Beware of Search Argument (SARG) Data Types

The other day a colleague and I were attempting to optimize a new medical insurance claim processing system. Performance is a major concern, and the goal was to be able to process 700 claims per minute. Our initial tests were only churning through about 50/min. The database is in SQL Server 2005 with Read Committed Snapshot Isolation enabled, this is a very cool feature that allows us to have very similar transactional behavior across both Oracle and SQL Server, but that's a topic for another day... The application is Java running on JBoss and it uses Hibernate as the Object Relational Mapper (ORM).

> *Just in case you are asking yourself "I wonder what Don thinks of ORM's?", I hate the stinking things because they force your data model to mirror the object model, effectively turning the database into a "tightly coupled" persistence mechanism for the application, when the application should be a "loosely coupled" presentation layer for the data. I have long advocated using stored procedures as the database API, but that too is a topic for another day.*

When looking at Profiler the vast majority of the statements had sub-second response times, but there were some that approached and sometimes exceeded the 1 second timeframe. 700 claims per minute may not sound like much but each claim can have dozens of claim lines, and each line can have several rules, each of which could have an override that may apply at some level of the enterprise hierarchy etc... each claim can result in hundreds of queries, inserts, and updates across dozens of tables. To reach our performance objectives the database queries must be extremely fast.

The Hibernate queries in Profiler look something like this:

```
declare @p1 int set @p1=26931 exec sp_prepexec @p1 output,N'@P0 decimal(38,0),@P1 nvarchar(4000),@P2
nvarchar(4000),@P3 decimal(38,0),@P4 datetime,@P5 datetime,@P6 decimal(38,0),@P7 datetime,@P8
datetime',N'select procedured0_.PROCEDURE_DIAGNOSIS_IID as PROCEDURE1_78_,
procedured0_.PROCEDURE_CODE as PROCEDURE2_78_, procedured0_.DIAGNOSIS_CODE as
DIAGNOSIS3_78_, procedured0_.EFFECTIVE_DATE as EFFECTIVE14_78_,
procedured0_.EXPIRATION_DATE as EXPIRATION15_78_ from
V_PROCEDURE_DIAGNOSIS_OVERRIDE procedured0_, ENTERPRISE_TREE enterprise1_ where
enterprise1_.DESCENDENT_IID=@P0 and procedured0_.ENTERPRISE_IID=enterprise1_.ANCESTOR_IID
and procedured0_.PROCEDURE_CODE=@P1 and procedured0_.DIAGNOSIS_CODE=@P2 and
(procedured0_.RULESET_IID=@P3 or procedured0_.RULESET_IID is null) and
procedured0_.EFFECTIVE_DATE<=@P4 and ((procedured0_.EXPIRATION_DATE is not null) and
procedured0_.EXPIRATION_DATE>=@P5 or procedured0_.EXPIRATION_DATE is null) and
(enterprise1_.SEPARATION in (select MIN(enterprise3_.SEPARATION) from
V_PROCEDURE_DIAGNOSIS_OVERRIDE procedured2_, ENTERPRISE_TREE enterprise3_ where
enterprise3_.DESCENDENT_IID=@P6 and
procedured0_.PROCEDURE_CODE=procedured2_.PROCEDURE_CODE and
procedured0_.DIAGNOSIS_CODE=procedured2_.DIAGNOSIS_CODE and
(procedured0_.RULESET_IID=procedured2_.RULESET_IID or (procedured0_.RULESET_IID is null) and
(procedured2_.RULESET_IID is null)) and procedured2_.ENTERPRISE_IID=enterprise3_.ANCESTOR_IID
and procedured2_.EFFECTIVE_DATE<=@P7 and ((procedured2_.EXPIRATION_DATE is not null) and
procedured2_.EXPIRATION_DATE>=@P8 or procedured2_.EXPIRATION_DATE is null))) order by
procedured0_.RULESET_IID DESC    ',5,N'92015',N'37950',1,'Aug  9 2001 12:00:00:000AM','Aug  9 2001
12:00:00:000AM',5,'Aug  9 2001 12:00:00:000AM','Aug  9 2001 2:00:00:000AM' select @p1
```

