# Projekt - News App

Bazy danych, 24 maj 2022

Mikołaj Klimek
Wojciech Jasiński
Andrzej Starzyk

## Wstęp

Jako temat projektu wybraliśmy stworzenie aplikacji webowej z newsami z kilku kategorii.

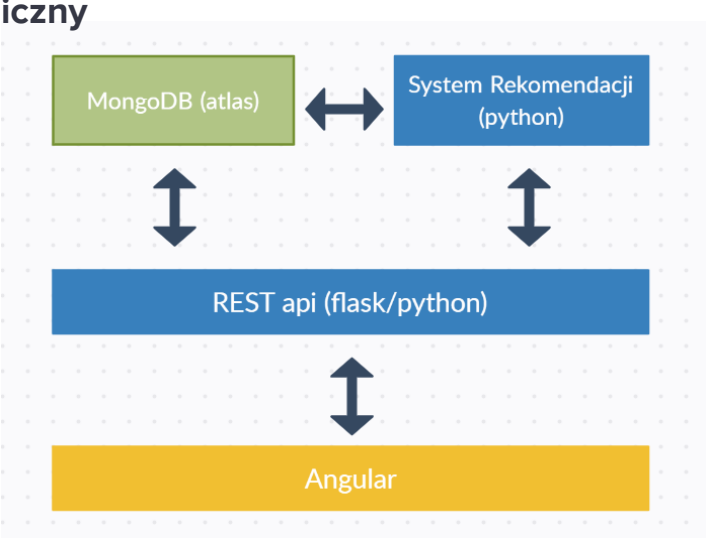
## Założenia projektu

- Strona posiada osobne sekcje tematyczne, np: finanse, sport, pogoda, technologie
- Newsy posiadają nagłówek, zdjęcie tytułowe, treść
- Użytkownicy mogą dodawać komentarze do wpisów oraz oceniać je
- Newsy mogą posiadać tagi
- Zapisywanie preferencji zalogowanych użytkowników
- System rekomendacji dla użytkownika na podstawie tagów

## Repozytorium

https://github.com/mklimek001/BazyDanych

## Stack technologiczny



## MongoDB/Atlas

Jako bazę danych wybraliśmy MongoDB. Dokumentowa baza danych bardzo dobrze sprawuje się z dynamicznie rozwijającą się aplikacją oraz z mało sztywnym modelem danych. Atlas bardzo ułatwił pracę nad projektem, ponieważ nie musieliśmy zajmować się konfiguracją lokalnego serwera bazodanowego, ani rozbieżnościami wynikającymi z różnic ustawień lub zawartości u osób pracujących nad projektem.

## Flask/python

Pierwotnym pomysłem było, by backend implementować w PHP, jednak szybko zmieniliśmy nasz wybór na Flask. Posiada on znacznie lepsze biblioteki i narzędzia do współpracy z bazami danych MongoDB m.in. pymongo, pydantic, flask-pymongo. System rekomendacji również o wiele łatwiej jest implementować w pythonie z powodu wygodnej pracy z macierzami rzadkimi używając bibliotek numpy i scipy.

## Angular

Choć zaplanowaliśmy dość prosty interfejs użytkownika, użyliśmy Angulara. Dość ciężki sprzęt, jak takie zadanie, ale jest to frontendowy framework jaki wszyscy dobrze znaliśmy.

## Schemat bazy danych

W bazach dokumentowych nie obowiązuje sztywny schemat, jednak w jakiś sposób musimy utrzymać porządek w bazie danych. Wyodrębniliśmy odpowiednie kolekcje oraz użyliśmy walidatorów (modeli) dokumentów.

## Kolekcje

W bazie danych umieściliśmy kolekcje:

- **articles** - kolekcja, w której przechowywane są dane dotyczące artykułów dostępnych dla użytkowników serwisu, wraz z tytułem, treścią, zdjęciem głównym, kategorią, tagami, datą dodania i ostatniej modyfikacji oraz podstawowymi danymi dotyczącymi interakcji z użytkownikami serwisu
- **comments** - kolekcja zawierająca informacje dotyczące komentarzy, każdy dokument, oprócz treści komentarza zawiera również identyfikatory pozwalające na połączenie go z konkretnym artykułem oraz użytkownikiem, który ten artykuł stworzył
- **users** - kolekcja przechowująca informacje dotyczące użytkowników serwisu. Każdy dokument zawiera imię i login użytkownika, a także hasło w zaszyfrowanej postaci. Baza

