

Raise the Flag ... there's more to bit-wise operators than BITAND

By Grant "Fuzzy" Allen

Nine years ago, I went to work for a company that developed a particular off-the-shelf style application. I arrived just as a heated debate was occurring about the need for flags (that is, Boolean indicators) for a variety of data that had been described in the logical model that now needed to be translated to the physical model. The camps were basically split in to three.

The first group was adamant that every flag needed its own field – but this group were a fragile coalition of the “Integers are It” faction, and the “Choose Char” bloc.

The second group were firmly into multivalue logic ... and insisted that binary was a passing fad, and we needed to think about trinary and even quaternary logic.

The third group were sure we could just pack everything into one field, and use bit-wise operators to extract what we needed.

The biggest complicating factor here was the need to produce something portable, for use not only on Oracle, but also several other RDBMS products. The arguments ebbed and flowed, and in the end we opted for some one-flag-to-a-field implementations, where we knew the business logic was best served by the power of native SQL, and a few pack-a-bunch-into-one-field instances. A decision that had a world of consequences, as you'll see.

We set about developing our physical model, when to our dismay we learnt that bit-wise operators weren't exactly a core part of the SQL standards (such as they were), and Oracle in particular – at version 7.1 and going strong – didn't have any! Imagine grown men and women data modellers, architects and DBAs consoling each other, speaking of the physical model as it stood in the past tense ... almost like being at a wake! But then a couple of our “brightest and best” hit upon a great idea. We could use other functions - part of the basic SQL set provided by all vendors – to mimic bit-wise operations.

(I must point out at this stage that I wasn't among this august group ... I'm just passing on the story.)

The answer to our dreams was the humble `MOD` function.

I'm going to imagine for a minute that there was a round of gasps coming from the reading audience, in shock at this revelation. In truth, I suspect many of you have hit upon this technique before, but I'll press ahead to help inform any of you that haven't struck this ingenious use of the function.

First, let's set the scene. A test table will help us illustrate `MOD` as a bit-wise impersonator in motion.

```
Create table modtest
```

```
(flag integer,  
  sometext varchar(254));
```

(I've omitted storage clauses and the like for brevity).

Armed with our test table, let's populate it with a few well-chosen values to illustrate.

```
Insert into modtest values (0, 'No flags');  
Insert into modtest values (1, 'First flag only');  
Insert into modtest values (2, 'Second flag only');  
Insert into modtest values (3, 'First and second flags');  
Insert into modtest values (4, 'Third flag only');  
Commit;
```

Some of you will see already what's coming. The crux to using the MOD function in *de-facto* bit-wise operations is to store your bit-values in the decimal representation of the binary code. So $0000_2 = 0_{10}$, $0001_2 = 1_{10}$, $0010_2 = 2_{10}$, $0011_2 = 3_{10}$, $0100_2 = 4_{10}$ and so on (nothing new there).

We can then apply MOD to the field, and isolate individual flags as follows

```
Select sometext  
from modtest  
where MOD(flag, [decimal value of next highest flag]) >= [decimal  
value of flag sought];
```

For example, let's assume we're after the set of rows with the second flag set in our test data above. The second flag equates to 0010_2 , or 2 in decimal, and the "decimal value of the next highest flag" (the third flag), based on binary 0100_2 , is 4 decimal. So we form the query

```
Select sometext  
from modtest  
where MOD(flag, 4) >= 2;
```

And the result?

```
SOMETEXT  
-----  
Second flag only  
First and second flags
```

Why does it work? OK, it's not rocket science, but a simple case of excluding the flags in which you have no interest. By basing the MOD function on the value of the next-highest flag, we are essentially discarding that part of the composite flag field represented by flags higher than the one we're interrogating. Our above example is discarding all the information that might be present in the flag field based on flag 3, 4, 5 ... 122 ... 5186 ... you name it. We then effectively discard the information about lower flags, by the greater-than-or-equal-to operator. In essence, so long as the remainder of the MOD

function is at least equal to our flag's value, we know we have a hit, and don't care what the lower flags may represent.

Here are a few more examples.

Show the set of rows where the first flag is set

```
Select sometext
from modtest
where MOD(flag,2) >= 1;
```

```
SOMETEXT
-----
First flag only
First and second flags
```

Show the set of rows where the third flag is set

```
Select sometext
from modtest
where MOD(flag,8) >= 4;
```

```
SOMETEXT
-----
Third flag only
```

Back to the story of the physical model. We all congratulated each other, and set about rolling out just such logic where needed, with quite a bit of success.

Oracle Corp didn't stand still on the bit-wise operator front, and soon introduced the `BITAND` function. Their choice of focusing on the AND bit-wise operator was prudent; it was just enough to allow people to fashion whatever Boolean-logic they wanted to into their SQL statements. `BITAND` works on data in just the same way as our `MOD` examples above, treating decimal integers as their binary equivalent flags.

A quick example based on the above data will fill in the gaps for those not familiar with `BITAND`.

```
Select bitand(flag,1) "FLAG1?", sometext from modtest;
```

```
FLAG1? SOMETEXT
-----
0 No flags
1 First flag only
0 Second flag only
1 First and second flags
0 Third flag only
```

From those results, you can see `BITAND` returns the flag value (in this case 1) where the flag is set, 0 where it is not. This is more clearly seen with in `BITAND`'s usual form, when combined with `DECODE`.

```
Select Decode(bitand(flag,2),2,sometext) "FLAG2?"  
From modtest;
```

```
FLAG2?
```

```
-----
```

```
Second flag only
```

```
First and second flags
```

Both the `MOD` technique and `BITAND` can be used to test the values of multiple flags in one predicate – but `BITAND` is slightly more flexible in this regard.

That wraps up the tale of using `MOD` for bit-wise operations. `BITAND` has been on the scene for many years now, and while few people know of it, it is just as good as the `MOD` approach. But wherever you need to ensure portability of your SQL when performing bit-wise operations, keep it in mind.