

Tasks

1. Create in your schema a score view that gives each row in the louis_v004.crawl table a score between 0.0 and 1.0 depending on the length (compared to the distribution) of the louis_v004.crawl.html_content column.

Answer:

```
CREATE OR REPLACE VIEW "sing1643@algonquinlive.com".score
AS SELECT crawl.id,
        round(cume_dist() OVER (ORDER BY (length(crawl.html_content))):numeric, 2) AS score,
        crawl.url,
        crawl.title,
        crawl.lang,
        crawl.html_content,
        crawl.last_crawled,
        crawl.last_updated,
        crawl.last_updated_date
FROM louis_v004.crawl;
```

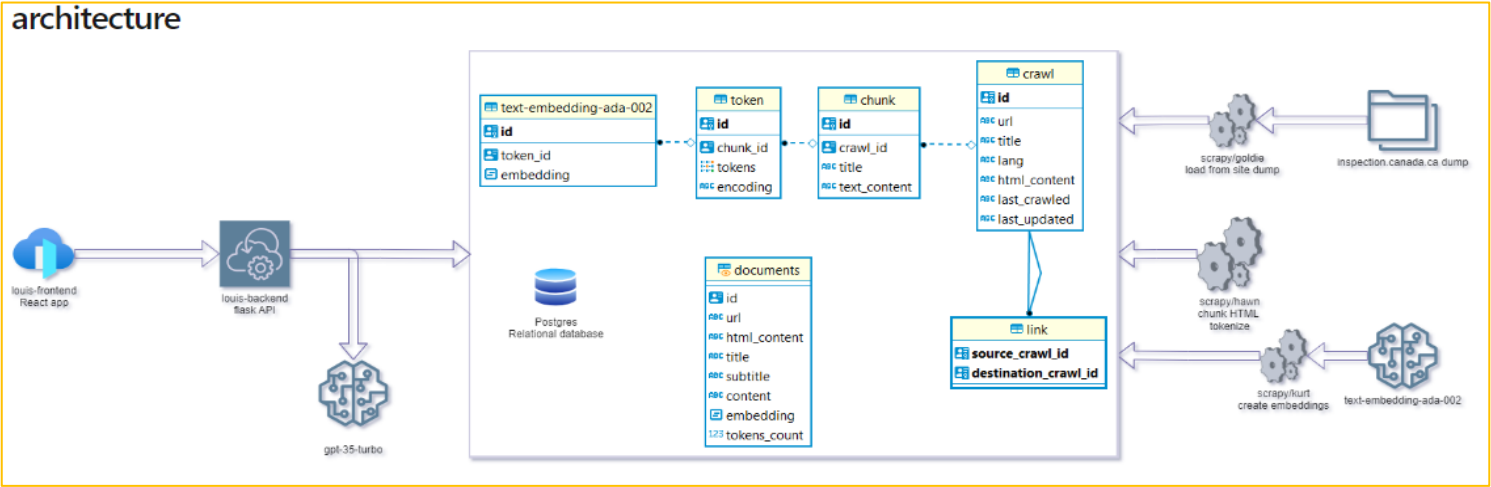
2. Write a search('keyword') function that returns the ten documents with the highest score containing the keyword.

