Lab 5. Metadata Queries

This lab presents recipes that allow you to find information about a given schema. For example, you may wish to know what tables you've created or which foreign keys are not indexed. All of the RDBMSs in the lab provide tables and views for obtaining such data. The recipes in this lab will get you started on gleaning information from those tables and views. There is, however, far more information available than the recipes in this lab can show. Consult your RDBMSs documentation for the complete list of catalog or data dictionary tables/views.

TIP

For purposes of demonstration, all the recipes in this lab assume the schema name SMEAGOL.

5.1. Listing Tables in a Schema

PROBLEM

You want to see a list all the tables you've created in a given schema.

SOLUTION

The solutions that follow all assume you are working with the SMEAGOL schema. The basic approach to a solution is the same for all RDBMSs: you query a system table (or view) containing a row for each table in the database.

DB2

Query SYSCAT.TABLES:

```
select tabname
  from syscat.tables
 where tabschema = 'SMEAGOL'
```

Oracle

Query SYS.ALL_TABLES:

```
select table_name
  from all_tables
 where owner = 'SMEAGOL'
```

PostgreSQL, MySQL, and SQL Server
Query INFORMATION_SCHEMA.TABLES:

```
select table_name
  from information_schema.tables
 where table_schema = 'SMEAGOL'
```

DISCUSSION

In a delightfully circular manner, databases expose information about themselves through the very mechanisms that you create for your own applications: tables and views. Oracle, for example, maintains an extensive catalog of system views, such as ALL_TABLES, that you can query for information about tables, indexes, grants, and any other database object.

TIP

Oracle's catalog views are just that, views. They are based on an underlying set of tables that contain the information in a very user-unfriendly form. The views put a very usable face on Oracle's catalog data.

Oracle's system views and DB2's system tables are each vendor-specific. PostgreSQL, MySQL, and SQL Server, on the other hand, support something called the information schema, which is a set of views defined by the ISO SQL standard. That's why the same query can work for all three of those databases.

5.2. Listing a Table's Columns

PROBLEM

You want to list the columns in a table, along with their data types, and their position in the table they are in.

SOLUTION

The following solutions assume that you wish to list columns, their data types, and their numeric position in the table named EMP in the schema SMEAGOL.

DB2
Query SYSCAT.COLUMNS:

```
select colname, typename, colno
  from syscat.columns
 where tabname   = 'EMP'
   and tabschema = 'SMEAGOL'
```

Oracle
Query ALL_TAB_COLUMNS:

```
select column_name, data_type, column_id
  from all_tab_columns
 where owner      = 'SMEAGOL'
   and table_name = 'EMP'
```

PostgreSQL, MySQL, and SQL Server
Query INFORMATION_SCHEMA.COLUMNS:

```
select column_name, data_type, ordinal_position
  from information_schema.columns
 where table_schema = 'SMEAGOL'
   and table_name   = 'EMP'
```

DISCUSSION

Each vendor provides ways for you to get detailed information about your column data. In the examples above only the column name, data type, and position are returned. Additional useful items of information include length, nullability, and default values.

5.3. Listing Indexed Columns for a Table

PROBLEM

You want list indexes, their columns, and the column position (if available) in the index for a given table.

SOLUTION

The vendor-specific solutions that follow all assume that you are listing indexes for the table EMP in the SMEAGOL schema.

DB2
Query SYSCAT.INDEXES:

```
select a.tabname, b.indname, b.colname, b.colseq
  from syscat.indexes a,
       syscat.indexcoluse b
 where a.tabname   = 'EMP'
   and a.tabschema = 'SMEAGOL'
   and a.indschema = b.indschema
   and a.indname   = b.indname
```

Oracle

Query SYS.ALL_IND_COLUMNS:

```
select table_name, index_name, column_name, column_position
  from sys.all_ind_columns
 where table_name  = 'EMP'
   and table_owner = 'SMEAGOL'
```

PostgreSQL

Query PG_CATALOG.PG_INDEXES and INFORMATION_SCHEMA.COLUMNS:

```
select a.tablename,a.indexname,b.column_name
  from pg_catalog.pg_indexes a,
       information_schema.columns b
 where a.schemaname = 'SMEAGOL'
   and a.tablename  = b.table_name
```

MySQL

Use the SHOW INDEX command:

```
show index from emp
```

SQL Server

Query SYS.TABLES, SYS.INDEXES, SYS.INDEX_COLUMNS, and SYS.COLUMNS:

```
select a.name table_name,
       b.name index_name,
       d.name column_name,
       c.index_column_id
  from sys.tables a,
       sys.indexes b,
       sys.index_columns c,
       sys.columns d
where a.object_id = b.object_id
  and b.object_id = c.object_id
  and b.index_id  = c.index_id
  and c.object_id = d.object_id
  and c.column_id = d.column_id
  and a.name      = 'EMP'
```

DISCUSSION

When it comes to queries, it's important to know what columns are/aren't indexed. Indexes can provide good performance for queries against columns that are frequently used in filters and that are fairly selective. Indexes are also useful when joining between tables. By knowing what columns are indexed, you are already one step ahead of performance problems if they should occur. Additionally, you might want to find information about the indexes themselves: how many levels deep they are, how many distinct keys, how many leaf blocks, and so forth. Such information is also available from the views/tables queried in this recipe's solutions.

