BRENT OZAR
UNLIMITED®

Improving SELECT * Query Performance

Because you're a horrible person who won't choose columns instead

Just like Jimi Hendrix …

We love to get feedback

Please complete the session feedback forms

SQLBits - It's all about the community...

Please visit Community Corner!

This year we're trying to get more people to learn about the SQL Community, equally if you would be happy to visit the community corner we'd really appreciate it.

Agenda

Here's what we'll be covering:

- Why SELECT * happens
- **When SELECT * isn't that bad**
- When SELECT * is that bad
- How SELECT * makes indexing hard
- How to make SELECT * less bad
- Picture to distract from short agenda

Why would you do that?

From lazy to here

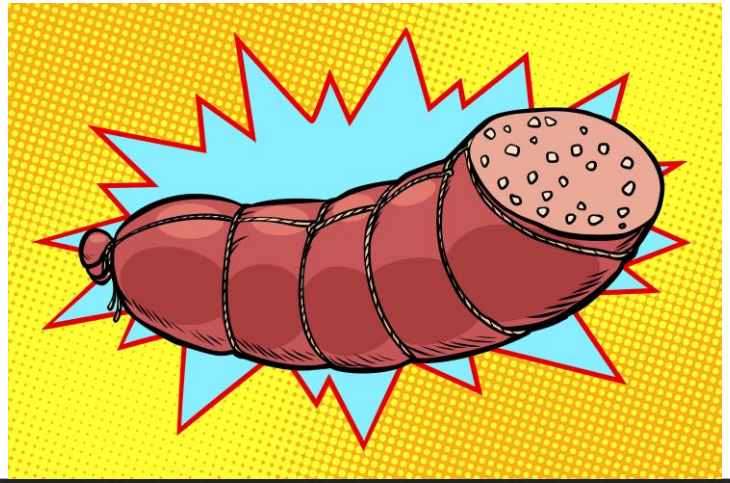
Sometimes it’s innocent

- Do I need all these columns?
- I better get them to be safe

Clown shoes

Sometimes it's just silly

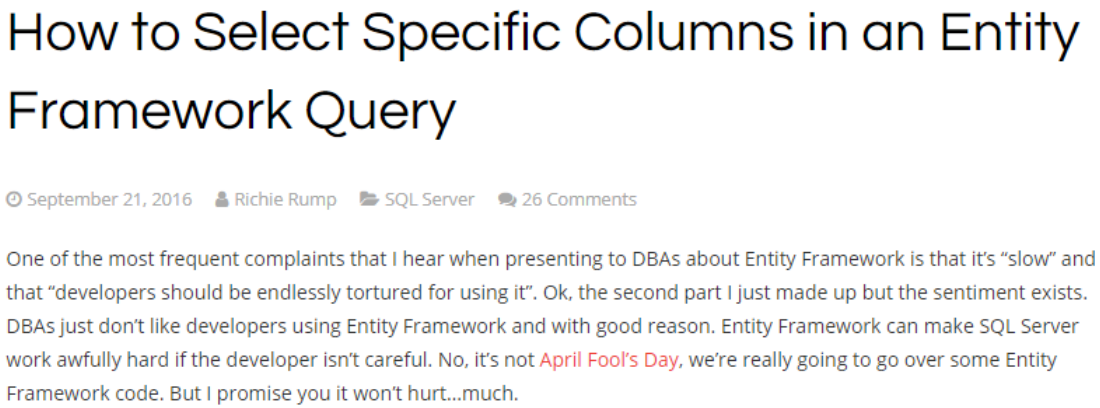
- The PM just kept asking for more data in the report
- We ran out of data so we just made extra columns up



You're the worst

Other times, it's laziness

- I didn't know you could choose columns
- Choosing columns in code is hard



How to Select Specific Columns in an Entity Framework Query

September 21, 2016 Richie Rump SQL Server 26 Comments

One of the most frequent complaints that I hear when presenting to DBAs about Entity Framework is that it's "slow" and that "developers should be endlessly tortured for using it". Ok, the second part I just made up but the sentiment exists. DBAs just don't like developers using Entity Framework and with good reason. Entity Framework can make SQL Server work awfully hard if the developer isn't careful. No, it's not April Fool's Day, we're really going to go over some Entity Framework code. But I promise you it won't hurt...much.

