

Re: Insert with response

Source:

<http://www.tech-archive.net/Archive/SQL-Server/microsoft.public.sqlserver.programming/2004-11/1869.html>

From: Hugo Kornelis (*hugo_at_pe_NO_rFact.in_SPAM_fo*)

Date: 11/07/04

Date: Sun, 07 Nov 2004 01:04:38 +0100

On Sat, 06 Nov 2004 14:20:16 -0800, Joe Celko wrote:

>>> *An IDENTITY can be great as an additional, alternate key[sic].*
> *Suppose you have a table whose natural key is composed of four columns,*
> . <<
>
> *Why did you call it a key when ****by definition****, it is not?*

Hi Joe,

I call any column a key that is declared as PRIMARY KEY – after all, that ***IS*** the definition of the PRIMARY KEY constraint, isn't it?

If you want to pick fleas, I'll gladly admit that I actually should have written "A column with the identity property can be great as an additional alternate key".

> *Do you*
> *like the accurate term "exposed non-relational physical locator"*
> *instead? :)*

Since an identity value has no relation whatsoever with the physical location of the data, I don't particularly like this term instead. How about "automatically generated increasing value", if you really must have a more accurate term?

> *Actually, IDENTITY is a horrible choice, There is no way to verify it,*
> *to port it or reference it safely.*

Did you really read my post? I did not advocate using identity as the ONLY key, but as an ADDITIONAL key, while still keeping a UNIQUE constraint on the column(s) that make up the natural key. You use the natural key to verify it and to reference it in the outside world; the identity surrogate key is solely used inside the database.

> *Since there is no way to relate*
> *these two "keys", which of these two tables is correct? That is, which*

>one has BOTH keys referencing the proper entity, as it exists in the
>real world? How did you put them in synch? if you cannot do that, you
>have no data integrity.

```
>  
>CREATE TABLE FooBar  
>(id IDENTITY NOT NULL PRIMARY KEY,  
> real_key CHAR(5) NOT NULL UNIQUE,  
> ...;  
>  
>CREATE TABLE BarFoo  
>( ...  
> foo_key INTEGER NOT NULL  
> REFERENCES FooBar(id),  
> bar_key CHAR(5) NOT NULL  
> REFERENCES FooBar(real_key),  
> PRIMARY KEY (bar_key, foo_key),  
> ..);  
(snip sample data)
```

Ugh! Are these two columns meant to be a redundant implementation of only one relationship between these tables? If so, then you should hunt down the person who made this design and run him over on the parking lot, it'll probably be an even better contribution to database design quality in this world than all your books together! :-)

If these two columns depict two separate relationships between BarFoo and FooBar, there's no way and no need to put them in synch. Though it's rather odd to not use the surrogate identity key for both relationships, as that's exactly what it's for.

```
>>> Now, the other tables that have to refer to this table can use just  
>one integer column (referring to the identity column) instead of four  
>long character columns. The space for the original table will increase,  
>but the space for the other tables will decrease and joins will be  
>carried out faster – the only downside is that you'll need to use a join  
>more often, but in this particular case the pros will outweigh the cons.  
><<  
>  
>Have you actually measured the performance differences in an INTEGER  
>versus CHAR(n) columns in joins? The usual answer is that nobody has  
>and the myth continues among lazy, sequential file programmer who do not  
>>want to learn RDBMS.
```

Have _you_?

Run the below scripts in SQL Server (apologies for not capitalizing the keywords as I normally do – it's late and I need to get some sleep, but I wanted to reply to you first).

The first script creates an n:m relationship using an integer column, the second script creates the same relationship using a char(300) column. The

test data generation uses a number table to generate 1000 rows in the base tables, then I use a cross join and the modulo operator to relate each column in the foo table to 250 columns in the bar table. After clearing the procedure cache, flushing dirty pages to disk and clearing the disk cache, I execute a query to retrieve all details for one row in Foo plus all details for all related rows in Bar. I use select .. into temptable to make sure that I measure the performance of the query, not the speed of my network connection or of the program that has to display the result set.

```
-- Start of script #1 --
-- Create the tables
create table Foo (FooID int not null primary key,
                 FooName char(300) not null unique)
create table Bar (BarID int not null primary key,
                 BarName char(300) not null unique)
create table FooBar (FooID int not null references Foo,
                    BarID int not null references Bar,
                    primary key (FooID, BarID))

go
-- Generate the test data
insert Foo (FooID, FooName)
select n, str(n)
from numbers
where n between 1 and 1000
insert Bar (BarID, BarName)
select n, str(n)
from numbers
where n between 1 and 1000
insert FooBar (FooID, BarID)
select FooID, BarID
from Foo, Bar
where FooID % 4 = BarID % 4
go
-- Eliminate caching effects
dbcc freeproccache
checkpoint
dbcc dropcleanbuffers
-- Now run the test
set statistics io on
set statistics time on
select Foo.FooID, Foo.FooName, Bar.BarID, Bar.BarName
into temptable
from Foo
inner join FooBar
    on FooBar.FooID = Foo.FooID
inner join Bar
    on Bar.BarID = FooBar.BarID
where Foo.FooID = 17
set statistics time off
set statistics io off
go
```

```
-- Clean up
drop table temptable
drop table FooBar
drop table Foo
drop table Bar
go
-- End of script #1 --
```

