федеральное государственное бюджетное образовательное учреждение
высшего образования
«Алтайский государственный технический университет им. И.И. Ползунова»

Факультет (институт) ФИТ

Кафедра прикладной математики

Отчет защищен с оценкой

_____

Зав.кафедрой _____

Боровцов Е.Г

(подпись) (Фамилия И.О.)

Отчет по практике

Вид	Производственная практика

Код и наименование направления подготовки (специальности): _____

09.03.04 Программная инженерия

_____

Направленность (профиль, специализация): _____

Разработка программно-информационных систем

_____

Форма обучения: очная

Студента Любимов Сергей Михайлович

(Фамилия Имя Отчество)

Группа ПИ-02

г. Барнаул

ФГБОУ ВО «Алтайский государственный технический университет
им. И. И. Ползунова»
Кафедра прикладной математики

Индивидуальное задание

на производственную практику
Технологическая (проектно-технологическая) практика
(вид и тип практики по УП)

студенту Любимов Сергей Михайлович группы ПИ-02
(Ф.И.О.)

График проведения практики:

№ п/п	Содержание работ, выполняемых на практике	Сроки выполнения
1	Изучение организации работы предприятия и используемого на нем инструментария и ПО	19.06-21.06
2	Формулировка задач для решения в ходе практики, вида и объема результатов	21.06
3	Изучение и анализ предметной области, библиографический поиск, изучение литературы.	21.06 – 23.06
4	Постановка задачи, проектирование состава и структуры ПО	24.06 – 26.06
5	Реализация программного обеспечения	27.06 -13.07
6	Тестирование программного обеспечения	13.07 – 14.07
7	Оформление и сдача отчета по практике	15.07-16.07

Руководитель практики от университета _____ Лукоянычев В.Г., доцент, к.т.н.
(подпись) (Ф.И.О., должность)

Задание принял к исполнению _____ Любимов С. М.
(подпись) (Ф.И.О.)

Инструктаж по ОТ, ТБ, ПБ, ПВТР

Инструктаж обучающегося по ознакомлению с требованиями охраны труда, техники безопасности, пожарной безопасности, а также правилами внутреннего трудового распорядка проведен «___» 2023 г.

Руководитель практики университета _____ Лукоянычев В.Г, доцентт, к.т.н
(подпись) (Ф.И.О., должность)

Оглавление

Введение

Производственная практика является необходимым этапом формирования у обучающихся требуемых компетенций. Ее ценность заключается в приобретении новых и закреплении уже полученных знаний. В период прохождения производственной практики, обучающиеся должны закрепить теоретический материал, приобрести практические навыки.

Цель практики - развитие практических умений и навыков, а также компетенций и накопление опыта на основе имеющихся теоретических знаний в процессе выполнения определенных видов работ в рамках своей профессиональной деятельности.

Производственная практика будет выполнена по заданию компании Postgres Professional (ООО "ПОСТГРЕС ПРОФЕССИОНАЛЬНЫЙ").

В рамках оставленной задачи будет подробно рассмотрен процесс написания расширения для СУБД PosrgreSQL. В ходе данной практики вы будут изучаены основы разработки расширений для PostgreSQL, включая создание новых типов данных, работу с композитными типами данных, управление памятью и межпроцессовое взаимодействие. Эта практика позволит вам углубить свои знания в области разработки баз данных и получить ценный опыт работы с PostgreSQL.

1 Описание предметной области и постановка задачи

1.1 Описание предметной области

Процесс проектирования БД представляет собой последовательность переходов от неформального словесного описания информационной структуры предметной области к формализованному описанию объектов предметной области в терминах некоторой модели.

1. Системный анализ и словесное описание информационных объектов предметной области.
2. Проектирование инфологической модели предметной области - частично формализованное описание объектов предметной области в терминах некоторой семантической модели, например, в терминах ER-модели.
3. Даталогическое или логическое проектирование БД, то есть описание БД в терминах принятой даталогической модели данных.
4. Физическое проектирование БД, то есть выбор эффективного размещения БД на внешних носителях для обеспечения наиболее эффективной работы приложения.

Если учитывать, что между вторым и третьим этапами необходимо принять решение, с использованием какой стандартной СУБД будет реализовываться наш проект, то условно процесс проектирования БД можно представить последовательностью выполнения пяти соответствующих этапов. Рассмотрим более подробно этапы проектирования БД[8].

Системный анализ предметной области

С точки зрения проектирования БД в рамках системного анализа, необходимо осуществить первый этап, то есть провести подробное словесное описание объектов предметной области и реальных связей, которые присутствуют между описываемыми объектами. Желательно, чтобы данное описание позволяло корректно определить все взаимосвязи между объектами предметной области.

В общем случае существуют два подхода к выбору состава и структуры предметной области:

- *Функциональный подход* -он реализует принцип движения "от задач" и применяется тогда, когда заранее известны функции некоторой группы лиц и комплексов задач, для обслуживания информационных

потребностей которых создается рассматриваемая БД. В этом случае мы можем четко выделить минимальный необходимый набор объектов предметной области, которые должны быть описаны[8].

- *Предметный подход* -когда информационные потребности будущих пользователей БД жестко не фиксируются. Они могут быть многоаспектными и весьма динамичными. Мы не можем точно выделить минимальный набор объектов предметной области, которые необходимо описывать. В описание предметной области в этом случае включаются такие объекты и взаимосвязи, которые наиболее характерны и наиболее существенны для нее. БД, конструируемая при этом, называется предметной, то есть она может быть использована при решении множества разнообразных, заранее не определенных задач. Конструирование предметной БД в некотором смысле кажется гораздо более заманчивым, однако трудность всеобщего охвата предметной области с невозможностью конкретизации потребностей пользователей может привести к избыточно сложной схеме БД, которая для конкретных задач будет неэффективной[8].

Чаще всего на практике рекомендуется использовать некоторый компромиссный вариант, который, с одной стороны, ориентирован на конкретные задачи или функциональные потребности пользователей, а с другой стороны, учитывает возможность наращивания новых приложений.

Системный анализ должен заканчиваться подробным описанием информации об объектах предметной области, которая требуется для решения конкретных задач и которая должна храниться в БД, формулировкой конкретных задач, которые будут решаться с использованием данной БД с кратким описанием алгоритмов их решения, описанием выходных документов, которые должны генерироваться в системе, описанием входных документов, которые служат основанием для заполнения данными БД.

Общие сведения о СУБД PostgreSQL.

В настоящее время термин «база данных» известен многим людям, даже далеким от профессиональной разработки компьютерных программ. Базы данных стали очень широко распространенной технологией, что потребовало, в свою очередь, большего числа специалистов, способных проектировать их и обслуживать. В ходе эволюции теории и практики баз данных стандартом де-факто стала реляционная модель данных, а в рамках этой модели сформировался и специализированный язык программирования, позволяющий выполнять все необходимые операции с данными -Structured Query Language (SQL). Таким образом, важным

компонентом квалификации специалиста в области баз данных является владение языком SQL[9].

PostgreSQL -это объектно-реляционная система управления базами данных (ОРСУБД, ORDBMS), основанная на POSTGRES, Version 4.2 - программе, разработанной на факультете компьютерных наук Калифорнийского университета в Беркли. В POSTGRES появилось множество новшеств, которые были реализованы в некоторых коммерческих СУБД гораздо позднее[5].

Объектно-реляционная система управления базами данных, именуемая сегодня PostgreSQL, произошла от пакета POSTGRES, написанного в Беркли, Калифорнийском университете. После двух десятилетий разработки PostgreSQL стал самой развитой СУБД с открытым исходным кодом. Проект POSTGRES, возглавляемый профессором Майклом Стоунбрейкером, спонсировали агентство DARPA при Минобороны США, Управление военных исследований (ARO), Национальный Научный Фонд (NSF) и компания ESL, Inc. Реализация POSTGRES началась в 1986 г. Первоначальные концепции системы были представлены в документе ston86, а описание первой модели данных появилось в rowe87. Проект системы правил тогда был представлен в ston87a. Суть и архитектура менеджера хранилища были расписаны в ston87b[10].

