

Query In PeopleSoft 8.9
Human Resources – Advanced Topics

May 30, 2007

SQL

SQL (Structured Query Language) is the language used to actually pass the queries you create to the database. Basically the database speaks one language, and Query acts as a translator to convert your data needs into something Oracle understands.

There are three major components of a SQL statement:

SELECT – the *fields* you want in your result set

FROM – the *records* in which to find the fields

WHERE – the *criteria* used to filter the data.

When you click on the ‘View SQL’ tab in Query, you’ll see the SQL query that has been created as a result of your query. As we find below, sometimes Query makes changes to what we have requested and also adds additional items at times. While you never need to write (or even be able to read) SQL to write queries, understanding its impact on your queries is important for two of the more advanced concepts.

Example 1: Effective Dating

Simply adding an effective dated row produces a lot of SQL code.

For example, opening the JOB record and selecting EMPL_ID with no criteria produces the following.

```
1      SELECT A.EMPLID
2,3,4  FROM PS_JOB A, PS_EMPLMT_SRCH_QRY A1
      WHERE A.EMPLID = A1.EMPLID
      AND A.EMPL_RCD = A1.EMPL_RCD
5      AND A1.OPRID = '0013693'
6      AND ( A.EFFDT =
      (SELECT MAX(A_ED.EFFDT) FROM PS_JOB A_ED
      WHERE A.EMPLID = A_ED.EMPLID
      AND A.EMPL_RCD = A_ED.EMPL_RCD
      AND A_ED.EFFDT <= SYSDATE)
      AND A.EFFSEQ =
      (SELECT MAX(A_ES.EFFSEQ) FROM PS_JOB A_ES
      WHERE A.EMPLID = A_ES.EMPLID
      AND A.EMPL_RCD = A_ES.EMPL_RCD
      AND A.EFFDT = A_ES.EFFDT) )
```

What is happening here?

1. You can see your fields in the select portion (EMPLID).
2. The record JOB has been added to the query.
3. A second record, EMPLMT_SRCH_QRY has been added and joined to job. This is the implicit row level security record which is added to queries.
4. All PeopleSoft records actually start with PS_ behind the scenes.
5. You will see your own EmplId which indicates it is checking your security against every row in the result set before return the data.
6. A bunch of ‘other stuff’ has been added to check EFFDT and EFFSEQ.

The first two items should make sense and be expected. Everything else is part of the overhead created by the query tool. Adding security (3 & 5) means the database is doing a lot of work on every row of every query you write – in my opinion that is a major factor affecting the speed of the database.

The ‘other stuff’ is where it gets interesting, basically if you look closely, they are little queries inside the larger query. These queries could stand on their own, they have individual *select*, *from* and *where* components. As you may have guessed, this is the primary meaning of what is meant by subqueries. But consider what it means for your query: *every* row generated by the master query you wrote is also firing off two other queries to check if the row is the latest effective dated row.

Earlier it was stated that effective dated records can’t be used in the ‘right’ side of a join – that is because subqueries are binding and it is impossible to make the join optional if the subqueries need to run to check validity of data. This is arguably the biggest drawback of using effective dating.

Flat File to the rescue

Let’s create the same query using the flat file, open UM_F_EMPL_VW and select EMPLID.

```
SELECT A.EMPLID  
FROM PS_UM_F_EMPL_VW A
```

That’s it.

No extra overhead, no subqueries, no security. Flat file queries are ridiculously easy for Oracle to process and produce results. That is why they are *so fast*. As someone who wrote SQL for years before I started using Query, this is what I expected to see when I viewed SQL. In everyday usage, SQL is actually very straightforward.

The point to take from this is that Query tries very hard to protect you from yourself by making the proper joins and checking what it produces in the result set; 99% of the time this is a good thing, but 1% of the time it will feel like the tool is putting up road blocks.

Distinct

Queries can produce result sets in which rows are duplicated.

