



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ
Кафедра програмного забезпечення та комп'ютерних систем

Лабораторна робота №3

з дисципліни
«Засоби оптимізації роботи СУБД PostgreSQL»

Виконав: студент III курсу

ФПМ групи КП-82

Суходольський Євгеній Віталійович

Перевірив:

Радченко К.О.

Київ – 2020

Завдання

Варіант 18

Завдання роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL(*GIN*, *BTree*).
3. Розробити тригер бази даних PostgreSQL(*after update, insert*).

Логічна модель бази даних наведена на Рис 1.

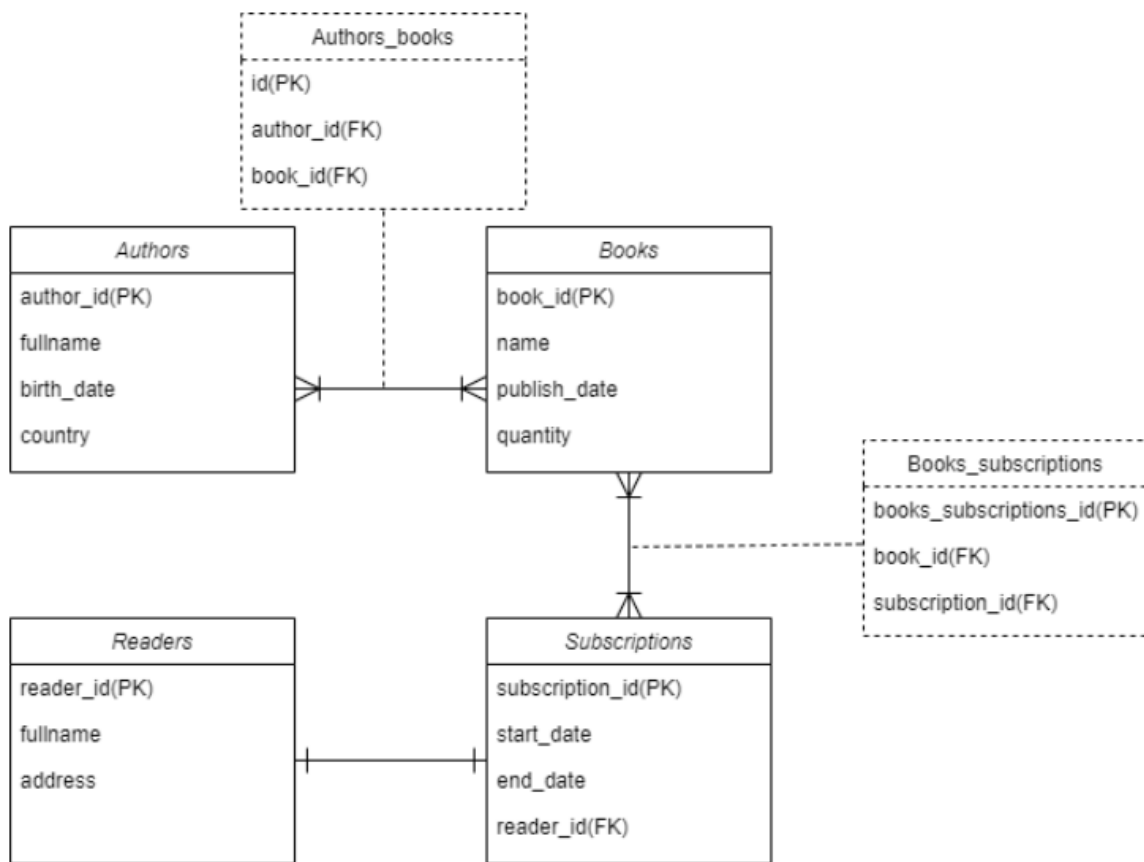


Рис 1. Логічна модель бази даних

Сутнісні класи програми наведені на Рис 2.