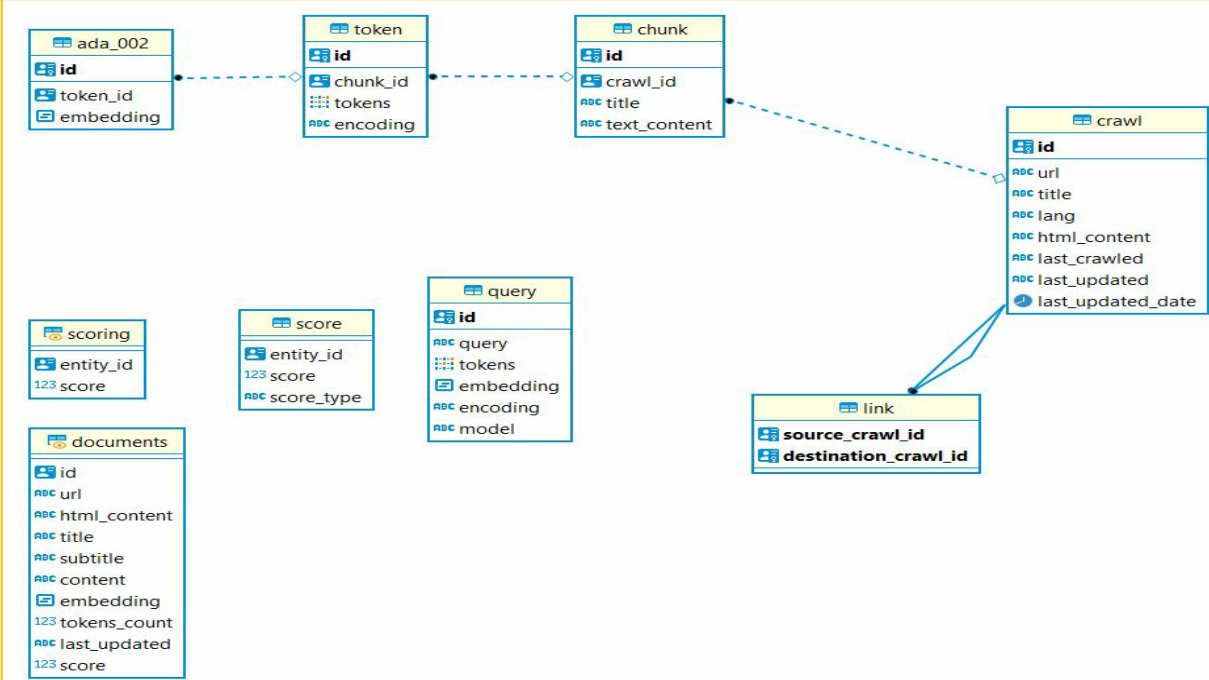
Answer:

```
CREATE OR REPLACE FUNCTION "sing1643@algonquinlive.com".search(_keyword text)
RETURNS TABLE(id uuid, html_content text, score numeric) AS $$
BEGIN
    RETURN QUERY
    SELECT SCORE.id,
            SCORE.html_content,
            SCORE.score
    FROM "sing1643@algonquinlive.com".score AS SCORE
    WHERE SCORE.html_content ILIKE '%' || _keyword || '%'
    ORDER BY SCORE.score DESC
    LIMIT 10;
END;
$$ LANGUAGE plpgsql;
```

3. What do the tables contain in the louis_v004 schema? Explain the relational structure and function of each table.

Answer:





With inference from the above architecture and provided schema, there are **7 tables** in the louis_v004 schema viz: “score, query, token, ada_002, crawl, link, chunk” and **2 views** “scoring and documents.”

Now, these tables convey the following information:

a.) **ada_002:**

This table appears to be associated with the results of a ‘Natural Language Processing (NLP)’ model as it is observed that it contains different vector weights being associated in the embedding column values. The token_id column links each entry to a specific token in the **token** table. Also, here ‘ada_002’ model is being utilised.

Number of Rows: ‘20571’ number of rows.

Relationships: ada_002(id)->Token(id) : One to Many with Source Optional.

Constraints: Primary Key= id, Foreign Key= token_id

DDL employed:

```
SQL Preview:
-- louis_v004.ada_002 definition
-- Drop table
-- DROP TABLE louis_v004.ada_002;

CREATE TABLE louis_v004.ada_002 (
  id uuid NOT NULL DEFAULT uuid_generate_v4(),
  token_id uuid NULL,
  embedding public.vector NULL,
  CONSTRAINT ada_002_pkey PRIMARY KEY (id)
);
CREATE INDEX ada_002_embedding_idx ON louis_v004.ada_002 USING ivfflat (embedding vector_cosine_ops) WITH (lists='83');

-- louis_v004.ada_002 foreign keys
ALTER TABLE louis_v004.ada_002 ADD CONSTRAINT ada_002_token_uuid_fkey FOREIGN KEY (token_id) REFERENCES louis_v004."token"(id) ON DELETE CASCADE;
```

b.) **token:**

This table contains tokens that are extracted from chunks of text. The chunk_id column provides a connection back to the chunk of text from which the token was derived, represented in the chunk table.

Number of Rows: ‘20572 rows’

Relationships: i.) Token(id)->ada_002(token_id) : Many to one with source optional

ii.) Token(chunk_id)-> chunk(id) : One to Many with source optional

Constraints: Primary Key= id, Foreign Key= chunk_id

DDL employed:

```
SQL Preview:
-- louis_v004."token" definition
-- Drop table
-- DROP TABLE louis_v004."token";

CREATE TABLE louis_v004."token" (
  id uuid NOT NULL DEFAULT uuid_generate_v4(),
  chunk_id uuid NULL,
  tokens_int4 NULL,
  "encoding" louis_v004."encoding" NULL,
  CONSTRAINT token_pkey PRIMARY KEY (id),
  CONSTRAINT token_tokens_key UNIQUE (tokens)
);

-- louis_v004."token" foreign keys
ALTER TABLE louis_v004."token" ADD CONSTRAINT token_chunk_uuid_fkey FOREIGN KEY (chunk_id) REFERENCES louis_v004.chunk(id) ON DELETE CASCADE;
```

c.) chunk:

The chunk table holds chunks of text that were derived from a crawled webpage. The crawl_id links each chunk back to a specific webpage crawl, represented in the crawl table.