С тех пор POSTGRES прошёл несколько этапов развития. Первая «демоверсия» заработала в 1987 и была показана в 1988 на конференции ACMSIGMOD. Версия 1, описанная в ston90a, была выпущена для нескольких внешних пользователей в июне 1989. В ответ на критику первой системы правил (ston89), она была переделана (ston90b), и в версии 2, выпущенной в июне 1990, была уже новая система правил. В 1991 вышла версия 3, в которой появилась поддержка различных менеджеров хранилища, улучшенный исполнитель запросов и переписанная система правил[10].

Последующие выпуски до Postgres95 в основном были направлены на улучшение портируемости и надёжности.

В 1994 Эндри Ю и Джолли Чен добавили в POSTGRES интерпретатор языка SQL. Уже с новым именем Postgres95 был опубликован в Интернете и начал свой путь как потомок разработанного в Беркли POSTGRES, с открытым исходным кодом. Postgres95 версии 1.0.x работал примерно на 30-50% быстрее POSTGRES версии 4.2 (по тестам

Wisconsin Benchmark). Помимо исправления ошибок, произошли следующие изменения[10]:

• На смену языку запросов PostQUEL пришёл SQL (реализованный в сервере). (Интерфейсная библиотека libpq унаследовала своё имя от PostQUEL.) Подзапросы не поддерживались до выхода PostgreSQL, хотя их можно было имитировать в Postgres95 с помощью пользовательских функций SQL. Были заново реализованы агрегатные функции. Также появилась поддержка предложения

• Был усовершенствован интерфейс для работы с большими объектами. Единственным механизмом хранения таких данных стали инверсионные объекты. (Инверсионная файловая система была удалена.)

• Удалена система правил на уровне экземпляров; перезаписывающие правила сохранились.

• С исходным кодом стали распространяться краткие описания возможностей стандартного SQL, а также самого Postgres95.

• Для сборки использовался GNU make (вместо BSD make). Кроме того, стало возможно скомпилировать Postgres95 с немодифицированной версией GCC. В 1996 г. стало понятно, что имя «Postgres95» не выдержит испытание временем. Было выбрано новое имя, PostgreSQL, отражающее связь между оригинальным POSTGRES и более поздними версиями с поддержкой SQL

В процессе разработки Postgres95 основными задачами были поиск и понимание существующих проблем в серверном коде. С переходом к PostgreSQL акценты сместились к реализации новых функций и возможностей, хотя работа продолжается во всех направлениях [10].

Сравнение СУБД PostgreSQL с аналогами.

СУБД PostgreSQL предоставляет большой объем как для самой базы данных, так и для ее элементов. Ограничение объемов таблиц и их элементов в различных СУБД представлено ниже в таблице 1.1. [11]

СУБД	Макс. объем БД	Макс. объем таблицы	Макс. размер строки	Макс. кол-во столбцов	Макс Blob/Clob размер
DB2	512 TB	512 TB	32,677 B	1012	2 GB
Firebird	Неограниченно	32 TB	65,536 B	Зависит от исп. типов.	2 GB
Microsoft Access	2 GB	2 GB	16 MB	255	64 KB
MySQL	Неограниченно	2 GB	64 KB	4096	4 GB
Oracle	Неограниченно	4 GB	Неограниченно	1000	Неограниченно
PostgreSQL	Неограниченно	32 TB	1.6 TB	250	1 GB
SQLite	32 TB	-	-	2000	1 GB

Также PostgreSQL может похвастаться значительным объемом, выделенным под основные типы данных. Для сравнения, в таблице 1.2 приведено ограничение длин типов данных в различных СУБД. [11]

СУБД	Макс. длина типа CHAR	Макс длина типа NUMBER	Мин. значение типа DATE	Макс. значение типа DATE
DB2	32 кб	64 бита	0001	9999
Firebird	32,767 б	64 бита	100	32768
Microsoft Access	255 б	32 бита	-	-
MySQL	64 кб	64 бита	1000	9999
Oracle	4000 б	126 бита	-4712	9999
PostgreSQL	1 гб	Неограниченно	-4713	5874897
SQLite	1 гб	64 бита	Нет типа «Дата»	Нет типа «Дата»

По функциональности СУБД PostgreSQL является одной из лучших в мире. PostgreSQL имеет следующие характеристики:

1. Поддержка обширного списка типов данных, включая как стандартные (числовые, текстовые, с плавающей точкой, логические), так и более редкие, например, денежные, геометрические, xml, json и другие.

2. Возможность использования многомерных массивов. Так как PostgreSQL – объектно-реляционная система управления базами данных, она позволяет осуществлять хранение многомерных массивов.

3. Возможность создания нового типа данных с помощью встроенных команд.

4. PostgreSQL соответствует требованиям ACID (атомарность, согласованность, изолированность, надежность).

5. В PostgreSQL осуществляется поддержка пользовательских функций и временных таблиц.

1.2 Постановка задачи

Поскольку реальные задачи компании слишком громоздкие для реализации одним человеком-новичком за ограниченный промежуток времени производственной практики было предложено разобраться с основными принципами разработки расширений и механизмами их работы без реальной внутренней логики добавляемых функций.

Конечным продуктом работы будет являться “пустое” расширение Postgres, добавляющее в базу данных функции, задействующие различные механизмы, но не имеющие полновесной внутренней логики.

Расширение будет формироваться путем выполнения следующей последовательности заданий:

Установить PostgreSQL из исходных кодов.

1) Создать и установить простейшее расширение, добавляющее в базу данных две функции уровня “Hello, World!”: одну SQL-функцию и одну C-функцию.
2) Реализовать передачу и возврат численных аргументов в функции расширения.
3) Реализовать передачу, возврат и обработку строк в функциях расширения.
4) Добавить в БД новый тип данных. Реализовать передачу и возврат этого типа в функции.
5) Реализовать передачу и возврат в функцию отдельной записи таблицы в функции.
6) Реализовать формирование и возврат таблицы в функции.
7) Добавить регрессионные тесты для функций созданных в заданиях 2-7.
8) Реализовать работу с разделяемой памятью в функциях расширения.
9) Реализовать работу с механизмом последовательного доступа latches.

Задания содержат прогрессию сложности. Так процесс установки БД имеет подробнейшую инструкцию в документации и бесчисленные статьи в сети Internet, а latches не описан в документации, из-за специфичности в сети тоже почти ничего нет (только теория без примеров работы с ними), поэтому разбираться в работе с ними необходимо с помощью исходных кодов соответствующих библиотек и примеров из уже существующих расширений. Все остальные задания представляют собой градацию между 1 и 10, постепенно приучая к работе с исходниками.

Таким образом целью данной работы можно обозначить формирование навыков работы с документацией и исходными кодами, поиска решений и ориентирования в источниках специфичных для области.

2 Разработка программного обеспечения

2.1 Инструменты и технологии

Работа поводилась как в офисе, так и удаленно.

В офисе мне был выдан ноутбук под управлением операционной системы Linux Ubuntu. В качестве редактора кода использовал Visual Studio Code.

Удаленно работал на стационарном компьютере ОС Windows 10 с использованием WSL(Windows Subsystem for Linux) - слой совместимости для запуска Linux-приложений (двоичных исполняемых файлов в формате ELF) в ОС Windows 10 без какой-либо виртуализации. В качестве редактора кода использовал Microsoft Visual Studio.

Для ведения разработки параллельно из двух мест использовал GitHub.

Сборку и компиляцию осуществлял непосредственно в среде ОС с помощью GCC и GNU make для автоматизации.

