



Chapter 1

Preliminaries

Before you can actually write ad hoc queries, there are some preliminaries that must receive attention. This chapter discusses the most frequently asked questions about issues that occur prior to actual query writing.

Query writers experienced with a specific database bring a wealth of information to bear on an ad hoc query. This information accrues over time in the form of database documentation, assembly of procedural manuals and institutional data standards, discussions with data entry staff and business policy makers, and much play and experimentation with the data. For the most part, a query writer does this on his own.

However, a formal part of the information gathering includes assembly of a set of tools that every query writer should own. This chapter begins by discussing the tools that will help you explore a database — simple queries that produce reports on table owners, table descriptions, data column dictionaries, index inventories, and constraint conditions. You'll refer to these reports frequently as you write queries.

Probably the most obvious indicator of a novice query writer is the absence of standards applied to the queries. The queries are unstructured and difficult to read; they're not modularized for ease of understanding and reuse; they don't utilize powerful techniques for simplifying the code; and they're not well documented. No generally recognized ad hoc query standards exist, but this chapter discusses several good programming practices that each query should employ.

To accommodate individual idiosyncrasies in preferred working style and to produce consistent environments for printed or screen reports, you'll want to customize the SQL*PLUS environment. This chapter discusses the options available and then concludes with an interesting problem that can arise when you're actually trying to run a query.

Exploring the Database

Whether you're new to a database, to a functional area within the database, to a data table, or even to a data column, you need to explore, play with, and learn about the unknown.

This section provides several tools that make the exploration easier. It includes reports on owners, tables, data dictionaries, indexes, and constraints — all items essential for writing ad hoc queries.

How do I get an overview of the database?

How can I get a broad overview of a new database I'm working with — just something to help orient me?

Approach

Understanding a database is a bit like peeling an onion. Aside from the tears that well up in your eyes occasionally, each time you peel back a database layer there's another layer underneath.

Data in Oracle databases is stored in tables, and tables have owners. Lists of owners and tables serve as helpful guides to the database landscape — a way of peeling back that first layer.

Owners

Run the program `owners.sql` listed in Appendix A (page 319). This produces a frequency distribution that counts the number of tables by owner and sketches the database at a high level of aggregation (see Figure 1-1). In database systems comprised of modules such as a finance module or a human resource module, table owners often distinguish the modular areas. The number of tables provides a rough indication of complexity.

Summary of table owners	
owner	count
GLEWIS	1
KOMENDA	21
OR2	3
SYS	79
SYSTEM	2
sum	106

Figure 1-1: Sample report of table owners produced by `owners.sql`.

Comments

▶ `SYS` and `SYSTEM` owners appear in all Oracle databases, providing considerable information about the database itself. `SYS`, for example, owns the views used to produce both the owner and table reports that appear in this section.

Tables

Run the program `tables.sql` listed in Appendix A (page 320). The report lists all tables and views for a specified owner and includes a description of the table or view if it's available. Figure 1-2 shows an example of this type of report.

Summary of OR2 tables			
name	type	owner	comments
ADDRESS	TABLE	OR2	Table of Addresses
CADDRESS	TABLE	OR2	Code Table of Address Types
NAME	TABLE	OR2	Table of Names

Figure 1-2: Sample table report produced by `tables.sql`.

Comments

▶ Table descriptions, unfortunately, are not required items during database design. So you may run the `tables SQL` script and find blank table descriptions. If this happens, talk to your database administrator about adding comments that describe the tables.

See Also

1. Chapter 6 on page 174 discusses frequency distribution reports in more detail.
2. Several `SYS` views appear in this book, including `ALL_TAB_COMMENTS` used to produce the report in Figure 1-2. Check the index under `SYS views` for a listing.

How do I get a dictionary for a table?

When I DESCRIBE a table, the result doesn't actually describe the contents of each data column. How can I get a data dictionary for a table that would provide more information?

Approach

Lists of owners and tables won't take you very far when you write ad hoc queries. You'll quickly need to identify what data columns exist in the tables you intend to use and determine what data are stored in these columns. A data dictionary gives you a place to begin this exploration. It defines the data columns in a table and provides a brief description of the data each column contains.

You're likely to find, however, that a complete understanding of a data column requires that you also consult office procedural manuals, written or de facto input and maintenance standards, users familiar with the data entry, or programmers who actually use the data column.

Dictionary

Run the program tabldict.sql listed in Appendix A (page 322). The SQL prompts you for a table name and a table owner and produces the type of report shown in Figure 1-3.

Column Name	Type	Width	Scale	Nulls	Column comments
ADDRESS_ID	VARCHAR2	5		NOT NULL	ID of person
ADDRESS_TYPE	VARCHAR2	2		NOT NULL	Code for address type
ADDRESS_STREET	VARCHAR2	25		NULL	Street address
ADDRESS_CITY	VARCHAR2	20		NULL	City of address
ADDRESS_LOC	VARCHAR2	5		NULL	State/province/etc of address
ADDRESS_PCODE	VARCHAR2	10		NULL	Postal code of address
ADDRESS_COUNTRY	VARCHAR2	3		NULL	Country code of address
ADDRESS_SERIESNO	NUMBER	2	0	NOT NULL	Sequential number for addresses of a specified address_type

Figure 1-3: Sample data dictionary produced by tabldict.sql.

If you run the report and find that the comments are blank, this means that the database designer or table owner did not document the columns properly. In such cases, we can only give you the standard response — talk to your database administrator.

Comments

▶ The data dictionary is date and time stamped and includes page numbers, the type of object (a table or view), its name and owner, and a description of the table or view. Documentation in the header also includes the name of the SQL program that produced the report (tabldict.sql) and the name of the report output file (here it's address.dic).

▶ Most of the columns appearing in the data dictionary are self-explanatory. The only one that needs additional detail is the report column labeled "Scale". The NUMBER datatype is used to store numeric data. NUMBER includes two parameters, one defining the *precision* of the data and the second defining its *scale*. Precision is the total number of *significant* digits a numeric value may contain; scale is the number of digits to the right of the decimal point. Significant digits exclude any leading zeroes and any trailing decimal zeroes. For example, a data column with the datatype NUMBER(5,2) can contain up to five significant digits, with two of the digits being to the right of the decimal point. The numbers 123.45 and 12345.00 both qualify.