> *Combining Hibernate's penchant for giving everything a weird alias, total lack of formatting, and using the old join syntax with SQL Server's less than helpful way of showing parameter values, this is an unintelligible mess! If anyone on the SQL Server team is reading this, would it kill you to give us the query with the parameter values already substituted? Then we could take the query right out of Profiler and generate an execution plan without a bunch of manual cutting and pasting.*

Anyway, notice the second and third parameters are Procedure_Code and Diagnosis_Code respectively (highlighted in blue). You couldn't know this from the query but those columns in the database are defined as varchar data types. Note too that they are being passed in as Unicode character strings. I saw this right away, but maybe like you I just kind of said "So what?"

I always prefer explicit conversions rather than implicit, primarily because I want to see what's going on and I don't like to assume. But I really didn't see this as a problem. I "knew" that SQL Server would convert the SARG value to a varchar and then perform the seek. When I substituted the SARG values and generated an execution plan for the query I was surprised to see that it wasn't properly utilizing the clustered index. After chasing my tail for a while, and looking at the clustered index scan in the execution plan at least a half dozen times, a colleague asked if the "CONVERT_IMPLICIT" statement we saw in the Seek Predicate might be the problem. At first I dismissed the idea because, once again, I "knew" that SQL Server would do an implicit conversion of the Unicode string to varchar, so that statement was to be expected. Then I saw it...the CONVERT_IMPLICIT was on the wrong side of the equation! SQL wasn't converting the SARG, it was converting the column.

We all know that when writing a query you should manipulate the SARG not the data right? I.e.

```
SELECT * FROM Table1

WHERE DATEDIFF(yy, DateOfBirth, GETDATE()) > 21
```

should be rewritten to be

```
SELECT * FROM Table1

WHERE DateOfBirth < DATEADD(yy, -21, GETDATE())
```

The first query can't possibly use an index and performs the DATEDIFF on every row in the table. The second dynamically calculates the search argument one time and then compares the result with the table and can use an index if one exists.

The Index Scan's Seek Predicate for the ugly query above was:

```
CONVERT_IMPLICIT(nvarchar(7),[dbo].[PROCEDURE_DIAGNOSIS].[PROCEDURE_CODE],0)=N'92015'
AND CONVERT_IMPLICIT(nvarchar(7),[dbo].[PROCEDURE_DIAGNOSIS].[DIAGNOSIS_CODE],0)
=N'37950'
```

You can see that rather than convert the SARG, SQL Server decided to convert the table data. The Subtree cost was 25.5403 with an I/O Cost of 22.8809.

Simply removing the "N" from the Procedure_Code and Diagnosis_Code SARGs changed the Seek Predicate to:

```
[dbo].[PROCEDURE_DIAGNOSIS].PROCEDURE_CODE, [dbo].
[PROCEDURE_DIAGNOSIS].DIAGNOSIS_CODE = '92015', '37950'
```

This changed from an index scan to a seek and the Subtree Cost dropped to 0.0032831 with an I/O Cost of 0.003125.

This represents nearly an 8000% improvement.

Further testing confirmed this behavior in other situations too.

Testing this for yourself is simple. Just create the following table and populate it like so:

