

Baza danych medium społecznościowego

Łukasz Fabia 272724

Mikołaj Kubś 272662

Martyna Łopianiak 272682

Piotr Schubert 272659

Projektowanie baz danych wt 18:55

8 grudnia 2024

Spis treści

1	Etap 1: Faza koncepcyjna	2
1.1	Analiza świata rzeczywistego	2
1.1.1	Streszczenie - Zarys wymagań projektu	2
1.1.2	Potrzeby informacyjne	2
1.1.3	Czynności, wyszukiwania	2
1.1.4	Cele projektu	2
1.1.5	Zakres projektu	3
1.2	Wymagania funkcjonalne	3
1.3	ERD	4
2	Etap 2: Faza logiczna	4
3	Etap 3: Faza fizyczna	5
4	Etap 4: Faza fizyczna 2	6
5	Etap 5: Faza fizyczna 3	7
5.1	Projekt interfejsu graficznego	7
5.2	Kwerendy SQL	8
6	Etap 6: Faza fizyczna 4	9
6.1	Różne typy indeksów i ich zastosowanie	9
6.2	Obserwacje	10
7	Etap 7: Faza fizyczna 5	11
7.1	Sprawozdanie powykonawcze	11
7.1.1	Mocne strony projektu (S)	11
7.1.2	Słabe strony projektu (W)	12
7.1.3	Możliwości (O)	12
7.1.4	Zagrożenia (T)	12
7.1.5	Podsumowanie	12
7.2	Wprowadzone modyfikacje	13
7.2.1	Porzucone pomysły	13
7.2.2	Nowe encje	13
7.2.3	Rozdzielenie lokacji na 3 table	13
8	Etap 8: Faza logiczna I	14
8.1	Update ERD'a	14

1 Etap 1: Faza koncepcyjna

1.1 Analiza świata rzeczywistego

1.1.1 Streszczenie - Zarys wymagań projektu

Celem projektu jest stworzenie bazy danych do obsługi medium społecznościowego. Ma ona przechowywać informacje o użytkownikach, ich treściach, relacjach i aktywnościach. Baza powinna być zaprojektowana w sposób wydajny i skalowalny.

1.1.2 Potrzeby informacyjne

- Rejestracja i logowanie użytkowników z różnymi poziomami dostępu.
- Publikowanie i interakcje z treściami użytkowników (posty, polubienia, komentarze).
- Zarządzanie relacjami społecznymi (znajomości).
- Przechowywanie wiadomości prywatnych i historii aktywności.
- Tworzenie konwersacji z innymi użytkownikami
- Reportowanie postów z nieodpowiednimi treściami
- Tworzenie i zarządzanie stronami organizacji, firm, fanpage itd.
- Tworzenie i zarządzanie wydarzeniami

1.1.3 Czynności, wyszukiwania

- Wyszukiwanie użytkowników.
- Wyszukiwanie treści po hasztagach lub słowach kluczowych.
- Filtrowanie aktywności użytkownika, np. przeglądanie polubień i komentarzy.
- Wyszukiwanie relacji (np. znajomi użytkownika, osoby obserwujące daną osobę).
- Dodawanie innych użytkowników do znajomych i interakcja z treściami - dodawanie komentarzy i reakcji
- Tworzenie postów, wydarzeń, grup
- Konwersacja grupowa, pisanie wiadomości

1.1.4 Cele projektu

- S (Specific):** Zaprojektowanie bazy danych dla medium społecznościowego.
- M (Measurable):** Baza musi być wydajna, tzn. musi być w stanie obsługiwać dużą ilość użytkowników, co najmniej 20 000.
- A (Achievable):** Projekt zostanie zrealizowany przy użyciu PostgreSQL. Do stworzenia struktury tabel wykorzystany zostanie mechanizm ORM (Object Relational Mapping). Na koniec baza zostanie wypełniona danymi, aby przetestować jej wydajność.
- R (Relevant):** Przechowywanie profili użytkowników oraz interakcje między nimi są kluczowe dla funkcjonowania medium społecznościowego.
- T (Time-bound):** Praca nad projektem powinna zająć 2 miesiące.

1.1.5 Zakres projektu

Multimedia: W bazie przechowywane będą wyłącznie linki do plików na zewnętrznym serwerze.

Obsługa haseł: Wszystkie hasła w bazie będą hashowane.

1.2 Wymagania funkcjonalne

Streszczenie

Użytkownikom przypisany jest jeden z tych poziomów dostępu: admin, user lub guest.

Guest(Gość)

- Może przeglądać wybrane dane.

