Summary test 3

Wouter van den Brink

Information retrieval and full-text search

Search engines are systems that help you find relevant information among a vast amount of irrelevant information. An index of the collection is built beforehand. Then, given a search query, the index is used to match relevant documents. Finally, the search results are ranked according to their relevance. Any unit of information may be indexed. They all need their own index / query / match / rank system.

Indexing documents is done by tokenizing documents into words or terms. The terms are normalized, meaning that they're turned into lower case and a more generic spelling. Is, be, are and am would all become be, for example. Stop words like 'the' and 'and' are removed.

The standard index for information retrieval is called an inverted index. It stores, per term, its posting list, containing the list of documents in which the term occurs.

Few terms like 'a', 'an', 'and', 'the', etc. occur very frequently, while many terms, like spelling mistakes and names, occur very infrequently. They are both quite non-informative words. Words with medium frequency are the most relevant.

Calculating ranking

The relevance of a document is decided based on two factors.

- **Term frequency (tf)**
  The number of times a term occurs in a document.
- **Inverse document frequency (idf)**
  Usually $idf(t) = \log\left(\frac{N}{df}\right)$ with N the total number of documents and df the document frequency (number of documents that contain the term).
  The rank for a given document d and a search query q is then determined as follows:
  $Rank(d, q) = \sum_{t \in Query} tf(d, t) \times idf(t)$

As an example, the following documents are given:

Document	Content
D_1	John likes to watch movies. Mary likes movies, too.
D_2	John also likes to watch football games.
D_3	Something completely different.

Table 1 - A set of documents

Term	idf	Occurrence
john	$0.176 = \log\left(\frac{3}{2}\right)$	[<1, D_1>, <1, D_2>]
likes	$0.176 = \log\left(\frac{3}{2}\right)$	[<2, D_1>, <1, D_2>]
movies	$0.477 = \log\left(\frac{3}{1}\right)$	[<2, D_1>]
games	$0.477 = \log\left(\frac{3}{1}\right)$	[<1, D_2>]

Table 2 - The matching terms with their idf and occurrences

If someone now searches for 'john', both D_1 and D_2 result in a rank of 0.176 (1×0.176). However, a query for 'likes' results in D_1 having a higher (0.352) relevancy than D_2 (0.176). This is because 'likes' occurs more frequently in D_1.

Full-text support in PostgreSQL

PostgreSQL has full full-text search support. If you tell it to, the DBMS will index text columns just like a mature search engine would. A parser tokenizes documents (split a document into words), while a dictionary converts tokens into lexemes (generalize spelling, map variations and synonyms).

After processing, a string is suddenly of type `tsvector`; a sorted array of tokens. In theory this is also called a bag of words.

A search query may be converted to a `tsquery`. This type is also a sorted array of words, tokenized and converted into lexemes. A `tsquery` may be matched against a `tsvector` using the @@ operator:

```
SELECT
  'John likes to watch movies. Mary likes movies, too.'::tsvector
@@
  'john likes'::tsquery
```

However, using the casting operator (`::`) does not let PostgreSQL do its magic to make a query of vector tokenized and converted into lexemes. This requires the functions `to_tsvector` and `to_tsquery`.

```
SELECT
  to_tsvector('John likes to watch movies. Mary likes movies, too.')
@@
  to_tsquery('john likes')
```

Now, suppose we have a table called example, with textual attributes title and body.

Title	Body
About John and movies	John likes to watch movies. Mary likes movies,

	too.
Update on Johns hobbies	John also likes to watch football games.
Test post please ignore	Something completely different.

Table 3 - The example table

We can then perform the same example query as before, in PostgreSQL, with the following query:

```
SELECT
  title
FROM example
WHERE
  to_tsvector('english', title || ' ' || body)
  @@
  to_tsquery('john & likes')
ORDER BY updated_at DESC
LIMIT 10
```

This query finds the titles of the last 10 modified rows containing 'john' and 'likes' in either their title and / or their body.