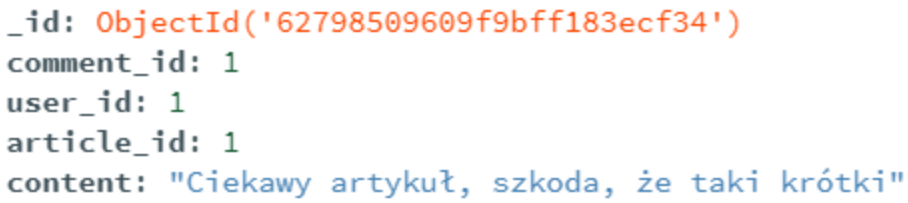
przechowuje też informacje o aktywności użytkownika w serwisie oraz o polecanych dla niego artykułach

- **interactions** - kolekcja, w której magazynowane są dane dotyczące aktywności użytkowników w serwisie w czasie kolejnych dni. Realizowana jako bucket pattern.

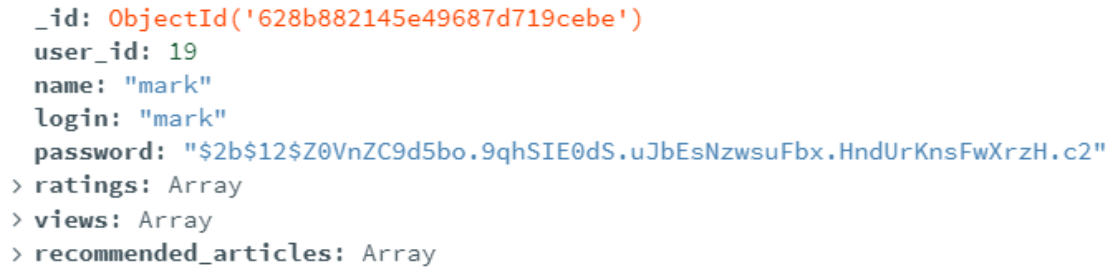
## Przykładowe dokumenty

- articles

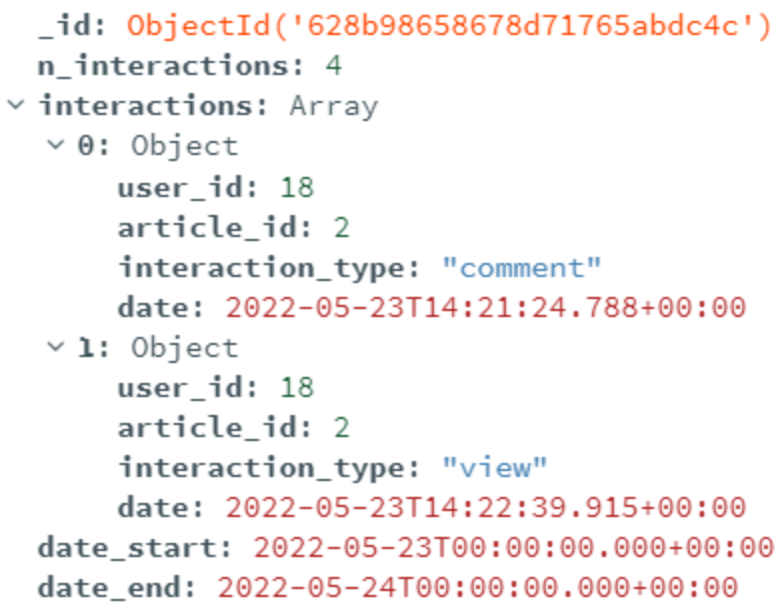

- comments




- users



- interactions




## Modele dokumentów

Article

Charakterystycznym dla dokumentowych baz danych jest użycie tzw Computed Field. Mimo, że moglibyśmy obliczać średnią ocenę dla artykułu, sumując wszystkie oceny cząstkowe w momencie gdy jej potrzebujemy, lepszym pomysłem jest dodatkowo zapisywać liczbę oraz sumę ocen. Ten drobny zabieg znacznie usprawnia obliczanie sre

```
class Article(BaseModel):
   id: Optional[PydanticObjectId] = Field(None, alias="_id")
   article_id: int
   title: str
   tags: List[str]
   category: str
   content: str
   image_source: str

   # computed properties
   n_of_grades: int = 0
   sum_of_grades: int = 0
   n_of_views: int = 0
```

```
    # dates
    date_added: Union[Optional[datetime], str] = datetime.utcnow()
    date_updated: Union[Optional[datetime], str] = datetime.utcnow()
```