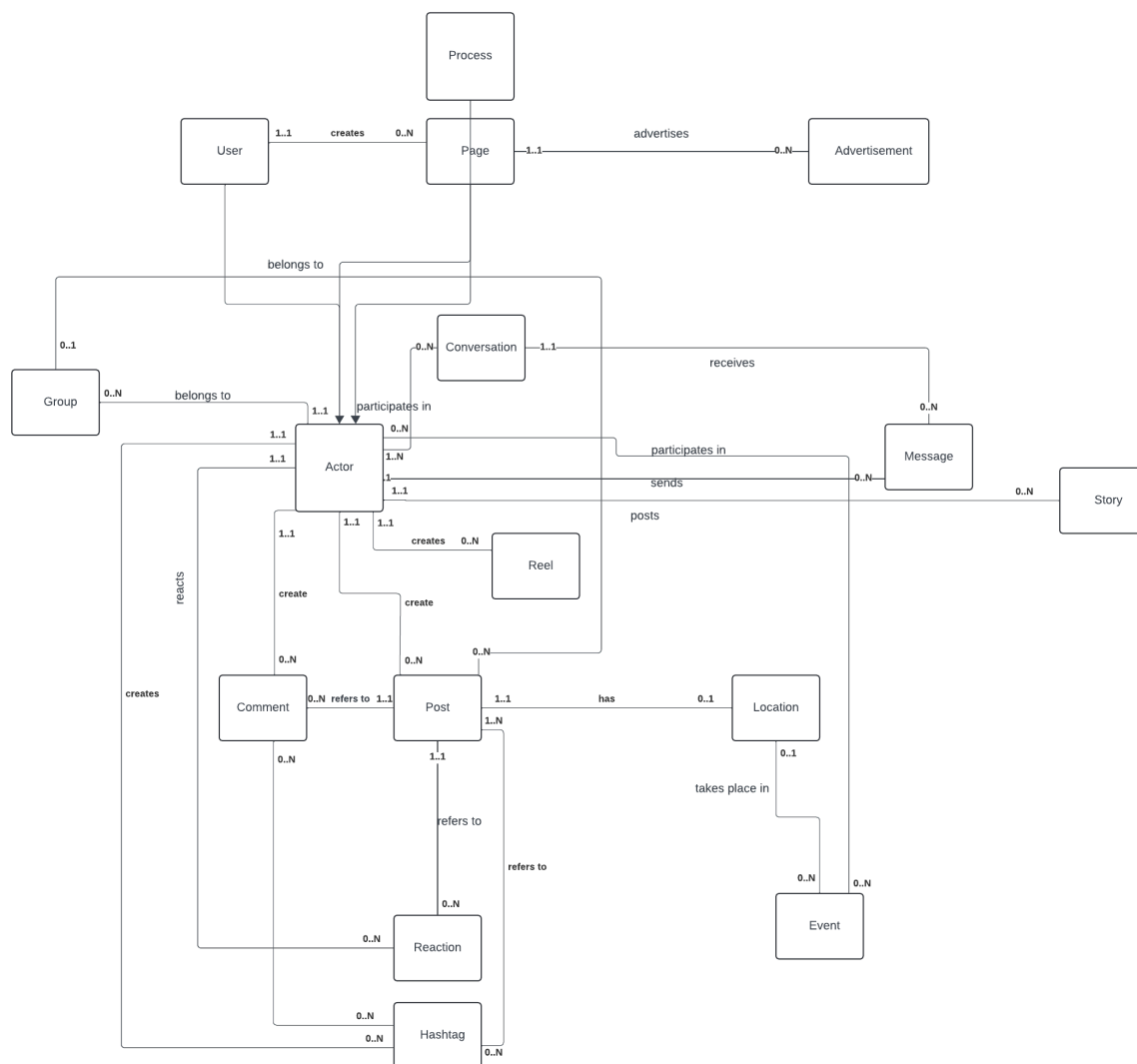
Admin

- Może przeglądać, edytować, usuwać, dodawać i przeglądać wszystkie treści, zarządza bazą i nadaje uprawnienia

User(Użytkownik)

- System umożliwia rejestrację oraz logowanie.
- Rejestracja wymaga imienia, nazwiska, daty urodzenia, hasła oraz maila.
- Logowanie wymaga maila i hasła.

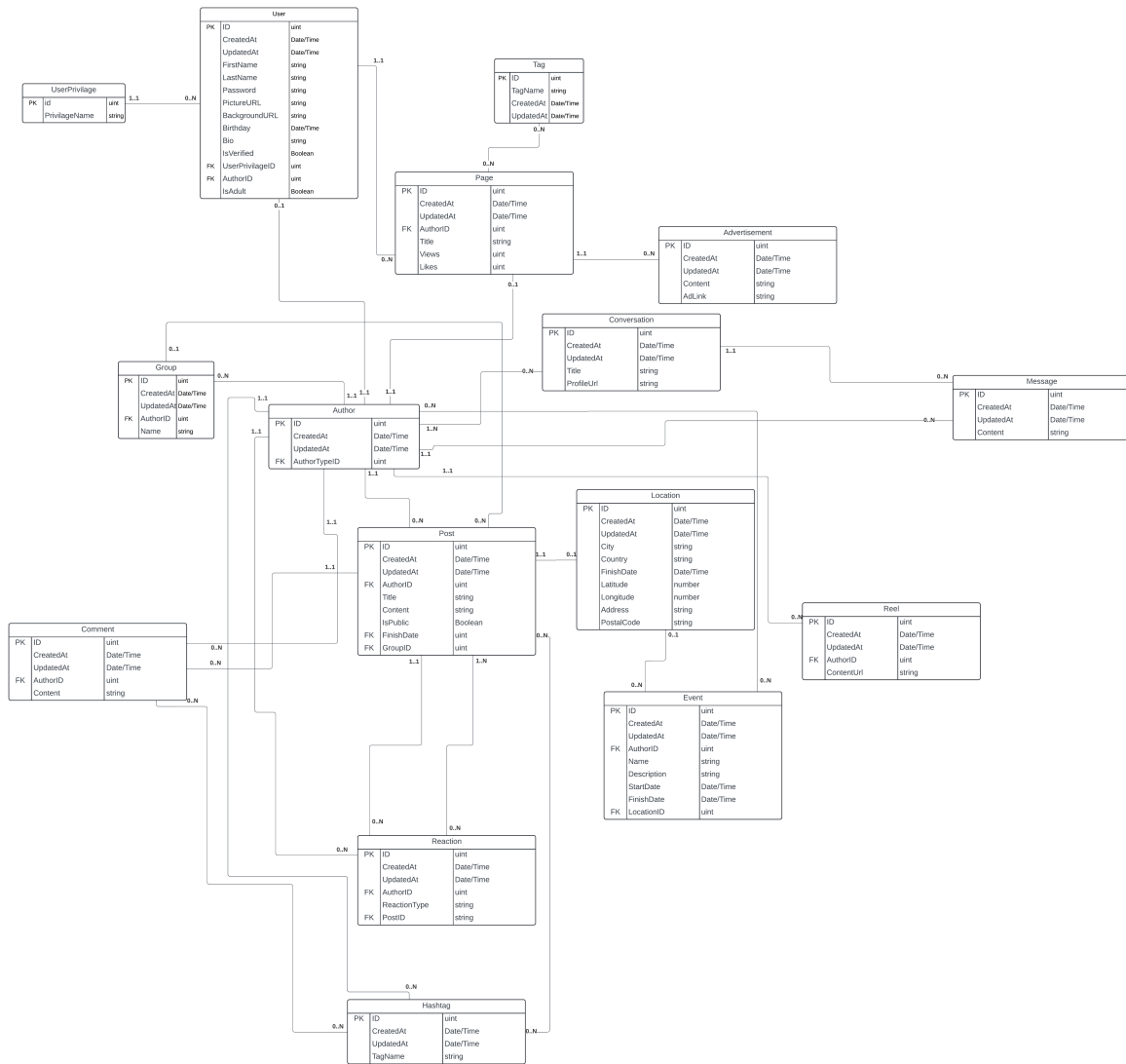
1.3 ERD



Rysunek 1: Diagram obiektowo-relacyjny

2 Etap 2: Faza logiczna

Do tabel zostały dodane atrybuty, tabele **Page** oraz **User** zostały uogólnione przez **Author**, który bierze udział w innych czynnościach. Baza została także sprowadzona do *III postaci normalnej*, przez co wydzielono kilka nowych tabel.