Number of Rows: ‘20575 rows’

Constraints: Primary Key= id, Foreign Key= chunk_id

- Relationships: I.) chunk(id)->Token(chunk_id) : Many to one with source optional
ii.) chunk(crawl_id) -> crawl(id) : one to many with source optional

DDL employed:

```
SQL Preview:
-- louis_v004.chunk definition
-- Drop table
-- DROP TABLE louis_v004.chunk;

CREATE TABLE louis_v004.chunk (
  id uuid NOT NULL DEFAULT uuid_generate_v4(),
  crawl_id uuid NULL,
  title text NULL,
  text_content text NULL,
  CONSTRAINT chunk_pkey PRIMARY KEY (id),
  CONSTRAINT chunk_text_content_key UNIQUE (text_content)
);

-- louis_v004.chunk foreign keys
ALTER TABLE louis_v004.chunk ADD CONSTRAINT chunk_crawl_uuid_fkey FOREIGN KEY (crawl_id) REFERENCES louis_v004.crawl(id) ON DELETE CASCADE;
```

d.) crawl:

The crawl table contains information regarding each webpage crawl, including metadata (like ‘title’, ‘url’, ‘last_updated’ & ‘last_crawled’ and the html content that was crawled.

Number of Rows: 13294 rows

Constraints: Primary Key= id

- Relationships: i.) crawl(id) -> chunk(crawl_id) : Many to one with source optional
ii.) crawl(id)-> link(source_crawl_id, destination_crawl_id) : one to one with ‘NOT NULL’ FOREIGN KEY

```
SQL Preview:
-- louis_v004.crawl definition
-- Drop table
-- DROP TABLE louis_v004.crawl;

CREATE TABLE louis_v004.crawl (
  id uuid NOT NULL DEFAULT uuid_generate_v4(),
  url text NULL,
  title text NULL,
  lang_bpchar(2) NULL,
  html_content text NULL,
  last_crawled text NULL,
  last_updated text NULL,
  last_updated_date date NULL,
  CONSTRAINT crawl_pkey PRIMARY KEY (id),
  CONSTRAINT crawl_url_last_updated_key UNIQUE (url, last_updated)
);
```

e.) link:

The link table maintains the relationships between different webpage crawls. This allows for the representation of the structure of the webpages that are being crawled. ‘source_crawl_id’ and ‘destination_crawl_id’ create a connection back to the crawl table, representing the origin and destination of the link.

Number of Rows: 0 rows

Constraints: Composite Primary Key= ‘source_crawl_id, destination_crawl_id’

```
SQL Preview:
-- louis_v004.link definition

-- Drop table
-- DROP TABLE louis_v004.link;

CREATE TABLE louis_v004.link (
  source_crawl_id uuid NOT NULL,
  destination_crawl_id uuid NOT NULL,
  CONSTRAINT link_pkey PRIMARY KEY (source_crawl_id, destination_crawl_id)
);

-- louis_v004.link foreign keys

ALTER TABLE louis_v004.link ADD CONSTRAINT link_destination_crawl_id_fkey FOREIGN KEY (destination_crawl_id) REFERENCES louis_v004.crawl(id) ON DELETE CASCADE;
ALTER TABLE louis_v004.link ADD CONSTRAINT link_source_crawl_id_fkey FOREIGN KEY (source_crawl_id) REFERENCES louis_v004.crawl(id) ON DELETE CASCADE;
```

f.) query:

This table holds information about queries that have been run, likely against an NLP model. As of the given information, it doesn't appear to have any direct relationships with the other tables.

Number of rows: 0

Constraints: Primary Key= id

```
SQL Preview:
-- louis_v004.query definition

-- Drop table
-- DROP TABLE louis_v004.query;

CREATE TABLE louis_v004.query (
  id uuid NOT NULL DEFAULT uuid_generate_v4(),
  query text NULL,
  tokens_int4 NULL,
  embedding public.vector NULL,
  "encoding" text NULL DEFAULT 'ada_002::text',
  model text NULL DEFAULT 'ada_002::text',
  CONSTRAINT query_pkey PRIMARY KEY (id),
  CONSTRAINT query_tokens_key UNIQUE (tokens)
);
```

g.) score:

This table stores scoring data that has been fetched from another table ('score'). It doesn't seem to have any direct relationships with the other tables, based on the provided information.