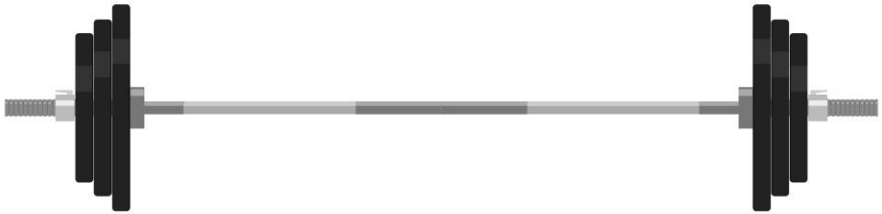
https://www.brentozar.com/archive/2016/09/select-specific-columns-entity-framework-query/

I didn’t know it was bad

SELECT Id

SELECT Id, DisplayName

SELECT Id,
DisplayName, AboutMe,
Website, CreationDate



SELECT *

Dealing with it

You're left with pretty grim choices.

1. Make a really wide nonclustered index
   - (some key columns) include (every other column)
2. Rearrange your existing clustered index
   - Maybe the wrong key column was chosen to begin with
3. Create a narrow nonclustered index
   - Just the (some key columns)

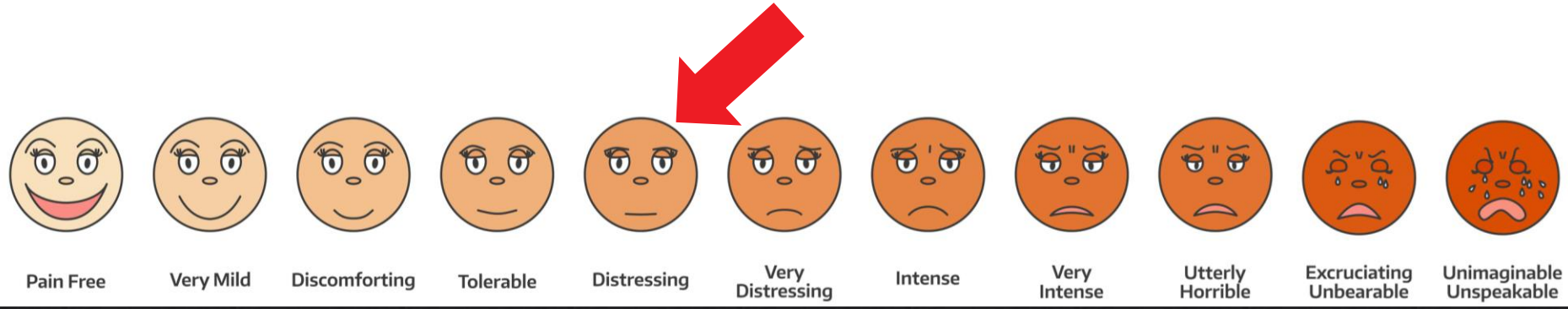
I don’t like any of those

Not that bad

Singles

If you're only grabbing one row, it hurts *less*

- Keep in mind, I'm not saying it's *good*
- You're still reading a lot of potentially unused columns
- If you have MAX cols, you could be reading a lot of extra junk