Comment

```
class Comment(BaseModel):
    id: Optional[PydanticObjectId] = Field(None, alias="_id")
    comment_id: Optional[int]
    user_id: int
    article_id: int
    content: str
```

User

```
class User(UserMixin, BaseModel):
    id: Optional[PydanticObjectId] = Field(None, alias="_id")
    user_id: int
    name: str
    login: str
    password: Optional[str]
    ratings: List[Rating] = []
    views: List[View] = []

    recommended_articles: List[int] = []

    date_added: Optional[datetime] = datetime.utcnow()
    date_updated: Optional[datetime] = datetime.utcnow()
```

```
class Rating(BaseModel):
    article_id: int
    grade: int
```

```
class View(BaseModel):
    article_id: int
    n_view: int
```

Interactions

Warte uwagi jest tutaj, charakterystyczne dla dokumentowych baz danych, zastosowanie Bucket Pattern. Spodziewamy się bardzo wielu interakcji, jakie mielibyśmy zapisać w bazie, więc zagnieżdżanie tej informacji w dokumencie użytkownika lub artykułu byłoby słabym pomysłem. Luźne wrzucanie pojedynczego dokumentu dla każdej interakcji także powodowałoby bałagan. Grupujemy więc interakcje dniami (większy serwis mógłby grupować minutowo). Technika ta pozwala też na wygodne sortowanie po datach oraz szybki dostęp do określonych agregowanych wartosci (w naszym przypadku liczbę wyświetleń dziennie).

```
class Interaction(BaseModel):
   _id = Field(None, alias="_id")
   user_id: int
   article_id: int

   interaction_type: str

   date: datetime = datetime.utcnow()
```

```
class InteractionBucket(BaseModel):
   _id = Field(None, alias="_id")
   n_interactions: int = 0
   interactions: List[Interaction] = []

   date_start: Optional[datetime] = datetime.utcnow()\
       .replace(hour=0, minute=0, second=0, microsecond=0)
   date_end: Optional[datetime] = date_start + BUCKET_SIZE
```

## REST api

### Articles

- Pobieranie artykułów z bazy wraz z paginacją

```python
@app.route("/articles/", methods=["GET"])
def list_articles():
    """
    GET a list of articles, paginated.

    Params:
        page (int): The page number to return.
        per_page (int): The number of results per page.
        sort (str): The field to sort by. Defaults to 'date_added'.
        order ('desc', 'asc'): The direction to sort by. Defaults to 'desc'.
    """
    paginate_params, paginate_metadata = get_sort_and_paginate_params(request)
    page, per_page, sort, order = paginate_params

    cursor = articles.find().sort(sort, order)\
        .skip(per_page * (page - 1)).limit(per_page)

    article_count = articles.count_documents({})

    return {
        "articles": [Article(**doc).to_json() for doc in cursor],
        **paginate_metadata,
        "article_count": article_count,
    }
```

- Pobieranie wszystkich artykułów z danej kategorii

```python
@app.route("/articles/category/<string:given_category>", methods=["GET"])
def find_articles_with_category(given_category):
    """GET a list of articles, paginated."""
    cursor = articles.find({"category": given_category})
    return {"articles": [Article(**doc).to_json() for doc in cursor]}
```

- Pobieranie wszystkich artykułów z podanym tagiem

```python
@app.route("/articles/tag/<string:given_tag>", methods=["GET"])
def find_articles_with_tag(given_tag):
    """GET a list of articles with a given tag."""
    cursor = articles.find({"tags": {"$all": [given_tag]}})
    return {"articles": [Article(**doc).to_json() for doc in cursor]}
```

- Pobieranie listy artykułów rekomendowanych dla zalogowanego użytkownika

```python
@app.route("/articles/recommended", methods=["GET"])
@login_required
def get_recommended_articles():
    """GET a list of recommended articles for a user that's logged in."""

    user_id = current_user.user_id
    recommended = users.find_one({"user_id": user_id})["recommended_articles"]

    paginate_params, paginate_metadata = get_sort_and_paginate_params(request)
    page, per_page, sort, order = paginate_params

    cursor = articles.find({"article_id": {"$in": recommended}}) \
        .sort(sort, order) \
        .skip(per_page * (page - 1)).limit(per_page)

    return {
        "articles": [Article(**doc).to_json() for doc in cursor],
        **paginate_metadata
    }
```

