# SQL: Sets and Bags

Note: In these examples, we use SQL commands that have not been covered yet in the course, such as `CREATE TABLE` and `INSERT INTO`. We will learn about them shortly, and in fact you don't need to understand these commands in any detail yet. Our goal here is to learn how and when duplicates can exist in a table, and can be returned by a query.

## Tables in SQL can have duplicate rows

Let's define a table Blah(x, y), with x as the primary key, and put some values into it:

```
csc343h-prof=> CREATE TABLE Blah(x int primary key, y int);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "blah_pkey" for table "blah"
CREATE TABLE

csc343h-prof=> INSERT INTO Blah VALUES
csc343h-prof-> (1, 3), (2, 4), (7, 9), (8, 9);
INSERT 0 4
csc343h-prof=> SELECT * FROM Blah;
 x | y
---+---
 1 | 3
 2 | 4
 7 | 9
 8 | 9
(4 rows)
```

Because `x` has been declared to be the primary key for table `Blah`, SQL will not allow any duplicate values for `x`, the rest of the row is different. Let's confirm that SQL enforces this.

```
csc343h-prof=> INSERT INTO Blah VALUES (1, 88);
ERROR:  duplicate key value violates unique constraint "blah_pkey"
```

It follows that we can't repeat a whole row, because that would mean repeating the `x` values.

```
csc343h-prof=> INSERT INTO Blah VALUES (1, 3);
ERROR:  duplicate key value violates unique constraint "blah_pkey"
```

So table `Blah` cannot have duplicate rows. But this is only due to having a primary key. In fact, SQL permits a table to have duplicate rows if this does not violate any constraint that has been defined. Here's an example:

```
csc343h-prof=> -- Here we define a table with no key.
csc343h-prof=> CREATE TABLE Keyless (x int, y int);
CREATE TABLE
csc343h-prof=> -- Let's put the same values into it as the first table.
csc343h-prof=> INSERT INTO Keyless VALUES
csc343h-prof-> (1, 3), (2, 4), (7, 9), (8, 9);
INSERT 0 4

csc343h-prof=> -- (a) Now we can repeat an x-value!  No problem.
csc343h-prof=> INSERT INTO Keyless VALUES (1, 88);
INSERT 0 1
csc343h-prof=> SELECT * FROM Keyless;
 x | y
---+----
 1 |  3
 2 |  4
 7 |  9
 8 |  9
 1 | 88
(4 rows)
csc343h-prof=> -- (b) We can repeat an entire row, in fact!
csc343h-prof=> INSERT INTO Keyless VALUES (1, 3);
INSERT 0 1
csc343h-prof=> SELECT  * FROM Keyless;
 x | y
---+----
 1 |  3
 2 |  4
 7 |  9
 1 | 88
 1 |  3
(5 rows)
```

We have created a table that stores duplicate rows.

## Tables in SQL are "bags"

We can have duplicate rows because a table in SQL is not a set, it is a bag. In mathematics, a bag (also known as a "multiset") is a generalization of the concept of a set; it allows duplicates. As with sets, order of the elements in a bag doesn't matter, so {6, 2, 7, 2, 2, 1, 9} is the same bag as {1, 2, 2, 2, 6, 7, 9}, for example.

## SELECT-FROM-WHERE queries operate under bag semantics

Even for a table that has a primary key (and therefore can't have duplicate rows), we can use it in a

Even for a table that has a primary key (and therefore can't have duplicate rows), we can use it in a query whose result has duplicate rows.

```
csc343h-prof=> -- Recall the table Blah from above, with primary key x.
csc343h-prof=> SELECT * FROM Blah;
 x | y
---+---
 1 | 3
 2 | 4
 7 | 9
 8 | 9
(4 rows)

csc343h-prof=> -- Blah has no duplicate rows, because it can't even repeat
csc343h-prof=> -- an x-value, due to its key.  But the following query on Blah
csc343h-prof=> -- reports a result with duplicate rows:
csc343h-prof=> SELECT y FROM Blah;
 y
---
 3
 4
 9
 9
(4 rows)
```

`SELECT-FROM-WHERE` statements leave duplicate rows by default. We say that they work under bag semantics. You may wonder why. One reason is that we may want the duplicates in the result because they tell us how many times something occurred. In addition, getting rid of duplicates is expensive! Imagine you have a table with n rows and you have to turn it from a bag into a set. How much work would that be? Now remember that n might be 100,000. SQL doesn't do the work of duplicate elimination unless we explictly ask for it.