LOB Data

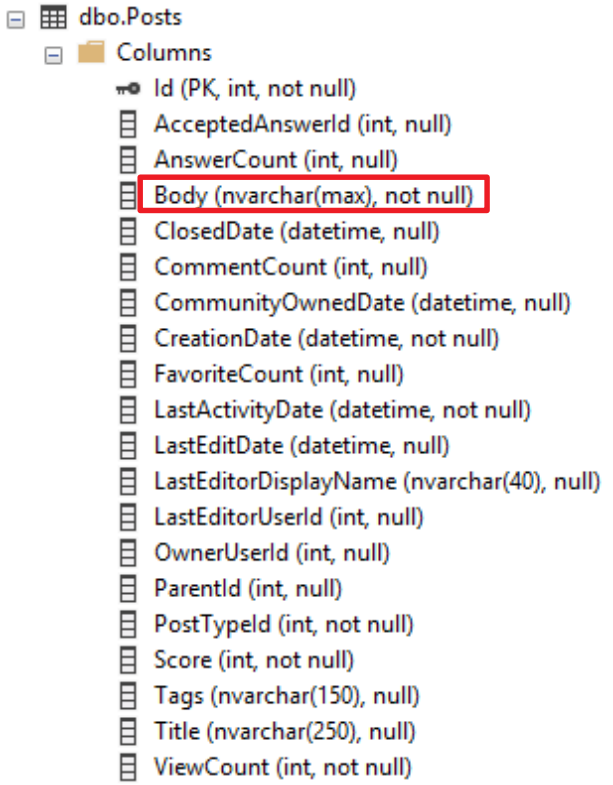
The Stack Overflowed, again

Drainage

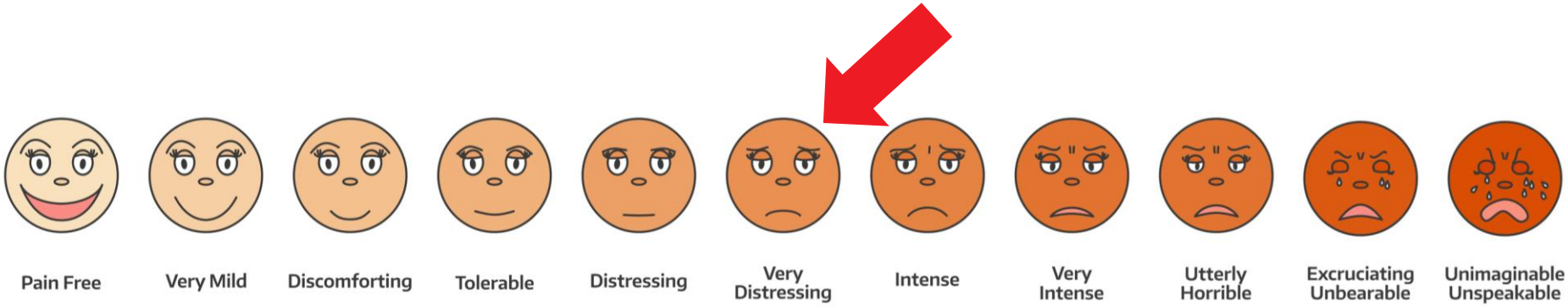
- The Posts table is fairly reasonable
- But the Body column is a nuisance
- NVARCHAR(MAX)
  - Has the text from every Q&A EVER

You may have similar tables

- String fields with 4000/8000/MAX
- XML, VARBINARY, JSON
- TEXT/NTEXT if you're really unlucky



Demo


Pain Free
Very Mild
Discomforting
Tolerable
Distressing
Very Distressing
Intense
Very Intense
Utterly Horrible
Excruciating Unbearable
Unimaginable Unspeakable

Data General

SQL and memory

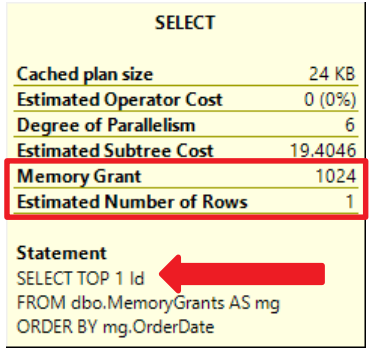
Queries sometimes ask for memory

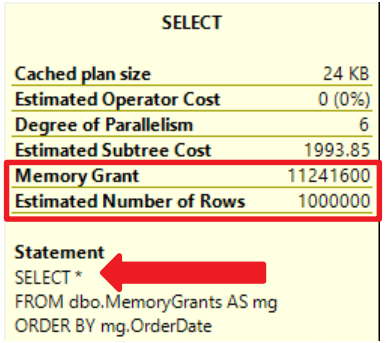
They ask for it for different reasons

- Sorts
- Hashes (joins and aggregates)

Some queries ask for more than others

- Number of rows
- Number of columns
- Data types
- Parallelism





Estimations

For variable length datatypes, SQL guesses

- Size of data in the column will be ½ of the size of the column

Times the estimated number of rows

- SQL uses cardinality estimates and row size to calculate additional memory needed for the query

Why does that happen?