In the ADDRESS data dictionary, the datatype for address_seriesno is NUMBER(2,0). A scale of zero means the number is an integer. Negative scales are also possible. For example, a datatype of NUMBER(5,-2) means that data values get rounded to the nearest hundred.

In data dictionaries produced by tabldict.sql, the report column labeled "Width" indicates the precision of NUMBER datatypes and the maximum possible characters that can be stored in CHAR or VARCHAR2 columns.

See Also

1. Report titles for the data dictionary in Figure 1-3 were created with the TTITLE command. See the index under TTITLE *command* for other examples of its use.
2. Data column comments for dictionaries appear in the SYS view called ALL_COL_COMMENTS. See the index under SYS *views*.

How do I identify the indexes for a table?

One of my queries takes a long time to run, and I'd like to ensure I'm using indexes effectively. How can I identify what indexes exist for a table?

Approach

Indexes serve an obvious purpose when tuning a query to improve its performance. But indexes also serve a second important function. For queries in which you expect one row returned for each population item, examining the unique indexes in joined tables can provide clues about the best way to structure the query so that row singularity is maintained.

Appendix A (page 324) lists a program called `indexes.sql` that produces a formatted report for all indexes on tables with a specified owner. The program produces a report like the one shown in Figure 1-4.

```

Indexes on tables owned by: OR2
Date: 10-MAY-1996      Time: 10:53
Report name: OR2.idx
Page: 1
SQL name: indexes.sql
    
```

table	index	uniqueness	column	pos
ADDRESS	PK_ADDRESS	UNIQUE	ADDRESS_ID	1
			ADDRESS_TYPE	2
			ADDRESS_SERIESNO	3
ADDRESS	UK_ID_ZIP	UNIQUE	ADDRESS_ID	1
			ADDRESS_PCODE	2
CADDRESS	PK_CADDRESS	UNIQUE	CADDRESS_CODE	1
NAME	NAMES	NONUNIQUE	NAME_LAST	1
			NAME_FIRST	2
NAME	PK_NAME	UNIQUE	NAME_ID	1
			NAME_SERIESNO	2

Figure 1-4: Sample index report produced by `indexes.sql`.

Comments

▶ Oracle7 creates an index on any data column or combination of data columns identified as *unique keys* or *primary keys*. The difference between the two types of keys is subtle but important. A table can have only one primary key, while many unique keys are possible. In either type of key, no two rows in a table may contain the same values for the data columns comprising the key. However, data columns appearing in a unique key may contain NULL values; this is not true for data columns in primary keys. In fact, if a row contains NULL values in all the data columns in a unique

key, it automatically meets the uniqueness criteria — implying that two or more rows may contain `NULL` values in the key columns and the uniqueness constraint will not be violated.

The distinctions between primary and unique keys mean one thing — queries involving table joins through unique keys require that you pay attention to `NULL` values.

In the *ADDRESS* example shown above, an index called `pk_address` was created on the primary key composed of `address_id`, `address_type`, and `address_seriesno`. An index called `uk_id_zip` was created on the unique key composed of `address_id` and `address_pcode` (i.e., in this example no two rows in the *ADDRESS* table may have the same combination of `address_id` and zip code). Note, too, that the *NAME* table includes a non-unique index based on the data columns `name_last` and `name_first`. Obviously, two people can have the same names.

Constraint reports (see Figure 1-5 on page 8) allow you to distinguish primary keys from unique keys.

► When tuning a query, you'll frequently need to consider the *leading edge* of an index. The index report shown in Figure 1-4 includes a position indicator for data columns that appear in an index. For example, the `pk_address` index includes `address_id` in position 1, `address_type` in position 2, and `address_seriesno` in position 3. The full index includes data columns in positions 1, 2, and 3. The leading edge of the index, however, only includes data columns in position 1 or in positions 1 and 2.

If you construct a query without referring to the columns in the leading edge of an index, the query cannot use the index. Improving performance often requires that you rewrite the query to take advantage of the leading edge of an index.

See Also

1. A discussion of the special problems presented to query writers by `NULL` values begins in Chapter 5 on page 144.
2. Data columns included in an index appear in the `sys` view `ALL_IND_COLUMNS`. The indexes themselves appear in `ALL_INDEXES`.
3. For other examples of primary keys, see Chapter 9 on page 254.

How do I find the constraints for a table?

I'm having trouble joining some of the tables needed for a query. How do I find out what primary and foreign keys exist for a table?

Approach

Run the `constrnt.sql` program listed in Appendix A (page 326). This produces a report identifying all the *integrity constraints* placed on a table. An integrity constraint is a rule that limits the values that may appear in one or more data columns in a table. Primary and foreign keys are only two types of integrity constraints that exist. Others identify unique keys and describe the check conditions enforced on individual data columns.

Figure 1-5 shows an example of the report produced by `constrnt.sql`.

```

Constraints on OR2.ADDRESS
Date: 12-MAY-1996      Time: 09:13
Report name: ADDRESS.cst
Page: 1
SQL name: constrnt.sql
    
```

constraint	type	search	referential constraint	status	column	pos
CHECK_LOC	C	address_loc = upper(address_loc)		ENABLED	ADDRESS_LOC	
NN_ADDRESSADDRESS_SERIESNO	C	ADDRESS_SERIESNO IS NOT NULL		ENABLED	ADDRESS_SERIESNO	
SYS_C00443	C	ADDRESS_ID IS NOT NULL		ENABLED	ADDRESS_ID	
SYS_C00444	C	ADDRESS_TYPE IS NOT NULL		ENABLED	ADDRESS_TYPE	
PK_ADDRESS	P			ENABLED	ADDRESS_ID ADDRESS_TYPE ADDRESS_SERIESNO	1 2 3
FK_ADDRESSADDRESS_TYPE	R		PK_CADDRESS	ENABLED	ADDRESS_TYPE	1
UK_ID_ZIP	U			ENABLED	ADDRESS_ID ADDRESS_PCODE	1 2

Figure 1-5: Sample constraint report produced by `constrnt.sql`.

Comments

▶ Different types of integrity constraints exist. Those coded C in the report identify *check constraints*, those coded P or U identify primary or unique keys, respectively, and those coded R identify

foreign keys (i.e., *referential constraints*).

