What I Learned This Month: CCSID Confusion

Scott Chapman
American Electric Power

I have been working with a JDBC application that uses DB2 on z/OS for its data storage. While everything is currently performing well enough, we might have some issues to address if the data volume increases substantially. It is generally true that one shouldn't spend time optimizing prematurely, but it is also true that the earlier you can fix a problem in the development cycle, the cheaper the fix is. In this case, spending a little time now to investigate some options is a lot easier than if we have to come back later and re-architect things after the application has been in production for a while.

One primary area of interest has been the process that inserts millions of rows into dozens of tables each day. These tables are currently pretty simple: no indexes on the tables and the inserts happen at a time of day when we can lock the entire table, so there's no lock contention with other processes. Therefore, a lot of the normal insert performance bottlenecks simply aren't there. But still, I'd be more comfortable with our scalability prospects if the inserts were faster.

One thing that came to mind was that the application is all Java, but our default data storage on z/OS DB2 is EBCDIC. Internally, data in Java is always in Unicode and it translates to the platform default when doing I/O. (Although you can make it do I/O in any character set.) So the data exists as Unicode within Java, gets converted to EBCDIC to be stored in the tables, and then converted back to Unicode again when read from the table. It seemed that if we switched the data storage on z/OS to be Unicode, we might save a little bit of overhead.

So I asked my friendly DBA to clone one of the tables and make it Unicode instead of EBCDIC. She happily obliged and I ran a test case, inserting the same data into both the EBCDIC and Unicode version of the tables. We were stunned by the results: the DB2 DDF thread consumed about 50% *more* CPU when it was inserting to a Unicode table vs. the EBCDIC table! We expected the Unicode result to perhaps save a little CPU, not cost more! We went back and double-checked that the tables were really exactly the same except for the encoding. We ran the tests multiple times, sending the data from JVMs on other z/OS systems as well as Windows. The DDF threads' CPU consumption on z/OS showed relatively little variation between runs, and the difference between the Unicode and EBCDIC results remained consistent as well.

Stumped, we opened a problem with IBM who suggested that we alter the subtype of the character columns from "FOR SBCS[1] DATA" to "FOR MIXED DATA". I was surprised that the columns were set to SBCS instead of MIXED

[1] Single Byte Character Set, i.e. a character set where every character is represented by exactly one byte.

since the documentation indicates that “For Unicode data, the default subtype is MIXED” and that the default for EBCDIC is SBCS. What had happened was that to create the Unicode table, my DBA had not simply re-run the original "create table" statement against a new Unicode tablespace, she had used a tool to generate a new create table statement for her based on the existing table. The tool included values for every setting, including ones that originally were defaulted by DB2. So the generated statement included “FOR SBCS DATA” on all the columns, because that’s what it had defaulted to on the EBCDIC table. If she had used the original statement to define the new Unicode table, the columns would have defaulted to “FOR MIXED DATA”.

Once the columns were altered to the correct default, the CPU time on the inserts to the Unicode table dropped down to the same as the EBCDIC table. There was no performance gain by using Unicode, but at least the egregious performance penalty disappeared.

In looking at things more closely, I discovered that the CCSID[2] used for the columns varied thusly:

Table Encoding	Column sub-type	CCSID used
EBCDIC	SBCS	37 (EBCDIC)
Unicode	SBCS	367 (ASCII)
Unicode	Mixed	1208 (UTF8)

I’m not sure why the CCSID 367 (ASCII) added so much overhead vs. 1208 (UTF8) and 37 (EBCDIC), but it may have something to do with our Unicode Services[3] configuration. I looked into that only briefly because it seemed clear that changing the table encoding wasn’t going to substantially improve anything.

My original guess was that the difference would be negligible. That the test showed a significant degradation in the situation where ASCII data was involved implies that there are cases where the translation cost is significant, so maybe there are situations where it would be a performance benefit to store the data in one character encoding vs. another. However, the fact that CCSID 1208 worked as well as 37 implies to me that perhaps the cause of the performance degradation with CCSID 367 is that our Unicode Services is not configured to efficiently translate to that. So I guess there are a few lessons this month:

- I need to better understand how Unicode Services works in z/OS.
- Storing data in Unicode in DB2 to match Java accessing the data via JDBC is not necessarily a significant performance improvement.

[2] Coded Character Set Identifier, see http://en.wikipedia.org/wiki/CCSID for a more complete explanation.

[3] Unicode Services is the z/OS component that handles translations between CCSIDs.

- Sometimes our tools try to be too complete: copying all the attributes of an object when creating a new object may not be ideal.

As always, if you have questions or comments, you can reach me via email at sachapman@aep.com.