Number of rows: 39,831 rows

Constraints: Primary Key= entity_id

```
SQL Preview:
-- louis_v004.score definition

-- Drop table
-- DROP TABLE louis_v004.score;

CREATE TABLE louis_v004.score (
  entity_id uuid NULL,
  score float8 NULL,
  "score_type" louis_v004."score_type" NULL
);
```

The relationships between the tables (ada_002, token, chunk, crawl, and link) illustrate a process of webpage crawling, content extraction, and further breaking down into tokens. The query and score tables appear to serve separate functions related to querying and scoring that might not be directly linked to the web scraping and processing pipeline.

Q. Which distribution take content length values?

Answer2.

The content length values were obtained from the database server as a CSV file. The file was then imported into Jupyter Notebook within the **Anaconda IDE**. Using the **Pandas library**, the CSV file was converted into a DataFrame for further analysis. A basic statistical description of the DataFrame was obtained using the **df.describe()** function, allowing us to gain insights into the central tendency and spread of the content length data. To visualize the distribution of content length values, a **histogram plot** using the **sns.histplot()** function from the Seaborn library was create.

EMPLOYED Python code:

```
import pandas as pd
import seaborn as sns
df=pd.read_csv('score_latest.csv')
```

```
df.describe()
```

content_length	
count	13294.000000
mean	0.500067
std	0.288689
min	0.000000
25%	0.250000
50%	0.500000
75%	0.750000
max	1.000000

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set(style='darkgrid')

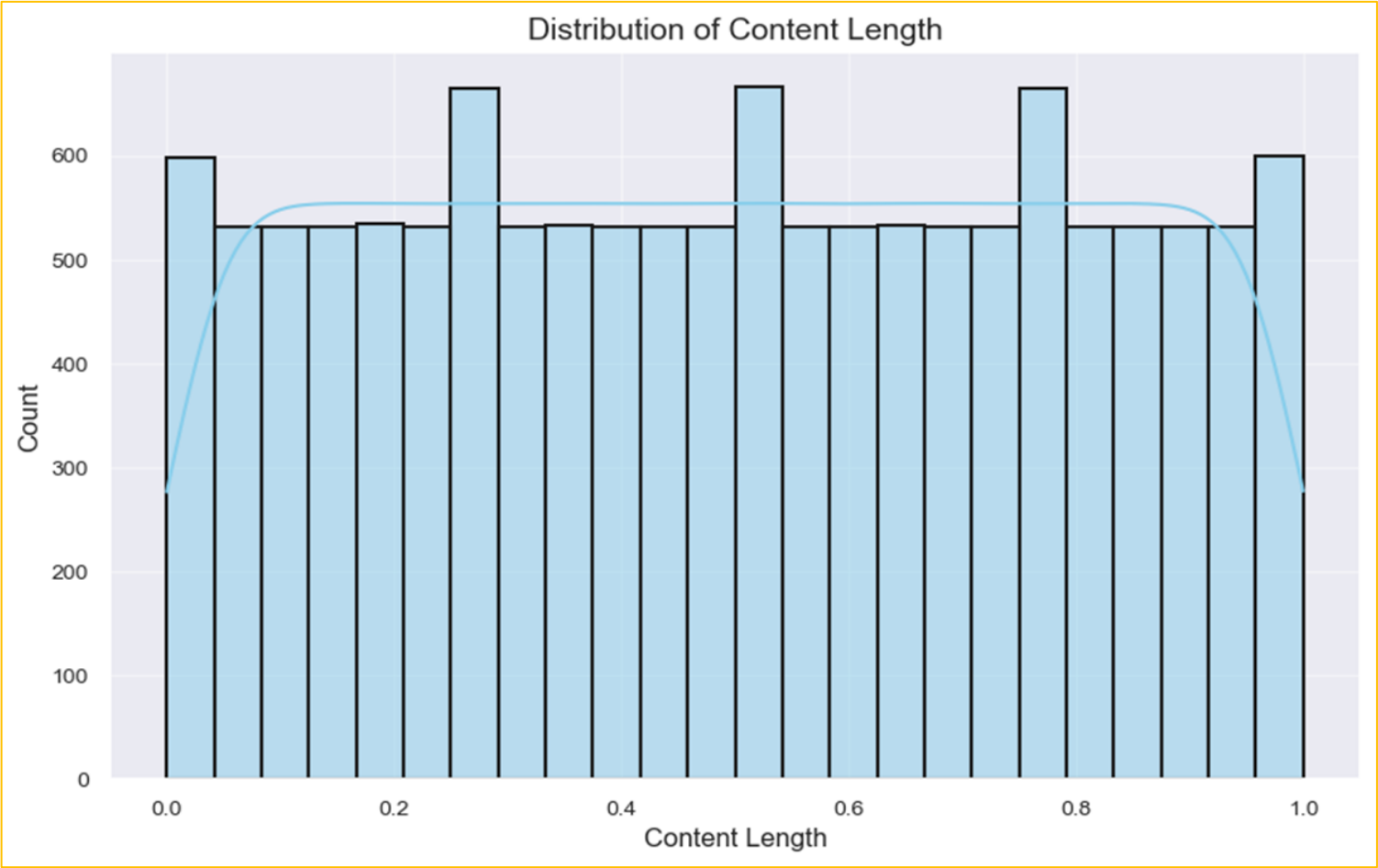
plt.figure(figsize=(10, 6))
sns.histplot(data=df, x='content_length', kde=True, color='skyblue', edgecolor='k', linewidth=1.5)