- Hash and Sort ops require all rows to arrive before they start
- The more they can process in memory, the better
- Spilling to disk is bad and slow

Demo

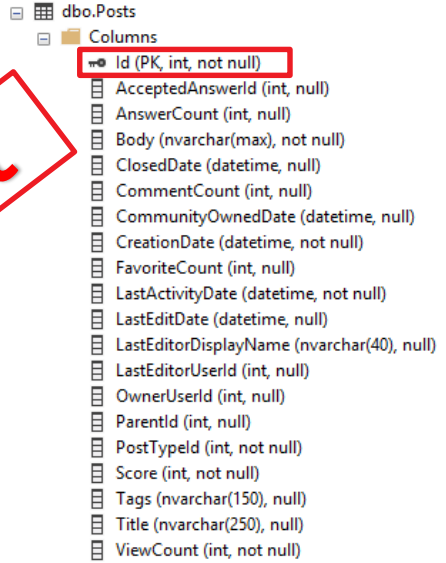

Pain Free
Very Mild
Discomforting
Tolerable
Distressing
Very Distressing
Intense
Very Intense
Utterly Horrible
Excruciating Unbearable
Unimaginable Unspeakable

Indexing Challenges

How do you index for this?

```
SELECT *
FROM dbo.Posts AS p
WHERE p.OwnerUserId = 22656
```

Not It



Key on OwnerUserId...

No Includes?

Some Includes?

All Includes?

What SQL thinks you should do

The optimizer is clearly out of its mind

```
/*
Missing Index Details from SQLQuery13.sql - NADAULTRA\SQL2016E.StackOverflow (sa (62))
The Query Processor estimates that implementing the following index could improve the query cost by 99.9688%.
*/

/*
USE [StackOverflow]
GO
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [dbo].[Posts] ([OwnerUserId])
INCLUDE ([Id],[AcceptedAnswerId],[AnswerCount],[Body],[ClosedDate],[CommentCount],[CommunityOwnedDate],[CreationDate],[FavoriteCount],[LastActivityDate],[LastEditDate],[LastEditorDisplayName],[LastEditorUserId],[ParentId],
   [PostTypeId],[Score],[Tags],[Title],[ViewCount])
GO
*/
```

I already have a clustered index

Did I choose poorly?

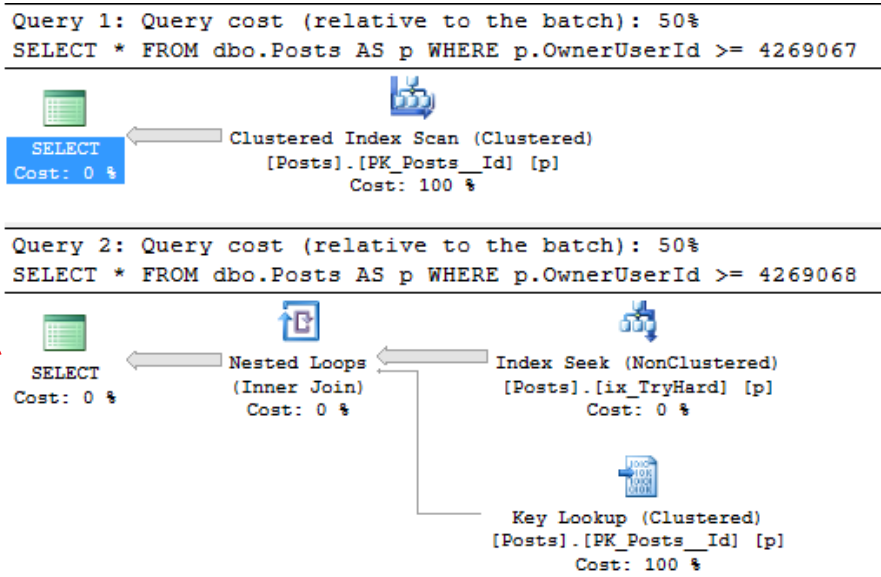
The optimizer is lazy

It doesn't want to make these choices

```
SELECT *
FROM dbo.Posts AS p
WHERE p.OwnerUserId >= 4269067

SELECT *
FROM dbo.Posts AS p
WHERE p.OwnerUserId >= 4269068
```

```
CREATE INDEX ix_TryHard on dbo.Posts (OwnerUserId)
```