В качестве основных источников использованы:

Официальная Документация PostgreSQL и Postgres Pro:

https://postgrespro.ru/docs/

Исходные тексты PostgresSQL на Doxygen:

https://doxygen.postgresql.org/

Официальные расширения в открытом репозитории postgres на GitHub:

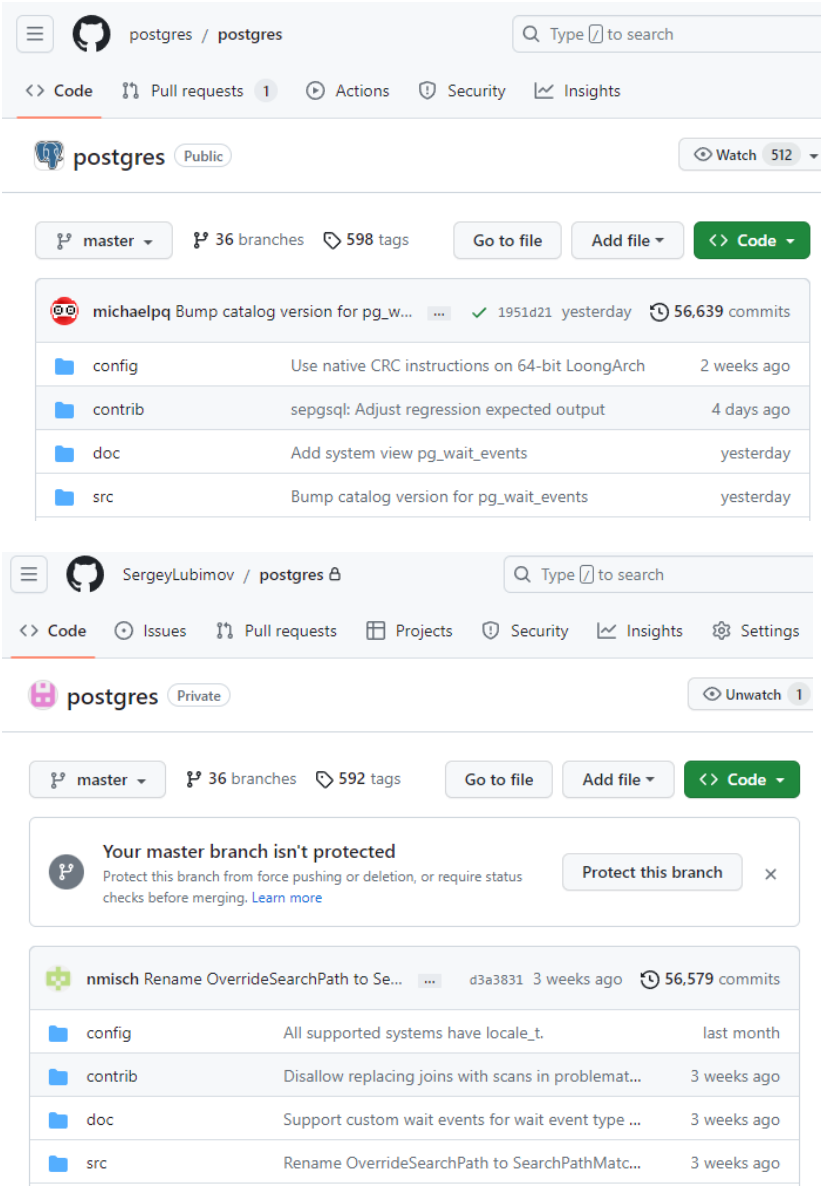
https://github.com/postgres/postgres/tree/2c2eb0d6b27f498851bace47fc19e4c7fc90af4f/contrib

2.2 Реализация

Последовательно рассмотрим реализацию всех десяти заданий

1) Установить PostgreSQL из исходных кодов.

Прежде всего необходимо получить исходный код. Для этого клонируем официальный репозиторий Postgres в свой аккаунт на GitHub.



Теперь можно удобно работать параллельно над одним и тем же проектом и из офиса, и из дома.

Клонируем в локальный репозиторий.

```
sergey@DESKTOP-DLT9385:~$ git clone https://github.com/SergeyLubimov/postgres.git
Cloning into 'postgres'...
Username for 'https://github.com': SergeyLubimov
Password for 'https://SergeyLubimov@github.com':
remote: Enumerating objects: 957448, done.
remote: Total 957448 (delta 0), reused 0 (delta 0), pack-reused 957448
Receiving objects: 100% (957448/957448), 207.92 MiB | 7.09 MiB/s, done.
Resolving deltas: 100% (839444/839444), done.
```

Теперь можем приступить непосредственно к установке. Данный процесс состоит из трех этапов: конфигурирование, сборка, установка.

Рассмотрим каждый из них подробно, но прежде установим компилятор – GCC, и еще несколько необходимых пакетов:

libreadline-dev(помогает обеспечить согласованность пользовательского интерфейса в отдельных программах, которым необходимо предоставлять интерфейс командной строки.[3]),

zlib1g-dev(библиотека, реализующая метод сжатия deflate, который используется в gzip и PKZIP. Данный пакет содержит различные файлы, требуемые для сборки приложений, использующих данную библиотеку, а также примеры и справочные материалы.[3]),

flex(инструмент для создания программ, которые распознают лексические шаблоны в тексте.[3]),

bison(генератор синтаксического анализа общего назначения, который преобразует описание грамматики для контекстно-свободной грамматики LALR (1) в программу на C для анализа этой грамматики.[3]).

Конфигурирование -запуск команды ./configure. Эта команда ищет необходимые библиотеки и заголовочные файлы, осуществляет настройку особых параметров и подключается дополнительные библиотеки (описанные выше —without-параметры из этой области). В результате работы создается файлы Makefiles, содержащий всю необходимую информацию для сборки, его можно просмотреть в текстовом редакторе.[1]

После установки выше перечисленных пакетов конфигурация не должна вызвать проблем. Команда configure имеет множество необязательных параметров, на данном этапе нам потребуется лишь два:

Зададим переменную окружения, указав каталог для установки базы.

--prefix=***ПРЕФИКС***

Разместить все файлы внутри каталога ***ПРЕФИКС***, а не в /usr/local/pgsql. Собственно файлы будут установлены в различные подкаталоги этого каталога; в самом каталоге ***ПРЕФИКС*** никакие файлы не размещаются[2].

Также включить отслеживание зависимостей, что важно для разработки
--enable-depend

Включает автоматическое отслеживание зависимостей. С этим параметром скрипты Makefile настраиваются так, чтобы при изменении любого заголовочного файла пересобирались все зависимые объектные файлы. Это полезно в процессе разработки, но, если вам нужно только скомпилировать и установить сервер, это будет лишней тратой времени. В настоящее время это работает только с GCC.[2]

Перейдем в локальный папку репозитория и запустим конфигурацию

```
sergey@DESKTOP-DLT9385:~/postgres$ ./configure --prefix=/home/sergey/postgres_bin --enable-depend
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking which template to use... linux
checking whether NLS is wanted... no
```

.
.
.

```
configure: creating ./config.status
config.status: creating GNUmakefile
config.status: creating src/Makefile.global
config.status: creating src/include/pg_config.h
config.status: creating src/include/pg_config_ext.h
config.status: creating src/interfaces/ecpg/include/ecpg_config.h
config.status: linking src/backend/port/tas/dummy.s to src/backend/port/tas.s
config.status: linking src/backend/port/posix_sema.c to src/backend/port/pg_sema.c
config.status: linking src/backend/port/sysv_shmem.c to src/backend/port/pg_shmem.c
config.status: linking src/include/port/linux.h to src/include/pg_config_os.h
config.status: linking src/makefiles/Makefile.linux to src/Makefile.port
sergey@DESKTOP-DLT9385:~/postgres$
```

Успешно.

