**Problem 3: Working with a Term-Document Matrix**

The reuters dataset can be considered a *term-document matrix*, which is an important representation for text analytics.

Each row of the matrix is a *document vector*, with one column for every term in the entire corpus. Naturally, some documents may not contain a given term, so this matrix is sparse. The value in each cell of the matrix is the term frequency. (You'd often want this this value to be a weighted term frequency, typically using tf-idf -- term frequency-inverse document frequency. But we'll stick with the raw frequency.)

What can you do with the term-document matrix *D*? One thing you can do is compute the similarity of documents. Just multiply the matrix with its own transpose *S = DDT*, and you have an (unnormalized) measure of similarity.

The result is a square document-document matrix, where each cell represents the similarity. Here, similarity is pretty simple: if two documents both contain a term, then the score goes up by the product of the two term frequencies. This score is equivalent to the dot product of the two document vectors.

To normalize this score to the range 0-1 and to account for relative term frequencies, the *cosine similarity* is perhaps more useful. The cosine similarity is a measure of the angle between the two document vectors, normalized by magnitude. You just divide the dot product by the magnitude of the two vectors. However, we would need a power function (x^2, x^(1/2)) to compute the magnitude, and sqlite has [built-in support for only very basic mathematical functions](http://www.sqlite.org/lang_corefunc.html). It is not hard to[extend sqlite to add what you need](https://www.google.com/search?q=math+function+extensions+sqlite), but we won't be doing that in this assignment.

**(h) similarity matrix:** Write a query to compute the similarity matrix *DDT*. (Hint: The transpose is trivial -- just join on columns to columns instead of columns to rows.) The query could take some time to run if you compute the entire result. But notice that you don't need to compute the similarity of both (doc1, doc2) and (doc2, doc1) -- they are the same, since similarity is symmetric. If you wish, you can avoid this wasted work by adding a condition of the form a.docid < b.docid to your query. (But the query still won't return immediately if you try to compute every result.)

What to turn in: On the assignment website, turn in a text document, similarity\_matrix.txt, which has the similarity of the two documents '10080\_txt\_crude' and '17035\_txt\_earn'.

You can also use this similarity metric to implement some primitive search capabilities. Consider a keyword query: It's a bag of words, just like a document (typically a keyword query will have far fewer terms than a document, but that's ok).

So if we can compute the similarity of two documents, we can compute the similarity of a query with a document. You can imagine taking the union of the keywords represented as a small set of (docid, term, count) tuples with the set of all documents in the corpus, then recomputing the similarity matrix and returning the top 10 highest scoring documents.

**(i) keyword search:** Find the best matching document to the keyword query "washington taxes treasury". You can add this set of keywords to the document corpus with a union of scalar queries:

SELECT \* FROM frequency

UNION

SELECT 'q' as docid, 'washington' as term, 1 as count

UNION

SELECT 'q' as docid, 'taxes' as term, 1 as count

UNION

SELECT 'q' as docid, 'treasury' as term, 1 as count

Then, compute the similarity matrix again, but filter for only similarities involving the "query document": docid = 'q'. Consider creating a view of this new corpus to simplify things.

What to turn in: On the assignment website, turn in a text document, keyword\_search.txt, with the maximum similarity score between the query and any document. Your query should return a list of (docid, similarity) pairs, but you will submit only a single number: the highest score in the list.