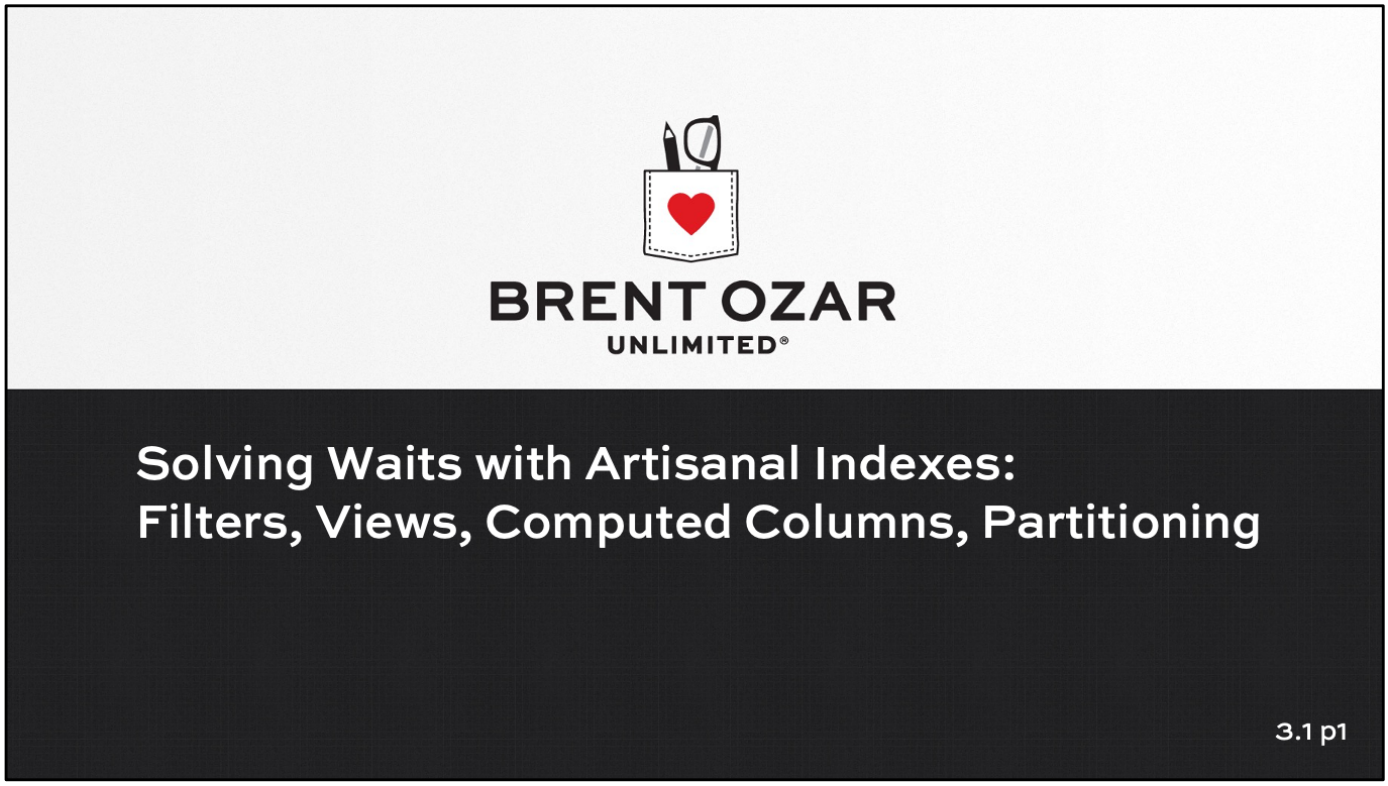


Indexes can benefit from hand-crafting, too

Here's a few key scenarios:

1. App frequently queries a small subset of values in a large table
2. App frequently queries aggregations
3. App frequently queries computations (and the app can't just write the computed value in the first place)
4. App loads a big chunk of data all at once, and then deletes the oldest chunk of data all at once

3.1 p3

A big gotcha

Some of these artisanal indexes have a huge gotcha: they won't work with clients who are using different "SET" options in their connect string

- Can't use it for reads
- Writes will fail

This can break your application

```
Msg 1934, Level 16, State 1, Line 1
INSERT failed because the following SET options have incorrect settings:
```

SET options that impact results

SET options	Required value
ANSI_NULLS	ON
ANSI_PADDING	ON
ANSI_WARNINGS	ON
ARITHABORT	ON
CONCAT_NULL_YIELDS_NULL	ON
NUMERIC_ROUNDABORT	OFF
QUOTED_IDENTIFIER	ON

https://technet.microsoft.com/en-us/library/ms175088

3.1 p5

To catch sessions like this:

```
SELECT *
FROM sys.dm_exec_sessions
WHERE is_user_process = 1 AND
(ansi_nulls = 0
OR ansi_padding = 0
OR ansi_warnings = 0
OR arithabort = 0
OR concat_null_yields_null = 0
OR quoted_identifier = 0)
```

Curse you, SQL Agent

	session_id	login_time	host_name	program_name	host_process_id	client_version	client_interface_name
1	52	2017-01-27 06:00:43.930	SQL2016A	SQLAgent - Email Logger	2452	7	ODBC
2	53	2017-01-27 06:00:43.930	SQL2016A	SQLAgent - Generic Refresher	2452	7	ODBC
3	54	2017-01-27 06:01:30.117	SQL2016A	Microsoft SQL Server Management Studio	3524	7	.Net SqlClient Data Provide
4	55	2017-01-27 06:01:30.677	SQL2016A	Microsoft SQL Server Management Studio	3524	7	.Net SqlClient Data Provide
5	57	2017-01-27 06:06:26.097	SQL2016A	SQLServerCEIP	1444	7	.Net SqlClient Data Provide

3.1 p7

Don’t let this stop you

Just test before you implement these indexes

- Validate application activity

Hey, that’s not so bad, is it?