The downside to a query like the above is that PostgreSQL re-indexes every single row every time the query is executed, resulting in a slow application. Imagine Google re-indexing everything they know every time you perform a search… This is solved by adding a full-text index.

```
CREATE INDEX example_index ON example
USING GIN( -- Generalized inverted index
  to_tsvector('english', title || ' '|| body)
)
```

In this case, GIN (generalized inverted index) is used. The column bust me of `tsvector` type. The alternative is GiST: a generalized search tree.
There are many ways to glue separate strings together in a document, to create one search index. The simplest way is gluing them with the || operator as shown in the previous queries. However, if strings are not in the same row, like in a one-to-many relation, the function `string_agg` is needed. Just like operators as MIN, MAX and AVG, the `string_agg` function may be used in a GROUP BY statement.

Genre	Movie
Crime	The Shawshank Redemption
Crime	The Godfather
Action	The Dark Knight
Drama	12 Angry Men
Drama	Schindler's List

Table 4 - An example dataabse of movies and their genres

```
SELECT
  genre,
  to_tsvector(string_agg(movie))
FROM movies
GROUP BY genre
```

Genre	Tsvector
Crime	shawshank redemption godfather
Action	dark knight
Drama	12 angry men schindler list

Table 5 - Example result of the above query. Untested code.

Continuing on the example of a movie database, one might have a database with more information, like a short description and a review. In this case, the titles and descriptions of a movie should have more importance over a review when finding a movie. In that case, the `setweight` function might be used.

```
SELECT
  genre,
  setweight(to_tsvector(string_agg(movie)), 'A'),
  setweight(to_tsvector(string_agg(description)), 'A'),
  setweight(to_tsvector(string_agg(review)), 'A')
FROM movies
GROUP BY genre;
```

Here, the title and movie have a higher weight (A) than a review (D). Weights are expressed as 'A' to 'D', with 'A' having the highest and 'D' having the lowest importance.
A better approach for preprocessing large amounts of searchable data is by creating a custom column containing a prepared `tsvector`.

```
ALTER TABLE example ADD COLUMN vector_column tsvector;

UPDATE example SET vector_column = to_tsvector('english',
  coalesce(title, '') || coalesce(body, '')
)

CREATE INDEX example_index ON example USING GIN(vector_column);
```

Note the use of coalesce. This function will return its second argument if the first argument is NULL, preventing PostgreSQL from complaining whenever we're trying to concatenate an empty column with another column.

Such a column would need to be automatically updated whenever a row is updated, otherwise new inserts and updates would have a NULL in the column, and will be absent from the index.

```
CREATE TRIGGER example_vector_update
BEFORE INSERT OR UPDATE ON example
FOR EACH ROW EXECUTE PROCEDURE
tsvector_update_trigger(vector_column, 'pg_catalog.english', title, body);
```

As mentioned before, a search engine requires ranking in order to display relevant results. Luckily, PostgreSQL is able to automagically rank results from a search query. The syntax is as follows:

```
SELECT title, ts_rank(tt, query) AS r
FROM example, to_tsquery('john likes') query
             -- Trick to construct the query once
WHERE tt @@ query
ORDER BY r DESC
LIMIT 10
```

Finally, one might want to indicate where the search query was found in a document. PostgreSQL solves this problem with the `ts_headline` function.

```
SELECT ts_headline(body, query), r)
FROM (
  SELECT title, ts_rank(tt, query) AS r
  FROM example, to_tsquery('john likes') query
  WHERE tt @@ query
  ORDER BY r DESC
  LIMIT 10
) AS foo
```

This results in the relevant terms being surrounded with HTML <b> tags.

Language models

A language model assigns a probability to a piece of unseen text, based on some training data. Given an English model, then for example $P([a, bit, of, text]) > P([een, stukje, tekst])$ and $P([a, bit, of, text]) > P([aw, pit, tov, tags])$.

Those probabilities can be used to rank documents in a search engine. Given a document in our index, if we choose at random a certain amount of words in a document, what is the probability that those words match the search query? Whichever document has the highest document is probably also the most relevant document.

Given a query as a number of search terms $T_1 \ldots, T_n$ and documents $D_1, \ldots, D_n$, we then sort on $P(D_i | T_1, \ldots, T_n)$.