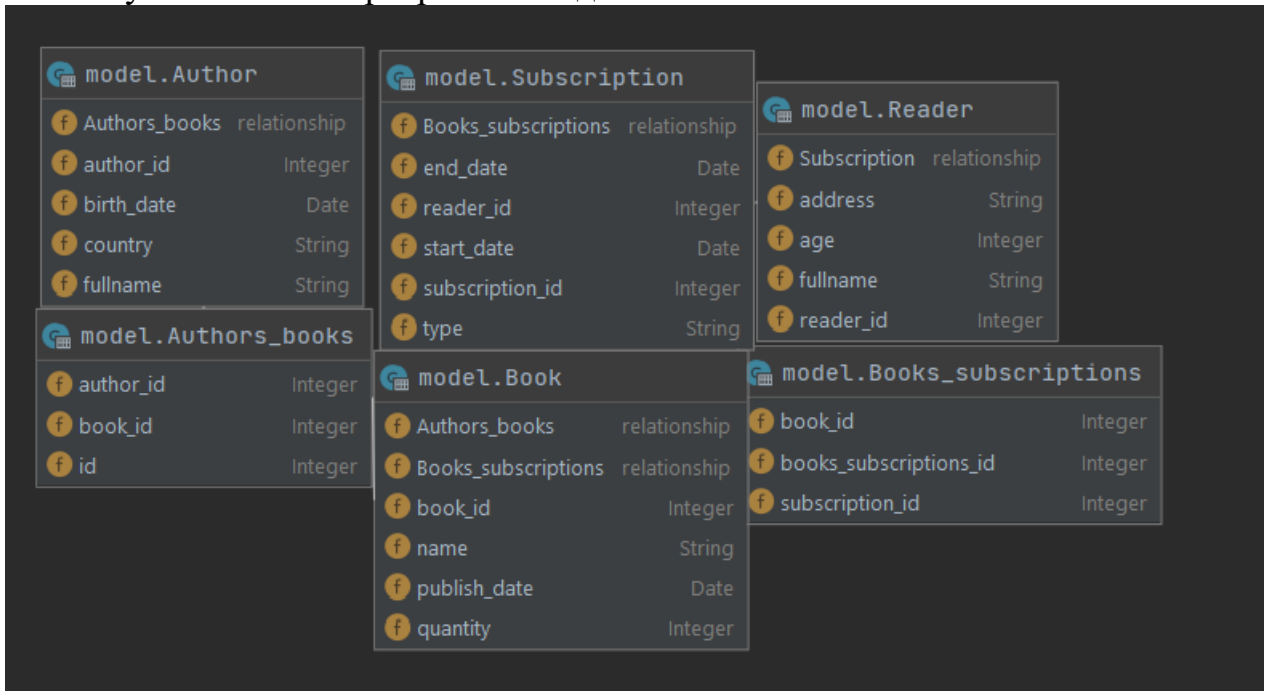


Рис 2. Фрагмент UML діаграми сутнісних класів

Зв'язки між сутнісними класами, згенеровані за допомогою SQLAlchemy, наведені на Рис 3.

