SQL

The Author

December 30, 2016

1 Introduction

Structured Query Language. A declarative language to handle a relational approach to DBMS. Declarative: tell the computer what you want as a result; not how to do it, that will be the computer's problem. Thus, the importance of the query optimizer: takes a query written in SQL language and it figures out the best way, the fastest way, to execute that on the database.

There are two parts to the sequel language: the data definition language (DDL), used mainly to create the relations in the first place; and the data manipulation language, used to query the relations.

2 Select Statement

The Select Statement consists of three basic clauses:

```
Select A1, A2,..., AN
From R1,..., RM
Where condition
```

The select says what attributes you want returned; the from says which relations to use in a cross product where you will later on enforce a $\sigma_{condition}$. That is, in relational algebra is the equivalent of:

$$\pi_{A_1,\ldots,A_N}\sigma_{condition}(R_1 \times R_2 \times \ldots \times R_M)$$

As a result of the cross product, we may get duplicate values. To get rid of them:

```
Select distinct A1, A2,..., AN
From R1,..., RM
Where condition
```

Note that in the where condition the different statements are joined by connectors only, not commas. e.g., Student.ID = Apply.ID and sizeHS ¡ 1000 and major = 'CS' and cname = 'Stanford';.

2.1 Order

SQL is at its heart an unordered model. If we care about the order, SQL provides a clause, order by, to order the tuples according to the value they take at some attributes. e.g.,

```
Select A1, A2,..., AN
From R1,..., RM
Where condition
Order By A1
```

This will order the tuples in ascending order. If we want the order to be descending:

```
Select A1, A2,..., AN
From R1,..., RM
Where condition
Order By A1 desc
```

For a set of attributes, separate them with a comma:

```
Select A1, A2,..., AN
From R1,..., RM
Where condition
Order By A1 desc, A2
```

2.2 Like

Like is a built-in operator in SQL that allows us to do simple string matching on attribute values. For example, to catch all applications to a major that had anything to do with bio:

```
Select sID, major
From Apply
Where major like '%bio%'
```

2.3 Select*

To get all the attributes from the resulting operation, use *:

```
Select *
From R1,..., RM
Where condition
```

2.4 Arithmetic

We can create attributes that are functions of old attributes by stating them in the select clause. For example:

```
Select sID, sName, GPA, sizeHS, GPA*(sizeHS/1000.0)
From R1,..., RM
Where condition
```

To change the labelling of the new attribute in the schema, do the following:

```
Select sID, sName, GPA, sizeHS, GPA*(sizeHS/1000.0) as scaledGPA
From R1,..., RM
Where condition
```

3 Table Variables and Set Operators

3.1 Table Variables

Table variables are constructs that allow us to name the different relationships we're using with shorter, more readable names. For example:

```
Select *
From Student, Apply
Where Student.ID = Apply.ID
 and sizeHS < 1000
 and major = 'CS'
 and cname = 'Stanford';
```

Can be written as:

```
Select *
From Student S, Apply A
Where S.ID = A.ID and sizeHS < 1000
and major = 'CS'
and cname = 'Stanford';
```

The construct is more useful when we must have a cartesian product and a relation itself. For example, let's find all the pair of students with the same GPA (take from the cartesian product that they are different students).

```
Select *
From Student S1, Student S2
Where S1.GPA = S2.GPA and S1.ID <> S2.ID;
```

However, this would list every pair of students twice. To have them only once, write:

```
Select *
From Student S1, Student S2
Where S1.GPA = S2.GPA and S1.ID < S2.ID;
```

Here, we only take the tuples such that the students have the same GPA, have different s.ID, but we only take each pair once, where the student with the lowest id is put first.

3.2 Set Operators

Remember, in relation algebra set operators are used to append, at the end, a new relation. For example:

```
Select cName as Name From Student
Union
Select cName as Name From College;
```

The union operator in SQL eliminates duplicate results. To override this use:

```
Select cName as Name From Student
Union all
Select cName as Name From College;
```

The intersect operator. Let's get the Student ID of all the students who applied both to EE and CS:

```
Select sID Apply Where major = 'CS'
Intersect
Select sID From Apply Where major = 'EE';
```

The difference is used in SQL as except. The sID of all the students that applied to CS but not to EE:

```
Select sID Apply Where major = 'CS'
Except
Select sID From Apply Where major = 'EE';
```

4 Subclasses: Where Clause

A subquery is a nested select statement within the where condition of another select statement.

4.1 In, not in Operators

An example, let's find the student id of all the students who applied to CS at some college:

```
Select sID, sNAME
From Student
Where sID in (Select sID From Apply Where major = 'CS');
```

Remember: Always select the attributes that define the key in the resulting relation. Thus, you'll be able to get which are duplicate values and which are not.

