thoughts

Choosing a database

I took my time in choosing the database, should it be sql or no sql?
the data is pretty structured and has a straight forward schema which hints to go for sql.on the other hand, sql databases don't scale well. in order to understand if I would need to scale or not I'm going to take some assumptions about the amount of data which is going to be in this database.

I researched the web using chatgpts aid to understand the traffic of NBC, one of the most widely used news website. Lets say there are 300 articles per day, in one year it amount to 100,000 and in 10 years to a million. NBC started its journey around the 2000's so lets assume there are roughly 3 million articles. About the users, lets say around 10,000 users are writing articles and each article gets around 100 comments, so 300 million comments.

	Size	Estimated amound	Total size
Article name	20 characters	3,000,000	60,000,000
Article content	5,000 words = 50,000 characters	3,000,000	150,000,000,000
User name	20 characters	10,000	200,000
Comment	50 words = 500 characters	300,000,000	150,000,000,000

Total amount of data in bytes is around 300 billion bytes which are 300gb. From what I understand, tables that are bigger than 1tb can encounter performance issues, so it is acceptable to put this data in sql tables even for a major news website like NBC. For the sake of this task I will use sqlite, but for a large scale and high traffic application I would use PostrageSQL or MySQL

Article schema

Another thing I was taking into consideration is the schema of the article. More specifically, should I keep the content inside the table? Or keep the actual content in some object storage like s3 and keep only a url to it? A quick search in the web showed me what is the best way to keep large files in a sql database

VARCHAR(X)

> **Max Length:** variable, up to 65,535 bytes (64KB)
> **Case:** user name, email, country, subject, password

TEXT

> **Max Length:** 65,535 bytes (64KB)
> **Case:** messages, emails, comments, formatted text, html, code, images, links

MEDIUMTEXT

> **Max Length:** 16,777,215 bytes (16MB)
> **Case:** large json bodies, short to medium length books, csv strings

LONGTEXT

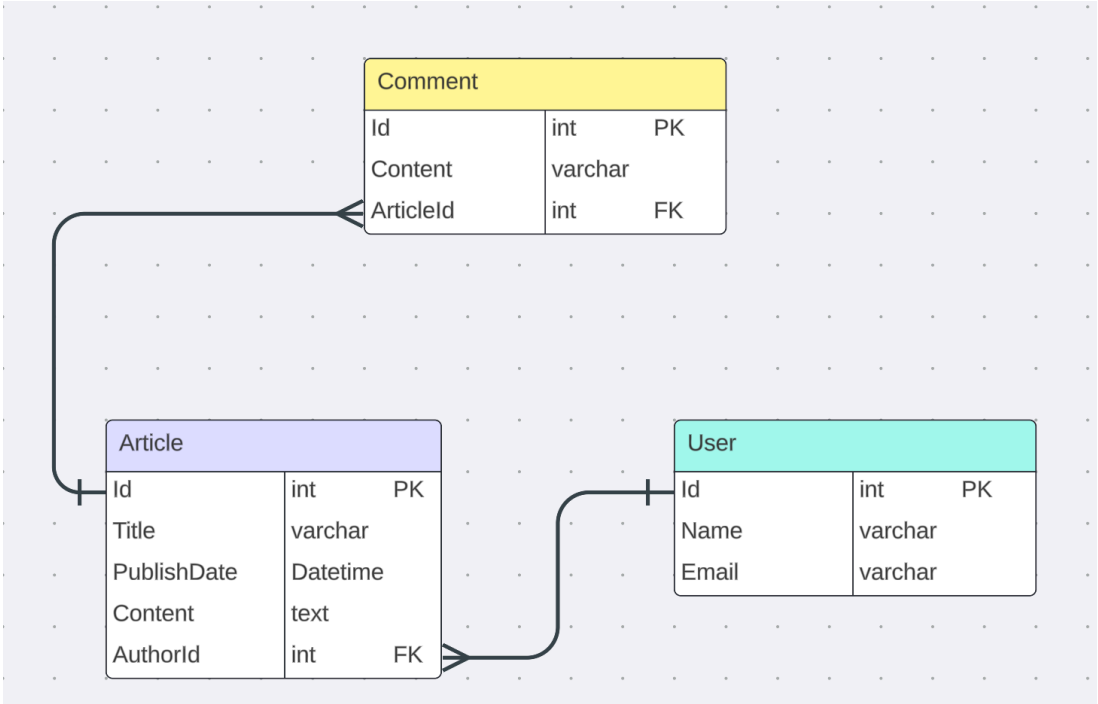
> **Max Length:** 4,294,967,29 bytes (4GB)
> **Case:** textbooks, programs, years of logs files, harry potter and the goblet of fire, scientific research logging