I know the secret of the…

Index matching: Like the dating game

- Query: My perfect date is a romantic dinner
- Table: Let’s get drunk with my friends
- Query: I’ll just write my number on the bathroom wall

Mirror in the bathroom

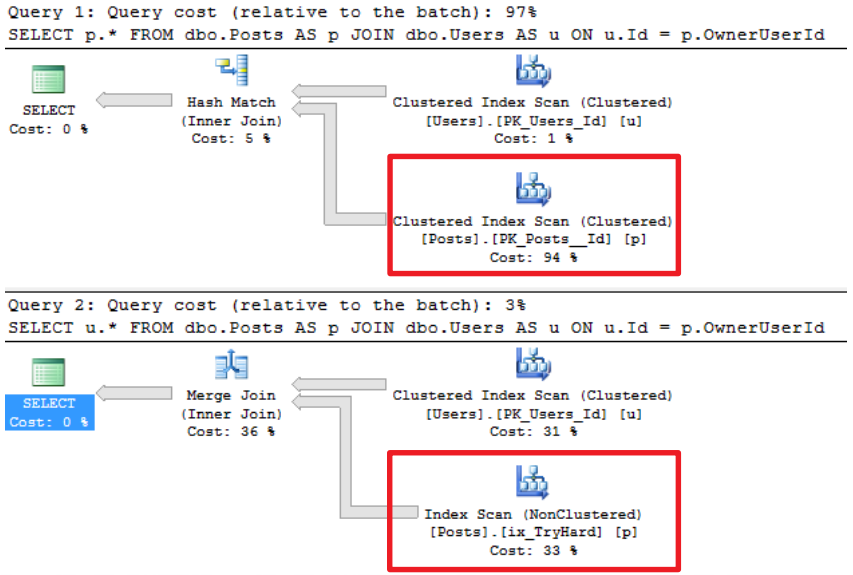
That’s what a missing index request is

The cry for help of a really bad date

This gets harder with joins

```
SELECT p.*
FROM dbo.Posts AS p
JOIN dbo.Users AS u
ON u.Id = p.OwnerUserId
```

```
SELECT u.*
FROM dbo.Posts AS p
JOIN dbo.Users AS u
ON u.Id = p.OwnerUserId
```



Annoying questions

How many columns can I select from Posts before my narrow index stops getting used?

Is there a certain combination of columns that changes this?

What about other predicates that reduce rows returned?

```
SELECT p.*
FROM dbo.Posts AS p
JOIN dbo.Users AS u
ON u.Id = p.OwnerUserId
```

Annoying questions

How many columns can I select from Posts before my narrow index stops getting used?

Eleventy

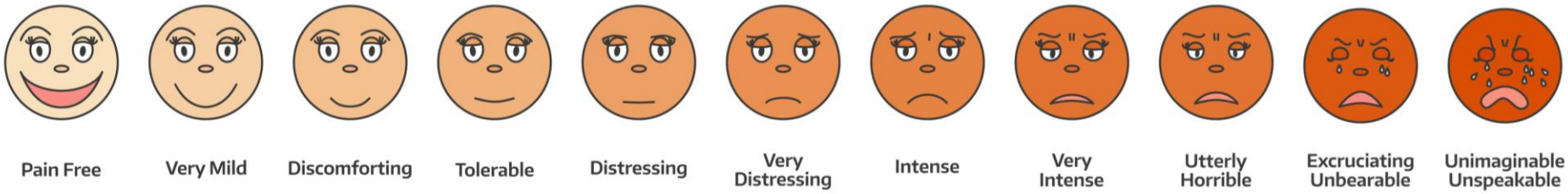
Is there a certain combination of columns that changes this?

