FULL VOLUME TITLE
*ASP Conference Series, Vol. *, © 2007*
Editor 1, Editor 2, and Editor 3, eds.

Chapter 1: Introduction to SQL

Thomas A. McGlynn

Maria Nieto-Santisteban

Introduction

This chapter provides a very brief introduction to the Structured Query Language (SQL) for getting information from relational databases. We make no pretense that this is a complete or comprehensive discussion of SQL. There are many aspects of the language the will be completely ignored in the presentation. The goal here is to provide enough background so that users understand the basic concepts involved in building and using relational databases. We also go through the steps involved in building a particular astronomical database used in some of the other presentations in this volume.

1. What is SQL?

The Structured Query Language (SQL) is a language which describes how to create, update and query relational databases. A relational database is conceptually quite simple. The basic elements of the relational database are simple tables of the kind we see commonly in both scientific and everyday publications.

Table 1. Pets

Owner	Pet Name	Pet Type
Fred	Spike	Dog
Fred	Slither	Snake
Allie	Messy	Dog
Nadia	Socks	Cat
Tom		
Janet	Rover	Dog

We have an example table above. The table has a name ("Pets") and a rectangular array of information. At the top of each column we have a column name, "Owner", "Pet Name" and "Pet Type". The information within a column is of the same type – we would know there was something wrong if "Fred" showed up in the "Pet Type" column.
Each row after the headers gives information about a single pet. Apparently Fred has two pets and Tom has none.

Tables can be visualized in just this way in a relational database. A table is given a name and has columns which each have a type – where the types are the integers or floating point numbers or strings. [An object-relational database allows the user to define new data types and put them in the database. Most relational databases provide for some special types, e.g., money and dates but these are not typically much used in astronomical databases.] Each row in the table provides information about some entity.

In addition to a mechanism for defining base tables, relational databases provided mechanisms which allow a user to 'query' a tables. We could ask how many dogs or cats were pets. The real power of relational database is evident when we have more than one table. E.g., suppose we also had a table

Table 2. Veterinarians

Veterinarian	Treats
Allen	Cats
Allen	Dogs
Weller	Fish
Johnson	Horses
Johnson	Dogs

With just this simple pair of tables there are lots of questions we can ask: Who can treat Spike? Are there any vets that cannot treat any of the pets? Are there any pets who must go without veterinary care? How many pets can Dr Allen treat? SQL makes it easy to combine tables and get information out of them.

In the past decade astronomy has seen publication of many very large tables with more than 10^8 objects in them. SQL is a natural framework for mining these resources. The Virtual Observatory has developed a general framework for providing relational databases on-line and performing queries that involve tables from one or more database. These *SkyNodes* are discussed elsewhere in this volume.

In the next section we will build an astronomical table using SQL statements. Section 3 shows how we can make interesting astronomical queries using SQL on this database.

In this chapter we shall assume use of the freely available MySQL database system for building and querying the table. Normally databases are concealed under layers of user interfaces, so both the queries and results may seem a bit crude, but this low level approach helps illustrate what's possible with relational databases.

2. Building a database

We are going to build a database consisting of three tables. These tables were part of the [ref...] who created a table of Spectroscopic Properties of Cools Stars and a comparison table from [] Santos [...]. We create two tables for the SPOCS database. The information is split into the xxx table and the yyy table. We will run a series of commands. These are found in the software distribution in the $NVOSS/sql/mySQL/spocs directory. If you have installed MySQL you should be able to run the commands exactly as shown below from the command line (modulo the changes between Windows and Unix-like environments). If you are using another database you will need to modify the examples somewhat, but the basic out-

lines should remain the same. We assume that you have started the MySQL CLI and are at the MySQL prompt.