➤ *Check constraints* place conditions on data columns. For example, each data column where `NULL` values are not permitted will have a constraint that prohibits `NULL` values. Check constraints can conveniently enforce many business rules. In Figure 1-5, the state address (`address_loc`) has a check condition that ensures that all state abbreviations appear in uppercase characters.

➤ A *referential constraint* identifies a *foreign key*, which is composed of one or more data columns that appear as a primary or unique key elsewhere in the database. Usually the primary or unique keys appear in other tables, but this need not be the case. To satisfy a referential constraint, one of two conditions must be met: (1) the values that appear in the foreign key must exist in the referenced primary or unique key, or (2) one or more of the data columns in the foreign key must be `NULL`.

In Figure 1-5, the `ADDRESS` table contains one foreign key (`fk_addressaddress_type`). This ensures that a value in the `address_type` data column must exist as a primary key (`pk_address`) in the table that maintains address codes (i.e., `CADDRESS`).

See Also

1. Integrity constraints placed on a table appear in the `SYS` view `ALL_CONSTRAINTS`. See the index under `SYS views` for examples using this view.

Query Standards

It's real easy to make a mess of your ad hoc queries. Yet, if you apply a few simple standards, you'll find queries much easier to construct, test, debug, and maintain.

This section discusses how to format a query to improve legibility and understanding, how to simplify a query so that its meaning is not lost in a maze of SQL, and how to document a query so that you or others have some hope for maintaining the SQL at a later time.

How do I format a query?

Sometimes when I need to revise an SQL query written weeks previously, I have trouble deciphering what the program does. Is there any way to structure a query so that it's easier to understand and maintain?

Approach

The Oracle engine can be very forgiving about some SQL syntax, making it possible to write valid queries that are equivalent to run-on sentences. Consider two versions of the same SQL program shown in Figure 1-6 and Figure 1-7. One uses a freeform unstructured approach; the other incorporates a few simple standards to structure the query.

```
column name format a30
select n1.name_id,n1.name_last||', '||n1.name_first
"name",address_city,address_loc
from address,name n1
where n1.name_seriesno = (select max(n2.name_seriesno) from
name n2 where n2.name_id = n1.name_id) and
address_id = n1.name_id and address_type = 'PR' order by 2;
```

Figure 1-6: Example of unstructured SQL.

The SQL programs themselves are identical, and either version produces the same report. But the unstructured version invites confusion, whereas the structured version makes the intent of the query much clearer. Revising an unstructured query of several hundred lines can be a daunting task if you return to the query weeks after it was first written.

No standards exist for structuring SQL queries. It would be convenient, in fact, if a utility existed that would parse a query and format it. Such a tool could produce a physically structured program. It also might convert all SQL and SQL*PLUS reserved keywords to uppercase characters.

```

COLUMN name FORMAT a30
SELECT
  n1.name_id,
  n1.name_last||', '||n1.name_first "name",
  address_city,
  address_loc
FROM
  address,
  name n1
WHERE
  n1.name_seriesno =
    (SELECT MAX(n2.name_seriesno)
     FROM name n2
     WHERE n2.name_id = n1.name_id)
  AND address_id = n1.name_id
  AND address_type = 'PR'
ORDER BY 2;

```

Figure 1-7: Example of a structured query.

Comments

- ▶ Highlight each of the six possible clauses (i.e., SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY) by placing each on a separate line at the left margin. Indent everything else.
- ▶ Place each data column and expression that appears in the SELECT and GROUP BY clauses on a separate line. Place each table in the FROM clause on a separate line.
- ▶ Structure all subqueries using the same standards that apply to the main query, or, as shown above, compress the subqueries slightly to conserve space but retain the meaning.
- ▶ Indent the WHERE clause to highlight every table join. Keep together all join criteria related to a single table, with one condition per line.
- ▶ Whenever possible, use numbers in the ORDER BY clause. These numbers refer to the positions of the data columns or expressions in the SELECT clause. Of course, if you sort by a data column or expression that does not appear in the SELECT clause, then you'll need to use the column name or the expression.

See Also

1. Chapter 4 (beginning on page 107) discusses the principal clauses of the SELECT command.

How do I make queries easier to construct?

My queries are often a jumble of SQL and SQL*PLUS commands. How can I simplify the programs so that they're easier to write and understand?

Approach

Using good programming practices can improve your ability to write and maintain programs. Two concepts are key — modularize and reuse. That is, write smaller program components that can be employed in a variety of situations.

There are no standards that can be cited, but here are some practices that we've found helpful.

Shell and retrieval programs

Rather than writing queries as one long program, break them up into at least two programs — one that formats the report and a second that actually retrieves the data. We call the first type of program the *shell* (or *main*) program because it calls the second data retrieval program.

Figure 1-1 on page 2 showed one way to explore a database through a frequency distribution of table owners. The shell program that generated the report is shown in Figure 1-8.

```

SET TERMOUT OFF
START &&object.\clears
START &&object.\printer
START &&object.\date
START &&object.\time
START &&object.\owncol

TTITLE 'Database tables and views by owner' skip 1 -
      LEFT 'Date: ' xDate '      Time: ' xTime -
      RIGHT 'Page: ' FORMAT 999 SQL.PNO SKIP 1 -
      LEFT 'Report name: owners.lis' -
      RIGHT 'SQL name: owners.sql' SKIP 2;

SPOOL &&query.\owners.lis
START &&tool.\xowners
SPOOL OFF

START &&object.\clears
START &&object.\screen

```

Figure 1-8: Example of a shell program.

Note the considerable use of other smaller SQL and SQL*PLUS programs, acting much like procedures in procedural programming languages. Modularizing the shell makes it much easier to see what's actually

happening in the program. Appendix A describes in detail what function each of these modular programs performs.

Comments

- ▶ Each `START` command executes a separate `SQL` or `SQL*PLUS` program. These programs are all stored in a directory whose name is stored in the `SQL*PLUS` variable `&&object` defined at the time of login. The programs clean up any existing mess made by previous queries, specify printer characteristics, and retrieve the date and time so that they can be used in the report title.
- ▶ The data retrieval program gets executed in this statement.
- ▶ Prior to exiting from the shell, the cleanup program is rerun, and settings appropriate for work at the screen get activated.

PL/SQL functions

By building reusable PL/SQL functions, you also can modularize and simplify ad hoc queries. Figure 1-7 on page 11 showed a query for a simple name and address report. The query included a correlated subquery that retrieved the most recent name for any individuals with name changes.