## We can override the default

The keyword `DISTINCT` can be used to override the default behaviour of SELECT-FROM-WHERE queries and eliminate duplicates.

```
-- A query with duplicate results
csc343h-prof=> SELECT oid FROM Took WHERE grade > 95;
 oid
-----
  16
  11
  13
  39
  11
  13
  16
  22
   1
  14
(10 rows)

-- Order the results so we can see the duplicates more clearly.
csc343h-prof=> SELECT oid
csc343h-prof-> FROM Took
csc343h-prof-> WHERE grade > 95
csc343h-prof-> ORDER BY oid;
 oid
-----
   1
  11
  11
  13
  13
  14
  16
  16
  22
  39
(10 rows)

-- Introduce DISTINCT to say we don't want the duplicates.
csc343h-prof=> SELECT DISTINCT oid
csc343h-prof-> FROM Took
csc343h-prof-> WHERE grade > 95
csc343h-prof-> ORDER BY oid;
 oid
-----
   1
  11
  13
  14
  16
  22
  39
(7 rows)
```

## We can only ask for distinct rows overall, not by column

Consider the different kinds of duplication in this query's result:

```
csc343h-prof=> SELECT oid, grade
csc343h-prof-> FROM Took
csc343h-prof-> WHERE grade > 95
csc343h-prof-> ORDER BY oid;
```

```
 oid | grade
-----+-------
   1 |    99
  11 |    99          <-- (a) These rows are duplicates
  11 |    99          <-- (same oid and same grade)
  13 |    99
  13 |    98          <-- (b) These rows have the same grade,
  14 |    98          <-- but different oids.
  16 |   100          <-- (c) These rows have the same oid,
  16 |    98          <-- but different grades
  22 |    96
  39 |    97
(10 rows)
```

The duplicates in column oid occur in different rows than the duplicates in column grade, So SQL won't let us write a query that asks for both columns to be distinct.

```
csc343h-prof=> SELECT DISTINCT oid, DISTINCT grade
csc343h-prof-> FROM Took
csc343h-prof-> WHERE grade > 95
csc343h-prof-> ORDER BY oid;
ERROR:  syntax error at or near "DISTINCT"
LINE 1: SELECT DISTINCT oid, DISTINCT grade FROM Took WHERE grade > ...
                             ^
```

`DISTINCT` actually works at the level of the row, not individual cells. It turns the result of the query into a set, rather than a bag. So we can only say `DISTINCT` once, right before we list the columns that we want in the result.

```
csc343h-prof=> -- This query, with one 'distinct' before the column list,
csc343h-prof=> -- works:
csc343h-prof=> SELECT DISTINCT oid, grade
csc343h-prof-> FROM Took
csc343h-prof-> WHERE grade > 95
csc343h-prof-> ORDER BY oid;
 oid | grade
-----+-------
   1 |    99
  11 |    99
  13 |    98
  13 |    99
  14 |    98
  16 |    98
  16 |   100
  22 |    96
  39 |    97
(9 rows)
```

Notice that we have 9 rows now rather than 10. The `DISTINCT` removed the the entirely repeated rows, marked (a) above. The partial duplications were not removed.

## Aside: The other place where DISTINCT can go

Above, we used DISTINCT outside of any aggregation. We have seen DISTINCT before, *inside* the brackets on an aggregation, for example:

```
csc343h-prof=> SELECT count(DISTINCT sid) FROM Took;
 count
-------
     5
(1 row)
```

With DISTINCT in this context, we *can* use it more than once in the same query.

```
csc343h-prof=> SELECT count(DISTINCT sid), count(distinct oid) FROM Took;
 count | count
-------+-------
     5 |    23
(1 row)
```

