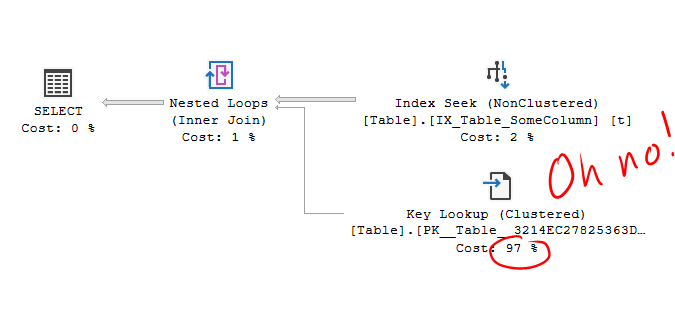
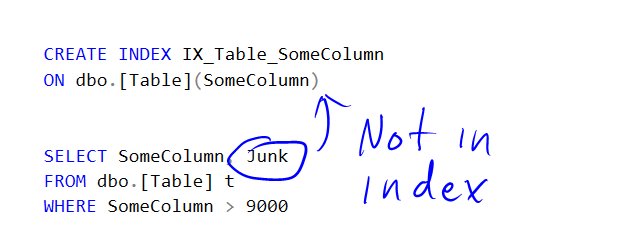
**[Forrest Shares Stuff](https://forrestmcdaniel.com/)**

Doing Horrible Things to SQL Server for Fun and Profit

**An Empirical Look at Key Lookups**

**FROM:** [**https://forrestmcdaniel.com/2020/12/30/an-empirical-look-at-key-lookups/**](https://forrestmcdaniel.com/2020/12/30/an-empirical-look-at-key-lookups/)

I’m going to presume you’re familiar with what key lookups are, and why they exist.



SQL Server’s cost optimizer hates them, because it thinks they’re random reads from a spinny disk – unlikely for a modern enterprise server. What I’m curious about is just how bad these assumptions are in practice.

**The Setup**

Here’s the million-row test table I’m going to use, along with fake data. I’m going to be testing on my laptop with a standard SSD, and an intel processor with a variable clock speed (it seems to average a bit below 4GHz for these tests).

CREATE TABLE dbo.InternetRecipes (

ID INT IDENTITY PRIMARY KEY,

Calories INT,

SizeOfIntro INT,

RecipeName NVARCHAR(300)

)

INSERT dbo.InternetRecipes

SELECT TOP (1000000)

(ROW\_NUMBER() OVER(ORDER BY 1/0))%1000,

(ROW\_NUMBER() OVER(ORDER BY 1/0))%10000+500,

'Paleo Fat Free Cotton Candy'

FROM master..spt\_values a,

master..spt\_values b

--With includes and not, for testing

CREATE INDEX IX\_InternetRecipes\_Calories

ON dbo.InternetRecipes(Calories)

CREATE INDEX IX\_InternetRecipes\_Calories\_incl

ON dbo.InternetRecipes(Calories)

INCLUDE (RecipeName)

**Time/Cost Comparison**

Using an index hint, we can see a direct comparison between a plan with key lookups, and without. The “without” plan makes use of the index that includes RecipeName. Normally this is described as a covering index, but today I’m designating it the “inclusive” index.

--7.7 query bucks

SELECT RecipeName

FROM dbo.InternetRecipes WITH(INDEX(IX\_InternetRecipes\_Calories\_incl))

WHERE Calories >= 0

AND RecipeName = 'Vegan Steak'

--200 query bucks

SELECT RecipeName

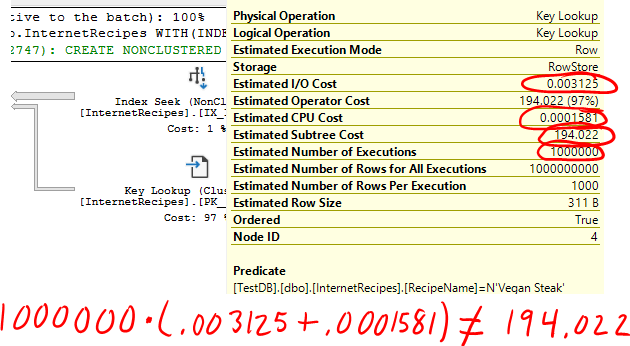
FROM dbo.InternetRecipes WITH(INDEX(IX\_InternetRecipes\_Calories))

WHERE Calories >= 0

AND RecipeName = 'Vegan Steak'

OPTION(MAXDOP 1)

Some interesting details emerge here. First: **key lookup cost does not scale linearly.**



Working the math backwards shows the cost comes from a million CPU costs, but only 11,495 IO costs. Incidentally, 11,495 is the number of data pages for the table. Key Lookup IO costs are capped at the number of pages in the table, although you’re unlikely to encounter this naturally. Of course, the next thing for me to test was everything in-between, and the experiments showed that the key lookup cost does not assume each will cause an IO.

The exact formula still eludes me though. I think it’s a delightfully nerdy problem that nobody will care about, so I’ll be working on it as soon as possible.