This correlated subquery surely will recur in other programs. You could reenter the subquery each time it's needed, but you also could write a PL/SQL function that has the same effect (see Figure 1-9).

```
CREATE OR REPLACE FUNCTION name_now
  (id VARCHAR2)
RETURN NUMBER
AS
  name_now NUMBER(2);

CURSOR main_cursor (p_id VARCHAR2) is
  SELECT MAX(name_seriesno)
  FROM name
  WHERE name_id = p_id;

BEGIN
  OPEN main_cursor(id);
  FETCH main_cursor INTO name_now;

  IF main_cursor%NOTFOUND THEN
    name_now := NULL;
  END IF;

  CLOSE main_cursor;
  RETURN name_now;
END;
/
```

Figure 1-9: Function that identifies most recent name data.

With the `name_now` function, you can revise the SQL query in Figure 1-7. This substantially reduces the `WHERE` clause criteria, making the program simpler and easier to understand (see Figure 1-10).

```
COLUMN name FORMAT A30
SELECT
    name_id,
    name_last||', '||name_first "name",
    address_city,
    address_loc
FROM
    address,
    name
WHERE
    name_seriesno = name_now(name_id)
    AND address_id = name_id
    AND address_type = 'PR'
ORDER BY 2;
```

Figure 1-10: Revised SQL from Figure 1-7 using `name_now` function.

Comments

► The PL/SQL function `name_now` replaces a correlated subquery to retrieve the most recent name information; one line of code replaces several lines. Conceptually, the query suddenly becomes less complex. Choosing a meaningful function name also makes it easier for people to understand its purpose.

Views

Carefully constructed views also can improve your ability to write queries easily. The danger with views is that they tend to grow uncontrollably. Information systems (IS) staff create a view, it gets used for a time, but then users begin requesting revisions. The request generally starts this way: “View xyz has most data columns I need, but it’s missing one thing.” So IS staff add another table join into the view. Over time this incremental growth deteriorates query performance because, even if a specific query does not require all tables in the view, all table joins still occur.

It’s far better to construct specialized views that perform very narrow functions and then employ these views in data queries. Here’s an example. The query in Figure 1-10 will return multiple addresses with `address_type` of PR (permanent residence). Suppose you only wished to see the most recent permanent address as identified by the last row added or updated. This requires a correlated subquery that makes the query messy. However, if you create a view of the most recent addresses, the correlated subquery gets included in the view and need not appear in the

query. Figure 1-11 shows an example of such a view called *RECENTADDRESS*. Now you need only join to the view and specify an *address_type*.

```
CREATE VIEW RecentAddress
  (RA_id,
   RA_type,
   RA_street,
   RA_city,
   RA_loc,
   RA_pcode,
   RA_country,
   RA_seriesno,
   RA_datestamp)
AS
SELECT * FROM address a1
WHERE
  a1.address_datestamp =
  (SELECT MAX(a2.address_datestamp)
   FROM address a2
   WHERE a2.address_id = a1.address_id
        AND a2.address_type = a1.address_type);
```

Figure 1-11: View that returns most recent addresses.

Comments

► The correlated subquery returning data on most recent addresses appears in the *RECENTADDRESS* view. Using this view means that the subquery need not appear in the query itself, making the query simpler and easier to understand.

One very nice feature of Oracle7 Release 7.2 and later releases is that subqueries can appear in the *FROM* clause of queries. In effect, this allows the query writer to construct data views “on the fly” instead of relying on IS to develop and maintain views. Later chapters provide several examples where such implicit views are used.

SQL segments

Sometimes you’ll find yourself writing a query that uses nearly the same code that you wrote for a previous program. If you save and generalize these code fragments, then they’re available for use in later queries. In a previous book by one of us, these code segments were called *SQL objects* and played a central role in simplifying ad hoc queries. To better distinguish *SQL objects* from other Oracle objects (especially when Oracle8 arrives), we now call these code segments *SQL segments*. The American Heritage Dictionary defines a segment as “any of the parts into which something can be divided.” There is also a nice analogy to biology where

segments mean differentiated subdivisions of an organism. Any sql query has these subdivisions, defined most obviously by the major clauses but then further defined as clause fragments.

Query writers can use SQL segments in many creative ways. A segment might define an adjective commonly used in the business (e.g., an “international” student at a college). SQL segments also can provide standard table joins based on a desired effect. For example, each of the following situations requires a different join to an address table: (1) return all addresses of a specified type if they exist, (2) return only the most recent address of a specified type if it exists, and (3) return the most recent address of a specified type if it exists or return NULL values if no such address exists. The join criteria used with the address table cannot be predetermined; you need to understand the intent of the query. By building an SQL segment for each situation, you can retrieve the appropriate one when constructing a new query.

SQL segments also find considerable use in defining business populations. Figure 1-12 shows an SQL query that counts the number of applicants who were accepted for college admissions.

```

SELECT COUNT(*)
FROM
  applicant
WHERE
  applicant_term = '199709'
  AND applicant_level = 'UG'
  AND applicant_college = 'AS'
  AND EXISTS
  (SELECT 'x'
   FROM
     cdecision,
     decision
   WHERE
     decision_id = applicant_id
     AND decision_applicant_term = applicant_term
     AND decision_applicant_seriesno =
       applicant_seriesno
     AND cdecision_code = decision_code
     AND cdecision_accept_ind = 'Y');

```

Figure 1-12: Applicants accepted for admission.

Comments

- ▶ The query includes codes for the term (e.g., Fall 1997), level (e.g., Undergraduate), and college (e.g., Arts & Sciences).
- ▶ A messy correlated subquery captures the business rules that identify an applicant accepted for admission. If the criteria EXISTS, then the applicant qualifies as accepted for admission.

By creating a simple PL/SQL function and generalizing the hardcoded values, you can create an SQL segment that defines accepted applicants. This segment can be reused in subsequent queries. Figure 1-13 shows one possible version of the segment.

```
applicant_term = '&&term'  
  AND applicant_level = '&&level'  
  AND applicant_college = '&&college'  
AND accepted(applicant_id, applicant_term,  
  applicant_seriesno) = 'Y'
```

Figure 1-13: SQL segment that defines an accepted applicant.

