ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»**

**ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК И ТЕХНОЛОГИЙ**

**ВЫСШАЯ ШКОЛА ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ**

**И СУПЕРКОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ**

# Отчет о прохождении технологической практики на тему: «Экспериментальное исследование возможностей языка PL/pgSQL для решения практических задач»

Бондарева Ольга Сергеевна, гр. 3530203/80101

**Направление подготовки:** 02.03.03 Математическое обеспечение и администрирование информационных систем.

**Место прохождения практики:** СПбПУ, ИКНТ, ВШИСиСТ.

**Сроки практики:** с 05.06.2021 по 03.07.2021.

**Руководитель практики от ФГАОУ ВО «СПбПУ»:** Сабинин Олег Юрьевич, к. т. н., доцент ВШИСиСТ.

**Консультант практики от ФГАОУ ВО «СПбПУ»:** Сабинин Олег Юрьевич, к. т. н., доцент ВШИСиСТ.

# Оценка:

Руководитель практики

от ФГАОУ ВО «СПбПУ» О.Ю. Сабинин

Консультант практики

от ФГАОУ ВО «СПбПУ» О.Ю. Сабинин

Обучающийся О.С. Бондарева

Дата:

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 2](#_Toc75904023)

[ВВЕДЕНИЕ 3](#_Toc75904024)

[ГЛАВА 1. Ознакомление с предметом практической работы: среда разработки DataGrip, DML, DDL команды на языке PostgreSQL и возможности Pl/pgSQL 5](#_Toc75904025)

[1.1. Обзор среды разработки DataGrip 5](#_Toc75904026)

[1.2. Создание базы данных и подключение к DataGrip 6](#_Toc75904027)

[1.3. DML, DDL команды на языке PostgreSQL 8](#_Toc75904028)

[1.4. Синтаксис языка Pl/pgSQL 9](#_Toc75904029)

[1.5. Особенности языка Pl/pgSQL 11](#_Toc75904030)

[Технология создания триггеров 11](#_Toc75904031)

[Работа с динамическим SQL 12](#_Toc75904032)

[Работа с файлами 12](#_Toc75904033)

[Возможности настройки и оптимизации 13](#_Toc75904034)

[Управление расписанием выполнения задач 14](#_Toc75904035)

[ГЛАВА 2. Описание подхода решения задачи 15](#_Toc75904036)

[ГЛАВА 3. Разработка программного обеспечения 17](#_Toc75904037)

[ЗАКЛЮЧЕНИЕ 25](#_Toc75904038)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 27](#_Toc75904039)

[ПРИЛОЖЕНИЕ 28](#_Toc75904040)

# ВВЕДЕНИЕ

В наше время базы данных используются повсеместно: государственные структуры, здравоохранение, бизнес и т.д.. Для работы с большими объектами используют различные системы управления базами данных. Одна из самых популярных и востребованных СУБД на момент написания отчета является PostgreSQL.

PostgreSQL поставляется со многими функциями, направленными на то, чтобы помочь разработчикам создавать приложения, отказоустойчивые среды, а также помогать управлять большим набором данных. Кроме того, PostgreSQL предоставляется бесплатно, что позволяет использовать его в учебных заведениях и малых бизнес – проектах. Также, обладает открытым исходным кодом, что дает возможность разработчиках расширять его возможности в соответствии с современными требованиями к СУБД.

PostgreSQL соответствует стандартам SQL-92, SQL-98, SQL-2000. По состоянию на выпуск версии 13 в сентябре 2020 года, PostgreSQL соответствует по крайней мере 170 из 179 обязательных функций для соответствия ядра SQL:2016 [6].

Согласно результатам автоматизированного исследования различного ПО на предмет ошибок, в исходном коде PostgreSQL было найдено 20 проблемных мест на 775 000 строк исходного кода (в среднем, одна ошибка на 39 000 строк кода). Для сравнения: MySQL — 97 проблем, одна ошибка на 4 000 строк кода; FreeBSD (целиком) — 306 проблем, одна ошибка на 4 000 строк кода; Linux (только ядро) — 950 проблем, одна ошибка на 10 000 строк кода) [6].

В СУБД PostgreSQL используется процедурное расширение языка SQL называемое PL/pgSQL. Этот процедурный язык может выполнять сложные вычисления и может использовать все пользовательские типы, функции и операторы [1].

PostgreSQL и большинство других СУБД используют SQL в качестве языка запросов. SQL хорошо переносим и прост в изучении. Однако каждый оператор SQL выполняется индивидуально на сервере базы данных.

После отправления запроса на сервер, приложение будет ждать пока запрос будет обработан, далее программа получит результат, сделает некоторые вычисления, а затем начнет отправлять следующие запросы на сервер. Всё это требует межпроцессного взаимодействия, а также несёт нагрузку на сеть, если клиент и сервер базы данных расположены на разных компьютерах [1].

PL/pgSQL даёт возможность соединить блок с вычислениями и последовательность запросов внутри сервера базы данных. Благодаря этому мы пользуемся возможностями процедурного языка при значительной экономии накладных расходов на клиент-серверное взаимодействие [6].

**Целью исследования** является исследование возможностей СУБД PostgreSQL и разработка запросов для решения практических задач с использованием языка PL/pgSQL. В соответствии с целью задачи, планируется решить следующие **задачи**:

* + - 1. Изучить синтаксис языка PL/pgSQL.
      2. Исследовать возможности СУБД PostgreSQL.
      3. Сформировать алгоритм решения практических задач.
      4. Разработать код на языке PL/pgSQL для решения практических задач.

Ожидаемые результаты работы заключаются в получении теоретических знаний о возможностях языка PL/pgSQL, применения СУБД PostgreSQL, а также код решения практических задач.

# ГЛАВА 1. Ознакомление с предметом практической работы: среда разработки DataGrip, DML, DDL команды на языке PostgreSQL и возможности Pl/pgSQL

В этой главе говорится о языке PostgreSQL, а также его расширении Pl/pgSQL. В первом параграфе рассматривается инструмент для работы с базами данных JetBrains DataGrip, его возможности и особенности. Второй параграф описывает базовый синтаксис PostgreSQL: написание DML, DDL команд. В третьем параграфе говорится о работе на Pl/pgSQL.

# Обзор среды разработки DataGrip

DataGrip - это база данных IDE от компании JetBrains. Она предназначенная для работы с базами данных MySQL, PostgreSQL, Oracle, SQL Server и т.д..

Скачать DataGrip можно на официальном сайте JetBrains. Это программа имеет 30-дневный бесплатный период, после чего следует оплатить стоимость подписки в размере 200$. Для студентов СПбПУ данный инструмент предоставляется бесплатно. После скачивания, следует установить программу следуя инструкциям установщика. Существует возможность добавить различные плагины для более комфортной работы с IDE. Данная IDE также поддерживает работу с системами контроля версий Git, GitHub SVN, Mercurial, CVS и т.д.

Для работы с объектами базы данных DataGrip предоставляет графический интерфейс. DataGrip получает информацию обо всех важных объектах вашей базы и отображает их в виде дерева (рис. 1.1.). Также существует интерфейс для создания таблиц, столбцов, индексов, ограничений и других объектов. Подобные изменения сопровождаются генерацией соответствующего скрипта: можно сразу выполнить сделанные изменения в базе или скопировать сгенерированный DDL-запрос в редактор и работать уже непосредственно с кодом.

