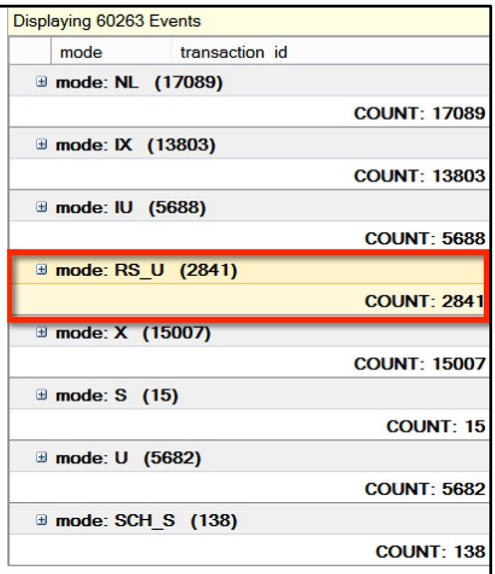
Huge drawback:

Whoa – range locks occurred

The cascading delete used the serializable isolation level secretly against the Posts table

This is an "isolation level upgrade"

http://blogs.msdn.com/b/conor_cunningham_msft/archive/2009/03/13/conor-vs-isolation-level-upgrade-on-update-delete-cascading-ri.aspx



Cascading deletes/updates are like drugs.

If you don't use them too often, they're not bad. The first hit is totally free.

If frequently used, serializable locks can cause locking issues and deadlocks.

Don't just cascade deletes to avoid coding: identify the most optimal delete patterns for an application based on performance and user requirements.

Say we use a 3rd party app

And it was originally designed to house data for lots of different company codes.

```
/* Say we have a CompanyCode column in Users: */
EXEC sp_rename 'dbo.Users.Age', 'CompanyCode'
GO
/* And say everyone has the same CompanyCode:
 (this will take a minute) */
UPDATE dbo.Users SET CompanyCode = 100;

/* And all of our queries always ask for that CompanyCode: */
SELECT *
    FROM dbo.Users
    WHERE CompanyCode = 100
    AND DisplayName = N'Brent Ozar';
```

All my missing indexes ask for CompanyCode, which blows

```
105  /* Say we have a CompanyCode column in Users: */
106  EXEC sp_rename 'dbo.Users.Age', 'CompanyCode'
107  GO
108 /* And say everyone has the same CompanyCode:
109    (this will take a minute) */
110 UPDATE dbo.Users SET CompanyCode = 100;
111
112  /* And all of our queries always ask for that CompanyCode: */
113 SELECT *
114      FROM dbo.Users
115      WHERE CompanyCode = 100
116      AND DisplayName = N'Brent Ozar';
117
```

150 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [dbo].[Users] WHERE [CompanyCode]=@1 AND [DisplayName]=@2
Missing Index (Impact 99.9808): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([CompanyCode],[DisplayName]
```

1.6 p25

Because SQL doesn't know

Unless...we tell it with a constraint, which the query optimizer can use when building a plan:

```
ALTER TABLE dbo.Users
    ADD CONSTRAINT CompanyCodeIsAlways100
    CHECK (CompanyCode = 100);
```


But...SQL Server, c'mon.

Even with the check constraint, he still asks for it.

```
121 Will a check constraint fix it? */
122 ALTER TABLE dbo.Users
123     ADD CONSTRAINT CompanyCodeIsAlways100
124     CHECK (CompanyCode = 100);
125
126 /* And then try your query again: */
127 SELECT *
128     FROM dbo.Users
129     WHERE CompanyCode = 100
130     AND DisplayName = N'Brent Ozar';
131
```

150 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [dbo].[Users] WHERE [CompanyCode]=@1 AND [DisplayName]=@2
Missing Index (Impact 99.9808): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Users] ([CompanyCode],[DisplayName])
```