Yep

What about other predicates that reduce rows returned?

Yep

```
SELECT p.*
FROM dbo.Posts AS p
JOIN dbo.Users AS u
ON u.Id = p.OwnerUserId
```

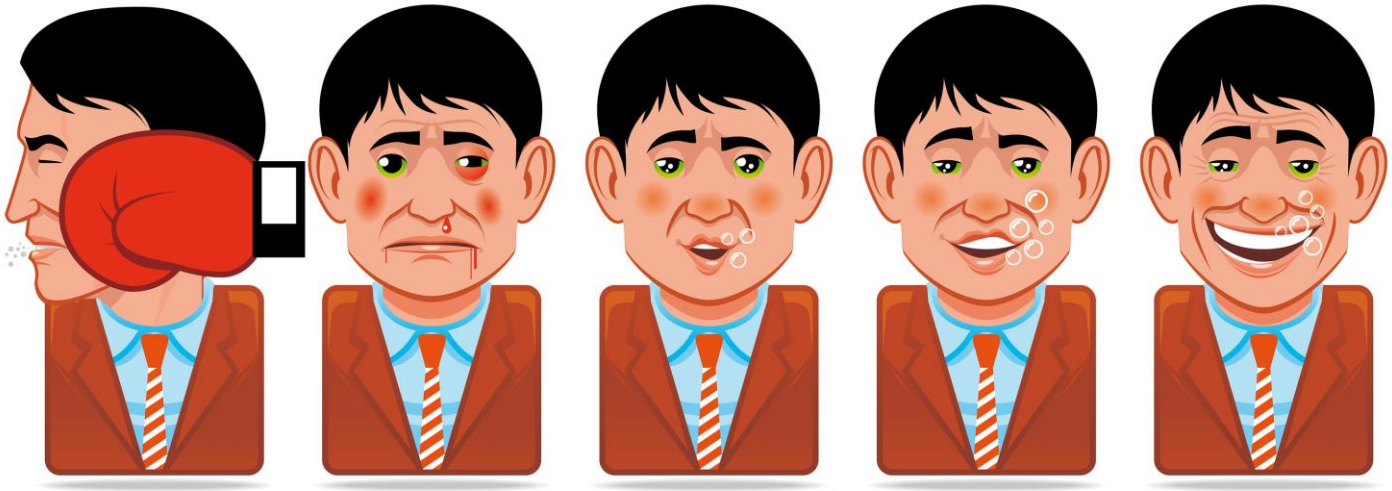

Pain Free
Very Mild
Discomforting
Tolerable
Distressing
Very Distressing
Intense
Very Intense
Utterly Horrible
Excruciating Unbearable
Unimaginable Unspeakable

Deferring The Pain

This will never be pain free

You want to use narrow nonclustered indexes

You're okay with changing code and experimenting with indexes



Our starting query

```
SELECT u.*, p.Id AS [PostId]
FROM   dbo.Users AS u
JOIN   dbo.Posts AS p
ON p.OwnerUserId = u.Id
WHERE  u.CreationDate > '20160101'
       AND u.Reputation > 100
       AND p.PostTypeId = 1
```

Joins the Users table to the Posts table

Our indexes

All the pretty little clusters

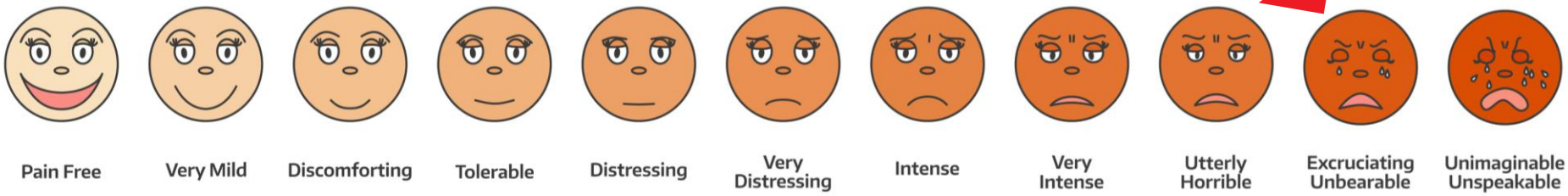
- Users: Id
- Posts: Id (but we join on OwnerUserId from here)

