Contents

SQL Course Notes

- Course: Stanford Online Databases
- Instructor: Jennifer Widom

Reference Tables

Table 1: College

CName	state	enrollment
0	0	0

Table 2: Student

sID	sName	GPS	sizeHS
0	0	0	0

Table 3: Apply

sID	cName	major	decision
0	0	0	0

Basic SELECT FROM WHERE Queries

They take the form

```
Select A1,A2,...,An
From R1,R2,...,Rn
Where condition
```

A very simple starting query is this one:

```
select sID, sName, GPA
from   Student
where  GPA > 3.6;
```

Relational algebra is based on a set model but SQL is not, so a query will return duplicates. Override that with the `distinct` keyword.

```
select distinct sName, major
from   Student, Apply
where student.sID = Apply.sID;
```

Now add constraints to the join.

```
select sName, GPA, decision
from Student, Apply
where Student.sID = Apply.sID
  and sizeHS < 1000 and major = 'CS';
```

This query fails with an error of 'ambiguous column name cName' because cName occurs in both the College table and the Apply table.

```
select cName
from College, Apply
where College.cName = Apply.cName
  and enrollment > 20000 and major = 'CS';
```

When a column name occurs in two or more tables, all occurrences of it in the query must refer to the specific table of interest.

```
select distinct College.cName
from College, Apply
where College.cName = Apply.cName
  and enrollment > 20000 and major = 'CS';
```

Join all three relations and apply the join conditions to make sure we're talking about the same college and student. Then we get the big list of all the info.

```
select distinct Student.sID, sName, GPA, Apply.cName, enrollment
from Student, College, Apply
where Apply.sID = Student.sID and Apply.cName = College.cName;
```

SQL is an ordered model, so we can specify the order of the results. The default behavior is ascending, so descending needs to be explicitly called for with `desc`.

```
select Student.sID, sName, GPA, Apply.cName, enrollment
from Student, College, Apply
where Apply.sID = Student.sID and Apply.cName = College.cName
order by GPA desc, enrollment;
```

Like is a built-in predicate operator for string matching.

```
select sID, major
from Apply
where major like '%bio%';
```

To get all the attributes associated with a relation, use the asterisk.

```
select *
from Student, College;
```

We can use arithmetic within sql clauses, so here we boost the GPA if the student is from a big high school.

```
select sID, sName, GPA, sizeHS, GPA*(sizeHS/1000.0)
from Student;
```

To specify a column name, use `as`.

```
select sID, sName, GPA, sizeHS, GPA*(sizeHS/1000.0) as scaledGPA
from Student;
```

Table Variables

Table variables can rename variables in the FROM clause.

```
select Student.sID, sName, GPA, Apply.cName, enrollment
from Student, College, Apply
where Apply.sID = Student.sID and Apply.cName = College.cName;
```

Add variables to make the conditions easier to read. This doesn't change the outcome of the query.

```
select Student.sID, sName, GPA, Apply.cName, enrollment
from Student S, College C, Apply A
where A.sID = S.sID and A.cName = C.cName;
```

We want all pairs of students who have the same GPA. We need two instances of the Student relation and then we'll do the cross-product of those to get all the possible pairs.

```
select S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA
from Student S1, Student S2
where S1.GPA = S2.GPA;
```

But this returns all student pairs, including a student with themselves, because SQL is not a set model. To eliminate an entry where a student is paired with themself, we add the not equals (<>) condition.

```
select S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA
from Student S1, Student S2
where S1.GPA = S2.GPA and S1.sID <> S2.sID;
```

We still get each pair of students twice, so to handle that we just pick > or < for the condition. We get every pair of students once because we'll be listing the student with the smallest ID first.

```
select S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA
from Student S1, Student S2
where S1.GPA = S2.GPA and S1.sID < S2.sID;
```

Set Operators

Set operators union, intersect, and the except operator.

Here's a query to list names of students with names of colleges.

```
select cName from College
union
select sName from Student;
```

But in that case, SQL returned a single column with all entries in both College.cName and Student.sName listed in alphabetical order. The column is labeled cName because that came first in the query. We can rename the column.

```
select cName as name from College
union
select sName as name from Student;
```

This result is sorted! The union operator eliminates the duplicates in the result automatically because it's SQLite and it sorts and removes dupes automatically. Depending on the system, this might be different. This query returns all entries, including all the duplicates.

```
select cName as name from College
union all
select sName as name from Student;
```

This query returns a sorted result on any system.

```
select cName as name from College
union all
select sName as name from Student
order by name;
```

This query uses the intersect operator to get all the students who have applied to both CS and EE.

```
select sID from Apply where major = 'CS'
intersect
select sID from Apply where major = 'EE';
```

This query does the same thing, but the long way by joining the tables. This will return all the duplicates that we didn't get when running the intersect operator.

```
select A1.sID
from Apply A1, Apply A2
where A1.sID = A2.sID and A1.major = 'CS' and A2.major = 'EE';
```

Adding distinct will solve the duplicates problem.

```
select distinct A1.sID
from Apply A1, Apply A2
```

```
where A1.sID = A2.sID and A1.major = 'CS' and A2.major = 'EE';
```

Now to find students who applied to CS but not to EE. That requires the minus or difference operator, called except in the SQL standard.

```
select sID from Apply where major = 'CS'
except
select sID from Apply where major = 'EE';
```

Rewriting that query without using the except operator. This is a self-join of Apply with Apply. But this query actually returns students who applied to CS and to any other major except EE, which would include CS itself.

```
select distinct A1.sID
from Apply A1, Apply A2
where A1.sID = A2.sID and A1.major = 'CS' and A2.major <> 'EE';
```

It's not possible to write the query we had without the except operator.