## SQL offers set union, intersection and difference

With all this talk about sets, you may be wondering about the set operators you know. SQL offers union, intersection and difference:

```
csc343h-prof=> -- Set union:
csc343h-prof=> (SELECT sid, grade FROM Took WHERE grade > 95)
csc343h-prof-> UNION
csc343h-prof-> (SELECT sid, grade FROM Took WHERE grade < 50);
  sid  | grade
-------+-------
 11111 |    40
 99999 |    96
 99132 |    39
   157 |    99
 99999 |   100
 11111 |    45
 98000 |    98
 11111 |    46
 11111 |    17
 99132 |    98
 11111 |     0
   157 |    39
 99132 |    99
 99132 |    99
 98000 |    97
   157 |    98
 99999 |    99
(16 rows)

csc343h-prof=> -- Set intersection:
csc343h-prof=> (SELECT sid FROM Took WHERE grade > 95)
csc343h-prof-> INTERSECT
csc343h-prof-> (SELECT sid FROM Took WHERE grade < 50);
  sid
-------
 99132
   157
(2 rows)

csc343h-prof=> -- Set difference:
csc343h-prof=> (SELECT sid FROM Took WHERE grade > 95)
EXCEPT
(SELECT sid FROM Took WHERE grade < 50);
  sid
-------
 98000
 99999
(2 rows)
```

There are two important syntactic requirements to notice:

- An operand to a set operator must be surrounded by round brackets.
- An operand to a set operator must be a complete query. For instance, if we wanted to intersect tables Elephant and Flower, we couldn't just write `(Elephant) INTERSECT (Flower);` Instead we would have to write `(SELECT * FROM Elephant) INTERSECT (SELECT * FROM Flower);`

## Set operators require compatible schemas

As you probably expect, the two operands to a set operator must have compatible schemas. In the following example, `sid` is an integer and `instructor` is a kind of string (a `VARCHAR(40)` to be precise), so the query fails:

```
csc343h-prof=> (SELECT sid FROM Took) UNION (SELECT instructor FROM Offering);
ERROR:  UNION types integer and character varying cannot be matched
LINE 1: (SELECT sid FROM Took) UNION (SELECT instructor FROM Offerin...
                                              ^
```

What exactly is considered compatible will vary from DBMS to DBMS. Postgresql will, for instance accept these queries, but another DBMS may not:

```
csc343h-prof=> -- sid and oid are both of type integer, but the
csc343h-prof=> -- column names are not the same:
csc343h-prof=> (SELECT sid FROM Took) UNION (SELECT oid FROM Took);
  sid
-------
 98000
 99132
     8
    11
 99999
    16
    39
     3
    14
    17
    28
    15
   157
 11111
    31
    35
    34
     1
    26
    13
    22
     9
    27
    38
     6
    21
     5
     7
(28 rows)

csc343h-prof=> -- And here in table Offering, column 'name' is of type
csc343h-prof=> varchar(40), while 'dept' is of a user-defined type which
csc343h-prof=> is a restriction on varchar(20).
csc343h-prof=> (SELECT name FROM Course) UNION (SELECT dept FROM Course);

              name
--------------------------------
 Human Biol & Evol
 Mediaeval Society
 CSC for future prime ministers
 Environmental Change
 Intro Archaeology
 Intro to Databases
 Organisms in Environ
 Data Struct & Anal
 ENV
 EEB
 Black Freedom
 Natl & Intl Env Policy
 ANT
 HIS
 The Graphic Novel
 COBOL programming
 Compar Vert Anatomy
 CSC
 ENG
 Narrative
 Intro to Comp Sci
 Intro to Visual Computing
 Rhetoric
 Software Design
 Marine Mammal Bio
(25 rows)
```