This can be derived using Bayes rule:

$$P(D|T_1, \ldots, T_n) = \frac{P(T_1, \ldots, T_n|D) \times P(D)}{P(T_1, \ldots, T_n)}$$

We're sorting on documents, so $P(T_1, \ldots, T_n)$ doesn't really matter (it does not depend on D). $P(D)$ can be described as the relevancy of a document without a search query. We assume that this is the same for any document, so that also doesn't matter that much.

We end up with $P(T_1, \ldots, T_n | D)$, which can be roughly described as "the probability that a user chooses query terms $T_1, \ldots, T_n$ when trying to find D". The estimation of this probability is done using the bag of words model. If we put all the words of a document D in a bag, and we randomly pick n words from the bag, what is the probability that you picked $T_1, \ldots, T_n$? Let's find out in the same set of documents as before:

Document	Content
D_1	John likes to watch movies. Mary likes movies, too.
D_2	John also likes to watch football games.

Table 6 - Same dataset. Document 3 is ignored as we only discuss ranking.

We know:

- $P(john|D_1 ¿= \frac{1}{9}; P(john|D_2) = \frac{1}{7}$
- $P(movies|D_1) = \frac{2}{9}; P(movies|D_2) = \frac{0}{7}$

If we search for 'john', then D_2 is more relevant than D_1. Searching for 'movies', D_1 is more relevant than D_2. If we search for 'john movies', then D_1 is more relevant than D_2. This assumes independence of words.

$P(john, movies|D ¿= P(john|D) * P(movies \vee D)$

$¿ \frac{2}{81} for\ D_1$

$¿ 0\ for\ D_2$

Database indices

An index is a (hidden) derived data structure used for speeding up certain queries. A UNIQUE index, for example, speeds up SELECT queries significantly when using the primary key. A downside to indices is that they may slow up updates, inserts and deletes. Indices are transparent to the application developer; they're only there to improve performance at the DBMS level. The DBMS also decides whether using an index is useful or not.

Database views

A view is a certain query, resulting in a dataset, which may then be used as a table in other queries. The query is known to the DBMS beforehand, improving the speed of other queries using the data.

While a normal view only approaches creating a new table, a materialized query really results in a new table. The table can be refreshed to reflect the current state of the database, and a database administrator may add indices, views and triggers to the table as would be possible with any other normal table.

Views may be used to adhere to a certain external schema, as in the ANSI/SPARC architecture. They might be optimized for the query/update load of specific applications.

Database transactions

Transactions are a set of queries that are meant to be part of a sequence. Transactions are a kind of protection mechanism. If anything fails during a transaction (the connection is lost, a constraint was violated, or somebody cancelled the transaction …) any executed queries in the transactions are cancelled.

A transaction is started in PostgreSQL as follows:

```
BEGIN [ TRANSACTION | WORK ] [ mode ]

where mode is

ISOLATION LEVEL { SERIALIZABLE |
                  REPEATABLE READ |
```

```
                  READ COMMITTED |
                  READ UNCOMMITTED }
               READ WRITE | READ ONLY

[ Statements ]

COMMIT | END TRANSACTION | ROLLBACK
```

Database constraints

A constraint places a certain requirement on data in a database. For example, a grocery store might want to add a constraint, where a discounted price should always be lower than the original price. Examples of constraints include:

- REFERENCES enforces that the referenced row exists;
- NOT NULL enforces that a value is never NULL;
- CHECK (x) enforces that x is always true;

Constraints are explicitly defined with CONSTRAINT name *constraint*. Constraints are checked at the end of every transaction. If a constraint is violated, a rollback is performed and the user is informed of the error.

Constraints exist to ensure that the many users and applications using a database never corrupt the data they're sharing.

Transactions and Isolation Levels

Simultaneous operations on a database might be described in abstract form, using $r_i(X)$, $w_i(X)$ and $c_i(X)$. $r_i(X)$ indicates that some user of the database i is reading from some value X. $c_i(X)$ means that the user i is committing a transaction.