The one million key lookup query is costed about 26X more than the simple seek, and takes about 18X longer to run. The optimizer does a decent job here! Or does it? I was running the test with a warm cache, so what if we ignore IO costs?

--MS will laugh at you if this breaks your server

DBCC SETIOWEIGHT(0)

The optimizer cost ratio balloons to 149X (while actual runtime remains 18X of course). Even though the actual executions involved no IO, the query optimizer including IO costs does a better job.

Let’s fix that IO costing thing…

DBCC SETIOWEIGHT(1)

And test with a cold cache…

CHECKPOINT --if we just created the table/indexes

DBCC DROPCLEANBUFFERS --repeat as needed

The execution time ratio is about 15X with IO – not much of a change. If I disable [prefetch](https://www.sql.kiwi/2013/08/sql-server-internals-nested-loops-prefetching.html) with [9115](https://www.sql.kiwi/2013/01/optimizing-t-sql-queries-that-change-data.html) (thanks Paul), the time ratio moves to 17X, and query buck costs stay unchanged. So, **the optimizer does a decent job costing between the inclusive index and the key lookup plan, but only when it assumes IO costs**.

**Tipping Point Testing**

Because SQL Server loathes key lookups so much, it will sometimes choose to scan the entire table just to avoid them. In my experience, it does this way too early. Can I prove it?

Let’s remove the inclusive index, and select results into a trash variable for testing.

DROP INDEX IX\_InternetRecipes\_Calories\_incl

ON dbo.InternetRecipes

SELECT RecipeName

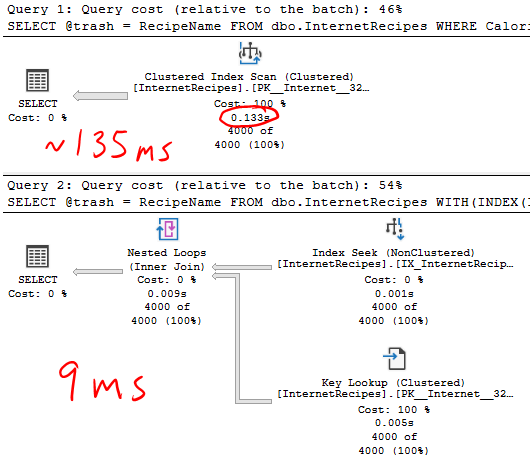
FROM dbo.InternetRecipes

WHERE Calories >= 996

OPTION(MAXDOP 1);

I find that the tipping point occurs between Calories values of 996 and 997, which is between 3000 and 4000 rows/lookups.

The full scan plan takes on average 135ms, while the key lookup plan for 4000 rows is only 9ms.



It’s not until Calories value 925, or 75k rows, that the true tipping point occurs, and the key lookup plan takes as long as the scan. Even testing with a cold cache, the tipping point is at 41k rows. So for my laptop with a warm cache, **the optimizer switches to a scan at just 4% of the optimum number of rows**, or 25X too soon (roughly).

**Throughput Testing**

What else can we learn about key lookup impact? Let’s cram it in a loop and set spin cycle to ultra high.

First, bring back the inclusive index.

CREATE INDEX IX\_InternetRecipes\_Calories\_incl

ON dbo.InternetRecipes(Calories)

INCLUDE (RecipeName)

Now, the general loop structure:

DECLARE @i INT = 0

DECLARE @trash NVARCHAR(300)

DECLARE @dt DATETIME2 = SYSDATETIME()

WHILE @i < 10000

BEGIN

SELECT TOP(1000) @trash = RecipeName

FROM dbo.InternetRecipes WITH(INDEX(IX\_InternetRecipes\_Calories))

WHERE Calories = 666 --Devil's food cake

OPTION(MAXDOP 1);

SET @i += 1

END

SELECT DATEDIFF(MILLISECOND,@dt,SYSDATETIME())

Using TOP to limit the number of rows/lookups, I measure how long it takes to run a loop of 10k queries. Here are my averaged numbers:

| **Rows** | **Key Lookup Time (ms)** | **Inclusive Seek Time (ms)** |
| --- | --- | --- |
| 1000 | 18,223 | 1,747 |
| 100 | 1,857 | 331 |
| 10 | 391 | 200 |
| 1 | 250 | 181 |
| 1k, random | 21,284 | 1,824 |

These numbers are specific to my laptop and particular test. Still, what I see is that **the impact of key lookups becomes larger the more there are**. This makes sense, as there is some overhead per query.

If I assume all extra time in the 1000 row test comes from the lookups, I get a value of **about 1.6 microseconds per lookup**. To put that in perspective, to occupy **an extra core of CPU would take 600,000 lookups/second**.

I was curious if using a repeated value took unfair advantage of the CPU cache, so I also tested with randomized “Calories” values, showing minor slowdown. An extra core under these conditions (probably more representative of a production server) would need **514k lookups/s**. And a big caveat: all these tests use a single table design, and are run on a personal laptop.

**Summary**

Lookups hurt performance, but probably not as much as you think, and definitely not as much as the optimizer thinks when considering the seek/scan tipping point. If you see a scary cost percentage number by a key lookup, remember that it’s probably [junk](https://forrestmcdaniel.com/2019/05/03/percentage-non-grata/).