

Problem Set 1: SQL

Assigned: 9/14/2010

Due: 9/21/2010

1 Introduction

The purpose of this assignment is to provide you some hands-on experience with the SQL programming language. SQL is a declarative language, or a “relational calculus” in which you specify the data you are interested in in terms of logical expressions.

We will be using the PostgreSQL open source database, which provides a fairly standard SQL implementation. In reality, there are slight variations between the SQL dialects of different vendors—especially with respect to advanced features, built-in functions, and so on. The SQL tutorial at <http://www.postgresql.org/docs/current/static/tutorial-sql.html>, included with the Postgres documentation, provides a good introduction to the basic features of SQL; after following this tutorial you should be able to answer most of the problems in this problem set (the last few questions may be a bit tricky). You may also wish to refer to Chapter 5 of “Database Management Systems.” This assignment mainly focuses on *querying* data rather than modifying it, though you should read the sections of the tutorial that describe how to create tables and modify records so you are familiar with these aspects of the language.

We have set up a Postgres database server for you to use; you will need to use the Postgres client (`psql`) – more details are given in Section 3 below.

2 NSF Grants Database

In this problem set, you will write queries over recent research grants from the National Science Foundation (NSF) awarded to professors at several major research universities. The National Science Foundation (NSF) is a significant funder of academic research in the United States. Researchers submit proposals to various programs (for example, the “INFO INTEGRATION & INFORMATICS” (III) program in the the “Computer Science Engineering” (CSE) directorate is primarily responsible for funding research on database systems.) Panels of peer reviewers (other academics in related fields) decide which proposals will be funded. After grants are funded (typical amounts range from \$50,000 to many millions of dollars), they are administered by the university on behalf of the researchers. Researchers pay graduate students, postdocs, and staff, purchase equipment and supplies, and pay expenses (such as travel) from these grants.

In this problem set, you will write a series of SQL queries using the `SELECT` statement over a database of all recent (since the early – mid 1990’s) grants awarded to MIT, Harvard, Carnegie Mellon, Stanford, and UC Berkeley (there are approximately 10,000 such grants) in all fields (not just computer science.)

Figure 1 is a simplified “Entity-Relationship Diagram” that shows the relationships (lines) between objects (entities) stored in the database. This is a popular approach for conceptualizing the schema of a database; we will not study such diagrams in detail in 6.830, but you should be able to read and recognize this type of diagram.

Our database consists of six entities:

- Grants, with their associated meta-data (e.g., amount of funding, start and end date, the researchers who work on them, etc.),
- Organizations (e.g., universities) which receive grants,
- Researchers who receive grants,
- Programs run by the NSF that award grants,
- Managers at NSF who run programs that award grants, and

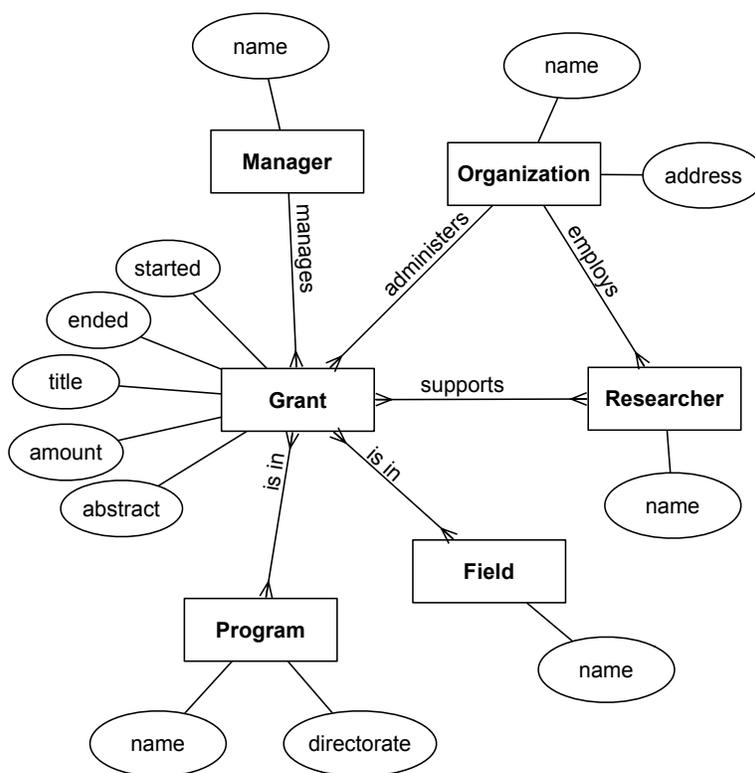


Figure 1: An entity-relationship diagram for the NSF database.