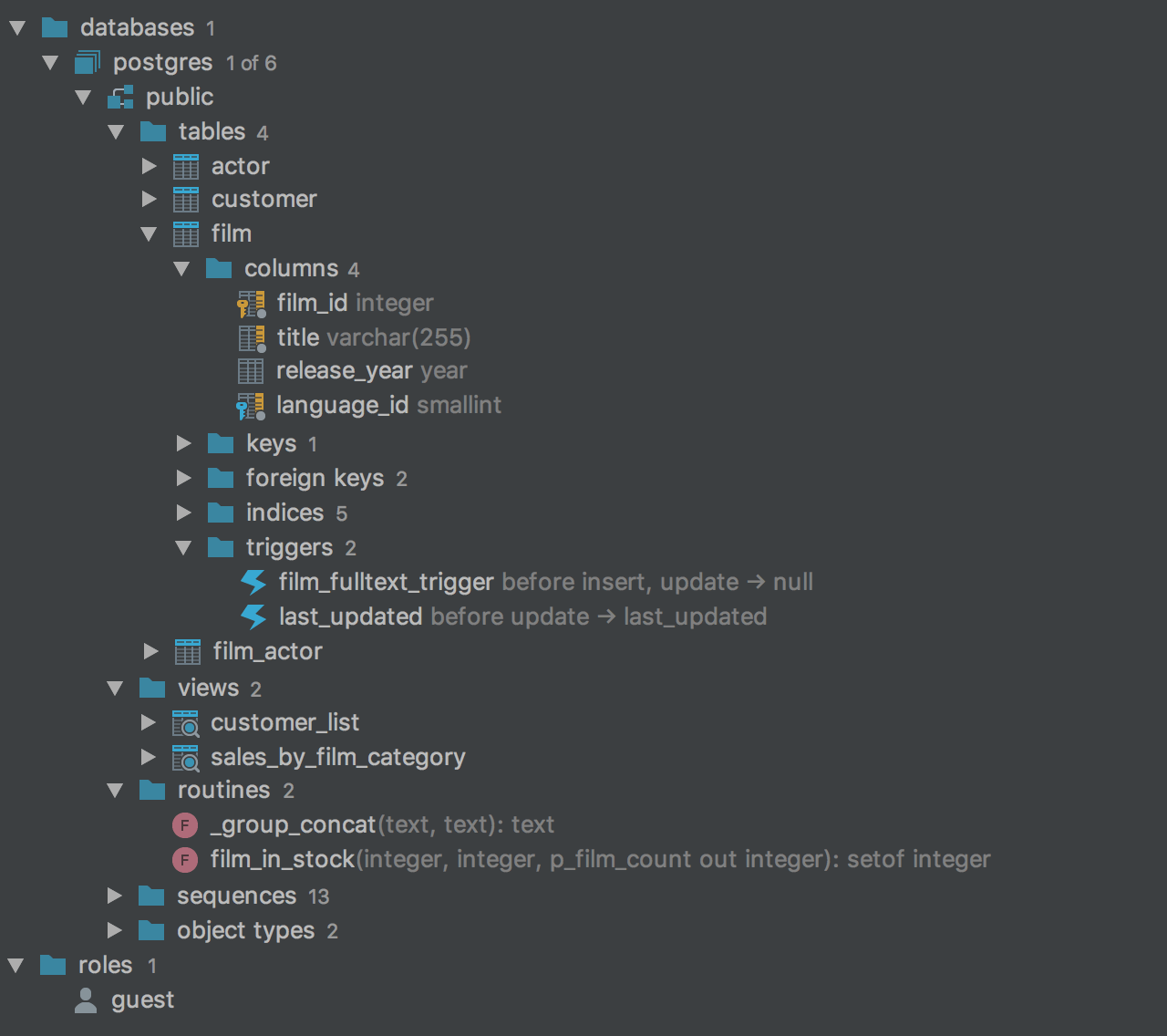


Рис. 1.1 Дерево объектов базы данных

DataGrip поддерживает автодополнение кода, что ускоряет написание запросов. Когда набирается код, IDE понимает контекст и делает работу за пользователя: не только помогает писать код, зная о ключевых словах и именах объектов БД, но и учитывает зависимости при написании JOIN, подсказывает тип параметров для выполнения функции, описывает структуру таблицы в предложениях INSERT [5].

# Создание базы данных и подключение к DataGrip

Перед началом работы с PostreSQL нам необходимо создать базу данных. Для работы я буду использовать терминальный клиент psql, который можно скачать с официального сайта PostreSQL.

После запуска программа попросит ввести данные о подключении, их можно оставить дефолтными. Необходимо ввести имя пользователя (по умолчанию - postgres) и пароль (рис. 1.2).

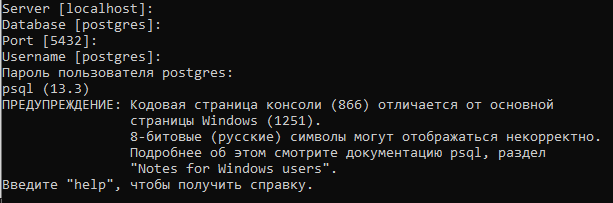


Рис. 2.2 Информация о подключении

Далее создадим базу данным с помощью команды языка SQL: «create database [name]». Присоединиться к базе данных для дальнейшего взаимодействия мы можем при помощи команды \c (рис. 1.3). Удалить базу данных можно при помощи команды «drop database [name]». В данной работе мы больше не будем обращаться к клиенту psql, все взаимодействия с кодом будут происходить в программе DataGrip.

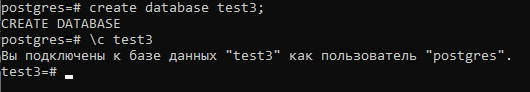


Рис. 3.3 Создание базы данных и подключение к ней при помощи psql

Для установки соединения с нашей базой данных в DataGrip, следует создать новый проект и добавить к нему подключение источника данных PostgreSQL (рис. 1.4).

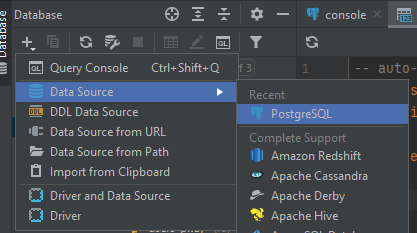


Рис. 4.4 Создание базы данных и подключение к ней при помощи psql

После скачивания необходимых драйверов всплывает окно, куда следует отобразить всю информацию о подключении (Рис. 1.5). Теперь IDE настроена и мы можем приступать к анализу языка PostgreSQL.

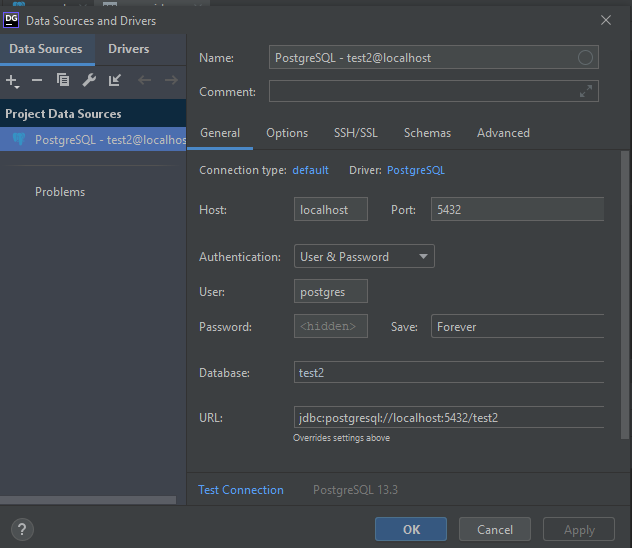


Рис. 5.5 Данные подключения

# DML, DDL команды на языке PostgreSQL

PostgreSQL соответствует большинству стандартов SQL, поэтому создание, изменение таблиц и добавление данных происходит аналогично SQL Oracle. Для примера создадим таблицу Users с полями Id типа Serial, Name типа Varchar:

create table users (  
 Id serial primary key,  
 Name varchar(30));

После изменим таблицу, добавив в нее колонку Age:

ALTER TABLE users ADD COLUMN age INTEGER;

Используя DML-команду INSERT добавим несколько записей в нашу таблицу Users:

INSERT INTO USERS VALUES (100,'King',15);  
INSERT INTO USERS VALUES (101,'Sherlock',20);  
INSERT INTO USERS VALUES (102,'Chamberlin',54);

Для просмотра записей воспользуемся конструкцией SELECT. Результат запроса изображен на рис. 1.6.

select\*from users;

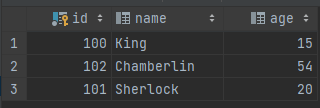


Рис. 6.6 Просмотр таблицы Users

Напишем более сложный запрос используя ключевое слово WHERE. Выведем возраст пользователя с фамилией Sherlock (рис. 1.7.).

select age Возраст from users where name = 'Sherlock'



Рис. 7.7 Возраст человека с именем Sherlock

# Синтаксис языка Pl/pgSQL

Структура Pl/pgSQL кода содержит блок объявления переменных Declare и тело программы, обрамленное ключевыми словами Begin и End.

declare  
--объявление переменных, опционально  
begin  
--код программы, обязательно  
end;

Каждая команда в psql должна заканчиваться точкой с запятой. Два знака доллара нужны для определения начала и конца кода. В общем случае между знаками доллара записывают метку, чтобы различать один блок кода от другого.

Для вывода сообщений используют конструкцию «Raise notice». В одинарных кавычках записывается сообщение, а вместо знака процента подставляются переменные, записанные в том же порядке, что они должны быть в тексте.

Для выполнения анонимного блока используется команда Do. В отличии от процедур и функций, анонимный блок нигде не хранится. В данном примере мы объявляем две переменные. В переменную b записываем квадратный корень из переменной a и выводим результат на экран (рис. 1.8.).

DO $$  
DECLARE  
 a integer := 25;  
 b integer;  
begin  
 b = sqrt(a);  
 raise notice 'The abs of (%) is (%)',a, b;  
end $$;



Рис. 8.8 Результат кода, который считает квадратный корень из 25

Pl/pgSQL даёт возможность создавать пользовательские процедуры и функции. Стоит отметить, что язык процедуры нужно указывать явно, что и делается в последней строке кода. Это связано с тем, что триггеры и хранимые процедуры для PostgreSQL можно писать не только на PL/pgSQL, но и на других языках, в частности, на Perl, Python и JavaScript [3;4].

Приведу пример написания процедуры. В ней мы будет возвращать строки «а!» n-ое количество раз. Для этого мы будем присвоим переменной Word значение «а!», а в Res пока запишем пустую строку. При помощи цикла For мы будем n раз добавлять Word в значение переменной Res. После окончания цикла вернем полученное значение переменной Res.

CREATE OR REPLACE FUNCTION *scream\_function*(len INT) RETURNS TEXT AS $$  
DECLARE  
 word TEXT := 'a!';  
 res TEXT = '';  
BEGIN  
 FOR i IN 1..LEN LOOP  
 res:= res||word;  
 end loop;  
 RETURN res;  
END  
$$ LANGUAGE 'plpgsql';

Для вызова функции создадим анонимный блок. В переменную Result при помощи конструкции «Select Into» запишем вывод функции. Вывод значения будем производить с использованием «Raise Notice». Пример результат программы изображён на рисунке 1.9.

DO $$  
DECLARE  
 result text;  
BEGIN  
 select *scream\_function*(5)  
 INTO result;  
 raise notice 'My feelings now are %', result;  
END $$;

Стоит отметить, что в Pl/pgSQL также существует поддержка пользовательских типов данных, создание триггеров, обработка исключений, работа с динамическим SQL.