Сборка (компиляция) -запуск команды **make**. Запускается компиляция программы из исходного кода. При работе используется Makefiles, в

которых описаны все параметры сборки приложения. Результат работы - собранная программа в текущей директории.

```
sergey@DESKTOP-DLT9385:~/postgres$ make
make -C ./src/backend generated-headers
make[1]: Entering directory '/home/sergey/postgres/src/backend'
make -C catalog distprep generated-header-symlinks
make[2]: Entering directory '/home/sergey/postgres/src/backend/catalog'
```

.
.
.

```
make[1]: Entering directory '/home/sergey/postgres/config'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/sergey/postgres/config'
sergey@DESKTOP-DLT9385:~/postgres$
```

Успешно. Сборка длилась около 10 минут.

Установка - команда make install. Запускается непосредственная установка собранного приложения. Результат - работоспособная программа (если не было ошибок).

```
sergey@DESKTOP-DLT9385:~/postgres$ make install
make -C ./src/backend generated-headers
make[1]: Entering directory '/home/sergey/postgres/src/backend'
make -C catalog distprep generated-header-symlinks
make[2]: Entering directory '/home/sergey/postgres/src/backend/catalog'
make[2]: Nothing to be done for 'distprep'.
make[2]: Nothing to be done for 'generated-header-symlinks'.
make[2]: Leaving directory '/home/sergey/postgres/src/backend/catalog'
make -C nodes distprep generated-header-symlinks
make[2]: Entering directory '/home/sergey/postgres/src/backend/nodes'
```

.
.
.

```
make[1]: Entering directory '/home/sergey/postgres/config'
/usr/bin/mkdir -p '/home/sergey/postgres_bin/lib/pgxs/config'
/usr/bin/install -c -m 755 ./install-sh '/home/sergey/postgres_bin/lib/pgxs/config/install-sh'
/usr/bin/install -c -m 755 ./missing '/home/sergey/postgres_bin/lib/pgxs/config/missing'
make[1]: Leaving directory '/home/sergey/postgres/config'
```

Перейдем в каталог, который указали в переменной окружения при конфигурации.

```
sergey@DESKTOP-DLT9385:~$ cd /home/sergey/postgres_bin/
sergey@DESKTOP-DLT9385:~/postgres_bin$ dir
bin  include  lib  share
```

Перейдем в директорию bin, где находятся исполняемые файлы PostgreSQL

```
sergey@DESKTOP-DLT9385:~/postgres_bin/bin$ dir
clusterdb   dropuser    pg_archivecleanup  pg_controldata  pg_isready      pg_restore      pg_upgrade       postgres
createdb    ecpg        pg_basebackup      pg_ctl          pg_receivewal   pg_rewind       pg_verifybackup  psql
createuser  initdb      pg_checksums       pg_dump         pg_recvlogical  pg_test_fsync   pg_waldump       reindexdb
dropdb      pg_amcheck  pg_config          pg_dumpall      pg_resetwal     pg_test_timing  pgbench          vacuumdb
```

Перед началом использования сервера, необходимо инициализировать кластер базы данных. Для этого используем утилиту initdb, указав место расположения кластера.

```
sergey@DESKTOP-DLT9385:~/postgres_bin/bin$ ./initdb -k -D /home/sergey/postgres_bin/data
```

Теперь запустим сервер и проверим его статус.

```
sergey@DESKTOP-DLT9385:~/postgres_bin/bin$ ./pg_ctl -D /home/sergey/postgres_bin/data -l logfile start
waiting for server to start.... done
server started
sergey@DESKTOP-DLT9385:~/postgres_bin/bin$ ./pg_ctl -D /home/sergey/postgres_bin/data status
pg_ctl: server is running (PID: 11171)
/home/sergey/postgres_bin/bin/postgres "-D" "/home/sergey/postgres_bin/data"
```

Можем подключиться к серверу

```
sergey@DESKTOP-DLT9385:~/postgres_bin/bin$ ./psql -d postgres
psql (15.3)
Type "help" for help.

postgres=#
```

2) **Создать и установить простейшее расширение, добавляющее в базу данных две функции уровня “Hello, World!”: одну SQL-функцию и одну C-функцию.**

Минимально расширение состоит из трех файлов:

1 Управляющий файл - файл с расширением .control. Содержит основные сведения о расширении - главным образом его название и версию.

2 make-файл. Содержит информацию о том, как собрать расширение, с какими флагами его скомпилировать, какие библиотеки следует прилинковать, и т.д.

3 sql-файл. Содержит запросы, которые будут выполнены, когда пользователь запустит команду CREATE EXTENSION на сервере postgres.

Так как нам требуется добавить также C-функцию, понадобиться также .c файл содержащий данную функцию.

Назовем расширение template.

Управляющий файл template.control

```
comment = 'This is a possible template for real extensions'
default_version = '1.0'
module_pathname = '$libdir/template'
relocatable = false
```

Comment отображается в выводе команды `\dx`, а также в представлении pg_available_extensions. Флаг relocatable говорит, можно ли перемещать расширение между схемами.

Sql - файл

Выводить “Hello, SQL!!!” будем, как сообщение об ошибке.

```
CREATE FUNCTION hello_sql() RETURNS void
AS
$BODY$
    BEGIN
        RAISE NOTICE 'Hello, SQL!!!';
    END
$BODY$
LANGUAGE 'plpgsql';
```

Назвать данный файл необходимо по схеме:

название_расширения--версия_расширения.sql

То есть template--1.0.sql

При обновлении версии расширения используются файлы миграции с именами вроде template--1.0--1.1.sql. Номера версий могут быть какими угодно, хоть foo и bar. PostgreSQL будет просто искать кратчайший путь от текущей версии расширения до версии, указанной в default_version, поочередно выполняя соответствующие файлы миграции.

C-функция

Пользовательские функции могут быть написаны на C (или на языке, который может быть совместим с C, например C++). Такие функции компилируются в динамически загружаемые объекты (также называемые разделяемыми библиотеками) и загружаются сервером по требованию.

Именно метод динамической загрузки отличает функции «на языке C» от «внутренних» функций.[5]

Чтобы гарантировать, что динамически загружаемый объектный файл не будет загружен на несовместимый сервер, PostgreSQL проверяет, содержит ли файл « магический блок » с соответствующим содержимым. Это позволяет серверу обнаруживать явные несовместимости, например код, скомпилированный для другой основной версии PostgreSQL . [5] Данный блок включается в файл с помощью макроса PG_MODULE_MAGIC.

В настоящее время для функций на C применяется только одно соглашение о вызовах («версии 1»). Поддержка этого соглашения обозначается объявлением функции с макросом (PG_FUNCTION_INFO_V1).

По соглашению о вызовах функцию должна возвращать тип Datum.

Datum - это общий тип для хранения внутреннего представления данных, которые можно хранить в таблице PostgreSQL. Для возврата определенного типа данных, как Datum, необходимо использовать соответствующий макрос. В нашем случае, функция не возвращает ничего, поэтому воспользуемся PG_RETURN_VOID().

Все аргументы в функцию должны объявляться через макрос PG_FUNCTION_ARGS. Его подробнее рассмотрим в следующем задании.

Для вызова сообщения ошибки используем функцию ereport.

Таким образом файл template.c будет выглядеть следующим образом.

```c
#include "postgres.h"
#include "fmgr.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(hello_c);

Datum hello_c(PG_FUNCTION_ARGS)
{
    ereport(NOTICE, errmsg("Hello, C!!!"));

    PG_RETURN_VOID();
}
```

Чтобы C-функцию можно было вызвать из базы данных, её также необходимо создать в sql-файле

```sql
CREATE FUNCTION hello_c()
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C;
```

Данная SQL команда создает функцию, чье тело определено на языке C в переменной module_pathname, заданной в управляющем файле.

3) Реализовать передачу и возврат численных аргументов в функции расширения.

Создадим простенькую функцию, возвращающую сумму двух своих аргументов. Чтобы было хоть немного интереснее, пусть сумма будет по модулю, указанному в третьем аргументе.

