МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания
по выполнению лабораторных работ
по дисциплине «**Базы данных**»
для студентов направления подготовки
02.03.01 «Математика и компьютерные науки»
направленность (профиль)
«Анализ данных и искусственный интеллект»

ВВЕДЕНИЕ

***Цель освоения дисциплины*:** формирование у студентов общепрофессиональных компетенций, основанных на освоении студентами знаний и умений в области информационных технологий для профессиональной деятельности, а также алгоритмизации и программирования, необходимых для применения технологий БД в их будущей профессиональной деятельности.

***Задачи освоения дисциплины*:**

- изучение основных характеристик баз данных,
- освоение языков запросов к реляционным базам данных,
- получение практических навыков использования современных СУБД,
- освоение приемов проектирования баз данных.

Дисциплина «Базы данных» относится к дисциплинам обязательной части.

Перечень планируемых результатов обучения по дисциплине, соотнесённых с планируемыми результатами освоения образовательной программы

Код, формулировка компетенции	**Код, формулировка индикатора**	**Планируемые результаты обучения по дисциплине, характеризующие этапы формирования компетенций, индикаторов**
ОПК-4. Способен находить, анализировать, реализовывать программно и использовать на практике математические алгоритмы, в том числе с применением современных вычислительных систем	**ИД-1$_{ОПК-4}$** разрабатывает и использует на практике новые математические алгоритмы	реализует этапы проектирования баз данных на концептуальном, физическом и логическом уровнях; владеет проектированием БД методом нормальных форм отношений; умеет строить семантическую модель сущность-связь (ER-диаграммы)
	ИД-2$_{ОПК-4}$ реализует программно новые математические алгоритмы с использованием современных вычислительных систем	разрабатывает SQL-запросы к серверу СУБД
	ИД-3$_{ОПК-4}$ проводит анализ разработанных и существующих алгоритмов с использованием современных вычислительных систем	применяет определения ограничений целостности, представлений БД, привилегий доступа к данным; реализует функции защиты данных
ОПК-5. Способен понимать принципы работы современных информационных технологий и использовать их для решения задач профессиональной деятельности	**ИД-1$_{ОПК-5}$** выбирает современные информационные технологии и программные средства, в том числе отечественного производства, для решения задач профессиональной деятельности	использует технологию физического хранения данных и доступа к ним; применяет файл-серверные и клиент-серверные платформы реляционных баз данных, в том числе отечественных разработчиков
	ИД-2$_{ОПК-5}$ демонстрирует способность использовать цифровые ресурсы для решения задач профессиональной деятельности	поддерживает жизненный цикл БД; понимает перспективы развития СУБД

ЛАБОРАТОРНАЯ РАБОТА 1.
Знакомство с возможностями консоли командной строки СУБД PostgreSQL

Продолжительность работы: 4 часа (+ 4 часа самостоятельной работы дома).

Теоретические сведения

База данных — это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе. База данных обычно управляется системой управления базами данных (СУБД). Данные вместе с СУБД, а также приложения, которые с ними связаны, называются системой баз данных, или, для краткости, просто базой данных.

PostgreSQL является одной из наиболее популярных систем управления базами данных. 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. Развитие postgresql началось еще в 1986 году, а в 1996 году проект был переименован в PostgreSQL, что отражало больший акцент на SQL. Текущей версией является версия 14. PostgreSQL поддерживается для всех основных операционных систем - Windows, Linux, MacOS. Голова слона является официальным логотипом этой базы.

- Официальный сайт проекта: https://www.postgresql.org/
- Документация по PostgreSQL на русском языке: https://postgrespro.ru/docs/
- PostgreSQL развивается как opensource. Исходный код проекта в репозитории по адресу:https://github.com/postgres/postgres.

PostgreSQL реализован в архитектуре клиент-сервер. Рабочий сеанс PostgreSQL включает следующие взаимодействующие процессы (программы):

• **Главный серверный процесс**, управляющий файлами баз данных, принимающий подключения клиентских приложений и выполняющий различные запросы клиентов к базам данных. Эта программа сервера БД называется **postgres**.

• **Клиентское приложение пользователя**, желающее выполнять операции в базе данных. Клиентские приложения могут быть очень разнообразными: это может быть текстовая утилита, графическое приложение, веб-сервер, использующий базу данных для отображения веб-страниц, или специализированный инструмент для обслуживания БД. Некоторые клиентские приложения поставляются в составе дистрибутива PostgreSQL, однако большинство создают сторонние разработчики.