Рис. 9.9 Результат кода, который выводит «а!» n раз

# Особенности языка Pl/pgSQL

## Технология создания триггеров

Триггеры – специальные процедуры, которые выполняются если какое-то событие происходит в приложении, например, изменение данных или создание таблиц. В PostgreSQL триггеры создаются на основе существующих функции, т.е. сначала командой «Create function» определяется триггерная функция, затем на ее основе командой «Create trigger» определяется собственно триггер [3]. Функция не должна иметь аргументов, а тип возвращаемого значения должен быть trigger или event\_trigger.

Также в триггерных функциях используются специальные переменные, содержащие информацию о сработавшем триггере. Например, имя NEW типа Record в которой находятся новые значения полей записи базы данных, созданной командой INSERT или обновленной командой UPDATE, при срабатывании триггера уровня записи (ROW).

Рассмотрим пример. Создадим триггер, который при добавлении строки в таблицу USERS будет вызывать ошибку, когда поле Age будет незаполненным. Сначала напишем функцию, которая будет проверять, имеет ли новое значение поля Age значение, и, если нет, вызывать ошибку. После функции создадим триггер, который при каждом добавлении новой строки в таблицу USERS будет вызывать нашу функцию. Проверим наш код, добавим в нашу таблицу пользователя без возраста (рис. 1.10).

-- Создание функции

CREATE FUNCTION *trigger\_after\_ins* () RETURNS trigger AS $$  
BEGIN  
if new.age is null  
then raise exception 'You not filled an Age field yet.';  
end if;  
return new;  
END;$$  
LANGUAGE plpgsql;

-- Создание триггера  
CREATE TRIGGER trigger\_after\_ins  
BEFORE INSERT on users FOR EACH ROW  
EXECUTE PROCEDURE *trigger\_after\_ins*();

-- Добавление пользователя без возраста

INSERT INTO USERS VALUES (103,'Baltemar');



Рис. 10.10 Результат работы триггера

## Работа с динамическим SQL

Обычно SQL операторы известны во время исполнения приложения. В некоторых случаях команды формируются во время исполнения или поступают из внешнего источника. В этих случаях операторы SQL нельзя внедрить непосредственно в исходный код C, но есть средство, позволяющее вызывать произвольные операторы SQL, передаваемые в строковой переменной. Динамический SQL это расширение, позволяющее писать подпрограммы на процедурном языке с использованием команд SQL [6]. PostgreSQL поддерживает несколько процедурных языков как PL/Tcl и PL/Perl.

Одним из способов выполнить произвольный оператор является команда «Execute Immediate». Команду «Execute Immediate» можно использовать для выполнения команд, которые не возвращают результаты, например команды «INSERT, UPDATE, DELETE».

Например добавление новой записи в таблицу users (рис. 1.11).

DO $$  
declare  
begin  
 EXECUTE 'INSERT INTO USERS VALUES (103,''Baltemar'');';  
end  
$$ LANGUAGE 'plpgsql';

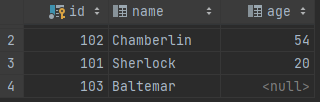


Рис. 11.11 Добавление данных при помощи Execute

Для выполнения оператора с входными параметрами следует подготовить команду при помощи команды «Prepare», где вместо входных данных нужно поставить знаки вопроса.

## Работа с файлами

Для работы с файлами в pl/pgSQL используют команду «Copy». Для копирования данных из таблицы в файл используют «Copy to», а для перемещения значений из файла в таблицу «Copy from». «Copy to» может также скопировать результаты запроса SELECT.

Оператор «Copy» работает с текстовыми форматами, где каждая строка является строкой таблицы. Столбцы в строке разделяются символом-разделителем. Значения самих столбцов — текстовые строки, выдаваемые функцией вывода, либо воспринимаемые функцией ввода, соответствующей типу данных столбца. «Copy from» выдаёт ошибку, если в любой из строк во входном файле оказывается больше или меньше столбцов, чем ожидается. Также у команды «Copy» существует поддержка работы с бинарными файлами и файлами CVS.

Существует мета-команда \copy для psql. Она выполняет операцию «COPY FROM stdin» или «COPY TO stdout», при этом данные хранятся в файле, доступном для psql [3]. Следовательно, права доступа к файлу определяются клиентом, а не серверным процессом postmaster. Например, добавим в файл users.txt данные из таблицы users (рис. 1.12, рис. 1.13).



Рис. 12.12 Результат работы триггера

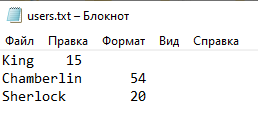


Рис. 13.13 Результат работы триггера

## Возможности настройки и оптимизации

Быстродействие запросов зависит от многих факторов. На некоторые из них могут воздействовать пользователи, а другие являются фундаментальными особенностями системы. Для улучшения производительности используют индексы. Поле индекса может являться выражением, вычисляемым из значений одного или нескольких столбцов в строке таблицы. Это помогает получить быстрый доступ к данным по некоторому преобразованию исходных значений. Например, индекс, построенный по выражению lower(last\_name), позволит использовать поиск по индексу в предложении WHERE lower(last\_name) = 'king'. После создания индекса система будет самостоятельно обновлять его при изменении данных в таблице и использовать его в запросах, где, это будет эффективнее, чем сканирование всей таблицы. Чтобы удалить индекс используют команду «DROP INDEX», добавление и удаление индексов можно производить в любое время.

В Oracle вместо того, чтобы перебирать добавляемые записи в цикле FOR используется заголовок FORALL. В pl/pgSQL используют FOREACH. Обход элементов проводится в том порядке, в котором они сохранялись, независимо от размерности массива. При положительном значении SLICE FOREACH выполняется итерации по срезам массива, а не по отдельным элементам. Для массовой выгрузки данных вместо команды Oracle «Bulk collect» используют array\_agg(). Для загрузки большого количества строк используют команду «Copy», которая является аналогом «Bulk copy».

Также при работе с функциями есть возможность избежать избыточных вызовов функций. Если функция ранее была вызвана с теми же аргументами, оптимизатор может использовать предыдущий результат. Аналогом ключевого слова «Determenistic» от Oracle pl/sql является слово «Immutable».

Кроме того, в функциях используется характеристика «Security invoker», которая показывает, что функция будет выполняться с правами пользователя, вызвавшего её. Этот вариант подразумевается по умолчанию. Эта опция является аналогом команды «Authid current\_user». Вариант «Security Definder» определяет, что функция выполняется с правами пользователя, создавшего её.

## Управление расписанием выполнения задач

Расширение pgpro\_scheduler позволяет планировать и контролировать задания, а также управлять их выполнением в базе данных Postgres. С помощью данного расширения можно выполнять SQL-команды задания в одной или в нескольких последовательных транзакциях, если требуется. Также можно назначать задания для немедленного или отложенного однократного выполнения одновременно с обычными планируемыми заданиями [6].

У pgpro\_scheduler есть ряд преимуществ. Например, то, что для планирования не обязательно перезапускать базу данных. Кроме того, pgpro\_scheduler отличается очень лёгкой реализацией, так как для планирования и контроля заданий, а также для управления ими он использует фоновые рабочие процессы. И при этом pgpro\_scheduler не задействует никакие клиентские подключения.

# ГЛАВА 2. Описание подхода решения задачи

Поставленная задача звучит следующим образом:

Создать пакет, выполняющий операции с многочленами с одной переменной (умножение, деление, сложение/вычитание, упрощение (раскрытие скобок, приведение подобных членов), возведение в степень). Параметры – выражения для многочленов, операция. При унарной операции задается выражение для одного многочлена и операция, при бинарной операции – выражения для двух многочленов и операция. Многочлены могут записываться, например, в виде: 3х+2х^2-5 [2].

Разберем упрощение многочлена. Дадим операторам сложение, вычитание, умножение и деление разные приоритеты выполнения операций. У сложения и вычитания будут приоритет равный 1. У умножения и деления приоритет 2. Также есть особые символы: открывающая и закрывающая скобки. Далее работаем с двумя стеками, в один мы будем складывать наши числа, во второй знаки операций [7].

Мы сравниваем последний элемент в стеке операций с текущим знаком в строке. Если последний элемент в стеке имеет приоритет меньше или равный, текущему в строке, то мы выполняем последнюю операцию в стеке с двумя последними числами в стеке значений. Иначе, если знак операции в стеке имеет меньший приоритет, что и текущий в строке, мы просто кладём знак в стек операций. Закрывающие и открывающие скобки являются специальными символами. Открывающая скобка просто записывается в конец стека операций, а при встрече с закрывающей мы должны выполнить все операции из стека, пока не встретим открывающую. После открывающая и закрывающая скобки удаляются из стека.

Рассмотрим пример: 1+2\*(3+4/2 – (1+2))\*2+1.

Будем изучать каждый символ слева направо. Видим единицу, кладем в стек чисел {1}. Далеe идет знак сложения, кладем его в стек операций {+}. После идет двойка, добавляем ее в стек чисел {1,2}. Далее знак умножения. Мы видим, что в стеке операций уже лежит знак сложения, у которого приоритет меньше, чем у умножения, поэтому просто кладем его сверху {+,\*}. После идет открывающая скобка. Она является особыми символом, как и закрывающая, поэтому просто кладем ее в стек операций {+,\*,(}. Цифру 3 и знак сложения также добавляем в стеки {1,2,3} и {+,\*,(,+}. Число 4 также добавляем в стек значений {1,2,3,4}.