Say you want the names of all employee in UMPISA and you go to the flat file and select ‘Name’ for a field and set the criteria for union to ‘002’. When I run a query for that criteria I get 2050 names system wide (that includes terminated individuals.) If I then go to the ‘Properties’ link at the bottom of ‘Query’ and click the check mark and re-run the query I get 2032 names (and the results are auto-sorted)

Adding the extra stipulation of Distinct says to the database, do not give me any rows which are an *exact* duplicate of another row in the result set. To perform that comparison it first sorts the results.

Discussion point: What would be the two causes of duplicates in the query as described?

The combination of the sorting plus the computation power to determine if rows are duplicates may cause your queries to be slow to the point of timing out. This was definitely a factor in 8.0 and I use it so rarely that I haven’t noticed if the same drawbacks affect 8.9 It is possible that the database has been optimized in 8.9 to make using *distinct* more attractive. As it stands, I would rarely use it except for data from the flat file or other records which do not depend on effective dating.

You may want to reconsider your criteria if you are queries produce duplicates because duplicates may be symptomatic of poorly specified criteria, or simply not selecting enough fields. In a few cases, duplicates are impossible to avoid. In SQL terms, all that changes is the word ‘distinct’ is placed after the word ‘Select’ in the query. This illustrates how the smallest change can have an enormous effect both in results in performance.

Technical Limits

There are limits to what you can do in Query. Result sets can be too large or they can take too long to process.

| | Maximum Result Set | Maximum Run Time |
|--------------------------------------|--------------------|------------------|
| Running a Query while editing: | 5 MB | 5 minutes |
| Running a Query to HTML or to Excel: | 15 MB | 10 minutes |
| Scheduling a Query | Unlimited | Unlimited |

Scheduled queries are unlimited up to theoretical database limits, and practical space/time considerations. You can't expect to write a query which does a Cartesian join between all of the records you can find, schedule it and hope to pick up the results sometime in 2012.

What this does mean is that you have some options if you are creating very large and/or very complex queries. Personally I get worried if I have a query running more than 30-40 seconds and it hasn't returned result, my initial reaction is that the query most likely won't return and I (or a DBA) will need to kill the query.

Scheduled queries are done through a separate option on the reporting tools menu. You need a run control ID, just as you would for a SQR report in production.

To kill a running Query, you go to:

PeopleTools > Utilities > Administration > Query Administration

Select the executing tab.

Make sure 'Queries that have been running longer than (n) minutes' is selected and type '1' into '(n)' box. Click search and you will see the queries that are running. Select your rogue query and kill it off by pressing the appropriate button.

With any luck I didn't already have to demonstrate this in the morning session.

Injected SQL

Injecting SQL is the concept of using features of the database which are not available directly from Query. Query only presents you with a subset of the capabilities of the underlying database. The primary reason is because PeopleSoft can run on different database architectures which have their own variants of SQL and commands to do things and Query is a generalized tool. Simple calculations like producing sums and averages are standard across virtually all databases, whereas statistical analysis functions may vary widely in name and usage.

Why bother?

A lot of data produced by Query is useful, but it would be more helpful if you could manipulate the data some before using it. Just as we did with expressions, being able to create our own fields enriches our opportunity to produce tight, succinct result sets. For example, we can report on date fields, but not the actual month in the date field. Some fields don't have easy translation schemes or we just want to produce a translation in a way that PeopleSoft doesn't think of our data.

Example II: EMPL_CLASS

When examining EMPL_CLASS, I know of no easy way to do a translation on that field when using job. I don't even know which lookup table to use to find the translations. (The flat file does have a translation for this field so it might be easiest to just bring that field in to do the translation, but this example will show the concept.)

So what do we do? Well Oracle has a SQL command called 'DECODE' which will take a field and produce different results based on the value in a field so if I ask the database to do the following:

```
DECODE( EMPL_CLASS, '1', 'PROFESSIONAL', 'OTHER' )
```

It will look at the field EMPL_CLASS, and check its value. If it finds a one in that field it will return the value 'PROFESSIONAL' otherwise it will just tell me 'OTHER' for any other value. The format is:

```
DECODE( FIELD_NAME, OPTION_1, TRANSLATE_1 ... OPTION_X, TRANSLATE_X,  
DEFAULT_TRANSLATION)
```

The 1...X notation indicates you have as many OPTION/TRANSLATE pairs in your decode statement as you need. Always use a default translation code at the end in case you find any unexpected values.