Comments

- ▶ The SQL segment replaces each of the hardcoded values that appeared in Figure 1-12 with a substitution variable. At runtime, the query prompts for these variables. This segment can now be used in queries that specify different terms, levels, and colleges.
- ▶ The PL/SQL `accepted` function (not shown) requires three arguments — an applicant's ID number, the term, and a sequence number if more than one application was submitted for a given term. The correlated subquery in Figure 1-12 makes clear why these three values must be specified.

The SQL segment shown in Figure 1-13 is more versatile, more compact, and more easily understood than the query in Figure 1-12.

See Also

1. Appendix A (beginning on page 313) includes several examples of shell programs. Also see Chapter 6 (Figure 6-27 on page 207).
2. See the discussion later in this chapter on page 24 showing the use of a `login.sql` file to define path names at time of login.
3. See the index under PL/SQL *functions* for examples of other PL/SQL functions used in this book.
4. Chapter 10 (Figure 10-15 on page 305) shows another example of view creation.
5. Report templates make wonderful SQL segments. Save the templates and reuse them whenever you need a similar report. Chapter 6 (beginning on page 173) discusses many such templates.

How do I place comments in a query?

My queries would be easier to understand if I could put comments in them to document the intent of a particular section. Is this possible?

Approach

Including documentation is another good programming practice that you should incorporate into your queries. Programs with ample comments are much easier to maintain than programs without comments.

Figure 1-2 on page 3 listed the tables for a specified owner. The shell program that produced this report appears in Figure 1-14.

```

SET TERMOUT OFF
/*
  sql:      tables.sql
  date:     22-mar-1994
  author:   glewis
  use:      list of tables for a specified owner
*/

START &&object.\clears      -- clear break, compute, column
START &&object.\printer     -- report in landscape 8pt
START &&object.\date        -- retrieve date
START &&object.\time        -- retrieve time

START &&object.\getowner    -- prompt for table owner
START &&tool.\deftabl       -- define xReport (report name)

TTITLE 'Tables for owner: ' xOwner SKIP 1 -
      LEFT 'Date: ' xDate '      Time: ' xTime -
      RIGHT 'Page: ' FORMAT 999 SQL.PNO SKIP 1 -
      LEFT 'Report name: ' xReport -
      RIGHT 'SQL name: tables.sql' SKIP 2;

SPOOL &&query.\&&xReport
  START &&tool.\xtables -- data retrieval program
SPOOL OFF

START &&object.\clears      -- cleans up after program run
START &&object.\screen      -- sets vars for work at monitor

```

Figure 1-14: Shell program for table report.

Comments

- ▶ SQL comment delimiters `/*` and `*/` are useful when you want to provide multiple lines of documentation. This type of comment may appear separately (as here), on an SQL*PLUS command line, or embedded within an SQL command.
- ▶ ANSI/ISO comments begin with a double hyphen (`--`). They may not span multiple lines but can appear on separate lines by

themselves or embedded within SQL commands. They may not, however, be used on the same line as an SQL*PLUS command.

Data for the table report get retrieved by the program xtables.sql. Figure 1-15 lists this program and illustrates several types of comments.

```

REM sql:      xtables.sql
REM author:   glewis
REM date:    15-may-1996
REM use:     retrieve data for tables report
REM
-- wrap comments at end of words
COLUMN comments FORMAT a40 WORD_WRAP
-- skip line between table types
BREAK ON type SKIP 1
SELECT
  t.table_name "name",
  t.table_type "type",
  c.comments "comments"
FROM
  /* retrieves only those tables
   available to the user */
  ALL_TAB_COMMENTS c,
  ALL_CATALOG t
WHERE
  t.owner = '&&xowner' -- xowner defined in shell
  AND c.owner = t.owner
        AND c.table_name = t.table_name
        AND c.table_type = t.table_type
ORDER BY 2, 1
;

```

Figure 1-15: Program that retrieves data for table report.

Comments

- The SQL*PLUS REMARK command also can be used for comments. Each must appear on a separate line and cannot be embedded within an SQL command.
- An ANSI/ISO comment cannot appear on the same line as an SQL*PLUS command.
- SQL comments can span multiple lines even within a query.

Comments cannot appear on the same line as the SQL terminator, which is usually the semicolon unless it's been reset to another character.

See Also

1. The SQL*PLUS SET command controls many system variables, including the SQL terminator. See the discussion on page 22 that describes how the SET command affects the SQL*PLUS environment.

How do I create default column headings for reports?

I find it time consuming to create column headings and formats each time I write a query. It's also confusing because I'll give a column one heading in a report and a second heading in another report. Users sometimes ask how the two columns differ. Can I create default column headings and formats that would be in effect automatically for my reports?

Approach

It makes sense to eliminate the repetitious creation of column formats and to standardize headings so that they're consistent from report to report. Fortunately, this is easy to do.

Typically, query writers concentrate in one section of a large database, perhaps in one functional area like human resources or even in a subset of tables within a functional area. The same data columns appear frequently in reports. For those data columns which you use most often, create a `COLUMN` command that defines a heading and specifies a format. Then place the `COLUMN` commands in an `SQL*PLUS` file that you can start from the shell of each query. This effectively creates default column information for each query. It relieves you from creating the headings and formats repeatedly and also standardizes the appearance of your reports.

Whenever you write a query that uses new data columns or expressions, add the `COLUMN` commands to your `SQL*PLUS` command file. Over time, you'll build an impressive collection of default headings and formats. Figure 1-16 illustrates how default column settings can be created.

```
/* name */
COLUMN name_id HEADING 'id'
COLUMN name_last HEADING 'last|name'
COLUMN name_first HEADING 'first|name'
COLUMN name_middle HEADING 'middle|name' FORMAT a6
COLUMN name_seriesno HEADING 'name|seqnum' FORMAT 999999

/* address */
COLUMN address_id HEADING 'id'
COLUMN address_type HEADING 'address|type' FORMAT a7
COLUMN address_street HEADING 'street|address'
```

Figure 1-16: Command file that creates default column headings and formats.

See Also

1. See Appendix A on page 315 for another discussion of how to create default column headings.

How do I run multiple reports without format changes interfering?

Sometimes when I run several queries in one SQL*PLUS session, my reports contain weird formatting problems. This is particularly true when I intermingle reports that display at the screen with reports intended for the printer. How can I prevent this from happening?