Далее идет знак деления, кладем его в стек {+,\*,(,+,\}. Двойка отправляется в массив значений {1,2,3,4,2}. Следующий знак вычитания имеет приоритет меньше, чем у знака деления. В таком случае мы берем два последних числа из стека значений и делим предпоследний элемент на последний. Результат записываем в стек значений {1,2,3,2}. Последним элементом в стеке операций {+,\*,(,+} становится знак сложения, который имеет такой же приоритет, как вычитание, поэтому мы опять вычисляем значение и записываем его в стек значений {1,2,5}.

Следующий символ — это открывающая скобка, у которой нет приоритета. Поэтому в стек операций мы добавляем наш минус {+,\*,(,-}. Открывающую скобку также записываем в стек {+,\*,(,-,(}. До закрывающей скобки мы просто записываем символы в стек значений {1,2,5,1,2} и в стек операций {+,\*,(,-,(,+}. Следующей встретили закрывающую скобку. Будем выполнять все операции в стеке, пока не найдем закрывающую скобку. В конце после двух закрывающих скобок в стеке значений будет {1,2,2}, а в стеке операций {+,\*}.

Далее идет знак умножения. Так как в стеке уже есть такая операция и она, соответственно, имеет такой же приоритет, выполняем эту операцию с двумя последними значениями из стека чисел. В результате стек чисел равен {1,4}, а в стек операций добавляем умножение, так как сложение имеет меньший приоритет {+,\*}.

Следующий элемент, число 2, записываем в стек значений {1,4,2}. Знак сложения, после двойки, имеет меньший приоритет чем умножение, поэтому вычисляем 4\*2 =8. Записываем число 8 в стек значений {1,8}. Последний элемент в стеке {+} теперь плюс, такой же приоритет, что у плюса в строке, поэтому складываем два последних числа в стеке значений {9}. Добавляем последний символ строки 1 в стек {9,1}.

Строка закончилась, в стеке осталась последняя операция {+}, поэтому просто складываем два числа и получаем результат. Результатом вычислений является число 10.

В задаче в качестве аргументов основной функции do\_operation() может передаваться 2 или 3 аргумента. При передаче трех аргументов, первые два являются многочленами, а третий символом операции: сложение, вычитание, умножение, деление, возведение в степень. При возведении в степень есть проверка на допустимое значение. В зависимости от знака выполняются соответственные операции, которые подробнее расписаны в третьей главе.

Если передаются 2 аргумента, первый является многочленом, а второй строкой, где collect - привести подобные, remove - раскрыть скобки, inverse - инвертировать многочлен. Если передается другое значение, вызывается ошибка «Некорректная операция».

# ГЛАВА 3. Разработка программного обеспечения

В программе будем использовать встроенные массивы Postgres. В них есть возможность обращаться к элементу по индексу, не имеют фиксированной длины, чем они похожи с индексными таблицами в Oracle. Для работы с первым и последним номером используют функции array\_lower() и array\_upper() соответственно.

Postgres не поддерживает работу с пакетами, поэтому вместо создания пакета в задаче создается схема. Для задачи придется создать тип данных, содержащий коэффициент одночлена и его степень.

Перед работой с вводимой строкой мы проводим ее нормализацию, а именно: удаляем лишние пробелы, избавляемся от умножения перед х и добавляем + в начало строки, если там нет знака.

create or replace FUNCTION *norm*(  
 op VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 new\_str VARCHAR(100);  
BEGIN  
 --заменяем все \*х на х, избавляемся от ненужных знаков умножения  
 new\_str := replace(  
 replace(op, ' ', ''), '\*x', 'x');  
 -- если в начале строки нет знака, то число положительное, добавляем +  
 IF (  
 (substr(new\_str,1,1) != '-')  
 AND (substr(new\_str,1,1) != '+')  
 ) THEN  
 new\_str := concat('+',new\_str);  
 END IF;  
 RETURN new\_str;  
END  
$$ LANGUAGE 'plpgsql';

Функция addition() выполняет операцию сложения. Она возвращает другую функцию collectingTerms(), в которую мы передаем наши конкатенированные строки. Функция collectingTerms() вызывает функцию splitToGroups(), в которой наша строка разделяется на одночлены. Если одночлены имеют одинаковую степень, то они складываются. Сложение является такой же операцией как и приведение подобных членов, поэтому при привидении подобных членов мы тоже обращаемся к функции collectingTerms().

Операция деления происходит с использованием функции addition(). Так как сложение является обратной операцией деления, поэтому при вычитании двух многочленов второй аргумент является инвертированным. Это происходит с помощью функции inverse(). Если мы встретили в строке минус, меняем его на плюс, а если встретили плюс, то на минус.

create function *inverse*(op character varying) returns character varying  
 language plpgsql  
as  
$$  
declare  
 rev\_str VARCHAR(64);  
BEGIN  
 FOR i IN 1..coalesce(char\_length(op),0)  
 LOOP  
 DECLARE  
 v\_character VARCHAR(1);  
 BEGIN  
 v\_character := substring(op from i for 1);  
  
 IF (v\_character = '+') THEN  
 rev\_str := concat(rev\_str,'-');  
 ELSIF (v\_character = '-') THEN  
 rev\_str := concat(rev\_str,'+');  
 ELSE  
 rev\_str := concat(rev\_str,v\_character);  
 END IF;  
 END;  
 END LOOP;  
  
 RETURN rev\_str;  
END  
$$;

Умножение происходит с использованием функции getPolynom(), которая преобразовывает одночлен в массив, где индекс массива является степенью одночлена, а значение в этой ячейке коэффициентом одночлена. В функции multiplication() мы принимаем на вход значения для перемножения. Разделяем их на массив одночленов и переводим их в числовой массив с помощью getPolynom(). После мы проходимся по массивам обоих аргументов и, если ячейка массива не пуста, перемножаем один одночлен с другим. Возвращаем строку, в которую записан результат умножения.

create or replace FUNCTION *multiplication*(  
 op\_1 VARCHAR,  
 op\_2 VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 polynom\_1 numeric(6, 2)[];  
 polynom\_2 numeric(6, 2)[];  
 polynom\_res numeric(6, 2)[];  
 tmp1 numeric(6, 2);  
 tmp2 numeric(6, 2);  
BEGIN  
 polynom\_1 := *getPolynom*(*splitToGroups*(op\_1));  
 polynom\_2 := *getPolynom*(*splitToGroups*(op\_2));  
  
  
 FOR i IN array\_lower(polynom\_1, 1)..array\_upper(polynom\_1, 1)  
 LOOP  
 IF (polynom\_1[i] is null) THEN  
 CONTINUE;  
 END IF;  
  
 FOR j IN array\_lower(polynom\_2, 1)..array\_upper(polynom\_2, 1)  
 LOOP  
 DECLARE  
 cur\_degree INTEGER;  
 cur\_coeff numeric(6, 2);  
 BEGIN  
 IF (polynom\_2[j] is null) THEN  
 CONTINUE;  
 END IF; -- аналогично  
  
 cur\_degree := i + j;  
 tmp1 = (select polynom\_1[i]);  
 tmp2 = (select polynom\_2[j]);  
 cur\_coeff := tmp1 \* tmp2;  
  
 if (polynom\_res[cur\_degree] is not null) then  
 polynom\_res[cur\_degree] := polynom\_res[cur\_degree] + cur\_coeff;  
 ELSE  
 polynom\_res[cur\_degree] := cur\_coeff;  
 END IF;  
 END;  
 END LOOP;  
 END LOOP;  
  
 RETURN *buildString*(polynom\_res);  
END  
$$ LANGUAGE 'plpgsql';

Операция деления происходит в функции division(). В функции getNumPolynom() мы преобразовываем целочисленный одночленов в массив типа t\_degree\_and\_coeff. То есть мы получаем массив у которого каждая запись имеет параметры степени и коэффициента. При делении действует следующий алгоритм: сначала мы делим наибольшую степень первого операнда на наибольшую степень второго операнда. После умножаем результат деления на второй операнд, чтобы посмотреть, что останется в качестве остатка результата деления двух операндов. Вычитаем получившиеся значение из первого операнда и получаем результат.

create or replace FUNCTION *division*(  
 op\_1 VARCHAR,  
 op\_2 VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 num\_polynom\_1 type\_degree\_and\_coeff[];  
 num\_polynom\_2 type\_degree\_and\_coeff[];  
 polynom\_1 numeric(6, 2)[];  
 polynom\_2 numeric(6, 2)[];  
 polynom\_res numeric(6, 2)[];  
BEGIN  
 polynom\_1 := *getPolynom*(*splitToGroups*(op\_1));  
 polynom\_2 := *getPolynom*(*splitToGroups*(op\_2));  
  
 num\_polynom\_1 := *getNumPolynom*(polynom\_1);  
 num\_polynom\_2 := *getNumPolynom*(polynom\_2);  
  
 WHILE (array\_upper(polynom\_1, 1) >= array\_upper(polynom\_2, 1))  
 LOOP  
 DECLARE  
 cur\_degree INTEGER;  
 cur\_coeff numeric(6, 2);  
 pol\_tmp numeric(6, 2)[];  
 BEGIN  
 cur\_degree := num\_polynom\_1[1].degree - num\_polynom\_2[1].degree;  
 cur\_coeff := num\_polynom\_1[1].coeff / num\_polynom\_2[1].coeff;  
  
 polynom\_res[cur\_degree] := cur\_coeff;  
 pol\_tmp[cur\_degree] := cur\_coeff;  
  
 polynom\_1 := *getPolynom*(  
 *splitToGroups*(  
 *addition*(  
 *buildString*(polynom\_1),  
 *inverse*(  
 *multiplication*(  
 *buildString*(pol\_tmp),  
 *buildString*(polynom\_2)  
 )  
 )  
 )  
 )  
 );  
 num\_polynom\_1 := *getNumPolynom*(polynom\_1);  
  
