

**Лабораторные работы по курсу  
Базы данных**

**Лабораторная работа 5  
«Команды модификации базы данных, транзакции»**

**Москва, 2023**

## Оглавление

1. Теоретическая часть .....	3
1.1. Добавление данных в таблицу.....	3
1.2. Изменение значений .....	4
1.3. Удаление значений .....	4
1.4. Индексы .....	4
1.5. Транзакции .....	5
1.6. Представления.....	7
2. Практическая часть .....	8
2.1. Задание 1.....	8
2.2. Задание 2.....	8
2.3. Задание 3.....	8
2.4. Задание 4.....	8
Список литературы.....	8

## 1. Теоретическая часть

В предыдущей лабораторной работе рассматривались вопросы, связанные с проектированием базы данных. В данной лабораторной работе предложены темы, связанные с изменением данных в таблицах – вставкой, обновлением и удалением. В конце работы рассмотрено понятие транзакции в базе данных.

### 1.1. Добавление данных в таблицу

Для добавления данных в таблицу существует оператор SQL INSERT. Его сокращенный синтаксис представлен ниже.

```
INSERT INTO имя_таблицы [ AS псевдоним ] [ ( имя_столбца [, ...] ) ]  
    { DEFAULT VALUES | VALUES ( { выражение | DEFAULT } [, ...] ) [, ...] |  
запрос }
```

Данный оператор добавляет строки в таблицу. С помощью INSERT возможно добавить одну или несколько указанных строк, либо ноль или более строк, возвращенных с помощью дополнительного запроса. После ключевых слов INSERT INTO и названия таблицы, в которую будет производиться добавление данных возможно в скобках указать порядок и названия атрибутов для добавления. В случае отсутствия подобного списка, порядок значений данных должен точно соответствовать порядку столбцов в таблице. Возможно не указывать часть названий столбцов – тогда их значения будут автоматически заполнены значением NULL или значением, заданным по умолчанию при создании таблицы.

Например, для заполнения таблицы «Группа» возможно воспользоваться следующим запросом.

```
INSERT INTO "Группа" ("Номер группы", "Форма обучения", "Номер структурного  
подразделения") VALUES  
( 'ИВТ-41', 'Очная', 1 ),  
( 'ИВТ-42', 'Очная', 1 ),  
( 'ИВТ-43', 'Очная', 1 ),  
( 'ИВТ-21В', 'Заочная', 1 ),  
( 'ИБ-21', 'Очная', 3 ),  
( 'ИТД-31', 'Очная', 4 ),  
( 'ИТД-32', 'Очная', 4 ),  
( 'ИТД-33', 'Очная', 4 );
```

В качестве строк для добавления в таблицу могут быть значения, сформированные в результате запроса к другой таблице. Для иллюстрации приведем следующий пример. Создадим ещё одну таблицу, которая будет содержать информацию о должниках по предметам (т.е. имеющих хотя бы одну оценку 2).

Для этого создадим еще одну таблицу

```
CREATE TABLE "Должники"  
(  
    ID SERIAL PRIMARY KEY,  
    "Фамилия" VARCHAR(30) NOT NULL,  
    "Имя" VARCHAR(30) NOT NULL,  
    "Отчество" VARCHAR(30) NULL,  
    "Номер группы" VARCHAR(7) NOT NULL,  
    "Число долгов" INTEGER NOT NULL  
)
```

Данная таблица будет хранить информацию о ФИО и группе студента, а также о числе его долгов.

Для заполнения таблицы составим SQL запрос.

```
INSERT INTO "Должники" ("Фамилия", "Имя", "Отчество", "Номер группы", "Число  
долгов")  
(
```

```

SELECT "Фамилия", "Имя", "Отчество", "Номер группы", COUNT(*) AS
"Число долгов"
FROM "Студент"
INNER JOIN "Результат освоения дисциплины" ON "Результат освоения
дисциплины"."Студент" = "Студент"."Номер студенческого билета"
WHERE "Результат освоения дисциплины"."Оценка" = 2
GROUP BY "Фамилия", "Имя", "Отчество", "Номер группы"
);

```

Данный запрос выбирает всех студентов, которые имеют хотя бы одну оценку 2 и записывает данные значения в созданную таблицу с должниками.

## 1.2. Изменение значений

Для изменения существующих значений в базе данных существует команда UPDATE.

```

UPDATE имя_таблицы [ * ] [ [ AS ] псевдоним ]
SET { имя_столбца = { выражение | DEFAULT } |
    ( имя_столбца [, ...] ) = ( { выражение | DEFAULT } [, ...] ) |
    ( имя_столбца [, ...] ) = ( вложенный_SELECT )
} [, ...]

```

После оператора UPDATE указывается целевая таблица, которая должна быть модифицирована. В предложении SET указывается, какие столбцы в выбранных строках таблицы должны быть обновлены, и для них задаются новые значения. Остальные столбцы сохраняют свои предыдущие значения. С помощью ключевого слова WHERE возможно отобрать строки таблицы, подлежащие обновлению. Если предложение WHERE отсутствует в записи оператора UPDATE, обновляются все строки целевой таблицы.