Things can get more interesting when we realize we can use multiple subqueries and connect them with the connectors. Students that applied to CS but not EE.

```
Select sID, sName
From Student
Where sID in  (Select sID From Apply Where major = 'CS')
and not in (Select sID From Apply Where major = 'EE');
```

4.2 Exists operator

The exists operator let us check that a relationship is not empty. For example, let's find all colleges that are not unique in the state; that is, some other college exists in the state.

```
Select cName, state
From College C1
Where exists (Select * From College C2 Where C1.state = C2.state
and C1.cName <> C2.cName);
```

As another example, we can use the not exists operator such that does not exist another relation such that a condition. That is, for each tuple, we will keep it if the relation with its attributes does not exist. For example, the college with the highest enrollment.

```
Select cName, state
From College C1
Where not exists (Select * From College C2 Where
C2.enrollment > C1.enrollment);
```

4.3 all operator

We check, for each tuple, whether it holds between it and all the other tuples in a relationship.

```
Select sName, GPA
From Student
Where GPA >= all (select GPA from Student)
```

The college with the highest enrollment can be re-written as:

```
Select cName, enrollment
From College C1
Where enrollment > all (Select enrollment
From College C2 Where C1.cname <> C2.cname);
```

4.4 any operator

The any operator check, for each tuple, whether for any of the tuples in the subquery a relationship holds between it and the tuples in the subquery. For example, let's find the

college with the minimum enrollment; that is, the college such that there's not another college with a lower enrollment.

```
Select cName, enrollment
From College C1
Where enrollment > not any (Select enrollment
From College C2 Where C1.cName <> C2.cName);
```

Another example, the query for the students that do not come from the smallest high school.

```
Select sID, sName, sizeHS
From Student
Where sizeHS > any (select sizeHS
From Student);
```

Let's write the last query with the exists operator:

```
Select sID, sName, sizeHS
From Student S1
Where exists (Select * From Student S2
Where S1.sizeHS > S2.sizeHS)
```

5 Subqueries: in From and Select

Using a sub-query in the From: generate one of the tables that we're going to use in the general query. Using subquery in the select, produces the value that comes out of the query.

5.1 in From

For example, let's compute a relation where we take all the students such that the difference between their GPA and their non-scaled GPA changes by more than one.

```
Select sID, GPA, (GPA*(sizeHS/1000.0)) as scaledGPA
From Student
Where abs(GPA - GPA*(sizeHS/1000.0)) > 1
```

That can be abbreviated if we have a relation with scaledGPA in the first place. For example:

```
Select *
From (Select sID, GPA, (GPA*(sizeHS/1000.0)) as scaledGPA
From Student)
Where abs(GPA - GPA*(sizeHS/1000.0)) > 1
```

5.2 in Select

Any subquery can be put in the select clause as long as the subquery always returns a single tuple (row).

6 Joins

Fundamentally, there are two types of joins in SQL: inner joins, that follow what we saw in relational algebra. And outer joins (left, right and outer), where a theta joins except that the result from one or more tables, when they don't match the condition, are still added.

To express this joins in SQL, instead of adding the relations in the from clause separated by commas, we'll use them with explicit words. For example, a theta join can be expressed as:

```
Select *
From Student inner join Apply
on Student.sID = Apply.sID;
```

We can still further use the Where condition after the on clause. For a natural join:

```
Select *
From Student join Apply using (sID)
```

With using, you specify the attributes that you want to equate. On the other hand, outer joins:

```
Select *
From Student left outer join Apply using (sID)
```

For the outer join in Pandas, tydir:

```
Select *
From Student full outer join Apply using (sID)
```

NONE OF THE OUTER JOINS ARE ASSOCIATIVE: BE CAREFUL!!!

7 Aggregation

Functions over values in multiple rows: min, max, sum, avg, count etc.. The aggregate functions allow us to add two new clauses: group by, that allows us to partition our database in groups and perform the aggregation group-wise; the having, test filters on the result of aggregate values.

The simplest aggregation:

```
select avg(GPA)
From Student;
```

For counting the number of distinct attributes in a relation:

```
select count(distinct sID)
from apply
where cName = 'Cornell';
```

A more complicated example. The difference in the average GPA between CS applicants and nonCS applicants:

```
select CS.avgGPA - nonCS.avgGPA
from (select avg(GPA) as avgGPA from Student
where sID in (select sID from Apply Where major = 'CS') ) as CS ,
(select avg(GPA) as avgGPA from Student
where sID not in (select sID from Apply Where major = 'CS' )) as nonCS;
```

7.1 Group By

We partition the relation and group tuples according to the values they take in an attribute:

```
Select cName, count(*)
From Apply
group by cName
```

Always select the variable you are grouping with. For example, let's find the college and the major with the largest spread in the GPA of its applicants:

```
Select cName, major, max(mx - mn)
From (Select cName, major, max(GPA) as mx, min(GPA) as mn
From Student inner join Apply using (sID)
group by cName, major) M;
```

7.2 Having By

The having by clause allows us to specify a condition to filter out the resulting groups after the aggregation:

```
Select cName
From Apply
group by cName
having by count(*) < 5;
```

For example, let's find the majors such that the GPA of their maximum applicants is lower than the overall mean GPA.

```
Select major
From Apply inner join Student using(sID)
```

```
group by major
having by max(GPA) < (select avg(GPA)
From Student)
```

8 Null

The existence of null values determines that every expression is evaluated in a three way logic. That is: every expression can be either true, false or unknown. Thus, be careful how you write logical expressions. SQL has a way to see if something is null: GPA is null. GPA is not null.

9 Data Modification Statements

Inserting, deleting and updating data.

9.1 Insert

```
Insert into Table
Values (A1, A2, ..., An)
```

Where values is the values of the tuple you want to insert. Or:

```
Insert into Table
Select Statement
```