plt.xlabel('Content Length', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.title('Distribution of Content Length', fontsize=14)

plt.xticks(fontsize=10)
plt.yticks(fontsize=10)

plt.grid(True, alpha=0.5)

plt.show()
```



INFERENCE DRAWN:

The distribution for the content_length was found to be almost follow a ‘Normal distribution’.

To further explain this, one more code was ran on jupyter notebook which shows the normally distributes density curve:


```
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm

sns.set(style='darkgrid')

plt.figure(figsize=(10, 6))

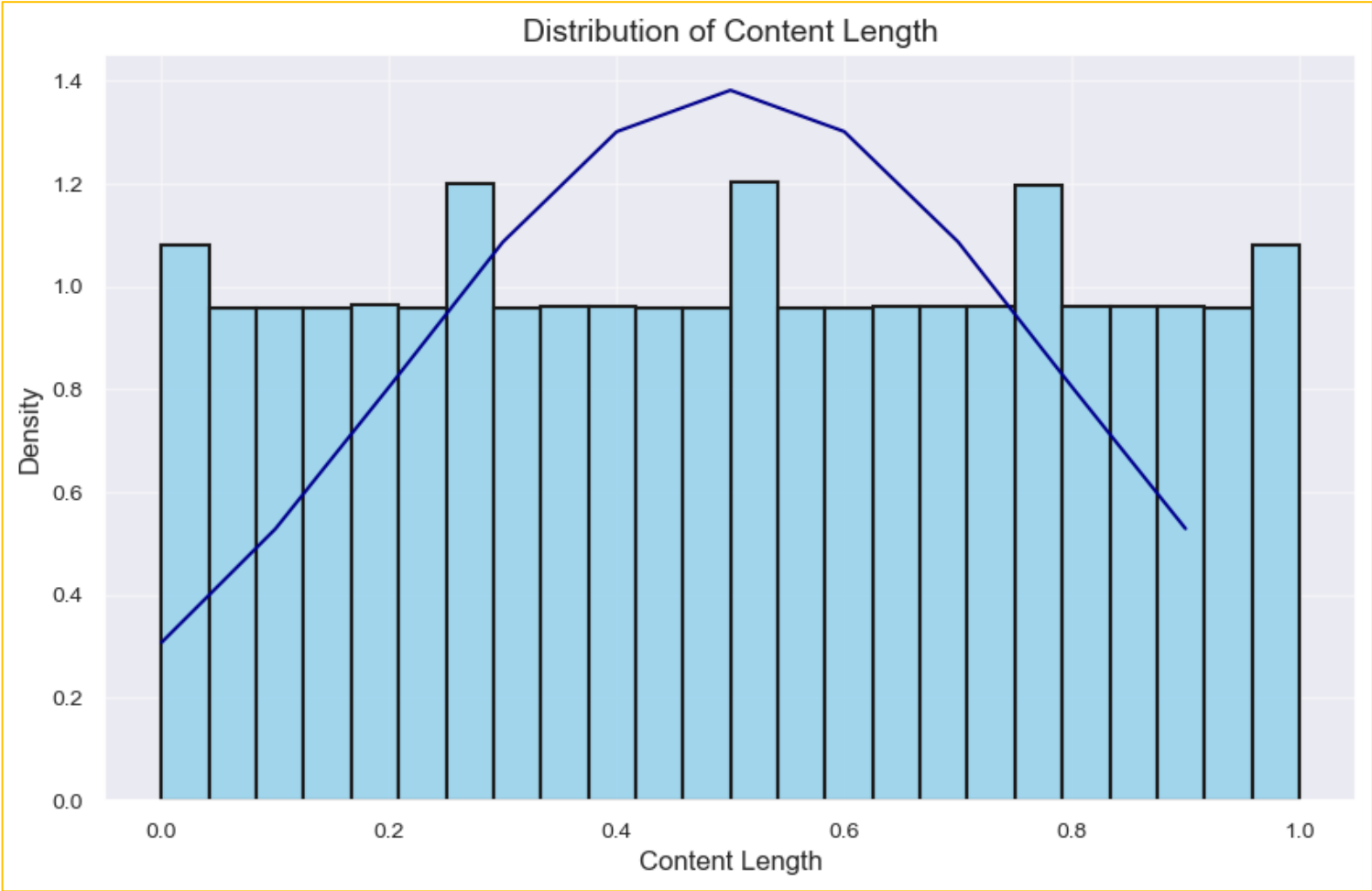
sns.histplot(data=df, x='content_length', kde=False, color='skyblue', edgecolor='k', linewidth=1.5, stat='density')
x_values = np.arange(min(df['content_length']), max(df['content_length']), 0.1)
normal_distribution = norm.pdf(x_values, np.mean(df['content_length']), np.std(df['content_length']))
plt.plot(x_values, normal_distribution, color='darkblue')

