

# Problem Set 1: SQL

Assigned: 9/15/09

Due: 9/22/09

## 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:

- Q1.** Grants, with their associated meta-data (e.g., amount of funding, start and end date, the researchers who work on them, etc.),
- Q2.** Organizations (e.g., universities) which receive grants,
- Q3.** Researchers who receive grants,
- Q4.** Programs run by the NSF that award grants,
- Q5.** Managers at NSF who run programs that award grants, and

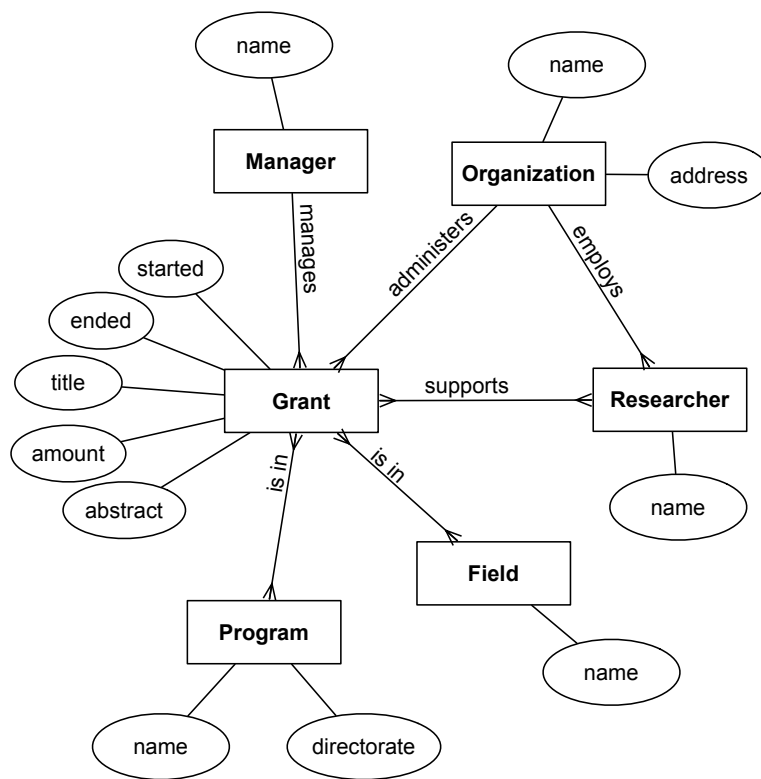


Figure 1: An entity-relationship diagram for the NSF database.

**Q6.** 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

To connect to the database, you will need the `psql` Postgres client. This 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 prompt:

```
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.

If 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.

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.

We have set up a database server for you to use:

- **hancock.csail.mit.edu**: To connect to the database on hancock, type the following (assuming you are in the directory in which you un tarred `psql`):

```
./psql -h hancock.csail.mit.edu -p 5433 6.830-2009 username
```

*username* is your Athena username that you signed up with on our sign up 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.

Finally, if you wish to install a postgres server locally to run the queries on, you can download the entire `nsf` database from <http://db.csail.mit.edu/6.830/assignments/nsf.sql.gz>. To install this locally, you will need to do something like (assuming you have postgres installed and running):

```
wget http://db.csail.mit.edu/6.830/assignments/nsf.sql.gz
gunzip nsf.gz
createdb nsf
psql nsf < nsf.sql
```

### 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.

**A note on good database behavior:** No correctly written query for this problem set should require more than about 10 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 10 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. Some of the more complex queries can take quite a while to run, so be patient!

**Q1.** Write a query to find the title and amount of the grants for which Professor Madden is the principal investigator (PI).

```
SELECT grants.title, grants.amount FROM researchers, grants WHERE researchers.id
    =grants.pi AND researchers.name = 'Samuel_Madden';
```

?	column?	amount
CAREER: MACAQUE - Managing Ambiguity and Complexity in Acqu...		450000
CSR-EHS: Collaborative Research: A General, Efficient and R...		200000
III-COR - ChunkyStore: Physical Database Design for Next-Ge...		819000
CSR-CSI: XStream, a Distributed Stream Processor for Hetero...		350000
Collaborative Research: IDBR: VoxNet- A Deployable Bioacous...		90000
Collaborative Research: A Comparative Study of Approaches t...		151213
2009 SIGMOD Programming Contest...		45000

(7 rows)

This question should be a fairly straightforward implementation of a table join. If you have any questions, see a TA. In case you’re curious, this output was modified for printing (so that it fits width-wise onto a page) by replacing “SELECT grants.title” with “SELECT substrings(grants.title from 0 for 60) || ‘...’ ”

**Q2.** Write a query that finds the total dollars received since the date 1/1/2000 by each organization (e.g., MIT, Harvard, etc.) from the NSF directorate ‘CSE’. CSE is the directorate that funds computer science research.