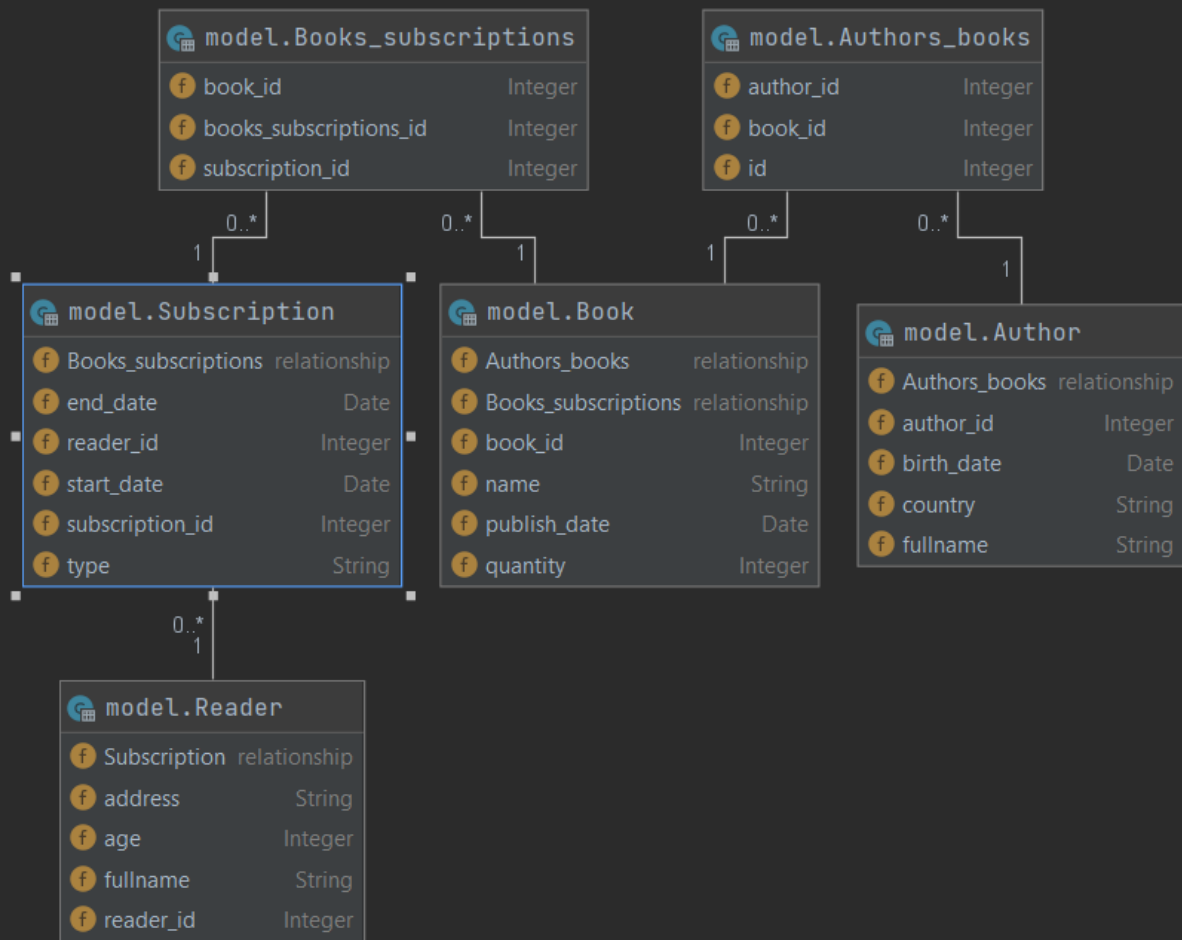


Рис 3. Зв'язки між сутнісними класами

Меню програми наведене на схемі нижче.

```
Select the table to work with | command:

1 - Authors
2 - Books
3 - Authors_books
4 - Readers
5 - Subscriptions
6 - Book_subscription_links
7 - Fill the Readers (random)
8 - Commit
9 - Exit

>> █
```

Пункт 1-6:

```
Table: Readers

1 - INSERT
2 - DELETE
3 - UPDATE
4 - Back

>> █
```

Пункт 7:

```
Successfully generated 1000 random Readers!
```

Пункт 8:

```
Commit success
```

Пункт 9 (вихід з програми).

Лістинг програми

main.py

```
from controller import Controller

if __name__ == '__main__':
    Controller().show_start_menu()
```

controller.py

```
from consolemenu import SelectionMenu

from view import View
from model import Model

TABLES_NAMES = ['Authors', 'Books', 'Authors_books', 'Readers', 'Subscriptions', 'Book_subscription_links']
TABLES = {
    'Authors': ['author_id', 'fullname', 'birth_date', 'country'],
    'Books': ['book_id', 'name', 'publish_date', 'quantity'],
    'Authors_books': ['id', 'author_id', 'book_id'],
    'Readers': ['reader_id', 'fullname', 'address', 'age'],
    'Subscriptions': ['subscription_id', 'start_date', 'end_date', 'reader_id', 'type'],
    'Book_subscription_links': ['id', 'book_id', 'subscription_id']
}

def get_input(msg, table_name=''):
    print(msg)
    if table_name:
        print(' | '.join(TABLES[table_name]), end='\n\n')
    return input()

def get_insert_input(msg, table_name):
    print(msg)
    print(' | '.join(TABLES[table_name]), end='\n\n')
    return input(), input()

def press_enter():
    input()

class Controller:
    def __init__(self):
        self.model = Model()
        self.view = View()

    def show_start_menu(self, msg=''):
        selection_menu = SelectionMenu(
            TABLES_NAMES + ['Fill the Readers (random)', 'Commit'],
            title='Select the table to work with | command:', subtitle=msg)
        selection_menu.show()
```

```

index = selection_menu.selected_option
if index < len(TABLES_NAMES):
    table_name = TABLES_NAMES[index]
    self.show_entity_menu(table_name)
elif index == len(TABLES_NAMES):
    self.random_data_for_readers_table()
elif index == len(TABLES_NAMES) + 1:
    self.model.commit()
    self.show_start_menu(msg='Commit success')
else:
    print(' ')

def show_entity_menu(self, table_name, input=''):
    options = ['INSERT', 'DELETE', 'UPDATE']
    methods = [self.insert, self.delete, self.update]

    selection_menu = SelectionMenu(options,
    f'Table: {table_name}',
    exit_option_text = 'Back',
    subtitle = input)
    selection_menu.show()
    try:
        method = methods[selection_menu.selected_option]
        method(table_name)
    except IndexError:
        self.show_start_menu()

def insert(self, table_name):
    try:
        columns, values = get_insert_input(
            f"INSERT {table_name}\nEnter columns divided with commas, then do the same for va
lues in format: [value1, value2, ...]",
            table_name)
        self.model.insert(table_name, columns, values)
        self.show_entity_menu(table_name, 'Insert is successful!')
    except Exception as err:
        self.show_entity_menu(table_name, str(err))

def delete(self, table_name):
    try:
        condition = get_input(
            f'DELETE {table_name}\nEnter condition (SQL):', table_name)
        self.model.delete(table_name, condition)
        self.show_entity_menu(table_name, 'Delete is successful')
    except Exception as err:
        self.show_entity_menu(table_name, str(err))

def update(self, table_name):
    try:
        condition = get_input(
            f'UPDATE {table_name}\nEnter condition (SQL):', table_name)
        statement = get_input(
            "Enter SQL statement in format [<key>=<value>]", table_name)