2.1. Creating a name space.

Since there may be many different users and uses of a given relational database system installation, most systems support a way to group related tables. A group of tables is called a 'database'. While this is rather confusing terminology, these databases are essentially just independent names spaces. A user of a database system selects the current default database using a system dependent command. [In the MySQL CLI the USE command selects the current database. When accessing a database through a JDBC connection, the JDBC URL indicates the chosen database.]
Generally tables in other databases can still be accessed and used, they just have to have the database name prefixed to the table name.

```
CREATE DATABASE database_name
```

creates a database with the given name. Normally database names must be unique within the database system. Example:

```
create database SPOCS;
```

To get rid of a database just

```
DROP DATABASE database_name
```

2.2. Creating a table

The command to create a table is simply

```
CREATE TABLE tablename (name1 type1, name2 type2, ...)
```

In addition to specifying whether a the type of a field, the column definition can indicate whether this column must be specified. SQL supports a special value, 'null', that indicates that this field has not been filled for this row. A column that allows null values is 'nullable'. It may not make sense for a column to allow nulls and the phrase 'not null' can be added to the type when that is the case.
To create a table of bank transactions we might have

```
create table bankTransfer (transaction     long not null,
                           srcAccount      int not null,
                           dstAccount      int not null,
                           amountInPennies int not null,
                           comment         varchar64);
```

This definition ensures that each record must have all essential data, but allows an optional comment of up to 64 characters.
To get rid of table

```
DROP TABLE tableName
```

2.3. Adding rows to a table

The INSERT command is used to add a row to a database. This simplest syntax for this command is

```
INSERT INTO tableName VALUES(val1, val2, val3 ...)
```

where val1, val2, ... are the values for the row in the in same order as specified in the create table command. If a string is being inserted, it should be enclosed in single quotes. E.g.,

```
insert into bankTransfer
   values(1111002, 10030001, 10030002, 390000,
          'Transfer checking to savings');
insert into bankTransfer
   values(1111003, 96300206, 10030002, 141117, null);
```

adds two rows to the bankTransfer table where there is no comment for the second row.

One can also add a row and only specify some of the columns (in any order). E.g., the second insert above could have been specified as

```
insert into bankTransfer
  (transaction, dstAccount, srcAccount, amountInPennies)
  values (1111003, 10030002, 96300206, 141117);
```

The following should give an error if we try it:

```
insert into bankTransfer
  (toAccount, srcAccount, amountInPennies)
   values(10030002, 96300206, 141117);
```

We have to specify all the non-nullable fields in the insert.

We delete records using the DELETE command, described below.

2.4. Creating the SPOCS and SANTOS tables.

To create the SPOCS and Santos tables we run run the build_all.sql script. This creates the tables, and then inserts all of the data into them. Let's take a look:

```
Create the database we are going to work in.
create database SPOCS;
use SPOCS;
\. schema.sql
\. spocs_build.sql
\. spocs_ext_build.sql
\. santos_build.sql
```

This script just loads a series of other scripts that define and populate the tables.

The schema.sql file defines the structure of the tables. It looks like:

```
DROP DATABASE IF EXISTS SPOCS;
CREATE DATABASE SPOCS;
USE SPOCS;
CREATE TABLE spocs (ID smallint not null,
                    Name varchar(20) not null,
                    Teff smallint DEFAULT null,
                    logg double DEFAULT null,
                    Metal double DEFAULT null,
                    Na double DEFAULT null,
                    Si double DEFAULT null,
                    Ti double DEFAULT null,
...
                    CONSTRAINT PRIMARY KEY (id)
);
...
```

There are a few MySQL specific additions here. The `IF EXISTS` in the `DROP DATABASE` is one such. Many database systems, including MySQL, provide for a `USE` command. That indicates that by default all tables are to be found in the specified database. The `CONSTRAINT` at the end of the `CREATE` command tells the database

how to organize the table. How this is done varies a fair bit from database to database and we could have left this out. It just makes some queries run faster.
The next three files load each of the tables. They are just a series of INSERT statements.. The first few lines of spocs_build.sql are

```
insert into spocs values(0,'Sun',5770,4.44,0.00,0.00, ...
insert into spocs values(1,'HD225261',5265,4.59,-0.31, ...
insert into spocs values(2,'HD105',6126,4.65,-0.0, ...
insert into spocs values(3,'HD142',6249,4.19,0.08, ...
...
```