That isn't very useful by itself, but consider this example:

```
DECODE(A.EMPL_CLASS, '1', 'PROFESSIONAL', '2', 'CLASSIFIED', '3', 'STUDENT', '4', 'NON-  
STUDENT', '5', 'GRADUATE ASSISTANT', '6', 'FACULTY FY', '7', 'FACULTY  
AY', '8', 'FACULTY LAW', '9', 'FEDERAL', 'OTHER')
```

That would translate any EMPL_CLASS for you. Incidentally, I can't claim credit for this way of solving the EMPL_CLASS problem, I blatantly lifted it from a query written by Bill Gilfillan. It goes to show the types of things that you can learn from examining the queries of others and after you get a 'How did they do *that?*' moment stopping to investigate the query a little further.

So how do you use it? Simply take that code and type or, ahem, copy-and-paste into an expression in your own query. Make sure your expression type is correct – you would want a character field that could hold 20 characters to make this work. Then simply use that expression as a field. Your query would have to have an EMPL_CLASS field in the record aliased as A, of course. Change the alias to suit your records as appropriate.

Date Field Manipulations

Have you ever looked at the SQL for a date field?

It would look something like this if I asked to report on HIRE_DT:

```
SELECT TO_CHAR(A.HIRE_DT, 'YYYY-MM-DD')
FROM PS_UM_F_EMPL_VW A
```

Note that in this case, Query is doing its *own* SQL injection. Internally in the database, dates are stored as a special format which often looks something like 30-MAY-2007. This says, convert that date to a character field in the format of 'YYYY-MM-DD' which should produce a date of 2007-05-30. However, the Query tool goes even farther (and doesn't even tell you) and changes the format to the local date format of MM/DD/YYYY before you actually see anything. If you use PeopleSoft in Europe it will presumably silently convert the dates to the international DD/MM/YYYY format for display.

The importance of understanding this is that you can use the same command to produce your own custom fields based on dates.

If you create an expression with the following command:

```
TO_CHAR(A.HIRE_DT, 'MM')
```

It should produce a character field with a length of 2 that extracts the month from the date.

Interestingly, you can use this expression to determine criteria, but not as a field. Why? Because when you use it as a field, it substitutes the translation for A.HIRE_DT to do the full translation and the SQL ends up looking like this:

```
TO_CHAR( TO_CHAR(A.HIRE_DT, 'YYYY-MM-DD'), 'MM')
```

The underlined section shows Query trying to be helpful again. The result is that my query produces an error as Query's meddling renders the expression nonsensical.

Let's try a different tact.

```
SUBSTR( A.HIRE_DT, 6, 2 )
```

What am I doing in this expression?

First, I'm letter Query do its magic and create a 10 character field in the format 'YYYY-MM-DD'. Query *thinks* it is being helpful so I just let it do what it wants.

Then I am asking for a SUBSTring of the result from the date conversion. Go over to the sixth position of the date and give me 2 characters.

| | |
|-------|------------|
| Field | 2007-05-30 |
| Count | 123456789X |
| | ^^ |

This works. I can use this as a field in my query.

It produces the following SQL:

```
SELECT TO_CHAR(A.HIRE_DT, 'YYYY-MM-DD' ),  
       SUBSTR( TO_CHAR(A.HIRE_DT, 'YYYY-MM-DD' ), 6, 2)  
FROM PS_UM_F_EMPL_VW A
```

The following expressions allow you to extract years and days from dates:

```
SUBSTR( A.HIRE_DT, 1, 4 )  
SUBSTR( A.HIRE_DT, 9, 2 )
```

Now admittedly I was only able to do this because I have some knowledge of how Query works and have used Oracle enough to know the internal command syntax. But it shows how many more possibilities are available beyond just what query shows me. Short references on Oracle's variant of SQL are cheaply available and should give you what you need to use this ability.