The results from this script:

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

Table 'temptable'. Scan count 0, logical reads 1, physical reads 0, read-ahead reads 0.

Table 'Bar'. Scan count 250, logical reads 760, physical reads 0, read-ahead reads 41.

Table 'FooBar'. Scan count 1, logical reads 3, physical reads 3, read-ahead reads 0.

Table 'Foo'. Scan count 1, logical reads 2, physical reads 2, read-ahead reads 0.

SQL Server Execution Times:

CPU time = 31 ms, elapsed time = 219 ms.

```
-- Start of script #2 --
-- Create the tables
create table Foo (FooID int not null unique,
                 FooName char(300) not null primary key)
create table Bar (BarID int not null unique,
                 BarName char(300) not null primary key)
create table FooBar (FooName char(300) not null references Foo,
                    BarName char(300) not null references Bar,
                    primary key (FooName, BarName))
go
-- Generate the test data
insert Foo (FooID, FooName)
select n, str(n)
from numbers
where n between 1 and 1000
insert Bar (BarID, BarName)
select n, str(n)
from numbers
where n between 1 and 1000
insert FooBar (FooName, BarName)
select FooName, BarName
from Foo, Bar
where FooID % 4 = BarID % 4
go
-- Eliminate caching effects
dbcc freeproccache
checkpoint
dbcc dropcleanbuffers
```

```
-- Now run the test
set statistics io on
set statistics time on
select Foo.FooID, Foo.FooName, Bar.BarID, Bar.BarName
into temptable
from Foo
inner join FooBar
    on FooBar.FooName = Foo.FooName
inner join Bar
    on Bar.BarName = FooBar.BarName
where Foo.FooID = 17
set statistics time off
set statistics io off
go
-- Clean up
drop table temptable
drop table FooBar
drop table Foo
drop table Bar
go
-- End of script #2 --
```

The results from this script:

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.
Table 'temptable'. Scan count 0, logical reads 1, physical reads 0,
read-ahead reads 0.
Table 'Bar'. Scan count 250, logical reads 1311, physical reads 1,
read-ahead reads 43.
Table 'FooBar'. Scan count 1, logical reads 26, physical reads 6,
read-ahead reads 23.
Table 'Foo'. Scan count 1, logical reads 2, physical reads 2, read-ahead
reads 0.

SQL Server Execution Times:

CPU time = 94 ms, elapsed time = 514 ms.

As you can see, the second script took 200% more CPU time and 130% more elapsed time. I think this proves that using integer columns for joins can indeed enhance the performance.

*>Given hard disk access time, the slowest operation in the hardware by
>microseconds versus nanoseconds, what does the extra column do for table
>scans, the most common database operation?*

If you add an extra integer column (with IDENTITY property) to a table, you add 4 bytes per row. This will decrease the number of rows per page and will therefore increase the required number of page reads for a table scan. Unless the size of the row was already very short to begin with, this effect will only be marginal.

At the same time, you'll be able to replace one (or even several) much bigger columns in all tables that refer to this table by one integer column. For these referring tables, the number of bytes per row will be reduced by a significant amount; the number of rows per page will increase and a table scan will require less page reads.

To sum it up: if you have a table with a long natural key, adding an integer surrogate key with the identity property will have a small negative impact on the speed of table scans for that table and a big positive impact on the speed of table scans for all referring tables.

*>One of the other problems is that IDENTITY is *much* more subject to bad
>data entry. If I have a valid IDENTITY of 56743, then I probably also
>have a valid IDENTITY of 56734 in my data. I'd have to look it up
>again, but the error rate in typing an integer of 5 digits long is <2%
>with a skilled typist.*

But as the natural key should still be used for data entry, this is not a problem at all.

Best, Hugo

--

(Remove _NO_ and _SPAM_ to get my e-mail address)