Approach

Typically, reports display in one of two places — at the screen or at the printer (or a file that will be printed). The SQL*PLUS environments suitable for these two report types differ considerably. Thus, as you run several queries, any changes you make to system variable settings, column headings and formats, BREAK actions, and COMPUTE actions all compound. With interactions possible, you can get some bizarre results.

Unfortunately, there is no single SQL*PLUS command or system setting that reestablishes the defaults in operation when you first login to the SQL*PLUS environment. You could always exit and login between each query, but this quickly becomes tiresome. Therefore, you're left with the task of ensuring that you clean up after every query.

The shell program in Figure 1-8 on page 12 coordinates several programs to accomplish the cleanup function. The `clears.sql` program gets started before and after the main query. This clears all BREAK, COMPUTE, and COLUMN commands. The `owncol.sql` program reestablishes default column headings and formats. The `printer.sql` program runs prior to the main query, disables system settings appropriate for screen displays (e.g., where PAUSE is ON), and enables settings appropriate for print displays. Then, after the query completes, `screen.sql` disables the settings appropriate for print displays and reenables settings for the screen.

These programs will help avoid strange formatting effects in reports, but they won't prevent them from occurring. You'll still need to be aware whenever you manually override a system variable or define a user variable to create a special effect in a report. When exiting the query, you'll want to undo the special actions, for example, by resetting the altered system variable to its default or by undefining a user variable.

See Also

1. Beginning on page 313, Appendix A shows the program listings for the utility programs `clears.sql`, `date.sql`, `time.sql`, `owncol.sql`, `printer.sql`, and `screen.sql`.

SQL*PLUS Environment

When you write queries, your home base is normally a monitor and the SQL*PLUS environment. To reflect your preferred way of working, you can customize this environment. Some of the changes may be cosmetic, but others serve functional roles.

This section describes how to change the SQL prompt to provide more information than the default (SQL>). It also illustrates ways that the SQL*PLUS environment can be customized automatically at login.

How do I change the SQL prompt?

I work with several different Oracle databases. Is there some way I can modify the SQL prompt to identify which Oracle instance I'm working with?

Approach

By default, the SQL prompt is set as SQL>, but you can define it many other ways using the SQL*PLUS command SET SQLPROMPT 'text'. You also can add the current time to the SQL prompt merely with SET TIME ON.

The program in Figure 1-17 shows one way to retrieve the current Oracle instance, store it in a user-defined variable, and then use that variable to set the SQL prompt. If, for example, the instance is a production database called PROD, then instead of SQL> the prompt becomes PROD>.

```

COLUMN instance NEW_VALUE xInstance NOPRINT
SELECT
    SUBSTR(GLOBAL_NAME,1,INSTR(GLOBAL_NAME,'.')-1) "instance"
FROM GLOBAL_NAME;
SET SQLPROMPT &&xInstance.>

```

Figure 1-17: Program that redefines SQL prompt as the current Oracle instance.

Comments

► The program defines a user variable called xInstance that contains the name of the current Oracle instance. If you know the value for a user variable, the DEFINE command is the most direct method of defining a variable. However, when the value of a user variable must be retrieved from the database, other methods are required. The procedure shown here is one such method. It

uses a simple `SELECT` query to return the instance name. The `NEW_VALUE` option in the `COLUMN` command creates an implicit `DEFINE` of the user variable `xInstance`.

▶ The name of the Oracle instance is extracted from `GLOBAL_NAME`, but it is also available in other views like `V$DATABASE` if you have the appropriate access rights. You could save and then run the program in Figure 1-17 whenever you wished to change the SQL prompt, or you could add the program to your `login.sql` file so that it got executed automatically whenever you logged into the `SQL*PLUS` environment. The next section discusses the `login.sql` file.

▶ The `SET` command changes the SQL prompt to the value contained in `xInstance` followed by the greater than (`>`) sign. The double ampersands (`&&`) identify the beginning of the substitution variable `xInstance`. The period (`.`) identifies the end of the substitution variable.

See Also

1. You'll find considerable use for implicitly defined user variables when writing ad hoc queries. It's the technique used, for example, to capture the date and time that datestamp a report. See the index under *user variables* for several examples.
2. `GLOBAL_NAME` is another `SYS` view. See the index under `SYS views` for examples of its use.
3. Beginning on page 36, Chapter 2 describes substitution variables and discusses several problems that arise with their use.
4. Another `SET` command (`SET CONCAT`) controls the character used to end a substitution variable. See the index under `SET commands`.

How do I ensure a consistent look to the SQL*PLUS environment?

Each time I login to the SQL*PLUS environment I need to reset some of the default system variables so that they work the way I prefer. For example, I always SET PAUSE ON so that output displayed on the monitor appears one screen at a time. Is there a way to automatically configure the SQL*PLUS environment to my liking?

Approach

SQL*PLUS supports something called a *site profile*, which is an SQL*PLUS command file generally named glogin.sql that allows the database administrator to create a default SQL environment for all users. The default name and location are system dependent.

SQL*PLUS also supports a *user profile*, which is a command file named login.sql that's run after glogin.sql. Each time you enter the SQL*PLUS environment, login.sql gets executed from your current directory if it exists or from a system-dependent path if it's not in your current directory. The login.sql program may contain SQL, SQL*PLUS, or PL/SQL commands that allow you to customize the SQL*PLUS environment to your choosing.

Figure 1-18 shows a simple login.sql file. Note that it uses the SQL*PLUS START command to execute other programs. By modularizing login.sql in this manner, it is easier to understand and easier to revise.

```
SET TERMOUT OFF
DEFINE object = d:\orawin95\komenda\objects
DEFINE tool = d:\orawin95\komenda\tools
DEFINE query = d:\orawin95\komenda\query

START &&object.\getuser
START &&object.\getinstc
SET SQLPROMPT &&xUser. @&&xInstance.>

DEFINE _editor = c:\windows\notepad.exe

START &&object.\screen
```

Figure 1-18: Example of a login.sql automatically run at login.

Comments

► The login program first defines several system-dependent local path names. This has two advantages: (1) by making a single change in the path definition in login.sql, you can change the location of commonly used utility programs without making

changes to each individual SQL query, and (2) if the defined names are short, they're much more convenient to use at the SQL prompt than longer path names.