```



```

        self.model.update(table_name, condition, statement)
        self.show_entity_menu(table_name, 'Update is successful')
except Exception as err:
    self.show_entity_menu(table_name, str(err))

def random_data_for_readers_table(self):
    try:
        self.model.fill_readers_table_with_random_data()
        self.show_start_menu('Successfully generated 1000 random Readers!')
    except Exception as err:
        self.show_init_menu(str(err))

```

model.py

```

from sqlalchemy import BigInteger, Column, Date, ForeignKey, Integer, String, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, sessionmaker

```

```

engine = create_engine('postgres://postgres:qwerty21qwerty@localhost:5432/Library')
Base = declarative_base()

```

```

class Reader(Base):
    __tablename__ = 'Readers'

    reader_id = Column(Integer, primary_key=True)
    fullname = Column(String)
    address = Column(String)
    age = Column(Integer)

    subscriptions = relationship('Subscription')

    def __init__(self, fullname=None, address=None, age=None):
        self.fullname = fullname
        self.address = address
        self.age = age

```

```

class Author(Base):
    __tablename__ = 'Authors'

    author_id = Column(Integer, primary_key=True)
    fullname = Column(String)
    birth_date = Column(Date)
    country = Column(String)

    authors_books = relationship('Authors_books')

    def __init__(self, fullname=None, birth_date=None, country=None):
        self.fullname = fullname
        self.birth_date = birth_date
        self.country = country

```

```

class Book(Base):

```

```

__tablename__ = 'Books'

book_id = Column(Integer, primary_key=True)
name = Column(String)
publish_date = Column(Date)
quantity = Column(Integer)

authors_books = relationship('Authors_books')
books_subscriptions = relationship('Books_subscriptions')

def __init__(self, name=None, publish_date=None, quantity=None):
    self.name = name
    self.publish_date = publish_date
    self.quantity = quantity

class Subscription(Base):
    __tablename__ = 'Subscriptions'

    subscription_id = Column(Integer, primary_key=True)
    start_date = Column(Date)
    end_date = Column(Date)
    reader_id = Column(Integer, ForeignKey('Readers.reader_id'))
    type = Column(String)

    books_subscriptions = relationship('Books_subscriptions')

    def __init__(self, start_date=None, end_date=None, reader_id=None, type=None):
        self.start_date = start_date
        self.end_date = end_date
        self.reader_id = reader_id
        self.type = type

class Authors_books(Base):
    __tablename__ = 'Authors_books'

    id = Column(Integer, primary_key=True)
    author_id = Column(Integer, ForeignKey('Authors.author_id'))
    book_id = Column(Integer, ForeignKey('Books.book_id'))

    def __init__(self, author_id=None, book_id=None):
        self.author_id = author_id
        self.book_id = book_id

class Books_subscriptions(Base):
    __tablename__ = 'Books_subscriptions'

    books_subscriptions_id = Column(Integer, primary_key=True)
    book_id = Column(Integer, ForeignKey('Books.book_id'))
    subscription_id = Column(Integer, ForeignKey('Subscriptions.subscription_id'))

    def __init__(self, book_id=None, subscription_id=None):
        self.book_id = book_id