- Pobieranie listy artykułów dla użytkownika o podanym ID

```python
@app.route("/articles/recommended/<int:user_id>", methods=["GET"])
def get_recommended_articles_userid(user_id):
    """GET a list of recommended articles for a user."""
    recommended = users.find_one({"user_id": user_id})["recommended_articles"]

    paginate_params, paginate_metadata = get_sort_and_paginate_params(request)
    page, per_page, sort, order = paginate_params

    cursor = articles.find({"article_id": {"$in": recommended}})\
        .sort(sort, order)\
        .skip(per_page * (page - 1)).limit(per_page)

    return {
        "articles": [Article(**doc).to_json() for doc in cursor],
        **paginate_metadata
    }
```

- Pobieranie artykułu z danym ID

```python
@app.route("/articles/<int:given_id>", methods=["GET"])
def get_article(given_id):
    """GET an article by its ID."""
    this_article = articles.find_one_or_404({"article_id": given_id})
    return Article(**this_article).to_json()
```

- Pobieranie listy tagów:

```python
@app.route("/articles/tags", methods=["GET"])
def find_all_tags():
    """GET a list of all tags."""
    alls = articles.find()
    all_tags = []
    for article in alls:
        c_article = Article(**article)
        for tag in c_article.tags:
            if tag not in all_tags:
                all_tags.append(tag)

    return {"tags": all_tags}
```

- Pobieranie listy kategorii:

```python
@app.route("/articles/categories", methods=["GET"])
def find_all_categories():
    """GET a list of all categories."""
    alls = articles.find()
    all_categories = []
    for article in alls:
        c_article = Article(**article)
        category = c_article.category
        if category not in all_categories:
            all_categories.append(category)

    return {"categories": all_categories}
```

- Dodawanie nowego artykułu:

```python
@app.route("/articles/", methods=["POST"])
def add_article():
    """POST a new article."""
    new_id = 0
    cursor = articles.find().sort("article_id", -1).limit(1)
    for curr_article in cursor:
        op_article = Article(**curr_article)
        new_id = op_article.article_id
    new_id += 1

    raw_article = request.get_json()
    raw_article["article_id"] = new_id

    try:
        article = Article(**raw_article)
    except ValidationError as e:
        return {"validation error": e.errors()}, 400

    articles.insert_one(article.to_bson())
    return article.to_json()
```

- Usuwanie artykułu o podanym ID:

```python
@app.route("/articles/<int:given_id>", methods=["DELETE"])
def delete_article(given_id):
    """DELETE an article by its ID."""
    deleted_article = articles.find_one_and_delete(
        {"article_id": given_id},
    )
    if deleted_article:
        return Article(**deleted_article).to_json()
    else:
        flask.abort(404, "Article not found")
```

- Aktualizowanie artykułu o podanym ID:

```python
@app.route("/articles/<int:given_id>", methods=["PUT"])
def update_article(given_id):
    """Update an article by its ID."""
    article = Article(**request.get_json())
    articles.find_one_and_update(
        {"article_id": given_id},
        {"$set": article.to_bson()}
    )
    up_article = articles.find_one_or_404({"article_id": given_id})
    return Article(**up_article).to_json()
```

## Comments

- Pobranie wszystkich komentarzy do artykułu o podanym ID:

```python
@app.route("/comments/article/<int:article_id>", methods=["GET"])
def find_comments_to_article(article_id):
    cursor = comments.find({"article_id" : article_id})
    return {"comments": [Comment(**doc).to_json() for doc in cursor]}
```

- Usunięcie komentarza o podanym ID:

```python
@app.route("/comment/<int:given_id>", methods=["DELETE"])
def delete_comment(given_id):
    deleted_comment = comments.find_one_and_delete(
        {"comment_id": given_id},
    )
    if deleted_comment:
        return Comment(**deleted_comment).to_json()
    else:
        flask.abort(404, "Comment not found")
```

- Aktualizacja komentarza o podanym ID:

```python
@app.route("/comment/<int:given_id>", methods=["PUT"])
def update_comment(given_id):
    comment = Comment(**request.get_json())
    comments.find_one_and_update(
        {"comment_id": given_id},
        {"$set": comment.to_bson()}
    )
    up_comment = comments.find_one_or_404({"comment_id": given_id})
    return Comment(**up_comment).to_json()
```