```
SELECT name, SUM(amount) FROM (
    SELECT DISTINCT o.name AS name, g.id, g.amount AS amount
    FROM orgs o, grants g, grant_programs gp, programs p
    WHERE
        g.started >= '2000-01-01' AND
        o.id = g.org AND
        g.id = gp.grantid AND
        p.id = gp.programid AND
        p.directorate = 'CSE') g1
GROUP BY name;
```

name	sum
Carnegie-Mellon University	269624626
Harvard University	55915317
Massachusetts Institute of Technology	148913945
Stanford University	132477205
University of California-Berkeley	180635412

It was very easy to introduce overly large results here through row duplication produced by table joins. Implementing this as a nested query with the inner query doing a “SELECT DISTINCT” prevents this. We gave a lot of partial credit to implementations that were correct other than these internal duplicated rows.

## 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 ...)
```

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. Use “...” in your query to denote the conditions in the `WHERE` predicate of the original query.

```
SELECT grants.title
FROM grants, grant_researchers, researchers
WHERE grants.id = grant_researchers.grantid
AND grant_researchers.researcherid = researchers.id
AND ...
```

This question was apparently worded somewhat confusingly, judging by responses; but what we were looking for was a conversion of the nested “`SELECT`” statement to a join condition on the outer “`SELECT`” statement.

**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 the next few 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 PIs who have received a grant in five consecutive years. Report the names of the PIs and the years in which they received the grants. If a researcher has received a grant for more than five consecutive years, only report the first five year period. (Hint: you can use the expression `extract('year' from date)` to get the year from a date column.)

```
SELECT DISTINCT r1.name,
               MIN(EXTRACT( year FROM g1.started)),
               MIN(EXTRACT( year FROM g2.started)),
               MIN(EXTRACT( year FROM g3.started)),
               MIN(EXTRACT( year FROM g4.started)),
               MIN(EXTRACT( year FROM g5.started))
FROM researchers r1, grants g1, grants g2, grants g3, grants g4, grants g5
WHERE r1.id=d1.pi
      AND g1.pi=g2.pi
      AND g2.pi=g3.pi
      AND g3.pi=g4.pi
      AND g4.pi=g5.pi
      AND EXTRACT( year FROM g2.started)=EXTRACT( year FROM g1.started)+1
      AND EXTRACT( year FROM g3.started)=EXTRACT( year FROM g2.started)+1
      AND EXTRACT( year FROM g4.started)=EXTRACT( year FROM g3.started)+1
      AND EXTRACT( year FROM g5.started)=extract( year FROM g4.started)+1
GROUP BY r1.name;
```

name	fifth	fourth	third	second	first
Barbara Romanowicz	2003	2002	2001	2000	1999
Brian Evans	2005	2004	2003	2002	2001
Christos Faloutsos	2003	2002	2001	2000	1999
Craig Moritz	2007	2006	2005	2004	2003
Daniel Rothman	1998	1997	1996	1995	1994
Daniel Schrag	2005	2004	2003	2002	2001
David Foster	2002	2001	2000	1999	1998
David Yaron	2009	2008	2007	2006	2005
Farish Jenkins	1994	1993	1992	1991	1990
Felice Frankel	2008	2007	2006	2005	2004
Fu-Kuo Chang	2005	2004	2003	2002	2001
George Johnson	2006	2005	2004	2003	2002
Gonzalo Giribet	2007	2006	2005	2004	2003
Henry Smith	2005	2004	2003	2002	2001
Irwin Shapiro	1990	1989	1988	1987	1986
Jacobo Bielak	1995	1994	1993	1992	1991
Jamie Callan	2007	2006	2005	2004	2003
Jimmy McGuire	2009	2008	2007	2006	2005
John Anderson	1990	1989	1988	1987	1986
John Doyle	2005	2004	2003	2002	2001
Jonathan Bray	2004	2003	2002	2001	2000
Jon Krosnick	2009	2008	2007	2006	2005
Karel Liem	1990	1989	1988	1987	1986
Lorenz Biegler	2001	2000	1999	1998	1997
Myung Jhon	2009	2008	2007	2006	2005
Nancy Kleckner	1990	1989	1988	1987	1986
Noel Holbrook	2003	2002	2001	2000	1999
None None	1990	1989	1988	1987	1986