Subqueries in the WHERE clause

Subqueries are nested Select statements within the Where condition.

This query finds the IDs and names of students who have applied to major in CS at some college.

```sql
select sID, sName
from Student
where sID in (select sID from Apply where major = 'CS';);
```

We can do this by taking the Student and Apply tables, joinng them to be sure we're talking about the same student, and then get their ID and their name.

```sql
select Student.sID, sName
from Student, Apply
where Student.sID = Apply.sID and major = 'CS';
```

The join returns the student IDs multiple times, once for every school they applied to. On the other hand, the subquery returns only the set. Fix the duplicates in the join by adding distinct.

```sql
select distinct Student.sID, sName
from Student, Apply
where Student.sID = Apply.sID and major = 'CS';
```

Get only the names of the students using a subquery.

`‘sql select sName from Student where sID in (select sID from Apply where major = 'CS';`

Now get the unique names of the students using a join. In the class example, there are two Craigs with different student ID numbers. Because this query doesn't take into account the ID numbers, it returns a list of names where the name ‘Craig' is only listed once, even though there are two Craigs represented by that data.

```sql
select distinct sName
from Student, Apply
where Student.sID = Apply.sID and major = 'CS';
```

Now looking for GPA only. This query works perfectly.

```sql
select GPA
from Student
where sID in (select sID from Apply where major = 'CS';
```

The same query using the join has a problem where it returns student GPAs multiple times. If you were to take the average, it would be a distorted average. If we make it distinct, we eliminate duplicates where we wanted duplicates.

```
select distinct sName
from Student, Apply
where Student.sID = Apply.sID and major = 'CS';
```

The difference operator. Students who have applied to major in CS but have not applied to major in EE. This is not possible to write this one with a join. You have to use subqueries in the where clause.

```
select sID, sName
from Student
where sID in (select sID from Apply where major = 'CS')
  and sID not in (select sID from Apply where major = 'EE');
```

You can also apply the not before the whole second where condition:

```
select sID, sName
from Student
where sID in (select sID from Apply where major = 'CS')
  and not sID in (select sID from Apply where major = 'EE');
```

So far we've used in and not in. We can also apply the exists query to test if a subquery itself is empty or not empty. This also introduces the correlated reference, where the subquery refers to the value C1 from outside that subquery.

For each college, we're going to check if there's another college in the same state… as long as that college is not being compared to itself.

```
select cName, state
from Colllege C1
where exists (select * from College C2
              where C2.state = C1.state
              and C1.CName <> C2.CName);
```

We want the largest value of this type. Find the college with the highest enrollment.

```
select cName
from College C1
where not exists (select * from College C2
                  where C2.enrollment > C1.enrollment);
```

Now to find the student (or students) with the highest GPA.

```
select sName, GPA
from Student C1
where not exists (select * from Student C2
                  where C2.GPA > C1.GPA);
```

It's not possible to get just the high GPA students with a join, but you can get all the students with GPAs greater than the lowest GPA:

```sql
select distinct S1.sName, S1.GPA
from Student S1, Student S2
where S1.GPA > S2.GPA;
```

This returns the student with highest GPA. All lets you check if a value has a relationship with all the results of a subquery.

```sql
select sName, GPA
from Student
where GPA >= all (select GPA from Student);
```

Let's use greater than all… this won't return anything, because we have multiple 3.9 GPAs. This query only works to check if you have one single valid result.

All tests a condition against every element in the result of a subquery. The condition is true if it's satisfied by every element.

```sql
select sName
from Student S1
where GPA > all (select GPA from Student
                where S2.sID <> S1.sID);
```

The any keyword. Give me all the college where it's not the case that enrollment is less than or equal to any other college. This also returns the college with the highest enrollment. It returns the unique case if using < and all the top enrollment cases if using <=.

Any tests a condition against every element in the result of a subquery. The condition is true if it's satisfied by one or more elements.

```sql
select cName
from College C1
where not enrollment > any (select enrollment from College C2
                        where C2.cName <> C1.cName);
```

Find all students who are not from the smallest high school.

```sql
select sID, sName, sizeHS
from Student
where sizeHS > any (select sizeHS from Student);
```

SQLite doesn't support any and all operators. We can use those using exists and not exists.

Any and all are convenient but not necessary. We can always write an equivalent query using exists and not exists.

```sql
select sID, sName, sizeHS
from Student S1
where exists (select * from Student S2
              where S2.sizeHS < S1.sizeHS);
```

Use the any operator to find the students who applied to CS but not EE. Here we find the students whose sID is in the set of students who applied to CS and whose sID is not in the set of students who applied to EE.

```
from Student
where sID = any (select sID from Apply where major = 'CS')
  and not sID = any (select sID from Apply where major = 'EE');
```

Subqueries in FROM and SELECT