For our usage, 5,000 words are translated to 50,000 characters which are 50kb. So It seems reasonable to use TEXT or MEDIUMTEXT. These types are stored off the table while keeping some pointer to it, in contrast to varchar which keeps the data inside the table.

I think for our usage this solution suits us perfectly well, but I am making an assumption that the article does not contain any videos, images etc…

Of course these types of files are much larger and should be kept in an object storage.

Schema



API hierarchy

Because we have this kind of chained relation, a user has many articles and an article has many comments, we could design the hierarchy of the api in a manner that shows the relation in the endpoints, like so:

http method	Endpoint	Meaning
Post	/v1/users/	Create a new user
Get	/v1/users/	Read all users
Get	/v1/users/{user_id}	Read a user with user_id
Post	/v1/users/{user_id}/articles/	Create a new article which is written by user_id
Get	/v1/users/{user_id}/articles/	Read all articles written by user_id
Get	/v1/users/{user_id}/articles/{article_id}	Read a specific article
Post	/v1/users/{user_id}/articles/{article_id}/comments/	Create a new comment for article_id
Get	/v1/users/{user_id}/articles/{article_id}/comments/	Read all comments of article_id
Get	/v1/users/{user_id}/articles/{article_id}/comments/{comment_id}	Read a specific comment

On the other hand, we might want to have a root endpoint for articles and comments which are unrelated to a user, and the relation to a user can be sent through query params or inside the request body. This approach might be more flexible and focus more on the objects themselves instead of the relations between them.

http method	Endpoint	Meaning
Post	/v1/users/	Create a new user
Get	/v1/users/	Read all users
Get	/v1/users/{user_id}	Read a user with user_id
Post	/v1/articles/	Create a new article which is written by user_id (user_id is sent in the body)
Get	/v1/articles/	Read all articles written by user_id (user_id is an optional parameter, if it is empty all articles would be retrieved)
Get	/v1/articles/{article_id}	Read a specific article
Post	/v1/comments/	Create a new comment for article_id (article_id is sent in the body)
Get	/v1/comments/	Read all comments of article_id (article_id is an optional parameter, if it is empty all comments would be retrieved)
Get	/v1/comments/{comment_id}	Read a specific comment

Async db session object

I started this project using SqlAlchemy to get the session object that communicates with the db. But then I figured CRUD are I/O bound operations, and because I am using FastAPI I might block the event-loop and make unwanted delays serving new clients while I am waiting for db operations. That's why I made a new branch switching to an async approach to make the Most out of the FastAPI server.

Caching mechanism

While implementing the find words endpoint that returns an object of the occurrences of each input word in each article I started to think about the nature of the articles table, which is rarely ever changed. Once an article is created - that's that. The number of occurrences of a word inside an article is something that doesn't change as well and I want to avoid calculating it twice. So I implemented a local cache object. When I need to find words occurrences, I check which ones are already inside the cache, query the db only for the new words that I never encountered before and finally update the cache for the words I queried. This makes the whole program much faster as finding words occurrences is an expensive operation that goes through all the articles in the system.

A problem with that approach is what happens when I create a new article? All word occurrences might get updated as all the words in the cache might be inside the new article as well.

What to do?

