How to Check 2 SQL Tables are the Same

Today Stanley asked me a simple question: how can we check if the contents of two SQL tables are the same? Well, you just do `SELECT * FROM t1 = t2` ... wait, that's wrong, comparison doesn't work on entire tables in SQL. My second attempt is a bit better: if we take the difference of the table both ways, and end up with empty results, then they must be the same, right? In SQL: `SELECT * FROM t1 EXCEPT SELECT * FROM t2` (and the other way). Wrong again! Because `EXCEPT` takes the *set* difference, it will be empty if, say, `t1` contains 2 copies of a tuple, but `t2` contains only one.

I gave up a little bit and started searching online, but surprisingly there was not a single satisfying answer! The solutions online either suffer from the same issue as the `EXCEPT` query, or use some obscure features that are not standard SQL (e.g. `CHECKSUM` which doesn't really work anyways). How hard can it be to compare two tables in SQL?!

Intrigued, I posted the problem as a challenge to my colleagues: **Write a query, using only standard SQL features, to check if two tables are the same**. Here "same" means the two table contains the same set of distinct tuples, and every tuple has the same number of copies in each table. Formally, they are the same bag/multiset.

If you've read the SQL standard (and every "non-standard") cover to cover, you'll come with the following query after a few campari drinks: `SELECT * FROM t1 EXCEPT ALL SELECT * FROM t2`. The key is `EXCEPT ALL` which takes the "bag difference" similar to how `UNION ALL` takes the "bag union". Alas, `EXCEPT ALL` is not implemented by SQLite! And probably for good reasons: whereas `EXCEPT` can be compiled to just an anti-join, executing `EXCEPT ALL` probably requires keeping track of which copy of the same tuple we've seen, or keeping a count per distinct tuple.

A more "vanilla SQL" solution looks like this:

```
SELECT *, COUNT(*)
  FROM t1
 GROUP BY x, y, z, ... -- all attributes of t1

EXCEPT

SELECT *, COUNT(*)
  FROM t2
 GROUP BY x, y, z, ... -- all attributes of t2
```

Here, we group by all attributes of the table in order to explicitly mark every distinct tuple with its count. And because all tuples are distinct after the grouping, we can use `EXCEPT` to compare the results. That's pretty good! I should be happy about it and go back to work.

But I can't get over one small ugliness: I had to manually list all the attributes in the `GROUP BY` clause, since `GROUP BY *` doesn't work. This means we have to change the query for every new schema. "Fine," you say, "just generate the query and get back to work". Problem is, I don't feel like working today, so I invite myself to another challenge: **write a *single* query that does the job for every pair of tables, where we are only allowed to change the table names**.

TBH it's not surprising I got nerd sniped by this problem: half of my PhD dealt with equivalence, and one idea in fact lead to the final solution. **This key idea is to view a table in bag semantics as a vector of numbers, and view joins of tables as polynomials**. Specifically, consider sorting all the distinct elements, and the `i` th entry of the vector stores the count of the `i` th distinct element. For example, the table `t=[a, b, b, c, c]` beomes the vector `[1 2 2]`. Then, a self-join becomes point-wise multiplication of the vector with itself. Using the same example, `t NATURAL JOIN t` contains 1 copy of `a`, 4 copies of `b`, and 4 copies of `c`, `[1 4 4] = [1 2 2] * [1 2 2]`.

With this, we can connect repeated self-joins of a table with the moments of the vector (or power sum, or p-norm, if you're more familiar with those). That is, the query:

```
SELECT COUNT(*)
  FROM t NATURAL JOIN t
         NATURAL JOIN t
         ... -- total of p copies of t's
         NATURAL JOIN t
```

computes , where is the vector representation of `t` and is its length, i.e. the number of distinct elements in `t`. Abusing notation, we'll write that as

The above connection lets us use a very elegant result: for any two vectors of length , if then must be a permutation of . In other words, the moments uniquely determines a bag of values! See Appendix A of Abo Khamis et.al. for a very elegant proof.