- Fields (research areas) that describe high level topic-areas for grants.

These entities are instantiated as a collection of nine tables in SQL. The “Data Definition Language” (DDL) SQL commands used to create these tables are as follows:

```
create table orgs (
  id int primary key,
  name varchar(100),
  streetaddr varchar(200),
  city varchar(100),
  state char(2),
  zip char(5),
  phone char(10)
);

create table researchers (
  id int primary key,
  name varchar(100),
  org int references orgs(id)
);

create table programs (
  id int primary key,
  name varchar(200),
  directorate char(3) -- the top-level part of NSF that runs this program
);

create table managers (
  id int primary key,
  name varchar(100)
```

```

);

create table grants (
    id int primary key,
    title varchar(300),
    amount float,
    org int references orgs(id),
    pi int references researchers(id), -- the principal investigator (PI) on the grant
    manager int references managers(id),
    started date,
    ended date,
    abstract text
);

create table grant_researchers (
    researcherid int references researchers(id),
    grantid int references grants(id)
);

create table fields (
    id int primary key,
    name varchar(100)
);

create table grant_fields (
    grantid int references grants(id),
    fieldid int references fields(id)
);

create table grant_programs(
    grantid int references grants(id),
    programid int references programs(id)
);

```

CREATE TABLE *name* defines a new table called *name* in SQL. Within each command is a list of field names and types (e.g., *id* INTEGER indicates that the table contains a field named *id* with type INTEGER). Each field definition may also be followed by one or more modifiers, such as:

- PRIMARY KEY: Indicates this is a part of the primary key (i.e., is a unique identifier for the row). Implies NOT NULL.
- NOT NULL: Indicates that this field may not have the special value NULL.
- REFERENCES: Indicates that this is a foreign key which references an attribute (i.e., column) in another table. Values of this field must match (join) with a value in the referenced attribute.

Phrases following the “--” in the above table definitions are comments.

Notice that the above tables reference each other via several foreign-key relationships. For example, grants are administered by an organization, as denoted by the line `org int REFERENCES orgs(id)`.

Note that there are three “many-to-many” relationships (between grants and researchers, grants and fields, and grants and programs). In SQL, these many-to-many relationships are represented by additional tables (e.g., `grant_researchers`), which is why there are six distinct entities, but nine SQL tables.

3 Connecting to the Database

We have set up a database server for you to use on **hancock.csail.mit.edu**. To connect to the database on hancock, you will need the `psql` Postgres client. Type the following (assuming `psql` is in your `PATH`):

```
psql -h hancock.csail.mit.edu -p 5432 6830-2010 username
```

username is your Athena* username that you signed up with on our signup sheet, or if you don't have an Athena* username, the part of the email address that you gave us prior to the @ sign. If you are unable to log in, please let us know. Currently, the database only permits incoming connections from an MIT or CSAIL machine, and we would prefer you use a machine within MIT (e.g., an Athena* machine) to connect to the database and run queries, if possible. If you need to use a machine that's not within MIT, you can use MIT's VPN software, at <http://ist.mit.edu/services/network/vpn>, or you can use an SSH client to log into `linux.mit.edu` and run the `psql` client from there.

3.1 psql Clients

- **Athena* Cluster:** The `psql` client is installed on the `linux.mit.edu` Athena* dialup server, which you can SSH into. It can also be installed on cluster computers running DebAthena, the new version of Linux Athena*, by running the following command at an Athena* prompt¹:

```
sudo aptitude install postgresql-client
```

Follow the prompts (and type your Athena password again if prompted). After running this command, you'll be able to use `psql` until you log out.