plt.xlabel('Content Length', fontsize=12)
plt.ylabel('Density', fontsize=12)
plt.title('Distribution of Content Length', fontsize=14)

plt.xticks(fontsize=10)
plt.yticks(fontsize=10)

plt.grid(True, alpha=0.5)

plt.show()
```



Q.Explain calculation based on specific distribution of lengths values from html_content script

Answer 3:

The calculation was done in following way:

```
SQL Preview:
-- "sing1643@algonquinlive.com".score source

CREATE OR REPLACE VIEW "sing1643@algonquinlive.com".score
AS SELECT crawl.id,
    round(cume_dist() OVER (ORDER BY (length(crawl.html_content))))::numeric, 2) AS score,
    crawl.url,
    crawl.title,
    crawl.lang,
    crawl.html_content,
    crawl.last_crawled,
    crawl.last_updated,
    crawl.last_updated_date
FROM louis_v004.crawl;
```

CREATE OR REPLACE VIEW "sing1643@algonquinlive.com".score AS- This line indicates that a new view with the name "score" will be created in the schema "sing1643@algonquinlive.com". The new view being specified here will take the place of any views with the same name that already exist in that schema.

'length(crawl.html_content)': Each row in the crawl table's html_content column has its length determined by this function.

‘cume_dist() OVER (ORDER BY (length(crawl.html_content)))’ : The cumulative distribution of the lengths of the html_content field is calculated by this window function. In simpler terms, it calculates the percentage of rows for each row whose html_content length is less than or equal to the html_content length of the current row.

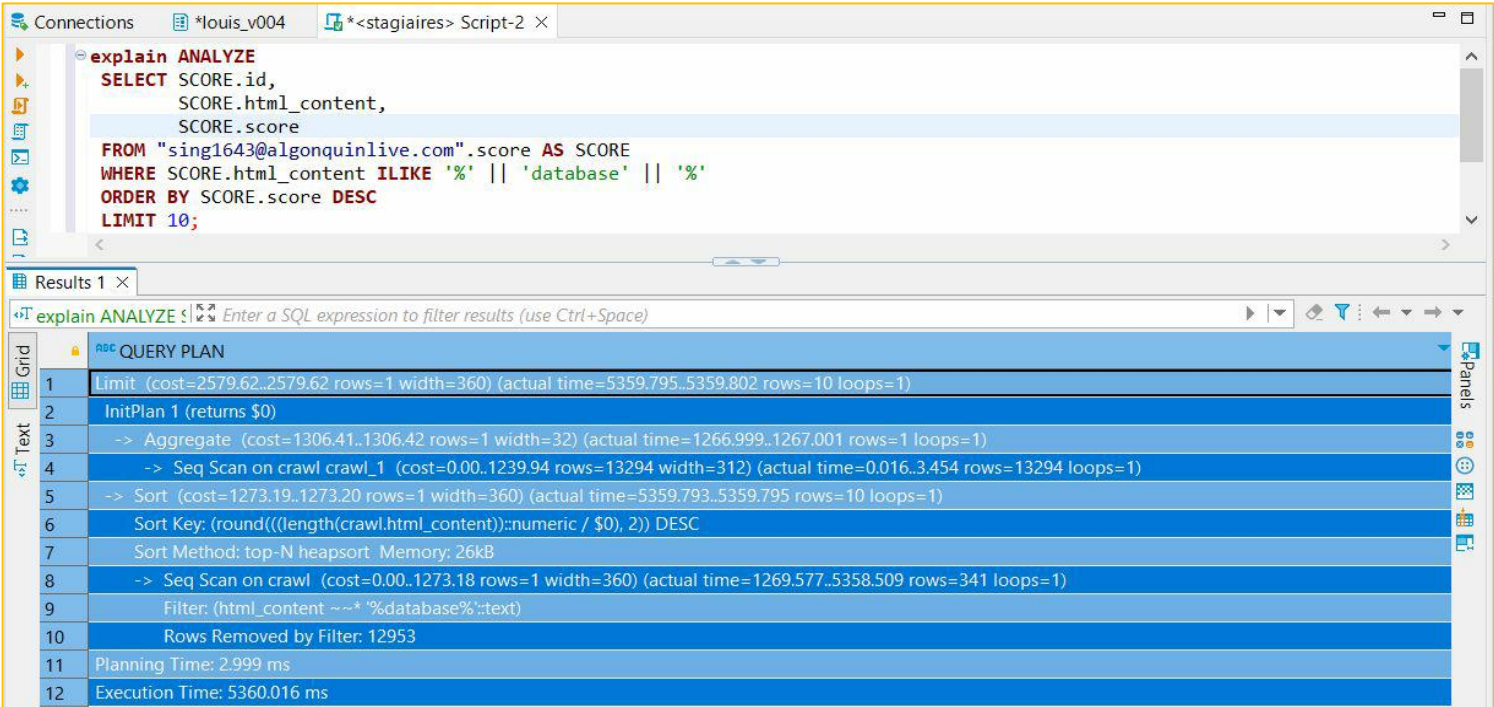
‘round(.....numeric, 2) AS score’ : The cume_dist() calculation's output is rounded to two decimal places by this function. After being converted to a numeric data format, the result is then assigned the name ‘score’.

‘crawl.url, crawl.title, crawl.lang, crawl.html_content, crawl.last_crawled, crawl.last_updated, crawl.last_updated_date’: These are the additional columns that have been chosen for the view from the crawl table. Similar to how they do in the original crawl table, these columns will also appear in the view.

So, to put it simply, this view gives users a **louis_v004.crawl** table version with a score column added. With 1 being the longest and 0 being the shortest, the values in this score column indicate where the **html_content** length of each row lies in the distribution of all lengths.

Q. Explain and discuss the performance of your search function

Answer 4:



The screenshot shows a PostgreSQL query editor with a query and its execution plan. The query is as follows:

