**[GUIDs as PRIMARY KEYs and/or the clustering key](http://www.sqlskills.com/blogs/kimberly/guids-as-primary-keys-andor-the-clustering-key/)**

By: [Kimberly Tripp](http://sqlskills.com/blogs/kimberly)

Posted on: March 5, 2009 1:06 pm

(Be sure to [join our community](http://www.sqlskills.com/joincommunity.asp) to get our monthly newsletter with exclusive content, advance notice of classes with discount codes, and other SQL Server goodies!)

Expanding on the topic of “are you kidding me”… one of the MOST PREVALENT problems I see today is the dreaded “GUIDs as PKs” problem. However, just to be clear, it’s not [*as much of a*] problem that your PRIMARY KEY is a GUID as much as it is a problem that the PRIMARY KEY is probably your clustering key. They really are two things BUT the default behavior in SQL Server is that a PRIMARY KEY uses a UNIQUE CLUSTERED INDEX to enforce entity integrity. So, I thought I’d take this post to really dive into why this is a problem and how you can hope to minimize it.

**Relational Concepts – What is a PRIMARY KEY?**(*quick and basic reminder for what is what and why*)

Starting at the very beginning… a primary key is used to enforce entity integrity. Entity integrity is the very basic concept that every row is uniquely identifiable. This is especially important in a normalized database because you usually end up with many tables and a need to reference rows across those tables (i.e. relationships). Relational theory says that every table MUST have a primary key. SQL Server does not have this requirement. However, many features – like replication – often have a requirement on a primary key so that they can guarantee which row to modify on a related database/server (like the subscriber in a replication environment). So, most people think to create one. However, not always…

**What happens when a column(s) is defined as a PRIMARY KEY – in SQL Server?**

The first thing that SQL Server checks is that ALL of the columns that make up the PRIMARY KEY constraint do not all NULLs. This is a requirement of a PRIMARY KEY but not a requirement of a UNIQUE KEY. They also check to make sure (if the table has data) that the existing data meets the uniqueness requirement. If there are any duplicate rows, the addition of the constraint will fail. And, to check this as well as to enforce this for [future] new rows – SQL Server builds a UNIQUE index. More specifically, if you don’t specify index type when adding the constraint, SQL Server makes the index a UNIQUE CLUSTERED index. So, why is that interesting…

**What is a clustered index?**

In SQL Server 7.0 and higher the internal dependencies on the clustering key CHANGED. (Yes, it’s important to know that things CHANGED in 7.0… why? Because there are still some folks out there that don’t realize how RADICAL of a change occurred in the internals (wrt to the clustering key) in SQL Server 7.0). It’s always (in all releases of SQL Server) been true that the clustered index defines the order of the data in the table itself (yes, the data of the table becomes the leaf level of the clustered index) and, it’s always been a [potential] source of fragmentation. That’s really not new. Although it does seem like it’s more of a hot topic in recent releases but that may solely because there are more and more databases out there in general AND they’ve gotten bigger and bigger… and you feel the effects of fragmentation more when databases get really large.

What changed is that the clustering key gets used as the “lookup” value from the nonclustered indexes. Prior to SQL Server 7.0, SQL Server used a volatile RID structure. This was problematic because as records moved, ALL of the nonclustered indexes would need to get updated. Imagine a page that “splits” where half of the records are relocated to a new page. If that page has 20 rows then 10 rows have new RIDs – that means that 10 rows in EACH (and ALL) of your nonclustered indexes would need to get updated. The more nonclustered indexes you had, the worse it got (this is also where the idea that nonclustered indexes are TERRIBLY expensive comes from). In 7.0, the negative affects of record relocation were addressed in BOTH clustered tables and heaps. In heaps they chose to use forwarding pointers. The idea is that the row’s FIXED RID is defined at insert and even if the data for the row has to relocate because the row no longer fits on the original page – the rows RID does not change. Instead, SQL Server just uses a forwarding pointer to make one extra hop (never more) to get to the data. In a clustered table, SQL Server uses the clustering key to lookup the data. As a result, this puts some strain on the clustering key that was never there before. It should be narrow (otherwise it can make the nonclustered indexes UNNECESSARILY wide). The clustering key should be UNIQUE (otherwise the nonclustered indexes wouldn’t know “which” row to lookup – and, if the clustering key is not defined as unique then SQL Server will internally add a 4-byte uniquifier to each duplicate key value… this wastes time and space – both in the base table AND the nonclustered indexes). And, the clustering key should be STATIC (otherwise it will be costly to update because the clustering key is duplicated in ALL nonclustered indexes).

In summary, the clustering key really has all of these purposes:

1. It defines the lookup value used by the nonclustered indexes (should be unique, narrow and static)
2. It defines the table’s order (physically at creation and logically maintained through a linked list after that) – so we need to be careful of fragmentation
3. It can be used to answer a query (either as a table scan – or, if the query wants a subset of data (a range query) and the clustering key supports that range, then yes, the clustering key can be used to reduce the cost of the scan (it can seek with a partial scan)

However, the first two are the two that I think about the **most**when I choose a clustering key. The third is just one that I \*might\* be able to leverage if my clustering key also happens to be good for that. So, some examples of GOOD clustering keys are:

* An identity column
* A composite key of date and identity – in that order (date, identity)
* A pseudo sequential GUID (using the NEWSEQUENTIALID() function in SQL Server OR a “homegrown” function that builds sequential GUIDs – like Gert’s “built originally to use in SQL 2000″ xp\_GUID here: [~~http://sqldev.net/xp/xpguid.htm~~](http://sqldev.net/xp/xpguid.htm)

But, a GUID that is not sequential – like one that has it’s values generated in the client (using .NET) OR generated by the newid() function (in SQL Server) can be a **horribly bad choice** – *primarily*because of the fragmentation that it creates in the base table but also because of its size. It’s *unnecessarily*wide (it’s 4 times wider than an int-based identity – which can give you 2 billion (really, 4 billion) unique rows). And, if you need more than 2 billion you can always go with a bigint (8-byte int) and get 263-1 rows. And, if you don’t really think that 12 bytes wider (or 8 bytes wider) is a big deal – estimate how much this costs on a bigger table and one with a few indexes…

* Base Table with 1,000,000 rows (3.8MB vs. 15.26MB)
* 6 nonclustered indexes (22.89MB vs. 91.55MB)

So, we’re looking at 25MB vs 106MB – and, just to be clear, this is JUST for 1 million rows and this is really JUST overhead. If you create an even wider clustering key (something horrible like LastName, FirstName, MiddlieInitial – which let’s say is 64bytes then you’re looking at 427.25MB \*just\* in overhead….. And, then think about how bad that gets with 10 million rows and 6 nonclustered indexes – yes, you’d be wasting over 4GB with a key like that.

And, fragmentation costs you even more in wasted space and time because of splitting. Paul’s covered A LOT about fragmentation on his blog so I’ll skip that discussion for now BUT if your clustering key is prone to fragmentation then you NEED a solid maintenance plan – and this has it’s own costs (and potential for downtime).

So…………… choosing a GOOD clustering key EARLY is very important!

Otherwise, the problems can start piling up!

kt