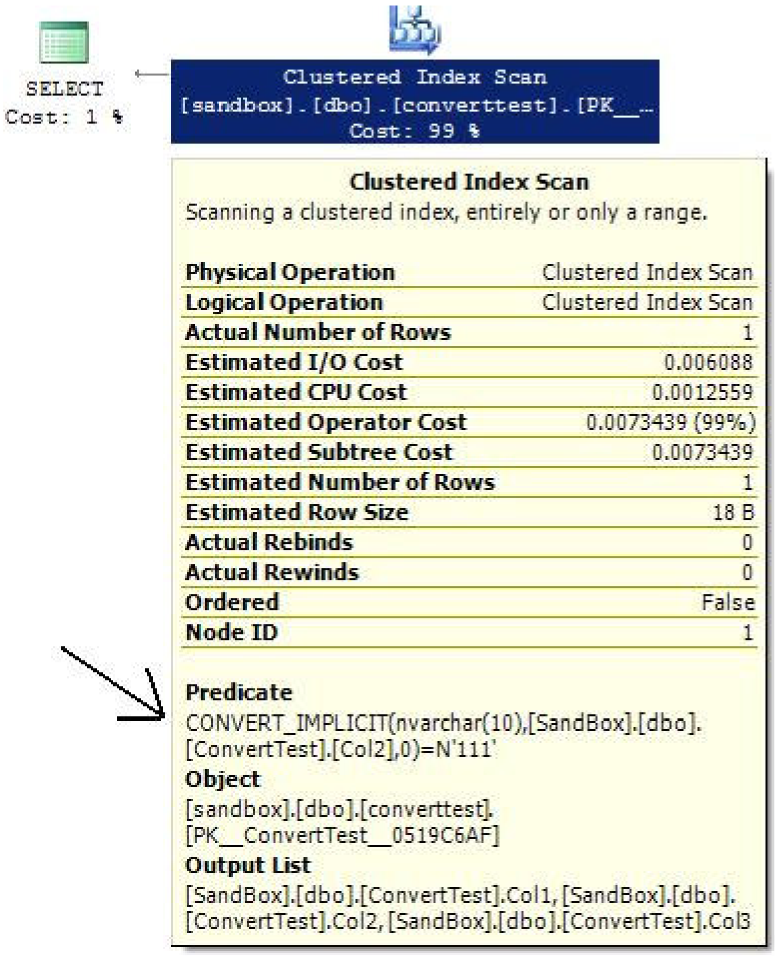
```
CREATE TABLE dbo.ConvertTest (Col1 int PRIMARY KEY, Col2 varchar(10))
CREATE INDEX IX1_ConvertTest ON dbo.ConvertTest (Col2)
DECLARE @Counter int
SET @Counter = 1
WHILE @Counter < 1000
BEGIN
```

```
        INSERT ConvertTest (Col1, Col2)
        VALUES (@Counter, CAST(@Counter AS varchar(10)))
        SET @Counter = @Counter + 1

    END
```

Now run the following query and look at the execution plan.

```
    SELECT * FROM dbo.ConvertTest WHERE Col2 = N'111'
```



Clustered Index Scan
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	1
Estimated I/O Cost	0.006088
Estimated CPU Cost	0.0012559
Estimated Operator Cost	0.0073439 (99%)
Estimated Subtree Cost	0.0073439
Estimated Number of Rows	1
Estimated Row Size	18 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

Predicate
CONVERT_IMPLICIT(nvarchar(10),[SandBox].[dbo].[ConvertTest].[Col2],0)=N'111'
Object
[sandbox].[dbo].[converttest].[PK__ConvertTest__0519C6AF]
Output List
[SandBox].[dbo].[ConvertTest].Col1, [SandBox].[dbo].[ConvertTest].Col2, [SandBox].[dbo].[ConvertTest].Col3

The execution plan shows a clustered index scan (effectively a table scan here) but removing the "N" magically turns the scan into a nonclustered index seek. If you actually step through this exercise you'll notice that the new and improved execution plan actually includes two index seeks and a merge join. This is because our non-clustered index is not a covering index so to actually get the data, SQL has to go to the clustered index (table). In this case the optimizer chooses to do a merge join rather than a bookmark lookup. Also, since our table is so small, the cost difference between the two queries is pretty small, but it serves to illustrate the point.