 END;  
 END LOOP;  
  
 raise notice '=== remainder of division is: %',*buildString*(polynom\_1);  
  
 RETURN *buildString*(polynom\_res);  
END  
$$ LANGUAGE 'plpgsql';

При возведении в степень используется функция exponentiating(). Сначала присваиваем степени второй аргумент функции, а результату первый. После до умножаем результат на операнд degree-1 раз, тем самым возводя операнд в степень.

create function *exponentiating*(op\_1 character varying, op\_2 character varying) returns character varying  
 language plpgsql  
as  
$$  
declare  
 res VARCHAR(100);  
 degree NUMERIC(6, 2);  
BEGIN  
 degree := to\_number(op\_2, '9999');  
 res := op\_1;  
 FOR i IN 1..degree - 1  
 LOOP  
 res := *multiplication*(res, op\_1);  
 END lOOP;  
  
 RETURN res;

Подробнее алгоритм упрощения описан во второй главе. Здесь мы опишем реализацию на языке pl/pgSQL. Функция f1() добавляет символ в стек. Функция f2() производит операцию с двумя последними значениями из стека чисел и записывает текущий знак в стек операций. Функция f3() удаляет число из стека значений. Функция f4() производит операцию с двумя последними значениями из стека чисел и повторно вызывает функцию упрощения, не записывая текущий знак в стек операций. В зависимости от операции и последнего символа в стеке вызываем определенную функцю. Например, открывающую скобку мы просто записываем в стек в независимости от последнего значения, поэтому вызываем операцию f1.

create or replace PROCEDURE *simplification*(  
 cur\_ch VARCHAR,  
 stack\_op IN OUT varchar(100)[],  
 stack\_func IN OUT varchar(100)[]  
 ) as  
$$  
declare  
 cur\_str VARCHAR(100);  
BEGIN  
 CASE cur\_ch  
 WHEN '('  
 THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+'  
 THEN CASE stack\_func[array\_upper(stack\_func, 1)]  
 WHEN '(' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '-' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '\*' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
  
 WHEN '-'  
 THEN CASE stack\_func[array\_upper(stack\_func, 1)]  
 WHEN '(' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '-' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '\*' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
  
 WHEN '\*'  
 THEN CASE stack\_func[array\_upper(stack\_func, 1)]  
 WHEN '(' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '-' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '\*' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
  
 WHEN '/'  
 THEN CASE stack\_func[array\_upper(stack\_func, 1)]  
 WHEN '(' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '-' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '\*' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
  
 WHEN ')'  
 THEN CASE stack\_func[array\_upper(stack\_func,1)]  
 WHEN '(' THEN call *f3*(stack\_func);  
 WHEN '+' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 WHEN '-' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 WHEN '\*' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
 END CASE;  
END  
$$ LANGUAGE 'plpgsql';

Примеры результатов:

1. Процедура с аргументами: ('0')

Результат:



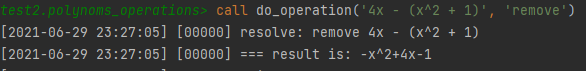
1. Процедура с аргументами: ('0','x')

Результат:



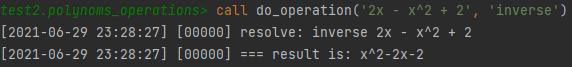
1. Процедура с аргументами: ('4x - (x^2 + 1)', 'remove')

Результат:



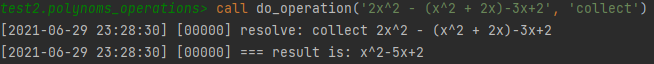
1. Процедура с аргументами: ('2x - x^2 + 2', 'inverse')

Результат:



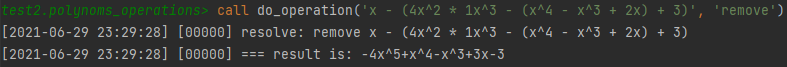
1. Процедура с аргументами: ('2x^2 - (x^2 + 2x)-3x+2', 'collect')

Результат:



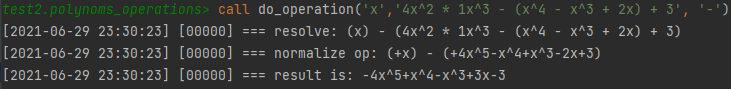
1. Процедура с аргументами: ('x - (4x^2 \* 1x^3 - (x^4 - x^3 + 2x) + 3)', 'remove')

Результат:



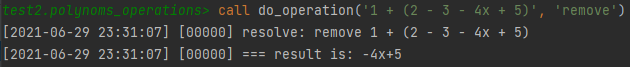
1. Процедура с аргументами: ('x','4x^2 \* 1x^3 - (x^4 - x^3 + 2x) + 3', '-')

Результат:



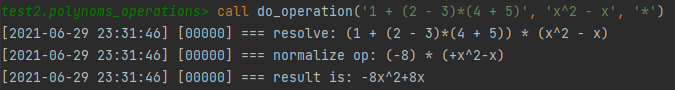
1. Процедура с аргументами: ('1 + (2 - 3 - 4x + 5)', 'remove')

Результат:



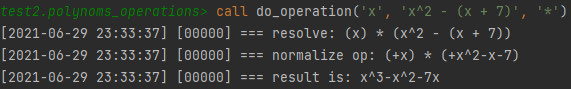
1. Процедура с аргументами: ('1 + (2 - 3)\*(4 + 5)', 'x^2 - x', '\*')

Результат:



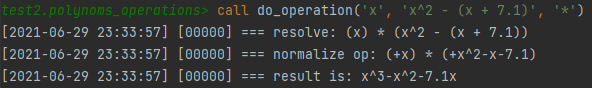
1. Процедура с аргументами: ('x', 'x^2 - (x + 7)', '\*')

Результат:



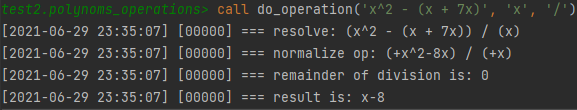
1. Процедура с аргументами: ('x', 'x^2 - (x + 7.1)', '\*')

Результат:



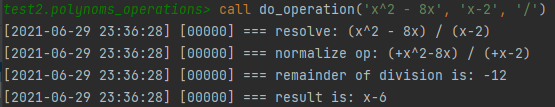
1. Процедура с аргументами: ('x^2 - (x + 7x)', 'x', '/')

Результат:



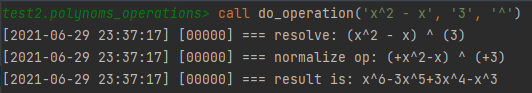
1. Процедура с аргументами: ('x^2 - 8x', 'x-2', '/')

Результат:



1. Процедура с аргументами: ('x^2 - x', '3', '^')

Результат:



1. Процедура с аргументами: ('x^2 - x', '3', '?')

Результат:



# ЗАКЛЮЧЕНИЕ

Таким образом, PostgreSQL — наиболее полнофункциональная, свободно распространяемая СУБД с открытым кодом.

PostgreSQL имеет лицензию, которая разрешает неограниченное использование СУБД, а также модификацию кода. Есть возможность добавлять другие продукты, например коммерческие.

В данной работе мы разработали решение задачи и рассмотрели особенности pl/pgSQL. PostgreSQL является функциональной СУБД с открытым кодом.

PL/pgSQL – это процедурное расширение языка SQL, используемое в СУБД PostgreSQL. Этот процедурный язык может дает возможность оптимизировать вычисления, поддерживает работу с файлами различных форматов. Также существует поддержка динамического SQL и технология создания триггеров.

При использовании PL/pgSQL появляется возможность сгруппировать запросы и вычислительные блоки в единую конструкцию, которая будет размещаться и выполняться на сервере, а клиент будет отправлять запрос на её выполнение и получать результат, минуя все промежуточные пересылки данных, что в большинстве случаев очень позитивно сказывается на производительности.

Так же существует функциональность анонимных блоков, позволяющий писать запросы не на SQL, а прямо на любом существующем процедурном языке сервера, в том числе pl/pgSQL, без создания хранимых функций на сервере СУБД.

# СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Лузанов, П. PostgreSQL для начинающих / П. Лузанов, Е. Рогов, И. Лёвшин. — 2017 : Компания Postgres Professional, 2017.
2. Андреева Н.В. Н.В. ПРОГРАММИРОВАНИЕ БАЗ ДАННЫХ НА PL\SQL ПРАКТИКУМ / Н.В. Андреева Н.В., О.Ю. Сабинин. — Санкт-Петербург: СПбПУ, 2021. — 50 с.
3. PostgreSQL : Документация. [Электронный ресурс]. URL: https://postgrespro.ru/docs/postgresql/ (дата обращения: 20.06.2021)
4. Introduction to PostgreSQL SQL. [Электронный ресурс]. URL https://postgres.cz/wiki/Introduction\_to\_PostgreSQL\_SQL/ (дата обращения: 20.06.2021)
5. PL/pgSQL. [Электронный ресурс]. URL https://ru.wikipedia.org/wiki/PL/pgSQL/ (дата обращения: 21.06.2021)
6. Что умеет DataGrip. [Электронный ресурс]. URL https://www.jetbrains.com/ru-ru/datagrip/features/ (дата обращения: 21.06.2021)
7. PL/pgSQL. [Электронный ресурс]. URL https://ru.wikipedia.org/wiki/PL/pgSQL/ (дата обращения: 21.06.2021)
8. Калькулятор на основе Обратной Польской записи. [Электронный ресурс]. URL: https://habr.com/ru/sandbox/56187/ (дата обращения: 22.06.2021)

# ПРИЛОЖЕНИЕ

create schema IF NOT EXISTS polynoms\_operations;  
SET search\_path = polynoms\_operations;  
  
--тип записи степеней и коэффициентов  
DO $$ BEGIN  
 create type type\_degree\_and\_coeff AS (  
 degree INTEGER,  
 coeff NUMERIC(6, 2)  
 );  
EXCEPTION  
 WHEN duplicate\_object THEN null;  
END $$;  
  
create or replace PROCEDURE *do\_operation*(  
 VARIADIC operands VARCHAR[]  
) AS  
$$  
BEGIN  
 IF (operands[1] is NULL OR operands[2] is NULL) THEN  
 raise exception 'Usage: do\_operation(op1, op2, math\_operation) or do\_operation(op, polynom\_operation)';  
 END IF;  
  
 IF (operands[3] is NULL) THEN  
 call *do\_operation1*( operands[1], operands[2] );  
 ELSE  
 call *do\_operation2*( operands[1], operands[2], operands[3] );  
 END IF;  
END  
$$ LANGUAGE 'plpgsql';  
  
-- Операции с двумя операндами  
create or replace PROCEDURE *do\_operation2*(  
 op\_1 VARCHAR,  
 op\_2 VARCHAR,  
 operation VARCHAR  
) AS  
$$  
declare  
 norm\_op\_1 VARCHAR(100);  
 norm\_op\_2 VARCHAR(100);  
 res VARCHAR(100);  
BEGIN  
 norm\_op\_1 := *norm*(op\_1);  
 norm\_op\_2 := *norm*(op\_2);  
  
 --если есть скобки, избавляемся от них  
 IF (*checkExistsBrackets*(norm\_op\_1)) THEN  
 norm\_op\_1 := *norm*(*removingBrackets*(norm\_op\_1));  
 END IF;  
 IF (*checkExistsBrackets*(norm\_op\_2)) THEN  
 norm\_op\_2 := *norm*(*removingBrackets*(norm\_op\_2));  
 END IF;  
  
 raise notice '=== resolve: (%) % (%)',op\_1,operation,op\_2;  
 raise notice '=== normalize op: (%) % (%)',norm\_op\_1,operation,norm\_op\_2;  
  
--в зависимости от перации выполняем следующие функции, иначе операция некорректна  
 CASE operation  
 WHEN '+' THEN res := *addition*(norm\_op\_1, norm\_op\_2);  
 WHEN '-' THEN res := *addition*(norm\_op\_1, *inverse*(norm\_op\_2));  
 WHEN '\*' THEN res := *multiplication*(norm\_op\_1, norm\_op\_2);  
 WHEN '/' THEN res := *division*(norm\_op\_1, norm\_op\_2);  
 WHEN '^' THEN res := *exponentiating*(norm\_op\_1, norm\_op\_2);  
 ELSE RAISE exception 'Некорректная операция! Допустимы +,-,\*,/,^';  
 END CASE;  
  
 raise notice '=== result is: %', trim(leading '+' from res);  
 /\* EXCEPTION  
 WHEN OTHERS  
 THEN RAISE exception 'Некорректные операнды!';\*/  
END  
$$ LANGUAGE 'plpgsql';  
  
-- Операции над одним многочленом  
create or replace PROCEDURE *do\_operation1*(  
 op VARCHAR,  
 operation VARCHAR  
) AS  
$$  
declare  
 norm\_op VARCHAR(100);  
 res VARCHAR(100);  
BEGIN  
 norm\_op := *norm*(op);  
 IF (*checkExistsBrackets*(norm\_op)) THEN  
 norm\_op := *norm*(*removingBrackets*(norm\_op));  
 END IF;  
 raise notice 'resolve: % %',operation,op;  
  
 -- collect - привести подобные  
 -- remove - раскрыть скобки  
 -- inverse - инверсировать многочлен  
 CASE operation  
 WHEN 'collect' THEN res := *collectingTerms*(norm\_op);  
 WHEN 'remove' THEN res := *removingBrackets*(norm\_op);  
 WHEN 'inverse' THEN res := *collectingTerms*(*inverse*(norm\_op));  
 ELSE RAISE exception 'Некорректная операция! Допустимые операции: collect, remove, inverse';  
 END CASE;  
  
 raise notice '=== result is: %',trim(leading '+' from res);  
 /\* EXCEPTION  
 WHEN OTHERS  
 THEN raise exception 'Некорректные операнды!'; \*/  
END  
$$ LANGUAGE 'plpgsql';  
--проверка скобок, если существует открытая или закрытая скобка, возвращаем верно  
create or replace FUNCTION *checkExistsBrackets*(  
 op VARCHAR  
) RETURNS BOOLEAN AS  
$$  
BEGIN  
 FOR i IN 1..coalesce(length(op),0)  
 LOOP  
 IF (  
 (substr(op,i,1) = '(')  
 OR (substr(op,i,1) = ')')  
 ) THEN  
 RETURN TRUE;  
 END IF;  
 END LOOP;  
  
 RETURN FALSE;  
END  
$$ LANGUAGE 'plpgsql';  
  
--фукнция нормализации операции  
create or replace FUNCTION *norm*(  
 op VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 new\_str VARCHAR(100);  
BEGIN  
 --заменяем все \*х на х, избавляемся от ненужных знаков умножения  
 new\_str := replace(  
 replace(op, ' ', ''), '\*x', 'x');  
 -- если в начале строки нет знака, то число положительное, добавляем +  
 IF (  
 (substr(new\_str,1,1) != '-')  
 AND (substr(new\_str,1,1) != '+')  
 ) THEN  
 new\_str := concat('+',new\_str);  
 END IF;  
 RETURN new\_str;  
END  
$$ LANGUAGE 'plpgsql';  
  
create or replace FUNCTION *addition*(  
 op\_1 VARCHAR,  
 op\_2 VARCHAR  
) RETURNS VARCHAR AS  
$$  
BEGIN  
 RETURN *collectingTerms*(concat(op\_1,op\_2));  
END  
$$ LANGUAGE 'plpgsql';  
  
create or replace FUNCTION *multiplication*(  
 op\_1 VARCHAR,  
 op\_2 VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 polynom\_1 numeric(6, 2)[];  
 polynom\_2 numeric(6, 2)[];  
 polynom\_res numeric(6, 2)[];  
 tmp1 numeric(6, 2);  
 tmp2 numeric(6, 2);  
BEGIN  
 polynom\_1 := *getPolynom*(*splitToGroups*(op\_1));  
 polynom\_2 := *getPolynom*(*splitToGroups*(op\_2));  
  
  
 FOR i IN array\_lower(polynom\_1, 1)..array\_upper(polynom\_1, 1)  
 LOOP  
 IF (polynom\_1[i] is null) THEN  
 CONTINUE;  
 END IF;  
  
 FOR j IN array\_lower(polynom\_2, 1)..array\_upper(polynom\_2, 1)  
 LOOP  
 DECLARE  
 cur\_degree INTEGER;  
 cur\_coeff numeric(6, 2);  
 BEGIN  
 IF (polynom\_2[j] is null) THEN  
 CONTINUE;  
 END IF; -- аналогично  
  
 cur\_degree := i + j;  
 tmp1 = (select polynom\_1[i]);  
 tmp2 = (select polynom\_2[j]);  
 cur\_coeff := tmp1 \* tmp2;  
  
 if (polynom\_res[cur\_degree] is not null) then  
 polynom\_res[cur\_degree] := polynom\_res[cur\_degree] + cur\_coeff;  
 ELSE  
 polynom\_res[cur\_degree] := cur\_coeff;  
 END IF;  
 END;  
 END LOOP;  
 END LOOP;  
  
 RETURN *buildString*(polynom\_res);  
END  
$$ LANGUAGE 'plpgsql';  
  
create or replace FUNCTION *division*(  
 op\_1 VARCHAR,  
 op\_2 VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 num\_polynom\_1 type\_degree\_and\_coeff[];  
 num\_polynom\_2 type\_degree\_and\_coeff[];  
 polynom\_1 numeric(6, 2)[];  
 polynom\_2 numeric(6, 2)[];  
 polynom\_res numeric(6, 2)[];  
BEGIN  
 polynom\_1 := *getPolynom*(*splitToGroups*(op\_1));  
 polynom\_2 := *getPolynom*(*splitToGroups*(op\_2));  
  
 num\_polynom\_1 := *getNumPolynom*(polynom\_1);  
 num\_polynom\_2 := *getNumPolynom*(polynom\_2);  
  