Rysunek 2: Diagram relacji

3 Etap 3: Faza fizyczna

Encje, które uległy zmianie:

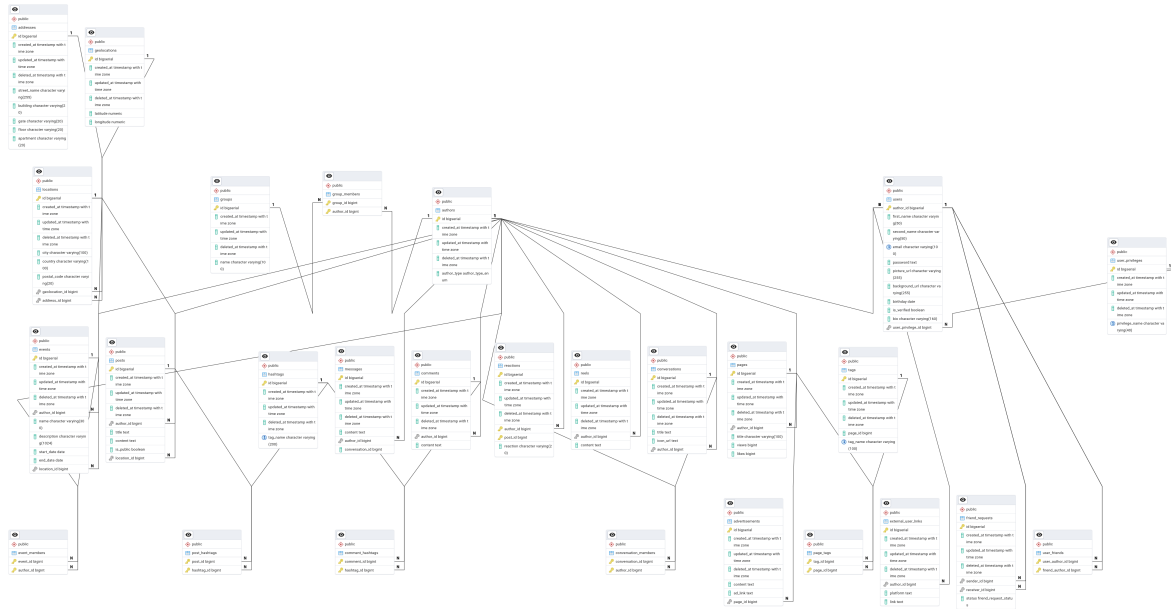
- Rozbicie lokalizacji na pomniejsze tabele
- Zamiana na enumy: typu autora oraz statusu zaproszenia do znajomych
- Dodanie do encji Conversation pole Members, które przechowuje id użytkowników biorących udział w konwersacji

Wykorzystano instrukcję CHECK do potwierdzenia poprawności danych w paru encjach. Przykładowo:

- 'A' nie może być przyjacielem 'A'
- 'A' nie może wysłać zaproszenia do znajomych do 'A'
- Czas rozpoczęcia wydarzenia musi być wcześniejszy niż czas zakończenia wydarzenia
- Grupa musi mieć od 1 do 10000000 członków.

Do stworzenia struktury bazy wykorzystano mechanizm ORM - ([gorm](#)). Sama baza wymagała dopracowania jeśli chodzi o enumeracje oraz reguły usuwania już bezpośrednio w systemie PostgreSQL (pgadmin).

Usuwanie: Każdy model posiada pole *DeletedAt* z indeksem. Podczas usuwania danego wiersza pole *DeletedAt* jest ustawiane na znacznik czasu (timestamp) wskazujący moment, w którym dane zostały usunięte. Dzięki temu baza obsługuje soft deleting. Oznacza to, że gdy użytkownik usunie swoje konto, będzie można je przywrócić. Taką operację obsługują na przykład serwisy takie jak Facebook.



Rysunek 3: Diagram relacji z PostgreSQL

4 Etap 4: Faza fizyczna 2

Standardowo kod można zobaczyć w repozytorium na gicie: [social media db](#).

Mając napisany ORM poprzecznie, w tym etapie pozostało napisać funkcje generujące dane do bazy. Rozwiązanie można podzielić na 3 części (od szczegółu do ogółu):

1. Dekorator, który będzie wykonywać określony blok `count` razy.
2. Funkcja, generująca dany typ danych np. generator użytkowników.
3. Odpowienie ułożenie wywołań.

Zapełnianie bazy danych dużą ilością danych zajmuje stosunkowo dużo czasu, może to być spowodowane przez unikalność niektórych atrybutów (biblioteka "męczy" się z generowaniem unikalnych sensownych danych), ale także przez wywołania które tworzą listy autorów dla np. eventów, ostatnim podejrzeniem może być nieoptymalnie napisany kod.

Język: [Go](#)

Generowanie sztucznych danych: [gofakeit](#)

Object relational mapping (ORM): [gorm](#)

Reszta rzeczy, które zostały wykonane:

- Funkcja usuwająca dane z tabel
- Własne implementacje niektórych danych np. `Title`, `Birthday`

- Konfiguracja loggera
- Funkcja haszująca hasło

5 Etap 5: Faza fizyczna 3

W tej fazie projektu stworzyliśmy projekt interfejsu graficznego w programie Figma oraz napisaliśmy 10 nietrywialnych kwerend w SQL.

5.1 Projekt interfejsu graficznego

Zaprojektowaliśmy następujące 10 widoków do aplikacji mobilnej:

- Strona główna
- Strona komentarzy i reakcji
- Strona profilu
- Strona strony (page)
- Strona grupy
- Strona wydarzenia
- Strona konwersacji
- Strona panelu konwersacji
- Strona rolek (reel)
- Strona ze znajomymi

Podczas projektowania zauważyliśmy parę drobnych rzeczy, które można by było poprawić - np. brak ikony czy grafiki tła dla strony. Tak więc poprawiliśmy takie nieścisłości i zaktualizowaliśmy kod seeder.go, który zajmuje się generowaniem tych danych.

Pełny projekt interfejsu graficznego znajduje się w załączonym pliku "figma.pdf".



