

Some problem solving and how it's easier with python

admin - Monday, March 23rd, 2009

There's a project I work on that required me to make an import utility for a CRM. The import should get a comma separated values file of clients and information about clients, and save it to the database. The database is split across several tables, so in the `clients` table I normally don't keep the name of the company, but just a foreign key. Now, our client is not very good with numbers and she needed to import files in which she could enter the name of the company instead of the database ID. A spreadsheet row representing a client looks like this:

```
FirstName | LastName | Email          | Company
John      | Doe      | johndoe@example.com | Coca Cola
```

But the database row in the `clients` table looks like this:

```
first_name | last_name | email          | company_id
John       | Doe       | john@example.com | 2
```

What I need to do is search for the company named 'Coca Cola' in the `companies` table and replace the name with it's ID. This is all fine except for one problem - typos. Moreover, the user could write "Apple Computer Inc." instead of "Apple Inc.". So I needed a way to compare the input strings with the ones in the database.

After poking around I found out about the [Levenshtein distance](#) between strings, but that solved only half of my problems - the typo part. The distance would be very small between "Apple" and "Aple" but very big between "ACME International Inc." and "International ACME Inc.", and the latter two are obviously the same.

I devised the following method to compare entries:

- Split up the terms by words and eliminate blanks
- Get the Levenshtein distance between each word from the first term and each word from the second term. Comparing "Apple Computer Inc." with "Apple Inc." for example, will give a matrix of 6 distances. ☒
- Get the shortest term (one with less words, not the one with less characters). It has 2 words in this case. Then choose the smallest values from each **row**. When you pick the smallest **row** value, you cannot pick anymore values from that **column**. This means that the word in the **column** is the best match for some word in the **rows**.
- Add these values up and add the difference between the word count of the 2 terms - and you have a score for the similarity of the terms. If the score is zero, they are the same. We are adding +1 for each extra word, but this can be weighted if needed. The point is that we don't care much for extra words since company names can have many words in them, but they are often called by one or two words.

But there is a problem with step 3. If, for example, a column has the lowest values for more than one row, we always choose the first, and this practice is not always the best answer. For instance, matching "Fast Cats" with "Fats Cats" (notice the typo) gets a total score of 3 - matching **Cats** to **Fats** and **Fast** to **Cats**, which is wrong - it will be 2 if we match **Fast** to **Fats** and **Cats** to **Cats**, which is the intended solution.



So to be sure we have the best match, we need to always have the lowest sum that is unique across rows and columns. One solution is to make all permutations of the words in the **columns** and join them to a single permutation of the words **in the rows** then see which one has the lowest score. If the words in the rows are fewer then we need to get all permutations **P(n,k)** of the words in **the columns**, where **n** is the number of columns and **k** is the number of rows. This is a $O(n!)$ algorithm but it's the best that I could think of - practically the same problem as finding every possible way to place 8 rooks on a chess table without making them attack each other.

And finally, here is the part where we get to write some code. I need a function that can calculate all permutations consisted of **k** elements out of a larger set consisted of **n** elements ($k \leq n$).

I decided first to write the algorithm in Python because it's cleaner and easier to think, and then to rewrite it in PHP. The first attempt

was really, really sucky and I won't talk about it because I'm a bit embarrassed. But I wasn't aware of a neat thing that Python has: the **yield** statement. The darn thing can be written in 6 lines with it:

```
def permutations(the_set, n):
    if n==0:
        yield []
    else:
        for i in xrange( len( the_set ) ):
            for x in permutations
            ( the_set[0:i] + the_set[i+1:], n-1 ):
                yield [the_set[i]]+x
```

I will go into the yield statement later, maybe I will extend this post, but for now, I'll say that it allows you to make a function that will calculate the combinations on the fly, without storing them in a huge list and then returning the list. It sort of lazy-loads the list of combinations when needed. There is no such thing in PHP (as far as I know). So here's my best shot at the function in PHP:

```
function permutations( $array, $size )
{
    $result = array();
    $x = count($array);
    for( $i=0; $i<$x; $i++ ) {
        $copy = $array; // copy: array_splice gets the arg by reference
        $item = array_splice( $copy, $i, 1 );
        if( $size == 1 )
            $result[] = $item;
        else {
            $rest = permutations( $copy , $size - 1 );
            foreach( $rest as $r )
                $result[] = array_merge($item, $r);
        }
    }
    return $result;
}
```

There really are excessive parts of the PHP code like storing the final result, but more importantly copying the array each time because `array_splice` takes the array argument by reference and modifies it (talking about orthogonality), plus its twice as long as the python code and half as readable.

Anyway, to get back at my original problem - the solution worked in terms of accuracy (at least for the first few test cases), but I fear it's going to be slow for large datasets. I have around 7 fields to compare with each respective table of the database, each table having 100 records on average; each record is 3 words long on average which gives 6 permutations per comparison. Importing a list of 1000 clients would require $1000 \times 7 \times 100 \times 6 = 4,200,000$ comparisons, plus 700,000 calls to the permutations function (not counting the recursive calls :). I still think that it's better than hammering the database with 7000 fulltext search queries, not to mention moving the database tables to MyISAM and indexing a bunch of fields. After all, it's an import. I could put one of those useless progress indicators like when you're starting Windows.