Ofer Bar-Yosef		1994		1993		1992		1991		1990
Peter Vitousek		2008		2007		2006		2005		2004
Philip Sadler		1999		1998		1997		1996		1995
Robert Finkel		2009		2008		2007		2006		2005
Robert Reilinger		2003		2002		2001		2000		1999
Samuel Bowering		2000		1999		1998		1997		1996
Sara Kiesler		2003		2002		2001		2000		1999
Stein Jacobsen		1993		1992		1991		1990		1989
Thomas Jordan		1998		1997		1996		1995		1994
Timothy Grove		1998		1997		1996		1995		1994
Umran Inan		1998		1997		1996		1995		1994

(39 rows)

The trick to this question was to realize that you can determine consecutiveness by joining a table against itself on a condition that looks like  $A_1 = A_2 + 1$ . Given that fact, it is possible to build up a table that lists out consecutive 5-year blocks for each PI. There's one additional subtlety, that we request the minimum year in case of duplicates; to meet this requirement, you simply have to take the minimum of the years (though this may not actually be so simple in practice given the complexity of the query).

- Q5.** Write a query that finds the total amount of grants given to each researcher by the 'CSE' directorate and reports the top 10 amounts and the name of the researcher receiving that amount. Include all proposals which the researcher works on (not just those for which he or she is the PI.) (Hint: the LIMIT clause returns a fixed number of records; combine it with the ORDER BY clause to return the top 10.)

```
SELECT rname, SUM(amount) as total FROM (
  SELECT DISTINCT r.id as rid, r.name as rname, g.amount as amount, g.id as
    gid
  FROM researchers r, grant_researchers gr, grants g, grant_programs gp,
    programs p
  WHERE r.id = gr.researcherid AND gr.grantid = g.id AND g.id = gp
    .grantid
    AND gp.programid = p.id AND p.directorate = 'CSE'
) AS DATA
GROUP BY rid, rname
ORDER BY total DESC
LIMIT 15
```

rname		total
None None		116533046
S. Shankar Sastry		49891877
Michael Reiter		32297440
Ken Koedinger		30984252
John C. Mitchell		29111702
R. White		28447419
Hector Garcia-Molina		26331234
Kurt VanLehn		24987245
David Klahr		24987245
Stephen Wicker		18980220
Janos Sztipanovits		18980220
Dan Boneh		17795988
Mitchel Resnick		17123362
Jaime Carbonell		16369101
Scott Shenker		15579722

(15 rows)

Many of you wisely noted that “None None” is probably not a real person, and so filtered it out; we certainly accepted that answer as well. In any case, the trickiest part of this question is to filter out internal duplicates, which the nested “SELECT DISTINCT” will do in this example solution.

- Q6.** Find the top 10 managers who have given the largest number of grants to a single PI (based on the value of the `grants.pi` field).

```
SELECT rname, mname, count FROM (
    SELECT COUNT(*) AS count, r.id, r.name AS rname, m.id, m.name AS mname
        FROM grants g, researchers r, managers m
        WHERE g.pi = r.id AND g.manager = m.id
        GROUP BY r.id, r.name, m.id, m.name
        ORDER BY count DESC
        LIMIT 15
    ) AS Data;
```

rname	mname	count
Simon Klemperer	Leonard E. Johnson	10
Edward Boyle	Donald L. Rice	10
None None	Susan W. Duby	9
Dawn Song	Karl N. Levitt	9
Ofer Bar-Yosef	John E. Yellen	9
Steven Gorelick	L. Douglas James	8 <-- tie
Michael Watts	Thomas J. Baerwald	8 <-- tie
None None	William J. Hahn	8 <-- tie
Kannan Ramchandran	John Cozzens	8 <-- tie
Paola Malanotte-Rizzoli	Eric C. Itsweire	8 <-- tie
Carl Wunsch	Eric C. Itsweire	8 <-- tie
Barbara Romanowicz	Robin Reichlin	8 <-- tie
Ragunathan Rajkumar	D. Helen Gill	7
Stephen Monismith	Eric C. Itsweire	7
Lorenz Biegler	Maria Burka	7

(15 rows)

Many people had different interpretations of the question; we tried to give partial credit for reasonable interpretations where we could figure them out. The trick to this query was to use “ORDER BY” and “LIMIT” in conjunction with an aggregate field to find the top some number of people. The example above shows how to do this.