Rysunek 4: 1 z widoków interfejsu graficznego

5.2 Kwerendy SQL

Poniżej znajdują się 10 kwerend SQL, które zostały napisane w celu przeszukania i stworzenia raportów na podstawie bazy danych:

- 10 najpopularniejszych hashtagów z ostatnich 7 dni
- Konwersacje autora
- Publiczne posty autora
- Średnia liczba znajomych
- Średnia liczba poszczególnych reakcji na posty w ostatnim miesiącu
- Wydarzenia w danym mieście (np. Saint Petersburg)
- Wydarzenia, w których wezmą udział znajomi autora
- Tagi stron, posortowane po średniej liczbie polubień i wyświetleń
- Użytkownicy, z którymi użytkownik ma wspólnych znajomych
- Użytkownicy z największą średnią liczbą reakcji na posty

	user_author_id bigint 🔒	first_name character varying (50) 🔒	second_name character varying (50) 🔒	mutualfriendscount bigint 🔒
1	2	Jazmin	Donnelly	3
2	247	Tess	Gusikowski	2
3	281	Francisca	Hoppe	2
4	154	Micheal	Jewess	2
5	326	Abby	Miller	2
6	244	Jana	Spinka	2
7	460	Lisette	Bednar	1
8	328	Darrion	Blanda	1
9	107	Danny	Boehm	1
10	441	Bernhard	Bogan	1

Rysunek 5: Wynik kwerendy wyszukującej użytkowników ze wspólnymi znajomymi

	id [PK] bigint	name character varying (300)	start_date date	end_date date	numberoffriendsattending bigint
1	5507	Seminar Civis Analytics	2024-11-18	2024-11-28	3
2	7711	Summit Robinson + Yu	2024-11-18	2024-11-24	2
3	4220	Festival Embark	2024-11-18	2024-11-21	2
4	10376	Party TransUnion	2024-11-18	2024-11-28	2
5	6989	Seminar Bekins	2024-11-18	2024-11-22	2
6	10096	Party Morgan Stanley	2024-11-18	2024-11-26	2
7	7596	Summit Weather Decision Technologies	2024-11-18	2024-11-28	2
8	11376	Party iRecycle	2024-11-18	2024-11-27	2
9	8206	Summit StreetCred Software, Inc	2024-11-18	2024-11-27	2
10	5778	Summit xDayta	2024-11-18	2024-11-28	2
11	4122	Webinar KPMG	2024-11-18	2024-11-25	2
12	153	Summit SmartProcure	2024-11-18	2024-11-21	2
13	5392	Summit Ensco	2024-11-18	2024-11-24	2
14	11168	Party StreamLink Software	2024-11-18	2024-11-26	2
15	2615	Seminar CitySourced	2024-11-18	2024-11-27	2
16	9197	Festival American Red Ball Movers	2024-11-18	2024-11-22	2
17	1941	Party Recargo	2024-11-18	2024-11-28	2
18	11513	Workshop SAS	2024-11-18	2024-11-22	2

Rysunek 6: Wynik kwerendy wyszukującej wydarzenia, na które zapisali się znajomi użytkownika

6 Etap 6: Faza fizyczna 4

Przykładowy wynik działania EXPLAIN

```
Seq Scan on foo (cost=0.00..155.00 rows=10000 width=4)
```

Ogólnie można powiedzieć, że komenda do dostarcza bardzo dokładne informacje w porównaniu np. do **MySQL**.

Powyższa komenda to sposób w jaki zapytanie może zostać wykonane. W tym wypadku mamy wykonanie sekwencyjne, czyli skanujemy całą tabelę w celu znalezienia wierszy spełniających warunki w kwerendzie.

Koszt(cost) - wartość po lewej to koszt początkowy zapytania tutaj jest on równy 0.00, kolejna wartość to szacowany koszt operacji, wartość ta jest obliczana przez system bazodanowy.

Wiersze(rows) - ilość wierszy spełniająca dane kryteria.

Szerokość(width) - waga wiersza w bajtach

Zatem optymalizacja zapytań będzie polegała na minimalizacji tych "kosztów". W tym celu wykorzystuje się **indeksy**.

6.1 Różne typy indeksów i ich zastosowanie

Typ indeksu	Zastosowanie
B-Tree	Wyszukiwanie, sortowanie, zakresy, klucze główne, indeksy unikalne.
Hash	Szybkie porównania równości (=).
GiST	Dane przestrzenne, zakresy, hierarchie.
BRIN	Duże tabele z posortowanymi danymi, dane archiwalne.

6.2 Obserwacje

Indeksy nie pomogą w każdej operacji - na przykład dla kwerend, gdzie największym kosztem jest grupowanie, mogą niekoniecznie pomóc.

Hash nie przynosi znaczącej poprawy wydajności w naszych zapytaniach w porównaniu z B-tree.

Zapytania wykorzystujące funkcje agregujące są trudne do optymalizacji, a dodanie indeksów w większości przypadków nie przyniosło zauważalnych korzyści.

Wyjątek stanowi zapytanie **show_events.in_st_petersburg**, gdzie zastosowanie indeksu złożonego przyniosło wyraźną poprawę:

- Zastosowanie indeksu typu **B-tree** zmniejszyło koszt zapytania o około 10%.
- Dodanie indeksu typu **BRIN** dodatkowo obniżyło koszt o kolejne 10%.

Dla zapytań niewykorzystujących funkcji agregujących, dodanie indeksów pozwoliło na kilkukrotne zmniejszenie kosztów ich wykonania.

#	Node	Rows Plan	#	Node	Rows Plan
1.	→ Sort (cost=387.64..387.65 rows=5 width=537)	5	1.	→ Sort (cost=54.05..54.06 rows=5 width=537)	5
2.	→ Nested Loop Left Join (cost=0.28..387.58 rows=5 width=537)	5	2.	→ Nested Loop Left Join (cost=0.56..53.99 rows=5 width=537)	5
3.	→ Seq Scan on posts as p (cost=0.346..0.347 rows=5 width=524) Filter: (p_public AND (author_id = 1))	5	3.	→ Index Scan using idx_author_id on posts as p (cost=0.28..12.48 rows=5 width=524) Filter: (p_public Index Cond: (author_id = 1))	5
4.	→ Index Scan using locations_pkey on locations as l (cost=0.28..8.3 rows=1 width=29) Index Cond: (l = p.location_id)	1	4.	→ Index Scan using locations_pkey on locations as l (cost=0.28..8.3 rows=1 width=29) Index Cond: (l = p.location_id)	1

Rysunek 7: Przykładowe czasy zapytań bez i z indeksami.