▶ The SQL prompt gets changed from SQL> to a string formed by concatenating the user name, the @ character, and the Oracle instance. If, for example, the user is glewis and the instance is PROD, then the SQL prompt becomes glewis@prod>. Note in the SET SQLPROMPT command that the two periods each identify the end of a user-defined variable (xUser and xInstance).

▶ The login program defines a default editor that will be used whenever you use the EDIT command at the SQL prompt. This allows you to use an editor of your own choosing. In Figure 1-18 the default editor is defined as the Windows Notepad editor. The user variable must be named _editor (or _EDITOR) in the DEFINE command.

getuser.sql

This program determines the user name for the person making the SQL*PLUS login. The value for the user name then gets used to change the SQL prompt (see Figure 1-19).

```
COLUMN user NEW_VALUE xUser NOPRINT  
SELECT LOWER(user) "user" FROM DUAL;
```

Figure 1-19: Program getuser.sql that retrieves current user name.

Comments

▶ The NEW_VALUE option in the COLUMN command coordinates with the SELECT query to implicitly define a variable called xUser that contains the current user name. This procedure is exactly analogous to the implicit DEFINE of the Oracle instance in Figure 1-17 on page 22. It's a somewhat convoluted way to define a user variable, but if the value must be retrieved from the database, this procedure conveniently captures the value.

getinstc.sql

This program retrieves the current Oracle instance, using the same technique shown in Figure 1-17 on page 22. The instance name then gets used along with the user name to set the SQL prompt (see Figure 1-20).

```
COLUMN instance NEW_VALUE xInstance NOPRINT
SELECT
    LOWER(SUBSTR(GLOBAL_NAME,1,INSTR(GLOBAL_NAME,'.')-1))
    "instance"
FROM GLOBAL_NAME;
```

Figure 1-20: Program getinstc.sql that retrieves current Oracle instance.

screen.sql

In the last step, login.sql runs a program called screen.sql that sets several system variables so that the SQL*PLUS environment is suitable for working from the monitor. Many of these settings reflect personal preference, so you may wish to change them to match your own preferences (see Figure 1-21).

```
SET PAUSE "ENTER to continue ..."  
SET PAUSE ON  
SET LINESIZE 80  
SET PAGESIZE 24  
TTITLE OFF  
SET ESCAPE OFF  
SET ECHO ON  
SET TERMOUT ON
```

Figure 1-21: Program screen.sql that sets system variables for monitor work.

Comments

- ▶ When working from the monitor, you generally want reports to pause at the end of the screen as opposed to scrolling from begin to end. The screen.sql program sets the PAUSE on and provides a prompt to the user that the next page of the report may be seen by pressing the enter (or return) key.
- ▶ Page and line sizes depend on the monitor being used. The setting of 80 characters by 24 lines is common for monitors. But if you're using a monitor with other capabilities, then revise these settings.
- ▶ SQL*PLUS uses the backslash (\) as an escape character that says essentially "the next character in the string is valid, so ignore any special uses it may have." With ESCAPE set on, a command of DEFINE x = 'AT\&T' will define x as 'AT&T'. But with ESCAPE set off, SQL*PLUS will assume that the ampersand identifies the beginning of a substitution variable. If you work on operating

systems that use the backslash as a valid character in path names, then it's important to redefine the escape character or to set it off. The default setting is off, but including a `SET ESCAPE OFF` in `screen.sql` ensures this to be the case in SQL*PLUS sessions where you've previously activated the escape character.

The counterpart to `screen.sql` that controls the SQL*PLUS environment when you want to print reports is illustrated in the shell program in Figure 1-8 on page 12. A listing of the `printer.sql` program appears in Appendix A (page 315). It turns off pausing so that the report spools continuously from begin to end; sets the `PAGESIZE` and `LINESIZE` characteristics to suit the printer, font, and orientation you've chosen; activates title lines; and deactivates the echoing of the SQL and any summary timing statistics that are more appropriate when working at the monitor.

See Also

1. The `SET` command controls many features of the SQL*PLUS environment. See the index under `SET commands` for other illustrations of its use.

Executing Queries

It can be frustrating when you've tested, debugged, and tuned a query only to experience problems when attempting to generalize the query for wider use. This section discusses problems that commonly occur when passing values to a query through command line arguments.

How do I pass values to a query?

Sometimes when I run a query by using arguments in the @ or START command, I get an error message. Obviously, the values specified in the arguments are not being passed correctly. How can I avoid this problem?

Approach

The START and @ commands allow optional arguments that get passed to parameters in a query. Elements in the argument list are separated by blank spaces. For example, "START test.sql 15 A" passes the values 15 and A to parameters &1 and &2, respectively, in a query called test.sql.

Argument errors depend on several factors: (1) how the arguments are specified, (2) how the query is written, and (3) whether you're running the query from the SQL*PLUS environment or from the operating system.

Specifying arguments

Blank spaces commonly produce errors when used in argument lists for any reason other than as a delimiter. The confusion probably occurs because blanks can be included when a user is prompted for data values at runtime. Figure 1-22 illustrates this interaction in a simple query prompting for ID numbers.

```
SQL> START test5
SQL> SELECT * FROM name WHERE name_id IN &&1;

Enter value for 1: ('a330', 'a505')
old 1: SELECT * FROM name WHERE name_id IN &&1
new 1: SELECT * FROM name WHERE name_id IN ('a330', 'a505')
```

id	last name	first name	middle name	name seqnum
a505	Carrington	Thomas	J	1
a330	Vincent	Caroline		1
a330	Healey	Caroline	V	2

Figure 1-22: Query where user is prompted for data values.

Comments

▶ When prompted for a data value, you can use embedded blanks, and they produce valid SQL.

However, note what happens in Figure 1-23 when you enter the ID numbers as arguments from the SQL command line.

```
SQL> START test5 ('@330', '@505')
SQL> SELECT * FROM name WHERE name_id IN &&1;

old 1: SELECT * FROM name WHERE name_id IN &&1
new 1: SELECT * FROM name WHERE name_id IN ('@330',
      SELECT * FROM name WHERE name_id IN ('@330',
      *
ERROR at line 1:
ORA-00936: missing expression
```

Figure 1-23: Query where argument improperly includes an embedded blank.

Comments

▶ When used in arguments from the SQL command line, embedded blanks delimit argument values. The error occurs because the blank space in the argument makes SQL*PLUS believe that you've passed two data values — and the value it believes is &&1 produces invalid SQL. You'll need to enclose any arguments containing blanks within double quotes to avoid errors.