Just make sure you include validating that WRITES can occur as part of the testing

Normal index creation

```
CREATE NONCLUSTERED INDEX IX_DisplayName
ON dbo.Users(DisplayName);
```

Filtered index

```
CREATE NONCLUSTERED INDEX IX_DisplayName
ON dbo.Users(DisplayName)
WHERE IsEmployee = 0;
```

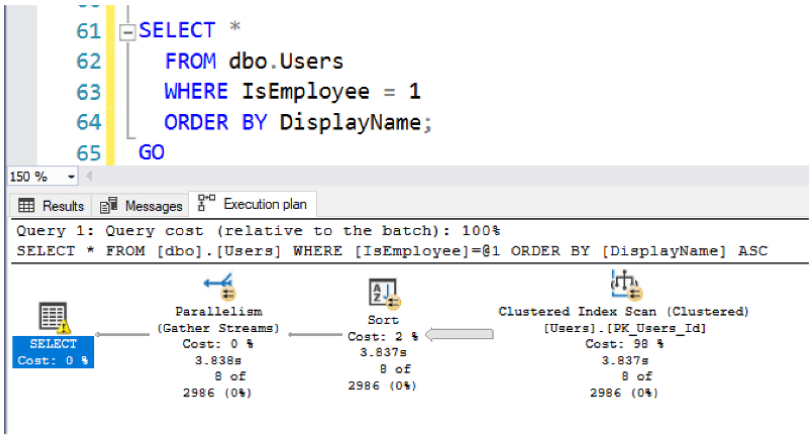
Setting up the Users table

```
ALTER TABLE dbo.Users
    ADD IsEmployee BIT NOT NULL DEFAULT 0;
GO

/* Populate some of the employees: */
UPDATE dbo.Users
    SET IsEmployee = 1
    WHERE Id IN (1, 2, 3, 4, 13249, 23354, 115866, 130213, 146719);
GO
```

Say we have an Employees page

When we filter for IsEmployee = 1, Clippy's quiet.

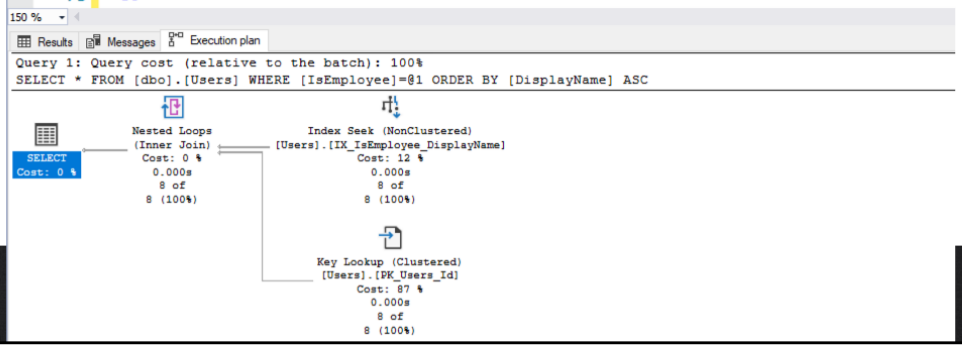


3.1 p13

But it will use a hand-crafted index

Unfiltered, like Mom's Camels

```
67  CREATE INDEX IX_IsEmployee_DisplayName ON dbo.Users(IsEmployee, DisplayName);
68  GO
69  SELECT *
70    FROM dbo.Users
71    WHERE IsEmployee = 1
72    ORDER BY DisplayName;
73  GO
```



3.1 p14

But it's big.

It has all the millions of rows, and takes up space.

When we insert/update/delete, the index changes.

Can we just index the IsEmployee = 1 people?

```
76  sp_BlitzIndex @TableName = 'Users';
77  GO
```

50 %

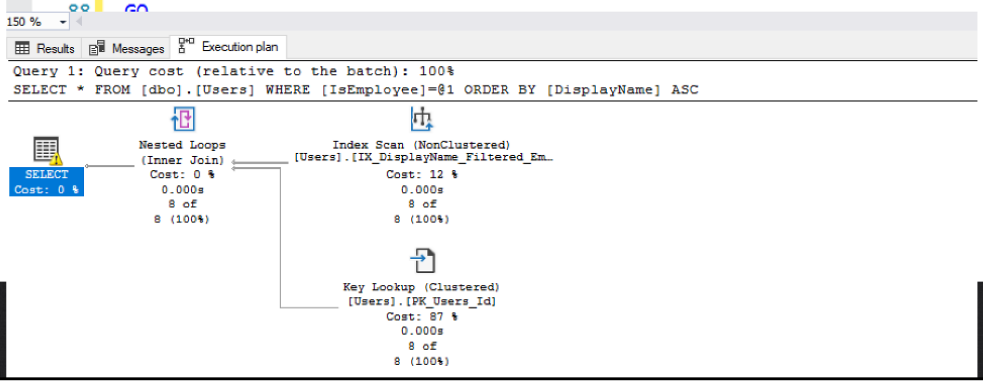
Results | Messages

	Details: db_schema.table.index(indexid)	Definition: [Property] ColumnName {datatype maxbytes}	Size
1	Database [StackOverflow] as of 2020-01-27 08:04 ...	http://FirstResponderKit.org	NULL
2	dbo.Users.PK_Users_Id (1)	[CX] [PK] [1 KEY] Id {int 4}	8,917,507 rows; 1.1GB; 1.2MB LOB; 0.0MB Row Ove...
3	dbo.Users.IX_IsEmployee_DisplayName (3)	[2 KEYS] IsEmployee {bit 1}, DisplayName {nvarchar ...	8,917,507 rows; 305.3MB

3.1 p15

Drop the old, add a filtered

```
79  DropIndexes;
80  GO
81  CREATE INDEX IX_DisplayName_Filtered_Employees ON dbo.Users(DisplayName)
82    WHERE IsEmployee = 1;
83  GO
84  SELECT *
85    FROM dbo.Users
86    WHERE IsEmployee = 1
87    ORDER BY DisplayName;
88  GO
```



3.1 p16

And it's tiny: just 8 rows!

```
91  sp_BlitzIndex @TableName = 'Users';
92  GO
```

150 %

Results | Messages

	Details: db_schema.table.index(indexid)	Definition: [Property] ColumnName {datatype maxbytes}	Size
1	Database [StackOverflow] as of 2020-01-27 08:06 ...	http://FirstResponderKit.org	NULL
2	dbo.Users.IX_DisplayName_Filtered_Employees (2)	[1 KEY] DisplayName {nvarchar 80} [FILTER] ([IsEmployee]=(1))	8 rows; 0.1MB
3	dbo.Users.PK_Users_Id (1)	[CX] [PK] [1 KEY] Id {int 4}	8,917,507 rows; 1.1GB; 1.2MB LOB; 0.0MB Row Ove...

Not only does it take less space, but it also doesn't have to be maintained for rows that don't match the filter.