```

```

        self.subscription_id = subscription_id

session = sessionmaker(engine)()
Base.metadata.create_all(engine)

TABLES = {'Readers': Reader, 'Authors': Author, 'Books': Book, 'Subscriptions': Subscription, 'Authors_books': Authors_books, 'Books_subscriptions': Books_subscriptions}

class Model:
    def pairs_from_str(self, string):
        lines = string.split(',')
        pairs = {}
        for line in lines:
            key, value = line.split('=')
            pairs[key.strip()] = value.strip()
        return pairs

    def filter_by_pairs(self, objects, pairs, cls):
        for key, value in pairs.items():
            field = getattr(cls, key)
            objects = objects.filter(field == value)
        return objects

    def insert(self, tname, columns, values):
        columns = [c.strip() for c in columns.split(',')]
        values = [v.strip() for v in values.split(',')]

        pairs = dict(zip(columns, values))
        object_class = TABLES[tname]
        obj = object_class(**pairs)

        session.add(obj)

    def commit(self):
        session.commit()

    def delete(self, tname, condition):
        try:
            pairs = self.pairs_from_str(condition)
        except Exception as err:
            raise Exception('Incorrect input')
        object_class = TABLES[tname]

        objects = session.query(object_class)
        objects = self.filter_by_pairs(objects, pairs, object_class)

        objects.delete()

    def update(self, tname, condition, statement):
        try:
            pairs = self.pairs_from_str(condition)
            new_values = self.pairs_from_str(statement)
        except Exception as err:

```

```

        raise Exception('Incorrect input')

    object_class = TABLES[tname]

    objects = session.query(object_class)
    objects = self.filter_by_pairs(objects, pairs, object_class)

    for obj in objects:
        for field_name, value in new_values.items():
            setattr(obj, field_name, value)

def fill_readers_table_with_random_data(self):
    sql = """
        CREATE OR REPLACE FUNCTION randomReaders()
            RETURNS void AS $$
        DECLARE
            step integer := 10;
        BEGIN
            LOOP EXIT WHEN step > 1000;
                INSERT INTO public."Readers" (fullname, address, age)
                VALUES (
                    substring(md5(random()::text), 1, 10),
                    substring(md5(random()::text), 1, 10),
                    (random() * (90 - 1) + 1)::integer
                );
                step := step + 1;
            END LOOP;
        END;
    $$ LANGUAGE PLPGSQL;
    SELECT randomReaders();
    """
    try:
        session.execute(sql)
    finally:
        session.commit()

```

Індекси

GIN індекс:

```
ALTER TABLE public."Readers"  
ADD COLUMN ts_vector tsvector;
```

```
UPDATE public."Readers"  
SET ts_vector = to_tsvector(fullname)  
WHERE true;
```

```
CREATE INDEX ginIndex ON public."Readers" USING gin (ts_vector);
```

Порядок звертання до таблиці без використання фільтру по колонці, на яку додано індекс (створений індекс не використовується):

1	EXPLAIN SELECT * FROM public."Readers"
<div>Data Output Explain Messages Notifications</div>	
	<div>QUERY PLAN</div> <div>text</div>
1	Seq Scan on "Readers" (cost=0.00..80.78 rows=3078 width=79)

Порядок звертання до таблиці з використанням фільтру по колонці, на яку додано індекс (пошук відбувається за допомогою створеного індексу):

1	EXPLAIN SELECT * FROM public."Readers" WHERE to_tsquery('c') @@ ts_vector;
<div>Data Output Explain Messages Notifications</div>	
	<div>QUERY PLAN</div> <div>text</div> <div>1 Bitmap Heap Scan on "Readers" (cost=12.27..22.86 rows=3 width=79)</div> <div>2 Filter: (to_tsquery('c':text) @@ ts_vector)</div> <div>3 -> Bitmap Index Scan on ginindex (cost=0.00..12.27 rows=3 width=0)</div> <div>4 Index Cond: (ts_vector @@ to_tsquery('c':text))</div>

Btree індекс:

```
CREATE INDEX treeIndex ON public."Readers" using btree (reader_id);
```

```
ALTER TABLE public."Readers"  
    ADD COLUMN ts_vector1 tsvector;
```

```
UPDATE public."Readers"  
SET ts_vector1 = to_tsvector(fullname)  
WHERE true;
```