#	Node	Rows Plan	#	Node	Rows Plan
1.	→ Sort (cost=1690.84..1692.04 rows=479 width=45)	479	1.	→ Sort (cost=1690.87..1691.26 rows=479 width=45)	478
2.	→ Aggregate (cost=1694.73..1699.52 rows=479 width=45)	479	2.	→ Aggregate (cost=1664.01..1668.79 rows=478 width=45)	478
3.	→ Nested Loop Inner Join (cost=16.92..1662.33 rows=479 width=45)	479	3.	→ Nested Loop Inner Join (cost=16.92..1661.62 rows=478 width=45)	478
4.	→ Hash Inner Join (cost=16.64..1204.66 rows=1314 width=16) Hash Cond: (on_event_id = l.event_id)	1314	4.	→ Hash Inner Join (cost=16.64..1204.66 rows=1312 width=16) Hash Cond: (on_event_id = l.event_id)	1312
5.	→ Seq Scan on event_members as em (cost=0..1013.96 rows=65796 width=16)	65796	5.	→ Seq Scan on event_members as em (cost=0..1013.96 rows=65796 width=16)	65796
6.	→ Hash (cost=16.49..16.49 rows=12 width=8)	12	6.	→ Hash (cost=16.49..16.49 rows=12 width=8)	12
7.	→ Index Only Scan using user_friends_pkey on user_friends as uf (cost=0.28..16.49 rows=12 width=8)	12	7.	→ Index Only Scan using user_friends_pkey on user_friends as uf (cost=0.28..16.49 rows=12 width=8)	12
8.	→ Index Scan using idx_event on events as e (cost=0.29..0.35 rows=1 width=37) Filter: (on_event_id = rowid) Index Cond: (e = on_event_id)	1	8.	→ Index Scan using idx_event on events as e (cost=0.29..0.35 rows=1 width=37) Filter: (on_event_id = rowid) Index Cond: (e = on_event_id)	1

Rysunek 8: Przykładowe czasy zapytań bez i z indeksami.

#	Node	Rows Plan	#	Node	Rows Plan	#	Node	Rows Plan
1.	→ Sort (cost=1578.76..1579.46 rows=279 width=42)	279	1.	→ Sort (cost=1578.76..1579.46 rows=279 width=42)	279	1.	→ Sort (cost=1578.76..1579.46 rows=279 width=42)	279
2.	→ Aggregate (cost=1684.66..1687.47 rows=279 width=42)	279	2.	→ Aggregate (cost=1684.66..1687.47 rows=279 width=42)	279	2.	→ Aggregate (cost=1684.66..1687.47 rows=279 width=42)	279
3.	→ Nested Loop Inner Join (cost=362.76..1681.91 rows=279 width=42)	279	3.	→ Nested Loop Inner Join (cost=362.76..1681.25 rows=279 width=42)	279	3.	→ Nested Loop Inner Join (cost=362.76..1681.25 rows=279 width=42)	279
4.	→ Hash Inner Join (cost=362.62..1681.76 rows=279 width=42) Hash Cond: (on_event_id = l.event_id)	279	4.	→ Hash Inner Join (cost=362.62..1681.76 rows=279 width=42) Hash Cond: (on_event_id = l.event_id)	279	4.	→ Hash Inner Join (cost=362.62..1681.76 rows=279 width=42) Hash Cond: (on_event_id = l.event_id)	279
5.	→ Bitmap Heap Scan on events as e (cost=0.11..792.51 rows=4384 width=42) Bitmap Cond: (on_event_id = rowid) AND (on_event_id = rowid) > 1 (cost=0.00)	4384	5.	→ Bitmap Heap Scan on events as e (cost=0.11..792.51 rows=4384 width=42) Bitmap Cond: (on_event_id = rowid) AND (on_event_id = rowid) > 1 (cost=0.00)	4384	5.	→ Bitmap Heap Scan on events as e (cost=0.11..792.51 rows=4384 width=42) Bitmap Cond: (on_event_id = rowid) AND (on_event_id = rowid) > 1 (cost=0.00)	4384
6.	→ Bitmap Heap Scan using idx_event_id_pkey on events as e (cost=0.11..792.51 rows=4384 width=42) Bitmap Cond: (on_event_id = rowid) AND (on_event_id = rowid) > 1 (cost=0.00)	4384	6.	→ Bitmap Heap Scan using idx_event_id_pkey on events as e (cost=0.11..792.51 rows=4384 width=42) Bitmap Cond: (on_event_id = rowid) AND (on_event_id = rowid) > 1 (cost=0.00)	4384	6.	→ Bitmap Heap Scan using idx_event_id_pkey on events as e (cost=0.11..792.51 rows=4384 width=42) Bitmap Cond: (on_event_id = rowid) AND (on_event_id = rowid) > 1 (cost=0.00)	4384
7.	→ Hash Inner Join (cost=108.22..249.56 rows=58 width=25) Hash Cond: (on_event_id = rowid)	58	7.	→ Hash Inner Join (cost=108.22..249.56 rows=58 width=25) Hash Cond: (on_event_id = rowid)	58	7.	→ Hash Inner Join (cost=108.22..249.56 rows=58 width=25) Hash Cond: (on_event_id = rowid)	58
8.	→ Seq Scan on addresses as a (cost=0..108 rows=5000 width=16)	5000	8.	→ Seq Scan on addresses as a (cost=0..108 rows=5000 width=16)	5000	8.	→ Seq Scan on addresses as a (cost=0..108 rows=5000 width=16)	5000
9.	→ Hash (cost=128.5..128.5 rows=58 width=25)	58	9.	→ Hash (cost=128.5..128.5 rows=58 width=25)	58	9.	→ Hash (cost=128.5..128.5 rows=58 width=25)	58
10.	→ Seq Scan on locations as l (cost=0..128.5 rows=58 width=25) Filter: (l.location_id = 16, l.location_id = 16)	58	10.	→ Seq Scan on locations as l (cost=0..128.5 rows=58 width=25) Filter: (l.location_id = 16, l.location_id = 16)	58	10.	→ Seq Scan on locations as l (cost=0..128.5 rows=58 width=25) Filter: (l.location_id = 16, l.location_id = 16)	58
11.	→ Index Only Scan using event_members_pkey on event_members as em (cost=0.28..0.37 rows=6 width=16) Index Cond: (on_event_id = rowid)	6	11.	→ Index Only Scan using event_members_pkey on event_members as em (cost=0.28..0.37 rows=6 width=16) Index Cond: (on_event_id = rowid)	6	11.	→ Index Only Scan using event_members_pkey on event_members as em (cost=0.28..0.37 rows=6 width=16) Index Cond: (on_event_id = rowid)	6
12.	→ Index Only Scan using event_members_pkey on event_members as em (cost=0.28..0.37 rows=6 width=16) Index Cond: (on_event_id = rowid)	6	12.	→ Index Only Scan using event_members_pkey on event_members as em (cost=0.28..0.37 rows=6 width=16) Index Cond: (on_event_id = rowid)	6	12.	→ Index Only Scan using event_members_pkey on event_members as em (cost=0.28..0.37 rows=6 width=16) Index Cond: (on_event_id = rowid)	6