Аргументы функции объявляются через макрос PG_FUNCTION_ARGS. Макрос содержит в себе fcinfo – структуру типа FunctionCallInfo. Она помимо прочей информации хранит аргументы переданные функции, приведенные к типу Datum.

Аргумент можно получить и привести в требуемому типу с помощью макроса формата PG_GETARG_XXX(n), n – порядковый индекс аргумента. В данной случае используем PG_GETARG_INT32.

Результат возвращать будем с помощью PG_RETURN_INT32(x).

Таком образом функция будет выглядеть следующим образом:

```
Datum add_modulo(PG_FUNCTION_ARGS)
{
    int summand1, summand2, mod, ret;

    summand1 = PG_GETARG_INT32(0);
    summand2 = PG_GETARG_INT32(1);
    mod = PG_GETARG_INT32(2);

    if (summand1 > 0 && summand2 > 0 && mod > 0)
        ret = (summand1 + summand2) % mod;
    else
    {
        ret = -1;
        ereport(NOTICE, errmsg("ERROR: negative argument"));
    }
    PG_RETURN_INT32(ret);
}
```

В sql-файл потребуется добавить:

```
CREATE FUNCTION add_modulo(integer, integer, integer)
RETURNS integer
AS 'MODULE_PATHNAME'
LANGUAGE C;
```

4) **Реализовать передачу, возврат и обработку строк в функциях расширения.**

Создадим функцию, получающую три строки внутреннего для PostgreSQL типа TEXT и преобразующую их в “уравнение” – строку формата:

$$arg1 + arg2 = arg3$$

Так как основной принцип аналогичен используемым в заданиях 2 и 3, будем лишь рассматривать ключевые моменты, а полный код будет приведен в приложении.

Получение аргумента типа текст происходит через макрос PG_GETARG_TEXT_P_COPY

Возврат типа текст через PG_RETURN_TEXT_P

Внутри с помощью функций text_to_cstring, cstring_to_text можно конвертировать TEXT в обычную с-строку и обратно, что позволяет удобно с ним работать.

В postgres существует крайне удобная функция psprintf, она формирует строку по указанному формату, выделяя память с помощью функции palloc. Использование стандартных функций calloc и malloc в postgres не рекомендуется, так как могут возникать проблемы в работе контекстов памяти.

5) Добавить в БД новый тип данных. Реализовать передачу и возврат этого типа в функции.

Создадим тип данных – Комплексное число.

Опишем его структуру в С-файле:

```
typedef struct
{
   double real;
   double imaginary;

} complex_number;
```

Составные типы данных в PostgreSQL в тексте программы приставляются, как строка с явным приведением типа. Например, в нашем случае:

‘(1,2)’::complex_number

Для этого за типом данных необходимо закрепить функцию преобразования экземпляр класса CSting в экземпляр этого типа.

В нашем случае:

```
Datum complex_number_in(PG_FUNCTION_ARGS)
{
   char* s = PG_GETARG_CSTRING(0);

   complex_number* a = (complex_number*)palloc(sizeof(complex_number));

   sscanf(s, "(%lf,%lf)", &(a->real), &(a->imaginary));


   PG_RETURN_POINTER(a);
}
```

Так же необходима обратная функция.

В SQL-файле необходимо объявить создание типа данных, затем объявить функции-обработчики, после чего создать тип данных задав для него эти функции, а также размер типа в байтах.

```
CREATE TYPE complex_number;

CREATE OR REPLACE FUNCTION complex_number_in(cstring)
RETURNS complex_number
AS 'MODULE_PATHNAME'
LANGUAGE C IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION complex_number_out(complex_number)
RETURNS cstring
AS 'MODULE_PATHNAME'
LANGUAGE C IMMUTABLE STRICT;

CREATE TYPE complex_number
(
    internallength = 16,
    input = complex_number_in,
    output = complex_number_out
   );
```

Передача/возврат аргумента составного типа происходит в виде указателя через макросы PG_GETARG_POINTER и PG_RETURN_POINTER.

6) Реализовать передачу и возврат в функцию отдельной записи таблицы в функции.

Название типа кортежа таблицы совпадает с названием таблицы. Так как по требованиям реляционной модели данных порядок полей в кортеже не должен играть роли, кортеж представляет собой не однозначно последовательно упорядоченную структуру, как С-структура, а кучу.

В программе мы должны получить доступ к куче, получив её хедер, с помощью PG_GETARG_HEAPTUPLEHEADER. С кучей нельзя работать как обычно через указатель, необходимо использовать специальную функцию GetAttributeByName.

Для формирования кортежа внутри функции необходимо прежде всего с помощью get_call_result_type получить дескриптор фактического типа результата, возвращаемого функцией. По дескриптору результата функцией TupleDescGetAttInMetadata получить информацию о структуре атрибутов кортежа, после чего сформировав данные кортежа, как массив с-строк, строим кортеж функцией BuildTupleFromCStrings.

Возврат кортежа из функции осуществляется макросом HeapTupleGetDatum.

Пример кода формирования кортежа:

```
Oid         resultTypeId;
TupleDesc   resultTupleDesc;
int*            dummy;

AttInMetadata *a;
char *values[2];
HeapTuple h;

if (get_call_result_type(fcinfo, &resultTypeId, & resultTupleDesc) != TYPEFUNC_COMPOSITE)
    elog(ERROR, "Function returning record called in context that cannot accept type record");

a = TupleDescGetAttInMetadata(resultTupleDesc);

values[0] = psprintf("Name");
values[1] = psprintf("%d", 1000);

h = BuildTupleFromCStrings(a, values);

return HeapTupleGetDatum(h);
```

7) Реализовать формирование и возврат таблицы в функции.

Метод, используемый в прошлом пункте для получения типа возвращаемого результата функции, подходит именно для отдельных записей, в данном случае необходимо действовать иначе.

Запускаем режим материализации для fcinfo функции с помощью InitMaterializedSRF.

Получаем указатель на структуру содержащую информацию результате функции приведя её явно к типу ReturnSetInfo.

После чего запускаем специальную функцию tuplestore_putvalues, принимающую в качестве аргументов информацию о типе результата, его дескриптор и массив с-строк со значениями полей, а так же зануленный булевый массив(для выставления флагов). Данная функция формирует только один кортеж таблицы за вызов.

В sql-файле в качестве возвращаемого функцией аргумента необходимо указать имя таблицы. Чтобы указать, что это именно таблица, а не кортеж, необходимо указать SETOF.

8) Добавить регрессионные тесты для функций созданных в заданиях 2-7.

Регрессионное тестирование (regression testing): Тестирование уже протестированной программы, проводящееся после модификации для уверенности в том, что процесс модификации не внес или не активизировал

ошибки в областях, не подвергавшихся изменениям. Проводится после изменений в коде программного продукта или его окружении. [6]

Для тестирования расширений PostgreSQL в каталоге расширения необходимо создать подкаталоги sql и expected, в которые поместить файлы с расширениями .sql и .out соответственно. Имя файлов совпадает с именем расширения.

В sql-файле содержится набор SQL-команд, которые будут последовательно выполнены при запуске make с параметром check. Под результаты выполнения команды будут созданы (если их еще нет) каталог и файл: results и template.out.

В template.out каталога expected содержится ожидаемый результат выполнения команд. Он будет посимвольно сравнен с соответствующим файлом каталога results и только при полном совпадении тест будет считаться пройдённым.

Также необходимо изменить makefile, добавив в него выставление переменной REGRESS.

Содержимое файла тестов и файла ожидаемых результатов приведено в Приложении Б вместе с исходным кодом программ. Скрин успешного выполнения тестирования приведен в Приложении А.

9) Реализовать работу с разделяемой памятью в функциях расширения.