Порядок звертання до таблиці без використання фільтру по колонці, на яку додано індекс (створений індекс не використовується):

```
1 EXPLAIN SELECT * FROM public."Readers"
```

Data Output Explain Messages Notifications

QUERY PLAN	
	text
1	Seq Scan on "Readers" (cost=0.00..80.78 rows=3078 width=79)

Порядок звертання до таблиці з використанням фільтру по колонці, на яку додано індекс (пошук відбувається за допомогою створеного індексу):

```
1 EXPLAIN ANALYSE SELECT * FROM public."Readers" where reader_id < 100100;
```

Data Output Explain Messages Notifications

QUERY PLAN	
	text
1	Bitmap Heap Scan on "Readers" (cost=4.51..51.00 rows=30 width=79) (actual time=0.032..0.044 rows=3 loops=1)
2	Recheck Cond: (reader_id < 100100)
3	Heap Blocks: exact=2
4	-> Bitmap Index Scan on treeindex (cost=0.00..4.50 rows=30 width=0) (actual time=0.022..0.022 rows=3 loops=1)
5	Index Cond: (reader_id < 100100)
6	Planning Time: 1.593 ms
7	Execution Time: 0.126 ms

Тригери

INSERT

Якщо читачу більше 50 років, то до типу його абонементу додається помітка (old), а якщо менше 50, то (young).







Код:

```
CREATE OR REPLACE FUNCTION InsertReader()
    returns trigger
    language plpgsql
AS
$$
BEGIN
    IF NEW.age > 50 THEN
        UPDATE public."Subscriptions" SET type= ' (old)' WHERE "Subscriptions".subscription_id =
NEW.subscription_id;
    ELSE
        UPDATE public."Subscriptions" SET type= ' (young)' WHERE "Subscriptions".subscription_id
= NEW.subscription_id;
    END IF;
    return NEW;
END;
$$;

CREATE TRIGGER setSubscriptionType
    INSERT
    ON public."Readers"
    FOR EACH ROW
EXECUTE PROCEDURE InsertReader();
```

Приклади результатів:

Абонементи до вставки нового читача:

	 subscription_id [PK] integer 	start_date date 	end_date date 	reader_id integer 	type character varying 
1	1	2020-09-25	2020-10-02	1	[null]
2	2	2020-09-26	2020-10-10	2	[null]
3	5	[null]	[null]	100500	(old)
4	6	[null]	[null]	101618	QWERTY
5	9	[null]	[null]	1	default

Додамо нового читача:

```
INSERT INTO public."Readers"(age,subscription_id)
VALUES (24,9);
```

Абонементи після цього:

Data Output		Explain	Messages	Notifications		
	subscription_id [PK] integer		start_date date	end_date date	reader_id integer	type character varying
1	1		2020-09-25	2020-10-02	1	[null]
2	2		2020-09-26	2020-10-10	2	[null]
3	5	[null]		[null]	100500	(old)
4	6	[null]		[null]	101618	QWERTY
5	9	[null]		[null]	1	(young)

AFTER UPDATE

Якщо оновлюється тип абонементу, то до повного імені читача додається новий тип абонементу.

Код:

```
CREATE OR REPLACE FUNCTION afterUpdateSubscription()
    returns trigger
    language plpgsql
AS
$$
DECLARE
    readers cursor is select *
                        from public."Readers"
                        where subscription_id = NEW.subscription_id;
BEGIN
    FOR _reader IN readers
    LOOP
        UPDATE public."Readers"
        SET fullname = NEW.type
        WHERE reader_id = _reader.reader_id;
    end loop;
    return NEW;
END ;
$$;

CREATE TRIGGER insertType
    AFTER UPDATE
    ON public."Subscriptions"
    FOR EACH ROW
EXECUTE PROCEDURE afterUpdateSubscription();
```


Приклади результатів:

UPDATE public."Subscriptions" SET type='BICBMCBC' WHERE id=6;

До оновлення :

2085	102207	03b114c390	5a536c05e7	38	'03b114c390':1	'03b114c390':1	[null]
2086	102216	[null]	[null]	68	[null]	[null]	[null]
2087	102217	QWERTY	[null]	68	[null]	[null]	6

Після оновлення :

1	102217	BICBMCBC	[null]	68	[null]	[null]	6
---	--------	----------	--------	----	--------	--------	---