We are not quite done... The positional information in the SPOCS table is encoded in sexagesimal form. It is much more convenient to be able to manipulate real numbers, so we may wish to add RA and Dec columns to the SPOCS database. The addcolumns.sql adds two columns to the database:

```
use SPOCS;
alter table spocs add ra double precision;
alter table spocs add `dec` double precision;
```

while fillcolumns.sql fills the new columns:

```
use SPOCS;
update spocs s, spocs_ext e
   set s.ra = 15*(e.RAh+e.RAm/60.+e.RAs/3600.)
   where s.ID = e.ID;
update spocs s, spocs_ext e
   set s.dec = e.DEd + e.DEm/60. + e.DEs/3600.
   where e.DEd >= 0 and s.ID = e.ID;
update spocs s, spocs_ext e
   set s.dec = e.DEd - e.DEm/60. - e.DEs/3600.
   where e.DEd < 0 and s.ID = e.ID;
```

We will not discuss the alter table and update commands beyond these examples since they will rarely be needed by typical users. The update uses some advanced features, joins, table prefixes and a where clause that are described below in the discussion of the SELECT statement. These commands do illustrate how tables in relational databases are quite flexible and can be easily changed after they are created.
One special MySQL issue. Note the backticks around the dec field in the second alter table. MySQL treats dec by itself as a reserved word (short for decimal, a data type). The backticks allow us to use such reserved words as column names. There is no problem referring to dec as a column when it is preceded by a table prefix. It is quite natural to use dec as a column name for the declination, but if you get mysterious syntax errors in MySQL this is one possible cause.

3. Querying a Database

While most astronomers have only limited interest in build tables in relational databases, getting information out of them is a different story. The SQL SELECT statement is used to query tables and get information out of them. SELECTs can be simple or quite complex. We will discuss only a few of the simpler features of the select.

3.1. A simple select example

Consider the bankTransfer table we described in section 2.3. Suppose we want to find all of the transfers of more than $1000. The SELECT statement might look like

```
select transaction, toAccount amountInPennies/100.
   from bankTransfer
   where amountInPennies > 100000
   order by toAccount,amountInPennies
```

Many simple select statements have the form

```
SELECT selectionList FROM tablelist WHERE condition ORDER BY fields
```

Here the selectionList is the list of columns that we want to get out. It can include columns in the original table or expressions involving those columns, or even constants.

The table list is the set of tables we are querying. In our example there is just one table. In addition to the name of the table, an alias for the table (often just one or two letters) can be given for each table, e.g., in `... from bankTransfer bt, accountDetails ad ...` we have an alias, `bt`, for the `bankTransfer` table.

In queries where more than one table is used, the name of a column may be ambiguous: a column of that name may be present in more than one table. The name of the table or more commonly the table alias may be prefixed to the column name to make it clear which table each column belongs to. ADQL, the dialect of SQL used in the Virtual Observatory SkyNodes requires that an alias be used whenever a column is specified in a query. The query above would need to be written as

```
select bt.transaction, bt.toAccount bt.amountInPennies/100.
   from bankTransfer bt
   where bt.amountInPennies > 100000
   order by bt.toAccount,bt.amountInPennies
```

The `order by` clause specifies the order in which the results will be displayed. Unless this clause is present the order of results is undefined and is not guaranteed to be the same even for two identical fields. One or more fields may be specified (expressions can also be used) and the results are ordered by the first element. If this is equal for two rows then the second element is used to break the tie and so forth. The keywords "ASC" or "DESC" can be appended to an element to specify the direction of the sort. ASC is the default.

The `where` clause determines which rows are to be output. In our example we have a very simple where clause with a single criterion. Any number of criteria can be put together using logical operators AND, OR and NOT.

3.2. Filters and joins.

Conceptually the criteria that show up in the where clause can be separated as filters and joins. Filters simply get rid of some of the available rows. For example the criterion we used above `bt.amountInPennies > 100000` filters the rows in the `bankTransfer` table and returns only a subset of these. Any number of filters can be placed on a table and they may involve expressions. If a table has B and V magnitudes for stars and we only want to consider stars with a particular color we might use `bmag-vmag > 1 and bmag-vmag < 1.5` (or equivalently `bmag-vmag between 1 and 1.5`).