Rysunek 9: Optymalizacja zapytania z wydarzeniami w Piotrogradzie

7 Etap 7: Faza fizyczna 5

7.1 Sprawozdanie powykonawcze

7.1.1 Mocne strony projektu (S)

Jedną z pierwszych mocnych stron projektu jest wydzielenie tabeli **authors**. Pozwoliło to na rozszerzenie możliwości tworzenia nowych tabel dla użytkowników (**user**, **page**). Dodatkowo w ten sposób pozbyto się *redundantnych* relacji, ponieważ zarówno **strona**, jak i **użytkownik** mogli w domyśle wykonywać te same czynności na innych tabelach.

Zastosowanie usuwania kaskadowego również okazało się być bardzo pomocne, ponieważ w ten sposób przy usuwaniu jakichkolwiek danych mielibyśmy pewność, że dane pozostaną spójne.

Będąc w temacie usuwania, dane z bazy nie były całkowicie usuwane (*hard delete*). To nie tylko pozwoliłoby na przywrócenie danych, na przykład konta niezdecydowanego użytkownika, ale również pozwoliłoby śledzić historię danych. Ponadto dane, które nie są widoczne dla użytkownika mogą posłużyć np. do analiz biznesowych.

Uzyskanie zgodności z przepisami, w tym z RODO w naszym projekcie wydaje się ułatwione przez dwa czynniki:

- Haszowanie haseł
- Wydzielenie danych osobowych do osobnej tabeli, dzięki czemu można łatwo zarządzać dostępem do tych danych, a także przeprowadzić *soft delete* użytkownika bez usuwania wszystkich rzeczy, które wytworzył.
- Usuwanie kaskadowe, które pozwala na szybkie usunięcie wszystkich danych związanych z użytkownikiem.
- Ograniczenie zbieranych danych do niezbędnych, co pozwala na zminimalizowanie ryzyka naruszenia RODO.

Skrypt generujący dane również jest mocną stroną projektu. Skrypt generuje tabele w bazie danych, dodając potrzebne ograniczenia do atrybutów, enumeracje oraz dość sensowne dane. Jest on solidną bazą pod część backendową (związaną z warstwą modeli np. dla *architektury warstwowej*) aplikacji. Dodatkowym atutem jest to, że generacja danych nie jest destruktywna - mając już dane w bazie danych, możemy spokojnie wywołać metodę odpowiedzialną za generowanie dodatkowych danych w określonych przez nas tabelach.

Dodatkowo można w prosty sposób zmienić system, dla którego ma być generowana baza - my korzystaliśmy z PostgreSQL, ale bez większych problemów dałoby się zmienić system na inny (na przykład SQLite, MySQL czy SQLServer) dzięki używaniu GORM'a, który jest właśnie kompatybilny z wieloma systemami.

Został napisany również skrypt, który automatycznie uruchomi bazę w kontenerze Dockerowym, dzięki czemu nie trzeba instalować samego PostgreSQL, PGAdmin czy innych pluginów, żeby zarządzać tą bazą - wystarczy podać parę podstawowych danych w pliku `.env` i uruchomić skrypt. Dodatkową zaletą jest to, że baza jest uruchamiana w izolowanym środowisku, co pozwala na łatwe testowanie różnych wersji bazy danych, a także na łatwe przenoszenie bazy na inne środowisko (np. z lokalnego na serwer produkcyjny).

Dodatkową automatyzacją jest integracja backendu z komendami umieszczonymi w Makefile'u. Dzięki temu można w prosty sposób uruchomić skrypt generujący dane, uruchomić bazę danych, przełożyć bazę i inne.

Co do wybranego systemu, system PostgreSQL, okazał się on być całkiem wydajnym pod względem operacji typu *CRUD* na tabelach. Dzięki dodaniu indeksów, kwerendy zostały jeszcze bardziej zoptymalizowane.

Wydaje nam się, że dzięki uogólnieniu wszystkich czatów do grupowych (czat 1-1 nie różni się formalnie od czatu wieloosobowego w naszej bazie) zmniejszamy ryzyko błędów przez utrzymywanie dwóch wersji konwersacji. Nie powinno to sprawić nam problemów implementacyjnych w przyszłości, a dodatkowo dodanie lub usunięcie użytkownika z konwersacji będzie bardzo proste.

Nasz projekt stara się oczywiście spełniać podstawowe założenia dobrej bazy danych - jest spójny, niepowtarzalny, znormalizowany, a także ma ograniczenia integralnościowe. Żadne dane nie występują

niepotrzebnie więcej niż raz. Encje wydają się być dobrze znormalizowane. Ograniczenia integralnościowe są zaimplementowane w postaci kluczy obcych, ograniczeń CHECK oraz unikalności. Wszędzie, gdzie to miało sens, wprowadziliśmy tabele asocjacyjne, aby uniknąć problemów związanych z relacjami wieloma do wielu - udało się je wszystkie usunąć z naszego projektu.

7.1.2 Słabe strony projektu (W)

Największą słabą stroną projektu jest dość wolno działający skrypt generujący dane. Dzieje się tak, gdy chcemy dodać więcej wierszy do tabeli - przez to, że skrypt wybiera foreign keys losowo wśród istniejących już elementów, często jest duża szansa niepowodzenia przez złamanie jakiejś zasady - daloby się to ulepszyć dzięki bardziej inteligentemu wybieraniu foreign keys. Wygenerowanie niektórych danych wymaga też skomplikowanej i zagnieżdżonej logiki tworzenia elementów. Być może problemem jest też biblioteka *faker*, która nie bardzo radziła sobie z generowaniem bardzo losowych, często unikalnych danych.

Okazuje się, że draw.io (narzędzie do modelowania diagramów, w naszym wypadku diagramów ERD) ma wbudowaną integrację z GORM'em, dzięki której wystarczyłoby przesunąć diagram nad okienko z kodem. Ostatecznie nie oszczędziłoby nam to dużo czasu, ale z pewnością ułatwiłoby pracę i zmniejszyłoby ryzyko literówek.