Subqueries

Consider the following queries:

| Query | Solution |
|--|---|
| Find all Faculty | Query JOB |
| Find all Faculty with Tenure Data | Query JOB and join EG_TENURE_DATA |
| Find all Faculty with or without Tenure Data | Query JOB and outer join EG_TENURE_DATA |

Now as a data entry check how would you find all faculty *without* any tenure data? The trick is to determine who isn't actually in the tenure table.

Consider the following query:

```
SELECT A.EMPLID, A.NAME
FROM PS_UM_F_EMPL_VW A
WHERE A.EMPL_CLASS = '7'
      AND A.FULL_PART_TIME = 'F'
      AND A.EMPLID NOT IN (SELECT B.EMPLID
FROM PS_EG_TENURE_DATA B)
```

This query first looks at the flat file (just for convenience over looking at effective dated rows in JOB) and selects EMPLID and NAME. The criteria restricts to results to academic year faculty who are full time.

However notice the bottom part of the query. It is asking if the EMPLID on the row is NOT IN a query of all of the EMPLIDS in the tenure record.

To create the query, I did the following.

Edit Criteria Properties

Choose Expression 1 Type

- Field
- Expression

Expression 1

Choose Record and Field

Record Alias.Fieldname:

A.EMPLID - EmplID

*Condition Type: not in list

Choose Expression 2 Type

- In List
- Subquery

Expression 2

Define Subquery

Define/Edit Subquery

I selected the 'not in list' criteria and choose 'Subquery' for my Expression 2 type. I then need to 'Define/Edit Subquery'

When I go into create my subquery, it basically starts the process all over again for creating a query. Pick records, set criteria, etc. I selected the tenure record (EG_TENURE_DATA) and picked emplid for my field.

There is one key difference when creating a subquery:

Subqueries only return one field. Subqueries are a way to further refine criteria, not to bring in more fields to the query. The goal is to compare a value in the main query with the result set from the subquery – there simply is no reason to return more than one field.

You can also join fields in the subquery to fields in the records of the main query:

```
SELECT A.EMPLID, A.NAME
FROM PS_UM_F_EMPL_VW A
WHERE A.EMPL_CLASS = '7'
      AND A.FULL_PART_TIME = 'F'
      AND A.EMPLID NOT IN (SELECT B.EMPLID
FROM PS_EG_TENURE_DATA B
WHERE B.EMPLID = A.EMPLID)
```

Now it is doing the following: “I have EMPLID XXXXXXXX, run a query to see if XXXXXXXX exists in tenure data. If it does I’ll use it, otherwise let’s skip it and look at the next EMPLID.” This is much more efficient than saying, “Give me a list of everybody in tenure data and I’ll check it and see if you have the one I want. Then we will do the whole thing again for the next EMPLID.” which we were doing before.

Unions

Unions are another way to get around the limitations of queries. Sometimes you want to look at data with two different sets of criteria, but it is not possible to use simple OR statements to produce the results you want. In this case it really is two wholly different result sets which you want to merge. When you see 'union' think 'merge.'

Consider the relatively simple query UM_EARNBAL_CY_YTD which looks at the earnings for an employee over a calendar year by earnings code, summarizes the earnings per employee record for an employee and displays a separate grand total. (I may have said in the morning session that subtotalling wasn't possible. Whoops.)

Here is some sample output:

First ◀ 1-7 of 7 ▶ Last

| ID | Empl Rcd# | Year | Earn Code | Gross YTD |
|---------|-----------|------|-----------|-----------|
| 0010571 | 0 | 2006 | 001 | 6666.64 |
| 0010571 | 0 | 2006 | 750 | 1666.66 |
| | 0 | 2006 | SUB | 8333.30 |
| 0010571 | 1 | 2006 | 001 | 38890.72 |
| 0010571 | 1 | 2006 | 750 | 580.44 |
| | 1 | 2006 | SUB | 39471.16 |
| | 999 | 2006 | TOT | 47804.46 |

Discussion point: How would you approach this problem?

