Interactive - 38 - key word lookup

(Note this set of information is the basis for Assignment 03 on keywords)

PostgreSQL includes an extensive set of tools for key word lookup. The word lookup is efficient and takes account of items like language and root words.

With regular SQL you have a pattern matching operator, `LIKE`.

```
SELECT real_name
    FROM name_list
    WHERE real_name  LIKE '%ilip%'
;
```

The `%` is wild card. Because of the use of a wild car this kind of a query will usually result in a full table scan. In some cases it is possible to use a GIST index on the field and that will make the search faster.

PostgreSQL has an extended set of patterns that allows matching a wider set of values. This is with the `SIMILAR TO` operator.

```
SELECT real_name
    FROM name_list
    WHERE real_name SIMILAR TO '(p|P)hilip%'
;
```

PostgreSQL also includes the ability to use regular expressions on text fields. This uses POSIX regular expressions. The Operator is the tilde `~` operator.

```
SELECT real_name
    FROM name_list
    WHERE real_name ~ '[pP]hilip.*'
;
```

None of these takes account of things like “San Francisco” is a single “term” in the English language. Also things like “the” are not words that need to be indexed. Other languages like French have similar constructs. PostgreSQL has an extensive set of capabilities for efficiently searching and indexing text data.

Comparable tools are Lucine and Elastisearch. Google also provides dedicated hardware for this kind of search. All of these alternatives are very expensive. Many tolls can benefit from a search like this and at a cost of close to 0 (PostgreSQL is open source) this makes for a very interning alternative.

tsvector

The database uses a pair of values and an index. The first of these is a `tsvector`. This converts text into a set of words and locations.

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog.');
```

Will result in

```
                      to_tsvector
-------------------------------------------------------
 'brown':3 'dog':9 'fox':4 'jump':5 'lazi':8 'quick':2
(1 row)
```

This set of data has "The" eliminated the "word" location for each word. The words are converted into the root words. "lazy" is confuted to "lazi" the root word.

tsquery

`tsquery` is the query side of this along with the `@@` operator.

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')
    @@ to_tsquery('fox')
;
```

Will return 't' for true.

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')
    @@ to_tsquery('bob')
;
```

Will return 'f' for false.

The query processing also allows for operators in a query.

```
SELECT to_tsvector('The quick brown fox jumped over the lazy dog')
    @@ to_tsquery('fox | duck')
;
```

Will return 't' for true.

There are and, `&`, or, `|` and not, `!` operators.

As a table

Let's put the tsvector data type into a table and add some data.

Run the file `hw38_07.sql`.

```
DROP TABLE if exists indexed_docs ;

CREATE TABLE indexed_docs (
    doc_id UUID NOT NULL DEFAULT uuid_generate_v4() primary key,
    document_title TEXT NOT NULL,
    document_body TEXT NOT NULL,
    document_tokens TSVECTOR
```

```
);

INSERT INTO indexed_docs ( document_title, document_body ) values
    ( 'On Tyrany', 'A book about how to stop tyrants and how to deal with the devaluation of
    ( 'How Democracies Die', 'A look at how other democracies around the world have failed.'
;
```

Note the use of the `TSVECTOR` data type.

The insert will leave the `document_tokens` as a null field. We will have to update the table to set the tokens. Also note that this ignores the "document_body" field. We will get to that in a little bit.

```
UPDATE indexed_docs d1
    SET document_tokens = to_tsvector(d1.document_title)
    FROM indexed_docs d2
;
```

Let's query for a pair of words.

```
SELECT doc_id, document_title
    FROM indexed_docs
    WHERE document_tokens @@ to_tsquery('die & democracy')
;
```

Having to update the table after it is changed is far from ideal. Let's replace this with a trigger so that the tokens are always updated. Also we will add a ranking and the `document_body`.

Run the file `hw38_10.sql`:

```
CREATE OR REPLACE function indexed_docs_ins_upd()
RETURNS trigger AS $$
DECLARE
    l_lang text;
BEGIN
    l_lang = 'english';
    NEW.document_tokens =
        setweight ( to_tsvector ( l_lang::regconfig, coalesce(NEW.document_title,'')), 'A' )
        setweight ( to_tsvector ( l_lang::regconfig, coalesce(NEW.document_body,'')), 'B' )
    ;
    RETURN NEW;
END
$$ LANGUAGE 'plpgsql';


DROP TRIGGER if exists  indexd_docs_trig_1 on indexed_docs;

CREATE TRIGGER indexd_docs_trig_1
    BEFORE insert or update ON indexed_docs
```

```
    FOR EACH ROW
    EXECUTE PROCEDURE indexed_docs_ins_upd()
;
```

and we will add some data to verify that the trigger works.

Run the file `hw38_11.sql`:

```
INSERT INTO indexed_docs ( document_title, document_body ) values
    ( 'Kleptopia: How Dirty Money Is Conquering the World', 'How the flow of dirty money is
    ( 'The End of Democracies', 'How democracies aroudn the world are failing.' )
;
```

A select on the indexed_documents will show that all 4 rows have the `document_tokens` field filled in.

Now add an index to the field so that searches will be fast.

Run the file `hw38_12.sql`:

```
CREATE INDEX if not exists indexed_docs_tsv_1 ON indexed_docs USING GIN (document_tokens);
```

FilesToRun: hw38_07.sql

FilesToRun: hw38_10.sql

FilesToRun: hw38_11.sql

FilesToRun: hw38_12.sql