7.1.3 Możliwości (O)

Tak jak już wcześniej wspominaliśmy, baza jest dobrą podstawą do stworzenia systemu na wzór **Facebooka** czy **Twittera/X** poprzez zbudowanie jakiegoś prostego serwisu REST'owego. Dzięki temu, że serwis byłby faktycznie używany na większą skalę, zauważylibyśmy więcej problemów optymalizacyjnych. Pozwoliłoby to na dopracowanie bazy danych w miejscach, gdzie brakowałoby wydajności.

Jeśli chodzi o możliwości rozbudowy bazy to można by dodać np. lepszą obsługę mediów (wydzielenie osobnej tabeli). Można być też poprawić tabele związane z lokalizacjami i zamiast przechowywania danych koordynatów przechowywać typ **Point** do obsługi współrzędnych geograficznych wykorzystując plugin **PostGIS**. Można by wtedy ulepszyć sposób wyszukiwania wydarzeń użytkownikowi, poprzez szukanie ich w promieniu n m/km od wybranego punktu.

7.1.4 Zagrożenia (T)

Problemem baz relacyjnych jest trudność w horyzontalnym skalowaniu, ponieważ zapewnienie spójności danych i transakcyjności (ACID) w rozproszonej architekturze jest skomplikowane. Tradycyjnie łatwiej jest skalować je pionowo, czyli ulepszać pojedynczą maszynę, dodając więcej zasobów, takich jak pamięć RAM, mocniejszy procesor czy szybsze dyski. Jednak przy zastosowaniu odpowiednich technik, takich jak sharding czy replikacja, możliwe jest również skalowanie horyzontalne baz relacyjnych. Tabela, która może być problematyczna w utrzymaniu to, na przykład, tabela z wiadomościami, które użytkownicy wysyłają sobie wzajemnie. W przypadku tabeli z wiadomościami może to być problem, ponieważ może ona gwałtownie rosnąć. Dlatego lepszym rozwiązaniem byłoby podejście hybrydowe - przeniesienie niektórych tabel do baz NOSQL, dzięki czemu można by rozproszyć obliczenia, a nasz system nie miałby problemów z powolnym przeszukiwaniem bazy.

Baza wymagałaby też dużo optymalizacji, by wspierać miliony użytkowników stabilnie - aktualnie ciężko powiedzieć, jak dobrze by sobie radziła w takiej sytuacji - można by wprowadzać więcej indeksów które najlepiej radzą sobie z konkretną sytuacją, ale ciężko powiedzieć, czy to by wystarczyło.

7.1.5 Podsumowanie

Próbując wprowadzić produkt z taką bazą danych w życie, mielibyśmy wielki problem z konkurencją - Facebook, Twitter, Instagram, TikTok, LinkedIn, Pinterest, Reddit, Snapchat, Tumblr, WhatsApp, YouTube, czyli największe platformy społecznościowe, mają już swoje miejsce na rynku. Nasz produkt musiałby być bardzo innowacyjny, aby przyciągnąć użytkowników.

Jednakże, nasz projekt jest dobrym punktem wyjścia do stworzenia takiego produktu. Baza danych jest solidna, choć wymagałaby jeszcze sporo optymalizacji. Skrypt generujący dane jest bardzo pomocny do debuggowania i byłby krytycznie potrzebny tworząc MVP, ale wymagałby jeszcze sporo pracy, aby działał szybciej i bardziej efektywnie.

Czym mógłby wyróżniać się potencjalny produkt oparty na naszej bazie danych? Dobrym punktem wyjścia jest to, że nasza baza jest dość generyczna - większość wcześniej wymienionych serwisów ma już podobne encje. Tak więc można by rozwinąć tę bazę w jednym z tych 4 przykładowych kierunków:

- **Baza danych dla GoLocal** - aplikacja do tworzenia i głosowania na wydarzenia dla samorządów i zwykłych ludzi, aplikacja w formie serwisu społecznościowego. Dodanie obsługi ankiet, opinii użytkowników o wydarzeniu, rozwinięcie tabel z lokalizacjami oraz wydarzeniami, obsługa użytkowników *premium*. Usprawnienie obsługi komunikacji między użytkownikami, połączenie serwera z bazą NOSQL do zarządzania wiadomościami. W dalekiej przyszłości można by zastanowić się nad dedykowaną bazą do obsługi systemu rekomendacji - hurtowanie danych historycznych (usuniętych profili, reakcji, wydarzeń, postów, komentarzy).
- **Baza danych dla serwisu społecznościowego dla programistów** - dodanie tabeli związanej z projektami, repozytoriami, commitami, pull requestami, itd. Można by też dodać tabelę związane z technologiami, językami programowania, frameworkami, itd.
- **Baza danych dla serwisu społecznościowego dla naukowców** - dodanie tabeli związanej z publikacjami, konferencjami, grantami, itd. Można by też dodać tabelę związane z dziedzinami nauki, konferencjami, itd.
- **Lepszy Twitter** - przez ostatnie zamieszanie w serwisie Twitter/X, można by stworzyć serwis, który byłby bardziej przejrzysty, miałby lepsze algorytmy rekomendacji, byłby bardziej stabilny i bardziej moderowany, itd.

7.2 Wprowadzone modyfikacje

7.2.1 Porzucone pomysły

Już w początkowej fazie tworzenia projektu doszliśmy do wniosku że opcja zgłaszania postów/treści przez użytkowników wymaga nieadekwatnego nakładu pracy w stosunku do korzyści jakie nasz projekt na tym etapie mógłby z tego zyskać. Może być to dobra funkcjonalność do dodania w ramach rozwoju aplikacji, jednak na początkowych etapach stwierdziliśmy, że postawimy na jakość, a nie ilość. Dlatego zrezygnowaliśmy z tego pomysłu, aby w pełni skupić się na pozostałych funkcjonalnościach. Podobnie stało się z encją “Story”. Jako że jej atrybuty praktycznie nie różniłyby się od obecnych już encji (Post, Reel) zdecydowaliśmy się na jej pominięcie w późniejszych etapach projektu. Jednak tak jak w przypadku zgłaszania postów jest to jak najbardziej funkcjonalność możliwa do dodania w przyszłości, a dzięki podobieństwu do obecnych już encji wdrożenie Story do naszej bazy danych powinno być nawet prostsze niż dodanie opcji zgłaszania postów.