 WHILE (array\_upper(polynom\_1, 1) >= array\_upper(polynom\_2, 1))  
 LOOP  
 DECLARE  
 cur\_degree INTEGER;  
 cur\_coeff numeric(6, 2);  
 pol\_tmp numeric(6, 2)[];  
 BEGIN  
 cur\_degree := num\_polynom\_1[1].degree - num\_polynom\_2[1].degree;  
 cur\_coeff := num\_polynom\_1[1].coeff / num\_polynom\_2[1].coeff;  
  
 polynom\_res[cur\_degree] := cur\_coeff;  
 pol\_tmp[cur\_degree] := cur\_coeff;  
  
 polynom\_1 := *getPolynom*(  
 *splitToGroups*(  
 *addition*(  
 *buildString*(polynom\_1),  
 *inverse*(  
 *multiplication*(  
 *buildString*(pol\_tmp),  
 *buildString*(polynom\_2)  
 )  
 )  
 )  
 )  
 );  
 num\_polynom\_1 := *getNumPolynom*(polynom\_1);  
  
 END;  
 END LOOP;  
  
 raise notice '=== remainder of division is: %',*buildString*(polynom\_1);  
  
 RETURN *buildString*(polynom\_res);  
END  
$$ LANGUAGE 'plpgsql';  
  
--получение многочлена из таблицы одночленов  
create or replace FUNCTION *getPolynom*(  
 monomials VARCHAR(100)[]  
) RETURNS numeric(6, 2)[] AS  
$$  
declare  
 polynom numeric(6, 2)[];  
BEGIN  
 for i in array\_lower(monomials, 1)..array\_upper(monomials, 1)  
 loop  
 DECLARE  
 degree\_and\_coeff type\_degree\_and\_coeff;  
 BEGIN  
 --получаем степень и коэффециент одночлена  
 degree\_and\_coeff := *getDegreeAndCoeff*(monomials[i]);  
 --складываем коэффициенты, если есть несколько одночленов с одинаковыми степенями  
 polynom[degree\_and\_coeff.degree] := coalesce(polynom[degree\_and\_coeff.degree],0) + degree\_and\_coeff.coeff;  
 END;  
 END LOOP;  
 RETURN polynom;  
END  
$$ LANGUAGE 'plpgsql';  
  
--запись степени и коэффициента полинома в таблицу num\_polynom  
create or replace FUNCTION *getNumPolynom*(  
 polynom numeric(6, 2)[]  
) RETURNS type\_degree\_and\_coeff[] AS  
$$  
declare  
 num\_polynom type\_degree\_and\_coeff[];  
 cnt INTEGER := 1;  
 degree\_and\_coeff type\_degree\_and\_coeff;  
BEGIN  
 FOR i IN REVERSE array\_upper(polynom, 1)..array\_lower(polynom, 1)  
 LOOP  
 --если полином существует, то записываем его степень и коэффициент в таблицу num\_polynom  
 if (polynom[i] is not null) then  
 degree\_and\_coeff.degree := i;  
 degree\_and\_coeff.coeff := polynom[i];  
 num\_polynom[cnt] := degree\_and\_coeff;  
  
 cnt := cnt + 1;  
 END IF;  
 END LOOP;  
  
 RETURN num\_polynom;  
END  
$$ LANGUAGE 'plpgsql';  
  
--возведение выражения в степень  
create or replace FUNCTION *exponentiating*(  
 op\_1 VARCHAR,  
 op\_2 VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 res VARCHAR(100);  
 degree NUMERIC(6, 2);  
BEGIN  
 --присваиваем степени второй аргумент функции, а результату первый  
 degree := to\_number(op\_2, '9999');  
 res := op\_1;  
 --до умножаем результат на операнд degree-1 раз, тем самым возводя операнд в степень  
 FOR i IN 1..degree - 1  
 LOOP  
 res := *multiplication*(res, op\_1);  
 END lOOP;  
  
 RETURN res;  
END  
$$ LANGUAGE 'plpgsql';  
  
--удаление скобок  
create or replace FUNCTION *removingBrackets*(  
 op VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 opForS VARCHAR(100);  
 tmp VARCHAR(100);  
 symbol VARCHAR(1) ;  
 stack\_op varchar(100)[];  
 stack\_func varchar(100)[];  
BEGIN  
 --если нет скобок, то ошибка "Некорректно расставлены скобки"  
 IF (NOT *checkBrackets*(op)) THEN  
 RAISE exception 'Некорректно расставлены скобки!';  
 END IF;  
 --добавляем к нашей операции скобки и ноль в начале  
 opForS := concat('(0',op,')');  
 FOR i IN 1..coalesce(char\_length(opForS),0)  
 LOOP  
 symbol := substr(opForS, i, 1);  
 --если символ x, цифра, точка или крышка добавляем к строке tmp  
 IF (  
 symbol similar to '[0-9]'  
 OR (symbol = '.')  
 OR (symbol = 'x')  
 OR (symbol = '^')  
 ) THEN  
 tmp := concat(tmp, symbol);  
 --иначе, если tmp не пуста (в ней записан одночлен),то доваляем новую запись в таблицу stack\_op  
 ELSE  
 BEGIN  
 IF (tmp IS NOT NULL) THEN  
 stack\_op[coalesce(array\_upper(stack\_op, 1), 0) + 1] := tmp;  
 END IF;  
 --обнуляем tmp и упрощаем выражение  
 tmp := null;  
 call *simplification*(substring(opForS from i for 1), stack\_op, stack\_func);  
 END;  
 END IF;  
 END LOOP;  
 RETURN stack\_op[array\_upper(stack\_op, 1)];  
END  
$$ LANGUAGE 'plpgsql';  
  
--добавляем последний символ в таблицу stack\_func  
create or replace PROCEDURE *f1*(  
 cur\_ch VARCHAR,  
 stack\_func IN OUT varchar(100)[]  
) AS  
$$  
BEGIN  
 stack\_func[coalesce(array\_upper(stack\_func, 1), 0) + 1] := cur\_ch;  
END  
$$ LANGUAGE 'plpgsql';  
  
--работа с двумя операндами, после добавляем символ в стек  
create or replace PROCEDURE *f2*(  
 cur\_ch VARCHAR,  
 stack\_op IN OUT varchar(100)[],  
 stack\_func IN OUT varchar(100)[]  
) AS  
$$  
declare  
 cur\_char VARCHAR(1);  
 cur\_op\_1 VARCHAR(100);  
 cur\_op\_2 VARCHAR(100);  
BEGIN  
 --забираем переменные и знак из стека  
  
 cur\_char := stack\_func[array\_upper(stack\_func,1)];  
  
 stack\_func = (select stack\_func[1:array\_upper(stack\_func, 1) - 1]);  
  
 cur\_op\_1 := stack\_op[array\_upper(stack\_op,1)];  
 stack\_op = (select stack\_op[1:array\_upper(stack\_op, 1) - 1]);  
  
 cur\_op\_2 := stack\_op[array\_upper(stack\_op, 1)];  
 stack\_op = (select stack\_op[1:array\_upper(stack\_op, 1) - 1]);  
  
 --в зависимости от знака складываем, вычитаем, умножаем или делим переменные, добавляем результат в стек чисел  
 CASE cur\_char  
 WHEN '+' THEN stack\_op[coalesce(array\_upper(stack\_op, 1), 0) + 1] :=  
 *addition*(*norm*(cur\_op\_2), *norm*(cur\_op\_1));  
 WHEN '-' THEN stack\_op[coalesce(array\_upper(stack\_op, 1), 0) + 1] :=  
 *addition*(*norm*(cur\_op\_2), *inverse*(*norm*(cur\_op\_1)));  
 WHEN '\*' THEN stack\_op[coalesce(array\_upper(stack\_op, 1), 0) + 1] :=  
 *multiplication*(*norm*(cur\_op\_2), *norm*(cur\_op\_1));  
 WHEN '/' THEN stack\_op[coalesce(array\_upper(stack\_op, 1), 0) + 1] :=  
 *division*(*norm*(cur\_op\_2), *norm*(cur\_op\_1));  
 END CASE;  
 stack\_func[coalesce(array\_upper(stack\_func, 1), 0) + 1] := cur\_ch;  
END  
$$ LANGUAGE 'plpgsql';  
  
--удаление одночлена из стека  
create or replace PROCEDURE *f3*(  
 stack\_func IN OUT varchar(100)[]  
) as  
$$  
BEGIN  
 stack\_func = (select stack\_func[1:array\_upper(stack\_func, 1) - 1]);  
END  
$$ LANGUAGE 'plpgsql';  
--выполняем операцию, вызываем упрощение  
create or replace PROCEDURE *f4*(  
 cur\_ch VARCHAR,  
 stack\_op IN OUT varchar(100)[],  
 stack\_func IN OUT varchar(100)[]  
) as  
$$  
declare  
 cur\_char VARCHAR(1);  
 cur\_op\_1 VARCHAR(100);  
 cur\_op\_2 VARCHAR(100);  
BEGIN  
 cur\_char := stack\_func[array\_upper(stack\_func, 1)];  
 stack\_func = (select stack\_func[1:array\_upper(stack\_func, 1) - 1]);  
  
 cur\_op\_1 := stack\_op[array\_upper(stack\_op, 1)];  
 stack\_op = (select stack\_op[1:array\_upper(stack\_op, 1) - 1]);  
  
 cur\_op\_2 := stack\_op[array\_upper(stack\_op, 1)];  
 stack\_op = (select stack\_op[1:array\_upper(stack\_op, 1) - 1]);  
  