- Dodanie nowego komentarza do bazy:

```python
@app.route("/comments/", methods=["POST"])
def add_comment():
    new_id = 0
    cursor = comments.find().sort("comment_id", -1).limit(1)
    for curr_com in cursor:
        op_com = Comment(**curr_com)
        new_id = op_com.comment_id
    new_id += 1

    raw_com = request.get_json()
    raw_com["comment_id"] = id

    comment = Comment(**raw_com)
    comments.insert_one(comment.to_bson())
    return comment.to_json()
```

## Users

- Pobranie listy wszystkich użytkowników serwisu wraz z paginacją:

```python
@app.route("/users/")
def list_users():
    page = int(request.args.get("page", 1))
    per_page = 10  # A const value.

    cursor = users.find().sort("name").skip(per_page * (page - 1)).limit(per_page)

    users_count = users.count_documents({})

    links = {
        "self": {"href": url_for(".list_users", page=page, _external=True)},
        "last": {
            "href": url_for(
                ".list_users", page=(users_count // per_page) + 1, _external=True
            )
        },
    }
    if page > 1:
        links["prev"] = {
            "href": url_for(".list_users", page=page - 1, _external=True)
        }
    if page - 1 < users_count // per_page:
        links["next"] = {
            "href": url_for(".list_users", page=page + 1, _external=True)
        }

    return {
        "users": [User(**doc).to_json() for doc in cursor],
        "_links": links,
    }
```

- Pobranie danych dotyczących użytkownika o wskazanym ID:

```python
@app.route("/users/<int:given_id>", methods=["GET"])
def get_user(given_id):
    this_user = users.find_one_or_404({"user_id": given_id})
    return User(**this_user).to_json()
```

- Usunięcie użytkownika o wskazanym ID:

```python
@app.route("/users/<int:given_id>", methods=["DELETE"])
def delete_user(given_id):
    deleted_user = users.find_one_and_delete(
        {"user_id": given_id},
    )
    if deleted_user:
        return User(**deleted_user).to_json()
    else:
        flask.abort(404, "Article not found")
```

## Auth

- Logowanie do systemu:

```python
@app.route("/login", methods=["POST"])
def login():
    """Log in a user."""
    login = request.json.get("login")
    password = request.json.get("password")

    user = users.find_one({"login": login})
    if user is None:
        return {"message": "User not found"}, 404
    if not bcrypt.checkpw(password.encode("utf-8"), user["password"].encode("utf-8")):
        return {"message": "Invalid credentials"}, 401

    login_user(User(**user), duration=timedelta(days=1))
    return {"message": "Logged in successfully."}, 200
```

- Rejestracja i utworzenie nowego użytkownika:

```python
@app.route("/signup", methods=["POST"])
def signup():
    """Sign up a user."""

    login = request.json.get("login")
    password1 = request.json.get("password1")
    password2 = request.json.get("password2")

    if users.find_one({"login": login}):
        return {'message': 'There already is a user with that login'}, 400
    if password1 != password2:
        return {'message': 'Passwords do not match'}, 400

    hashed = bcrypt.hashpw(password1.encode('utf-8'), bcrypt.gensalt())

    # create new user
    cursor = users.find().sort("user_id", -1).limit(1)
    user_id = cursor[0]["user_id"] + 1 if cursor else 1
    try:
        user = User(**request.json, password=hashed, user_id=user_id)
    except ValidationError as e:
        return {"validation error": e.errors()}, 400
    users.insert_one(user.to_bson())
    return {
        "message": "Signed up successfully.",
        "user": user.to_json()
    }, 200
```

- Wylogowanie:

```python
@app.route("/logout", methods=["POST"])
@login_required
def logout():
    """Log out a user."""
    logout_user()
    return {"message": "Logged out."}, 200
```