3.1 p17

Who it's for

Very selective column where only 5% of rows match

Queue table:

- ProcessedDate column defaults to null, set to a valid date after row is processed
- Queries ask for WHERE ProcessedDate IS NULL
- Only a few rows need to be processed, but you want to find them quickly
- You never query the rest of the rows by date

Filtered index limitations

SQL Server 2008+, all editions

Filter can only contain string literals:

- No functions
- Frustrating with “ascending date” type columns

Statistics automatically update at the same rate as the base table (~ 20% of rows updated)

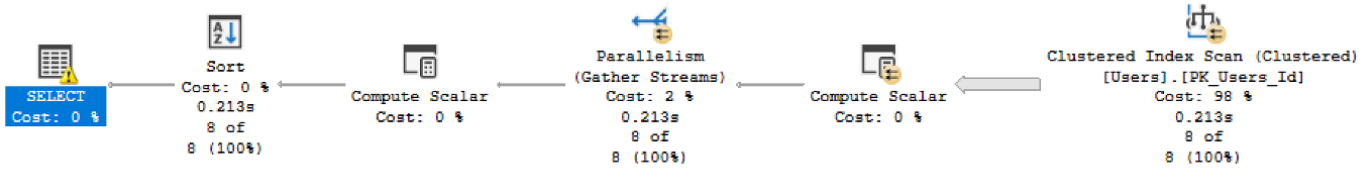
Parameterized queries don't use 'em

```
96  CREATE OR ALTER PROC dbo.usp_SearchUsers @IsEmployee BIT AS
97  SELECT *
98     FROM dbo.Users
99     WHERE IsEmployee = @IsEmployee
100    ORDER BY DisplayName;
101 GO
102 EXEC usp_SearchUsers @IsEmployee = 1;
103 GO
```