 CASE cur\_char  
 WHEN '+' THEN stack\_op[coalesce(array\_upper(stack\_op, 1), 0) + 1] :=  
 *addition*(*norm*(cur\_op\_2), *norm*(cur\_op\_1));  
 WHEN '-' THEN stack\_op[coalesce(array\_upper(stack\_op, 1), 0) + 1] :=  
 *addition*(*norm*(cur\_op\_2), *inverse*(*norm*(cur\_op\_1)));  
 WHEN '\*' THEN stack\_op[coalesce(array\_upper(stack\_op, 1), 0) + 1] :=  
 *multiplication*(*norm*(cur\_op\_2), *norm*(cur\_op\_1));  
 WHEN '/' THEN stack\_op[coalesce(array\_upper(stack\_op, 1), 0) + 1] :=  
 *division*(*norm*(cur\_op\_2), *norm*(cur\_op\_1));  
 END CASE;  
 call *simplification*(cur\_ch, stack\_op, stack\_func);  
END  
$$ LANGUAGE 'plpgsql';  
  
--упрощение  
create or replace PROCEDURE *simplification*(  
 cur\_ch VARCHAR,  
 stack\_op IN OUT varchar(100)[],  
 stack\_func IN OUT varchar(100)[]  
 ) as  
$$  
declare  
 cur\_str VARCHAR(100);  
BEGIN  
  
 CASE cur\_ch  
 WHEN '('  
 THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+'  
 THEN CASE stack\_func[array\_upper(stack\_func, 1)]  
 WHEN '(' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '-' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '\*' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
  
 WHEN '-'  
 THEN CASE stack\_func[array\_upper(stack\_func, 1)]  
 WHEN '(' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '-' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '\*' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
  
 WHEN '\*'  
 THEN CASE stack\_func[array\_upper(stack\_func, 1)]  
 WHEN '(' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '-' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '\*' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
  
 WHEN '/'  
 THEN CASE stack\_func[array\_upper(stack\_func, 1)]  
 WHEN '(' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '+' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '-' THEN call *f1*(cur\_ch, stack\_func);  
 WHEN '\*' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f2*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
  
 WHEN ')'  
 THEN CASE stack\_func[array\_upper(stack\_func,1)]  
 WHEN '(' THEN call *f3*(stack\_func);  
 WHEN '+' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 WHEN '-' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 WHEN '\*' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 WHEN '/' THEN call *f4*(cur\_ch, stack\_op, stack\_func);  
 END CASE;  
 END CASE;  
END  
$$ LANGUAGE 'plpgsql';  
  
--проверка, что открывающих и закрывающих скобок равное количество  
create or replace FUNCTION *checkBrackets*(  
 op VARCHAR  
) RETURNS BOOLEAN AS  
$$  
declare  
 cnt NUMERIC := 0;  
BEGIN  
 FOR i IN 1..coalesce(char\_length(op),0)  
 LOOP  
 IF (cnt < 0) THEN  
 RETURN FALSE;  
 END IF;  
 IF (substring(op from i for 1) = '(') THEN  
 BEGIN  
 cnt := cnt + 1;  
 END;  
 ELSIF (substring(op from i for 1) = ')') THEN  
 cnt := cnt - 1;  
 END IF;  
 END LOOP;  
  
 IF (cnt = 0) THEN  
 RETURN true;  
 END IF;  
  
 RETURN false;  
END  
$$ LANGUAGE 'plpgsql';  
  
--выполняем инверсию, меняем плюс на минус и наоборот  
create or replace FUNCTION *inverse*(  
 op VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 rev\_str VARCHAR(64);  
BEGIN  
 FOR i IN 1..coalesce(char\_length(op),0)  
 LOOP  
 DECLARE  
 v\_character VARCHAR(1);  
 BEGIN  
 v\_character := substring(op from i for 1);  
  
 IF (v\_character = '+') THEN  
 rev\_str := concat(rev\_str,'-');  
 ELSIF (v\_character = '-') THEN  
 rev\_str := concat(rev\_str,'+');  
 ELSE  
 rev\_str := concat(rev\_str,v\_character);  
 END IF;  
 END;  
 END LOOP;  
  
 RETURN rev\_str;  
END  
$$ LANGUAGE 'plpgsql';  
  
--приведение подобных членов  
create or replace FUNCTION *collectingTerms*(  
 op VARCHAR  
) RETURNS VARCHAR AS  
$$  
declare  
 monomials varchar(100)[];  
BEGIN  
 monomials = *splitToGroups*(op);  
  
 RETURN *buildString*(*getPolynom*(monomials));  
END  
$$ LANGUAGE 'plpgsql';  
  
--разбиение операции на группы, запись в подобных членов в таблицу  
create or replace FUNCTION *splitToGroups*(  
 op VARCHAR  
) RETURNS varchar(100)[] AS  
$$  
DECLARE  
 monomials varchar(100)[];  
 first\_pos integer := 1;  
 cnt INTEGER := 1;  
 tmp varchar;  
BEGIN  
 FOR i IN 2..coalesce(char\_length(op),0)  
 LOOP  
 IF (  
 (substring(op from i for 1) = '+')  
 OR (substring(op from i for 1) = '-')  
 ) THEN  
 monomials[cnt] := substring(op from first\_pos for (i - first\_pos));  
 first\_pos := i;  
 cnt := cnt + 1;  
 END IF;  
 END LOOP;  
 tmp := substring(op from first\_pos);  
 monomials[cnt] := tmp;  
 RETURN monomials;  
END  
$$ LANGUAGE 'plpgsql';  
  
--вычисляем коэффициент и степень одночлена и добавляем его в таблицу  
create or replace FUNCTION *getDegreeAndCoeff*(  
 monomial VARCHAR  
) RETURNS type\_degree\_and\_coeff AS  
$$  
declare  
 degree INTEGER := 1;  
 coeff NUMERIC(6, 2) := 0;  
 degree\_and\_coeff type\_degree\_and\_coeff;  
 isNotDigit BOOLEAN := false;  
BEGIN  
 FOR i IN 2..coalesce(char\_length(monomial),0)  
 LOOP  
 --если встретили крышку, то добавляем значение после крышки в степень  
 IF (substring(monomial from i for 1) = '^') THEN  
 degree := to\_number(substring(monomial from i + 1), '9999');  
 EXIT;  
 END IF;  
 --если встретили х, то добавляем значение до цифры в коэффициент  
 IF (substring(monomial from i for 1) = 'x') THEN  
 isNotDigit := true;  
  
 IF (i = 2) THEN  
 coeff := to\_number(concat(substr(monomial,1,1),'1'), '9999');  
 ELSE  
 coeff := to\_number(substr(monomial,1,i-1),'9999');  
 END IF;  
 END IF;  
 END LOOP;  
 --если это число, то степень равна нулю, а коэффициент самому числу  
 IF (NOT isNotDigit) THEN  
 degree := 0;  
 coeff := (monomial)::numeric;  
 END IF;  
 --добавляем значения в таблицу  
 degree\_and\_coeff.degree := degree;  
 degree\_and\_coeff.coeff := coeff;  
  
 RETURN degree\_and\_coeff;  
END  
$$ LANGUAGE 'plpgsql';  
  
--создание строки  
create or replace FUNCTION *buildString*(  
 polynom numeric(6, 2) []  
) RETURNS VARCHAR AS  
$$  
declare  
 res VARCHAR(64);  
BEGIN  
 FOR i IN REVERSE coalesce(array\_upper(polynom, 1),0) ..coalesce(array\_lower(polynom, 1),0)  
 LOOP  
 IF (polynom[i] is null) THEN  
 CONTINUE;  
 END IF;  
  
 IF (polynom[i] = 0) THEN  
 CONTINUE;  
 END IF;  
  
 IF (polynom[i] > 0) THEN  
 res := concat(res,'+');  
 END IF;  
  
 IF ((polynom[i] = 1) AND (i = 0)) THEN  
 res := concat(res,'1');  
 ELSIF (polynom[i] = -1) THEN  
 res := concat(res,'-');  
 IF (i = 0) THEN  
 res := concat(res,'1');  
 END IF;  
 ELSIF (polynom[i] = 1) THEN  
 NULL;  
 ELSE  
 res := concat(res,rtrim(to\_char(polynom[i],'FM999999.99'),'.'));  
 END IF;  
  
 IF (i = 0) THEN  
 CONTINUE;  
 ELSIF (i = 1) THEN  
 res := concat(res,'x');  
 ELSE  
 res := concat(res,'x^',i);  
 END IF;  
 END LOOP;  
-- raise notice 'buildString: res res=%',res;  
 RETURN coalesce(res,'0');  
END  
$$ LANGUAGE 'plpgsql';  
  
call *do\_operation*('0');  
  
call *do\_operation*('0','x');  
  
call *do\_operation*('4x - (x^2 + 1)', 'remove');  
call *do\_operation*('2x - x^2 + 2', 'inverse');  
call *do\_operation*('2x^2 - (x^2 + 2x)-3x+2', 'collect');  
  
call *do\_operation*('x - (4x^2 \* 1x^3 - (x^4 - x^3 + 2x) + 3)', 'remove');  
call *do\_operation*('x','4x^2 \* 1x^3 - (x^4 - x^3 + 2x) + 3', '-');  
  
call *do\_operation*('1 + (2 - 3 - 4x + 5)', 'remove');  
call *do\_operation*('1 + (2 - 3)\*(4 + 5)', 'remove');  
  
  
call *do\_operation*('1 + (2 - 3)\*(4 + 5)', 'x^2 - x', '\*');  
call *do\_operation*('x', 'x^2 - (x + 7)', '\*');  
call *do\_operation*('x', 'x^2 - (x + 7.1)', '\*');  
call *do\_operation*('x^2 - (x + 7x)', 'x', '/');  
call *do\_operation*('x^2 - 8x', 'x-2', '/');  
call *do\_operation*('x^2 - x', '3', '^');