Как и в других типичных клиент-серверных приложениях, клиент и сервер могут располагаться на разных компьютерах. В этом случае они взаимодействуют по сети TCP/IP. Важно не забывать это и понимать, что файлы, доступные на клиентском компьютере, могут быть недоступны (или доступны только под другим именем) на компьютере-сервере.

Сервер PostgreSQL может обслуживать одновременно несколько подключений клиентов. Для этого он запускает («порождает») отдельный процесс для каждого подключения. Можно сказать, что клиент и серверный процесс общаются, не затрагивая главный процесс **postgres**. Таким образом, главный серверный процесс

всегда работает и ожидает подключения клиентов, принимая которые, он организует взаимодействие клиента и отдельного серверного процесса.

Чтобы подключиться к серверу СУБД и выполнить какие-либо команды, требуется программа-клиент. Можно посылать запросы к серверу СУБД из программ на разных языках программирования, но в данной ЛР речь пойдет о **терминальном клиенте psql**, работа с которым происходит интерактивно в режиме командной строки.

SQL Shell (psql) – это консоль командной строки для Postgres. Почему имеет смысл научиться с ней работать? Во-первых, psql — стандартный клиент, он входит в любую сборку PostgreSQL и поэтому всегда под рукой. Во-вторых, psql действительно удобен для решения повседневных задач по администрированию баз данных, для написания небольших запросов и автоматизации процессов, например, для периодической установки изменений программного кода на сервер СУБД. Он имеет собственные команды, позволяющие сориентироваться в объектах, хранящихся в базе данных, и удобно представить информацию из таблиц.

Ход работы

1. С помощью учебного пособия:

- Моргунов, Е. П. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. — СПб.: БХВ-Петербург, 2018. — 336 с.: ил.
- *см. файл «Моргунов_Учебник по PostgreSQL.psd»,*
- *оно же по ссылке: https://static-ru.insales.ru/files/1/526/10019342/original/786624396.pdf?1566571893)*

ознакомьтесь с материалами следующих глав (стр. 13-32):

Глава 1. Введение в базы данных и SQL

Глава 2. Создание рабочей среды

2. Осуществите подключение к SQL Shell.

В ОС Windows запустите программу «SQL Shell (psql)» из меню «Пуск». В ответ на запрос введите пароль пользователя postgres, который вы указали при установке PostgreSQL*(в компьютерном классе вход без пароля)*.В итоге вы увидите приглашение:

«Postgres» здесь — имя базы данных, к которой вы сейчас подключены. Один сервер PostgreSQL может обслуживать несколько баз данных, но одновременно вы работаете только с одной из них.

Следует различать **команды языка SQL** и **команды утилиты psql.**

Все внутренние команды, не являющиеся операторами БД, начинаются с косой черты (обратного слеша) — «\». Команды, начинающиеся с символа «\», являются командами, которые утилита psql предлагает для удобства пользователя.

Кроме внутренних команд консоль позволяет вводить непосредственно команды SQL. Каждая из них должна заканчиваться символом «;», нажатие «ENTER» — лишь перенесет ввод на следующую строку.

Стоит отметить, что по умолчанию консоль в Windows поддерживает кодировку CP866, тогда как базы данных могут работать совсем с другой кодировкой, например,

1251. Пользователи Windows могут столкнуться с проблемой неправильного отображения символов русского языка в терминале. В этом случае убедитесь, что свойствах окна терминала установлен TrueТурешрифт (обычно «LucidaConsole» или «Consolas»). Чтобы сменить кодировку достаточно набрать:

\! chcp 1251, предварительно изменив шрифт на Lucida.

3. Используя консольный клиент psql выполните следующие действия:

1) создайте базу данных demo:

postgres=# CREATEDATABASEdemo;

Не забудьте про точку с запятой в конце команды — пока PostgreSQL не увидит этот символ, он будет считать, что вы продолжаете ввод (так что команду можно разбить на несколько строк).

На экране появится

CREATE DATABASE

2) Теперь переключимся на созданную базу:

postgres=# \cdemo

Должно получиться:

You are now connected to database " demo " as user "postgres".

demo =#

Как вы видите, приглашение сменилось на demo =#.

Команда, которую мы только что ввелиначинается с обратной косой черты.