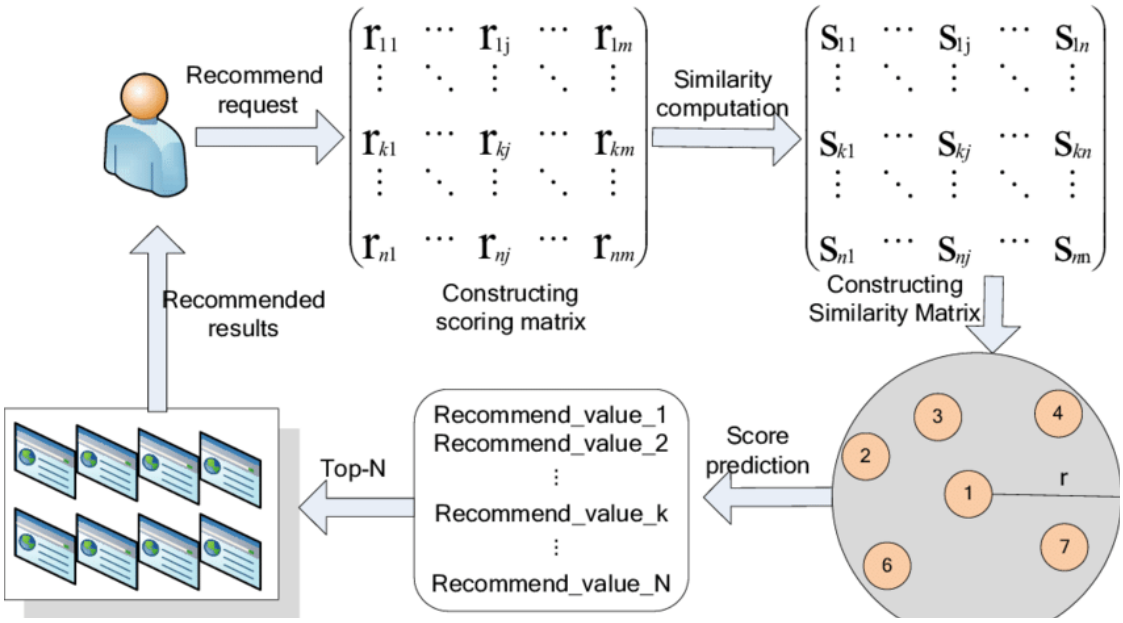
## Rekomendacje

System rekomendacji nie opiera się jawnie o tagi ani kategorie. Używamy zgromadzonych danych interakcji użytkowników z artykułami: wyświetleń, komentarzy oraz ocen, by wyznaczyć

pewne grupy zainteresowań i dopasować użytkownikowi artykuły, jakie mają największą szansę się mu spodobać.

## Koncepcja działania

Dla każdej pary użytkownik - artykuł możemy wyznaczyć pewną metrykę zaangażowania (na podstawie tego, ile razy użytkownik wyświetlił artykuł, czy pojął wysiłek napisania komentarza, czy ocenił artykuł - jeśli tak, jak wysoko). Wpisujemy wartości tej metryki w macierz rzadką.




Używając różnych algorytmów możemy przewidzieć wartość nieuzupełnionych pól - zgadujemy jak bardzo artykuł zainteresuje użytkownika. Wybraliśmy do tego prosty algorytm redukcji wymiarowości - Singular Value Decomposition. Metoda *Weighted Alternating Least Square* może się jednak okazać skuteczniejsza.

Dla danego użytkownika możemy uszeregować uzyskane wyniki malejąco oraz pominąć te, które dotyczą artykułów już wyświetlonych. Dostatecznie długą listę rekomendacji możemy zapisać w bazie danych.

## W praktyce

Docelowo raz dziennie, najlepiej w porze małego ruchu w serwisie, uruchamiany jest skrypt, który pobiera z bazy dane o interakcjach użytkowników i przeprowadzamy odpowiednie obliczenia. Jest to proces, który wymaga skomplikowanych zapytań do bazy danych, mocy obliczeniowej

oraz pamięci RAM, dlatego aktualizacji rekomendacji użytkowników nie przeprowadzamy niepotrzebnie często. Uzyskane wyniki dla każdego użytkownika zapisujemy w bazie danych.

## Przykładowe zapytanie

Pracując z MongoDB zdarza się pisać dość skomplikowane zapytania. Poniżej przykładowe zapytanie zwracające wartości metryki zaangażowania dla każdej pary użytkownik - artykuł na podstawie danych z ostatnich 90 dni.

```
scores = interactions.aggregate( [
{ "$match": { "date_start": {"$gt": datetime.utcnow() - timedelta(days=90)}}},
{ "$project": {"interactions": "$interactions"} },
{ "$unwind": "$interactions" },
{ "$group": {
   "_id": {
       "user_id":"$interactions.user_id",
       "article_id":"$interactions.article_id"
   },
   "score": {
       "$sum": {
           "$switch": {
               "branches": [
                   {"case": { "$eq": ["$interactions.type", "view"] }, "then": 1},
                   {"case": { "$eq": ["$interactions.type", "comment"] }, "then": 5},
                   {"case": { "$eq": ["$interactions.type", "grade"] }, "then": 3}
               ],
               "default": 0
           }
       }
   }
}}
])
```