The main approach to producing sub or grand totals is to create additional queries which has a result set with the same *number* of fields as the original query but have different criteria and, in this case, calculation methods.

It is difficult to display a union in Query because each query is handled separately and you navigate between them. The SQL below might better illustrate the point. It consists of a main query and two unioned queries. I have touched up the SQL for better readability.

```
SELECT C.NAME, A.EMPLID, A.EMPL_RCD, A.BALANCE_YEAR, A.ERNCD, A.GRS_YTD
FROM PS_EARNINGS_BAL A, PS_PERSONAL_DATA C
WHERE C.EMPLID = C1.EMPLID
      AND ( A.EMPLID = :1
            AND A.SPCL_BALANCE = 'N'
            AND A.BALANCE_ID = 'CY'
            AND A.BALANCE_PERIOD = (SELECT MAX( B.BALANCE_PERIOD)
FROM PS_EARNINGS_BAL B
WHERE B.EMPLID = A.EMPLID
      AND B.ERNCD = A.ERNCD
      AND B.BALANCE_YEAR = A.BALANCE_YEAR
      AND B.BALANCE_ID = A.BALANCE_ID
      AND B.EMPL_RCD = A.EMPL_RCD)
      AND A.BALANCE_YEAR = :2
      AND A.EMPLID = C.EMPLID )
```

UNION

```
SELECT '', '', D.EMPL_RCD, D.BALANCE_YEAR, 'SUB', SUM( D.GRS_YTD)
FROM PS_EARNINGS_BAL D
WHERE D.EMPLID = :1
      AND D.BALANCE_YEAR = :2
      AND D.SPCL_BALANCE = 'N'
      AND D.BALANCE_ID = 'CY'
      AND D.BALANCE_PERIOD = (SELECT MAX( E.BALANCE_PERIOD)
FROM PS_EARNINGS_BAL E
WHERE E.EMPLID = D.EMPLID
      AND E.ERNCD = D.ERNCD
      AND E.BALANCE_YEAR = D.BALANCE_YEAR
      AND E.BALANCE_ID = D.BALANCE_ID
      AND E.EMPL_RCD = D.EMPL_RCD)
GROUP BY '', '', D.EMPL_RCD, D.BALANCE_YEAR, 'SUB'
```

UNION

```
SELECT '', '', 999, G.BALANCE_YEAR, 'TOT', SUM( G.GRS_YTD)
FROM PS_EARNINGS_BAL G
WHERE G.EMPLID = :1
      AND G.BALANCE_YEAR = :2
      AND G.SPCL_BALANCE = 'N'
      AND G.BALANCE_ID = 'CY'
      AND G.BALANCE_PERIOD = (SELECT MAX( H.BALANCE_PERIOD)
FROM PS_EARNINGS_BAL H
WHERE H.EMPLID = G.EMPLID
      AND H.ERNCD = G.ERNCD
      AND H.BALANCE_YEAR = G.BALANCE_YEAR
      AND H.BALANCE_ID = G.BALANCE_ID
      AND H.EMPL_RCD = G.EMPL_RCD)
GROUP BY '', '', 999, G.BALANCE_YEAR, 'TOT'
ORDER BY 3, 5
```

This query uses nearly all of the concepts used today: prompts, aggregations, expressions, subqueries (noted in italics) and unions.

Criteria/Join Order

Does it matter?

Maybe. It makes more sense to narrow your results as much as possible in the first bits of criteria and then go from there. I haven't tested this extensively but anecdotally well constructed queries with highly restricted criteria near the 'top' of the SQL statement and joins that bring in progressively smaller data sets are more efficient. It is entirely possible that when Oracle internally analyses what it is given that some optimization and efficiency management *is* being done on the query. The DBAs might know, but in practical terms you will rarely (or never) write queries that would benefit from any optimizations.

The best technique may be to avoid using unnecessary complications in the data you are using (*i.e.*, forgetting what you just learned and never using unions and subqueries) and sticking to the flat file.