- **Your computer:** You may also connect to the database using `psql` installed on your local computer. In general, it can be installed through the "postgresql" or "postgresql-client" package in a package manager program, such as `apt` or `yum` for Linux, MacPorts for Mac, or Cygwin for Windows. Self-installation can be somewhat involved on some platforms; note that you can always use Athena if you run into trouble.

3.2 Running locally

Finally, if you wish to install a postgres server locally to run the queries on, please email the staff and we will provide you with a gzipped file that will load the tables into your server.

4 Using the Database

Once connected, you should be able to type SQL queries directly. All queries in Postgres must be terminated with a semi-colon. For example, to get a list of all records in the `grants` table, you would type (note that running this query may take a long time, since it will yield thousands of answers, so please don't do it – keep reading instead!):

```
SELECT * FROM grants;
```

A less expensive way to sample the contents of a table (which you may find useful) is to use the `LIMIT SQL` extension which specifies a limit to the number of records the database should return as an answer to a query. This is not part of the SQL standard, but is supported by many relational DBMSes, including Postgres. For example, to view 20 rows from the `grants` table, use the following query in Postgres:

```
SELECT * FROM grants LIMIT 20;
```

The `LIMIT` clause above returns only the first 20 rows of the result for the query `SELECT * FROM grants`. However, keep in mind that the relational model does not specify an ordering for rows in a relation. Therefore, there is no guarantee on which 20 rows from the result are returned, unless the query itself includes an `ORDER BY` clause, which sorts results by some expression. Nevertheless, `LIMIT` is very useful for playing around with a large database and sampling data from tables in it.

You can use the `\h` and `\?` commands for help on SQL syntax and Postgres specific commands, respectively. For example, `\h SELECT` will give the syntax of the `SELECT` command.

You may find the `\dt` command particularly useful; it allows you to see the names of the tables that are available and (when invoked with a particular table name) their schemas. This may not work for `psql` clients that are not version 7.4, thus we have

¹If you are familiar with the Linux "sudo" command but are new to DebAthena, you may have concerns about running "sudo" on a public workstation. Fear not; its use is fully supported, and any changes you make to the filesystem will be reverted on logout.

provided a 7.4 version of `psql`, compiled for the `linux.dialup.mit.edu` machines. Download and extract it using the following commands:

```
wget http://db.csail.mit.edu/6.830/psql-7.4.tar.gz
gzip -d psql-7.4.tar.gz
tar -xf psql-7.4.tar
```

A note on good database behavior: No correctly written query for this problem set should require more than about 2 minutes to execute, although incorrect queries (that, for example, join large tables without a join condition) can run for much longer. Please kill queries (by typing *Ctrl-C* in `psql`) that have been running for a long time – such queries consume valuable resources on the database server and decrease the performance of every other user’s queries. We will kill connections to the database that run for longer than 2 minutes if server performance starts to become a problem.

5 Places to Look for More Help on SQL

The Postgres manual provides a good introduction to the basic features of SQL. There are a number of other online guides available; in particular, <http://sqlzoo.net> provides a “Gentle Introduction to SQL” which is quite thorough and includes many examples.

6 Questions

For each of the following eight questions, please include both the **SQL query** and the **result** in your answer. None of the queries should take more than a couple minutes.

- Q1.** Write a query to find the title and amount of the grants for which Professor Stonebraker is the principal investigator (PI). (Note: Take a look at the `researchers` table to understand the format of researcher names. In general, it’s helpful to explore the tables when encountering a new database. When doing so, please use the `LIMIT` clause to avoid exploding the database).
- Q2.** Write a query that finds the number of times an MIT researcher has received (not necessarily a PI) a grant over \$1 million since 1/1/2008. Retrieve the name of the researcher and the number of such grants awarded.

6.1 Additional features of SQL you should know:

For the remainder of the queries, you will need to use some more advanced SQL features, which we briefly describe here.