150 %

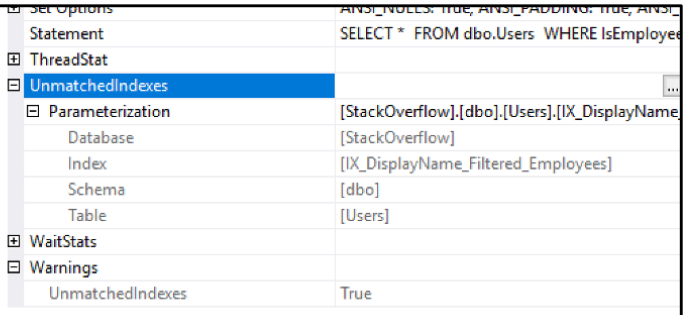
Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM dbo.Users WHERE IsEmployee = @IsEmployee ORDER BY DisplayName
```



Hidden warning

SQL Server has to build a plan that's safe and reusable for everyone.



This same proc could be called for @IsEmployee = 0.

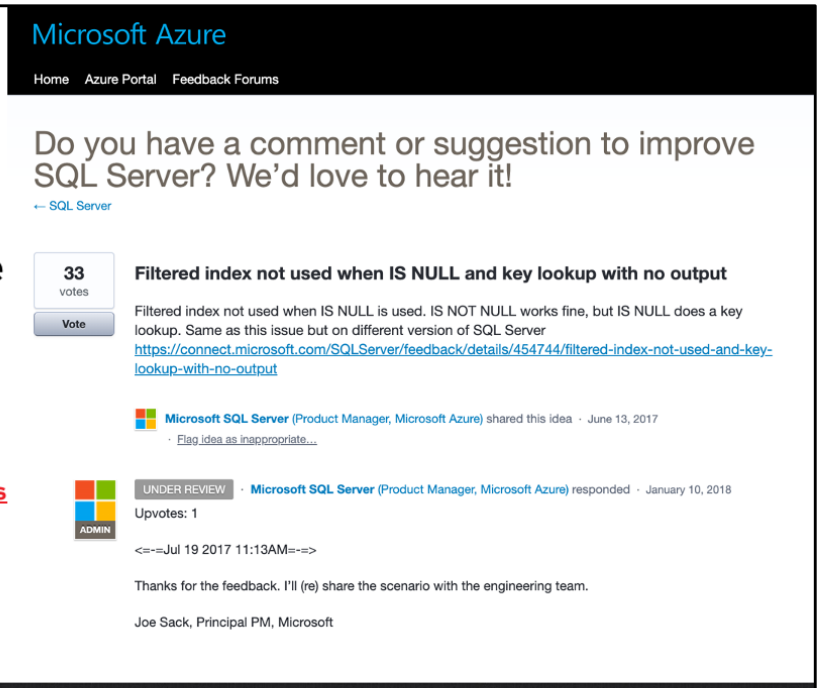
If you really want filtered index usage, you may have to hard-code the filter into the query itself, too.

3.1 p21

WHERE Bugs = 1

You usually need the filtered column in the index somewhere (either keys or includes) or else:

https://feedback.azure.com/forums/908035-sql-server/suggestions/32896348-filtered-index-not-used-when-is-null-and-key-looku

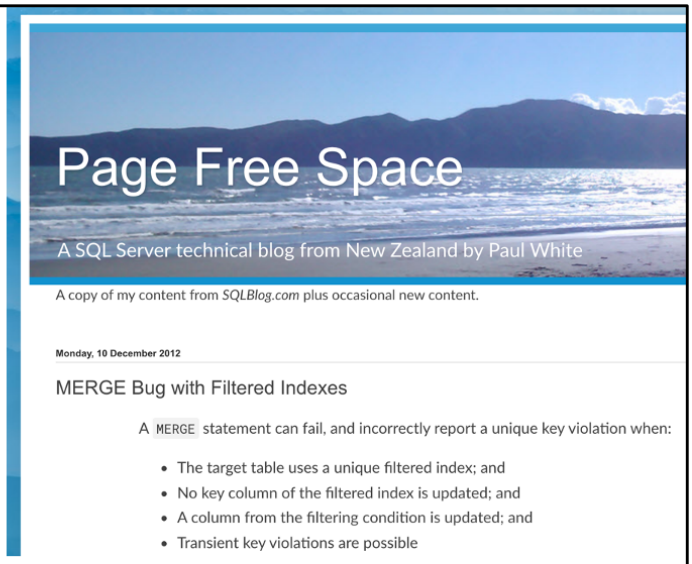


WHERE Bugs = 1

https://www.sql.kiwi/2012/12/merge-bug-with-filtered-indexes.html

Closed as won't-fix, still present in SQL Server 2019

MERGE has many more issues, though



Page Free Space

A SQL Server technical blog from New Zealand by Paul White

A copy of my content from *SQLBlog.com* plus occasional new content.

Monday, 10 December 2012

MERGE Bug with Filtered Indexes

A `MERGE` statement can fail, and incorrectly report a unique key violation when:

- The target table uses a unique filtered index; and
- No key column of the filtered index is updated; and
- A column from the filtering condition is updated; and
- Transient key violations are possible

Use Caution with SQL Server's MERGE Statement

By: Aaron Bertrand | Updated: 2018-07-24 | Comments (25) | Related: 1 | 2 | 3 | 4 | 5 | 6 | More > T-SQL

Problem

SQL Server 2008 introduced the MERGE statement, which promised to be a simpler way to combine insert/update/delete statements, such as those used during ETL (extract, transform and load) operations. However, MERGE originally shipped with several "wrong results" and other bugs - some of which have been addressed, and some of which continue to exist in current versions. People also tend to make some leaps of faith regarding atomicity - they don't realize that the single statement actually performs different operations separately, and thus can suffer from issues due to concurrency and race conditions just like separate statements can.

Solution

I have been recommending that - for now - people stick to their tried and true methods of separate statements. Here are the major reasons:

Bugs with the SQL Server Merge Statement

It can be quite difficult to validate and guarantee that you are immune from any of the bugs that still exist. A few Connect items that you should be aware of, that are either still active, closed as "Won't Fix"/"By Design", or have only been fixed in specific versions (often requiring a cumulative update or on-demand hotfix):

Connect issue	Current / last known status
#773895 : MERGE Incorrectly Reports Unique Key Violations	Won't Fix
#771336 : Indexed view is not updated on data changes in base table	Fixed only in 2012+
#766165 : MERGE evaluates filtered index per row, not post operation, which causes filtered index violation	Won't Fix
#723696 : Basic MERGE upsert causing deadlocks	By Design
#713699 : A system assertion check has failed ("cxrowset.cpp":1528)	Won't Fix
#699055 : MERGE query plans allow FK and CHECK constraint violations	Active

3.1 p24

Connect issue	Current / last known status
#773895 : MERGE Incorrectly Reports Unique Key Violations	Won't Fix
#771336 : Indexed view is not updated on data changes in base table	Fixed only in 2012+
#766165 : MERGE evaluates filtered index per row, not post operation, which causes filtered index violation	Won't Fix
#723696 : Basic MERGE upsert causing deadlocks	By Design
#713699 : A system assertion check has failed ("cxrowset.cpp":1528)	Won't Fix
#699055 : MERGE query plans allow FK and CHECK constraint violations	Active
#685800 : Parameterized DELETE and MERGE Allow Foreign Key Constraint Violations	Won't Fix
#654746 : merge in SQL2008 SP2 still suffers from "Attempting to set a non-NULL-able column's value to NULL"	Active
#635778 : NOT MATCHED and MATCHED parts of a SQL MERGE statement are not optimized	Won't Fix
#633132 : MERGE INTO WITH FILTERED SOURCE does not work properly	Won't Fix
#596086 : MERGE statement bug when INSERT/DELETE used and filtered index	Won't Fix
#583719 : MERGE statement treats non-nullable computed columns incorrectly in some scenarios	Won't Fix
#581548 : SQL2008 R2 Merge statement with only table variables fails	Fixed only in 2012+
#539084 : Search condition on a non-key column and an ORDER BY in source derived table breaks MERGE completely	Won't Fix
#357419 : MERGE statement bypasses Referential Integrity	Fixed only in 2012+
Merge statement fails when running db in Simple recovery model (2016)	Fixed only with trace flag 692
MERGE statement assertion error when database is in simple recovery model (2017)	Fixed only with trace flag 692
MERGE and INSERT with COLUMNSTORE index creates crash dump	Unacknowledged
Support MERGE INTO target for memory optimized tables	Under Review
MERGE fails with a duplicate key error when using DELETE and INSERT actions	Under Review
CDC logging wrong operation type for a MERGE statement, when Unique index is present	Under Review
Query optimizer cannot make plan for merge statement	Under Review
Poor error message with MERGE when source/target appear in impossible places	Won't Fix
MERGE statement provokes deadlocking due to incorrect locking behavior	Under Review
Merge statement Delete does not update indexed view in all cases	Under Review
EXCEPTION_ACCESS_VIOLATION When referencing the "deleted" table from an OUTPUT statement during MERGE	Under Review
Fulltext index not updating after changing text column via MERGE statement on a partitioned table	Under Review
MERGE on not matched by source UPDATE ignores variable declaration	Under Review
"A severe error occurred on the current command." with MERGE and OUTPUT	Fixed only in 2014+

3.1 p25

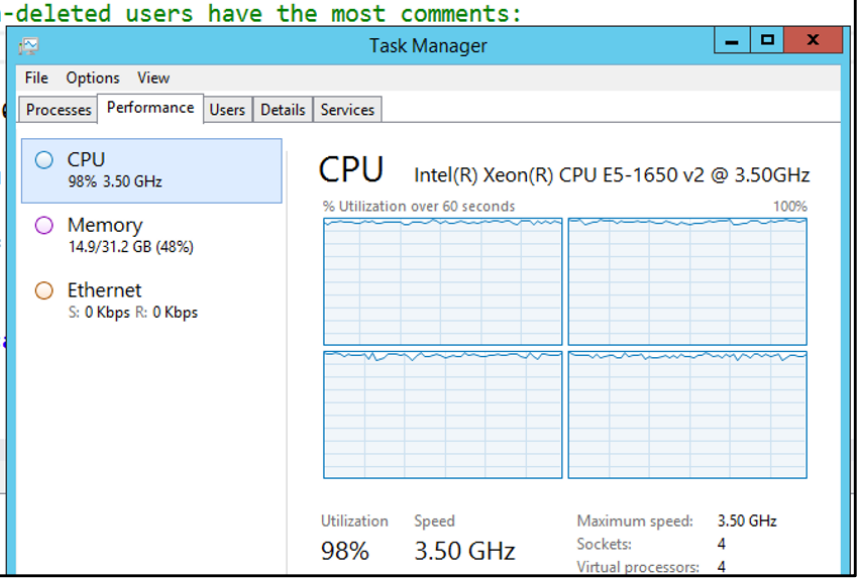
Okay. Here's the situation.

```
/*
Say we need to quickly find which non-deleted users have the most comments:
*/
ALTER TABLE dbo.Comments
  ADD IsDeleted BIT NOT NULL DEFAULT 0;
GO
SELECT TOP 100 u.Id, u.DisplayName, u.Location, u.AboutMe, SUM(1) AS CommentCount
  FROM dbo.Users u
  INNER JOIN dbo.Comments c ON u.Id = c.UserId
  WHERE u.IsDeleted = 0
    AND c.IsDeleted = 0
  GROUP BY u.Id, u.DisplayName, u.Location, u.AboutMe
  ORDER BY SUM(1) DESC;
GO
```

A lot of comments, so this sucks.