Например, составим запрос, убирающий один долг у всех студентов групп ИВТ.

```

UPDATE "Должники"
SET "Число долгов" = "Число долгов" - 1
WHERE "Должники"."Номер группы" LIKE 'ИВТ%'

```

## 1.3. Удаление значений

Для удаления значений из базы данных существует команда DELETE.

```

DELETE FROM имя_таблицы [ * ] [ [ AS ] псевдоним ]
[ WHERE условие ]

```

Команда DELETE удаляет из указанной таблицы строки, удовлетворяющие условию WHERE. Если предложение WHERE отсутствует, она удаляет из таблицы все строки.

Составим запрос на удаление из списка должников всех студентов, не имеющих задолженностей (число долгов = 0).

```

DELETE FROM "Должники"
WHERE "Число долгов" = 0

```

В случае отсутствия оператора WHERE из таблицы будут удалены все значения. Аналогичного результата возможно добиться, используя команду TRUNCATE.

```

DELETE FROM "Должники"
TRUNCATE "Должники"

```

## 1.4. Индексы

При работе с базами данных очень часто необходимо выполнять задачи, связанные с поиском строк в таблицах. Для ускорения подобных запросов используются индексы.

Индекс – специальная структура данных, которая связана с таблицей и создается на основе данных, содержащихся в ней. Основная цель создания индексов – повышение производительности функционирования базы данных. [1]

В общем случае индексы представляют собой дополнительную структуру, содержащую значение индексируемого атрибута и указатель на данный элемент. Все

записи в индексе являются упорядоченными, поэтому поиск данных значительно ускоряется.

В качестве примера можно привести алфавитный указатель, расположенный в конце книги. Обычно, указатель упорядочен по алфавиту и помимо списка определений содержит страницы, на которой располагается о них информация. Когда мы хотим найти значение некоторого слова, то обращаемся к указателю, быстро находим требуемое значение и переходим на страницу, указанную рядом с ним. Таким образом, отпадает необходимость последовательного просмотра всех страниц в книге.

В данном примере книга – это база данных, индекс – алфавитный указатель.

На практике, индексы применяются, когда объемы базы данных существенно велики. Недостатком применения индексов является то, что они занимают отдельное место на диске и требуют накладных расходов для поддержания их в актуальном состоянии при выполнении обновления таблицы.

Создание индекса происходит с помощью следующей команды:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] имя_индекса ] ON
имя_таблицы [ USING метод ]
( { имя_столбца | ( выражение ) } [ COLLATE правило_сортировки ] [
класс_операторов ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
```

Например, создание индекса для атрибута «Номер группы» таблицы Студент происходит следующим образом.

```
CREATE INDEX id_index ON "Студент" ("Номер группы");
```

Приведем пример работы с индексами. Для этого создадим простую таблицу, содержащую три поля – суррогатный ключ и два атрибута, содержащие строки.

```
CREATE TABLE Test(
    id SERIAL PRIMARY KEY,
    CODE_1 VARCHAR(64),
    CODE_2 VARCHAR(64)
);
```


С помощью следующего скрипта заполним нашу таблицу данными.

```
DO
$$
BEGIN
FOR i IN 1..1000000 LOOP
    INSERT INTO Test(code_1,code_2) VALUES (md5(random()::text),
md5(random()::text));
END LOOP;
END
$$ language plpgsql;
```

Обратите внимание, что данный скрипт написан на процедурном языке PL/pgSQL, его изучение выходит за рамки нашего курса. Более подробно можно прочитать в дополнительной литературе. [2]

С помощью функций EXPLAIN ANALYZE в PostgreSQL возможно производить анализ запросов и вычислять затрачиваемое время на его выполнение. Например,

```
EXPLAIN ANALYZE SELECT * FROM "Трудоустройство";
```

	QUERY PLAN	
	text	
1	Seq Scan on "Трудоустройство" (cost=0.00..22.70 rows=1270 width=36) (actual time=...	
2	Planning Time: 3.697 ms	
3	Execution Time: 1.030 ms	

В данном случае запрос выполнялся 1,03 мс.

## 1.5. Транзакции

При работе с базами данных, часто требуется выполнять несколько запросов последовательно. Например при добавлении информации о студенте в учебную базу

данных необходимо произвести две записи с помощью запроса INSERT – на вставку информации в таблицу «Студент» и «Студенческий билет». В случае, если произойдет сбой при выполнении одного из запросов, а второй выполнится, то в базе данных будет храниться лишь частичная информация о студенте. Для борьбы с подобными ошибками используются **транзакции**.

Транзакции – совокупность операций над базой данных, которые вместе образуют логически целостную процедуру, и могут быть либо выполнены все вместе, либо не будет выполнена ни одна из них. [1]

Еще одно применение транзакций – параллельная работа с базой данных нескольких пользователей. Если они вместе одновременно попытаются изменить значение одного из значений, то будет неизвестно, какое из них в итоге сохранится. В Postgres транзакция определяется набором SQL-команд, окружённым командами BEGIN и COMMIT.