Subqueries and nesting: In SQL, a subquery is a query over the results of another query. You can use subqueries in the `SELECT` list, the `FROM` list, or as a part of the `WHERE` clause. For example, suppose we want to find the name of the grant with the smallest id. To find the smallest id, we would write the query `SELECT min(id) FROM grants`, but SQL doesn’t provide a way to get the other attributes of that minimum-id grant without using a nested query. We can do this either with nesting in the `FROM` list or in the `WHERE` clause. Using the nesting `FROM` list, we would write:

```
SELECT title
FROM grants,
      (SELECT min(id) AS minid
       FROM grants) AS nested
WHERE grants.id = nested.minid;
```

Using nesting in the `WHERE` clause, we would write:

```
SELECT title
FROM grants
WHERE grants.id = (SELECT min(id) FROM grants);
```

As we discussed in class, there are usually several possible ways to write a given query, and some of those ways may provide different performance (despite the best efforts of database system designers to build optimizers that yield query performance that is independent of the query's formulation).

Note that if you were interested in finding grants with researchers that matched a list of ids, you could replace the “=” sign in the query above with the `IN` keyword; e.g.:

```
SELECT grants.title
FROM grants, grant_researchers
WHERE grants.id = grant_researchers.grantid
AND grant_researchers.researcherid IN (SELECT id FROM researchers WHERE PREDICATE)
```

It is usually the case that when confronted with a subquery of this form, it is possible to un-nest the query by rewriting it as a join.

Q3. Show what the `SELECT ...WHERE ...IN` query above would look like when the `IN` portion of the query is “un-nested” by rewriting it as a join that does not contain a subquery. Use `PREDICATE` in your query to denote the conditions in the `WHERE` predicate of the original subquery.

Temporary Tables: In addition to allowing you to nest queries, SQL supports saving the results of a query as temporary table. This table can be queried just like a normal table, providing similar functionality to nested queries, with the ability to reuse the results of a query. The command to create a temporary table is:

```
CREATE LOCAL TEMP TABLE name AS SELECT ...
```

where “...” are the typical `SELECT` arguments. This creates a table called `name`. `LOCAL` makes the table only visible to the current session. `TEMP` causes the table to automatically be deleted (or “dropped” in SQL nomenclature) when the session is over, such as when you quit `psql`.

Temporary tables can be useful when interactively developing a SQL query, since you can explore or build on previous results. It is usually possible to use nesting in place of temporary tables and vice versa; nesting will (generally) lead to better performance as query optimizers include special optimizations to “de-nest” queries that cannot be easily applied on temporary tables.

In addition to nesting (or temporary tables), to answer some of the following questions, you must learn two concepts: *self-joins* and *aggregates*.

Self joins: A self-join is a join of a table with itself. This is often useful when exploring a transitive relationship. For example, the following query:

```
SELECT gr2.grantid
FROM grant_researchers AS gr1, grant_researchers AS gr2
WHERE gr1.researcherid = 0
AND gr1.researcherid = gr2.researcherid;
```

would return the list of the ids all grants written by the researcher with id 0.

6.2 Advanced Questions

Q4. Write a query that finds the organizations that have received the same number of grants for 3 years in a row, with at least 10 grants per year. Report the name of the organization, the number of grants and the years they received the grants. (*Hint: you can use the expression `extract('year' from date)` to get the year from a date column.*)

- Q5.** There are a total of 9 distinct directorates in our NSF database (though there are several hundred programs). Find the researchers that have received grants from 6 or more directorates. Retrieve the researcher's name and the number of directorates he/she has received grants from. Order the results in descending order by the number of directorates, and return the first 10. (*Hint: use the HAVING, ORDER BY and LIMIT operators.*)
- Q6.** Find the managers who have given the most amount of money to a single organization. For the 5 highest amounts, retrieve the manager name, organization name and total amount given.
- Q7.** It turns out that many grants are multidisciplinary (i.e., spans multiple fields). Among the many pairs of fields (e.g., computer science and biology) that have shared grants, find the 5 pairs that have historically shared the most grants. Report the number of all grants shared by each pair, and the names of the fields.
- Q8. Hard!** Find the MIT researchers that received more than \$1 million in grants one year, did not receive any grants the subsequent year, then received more than \$1 million in grants the 3rd year. (*Note: this is a fairly difficult and computationally intensive query. Poorly written queries may run very long and will be killed. We suggest using multiple temporary tables. We used one to represent a query that finds the researchers that received more than \$1 million in the 1st and 3rd years. Our queries took a total of 10 to 15 sec to run.*)

MIT OpenCourseWare
<http://ocw.mit.edu>

6.830 / 6.814 Database Systems
Fall 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.