Расширения PostgreSQL представляют собой обычные динамические библиотеки. По умолчанию они подгружаются бэкендом по мере надобности. При таком поведении системы выделить разделяемую память из расширения несколько затруднительно. Обойти проблему можно при помощи конфигурационного параметра shared_preload_libraries. В нем указывается список динамических библиотек, которые postmaster должен подгрузить во время запуска СУБД. [7]

В каталоге кластера БД необходимо найти файл postgresql.conf, и нем добавить расширение в список динамических библиотек.Если библиотека экспортирует функцию _PG_init(), происходит ее вызов.

Сначала требуемое количество памяти нужно *запросить* при помощи функции RequestAddinShmemSpace(). В PostgreSQL < 15 вызов осуществлялся прямо из функции _PG_init(), но с этим были

связаны некоторые неудобства. Поэтому в PostgreSQL ≥ 15 вызов делается из хука shmem_request_hook. [7]

Чтобы код корректно работал на всех поддерживаемых платформах, включая Windows, хуку необходимо захватывать AddinShmemInitLock. По этой же причине lock принято делать частью структуры в разделяемой памяти, а не хранить в локальной памяти процесса.

AddinShmemInitLock имеет тип LWLock*. LWLock расшифровывается, как lightweight lock. Это не локи операционной системы, а собственная реализация PostgreSQL на основе разделяемой памяти, спинлоков и семафоров.[7]

Далее выделенную память нужно найти и присвоить ей имя при помощи функции ShmemInitStruct(). Это уже можно сделать и из кода какой-нибудь хранимки, но удобнее воспользоваться еще одним хуком, shmem_startup_hook. Хук позволяет сосредоточить весь код, связанный с инициализацией разделяемой памяти, в одном месте, и не мешать его с основной логикой.

Таким образом, создадим структуру SharedStruct. И зададим статическую глобальную переменную – указатель на этот тип. Также создадим статическую глобальную переменную типа shmem_request_hook_type для сохранения текущего хука. В _PG_init() присвоим этой переменной shmem_request_hook, а текущим хуком сделаем свою функцию запроса памяти - назовем ее requestSharedMemory. Учтем, что новый хук обязан вызывать предыдущий.

Создадим две функции добавляемые непосредственно в БД. Одна устанавливает значение SharedStruct, другая его считывает и выводит. Перед работой со структурой, каждая эта функция должна убедится, что она инициирована, в противном случае инициировать ее самостоятельно.

10) Реализовать работу с механизмом последовательного доступа latches.

Защелки (latches) – это механизм синхронизации потоков. PostgreSQL имеет собственную реализацию данного механизма. Защелка может быть установлена при помощи SetLatch() и сброшена при помощи ResetLatch(). Еще есть WaitLatch(). Он ждет на сброшенной защелке до тех пор, пока ее кто-нибудь не установит, либо до истечения таймаута. Другими словами, защелки позволяют одному процессу просигнализировать второму о каком-то событии.

Latch представляет собой объект в разделяемой памяти, к которому будут иметь доступ несколько процессов. Необходимо объявить указатель на Latch, как статическую глобальную переменную, в хуке созданном в предыдущем пункте запросить для него память. Инициализация защелки перед её использованием в функции происходит аналогично предыдущему пункту, но необходимо дополнительно использовать InitSharedLatch. InitSharedLatch необходимо вызывать в postmaster перед разветвлением дочерних процессов, обычно сразу после выделения блока общей памяти содержащий защелку с помощью ShmemInitStruct.

Процесс, которому нужно ожидать сигнала других процессов, должен присвоить защелку функцией OwnLatch. После чего войти в цикл ожидания события, состоящий из проверки наступления условий выхода и вызова WaitLatch. Другой процесс для подачи сигнала должен вызвать SetLatch.

Заключение

В ходе производственной практики был рассмотрен процесс написания расширения для СУБД PostgreSQL, были изучены и опробованы механизмы передачи и возврата в/из функции аргументов различных типов, в том числе, пользовательских и композитных типов, принципы регрессионного тестирования, а также принципы использования разделяемой памяти и один из способов организации межпроцессового взаимодействия.

Результатом практики является расширение, не несущее в себе практической логики, но содержащее функции, которые можно использовать в качестве шаблона. Добавив в них полновесное наполнение, можно создать полноценное расширение.

Но главным результатом производственной практики является приобретение навыков работы с документацией и исходными кодами, умения ориентироваться в библиотеках компании, читать, понимать и применять уже разработанный инструментарий. Данный опыт также позволит разбираться и выполнять различные задачи, отличные от выполненных в ходе практики, в рамках PostgreSQL и не только.

Список используемых источников

1) Установка PostgreSQL из исходных кодов // ТОЛМАЧЕВ ПАВЕЛ ВЛАДИМИРОВИЧ: ПЕРСОНАЛЬНЫЙ САЙТ URL: https://ptolmachev.ru/ustanovka-postgresql-iz-isxodnyx-kodov.html?ysclid=llkduipiv6284057872

2) Процедура установки // PostgresPro URL: https://postgrespro.ru/docs/postgresql/15/install-procedure

3) Пакеты системы // Debian URL: https://www.debian.org/distrib/packages

4) Учимся писать расширения на языке C для PostgreSQL // Записки программиста URL: https://eax.me/postgresql-extensions/

5) Документация к PostgreSQL 15.4 // The PostgreSQL Global Development Group Официальная страница русскоязычной документации: https://postgrespro.ru/docs/

6) VladislavEremeev Регрессионные виды тестирования (Regression testing) // GitHub URL: https://github.com/VladislavEremeev/QA_bible/blob/master/vidy-metody-urovni-testirovaniya/regressionnye-vidy-testirovaniya-regression-testing.md

7) Расширения PostgreSQL: разделяемая память и локи // Записки программиста URL: https://eax.me/postgresql-shmem-locks/

8) Лекция №6 Безопасность систем баз данных // Саратовский Государственный Технический Университет им. Ю.А. Гагарина URL: https://studfile.net/preview/1602933/

9) PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. — СПб.: БХВ-Петербург, 2018. — 336 с.: ил. ISBN 978-5-9775-4022-3

10) Краткая история PostgreSQL // PostgreSQL.Ru.Net URL: https://pgdocs.ru/manual/history.html

11) Никулин В. И. Разработка программного обеспечения формирования и ведения базы данных на основе PostgreSQL для системы недропользования: дис. техн. наук: 02.03.02. - Белгород 2017, 68 с.

Приложение А. Тестирование программного обеспечения

Задание 2

```
postgres=# CREATE EXTENSION template;
CREATE EXTENSION
postgres=# SELECT hello_SQL();
NOTICE:  Hello, SQL!!!
 hello_sql
-----------

(1 row)

postgres=# SELECT hello_c();
NOTICE:  Hello, C!!!
 hello_c
---------

(1 row)
```

Задание 3

```
postgres=# SELECT add_modulo(4, 2, 10);
 add_modulo
------------
          6
(1 row)

postgres=# SELECT add_modulo(4, 2, 5);
 add_modulo
------------
          1
(1 row)
```

Задание 4

```
postgres=# SELECT equation('x', '2', '0');
 equation
-----------
 x + 2 = 0
(1 row)
```

Задание 5

```
postgres=# SELECT complex_add('(1,2)'::complex_number, '(3,4)'::complex_number);
     complex_add
---------------------
 (4.000000,6.000000)
(1 row)
```

Задание 6

```
postgres=# SELECT name, check_salary(emp, 1500) FROM emp;
 name  | check_salary
-------+--------------
 name1 | f
 name2 | t
(2 rows)
```

```
postgres=# SELECT generate_record('name', 1500);
 generate_record
-----------------
 (name!!!,3000)
(1 row)
```

Задание 7

```
postgres=# SELECT generate_table('name1', 1, 'name2', 2);
 generate_table
----------------
 (name1,1)
 (name2,2)
(2 rows)
```

Задание 8

```
sergey@DESKTOP-DLT9385:~/postgres/contrib/template$ make installcheck
make -C ../../src/test/regress pg_regress
make[1]: Entering directory '/home/sergey/postgres/src/test/regress'
make -C ../../../src/port all
```