5.4. Listing Constraints on a Table

PROBLEM

You want to list the constraints defined for a table in some schema and the columns they are defined on. For example, you want to find the constraints and the columns they are on for table EMP.

SOLUTION

DB2

Query SYSCAT.TABCONST and SYSCAT.COLUMNS:

```
select a.tabname, a.constname, b.colname, a.type
  from syscat.tabconst a,
       syscat.columns b
where a.tabname   = 'EMP'
  and a.tabschema = 'SMEAGOL'
  and a.tabname   = b.tabname
  and a.tabschema = b.tabschema
```

Oracle
Query SYS.ALL_CONSTRAINTS and SYS.ALL_CONS_COLUMNS:

```
select a.table_name,
       a.constraint_name,
       b.column_name,
       a.constraint_type
  from all_constraints a,
       all_cons_columns b
where a.table_name      = 'EMP'
  and a.owner           = 'SMEAGOL'
  and a.table_name      = b.table_name
  and a.owner           = b.owner
  and a.constraint_name = b.constraint_name
```

PostgreSQL, MySQL, and SQL Server
Query INFORMATION_SCHEMA.TABLE_CONSTRAINTS and INFORMATION_
SCHEMA.KEY_COLUMN_USAGE:

```
select a.table_name,
       a.constraint_name,
       b.column_name,
       a.constraint_type
  from information_schema.table_constraints a,
       information_schema.key_column_usage b
where a.table_name      = 'EMP'
  and a.table_schema    = 'SMEAGOL'
  and a.table_name      = b.table_name
  and a.table_schema    = b.table_schema
  and a.constraint_name = b.constraint_name
```

DISCUSSION

Constraints are such a critical part of relational databases that it should go without saying why you need to know what constraints are on your tables. Listing the constraints on tables is useful for a variety of reasons: you may want to find tables missing a primary key, you may want to find which columns should be foreign keys but are not (i.e., child tables have data different from the parent

tables and you want to know how that happened), or you may want to know about check constraints (Are columns nullable? Do they have to satisfy a specific condition? etc.).

5.5. Listing Foreign Keys Without Corresponding Indexes

PROBLEM

You want to list tables that have foreign key columns that are not indexed. For example, you want to determine if the foreign keys on table EMP are indexed.

SOLUTION

DB2
Query SYSCAT.TABCONST, SYSCAT.KEYCOLUSE, SYSCAT.INDEXES, and SYSCAT.INDEXCOLUSE:

```
select fkeys.tabname,
       fkeys.constname,
       fkeys.colname,
       ind_cols.indname
  from (
select a.tabschema, a.tabname, a.constname, b.colname
  from syscat.tabconst a,
       syscat.keycoluse b
where a.tabname    = 'EMP'
  and a.tabschema  = 'SMEAGOL'
  and a.type       = 'F'
  and a.tabname    = b.tabname
  and a.tabschema  = b.tabschema
      ) fkeys
      left join
      (
select a.tabschema,
       a.tabname,
       a.indname,
       b.colname
  from syscat.indexes a,
       syscat.indexcoluse b
where a.indschema  = b.indschema
  and a.indname    = b.indname
      ) ind_cols
   on (fkeys.tabschema = ind_cols.tabschema
       and fkeys.tabname   = ind_cols.tabname
       and fkeys.colname   = ind_cols.colname )
where ind_cols.indname is null
```

Oracle

Query SYS.ALL_CONS_COLUMNS, SYS.ALL_CONSTRAINTS, and SYS.ALL_ IND_COLUMNS:

```
select a.table_name,
       a.constraint_name,
       a.column_name,
       c.index_name
  from all_cons_columns a,
       all_constraints b,
       all_ind_columns c
where a.table_name      = 'EMP'
  and a.owner           = 'SMEAGOL'
  and b.constraint_type = 'R'
  and a.owner           = b.owner
  and a.table_name      = b.table_name
  and a.constraint_name = b.constraint_name
  and a.owner           = c.table_owner (+)
  and a.table_name      = c.table_name (+)
  and a.column_name     = c.column_name (+)
  and c.index_name      is null
```