For the dialect of SQL used in the Virtual Observatory, joins between tables are also specified in the where clause. When more than one table is specified in the table list for the query, the query is performed on the cross-product of the two tables. E.g., if one table has 30 rows and a second 20, then there are potentially 600 rows in a query that involves both of them. Normally for each row in the first table we only want to

consider some small subset of the second. E.g., if we are doing a query that involves bank transfers and bank accounts, then for each transfer we probably only want to look at the accounts involved in that transfer – not every account in the system. We might have a query like

```
select acct.name,acct.address,bt.amountInPennies
    from  bankAccount acct, bankTransfer bt
    where bt.amountInPennies > 100000 and
              acct.id = bt.toAccount
```

where we have an account table with the account holders name and address and their account number for each account in the system. There are two criteria in the where clause. The first is our familiar filter. We are only interested in transaction above $1000. The second says that for each transaction, find the row in the account table which has the name and address for the account to which the money is being transferred.

There are other ways of specifying joins in most modern dialects of SQL (using a JOIN field in the FROM clause), but the effect is the same.

Perhaps the most common join for astronomers is to find rows in two tables where the objects are close to each other in the sky, spatial cross-correlations. Which it is easy to handle proximity in one dimension in standard SQL, doing matches on the sky for large databases often requires special handling. For smaller databases (e.g., less than a million rows), it is usually reasonably fast to just use standard geometric expressions. These may seem clumsy expressed in SQL, but databases are typically I/O bound so that evaluating complex expressions may have no effect on the actual query time. The issues involved in doing spatial joins are discussed in the [building a skynode server].

4. Querying the SPOCS database

In this section we will do a series of queries of the SPOCS database we created in section 2. Each of these queries is available as file in the software distribution.

4.1. Find analogs to the sun.

The Sun has a temperature of T_{eff}=5770 K and a gravity of log(g)=4.4 and a metallicity of [M/H]=0.0. Stars like the Sun will have values of these three parameters similar to the solar values. To look for stars with temperatures within 50 K of the Sun and similar gravities and metallicities we have

```
SELECT s.ID, s.Name, s.Teff, s.Logg, s.Metal
  FROM spocs s
  WHERE s.Teff BETWEEN 5720 AND 5820 AND
        s.Logg BETWEEN 4.34 AND 4.54 AND
        s.Metal BETWEEN -0.1 AND +0.1
ORDER BY s.Teff;  -- query1.sql
```

4.2. Look for systematic trends vs temperatures.

Making a star a little bit cooler, reducing the gravity a little bit, or increasing the metallicity a little bit all tend to make spectral lines a little bit deeper. This unfortunate reality makes it hard to get the temperature, gravity, and metallicity exactly correct.

When astronomers measure gravity, any error the temperature or metallicity will cause a compensating error in gravity. Similarly, when measuring metallicity, any error in temperature and gravity will cause a compensating error in metallicity. All three parameters are interlinked.* Detecting and characterizing systematic errors can be difficult.
Here we compare stellar parameters from two different spectroscopists, using different analysis techniques. Ordering the differences as a function of increasing temperature shows trends. The size of these trends may give a sense of the systematic errors in the results.

```
SELECT s.ID, s.Name, s.Teff,
       (s.Teff - e.Teff) as dTeff, s.Logg,
       format(s.Logg - e.Logg,2) as dLogg, s.Fe,
       format(s.Fe - e.Fe,2) as dFe
   FROM spocs s, santos e
   WHERE substring(s.Name,3) = trim(e.HD)
   ORDER BY s.Teff; -- Query 3
```

This query uses special functions in the database to make the results look pretty and also shows how we may need to be careful when we do joins. We are querying both the SPOCS and Santos tables. The SPOCS and Santos table both have the HD number of the stars as columns, but in the SPOCS table they are found in the name column prefixed with the string 'HD '. So we need to use string manipulation function to match the columns. We also use the MySQL `format` function to make the output look nicer with just two digits to the right of the decimal point for the computed quantities. These calls could be omitted.
If you are careful you may have noted that the computed columns were given names, for example in `(s.Teff - e.Teff) as dTeff`. This is standard SQL. One can change the names of columns, or assign names to expressions in the output.
This query gives us a table of all of the HD objects included in both the SPOCS and Santos tables and gives the difference in measured temperature and gravity. The results are ordered by temperature as measured in SPOCS. The output of this query can be plotted or analyzed statistically.