.
.
.

```
============== running regression test queries        ==============
test template                     ... ok           15 ms

======================
 All 1 tests passed.
======================
```

Задание 9

```
postgres=# SELECT get_shered_message();
 get_shered_message
--------------------
 (null)
(1 row)

postgres=# SELECT set_shered_message('test');
 set_shered_message
--------------------

(1 row)

postgres=# SELECT get_shered_message();
 get_shered_message
--------------------
 test
(1 row)
```

Задание 10

```
postgres=# SELECT wait_signal();
 wait_signal
-------------
 TIMEOUT!!!
(1 row)
```

```
postgres=# SELECT wait_signal();
 wait_signal
-------------
 It's OK!!!
(1 row)
```

```
postgres=# SELECT give_signal();
 give_signal
-------------

(1 row)
```

Приложение Б. Исходный текст программы

template.control

```
comment = 'This is a possible template for real extensions'
default_version = '1.0'
module_pathname = '$libdir/template'
relocatable = false
```

Makefile

```
MODULES = template
EXTENSION = template
REGRESS = template

DATA = template--1.0.sql

ifdef USE_PGXS
PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
else
subdir = contrib/cube
top_builddir = ../..
include $(top_builddir)/src/Makefile.global
include $(top_srcdir)/contrib/contrib-global.mk
endif
```

template--1.0.sql

```
/*********************************************************
**
**  Zadanie 2
**
*********************************************************/

CREATE FUNCTION hello_sql() RETURNS void
AS
$BODY$
    BEGIN
        RAISE NOTICE 'Hello, SQL!!!';
    END
$BODY$
LANGUAGE 'plpgsql';

CREATE FUNCTION hello_c()
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C;
```

```
/*****************************************************
**
**  Zadanie 3
**
*****************************************************/

CREATE FUNCTION add_modulo(integer, integer, integer)
RETURNS integer
AS 'MODULE_PATHNAME'
LANGUAGE C;

/*****************************************************
**
**  Zadanie 4
**
*****************************************************/

CREATE FUNCTION equation(text, text, text)
RETURNS text
AS 'MODULE_PATHNAME'
LANGUAGE C;

/*****************************************************
**
**  Zadanie 5
**
*****************************************************/

CREATE TYPE complex_number;


CREATE OR REPLACE FUNCTION complex_number_in(cstring)
RETURNS complex_number
AS 'MODULE_PATHNAME'
LANGUAGE C IMMUTABLE STRICT;


CREATE OR REPLACE FUNCTION complex_number_out(complex_number)
RETURNS cstring
AS 'MODULE_PATHNAME'
LANGUAGE C IMMUTABLE STRICT;


CREATE TYPE complex_number
(
    internallength = 16,
    input = complex_number_in,
    output = complex_number_out
);

CREATE OR REPLACE FUNCTION complex_add(complex_number, complex_number)
RETURNS complex_number
AS 'MODULE_PATHNAME'
LANGUAGE C;

/*****************************************************
**
**  Zadanie 6
**
*****************************************************/

CREATE FUNCTION check_salary(emp, integer) RETURNS boolean
AS 'MODULE_PATHNAME'
```

```
LANGUAGE C STRICT;

CREATE FUNCTION generate_record(text, int) RETURNS emp
AS 'MODULE_PATHNAME'
LANGUAGE C;

/*****************************************************
**
**  Zadanie 7
**
*****************************************************/

CREATE OR REPLACE FUNCTION generate_table(text, integer, text, integer)
RETURNS SETOF emp
AS 'MODULE_PATHNAME'
LANGUAGE C;

/*****************************************************
**
**  Zadanie 9
**
*****************************************************/

CREATE FUNCTION get_shered_message()
RETURNS text
AS 'MODULE_PATHNAME'
LANGUAGE C;

CREATE FUNCTION set_shered_message(text)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C;

/*****************************************************
**
**  Zadanie 10
**
*****************************************************/

CREATE OR REPLACE FUNCTION wait_signal()
RETURNS text
AS 'MODULE_PATHNAME'
LANGUAGE C;

CREATE OR REPLACE FUNCTION give_signal()
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C;
```