```
/*
Say we need to quickly find which non-deleted users have the most comments:
*/
ALTER TABLE dbo.Comments
  ADD IsDeleted BIT NOT NULL DEFAULT (
GO
SELECT TOP 100 u.Id, u.DisplayName, u
  FROM dbo.Users u
  INNER JOIN dbo.Comments c ON u.Id =
  WHERE u.IsDeleted = 0
    AND c.IsDeleted = 0
  GROUP BY u.Id, u.DisplayName, u.Loca
  ORDER BY SUM(1) DESC;
GO
```

Messages



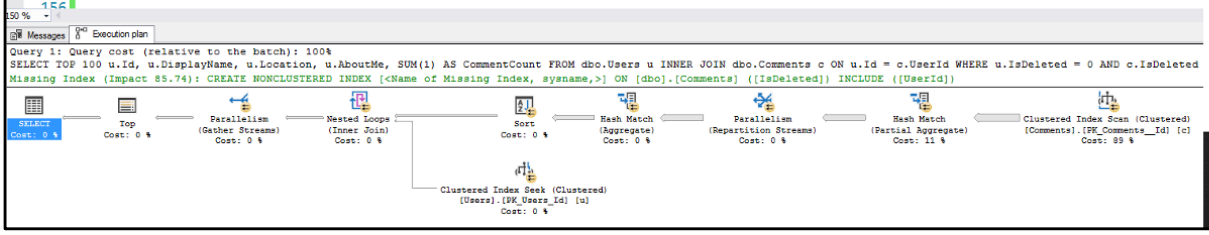
The plan does suggest an index.

And it's one of the worst indexes you could design.

It keys on Comments.IsDeleted, which is all zeroes.

Then it just INCLUDES UserId without even sorting it!

```
148 SELECT TOP 100 u.Id, u.DisplayName, u.Location, u.AboutMe, SUM(1) AS CommentCount
149   FROM dbo.Users u
150   INNER JOIN dbo.Comments c ON u.Id = c.UserId
151   WHERE u.IsDeleted = 0
152     AND c.IsDeleted = 0
153   GROUP BY u.Id, u.DisplayName, u.Location, u.AboutMe
154   ORDER BY SUM(1) DESC;
155 GO
```



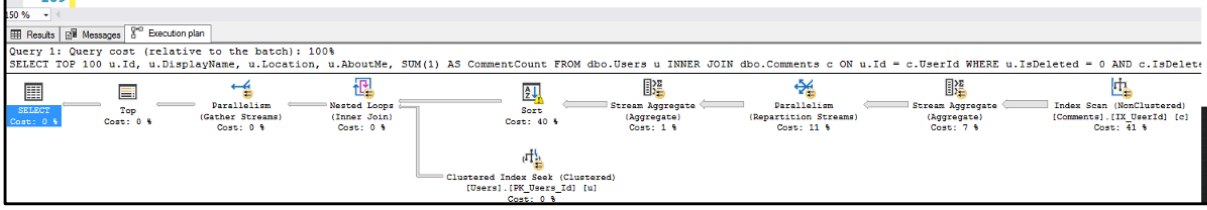
3.1 p29

We could do better manually, but...

We can create a filtered index on UserId just for non-deleted comments.

However, we're still gonna have to scan the whole thing and run totals.

```
161 SELECT TOP 100 u.Id, u.DisplayName, u.Location, u.AboutMe, SUM(1) AS CommentCount
162   FROM dbo.Users u
163   INNER JOIN dbo.Comments c ON u.Id = c.UserId
164   WHERE u.IsDeleted = 0
165     AND c.IsDeleted = 0
166   GROUP BY u.Id, u.DisplayName, u.Location, u.AboutMe
167   ORDER BY SUM(1) DESC;
168 GO
169
```



3.1 p30

The query is much faster now.

It's way better than it was, finishing in seconds.

It's only doing 81K logical reads, which is good.

However, it's going parallel and burning multiple cores to do all this totaling every time it runs, and that never gets cached.

```
(100 rows affected)
Table 'Comments'. Scan count 5, logical reads 80694, physi
Table 'Users'. Scan count 0, logical reads 488, physical r
Table 'Worktable'. Scan count 0, logical reads 0, physical
Table 'Worktable'. Scan count 0, logical reads 0, physical

(1 row affected)

 SQL Server Execution Times:
   CPU time = 9844 ms,  elapsed time = 5727 ms.
```

3.1 p31

Enter the indexed view, also known as a materialized view.

Normally we think of views as syntax shortcuts.

Indexed views are good for:

- **Pre-baking CPU-intensive aggregations**
- **Pre-baking joins (which isn't usually a big deal, except at scale)**

Creating the view and the clustered index

```
CREATE OR ALTER VIEW dbo.vwCommentsByUser WITH SCHEMABINDING AS
    SELECT UserId,
        SUM(1) AS CommentCount,
        COUNT_BIG(*) AS MeanOldSQLServerMakesMeDoThis
    FROM dbo.Comments
    WHERE IsDeleted = 0
    GROUP BY UserId;
GO
CREATE UNIQUE CLUSTERED INDEX CL_UserId ON dbo.vwCommentsByUser(UserId);
GO
```

3.1 p33

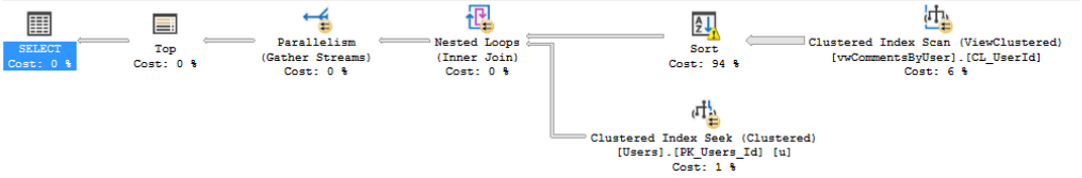
Rerun your query with no changes

```
185 SELECT TOP 100 u.Id, u.DisplayName, u.Location, u.AboutMe, SUM(1) AS CommentCount
186   FROM dbo.Users u
187   INNER JOIN dbo.Comments c ON u.Id = c.UserId
188   WHERE u.IsDeleted = 0
189     AND c.IsDeleted = 0
190   GROUP BY u.Id, u.DisplayName, u.Location, u.AboutMe
191   ORDER BY SUM(1) DESC;
192 GO
193
```

150 %

Results | Messages | Execution plan

Query 1: Query cost (relative to the batch): 100%
SELECT TOP 100 u.Id, u.DisplayName, u.Location, u.AboutMe, SUM(1) AS CommentCount FROM dbo.Users u INNER JOIN dbo.Comment



3.1 p34

Reads and CPU time drop, too