Напомню, что так выглядят специальные команды, которые понимает только psql.Команд psql довольно много, и с некоторыми из них мы познакомимся чуть позже, а полный список с кратким описанием можно получить прямо сейчас:

\?

Поскольку справочная информация довольна объемна, она будет показана с помощью настроенной в операционной системе команды-пейджера (обычно more или less).Часто используемые команды консоли psql приведены в приложении к ЛР1.

3) **Откройте учебное пособие «Моргунов_Учебник по PostgreSQL.psd»** на стр. 32 и проделайте все задания из Главы 3: Основные операции с таблицами (стр. 32-47).

В процессе выполнения уражнений вы ознакомитесь с основными командами языка SQL, которые позволяют выполнять базовые операции.

Описание хода работы, т.е.текст SQL-запросов на создание БД и таблиц, запросы на заполнение таблиц, на просмотр их содержимого, запросы на операции с данными; результаты каждой команды и всех запросов - следует сохранять в файле отчета и проиллюстрировать скриншотами экрана.

4. Теперь подготовьте ответы на Контрольные вопросы со **стр. 48-49**.

5. Оформите отчет и защитите его у преподавателя.

В отчет включить:

- титульный лист;
- входные данные в виде схемы данных, созданной в ЛР1;
- ход работы (текст команд и результаты их выполнения проиллюстрировать скриншотами экрана);
- письменные ответы на контрольные вопросы.

Приложение к ЛР1

Часто используемые команды консоли psql

\? — Справка по командам psql
\h — Справка по SQL: список доступных команд или синтаксис конкретной команды
\l — Получить список баз данных
\du — список всех пользователей и их привилегий
\c *dbname*— выбрать базу данных
\dt — получить список всех таблиц
\dt+ — получить расширенный список таблиц с описанием
\dt *s* — список всех таблиц, содержащих s в имени
\d *table* — структура таблицы table
\x — переключает традиционный табличный вывод (столбцы и строки) на расширенный (каждый столбец на отдельной строке) и обратно. Удобно для просмотра нескольких «широких» строк.
\i FILE — выполнить команды из файла FILE
\o FILE — сохранить результат запроса в файл FILE
\a — переключение между режимами вывода: с/без выравнивания
\q – команда выхода из командной строки

Подробнее см. по ссылке
https://www.oslogic.ru/knowledge/598/shpargalka-po-osnovnym-komandam-postgresql/

ЛАБОРАТОРНАЯ РАБОТА 2.
Типы данных СУБД PostgreSQL

Продолжительность работы – 6 часов.

Ход работы

1. Ознакомьтесь с материалами Главы 4 Типы данных СУБД PostgreSQL (стр. 51-72) учебного пособия:
   - Моргунов, Е. П. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. — СПб.: БХВ-Петербург, 2018. — 336 с.: ил.
   - *см. файл «Моргунов_Учебник по PostgreSQL.psd»,*
   - *оно же по ссылке: https://static-ru.insales.ru/files/1/526/10019342/original/786624396.pdf?1566571893)*
2. Используя консольный клиент psql и БД demo из ЛР № 1 проделайте все задания из пункта Контрольные вопросы и задания к Главе 4 (стр. 72-93), кроме заданий со звездочкой*. Результаты каждой команды и всех запросов следует сохранять в файле отчета и проиллюстрировать скриншотами экрана.
3. Оформите отчет и защитите его у преподавателя.

   В отчет включить:
   - титульный лист;
   - ответы на контрольные вопросы с 1 по 37 (номер вопроса, задание, текст команд и результаты их выполнения);
   - таблицу стандартных представителей разных типов данных (в формате **Название типа данных** и его **Описание**): числовые, символьные, даты и времени, логический, перечисления, массивы.

Теоретические сведения

Типы данных — это одно из базовых понятий любого языка программирования, и язык SQL в этом плане не является исключением. В стандарте SQL описаны следующие типы (или их имена): bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (с часовым поясом и без), timestamp (с часовым поясом и без), xml.

При определении таблицы для всех ее столбцов необходимо указать тип данных. Тип данных определяет диапазон значений, которые могут храниться в столбце, сколько они будут занимать места в памяти.

PostgreSQL имеет очень разнообразный набор встроенных типов данных, т. е. тех типов, которые СУБД предоставляет в распоряжение пользователя по умолчанию. Их условно можно разделить на подгруппы: числовые, символьные, логические, дата и время, бинарные и ряд других, в рамках этих групп они имеют некоторые общие свойства, но также они имеют и различия.

