AN INCREMENTAL APPROACH TO DEVELOPING SQL QUERIES

July 1, 2000

The key to getting accurate information from your database is to write good SELECT statements. In fact, it's the key to just about everything you do with your data. Even the tasks of updating and deleting data depend on your ability to select the relevant data in the first place. Because selecting the correct data is the foundation for much of what you do with your database, it's worth taking some time to think about how you write SELECT statements.

Many people begin with the SELECT clause itself: They start by listing the columns they want returned and then work through the rest of the statement in the order in which the different clauses, FROM, WHERE, etc., occur. The problem with this approach is that the order of the clauses in the SELECT statement represents a suboptimal sequence of steps for developing a query. They lead you to focus on the wrong clause at the wrong time, opening the door for subtle bugs to work their way into your query. And by writing the entire statement in one go, you miss the opportunity to discover problems that would become apparent if you were to use an incremental approach.

For example, summarizing (using the GROUP BY clause) can mask poorly specified join conditions as well as other mistakes in the WHERE

clause. Summarize only after you are certain you are summarizing the correct data. You might also incorrectly specify join conditions right from the start, but other conditions in the WHERE clause might make the results of the query appear to be working. You will get bitten down the road when the data changes or when someone else modifies the query.

Incremental Methodology

I've long felt that a rigorous and incremental approach to developing a SQL query greatly improves the odds of my writing the query correctly the first time. I've also long felt that the optimal order for writing the clauses is not necessarily the order in which they appear in the final statement. Over the years, I've developed an approach that I follow for almost every query I write. The two keys are to work through each clause one at a time, in what I consider to be the optimal order, and to sanity-check the results at each step. Here's the order in which I work through the clauses:

1. **The FROM clause:** First, I identify the tables that hold the data I need. Then, I work through the process of joining them correctly.
2. **The WHERE clause:** Once I'm sure I've specified the join conditions correctly, I add any other necessary conditions to the WHERE clause.
3. **The GROUP BY clause:** Only after I'm sure I have the correct data do I even think about summarizing it.
4. **The HAVING clause:** Once I've summarized the data correctly, I eliminate any summary rows I don't want in the result set.
5. **The SELECT statement:** Having gotten this far, if I still need to add any columns to the SELECT list, I add them at this time.
6. **The ORDER BY clause:** Usually the very last thing I do in this process is specify the sort criteria.

It's true that I need to at least think about the columns I list in the SELECT statement in order to identify the tables, and it's also true that I need to select some data in order to test the join. However, I'm not contradicting myself. My focus during Step 1 is on the join, and any columns I select will be the primary- and foreign-key columns necessary to verify that I've done the join correctly.

When I'm working through these steps, I take care to sanity-check the query at each step. To verify that I've joined the tables correctly, for example, I select the columns involved in the join and review them. Equijoins are easy to specify and easy to validate—you just need to be sure the join columns from each table match. Things get a little tougher if you are joining two tables based on a range of values, such as a range of dates.

Where possible, I generate test data to verify that boundary conditions and other exceptional cases don't cause my query to fail. When joining tables on a range of dates, I may generate some test data that has a time component, because it's not uncommon for DATE fields to also have a time associated with them.

When building complex queries, here are some other things I do:

- Test each of the queries in a union query separately.
- Test subqueries separately.
- Test complex expressions separately.
- Look at raw data before I apply built-in functions to it.