```
(100 rows affected)
Table 'Users'. Scan count 0, logical reads 481, physical rea
Table 'Worktable'. Scan count 0, logical reads 0, physical r
Table 'vwCommentsByUser'. Scan count 5, logical reads 5346,
Table 'Worktable'. Scan count 0, logical reads 0, physical r

(1 row affected)

 SQL Server Execution Times:
   CPU time = 1250 ms,  elapsed time = 474 ms.
```

THIS ALMOST NEVER WORKS.

SQL Server Enterprise Edition is supposed to auto-match your query to the newly created indexed view, without changing your query.

In reality, this almost never works, and you have to:

- Change your query to point to the view
- Use the WITH (NOEXPAND) hint on your query to force SQL Server to use not just the view, but the index on the view
- Do a little dance to the indexed view gods

Indexed view fine print

View must be created with schemabinding

- References only tables in the same database
- No outer joins, self joins, or subqueries
- No OVER clause (or ranking/windowing functions)
- No OUTER or CROSS APPLY

If you're grouping (most common use)

- The view must contain a COUNT_BIG column
- No HAVING
- No MIN, MAX, TOP, or ORDER BY

The clustered index on the view must be unique

Lots more rules – Books Online has a full list:

- BrentOzar.com/go/viewrules

Indexed view limitations

"Why isn't SQL using my indexed view?" ← asks everyone who tries this feature

There's an Enterprise Edition feature to match queries with the view automatically

- Watch your query plans closely– it may not use the view as often as you want
- Even in EE, you may often need to reference the view by name and add a NOEXPAND hint

Every indexed view (and nonclustered index you add to an indexed view) adds overhead:

- More writes for each insert/update/delete
- More to check for corruption

Indexed views have overhead.

Just like regular nonclustered indexes, they slow down DUI operations.

If the view spans multiple tables, then data modification causes serializable range locks, too:

https://www.brentozar.com/archive/2018/09/locks-taken-during-indexed-view-modifications/

Indexed view corruption

Indexed view corruption

This is the worst kind of corruption:
you can still query it, but it returns null when there's really valid data

To detect the corruption, you have to run DBCC CHECKDB with EXTENDED_LOGICAL_CHECKS for compatibility level 110 and higher

One clue that you might be at risk: if there's not an obvious set of columns that makes a unique clustering key, such as a set of GROUP BY columns, the view might be at risk

Read more in Paul White's post:
http://sqlperformance.com/2015/04/sql-indexes/an-indexed-view-bug-with-scalar-aggregates

3.1 p41

Say we have a legacy app.

It doesn't trust the contents of the data.

It LTRIM/RTRIMs everything.

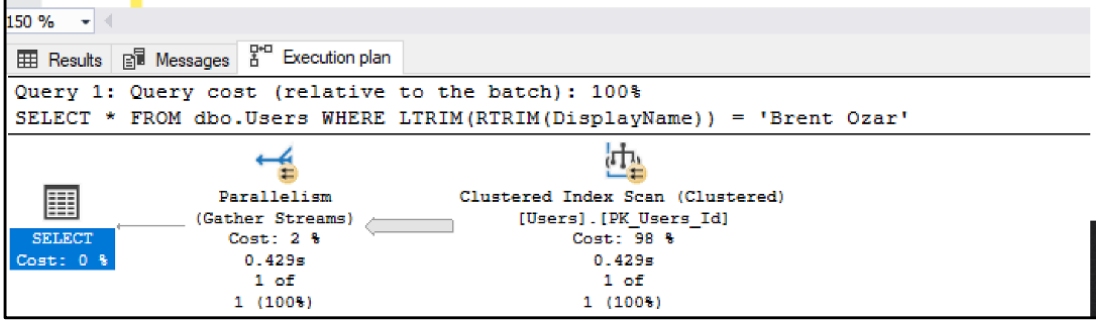
We can't change the code.

```
200 SELECT *
201   FROM dbo.Users
202   WHERE LTRIM(RTRIM(DisplayName)) = 'Brent Ozar';
203 GO
204
205 /* Will it use an index on DisplayName? */
206 CREATE INDEX IX_DisplayName ON dbo.Users(DisplayName);
207 GO
```

3.1 p43

Diabolical.