4.3. Is there a systematic difference between the 2 catalogs?

In the previous query we found the individual differences between measurements in SPOCS and Santos. Using aggregate functions we can do overall statistics as well.

```
SELECT format(avg (s.Teff - e.Teff), 1) as mean_dTeff,
       format(avg(s.Logg - e.Logg), 3) as mean_dLogg,
       format(avg(s.Fe - e.Fe), 3) as mean_dFe
  FROM spocs s, santos e
  WHERE substring(s.Name, 3) = trim(e.HD);  (Query 10)
```

Aggregate functions take a set of rows and compute an aggregate quantity. The most common functions are AVG, MIN, MAX and COUNT. If the SELECT statement has no GROUP BY clause, then only one row will be output. However you can do aggregates for subsets of the table. E.g., the query

```
SELECT s.ID, s.Name, count(e.HD) as c
   FROM spocs s, santos e
```

```
WHERE substring(s.Name, 3) = trim(e.HD)
GROUP BY s.ID, s.Name
ORDER BY c desc;  (Query 4)
```

looks for the matching stars in the SPOCS and Santos catalogs and finds how many entries there are in the Santos catalog for each one. In a number of cases there are multiple entries in the Santos catalog.

Grouping is one of many advanced capabilities for queries in SQL. Subselects, EXISTS and IN qualifiers, autocorrelations are just a few of the features that are discussed in more comprehensives treatments of SQL queries. Our goal here has been to give a sense of how SQL can be used to extract interesting astronomical information from databases.

5. Astronomical SQL databases on the Web.

In addition to the Virtual Observatory SkyNode resources discussed elsewhere in this volume there are a number of astronomical databases that support SQL queries.
The Sloan Digital Sky Survey - SDSS at http://skyserver.sdss.org
Galaxy Evolution Explorer – Galex at http://galex.stsci.edu/
SuperCOSMOS Science Archive - SSA at http://surveys.roe.ac.uk/
To use any relational database you will need to understand the meanings of the tables and the columns it comprises. Either the documentation or on-line tools should give you a overview of the *schema* of the database. You can then build increasingly complex queries to build familiarity with the database and its response. For large databases it is very easy to run queries that require large resources of either storage or compute time to complete. Normally through the Web you will be limited to queries that return no more than a few thousands of rows and take no more than several minutes to complete. The precise policies will vary from site to site.

6. What's Left?

This introduction to SQL has left many areas of the language entirely unexplored.
Designing databases requires careful understanding of the roles and relationships between the tables. How should tables be joined? How does one define a unique row of a table? Which datatypes should be used? Building databases is a real art.
For the large astronomical databases coming online building efficient indexes for spatial queries is a major issue. Often special columns are included in the database purely to maximize the efficiency of some searches.
SQL has a whole host of features to insure that databases can be built robustly. There are features to ensure transactional integrity where consistency can be maintained even when there is a problem in the middle of a sequence of updates. Other SQL features can be used to ensure that table columns never violate constraints that the users wish to specify.
There are many of capabilities available in queries that we have not described: subselects, GROUP BY and HAVING clauses, a myriad of functions to manipulate numbers, strings and other types.
The SQL standard describes a framework for metadata about tables though the quality of the implementation varies from system to system.

There are commonly used but non-standard extensions to SQL. Users can create tables directly from queries. Temporary tables that disappear automatically when a session is complete are usually supported.
For low level access, users need to be aware of the differences between SQL implementations by different vendors. These differences range from such basic issues as how an SQL statement is terminated, to what metadata is available.

7. Further information

There are many books available on SQL. A few are described at http://databases.about.com/cs/sql/tp/sqlbooks.htm. There are also many on-line resources.

7.1. Vendor Links

MySQL -- http://www.mysql.com
SQL Server -- http://www.microsoft.com/sql/default.mspx
Sybase – http://www.sybase.com

7.2. General SQL Links

A Gentle Introduction to SQL -- http://sqlzoo.com
SQL Course.com -- http://sqlcourse.com/, http://www.sqlcourse2.com/
SQL Tutorial -- http://www.w3schools.com/sql/
FirstSQL -- http://www.firstsql.com/tutor.htm
SDSS SQL Tutorial http://cas.sdss.org/dr4/en/help/docs/sql_help.asp