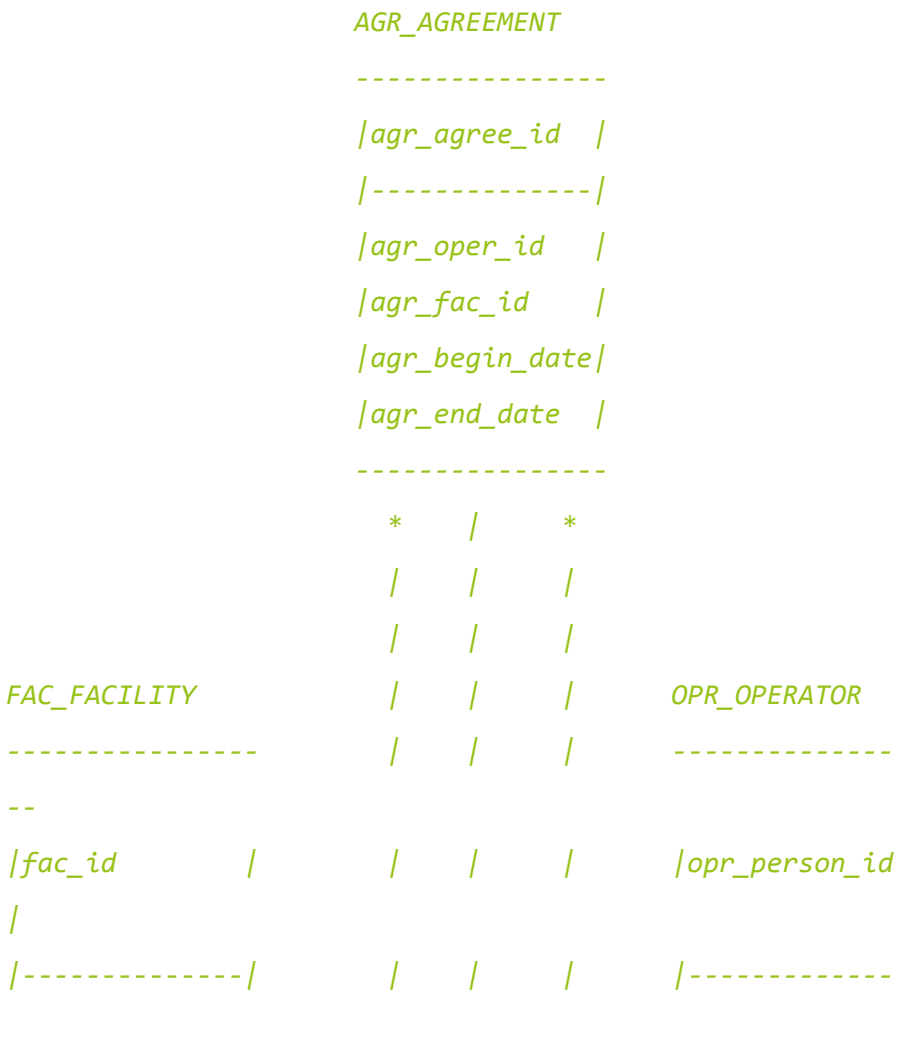
At times I am forced to violate my usual query-development order—if I'm joining tables based on summarized data, for example. In such a case, I work on that join after I've written the GROUP BY clause. But typically I find that using the stepwise approach I've just described is a good way to ensure that relationships among all data elements—even things that may be hidden from me at first, as we'll see in a minute—are taken into account to provide accurate information.