```
explain ANALYZE
SELECT SCORE.id,
       SCORE.html_content,
       SCORE.score
FROM "sing1643@algonquinlive.com".score AS SCORE
WHERE SCORE.html_content ILIKE '%' || 'database' || '%'
ORDER BY SCORE.score DESC
LIMIT 10;
```

The execution plan is shown in a table with 12 rows:

Step	Operation	Cost	Rows	Width	Actual Time	Actual Rows	Loops
1	Limit	(cost=2579.62..2579.62 rows=1 width=360)	1	360	(actual time=5359.795..5359.802 rows=10 loops=1)	10	1
2	InitPlan 1 (returns \$0)						
3	-> Aggregate	(cost=1306.41..1306.42 rows=1 width=32)	1	32	(actual time=1266.999..1267.001 rows=1 loops=1)	1	1
4	-> Seq Scan on crawl crawl_1	(cost=0.00..1239.94 rows=13294 width=312)	13294	312	(actual time=0.016..3.454 rows=13294 loops=1)	13294	1
5	-> Sort	(cost=1273.19..1273.20 rows=1 width=360)	1	360	(actual time=5359.793..5359.795 rows=10 loops=1)	10	1
6	Sort Key: (round(((length(crawl.html_content)):numeric / \$0), 2)) DESC						
7	Sort Method: top-N heapsort	Memory: 26kB					
8	-> Seq Scan on crawl	(cost=0.00..1273.18 rows=1 width=360)	1	360	(actual time=1269.577..5358.509 rows=341 loops=1)	341	1
9	Filter: (html_content ~~~ '%database%':text)						
10	Rows Removed by Filter: 12953						
11	Planning Time: 2.999 ms						
12	Execution Time: 5360.016 ms						

EXPLANATION:

Firstly, **Limit (cost=2579.62..2579.62 rows=1 width=360) (actual time=5359.795..5359.802 rows=10 loops=1)**: According to this, the limit operation—which is equivalent to the SQL query LIMIT 10—is the outermost operation. The cost value represents an estimation of the amount of computation needed to complete this query step. Following this, **The (actual time=5359.795..5359.802 rows=10 loops=1)** section states that this step took approximately 5359 milliseconds to complete and returned 10 rows.

Secondly, **InitPlan 1 (returns \$0)**: This is a subquery or operation that is performed before the main query.

Third, **Aggregate (cost=1306.41..1306.42 rows=1 width=32) (actual time=1266.999..1267.001 rows=1 loops=1)**: The Aggregate operation determines the crawl table's maximum html_content length.