Just for kicks I decided to try a few more things to test the ins and outs of implicit conversion, so I ran the following query:

```
    SELECT * FROM dbo.ConvertTest WHERE Col1 = N'111'
```

When the target column is an integer SQL Server performs a convert on the SARG and an index seek. Now run the following query:

```
    SELECT * FROM dbo.ConvertTest WHERE Col1 =111
```

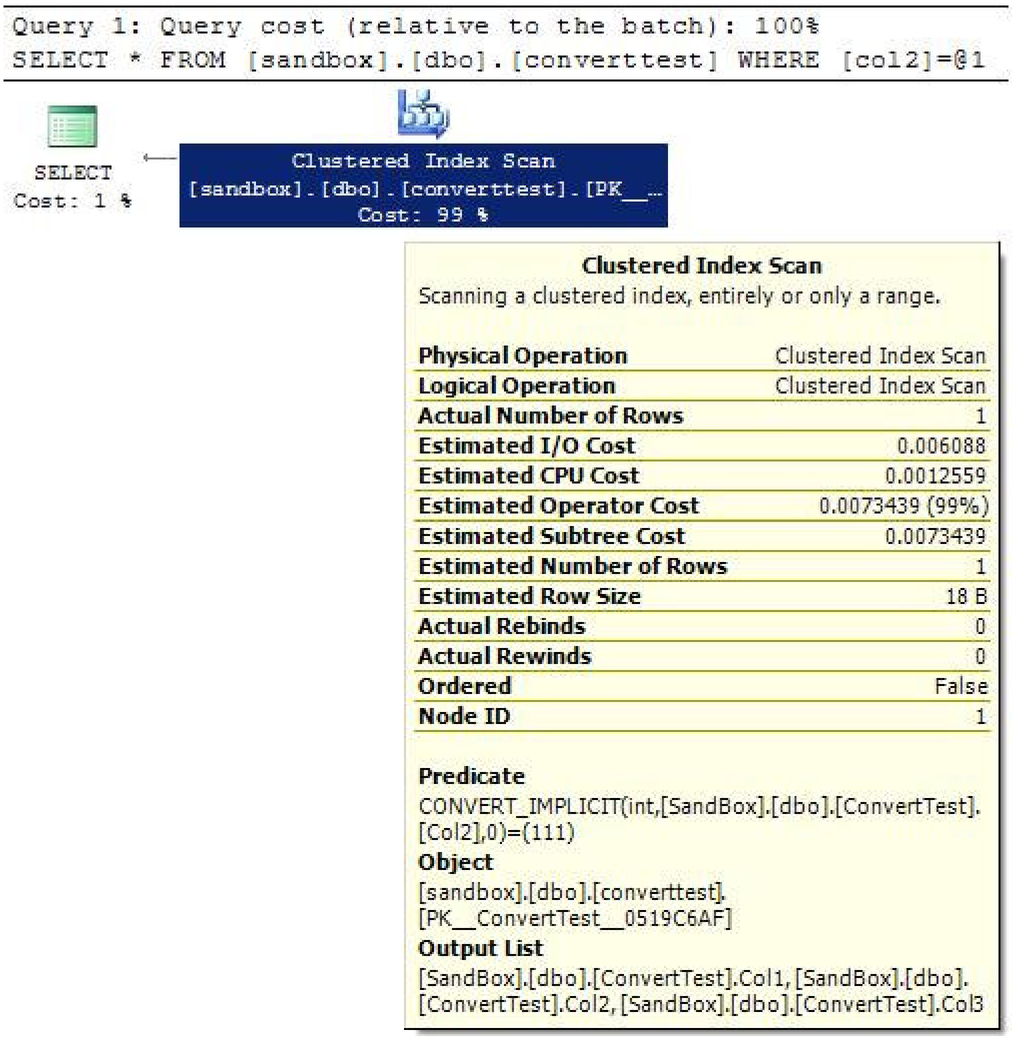
I'd never noticed before, and I'm not quite sure what to make of it, but SQL Server does an implicit conversion of the SARG even when the colum and SARG are both integers. This isn't a problem per se, but it is a bit surprising.

```
    SELECT * FROM dbo.ConvertTest WHERE col1 = 111222333444 --Bigint value
```

I actually expected to get a conversion error on this one, but the query runs without error. I assume that the optimizer is smart enough to know that it can't convert the bigint value to an int so it doesn't even try, however rather than just evaluating the expression to false and returning nothing, it still does an index seek even though there's no way that the query will ever return any records, go figure...