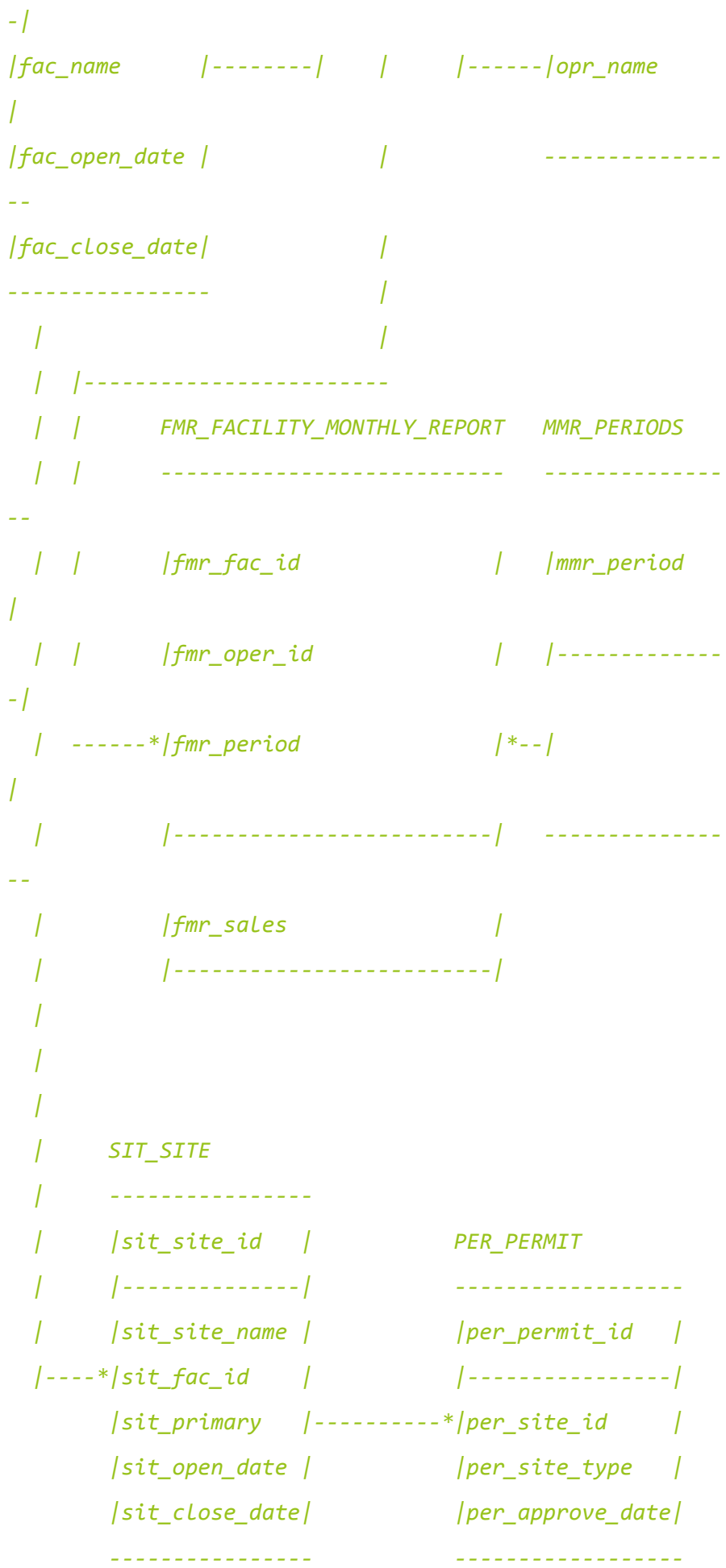
Working Through an Example

Let's work through an example, based on the database design (see Figure 1) for the Michigan Commission for the Blind's Business Enterprise Program, to illustrate how useful this process is.

Figure 1: Business Enterprise Program Database Design

The following is a small snapshot of a larger design that I helped implement for the Michigan Commission for the Blind's Business Enterprise Program. The program enables blind and sight-impaired people to make a living by operating vending concessions, stores, and cafeterias in office buildings and rest areas throughout the state. The database contains several many-to-one relationships, and many of the relationships have a time component.




fac_name
fac_open_date
fac_close_date
opr_name
FMR_FACILITY_MONTHLY_REPORT
fmr_fac_id
fmr_oper_id
fmr_period
fmr_sales
MMR_PERIODS
mmr_period
SIT_SITE
sit_site_id
sit_site_name
sit_fac_id
sit_primary
sit_open_date
sit_close_date
PER_PERMIT
per_permit_id
per_site_id
per_site_type
per_approve_date

The Business Enterprise Program aims to enable blind and sight-impaired people to make a living by operating vending concessions, stores, and cafeterias in office buildings and rest areas throughout the State of Michigan. As you can see from Figure 1, the database design contains several relatively typical-looking many-to-one relationships among tables that hold data for facilities, operators, sites, permits, and periods. The one unusual thing is that many of the relationships have a time component, which makes the task of joining these tables a bit more "interesting" than usual.

Results of One-Pass Query Approach Can Be Misleading

Let's say that we need to report the number of new facilities opened during the calendar year

2000 and that we need to break out this count by facility type. The facility type is derived from the permit record for the facility's site, so you have to join the facility, site, and permit tables in order to properly count by type. Listing 1 shows one possible solution to this problem, along with the results it generates.

Listing 1: The first version of a query to report on new facilities opened during the year 2000, broken out by facility type. The results appear to be correct, but they are not.

```
SELECT COUNT(*),
       DECODE (per_site_type,
          'CAFE','Cafeteria',
          'STOR','Store',
         'VEND','Vending Machines') "Facility Type"
```

```
    FROM fac_facility, sit_site, per_permit
    WHERE fac_id = sit_fac_id
      AND sit_site_id = per_site_id
      AND fac_open_date >= TO_DATE('1-Jan-2000','dd-mon-
yyyy')
      AND fac_open_date <= TO_DATE('31-Dec-2000','dd-mon-
yyyy')
    GROUP BY per_site_type
    ;

      COUNT(*) Facility Type
    ---------- ----------------
             2  Cafeteria
             2  Store
             2  Vending Machines
```

These results look great, don't they? Two facilities of each type—two cafeterias, two stores, and two vending machines—were opened during the calendar year 2000. Although they may look good, these results are actually incorrect (as we'll see in a minute as we step through the incremental approach). If you just type the query in as a whole, that's not obvious. If, on the other hand, you work through the query methodically and sanity-check the results as you go, the flaws quickly become apparent.

