Datatypes: Worst Practices

Christian Antognini
Trivadis AG
Zürich, Switzerland

In recent years, I have witnessed a worrying trend in physical database design. This trend may be called *wrong datatype selection*. At first glance, choosing the datatype seems like a very straightforward decision to make. Nevertheless, in a world where software architects spend a lot of time discussing things like agile software development, SOA, AJAX or persistence frameworks, I am convinced it is essential to get back to the basics and understand why datatype selection is important. To illustrate this, I will present four examples of typical problems that I have encountered over and over again. Even though this may all seem very basic, don't underestimate the number of systems that are now running and suffering because of these problems.

The first problem caused by wrong datatype selection is wrong or lacking validation of data, when it is inserted or modified in the database. For example, if a column is supposed to store numeric values, choosing a character datatype for it calls for an external validation. In other words, the database engine is not able to validate the data. It leaves the application to do it. Even if such a validation is easy to implement, bear in mind that every time the very same piece of code is spread to several locations, sooner or later there will be a mismatch in functionality. The example I would like to bring forward is related to the initialization parameter NLS_NUMERIC_CHARACTERS. Remember that this initialization parameter specifies the characters used as decimals and group separators. For example, in Switzerland it is usually set to ".," and, therefore, the value Pi is formatted as follows: 3.14159. Instead, in Germany it is commonly set to ",." and, therefore, the same value is formatted as follows: 3,14159. Sooner or later, running an application with different client-side settings of this initialization parameter will run into a "ORA-01722: invalid number" if conversions take place because of the use of a wrong datatype in the database.

The second problem caused by wrong datatype selection is the loss of information. In other words, during the conversion of the original (correct) datatype to the database datatype, information gets lost. For example, let's imagine what happens when the date and time of an event is stored with a DATE datatype instead of a TIMESTAMP WITH TIME ZONE datatype. Fractional seconds and time zone information get lost. While the former leads to a very small error (less than 1 second), the latter might be a bigger problem. In one case I have witnessed personally, a customer's data was always generated using local standard time (without daylight-savings time adjustments) and stored directly in the database. The problems arose when, for reporting purposes, a correction for daylight-savings time had to be applied. A function designed to make a conversion between two time zones was implemented. Its signature was the following:
new_time_dst(in_date DATE, tz1 VARCHAR2, tz2 VARCHAR2) RETURN DATE
Calling such a function once was very fast. The problem was calling it thousands of times for each report. The response time increased by a factor of 25 as a result. Clearly, with the correct datatype, everything would not only be faster but also easier (the conversion would be performed automatically).

The third problem caused by wrong datatype selection is that things do not work as expected. Let's say you have to range partition a table, based on a column storing date and time information. This is usually no big deal. The problem is that the column used as partition key contains numeric values. If the conversion from the numeric values is performed with a format mask like YYYYMMDDHH24MISS, the definition of the range partitions is still possible. However, if the conversion is based on a format mask like DDMMYYYYHH24MISS, you have no chance of solving the problem without changing the datatype or format of the column (as of Oracle Database 11g it is possible to workaround the problem by implementing virtual column based partitioning).

The fourth problem caused by wrong datatype selection is related to the query optimizer. This is probably the least obvious of this short list, and also the one leading to the subtlest problems. The reason for this is that the query optimizer will perform wrong estimates and consequently, will choose suboptimal access paths. Frequently, when something like that happens, most people think it's the fault of the query optimizer, that once again is not doing its job correctly. In reality, the problem is that information is hidden from it and so the query optimizer *cannot* do its job. To illustrate this problem, let's take a look at the following example. Here, we are checking the estimated cardinality of similar restrictions based on three columns that store the very same set of data (the date of each day in 2008), but based on different datatypes. As you can see, the query optimizer is only able to make meaningful estimations (the correct cardinality is 29) for the column which is correctly defined.

```
SQL> CREATE TABLE t (d DATE, n NUMBER(8), c VARCHAR2(8));

SQL> INSERT INTO t (d)
  2  SELECT trunc(sysdate,'year')+level-1
  3  FROM dual
  4  CONNECT BY 1 = 1
  5  AND level < trunc(sysdate+366,'year')-trunc(sysdate,'year')+1;

SQL> UPDATE t SET n = to_number(to_char(d,'YYYYMMDD')), c = to_char(d,'YYYYMMDD');

SQL> SELECT * FROM t ORDER BY d;

D                  N C
--------- ---------- --------
01-JAN-08   20080101 20080101
02-JAN-08   20080102 20080102
…
30-DEC-08   20081230 20081230
31-DEC-08   20081231 20081231

SQL> execute dbms_stats.gather_table_stats(user,'t')

SQL> EXPLAIN PLAN SET STATEMENT_ID = 'd' FOR
  2  SELECT * FROM t WHERE d BETWEEN to_date('20080201','YYYYMMDD')
  3                              AND to_date('20080229','YYYYMMDD');

SQL> EXPLAIN PLAN SET STATEMENT_ID = 'n' FOR
  2  SELECT * FROM t WHERE n BETWEEN 20080201 AND 20080229;

SQL> EXPLAIN PLAN SET STATEMENT_ID = 'c' FOR
  2  SELECT * FROM t WHERE c BETWEEN '20080201' AND '20080229';

SQL> SELECT statement_id, cardinality FROM plan_table WHERE id = 0;

STATEMENT_ID CARDINALITY
------------ -----------
d                     30
n                     11
c                     11
```

In summary, there are plenty of good reasons for selecting your datatypes correctly. Doing so may just save you a lot of problems.

Since 1995, Christian Antognini has been focusing on understanding how the Oracle database engine works. He is currently working as a principal consultant and trainer at Trivadis AG (www.trivadis.com) in Zurich, Switzerland. If he is not helping one of his customers get the most out of Oracle, he is somewhere lecturing on optimization. Christian is the author of *Troubleshooting Oracle Performance* (Apress 2008).