PostgreSQL

Query INFORMATION_SCHEMA.KEY_COLUMN_USAGE, INFORMATION_ SCHEMA.REFERENTIAL_CONSTRAINTS, INFORMATION_SCHEMA.COL-UMNS, and PG_CATALOG.PG_INDEXES:

```
select fkeys.table_name,
       fkeys.constraint_name,
       fkeys.column_name,
       ind_cols.indexname
  from (
select a.constraint_schema,
       a.table_name,
       a.constraint_name,
       a.column_name
  from information_schema.key_column_usage a,
       information_schema.referential_constraints b
 where a.constraint_name   = b.constraint_name
   and a.constraint_schema = b.constraint_schema
   and a.constraint_schema = 'SMEAGOL'
   and a.table_name        = 'EMP'
       ) fkeys
       left join
       (
select a.schemaname, a.tablename, a.indexname, b.column_name
  from pg_catalog.pg_indexes a,
       information_schema.columns b
 where a.tablename  = b.table_name
   and a.schemaname = b.table_schema
       ) ind_cols
    on (  fkeys.constraint_schema = ind_cols.schemaname
          and fkeys.table_name     = ind_cols.tablename
          and fkeys.column_name    = ind_cols.column_name )
 where ind_cols.indexname is null
```

MySQL

You can use the SHOW INDEX command to retrieve index information such as index name, columns in the index, and ordinal position of the columns in the index. Additionally, you can query INFORMATION_SCHEMA.KEY_COLUMN_USAGE to list the foreign keys for a given table. In MySQL 5, foreign keys are said to be indexed automatically, but can in fact be dropped. To determine whether a foreign key column's index has been dropped you can execute SHOW INDEX for a particular table and compare the output with that of INFORMATION_SCHEMA.KEY_COLUMN_USAGE.COLUMN_NAME for the same table. If the COLUMN_NAME is listed in KEY_COLUMN_USAGE but is not returned by SHOW INDEX, you know that column is not indexed.

SQL Server

Query SYS.TABLES, SYS.FOREIGN_KEYS, SYS.COLUMNS, SYS.INDEXES, and SYS.INDEX_COLUMNS:

```
 select fkeys.table_name,
        fkeys.constraint_name,
        fkeys.column_name,
        ind_cols.index_name
   from (
 select a.object_id,
        d.column_id,
        a.name table_name,
        b.name constraint_name,
        d.name column_name
   from sys.tables a
        join
        sys.foreign_keys b
     on ( a.name          = 'EMP'
          and a.object_id = b.parent_object_id
        )
        join
        sys.foreign_key_columns c
    on (  b.object_id = c.constraint_object_id )
       join
       sys.columns d
    on (    c.constraint_column_id = d.column_id
        and a.object_id            = d.object_id
       )
       ) fkeys
       left join
       (
select a.name index_name,
       b.object_id,
       b.column_id
  from sys.indexes a,
       sys.index_columns b
 where a.index_id = b.index_id
       ) ind_cols
    on (     fkeys.object_id = ind_cols.object_id
         and fkeys.column_id = ind_cols.column_id )
 where ind_cols.index_name is null
```

DISCUSSION

Each vendor uses its own locking mechanism when modifying rows. In cases where there is a parent-child relationship enforced via foreign key, having indexes on the child column(s) can reducing locking (see your specific RDBMS documentation for details). In other cases, it is common that a child table is joined to a parent table on the foreign key column, so an index may help improve performance in that scenario as well.

5.6. Using SQL to Generate SQL

PROBLEM

You want to create dynamic SQL statements, perhaps to automate maintenance tasks. You want to accomplish three tasks in particular: count the number of rows in your tables, disable foreign key constraints defined on your tables, and generate insert scripts from the data in your tables.