template.c

```
#include "postgres.h"

#include <string.h>
#include <stdio.h>

#include "executor/executor.h"
#include "fmgr.h"
#include "funcapi.h"
#include "utils/builtins.h"
#include "utils/geo_decls.h"
#include "utils/memutils.h"
#include <miscadmin.h>
#include <storage/ipc.h>
#include <storage/shmem.h>
#include <storage/lwlock.h>
#include <storage/latch.h>
#include "utils/wait_event.h"


PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(hello_c);
PG_FUNCTION_INFO_V1(add_modulo);
PG_FUNCTION_INFO_V1(equation);
PG_FUNCTION_INFO_V1(complex_number_in);
PG_FUNCTION_INFO_V1(complex_number_out);
PG_FUNCTION_INFO_V1(complex_add);
PG_FUNCTION_INFO_V1(check_salary);
PG_FUNCTION_INFO_V1(generate_record);
PG_FUNCTION_INFO_V1(generate_table);
PG_FUNCTION_INFO_V1(get_shered_message);
PG_FUNCTION_INFO_V1(set_shered_message);
PG_FUNCTION_INFO_V1(wait_signal);
PG_FUNCTION_INFO_V1(give_signal);

/////////////////////////////////////////////////////////
//
//  Задание 2
//
/////////////////////////////////////////////////////////

// Функция выводит строку "Hello, C!!!", как сообщение об ошибке
//
Datum
hello_c(PG_FUNCTION_ARGS)
{
	ereport(NOTICE, errmsg("Hello, C!!!"));

	PG_RETURN_VOID();
}

/////////////////////////////////////////////////////////
//
//  Задание 3
//
/////////////////////////////////////////////////////////

// Функция выводит сумму двух первых аргументов
// по модулю третьего
```

```
//
Datum
add_modulo(PG_FUNCTION_ARGS)
{
	int summand1, summand2, mod, ret;

	summand1 = PG_GETARG_INT32(0);
	summand2 = PG_GETARG_INT32(1);
	mod = PG_GETARG_INT32(2);

	if (summand1 > 0 && summand2 > 0 && mod > 0)
		ret = (summand1 + summand2) % mod;
	else
	{
		ret = -1;
		ereport(NOTICE, errmsg("ERROR: negative argument"));
	}
	PG_RETURN_INT32(ret);
}

////////////////////////////////////////////////////////
//
//  Задание 4
//
////////////////////////////////////////////////////////

// Функция выводит строку представлющую аргументы, как уравнение
//
Datum
equation(PG_FUNCTION_ARGS)
{
	char *arg1, *arg2, *arg3;

	arg1 = text_to_cstring(PG_GETARG_TEXT_P_COPY(0));
	arg2 = text_to_cstring(PG_GETARG_TEXT_P_COPY(1));
	arg3 = text_to_cstring(PG_GETARG_TEXT_P_COPY(2));

	PG_RETURN_TEXT_P(cstring_to_text(psprintf("%s + %s = %s", arg1, arg2,
arg3)));
}

////////////////////////////////////////////////////////
//
//  Задание 5
//
////////////////////////////////////////////////////////

// Определение типа данных - комплекное число
//
typedef struct
{
	double real;
	double imaginary;

} complex_number;

// Опредление input-функции нового типа
//
Datum
complex_number_in(PG_FUNCTION_ARGS)
{
	char* s = PG_GETARG_CSTRING(0);
```

```
	complex_number* a = (complex_number*)palloc(sizeof(complex_number));

	sscanf(s, "(%lf,%lf)", &(a->real), &(a->imaginary));

	PG_RETURN_POINTER(a);
}

// Определение output-функции нового типа
//
Datum
complex_number_out(PG_FUNCTION_ARGS)
{
	complex_number* a = (complex_number*)PG_GETARG_POINTER(0);

	PG_RETURN_CSTRING(psprintf("(%lf,%lf)", a->real, a->imaginary));
}

// Функция сложения комплексных чисел
//
Datum
complex_add(PG_FUNCTION_ARGS)
{
	complex_number* a = (complex_number*)PG_GETARG_POINTER(0);
	complex_number* b = (complex_number*)PG_GETARG_POINTER(1);

	complex_number* c = (complex_number*)palloc(sizeof(complex_number));

	c->real = a->real + b->real;
	c->imaginary = a->imaginary + b->imaginary;

	PG_RETURN_POINTER(c);
}

////////////////////////////////////////////////////////
//
//  Задание 6
//
////////////////////////////////////////////////////////

// Функция принимает кортеж типа emp и целочисленный параметр
// Возращает true,если поле salary кортежа превышает этот параметр
//
Datum
check_salary(PG_FUNCTION_ARGS)
{
	HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
	int32            limit = PG_GETARG_INT32(1);
	bool isnull = false;
	Datum salary;

	salary = GetAttributeByName(t, "salary", &isnull);
	if (isnull)
		PG_RETURN_BOOL(false);

	PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}

// Функция принимает два парметра: text и integer
// формирует из них кортерж таблицы emp и возвращает его
//
Datum
generate_record(PG_FUNCTION_ARGS)
{
```

```
	Oid			resultTypeId;
	TupleDesc		resultTupleDesc;
	AttInMetadata* a;
	char* str;
	int32			x;
	HeapTuple		h;
	char* values[2];

	str = text_to_cstring(PG_GETARG_TEXT_P_COPY(0));

	x = PG_GETARG_INT32(1);

	if (get_call_result_type(fcinfo, &resultTypeId, &resultTupleDesc) !=
TYPEFUNC_COMPOSITE)
		elog(ERROR, "Function returning record called in context that cannot
accept type record");

	a = TupleDescGetAttInMetadata(resultTupleDesc);

	values[0] = psprintf("%s!!!", str);
	values[1] = psprintf("%d", 2 * x);

	h = BuildTupleFromCStrings(a, values);

	return HeapTupleGetDatum(h);
}

////////////////////////////////////////////////////
//
//  Задание 7
//
////////////////////////////////////////////////////

// Функция принимает две пары: text и integer
// формирует из них таблицу emp, состоящую двух кортежей
// и возвращает её
//
Datum
generate_table(PG_FUNCTION_ARGS)
{
	ReturnSetInfo* rsinfo;
	Datum values[2];
	bool nulls[2];

	InitMaterializedSRF(fcinfo, 0);
	rsinfo = (ReturnSetInfo*)fcinfo->resultinfo;

	memset(nulls, 0, sizeof(nulls));

	values[0] = PG_GETARG_DATUM(0);
	values[1] = PG_GETARG_DATUM(1);
	tuplestore_putvalues(rsinfo->setResult, rsinfo->setDesc, values, nulls);

	values[0] = PG_GETARG_DATUM(2);
	values[1] = PG_GETARG_DATUM(3);
	tuplestore_putvalues(rsinfo->setResult, rsinfo->setDesc, values, nulls);

	return (Datum)0;
}

////////////////////////////////////////////////////
//
//  Задания 9-10
```

```
//
////////////////////////////////////////////////////

typedef struct SharedStruct
{
	char* message;
} SharedStruct;

static shmem_request_hook_type prev_shmem_request_hook = NULL;
static void requestSharedMemory(void);

static SharedStruct* shared_struct = NULL;
static Latch* lt;

void _PG_init(void);

void
_PG_init(void)
{
	prev_shmem_request_hook = shmem_request_hook;
	shmem_request_hook = requestSharedMemory;
}

static void
requestSharedMemory(void)
{
	if (prev_shmem_request_hook)
		prev_shmem_request_hook();

	RequestAddinShmemSpace(sizeof(shared_struct));
	RequestNamedLWLockTranche("SharedStruct", 1);

	RequestAddinShmemSpace(sizeof(lt));
	RequestNamedLWLockTranche("Latch", 1);

}

static void
startupSharedMemory1(void)
{
	bool found;

	LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
	shared_struct = ShmemInitStruct("SharedStruct", sizeof(shared_struct),
&found);

	if (!found)
		shared_struct->message = NULL;

	LWLockRelease(AddinShmemInitLock);
}

static void
startupSharedMemory2(void)
{
	bool found;

	LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
	lt = ShmemInitStruct("Latch", sizeof(lt), &found);

	if (!found)
		InitSharedLatch(lt);
```

```
	LWLockRelease(AddinShmemInitLock);
}

// Функция возвращает значение
// структуры разделяемой памяти
//
Datum
get_shered_message(PG_FUNCTION_ARGS)
{
	if (shared_struct == NULL)
		startupSharedMemory1();

	PG_RETURN_TEXT_P(cstring_to_text(psprintf("%s", shared_struct->message)));
}

// Функция устанавливет значение
// структуры разделяемой памяти
//
Datum
set_shered_message(PG_FUNCTION_ARGS)
{
	if (shared_struct == NULL)
		startupSharedMemory1();

	if (shared_struct->message != NULL)
		pfree(shared_struct->message);

	shared_struct->message = text_to_cstring(PG_GETARG_TEXT_P_COPY(0));

	PG_RETURN_VOID();
}

// Функция ожидающая утсановку защелки
//
Datum
wait_signal(PG_FUNCTION_ARGS)
{
	int buff = 0;
	char* str;

	if (lt == NULL)
		startupSharedMemory2();

	OwnLatch(lt);

	for (;;)
	{
		ResetLatch(lt);
		if (buff & WL_LATCH_SET)
		{
			str = psprintf("It's OK!!!");
			break;
		}
		if (buff & WL_TIMEOUT)
		{
			str = psprintf("TIMEOUT!!!");
			break;
		}
		buff = WaitLatch(lt, WL_LATCH_SET | WL_TIMEOUT, 10000,
PG_WAIT_EXTENSION);
	}
	DisownLatch(lt);
```

```
	PG_RETURN_TEXT_P(cstring_to_text(str));
}

// Функция устанавливающая защелку
//
Datum
give_signal(PG_FUNCTION_ARGS)
{
	if (lt == NULL)
		startupSharedMemory2();

	SetLatch(lt);
	PG_RETURN_VOID();
}
```

/sql/template.sql

```
DROP TABLE emp;
CREATE TABLE emp(name text, salary integer);
INSERT INTO emp
VALUES ('name1', 1000),('name2', 2000);
CREATE EXTENSION template;
SELECT hello_SQL();
SELECT hello_c();
SELECT add_modulo(4, 2, 10);
SELECT add_modulo(4, 2, 5);
SELECT equation('x', '2', '0');
SELECT complex_add('(1,2)'::complex_number, '(3,4)'::complex_number);
SELECT name, check_salary(emp, 1500) FROM emp;
SELECT generate_record('name', 1500);
SELECT generate_table('name1', 1, 'name2', 2);
```

/expected/ template.out

```
 hello_c
---------

(1 row)

SELECT add_modulo(4, 2, 10);
 add_modulo
------------
          6
(1 row)

SELECT add_modulo(4, 2, 5);
 add_modulo
------------
          1
(1 row)

SELECT equation('x', '2', '0');
 equation
-----------
 x + 2 = 0
(1 row)

SELECT complex_add('(1,2)'::complex_number, '(3,4)'::complex_number);
     complex_add
--------------------
 (4.000000,6.000000)
(1 row)
```

```
SELECT name, check_salary(emp, 1500) FROM emp;
 name  | check_salary
-------+--------------
 name1 | f
 name2 | t
(2 rows)

SELECT generate_record('name', 1500);
 generate_record
-----------------
 (name!!!,3000)
(1 row)

SELECT generate_table('name1', 1, 'name2', 2);
 generate_table
----------------
 (name1,1)
 (name2,2)
(2 rows)
```