Take an Incremental Approach

To start over again, let's revisit the join conditions. Facilities have sites, and the SIT_FAC_ID column of the SIT_SITE table ties a site to a facility. In addition, sites have permits, and the PER_SITE_ID column in the PER_PERMIT table ties a permit to a site. With this in mind, let's strip the previous query down to just the join, and let's display the primary- and foreign-key information so that we can verify the join between the site table and the permit table (see Listing 2)

Listing 2: Displaying the primary- and foreign-key data to verify the JOIN for the query in Listing 1.

```
    SELECT fac_id, sit_site_id, sit_fac_id,
           per_permit_id, per_site_id
    FROM fac_facility, sit_site, per_permit
    WHERE fac_id = sit_fac_id
      AND sit_site_id = per_site_id
    ;

       FAC_ID SIT_SITE_ID SIT_FAC_ID PER_PERMIT_ID
PER_SITE_ID
    ---------- ----------- ---------- ------------- ---------
--
             1          10          1           100
10
             2          20          2           200
20
             3          30          3           300
30
             3          31          3           301
31
             4          40          4           400
40
             5          50          5           500
50
             5          51          5           501
51
             6          60          6           600
60
             7          70          7           700
70
             8          80          8           800
80
```

Let's look at the results of this interim query: The results list two lines of detail each for both sites 3 and 5 . But why, when we were expecting only one line each? Because our query doesn't take into account the one-to-many relationship between facility and site into account. We need a site record in order to get to a permit, but which site should we use, if a given facility has multiple sites?

The answer is that we should use the site flagged as the primary site. This doubtless comes as a surprise, because I haven't mentioned anything about the primary-site flag. Trust me, though, your clients will occasionally surprise you in the same way—sometimes you have to ferret out what they haven't thought to tell you. If we add the primary-site restriction to our join, our query now looks like that in Listing 3.

Listing 3: Adding the primary-site restriction to the join.

```
SELECT fac_id, sit_site_id, sit_fac_id,
       per_permit_id, per_site_id
FROM fac_facility, sit_site, per_permit
WHERE fac_id = sit_fac_id
  AND sit_site_id = per_site_id
  AND sit_primary = 'Y'
;
```

But wait! Can a site have more than one permit? The answer is yes. The database design allows multiple permits per site. This is a great example of how finding one problem can lead you to think of another. To resolve this issue, let's assert the business rule that the only permit that matters is the one most recently approved. To restrict the join to only the most recently approved permit, we must add a subquery to our WHERE clause, as in Listing 4.

Listing 4: Adding a subquery to the WHERE clause to restrict the join to the most recently approved permit.

```
SELECT fac_id, sit_site_id, sit_fac_id,
       p1.per_permit_id, p1.per_site_id
FROM fac_facility, sit_site, per_permit p1
WHERE fac_id = sit_fac_id
  AND sit_site_id = p1.per_site_id
  AND sit_primary = 'Y'
  AND p1.per_approve_date = (
        SELECT max(per_approve_date)
        FROM per_permit p2
        WHERE p2.per_site_id = p1.per_site_id)
;
```

This would also be a good time to generate some test cases for sites that do have multiple permits, so we can be sure that our subquery is behaving as we expect it to, but let's skip that step for now. If this subquery were any more complex, I would be inclined to test it separately from the main query, but it's not complex, so let's just trust ourselves on this one.

Now that we've specified our join conditions, it's time to work on the rest of the WHERE clause. The only thing left is to restrict the query results to facilities that were opened during the calendar year 2000. But first, let's see what our facility-open dates really look like. To do this, we modify the SELECT list of our query to return just the facility ID and the open date. Listing 5 shows the results.

Listing 5: Modifying the query's SELECT list to return just the facility ID and the date it opened.

```
SELECT fac_id, TO_CHAR(fac_open_date,
       'dd-Mon-yyyy hh:mi:ss am') fac_open_date
FROM fac_facility, sit_site, per_permit p1
WHERE fac_id = sit_fac_id
AND sit_site_id = p1.per_site_id
AND sit_primary = 'Y'
```

```
AND p1.per_approve_date = (
        SELECT max(per_approve_date)
        FROM per_permit p2
        WHERE p2.per_site_id = p1.per_site_id)
;

    FAC_ID FAC_OPEN_DATE
---------- --------------------------------
         1 01-Dec-1999 12:00:00 am
         2 01-Jan-2000 12:00:00 am
         3 31-Dec-2000 03:15:00 pm
         4 28-Feb-2000 12:00:00 am
         5 15-Jun-2000 12:00:00 am
         6 01-Mar-2000 12:00:00 am
         7 24-Aug-1999 12:00:00 am
         8 27-Apr-2000 12:00:00 am
```

Notice that the open date for facility 3 includes a time component, which the original query didn't take into account. We need to write our query in such a way as to make the time of day irrelevant. Instead of checking to see if the open date is less than or equal to 31-Dec-2000, we can check to see if it is less than 1-Jan-2000. That way, any time up to midnight on December 31 will be included. Avoid using the TRUNC function on the date column, because that might preclude the Oracle database server from using any indexes on that column. Listing 6 shows the new version of our query.