1.6 p27

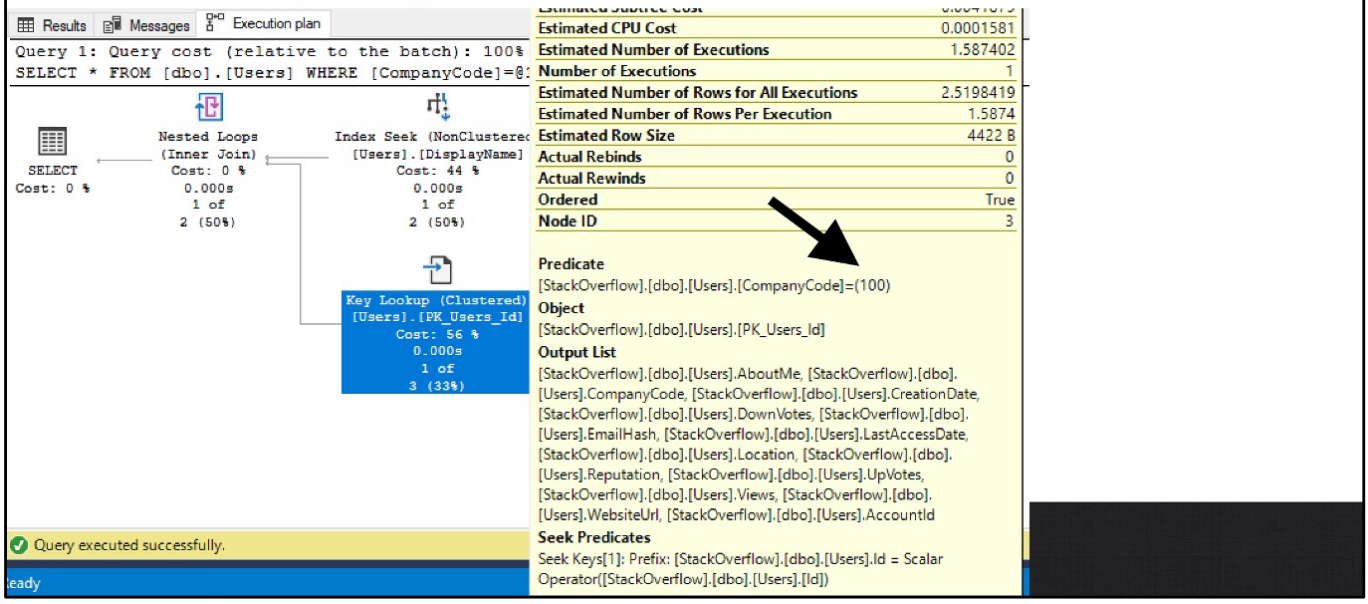
I can index on DisplayName

But…I don't need a constraint to do this:

```
/* Add an index just on DisplayName: */
CREATE INDEX DisplayName ON dbo.Users(DisplayName);

/* And then try your query again, and check the key lookup predicates: */
SELECT *
    FROM dbo.Users
    WHERE CompanyCode = 100
    AND DisplayName = N'Brent Ozar';
```

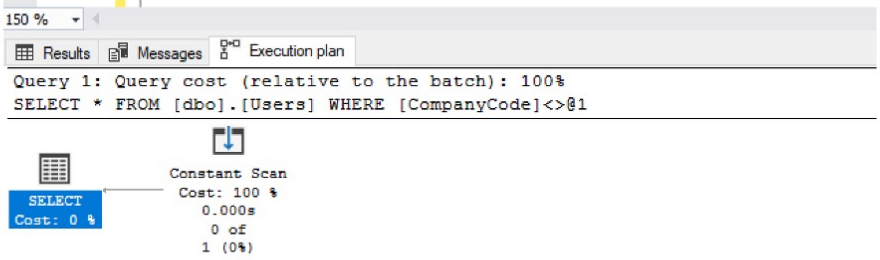

And SQL Server still checks the CompanyCode of each user.



The only benefit of the constraint

If you query for data that truly isn't there, SQL Server can avoid looking at the table altogether.

```
141  /* SQL Server will eliminate this, at least: */
142  SELECT * FROM dbo.Users WHERE CompanyCode <> 100;
143
```



So constraints help when...

Users are querying for data that can't exist, or

You need to stop bad data from going into the database

Aaaaaand that's about it.

So what do we do?

FKs & constraints in one slide

For new applications, new databases, or new tables:

- Every table starts with a clustered primary key
- Implement foreign key constraints
- Index them by default (because you'll probably join on 'em too)
- They may prevent new, buggy code from inserting bad data

For existing databases:

- There's likely already bad data you won't be allowed to delete
- Implement clustered indexes (but may not be unique)
- Implement constraints only to solve a pain point
- Otherwise, if you're facing performance pains, don't be surprised if constraints hurt as much as they help