Type of query

How you write a query influences how the argument must be specified. Some queries require arguments that are more easily specified than others. Consider a simple query in which the user must specify an address type. Figure 1-24 shows one method of specifying the substitution variable. Note that the substitution variable is embedded in single quotes.

```
SQL> START test9 BU
SQL> SELECT * FROM address WHERE address_type = '&&1';

old 1: SELECT * FROM address WHERE address_type = '&&1'
new 1: SELECT * FROM address WHERE address_type = 'BU'
```

id	address type	street address	city address	state/ prov	postal code
@B&N	BU	3501 Western Ave	Cambridge	MA	02140
@330	BU	McArthur and Phillips	Westchester	CO	80312

Figure 1-24: Query where character arguments are easily specified.

Comments

▶ When passing character data values to queries, structure the query so that the argument does not need a single quote in position 1. In this example the substitution variable is embedded in single quotes and the argument that gets passed is simple (i.e., BU).

Figure 1-25 shows exactly the same query with a slightly different way of handling the substitution variable. Instead of being embedded in single quotes, the substitution variable stands by itself. This method requires the user to pass the single quotes to avoid an error. Note that the argument that produces a valid SQL query is more complicated than the one in Figure 1-24. With the additional complication, it is harder to remember the correct syntax to use when passing arguments and therefore easier to generate errors. Choose the simpler method in Figure 1-24; it will make generalizing queries more enjoyable.

```
SQL> START test10 'BU'
SQL> SELECT * FROM address WHERE address_type = &&1;

old 1: SELECT * FROM address WHERE address_type = &&1
new 1: SELECT * FROM address WHERE address_type = BU
SELECT * FROM address WHERE address_type = BU
                                *
ERROR at line 1:
ORA-00904: invalid column name

SQL> START test10 "BU"
SQL> SELECT * FROM address WHERE address_type = &&1;

old 1: SELECT * FROM address WHERE address_type = &&1
new 1: SELECT * FROM address WHERE address_type = 'BU'
```

id	address type	street address	city address	state/ prov	postal code
aB&N	BU	3501 Western Ave	Cambridge	MA	02140
a330	BU	McArthur and Phillips	Westchester	CO	80312

Figure 1-25: Query where character arguments are difficult to specify.

Comments

▶ Revising the query so that the substitution variable is not embedded in single quotes complicates the argument. Note that when 'BU' is used as the argument, an error occurs because the single quotes get stripped from the argument before insertion into the program. This produces invalid SQL and generates error ORA-00904 (invalid column name) because the parsing process

believes that BU is a data column and cannot locate it in the tables that appear in the FROM clause.

▶ To enter a valid argument, you'll need to use "BU". The double quotes get stripped before insertion in the query, and this produces valid SQL. It's easier just to write the query as shown in Figure 1-24 on page 29 and then use BU as the argument.

Command level

You can start SQL queries from either the SQL*PLUS environment or from the operating system command line. The syntax differs slightly when you use the operating system form. In general, the syntax is `sqlplus uid/pwd @qname [arg . . .]`, where `uid` is the username, `pwd` is the password, and `qname` is the query command file started with the `@` command. Note, that in some systems the executable may not be named `sqlplus` (e.g., it may be `plus32` or some other variation).

Errors generated from improperly specified arguments can be minimized if you again pay attention to how you write the SQL query. Try to avoid writing parameters so that the first character in the data value that you want to pass to the query is a single quote. Figure 1-26 shows one way to pass an argument from an operating system command line that only invites trouble.

```
D:\plus32 uid/pw @test6 "'@330','@505'"
SQL> SELECT * FROM name WHERE name_id IN (&&1);

old 1: SELECT * FROM name WHERE name_id IN (&&1)
new 1: SELECT * FROM name WHERE name_id IN (@330)

SELECT * FROM name WHERE name_id IN (@330)
                                *
ERROR at line 1:
ORA-00936: missing expression

D:\plus32 uid/pw @test6 "\"@330','@505'\""
SQL> SELECT * FROM name WHERE name_id IN (&&1);

old 1: SELECT * FROM name WHERE name_id IN (&&1)
new 1: SELECT * FROM name WHERE name_id IN ('@330','@505')
```

id	Last name	first name	middle name	name seqnum
@505	Carrington	Thomas	J	1
@330	Vincent	Caroline		1
@330	Healey	Caroline	V	2

Figure 1-26: Query requiring complex argument from operating system.

Comments

- ▶ Passing character arguments from the operating system command line can be quite complicated unless you write the query properly. In this example, note that the user specified an argument that would have produced valid SQL if run from within SQL*PLUS. However, from the operating system command line (Windows 95 here) it produces an error. The problem occurs because the query, while valid, made it difficult to specify arguments. Avoid errors by rewriting the query so that the arguments are easier to construct.
- ▶ Correctly specifying the argument for the query requires that you use backslashes in nonintuitive ways when executing the query from a Windows 95 system prompt. The format for the argument is system-dependent. Unless you like a challenge, revise the query to accept a simpler argument.

Revising the SQL query so that the argument need not start with a single quote, as shown in Figure 1-27, makes passing the argument much easier and less likely to produce errors.

```
D:\plus32 uid/pw @test5 ('a303','a505')
SQL> SELECT * FROM name WHERE name_id IN &&1;
old 1: SELECT * FROM name WHERE name_id IN &&1
new 1: SELECT * FROM name WHERE name_id IN ('a330','a505')
```

id	last name	first name	middle name	name seqnum
a505	Carrington	Thomas	J	1
a330	Vincent	Caroline		1
a330	Healey	Caroline	V	2

Figure 1-27: Query requiring simple argument from operating system.

Comments

- ▶ This query simplifies the argument. Note that no double quotes or backslashes are required.

See Also

1. All the examples in this section used the default system setting which lists the text of a SQL statement containing a substitution variable. Since the default lists the SQL statement both before and after making the substitution, you get the statement listed twice. You can

avoid these listings with `SET VERIFY OFF`. For an example, see Chapter 2 (Figure 2-9 on page 50).

2. Properly using single quotes in queries can be tricky. For another example, see Chapter 2 (Figure 2-3 on page 42) to see the machinations needed during a query to concatenate a single quote to a string.