- One option is to remember all the articles that got created since the last cache update, and in the next find_words request, query all the words in the cache only for the new articles, and query all the words that are not in the cache on all the articles. Finally update the cache as always.
- Another option is for every article creation update the cache and find all the occurrences of the words in the cache in the new article. I think this approach might work well when articles get rarely created and find_words endpoint gets called much more.

This logic was a bit to complex for me to implement in this time frame, so I compromised for just clearing the cache after a new article gets created. I didn't mention at all the size of this cache, of course it cannot remember infinite amount of words, I would add something like LRU cache on the words on top of everything.

Blocking CPU operations

```python
@router.post("/find_words", tags=["articles"], response_model=WordsOccurecnes)
async def find_words(words: list[str], session: AsyncSession = Depends(get_session)):
    words_occurecnes = words_cache.get_words_occurences(words)
    logger.info(f"finished quering cache for word occurnces")

    words_outside_cache = words_cache.get_words_outside_cache(words)
    if words_outside_cache:
        articles = await crud.articles.get_all(session=session, author_id=None)
        articles_list = [article for article in articles]
        words_occurecnes_outside_cache = build_word_occurences_object(words_outside_cache, articles_list)
        logger.info(f"finished building word occurences from database")
        words_occurecnes.extend(words_occurecnes_outside_cache)
        words_cache.update_cache(words_occurecnes_outside_cache, words_outside_cache)

    return words_occurecnes


@router.post("/most_common_word", tags=["articles"], response_model=int | None)
async def most_common_word(word: str, session: AsyncSession = Depends(get_session)):
    articles = await crud.articles.get_all(session=session, author_id=None)
    articles_list = [article for article in articles]
    article_id = get_article_with_most_occurences_of_word(word, articles_list)
    logger.info(f"calculated artile id with most occurences of {word}")
    return article_id
```

These functions are endpoints in my service. Note that these are async function that use await to make the most use out of FastAPI. But the highlighted lines of code are sync, and are CPU bound operations. Potentially, they go through huge amounts of data and we really don't want those things running and blocking our server. What can be done?

- We can utilize a module like celery. This module helps us keep a queue of tasks using a message broker like RabbitMQ or Redis in the backend. The tasks will be ran asynchronously with workers that run as different processes and the api will respond immediately with a task_id. We will have to add another endpoint for get_task_result(task_id) to retrieve the data. As clients, we will need to poll for the answer as the server cannot reach us directly unless we use some bi-directional protocol like web sockets.
- Another option that might be more suitable if the api would be exposed to huge amounts of network traffic - have a completely different service that runs on a cluster of Kubernetes or another container orchestration technology. Most principles would be similar to the first solution

like queueing tasks and querying for the tasks status afterwards, but we would have to implement the message publishing and have the new service subscribe to get tasks. We could have the service replicated and auto scaling based on the amount of messages in the queue.

```python
def build_word_occurences_object(words: list[str], articles: list[models.Article]) -> WordsOccurecnes:
    words_occurences: WordsOccurecnes = []
    for word in words:
        word_occurences = {}
        for article in articles:
            offsets = find_word_offsets_in_text(word, article.content)
            if offsets:
                if word in word_occurences:
                    word_occurences[word].append({"article_id": article.id, "offsets": offsets})
                else:
                    word_occurences[word] = [{"article_id": article.id, "offsets": offsets}]
        if word_occurences:
            words_occurences.append(word_occurences)
    return words_occurences
```

Something else that I just stumbled my mind upon is that this particular functions runs a nested loops to check for offsets of each word in each article. Every couple of (word, article) is independent of every other couple, so those things could have been calculated as difference processes. We could use some process pool and give each a (word, article) and gather all results in the end. This type of approach works well on CPU heavy tasks, of course this task is not too heavy computationally but it could be more reasonable if it was.

If I had more time

I would implement more testing.

- unit testing all the components.
- Testing the api endpoints.
- Making complex system test cases of creating objects, reading objects, making text processing queries and checking that all results are as expected.