Nothin' but nonclustered:

- Users: CreationDate, Reputation, Id
- Posts: OwnerUserId, Id INCLUDE PostTypeId

These should make our query happy

Demo


Pain Free
Very Mild
Discomforting
Tolerable
Distressing
Very Distressing
Intense
Very Intense
Utterly Horrible
Excruciating Unbearable
Unimaginable Unspeakable

How do we make this smarter?

We don't want to keep that hint

- Forcing an index hint takes choices away from the optimizer
- The Key Lookup plan might not be awesome for all predicates
- Data may change, and it might not even *stay* awesome for this query

We do want to use our index

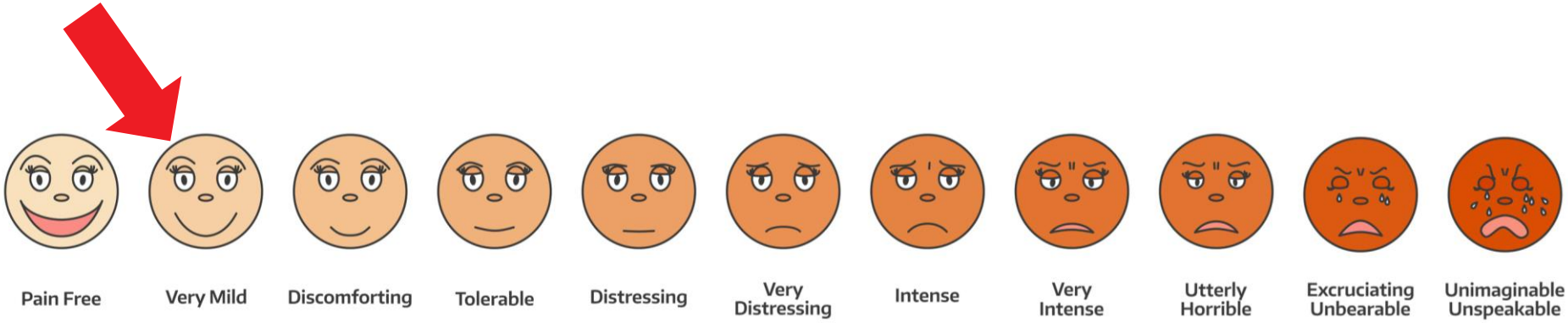
- How can we tell SQL that it's safe to use?
- Let's take temp tables off the table
  - If you're thinking table variables, go home

Demo

Commonality

```
WITH precheck AS (
    SELECT u.Id, p.Id AS [PostId]
    FROM dbo.Users AS u
    JOIN   dbo.Posts AS p
    ON p.OwnerUserId = u.Id
    WHERE u.CreationDate > '20160101'
    AND u.Reputation > 100
    AND p.PostTypeId = 1
                )
SELECT u.*, p.PostId
FROM precheck p
JOIN dbo.Users AS u
ON p.Id = u.Id
```

CTEs can help!


Pain Free
Very Mild
Discomforting
Tolerable
Distressing
Very Distressing
Intense
Very Intense
Utterly Horrible
Excruciating Unbearable
Unimaginable Unspeakable

Recap

This shouldn’t be your first stop

When you see queries that are SELECT * or just wide lists…

- Ask if all the columns are necessary
  - If no one knows, just start removing them until someone complains
- Don’t immediately jump to add wide indexes
  - This should be a last resort for very important queries
- Try indexing just key columns first
  - Sometimes the Key Lookup plan is fine (few rows don’t hurt here)

Put the quill down, Chaucer

If all else fails, go for the rewrite

- You may need to experiment with index key column order
  - Watch out for Sorts, Spools, other nonsense
- It really helps when the column that drives your joins is unique
  - If it's not, you may need to use a unique SET of columns
  - DISTINCT and GROUP BY can be used as well

Try this at home