```
200  SELECT *
201    FROM dbo.Users
202    WHERE LTRIM(RTRIM(DisplayName)) = 'Brent Ozar';
203  GO
204
205  /* Will it use an index on DisplayName? */
206  CREATE INDEX IX_DisplayName ON dbo.Users(DisplayName);
207  GO
```

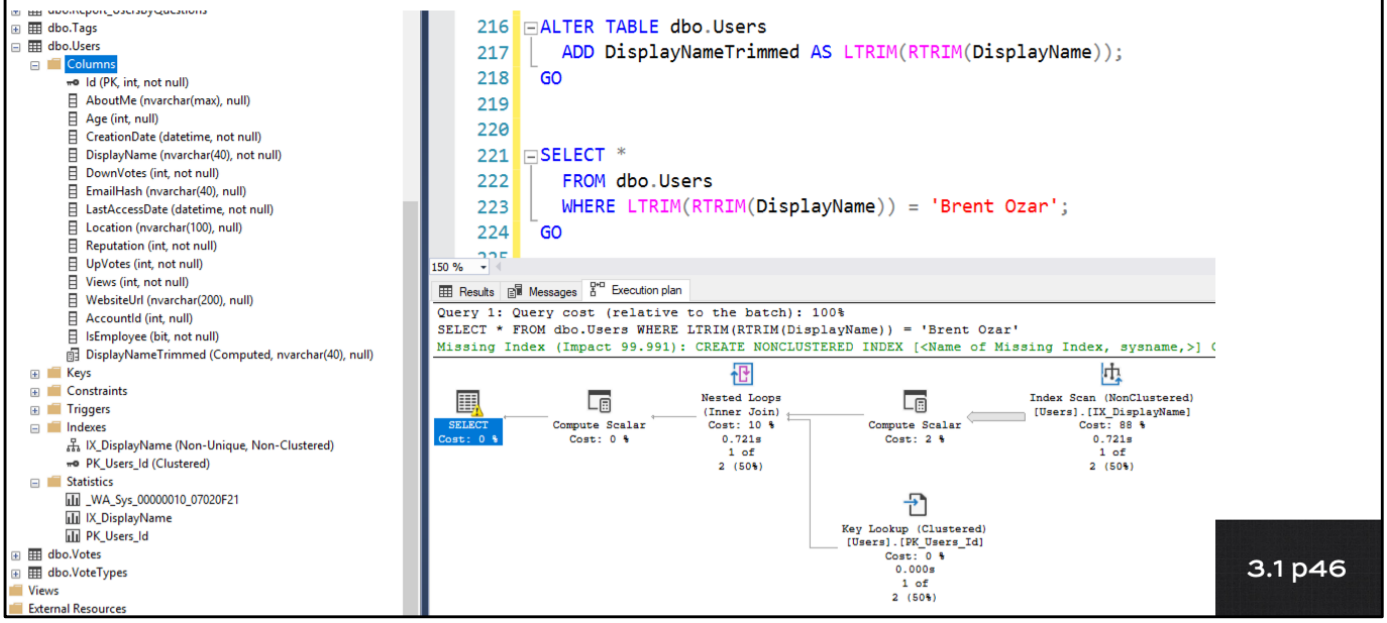


3.1 p44

The 3 problems

1. Our estimates may be way off
2. We're ignoring nonclustered indexes
3. We're not getting index seeks

Step 1: add a computed column.



3.1 p46

There's a lot to take in here.

This really does add a new column to the table.

I did NOT persist it for a bunch of reasons:

- It's a metadata-only change
- It finishes nearly instantly with low blocking
- Doesn't need to rewrite all of the 8KB pages, so no big logging problems either
- I didn't *need* to persist it: suddenly the query scans the index!

We fixed problem #1 and #2

1. Our estimates may be way off
2. We're ignoring nonclustered indexes
3. We're not getting index seeks

If we want to fix #3, we need an index on the new computed column.

Index the computed field.

Create a nonclustered index on it:

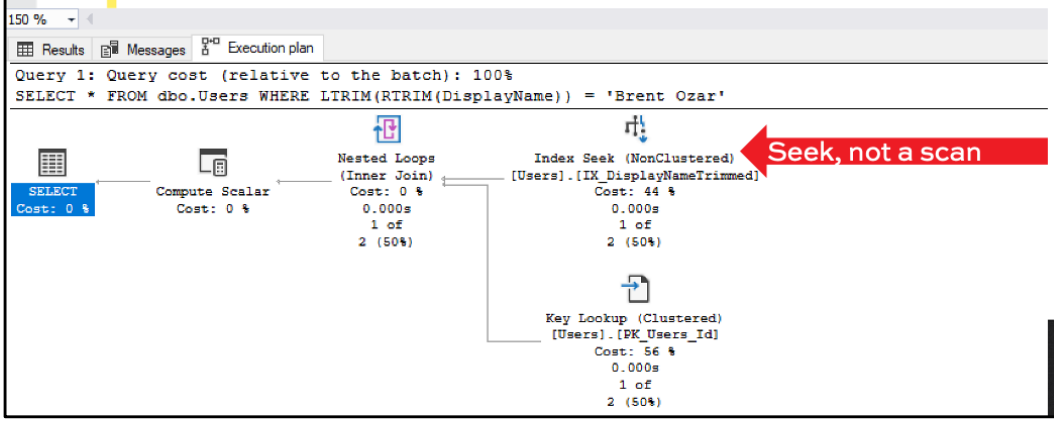
```
CREATE INDEX IX_DisplayNameTrimmed
  ON dbo.Users(DisplayNameTrimmed);
```

This persists the computed field's data in the index, but not in the clustered index of the table.

This isn't a metadata-only operation: we're actually creating data pages.

Pardon my French, but voila!