Next, **Seq Scan on crawl crawl_1 (cost=0.00..1239.94 rows=13294 width=312) (actual time=0.016..3.454 rows=13294 loops=1)**: This is a sequential scan of the crawl table. The query planner estimated, that it would need to scan 13294 rows, which turned out to be accurate (rows=13294).

Further, **Sort (cost=1273.19..1273.20 rows=1 width=360) (actual time=5359.793..5359.795 rows=10 loops=1)**: The Sort operation organises the results in descending order according to score. In fact, this procedure yielded 10 rows (rows=10), which is in accordance with the LIMIT clause.

Moving ahead, **Seq Scan on crawl (cost=0.00..1273.18 rows=1 width=360) (actual time=1269.577..5358.509 rows=341 loops=1)**: Another sequential scan on the crawl table, this time with a filter applied (i.e., looking for rows where html_content contains the word 'database'). This scan was estimated to return 1 row (rows=1), but it actually returned 341 rows (rows=341).

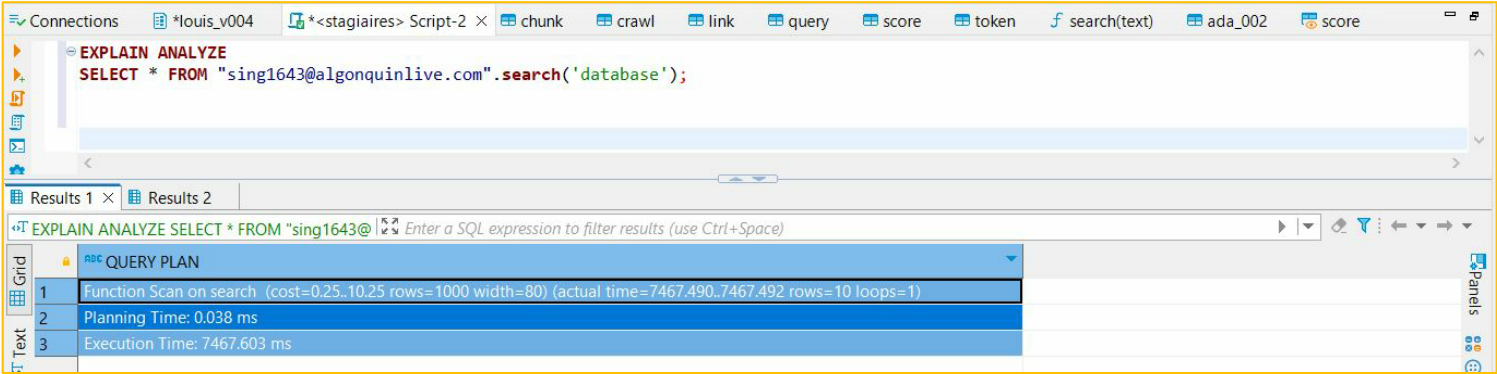
Lastly,

‘Rows Removed by Filter: 12953’ : Out of the total rows scanned, this line indicates that 12953 did not meet the filter requirements and were therefore excluded from the result set.

Planning Time: 2.999 ms: This is the length of time PostgreSQL's query planner took to plan the query's execution.

Execution time for the query was **5360.016 milliseconds**, which includes both planning and acting time.

ALSO,



EXPLANATION:

Function Scan on search: This only indicates that PostgreSQL is going through the search function's findings.

(cost=0.25..10.25 rows=1000 width=80): This is PostgreSQL's estimation about how "expensive" the operation is, in arbitrary cost units.

(actual time=7467.490..7467.492 rows=10 loops=1): These are the performance metrics that were actually recorded when the query was run. In actually, the function started returning rows in 7467.490 milliseconds and finished at 7467.492 milliseconds. Ten rows were returned. The loops=1 clause denotes a single execution of the procedure.

Planning Time: This is the amount of time the query plan was developed by PostgreSQL's query planner. It was 0.038 milliseconds in this instance.

Execution Time: This shows how long the query actually took to run. It was 7467.603 milliseconds in this instance.

-----END-----