Our game plan is now this: for each table, compute all moments and compare the results. We can do this with a recursive query:

```
CREATE TABLE t1_moments AS

WITH RECURSIVE r1 AS (
  -- first iteration, return t1 as-is
  SELECT 1 as i, t1.*
    FROM t1

  UNION ALL

  -- iterations i+1 joins together i+1 copies of t1
  SELECT r1.i + 1 AS i, t1.*
    FROM r1 NATURAL JOIN t1
  -- we could have stopped at COUNT(DISTINCT *) ...
  -- but that's not valid SQL :(
   WHERE i < (SELECT COUNT(*) FROM t1)
)

-- compute the moment with COUNT
SELECT COUNT(*) FROM r1 GROUP BY i;
```

After computing `t2_moments` in the same way, we can compare them with `EXCEPT` because they do not contain duplicates.

But that's not enough, since having the same moments only guarantees the vectors are permutations of each other. In terms of the original relation, the table `[a, b, b]` will be indistinguishable from the table `[a, a, b]`, because `[1 2]` has the same moments as `[2 1]`. To rule out this case, we use a simple fact from linear algebra: if is a permutation of and , then . In SQL, this means we need to take the natural join of `t1` with `t2` and compare the count with the self join of `t1` (and of `t2`):

```
SELECT (SELECT COUNT(*) FROM t1 NATURAL JOIN t1)
     - (SELECT COUNT(*) FROM t1 NATURAL JOIN t2)
    AS d WHERE d <> 0;

SELECT (SELECT COUNT(*) FROM t2 NATURAL JOIN t2)
     - (SELECT COUNT(*) FROM t1 NATURAL JOIN t2)
    AS d WHERE d <> 0;
```

See the complete query at the end of this post. All together, the query uses only standard SQL features, and to use it for a new pair of tables we only need to change the table names. Of course, it is completely impractical for any table of decent size (it runs in time $O(N^N)$), but that's not the point :)

But even the simpler query using `GROUP BY` was not trivial to come up with, which brings the question: why isn't it a standard feature of SQL to just compare two tables? I imagine it can be very useful for testing, e.g. you write a simple but slow version, and check a more complex but fast version returns the same result.

```
CREATE TABLE t1 (x INTEGER);
CREATE TABLE t2 (x INTEGER);

INSERT INTO t1 VALUES (1), (1), (2), (3);
INSERT INTO t2 VALUES (2), (1), (3), (2);

-- If t1=t2 (meaning they are the same bag/multiset), then the following should r

-- Sanity check: do they contain the same *set* of elements (ignoring duplicates)

SELECT * FROM t1 EXCEPT SELECT * FROM t2;
SELECT * FROM t2 EXCEPT SELECT * FROM t1;

-- Now compare the moments/power sums

CREATE TABLE t1_moments AS

WITH RECURSIVE r1 AS (
  -- First iteration, return t1 as-is
  SELECT 1 AS i, t1.*
    FROM t1

  UNION ALL

  -- Iterations i+1 joins together i+1 copies of t1
  SELECT r1.i + 1 AS i, t1.*
    FROM r1 NATURAL JOIN t1
  -- We could have stopped at |t1| (number of distinct elements in t1)
   WHERE i < (SELECT COUNT(*) FROM t1)
)

-- Compute the power sum with COUNT
SELECT COUNT(*) FROM r1 GROUP BY i;


-- Repeat for the other table...
CREATE TABLE t2_moments AS

WITH RECURSIVE r2 AS (
  SELECT 1 AS i, t2.*
    FROM t2

  UNION ALL

  SELECT r2.i + 1 AS i, t2.*
```

```
    FROM r2 NATURAL JOIN t2
   WHERE i < (SELECT COUNT(*) FROM t2)
)

SELECT COUNT(*) FROM r2 GROUP BY i;

SELECT * FROM t1_moments EXCEPT SELECT * FROM t2_moments;
SELECT * FROM t2_moments EXCEPT SELECT * FROM t1_moments;

-- To rule out the case where they have the same moments, but are "permutations"
SELECT (SELECT COUNT(*) FROM t1 NATURAL JOIN t1) - (SELECT COUNT(*) FROM t1 NATUR
SELECT (SELECT COUNT(*) FROM t2 NATURAL JOIN t2) - (SELECT COUNT(*) FROM t1 NATUR
```