```
227  CREATE INDEX IX_DisplayNameTrimmed ON dbo.Users(DisplayNameTrimmed);
228  GO
229  SELECT *
230    FROM dbo.Users
231    WHERE LTRIM(RTRIM(DisplayName)) = 'Brent Ozar';
232  GO
233
```



3.1 p50

Fixing non-sargable queries

	Better estimates	Smaller scans	Index seeks
Add an index on the filtered field	No	Sometimes	No
Add a computed column closely matching the query's filter	Yes	Sometimes	Sometimes
Index the computed column	Yes	Usually	Usually

I can't just say "yes always" because queries are complicated: for example, if you're doing a SELECT * and getting >5% of the rows, you're likely going to end up doing scans to avoid key lookups.

Here's the magic

Queries can pick up the indexed computed column without directly referring to it by name

- SQL Server matches the computation in the query to the computation in the computed column
- Then it can use the index on it

Computed columns can also generate statistics that improve estimates and get better plans

This allow you to optimize some queries without changing their syntax!

Indexed computed column perks

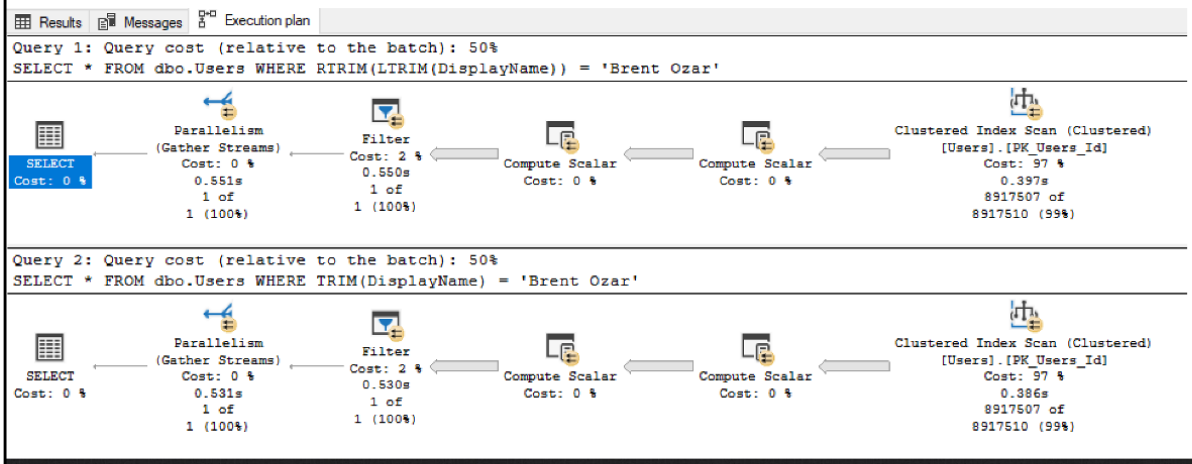
Works in Standard Edition

Works in SQL Server 2005+

Can optimize code without requiring code changes in the app

Drawbacks: it's gotta match

If I change it to RTRIM/LTRIM, or TRIM, no go:



3.1 p54

Classic scenario: data warehouse

Every morning, the ETL process:

- Loads yesterday's sales, all at once
  (and there is absolutely no overlap between days)
- Deletes all sales older than 3 years, all at once

Users want to be able to keep querying while loads happen, and absolutely will not tolerate seeing partial table loads as data comes in

Sliding window loads for dbo.Sales

Every day, we load last night's sales. Today, let's say we're adding 2018-10-01's sales.

And every day, we delete sales older than X. Today, let's say we're deleting 2015-10-01's sales.

SaleDate	Product	QtySold
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10
2018-09-28	Red Wine	4
2018-09-28	Rosé Wine	1
2018-09-28	White Wine	9
...		
2015-10-02	White Wine	5
2015-10-01	Red Wine	11
2015-10-01	White Wine	9

3.1 p57

Create a new partition for 2018-10-01

SaleDate	Product	QtySold
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10
2018-09-28	Red Wine	4
2018-09-28	Rosé Wine	1
2018-09-28	White Wine	9
…		
2015-10-02	White Wine	5
2015-10-01	Red Wine	11
2015-10-01	White Wine	9

SaleDate	Product	QtySold

Create a new table with exactly the same structure as our sales table, and load 2018-10-01's data in here. While we work, people can still query the dbo.Sales table.

3.1 p58

Switching the new partition in

SaleDate	Product	QtySold
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10
2018-09-28	Red Wine	4
2018-09-28	Rosé Wine	1
2018-09-28	White Wine	9
…		
2015-10-02	White Wine	5
2015-10-01	Red Wine	11
2015-10-01	White Wine	9

SaleDate	Product	QtySold
2018-10-01	Red Wine	11
2018-10-01	White Wine	8

When the new day's table is fully populated and ready to go, we can use ALTER TABLE SWITCH PARTITION to swap it into the main table…

3.1 p59

Switching the new partition in

Presto! The new data slides into dbo.Sales instantly, with very minimal locking, because dbo.Sales isn't really one table. It's a bunch of tables smashed together.

SaleDate	Product	QtySold
2018-10-01	Red Wine	11
2018-10-01	White Wine	8
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10
2018-09-28	Red Wine	4
2018-09-28	Rosé Wine	1
2018-09-28	White Wine	9
...		
2015-10-02	White Wine	5
2015-10-01	Red Wine	11
2015-10-01	White Wine	9

SaleDate	Product	QtySold

We also end up with an empty table swapped out from the sales table. Long story.

3.1 p60

Deleting the old data

SaleDate	Product	QtySold
2018-10-01	Red Wine	11
2018-10-01	White Wine	8
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10
2018-09-28	Red Wine	4
2018-09-28	Rosé Wine	1
2018-09-28	White Wine	9
...		
2015-10-02	White Wine	5
2015-10-01	Red Wine	11
2015-10-01	White Wine	9

And when we need to delete the oldest date – as long as it's all in one single partition, we can use that same ALTER TABLE PARTITION SWITCH trick...

Except now, we're switching it out with our prepped empty table, with no rows loaded into it.

SaleDate	Product	QtySold

3.1 p61

Deleting the old data

SaleDate	Product	QtySold
2018-10-01	Red Wine	11
2018-10-01	White Wine	8
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10
2018-09-28	Red Wine	4
2018-09-28	Rosé Wine	1
2018-09-28	White Wine	9
…		
2015-10-02	White Wine	5

So when we do the partition switch, we drop out all the old data fast, with minimal blocking.

SaleDate	Product	QtySold
2015-10-01	Red Wine	11
2015-10-01	White Wine	9

3.1 p62

Partitioned tables work best when

Your partitions line up neatly, like:

- Dates (can be day, week, month, quarter)
- Regions or states
- Product numbers

SaleDate	Product	QtySold
2018-10-01	Red Wine	11
2018-10-01	White Wine	8
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10

And when you follow these rules:

- You load an entire partition all at once, ***no exceptions***
- Your indexes need to start with SaleDate (I'm simplifying this a little)
- Your queries need to have the partitioning key in the where clause

3.1 p63

Partition elimination

If your query has the partitioning key in the WHERE clause, like this:

```
SELECT *
FROM dbo.Sales
WHERE SaleDate = '2018-09-30';
```

SaleDate	Product	QtySold
2018-10-01	Red Wine	11
2018-10-01	White Wine	8
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10

Then SQL Server can do partition elimination:
jump to one or more partitions, the fewer the better.

3.1 p64

If you don't...

If you don't have the key in the WHERE, SQL Server can't eliminate partitions:

```
SELECT *
FROM dbo.Sales
WHERE Product = 'Red Wine';
```

SaleDate	Product	QtySold
2018-10-01	Red Wine	11
2018-10-01	White Wine	8
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10

Then you're going to read data in all of the partitions, and your query can even perform *worse* as data is sorted/merged together for subsequent join operations in your query. (Assuming multi-table queries.)

3.1 p65

Partitioning is not a performance feature.

It's not designed to make your SELECT queries faster.

It's a maintenance feature
designed to make your loading and deleting faster.

Partitioning only makes sense when you're
loading/dropping an entire partition at a time.

Otherwise, think about it...

To make queries go faster...

And if you can guarantee that your queries have the partitioning key in the WHERE:

```
SELECT *
FROM dbo.Sales
WHERE SaleDate = '2018-09-30';
```

SaleDate	Product	QtySold
2018-10-01	Red Wine	11
2018-10-01	White Wine	8
2018-09-30	Red Wine	5
2018-09-30	White Wine	4
2018-09-29	Red Wine	7
2018-09-29	White Wine	10

Then you don't need a partitioned table.

You just need an index that starts with SaleDate (and then has other supporting fields as well, like we've talked about all week.)

Recap

Look for these app symptoms

1. App frequently queries a small subset of values in a column in a large table: Filtered indexes
2. App frequently queries an aggregation on low or moderate write tables: Indexed views
3. App frequently queries computations (and the app can't just write the computed value in the first place): Indexed computed columns
4. App loads a big chunk of data all at once, then deletes the oldest chunk of data all at once: Partitioned tables

Don’t overuse these.

90% of the time, you just need plain ol’ clustered & nonclustered indexes.

9% of the time, getting creative with the 5 & 5 guideline is enough.

This session is about the 1%.