Naturally I couldn't leve it alone yet, so I decided to return to the varchar column and run the following query:

```
SELECT * FROM dbo.ConvertTest WHERE Col2 =111
```



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [sandbox].[dbo].[converttest] WHERE [col2]=@1
```

Clustered Index Scan
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	1
Estimated I/O Cost	0.006088
Estimated CPU Cost	0.0012559
Estimated Operator Cost	0.0073439 (99%)
Estimated Subtree Cost	0.0073439
Estimated Number of Rows	1
Estimated Row Size	18 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

Predicate
CONVERT_IMPLICIT(int,[SandBox].[dbo].[ConvertTest].[Col2],0)=(111)
Object
[sandbox].[dbo].[converttest].[PK__ConvertTest__0519C6AF]
Output List
[SandBox].[dbo].[ConvertTest].Col1, [SandBox].[dbo].[ConvertTest].Col2, [SandBox].[dbo].[ConvertTest].Col3

Now we're right back to doing the convert on the column rather than the SARG...WHAT?!?! This one blew me away, because it destroyed my pet theory on why SQL Server chooses one over the other to convert.

My original hypothesis was that this behavior is by design. Any non-Unicode string should be convertable into Unicode, the reverse is not true. Therefore in order to avoid conversion errors, the folks at MS decided to always convert the non-Unicode characters to Unicode wherever they are compared. However in the last query my hypothesis meets an untimely end, SQL Server attempts to convert the varchar column into an integer. It would be a much safer operation to convert the integer to varchar, afterall any number can be converted into a varchar (subject to length constraints of course) but the reverse is certainly not true. Furthermore in the case of the bigint to int conversion, SQL Server apparently "knows" it's impossible, so doesn't even try, even though it does do an implicit conversion from int to int, so why couldn't the optimizer do the same thing with Unicode to varchar and int to varchar? Go figure...

After looking through the MS Knowledgebase and several other search engines, I couldn't seem to come up with much of anything on this subject. Based on the lack of articles and fourum posts, it would appear that not many folks have run across this problem, or if they have, they just don't know it...or I am just totally clueless and everybody else knows something I don't. So I figured that this would be a good bit of information to share and maybe get some additional feedback and/or clarification. I also wonder if this might be at least part of the reason for the persistent myth that an index on an integer column is inherently more efficient than an index on a character column.

I finally did find the answer in the Data Type Precedence as defined in SQL Server Books Online. If you have BOL for SQL Server 2005 installed this link should take you to the article. Every data type is assigned a prececence and if you attempt to compare two different (but compatible) data types the one with the lower precedence will always be converted. As we have just seen varchar has a lower precedence than nvarchar and int. In hindsignt, this information has been around for some time, and I know I've seen that article before, but I certainly didn't fully appreciate all the implications. Finally, the whole thing makes sense; SQL Server follows a set of rules when it has to decide what to convert, I can understand that, but it sure would be nice if the performance implications were better known and publicized.

To finish my story... now I knew what the problem was, we needed to fix it, the options were to make all of our character columns to Unicode or to change JBoss to stop sending Unicode strings. Since Unicode is a waste of resources-unless you really need it-we opted for the latter. After digging around we finally managed to find the proper configuration file for JBoss and stopped it from passing character strings as Unicode. As if by magic, the database now purred like a finely tuned machine. The throughput on that server wound up jumping to nearly 800 claims per minute with that one change.

The moral of the story is that you need to be very careful with data types and implicit conversion. SQL Server doesn't always choose the "best" way of dealing with conversions, in some cases the difference is far more than just academic. I know I'll never just gloss over implicit conversions in the future.

By Don Peterson Thursday, June 22, 2006