- Q7.** Write a query to find the top 10 pairs of researchers from different organizations that have worked together on the most grants, and the number of grants they have worked together on.

```
SELECT rname1 as researcher1, rname2 as researcher2, count FROM (
    SELECT r1.id AS rid1, r2.id AS rid2, r1.name AS rname1, r2.name AS
        rname2, COUNT(*) AS count
        FROM researchers r1, researchers r2, grant_researchers gr1,
            grant_researchers gr2
        WHERE r1.id = gr1.researcherid
            AND r2.id = gr2.researcherid
            AND r1.org != r2.org
            AND gr1.grantid = gr2.grantid
            AND r1.id < r2.id
        GROUP BY r1.id, r1.name, r2.id, r2.name
        ORDER BY count DESC, rname1
        LIMIT 23
    ) AS DATA;
```

researcher1	researcher2	count
Charles Leiserson	Bradley Kuszmaul	7
Thomas Herring	Bradford Hager	6
Tze Lai	David Siegmund	6
Daniel Kleppner	Thomas Greytak	5
James Rice	Renata Dmowska	5
Stephen Mende	Harald Frey	5
Donald Ornstein	Yitzhak Katznelson	4 <-- tie
Haynes Miller	Michael Hopkins	4 <-- tie
John Foster	Philip Erickson	4 <-- tie
John Richardson	Karolen Paularena	4 <-- tie
Juhn Liou	W. Gary Ernst	4 <-- tie
Leonard Susskind	Renata Kallosh	4 <-- tie
Leonard Susskind	Andrei Linde	4 <-- tie
Leonard Susskind	Savas Dimopoulos	4 <-- tie
Marcia Linn	James Slotta	4 <-- tie
Michael Levine	Ralph Roskies	4 <-- tie
Patrick Thaddeus	Carl Gottlieb	4 <-- tie
Roberta Rudnick	William McDonough	4 <-- tie
Robert Gray	Richard Olshen	4 <-- tie
Robert King	Thomas Herring	4 <-- tie
Steven Shreve	John Lehoczky	4 <-- tie
Walter Murray	Michael Saunders	4 <-- tie
Adam Dziewonski	Robert Woodward	3

(23 rows)

The method behind this question was fairly similar to the method behind Question 4. The one added twist is to remove the duplicates where “researcher1” and “researcher2” switch places (ie., don’t list both A,B and B,A). The trick that we employ for this is that we note that people are ordered (by ID number in this example, though ordering by name works as well). So, we use the “|” operator in a comparison to guarantee that the person in “researcher1” in any given row must be sorted before the person in “researcher2”. This, as a side effect, exactly removes all duplicates.

**Q8.** Write a query that finds every organization that has grant support from all NSF directorates (e.g., CSE, ENG, etc.)

```
SELECT t1.ename, t1.cnt FROM (
    SELECT o.id AS oid, o.name AS oname, COUNT(DISTINCT directorate) AS cnt
    FROM orgs o, grant_programs gp, grants g, programs p
    WHERE g.org = o.id
        AND g.id = gp.grantid
        AND gp.programid = p.id
    GROUP BY o.id, o.name
) AS t1, (
    SELECT COUNT(DISTINCT directorate) AS cnt
    FROM programs
) AS t2
WHERE t1.cnt = t2.cnt;
```

oname	cnt
Massachusetts Institute of Technology	9
University of California-Berkeley	9
Carnegie-Mellon University	9
Stanford University	9
Harvard University	9

(5 rows)

Doing this all in one query is tricky; we saw many implementations where students counted the number of orgs, then inserted that count into a subsequent query. However, the challenge of this question was to put these two together. The idea of this example is to find the count of each org, and find the count of orgs, and filter for when the two are equal. It's also possible to filter by directorates, and exclude any org for which there is a grant that they do not get funding from; this is substantially more complex, though.