7.2.2 Nowe encje

Na pierwszym diagramie ERD naszego projektu nie uwzględniliśmy encji “Tag”, która została dodana w następnym etapie. Encja ta określa typ prowadzonej strony. Nie uwzględniliśmy również encji “UserPrivilege”, która została dodana dopiero w fazie logicznej. Zdecydowaliśmy się wyodrębnić User Privilege do osobnej encji zamiast używać enuma, aby ułatwić modyfikację czy rozbudowę bazy danych oraz zmniejszyć ryzyko niespójności. Pojawiła się też encja “ExternalAuthorLinks” która pozwala przechowywać linki do innych social mediów użytkownika.

7.2.3 Rozdzielenie lokacji na 3 table

Encja związana z lokacją została rozdzielona na 3 osobne encje: Location, Geolocation, Address. Taka struktura danych niweluje powielanie informacji, ponieważ miasta, kody pocztowe czy nawet współrzędne geograficzne mogą być takie same dla wielu adresów. Dodatkowo możemy teraz w bardzo precyzyjny sposób określić adres, do jakiego się odwołujemy - mamy możliwość sprecyzowania ulicy, budynku, klatki schodowej, piętra i mieszkania. Dzięki temu, niezależnie od konwencji zapisu adresu (które mogą się różnić w zależności od kraju) użytkownik będzie w stanie wprowadzić go do naszej bazy danych. Poza tym rozdzielenie lokacji na osobne encje zmniejsza ryzyko niespójności, ułatwia modyfikacje i rozszerzanie bazy danych oraz sprawia, że zapytania na tabeli Location są bardziej wydajne. Jako że atrybuty tej encji zostały ograniczone do minimum, a bardziej złożone dane przechowywane są

w osobnych tabelach, podczas wykonywania na przykład zapytań dotyczących wydarzeń w danym miesiącu nie potrzebujemy importować danych ze wszystkich tabel - wystarczy nam sama tabela `Location` co zwiększa wydajność zapytań.

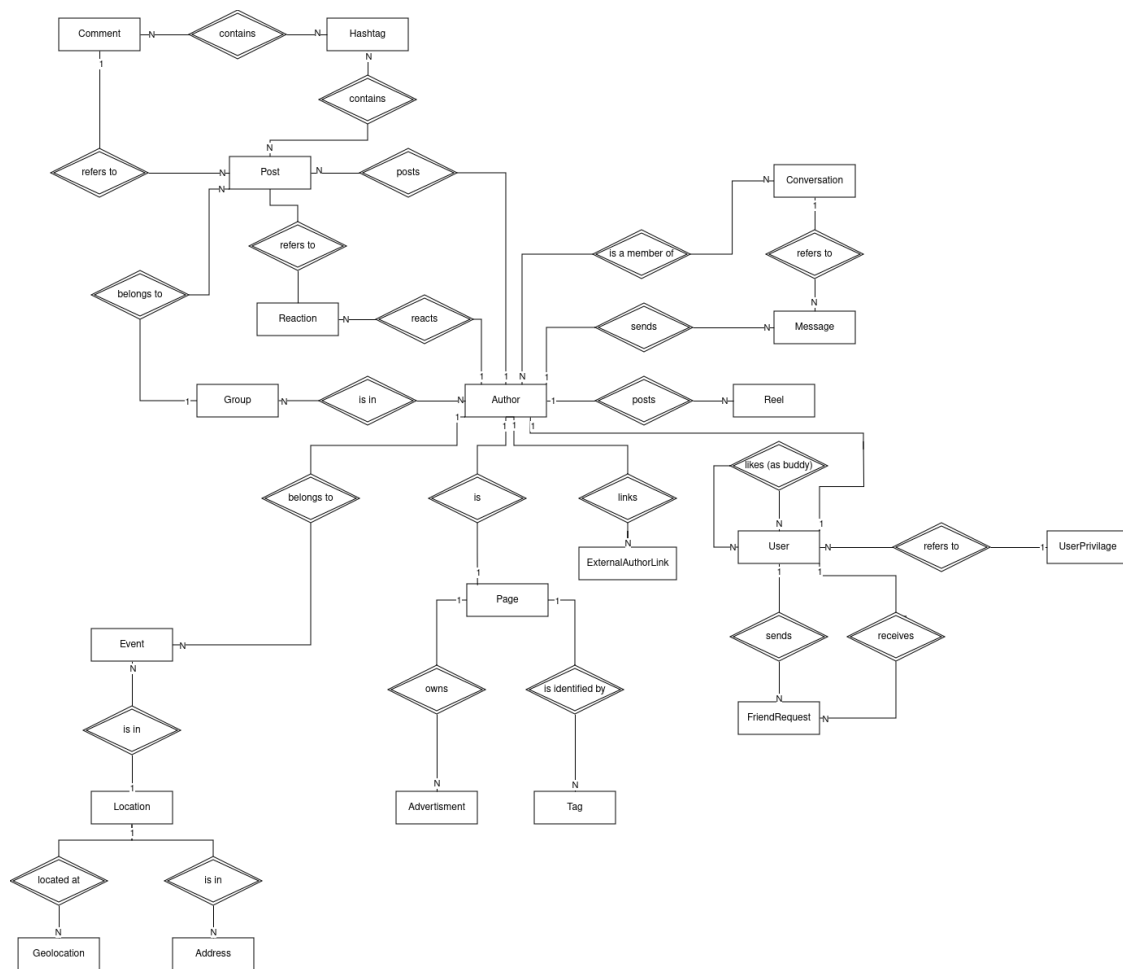
8 Etap 8: Faza logiczna I

8.1 Update ERD'a

Diagram erd został zaktualizowany - odzwierciedla schemat relacji w bazie oraz jest czytelniejszy względem poprzedniego. Zdecydowaliśmy się nie umieszczać szczytkowych artefaktów dla encji, ponieważ wprowadziło by to zbędne zamieszanie i wpłynęło by to na czytelność.

Zmieniono:

- Dodano `User Privilege`
- Rozbito `Location`, na pomniejsze encje
- Wyodrębniono nową encję `Tag`
- Usunięto encję `Story`
- Pokazaliśmy na diagramie relacje użytkownika z innymi użytkownikami (asocjacja *likes (as buddy)*) oraz podobnie pokazaliśmy wysyłanie zaproszeń do znajomych (asocjacji *receives, sends*).
- No i usuneliśmy encję `Process`, dodaną losowo.



Rysunek 10: Zaktualizowany ERD