## Set operators work under set semantics

We saw earlier that `SELECT-FROM-WHERE` queries work under bag semantics. In contrast, the set operators work under set semantics. Here's an example to demonstrate.

```
-- Here are two SELECT-FROM-WHERE queries that we'll union together in a moment.
-- The first one has duplicates (and they're left in because SELECT-FROM-WHERE
-- uses bag semantics by default).
csc343h-prof=> SELECT sid
csc343h-prof-> FROM Took
csc343h-prof-> WHERE grade > 95;
  sid
-------
 99132
 99132
 98000
 98000
 99999
 99999
 99999
 99999
   157
   157
(10 rows)

-- The second query has duplicates too.
csc343h-prof=> SELECT sid
csc343h-prof-> FROM Took
csc343h-prof-> WHERE grade < 50;
  sid
```

```
-------
 99132
   157
 11111
 11111
 11111
 11111
 11111
(7 rows)

-- But when we do union, we don't get all 17 rows.  The duplicates are
-- eliminated, by default.
csc343h-prof=> (SELECT sid FROM Took WHERE grade > 95)
csc343h-dianeh-> UNION
csc343h-dianeh-> (SELECT sid FROM Took WHERE grade < 50);
  sid
-------
 98000
 99132
 99999
   157
 11111
(5 rows)
```

## We can override the default

As we saw, the set operators use set semantics by default. We can override this with the keyword `ALL`. In that case, bag semantics are used, meaning that duplicates are *not* eliminated.

```
csc343h-prof=> -- UNION ALL says we want the duplicates.  Now we get
csc343h-prof=> -- all 17 rows.
csc343h-prof=> (SELECT sid FROM Took WHERE grade > 95)
csc343h-prof-> UNION ALL
csc343h-prof-> (SELECT sid FROM Took WHERE grade < 50);
  sid
-------
 99132
 99132
 98000
 98000
 99999
 99999
 99999
 99999
   157
   157
 99132
   157
 11111
 11111
 11111
 11111
 11111
(17 rows)
```

## The meaning of bag union, intersection and difference

The exact result of a union, intersection, or difference under bag semantics requires some thought. (But it makes total sense.)

For union, you can imagine two bags full of marbles. You might have a red marble, even several red marbles, in each bag. The union of the two bags is just what you get when you dump them together. So, mathematically, {1, 1, 1, 3, 7, 7, 8} ∪ {1, 5, 7, 7, 8, 8} = {1, 1, 1, 1, 3, 5, 7, 7, 7, 7, 8, 8, 8}. For intersection and difference, think of matching up individual elements across the bags. {1, 1, 1, 3, 7, 7, 8} ∩ {1, 5, 7, 7, 8, 8}= {1, 7, 7, 8} and {1, 1, 1, 3, 7, 7, 8} − {1, 5, 7, 7, 8, 8}= {1, 1, 3} For this last case (difference with bags), you take each marble from the second bag and if you can, remove a matching marble from the first. The result is what is left in the first bag.

## A special note about EXCEPT ALL

Whether we use `EXCEPT` or `EXCEPT ALL` determines not only how many times a row occurs in the result, but whether or not it occurs at all.

Suppose we have tables P and Q with these contents:

```
csc343h-prof=> SELECT * FROM P;
 a |  b
---+-----
 1 | 151
 2 | 123
 3 | 432
 1 | 333
 1 | 345
 4 | 912
 5 | 123
(7 rows)

csc343h-prof=> SELECT * FROM Q;
 a | c
---+----
 1 | 44
 3 | 88
 3 | 12
 9 | 12
```

```
(4 rows)
```

With EXCEPT ALL, we match values up individually. For example, when we remove the one 1 that we get out of Q from the three 1s that we get out of P, there are two 1s left.

```
csc343h-prof=> (SELECT a FROM P) EXCEPT ALL (SELECT a FROM Q);
 a
---
 1                  <-- There are two 1s in this result
 1
 2
 4
 5
(5 rows)
```

But with EXCEPT, a single occurence of a value for a in Q wipes out *all* occurences of it from P. So if we use EXCEPT in our query from before, every 1 value in P is removed as a result of a single 1 value in Q.

```
csc343h-prof=> (SELECT a FROM P) EXCEPT (SELECT a FROM Q);
 a
---
 2
 4                  <-- There are no 1s anywhere in this result
 5
(3 rows)
```

`EXCEPT` without `ALL` will always give you a set, because the default for set operators is set semantics. But notice that it is the set you would get from first transforming the result of each subexpression into a set and then applying the difference operator. This is not the same as what you would get from applying the difference using bag semantics and then transofrming the result into a set.