SOLUTION

The concept is to use strings to build SQL statements, and the values that need to be filled in (such as the object name the command acts upon) will be supplied by data from the tables you are selecting from. Keep in mind, the queries only generate the statements; you must then run these statements via script, manually, or however you execute your SQL statements. The examples below are queries that would work on an Oracle system. For other RDBMSs the technique is exactly the same, the only difference being things like the names of the data dictionary tables and date formatting. The output shown from the queries below are a portion of the rows returned from an instance of Oracle on my laptop. Your result sets will of course vary.

```
/* generate SQL to count all the rows in all your tables */
select 'select count(*) from '||table_name||';' cnts
  from user_tables;

CNTS
----------------------------------------
select count(*) from ANT;
select count(*) from BONUS;
select count(*) from DEMO1;
select count(*) from DEMO2;
select count(*) from DEPT;
select count(*) from DUMMY;
select count(*) from EMP;
select count(*) from EMP_SALES;
select count(*) from EMP_SCORE;
select count(*) from PROFESSOR;
select count(*) from T;
select count(*) from T1;
select count(*) from T2;
select count(*) from T3;
select count(*) from TEACH;
select count(*) from TEST;
select count(*) from TRX_LOG;
select count(*) from X;

/* disable foreign keys from all tables */


select 'alter table '||table_name||
       ' disable constraint '||constraint_name||';' cons
  from user_constraints
 where constraint_type = 'R';

CONS
------------------------------------------------
alter table ANT disable constraint ANT_FK;
alter table BONUS disable constraint BONUS_FK;
alter table DEMO1 disable constraint DEMO1_FK;
alter table DEMO2 disable constraint DEMO2_FK;
alter table DEPT disable constraint DEPT_FK;
alter table DUMMY disable constraint DUMMY_FK;
alter table EMP disable constraint EMP_FK;
alter table EMP_SALES disable constraint EMP_SALES_FK;
alter table EMP_SCORE disable constraint EMP_SCORE_FK;
alter table PROFESSOR disable constraint PROFESSOR_FK;
/* generate an insert script from some columns in table EMP */


select 'insert into emp(empno,ename,hiredate) '||chr(10)||
       'values( '||empno||','||''''||ename
       ||''',to_date('||''''||hiredate||''') );' inserts
```

```
  from emp
 where deptno = 10;

INSERTS
------------------------------------------------
insert into emp(empno,ename,hiredate)
values( 7782,'CLARK',to_date('09-JUN-1981 00:00:00') );

insert into emp(empno,ename,hiredate)
values( 7839,'KING',to_date('17-NOV-1981 00:00:00') );

insert into emp(empno,ename,hiredate)
values( 7934,'MILLER',to_date('23-JAN-1982 00:00:00') );
```

DISCUSSION

Using SQL to generate SQL is particularly useful for creating portable scripts such as you might use when testing on multiple environments. Additionally, as can be seen by the examples above, using SQL to generate SQL is useful for performing batch maintenance, and for easily finding out information about multiple objects in one go. Generating SQL with SQL is an extremely simple operation, and the more you experiment with it the easier it will become. The examples provided should give you a nice base on how to build your own “dynamic” SQL scripts because, quite frankly, there’s not much to it. Work on it and you’ll get it.

5.7. Describing the Data Dictionary Views in an Oracle Database

PROBLEM

You are using Oracle. You can’t remember what data dictionary views are available to you, nor can you remember their column definitions. Worse yet, you do not have convenient access to vendor documentation.

SOLUTION

This is an Oracle-specific recipe. Oracle not only maintains a robust set of data dictionary views, but there are even data dictionary views to document the data dictionary views. It’s all so wonderfully circular.

Query the view named DICTIONARY to list data dictionary views and their purposes:

```
select table_name, comments
  from dictionary
  order by table_name;

TABLE_NAME                     COMMENTS
------------------------------ --------------------------------------------
ALL_ALL_TABLES                 Description of all object and relational
                               tables accessible to the user

ALL_APPLY                      Details about each apply process that
                               dequeues from the queue visible to the
                               current user
…
```

Query DICT_COLUMNS to describe the columns in given a data dictionary view:

```
select column_name, comments
     from dict_columns
 where table_name = 'ALL_TAB_COLUMNS';

COLUMN_NAME                    COMMENTS
------------------------------ --------------------------------------------
OWNER
TABLE_NAME                     Table, view or cluster name
COLUMN_NAME                    Column name
DATA_TYPE                      Datatype of the column
DATA_TYPE_MOD                  Datatype modifier of the column
DATA_TYPE_OWNER                Owner of the datatype of the column
DATA_LENGTH                    Length of the column in bytes
DATA_PRECISION                 Length: decimal digits (NUMBER) or binary
                               digits (FLOAT)
```

DISCUSSION

Back in the day when Oracle's documentation set wasn't so freely available on the Web, it was incredibly convenient that Oracle made the DICTIONARY and DICT_ COLUMNS views available. Knowing just those two views, you could bootstrap to learning about all the other views, and from thence to learning about your entire database.

Even today, it's convenient to know about DICTIONARY and DICT_COLUMNS. Often, if you aren't quite certain which view describes a given object type, you can issue a wildcard query to find out. For example, to get a handle on what views might describe tables in your schema:

```
select table_name, comments
  from dictionary
 where table_name LIKE '%TABLE%'
 order by table_name;
```

This query returns all data dictionary view names that include the term “TABLE”. This approach takes advantage of Oracle’s fairly consistent data dictionary view naming conventions. Views describing tables are all likely to contain “TABLE” in their name. (Sometimes, as in the case of ALL_TAB_COLUMNS, TABLE is abbreviated TAB.)