Listing 6: Making the time of day irrelevant, to include any time up to midnight on December 31.

```
    SELECT fac_id, TO_CHAR(fac_open_date,'dd-Mon-yyyy
hh:mi:ss am') fac_open_date
    FROM fac_facility, sit_site, per_permit p1
    WHERE fac_id = sit_fac_id
      AND sit_site_id = p1.per_site_id
```

```
  AND sit_primary = 'Y'
  AND p1.per_approve_date = (
        SELECT max(per_approve_date)
        FROM per_permit p2
        WHERE p2.per_site_id = p1.per_site_id)
  AND fac_open_date >= TO_DATE('1-Jan-2000')
  AND fac_open_date < TO_DATE('1-Jan-2001')
;
```

This date example underscores the importance of looking at your data and sanity-checking your results. It also pays to validate any underlying assumptions your query makes about the data being retrieved.

Now that we know our query is returning the correct data, the next step is to summarize it. Our goal was to count the number of facilities of each type opened during the year. To achieve that, we need to summarize on the PER_SITE_TYPE column and we need to use the aggregate function COUNT to report the number of facilities of each type. Our final query, together with the results it returns, is shown in Listing 7.

Listing 7: The final query counts total facilities of each type opened during the year 2000.

```
SELECT COUNT(*),
    DECODE (per_site_type,
            'CAFE','Cafeteria',
            'STOR','Store',
            'VEND','Vending Machines',
            'Unknown') "Facility Type"

FROM fac_facility, sit_site, per_permit p1
WHERE fac_id = sit_fac_id
AND sit_site_id = p1.per_site_id
AND sit_primary = 'Y'
AND p1.per_approve_date = (
```

```
        SELECT max(per_approve_date)
        FROM per_permit p2
        WHERE p2.per_site_id = p1.per_site_id)
AND fac_open_date >= TO_DATE('1-Jan-2000','dd-Mon-yyyy')
AND fac_open_date < TO_DATE('1-Jan-2001','dd-Mon-yyyy')
GROUP BY per_site_type
;

COUNT(*) Facility Type
---------- ----------------
        1 Cafeteria
        2 Store
        3 Vending Machines
```

These results are different from those reported by the first version of the query. By working through the query-development process in a methodical manner and sanity-checking our results after each step, we identified and resolved several problems that caused the query to return incorrect results.

With a large database, it may not always be feasible to look at every row returned by a query. In such a case, you may find it advantageous to create a test database containing some carefully constructed test data. You may also find it helpful to write queries that look for cases in which your assumptions about the data are incorrect. For example, with respect to the query we've just developed, are 'CAFE', 'STOR', and 'VEND' really the only possible site types? Do all facilities have a primary site? These assumptions built into the query are easy to test. For the real-world report on which I based this example, I developed several queries to verify the business rules and assumptions on which the main query rested. These queries generate an exceptions report that always accompanies the main report.

Summary

When you develop a query, you can greatly improve your chances of success by doing the following:

- Develop the query incrementally.
- Sanity-check the results after each step.
- Work through the clauses in the following order: FROM, WHERE, GROUP BY, HAVING, SELECT, and ORDER BY.

In this article, we detected several problems just by looking at the data returned by the query in its various stages of development. An initial query gave erroneous results—which appeared to be just fine—because it didn't take into account the one-to-many relationship between facilities and sites. Other details of database design that might not have been immediately apparent—the primary site flag, for instance, were revealed only as we constructed our query piece by piece, testing the results as we went along. Had we not followed a methodical process, we might have missed some of these subtle, data-related issues.

Acknowledgments

- *Thanks to Oracle Corporation for kindly allowing me to reprint an article I wrote for their July/August 2000 issue of Oracle Magazine.*
- *Thanks (again) to the Michigan Commission for the Blind Business Enterprise Program for its gracious permission to use real-world examples from its database for this article.*