```
BEGIN;  
-- ...  
COMMIT;
```

Если в процессе выполнения транзакции мы решим, что не хотим фиксировать её изменения, то возможно выполнить команду ROLLBACK вместо COMMIT, чтобы все наши изменения были отменены.

Рассмотрим следующий пример:

Запустим на выполнение транзакцию и выведем содержимое атрибута «Ставка» из таблицы «Трудоустройство2 на экран.

```
BEGIN  
SELECT "Ставка" FROM "Трудоустройство";
```

	Ставка numeric (3,2)
1	0.25
2	0.25
3	0.25
4	0.50
5	0.60
6	0.40

Обновим значение ставки и выведем новые значения на экран.

```
UPDATE "Трудоустройство"  
SET "Ставка" = "Ставка" * 1.10;  
SELECT "Ставка" FROM "Трудоустройство";
```

	Ставка numeric (3,2)
1	0.28
2	0.28
3	0.28
4	0.55
5	0.66
6	0.44

Откроем еще один экземпляр Query tool и выполним из под нового клиента запрос на вывод ставок.

```
SELECT Ставка FROM "Трудоустройство";
```

	Ставка numeric (3,2)
1	0.25
2	0.25
3	0.25
4	0.50
5	0.60
6	0.40

Обратите внимание, что т.к. транзакция не была завершена, то значения ставок еще не изменились. Для её завершения выполним команду COMMIT

COMMIT

Повторим запрос со второго клиента, и убедимся, что число ставок обновилось.

	Ставка numeric (3,2) 🔒
1	0.28
2	0.28
3	0.28
4	0.55
5	0.66
6	0.44

Вернем обратно значения ставок. Для этого выполним запрос:

```
UPDATE "Трудоустройство"  
SET "Ставка" = "Ставка" / 1.10;
```

	Ставка numeric (3,2) 🔒
1	0.25
2	0.25
3	0.25
4	0.50
5	0.60
6	0.40

Повторим предыдущую транзакцию, только вместе команды COMMIT выполним команду ROLLBACK.

```
BEGIN;  
UPDATE "Трудоустройство"  
SET "Ставка" = "Ставка" * 1.10;  
ROLLBACK;
```

Убедимся в том, что значения не были изменены в результате транзакции.

## 1.6. Представления

При работе с базами данных очень часто приходится выполнять одинаковые сложные и объемные запросы. Для упрощения работы возможно сформировать из такого запроса представление, к которому далее возможно обращаться, как будто это обычная таблица.

Например, создадим представление на основе запроса, выводящего трудоемкость дисциплины в формате «Количество часов/ЗЕТ»

```
CREATE VIEW "Трудоемкость дисциплин" AS  
SELECT "Название дисциплины", (36*"ЗЕТ"::numeric)::varchar || '/' || "ЗЕТ" as  
"Трудоемкость"  
FROM Дисциплина  
ORDER BY "Трудоемкость"
```

Далее к созданному представлению возможно обратиться, как к таблице

```
SELECT * FROM "Трудоемкость дисциплин"
```

Название дисциплины character varying (40) 🔒	Трудоемкость text 🔒
Основы информационной безопасности	108/3
Философия	108/3
Базы данных	144/4
Операционные системы	144/4
Сети и телекоммуникации	144/4
Дизайн цифрового контента	180/5

## 2. Практическая часть

### 2.1.Задание 1.

Проведите следующий эксперимент:

Добавьте в таблицу *Test* из пункта 1.4 одно значение, измерив время данной операции. Далее измерьте время выполнения запроса, выводящего содержимого таблицы в отсортированном виде по столбцу CODE\_1.

Добавьте индекс на столбец CODE\_1. Повторите предыдущие две операции. Сравните полученное время. Во сколько раз оно изменилось? Результаты вычисления занесите в таблицу.

### 2.2. Задание 2.

Добавьте в учебную базу данных информацию о 5 студентах, восстановившихся в **разные** группы.

### 2.3.Задание 3.

Разработайте 3 **осмысленных** запроса на обновление и 3 **осмысленных** запроса на удаление строк в базе данных.

### 2.4. Задание 4.

Разработайте 5 **осмысленных** представлений к учебной базе данных. Возможно использовать запросы из предыдущих лабораторных работ.

## Список литературы

- [1] Е. П. Моргунов, PostgreSQL. Основы языка SQL, 1-е ред., Санкт-Петербург: БХВ-Петербург, 2018, р. 336.
- [2] «PL/pgSQL — процедурный язык SQL,» [В Интернете]. Available: <https://postgrespro.ru/docs/postgresql/15/plpgsql>. [Дата обращения: 09 03 2023].
- [3] «Исходный код СУБД postgres,» [В Интернете]. Available: <https://github.com/postgres/postgres>. [Дата обращения: 30 01 2023].
- [4] Документация к PostgreSQL 15.1, 2022.
- [5] Е. Рогов, PostgreSQL изнутри, 1-е ред., Москва: ДМК Пресс, 2023, р. 662.
- [6] Б. А. Новиков, Е. А. Горшкова и Н. Г. Графеева, Основы технологии баз данных, 2-е ред., Москва: ДМК пресс, 2020, р. 582.