Пользователь имеет возможность создавать и свои собственные типы данных, которые затем можно включить в систему и использовать их так же, как и встроенные, используя команду CREATE TYPE. Такая возможность адаптации системы типов данных к конкретным ситуациям является одной из отличительных черт PostgreSQL.

Каждый тип данных имеет внутреннее представление, скрытое функциями ввода и вывода. При этом многие встроенные типы стандартны и имеют очевидные внешние форматы. Однако есть типы, уникальные для PostgreSQL, например геометрические пути, и есть типы, которые могут иметь разные форматы, например, дата и время.

Рассмотрим некоторые типы данных СУБД PostgreSQL.

Числовые типы данных

- **serial**: представляет автоинкрементирующееся числовое значение, которое занимает 4 байта и может хранить числа от 1 до 2147483647. Значение данного типа образуется путем автоинкремента значения предыдущей строки. Поэтому, как правило, данный тип используется для определения идентификаторов строки.
- **smallserial**: представляет автоинкрементирующееся числовое значение, которое занимает 2 байта и может хранить числа от 1 до 32767. Аналог типа serial для небольших чисел.
- **bigserial**: представляет автоинкрементирующееся числовое значение, которое занимает 8 байт и может хранить числа от 1 до 9223372036854775807. Аналог типа serial для больших чисел.
- **smallint**: хранит числа от -32768 до +32767. Занимает 2 байта. Имеет псевдоним **int2.**
- **integer**: хранит числа от -2147483648 до +2147483647. Занимает 4 байта. Имеет псевдонимы **int** и **int4.**
- **bigint**: хранит числа от -9223372036854775808 до +9223372036854775807. Занимает 8 байт. Имеет псевдоним **int8.**
- **numeric**: хранит числа с фиксированной точностью, которые могут иметь до 131072 знаков в целой части и до 16383 знаков после запятой.
  Данный тип может принимать два параметра precision и scale: numeric(precision, scale).
  Параметр precision указывает на максимальное количество цифр, которые может хранить число.
  Параметр scale представляет максимальное количество цифр, которые может содержать число после запятой. Это значение должно находиться в диапазоне от 0 до значения параметра precision. По умолчанию оно равно 0.
  Например, для числа 23.5141 precision равно 6, а scale - 4.
- **decimal**: хранит числа с фиксированной точностью, которые могут иметь до 131072 знаков в целой части и до 16383 знаков в дробной части. То же самое, что и numeric.
- **real**: хранит числа с плавающей точкой из диапазона от 1E-37 до 1E+37. Занимает 4 байта. Имеет псевдоним float4.
- **double precision**: хранит числа с плавающей точкой из диапазона от 1E-307 до 1E+308. Занимает 8 байт. Имеет псевдоним float8.

Примеры использования:

```
1  Id SERIAL,
2  TotalWeight NUMERIC(9,2),
3  Age INTEGER,
4  Surplus REAL
```

Типы для работы с валютой (денежными единицами)

Для работы с денежными единицами определен тип **money**, который может принимать значения в диапазоне от -92233720368547758.08 до +92233720368547758.07 и занимает 8 байт.

Символьные типы

- **character(n)**: представляет строку из фиксированного количества символов. С помощью параметра задается задается количество символов в строке. Имеет псевдоним **char(n).**
- **character varying(n)**: представляет строку из переменной длины. С помощью параметра задается задается максимальное количество символов в строке. Имеет псевдоним **varchar(n).**
- **text**: представляет текст произвольной длины.

Бинарные данные

Для хранения бинарных данных определен тип **bytea**. Он хранит данные в виде бинарных строк, которые представляют последовательность октетов или байт.

Типы для работы с датами и временем

- **timestamp**: хранит дату и время. Занимает 8 байт. Для дат самое нижнее значение - 4713 г до н.э., самое верхнее значение - 294276 г н.э.
- **timestamp with time zone:** то же самое, что и timestamp, только добавляет данные о часовом поясе.
- **date**: представляет дату от 4713 г. до н.э. до 5874897 г н.э. Занимает 4 байта.
- **time**: хранит время с точностью до 1 микросекунды без указания часового пояса. Принимает значения от 00:00:00 до 24:00:00. Занимает 8 байт.
- **time with time zone**: хранит время с точностью до 1 микросекунды с указанием часового пояса. Принимает значения от 00:00:00+1459 до 24:00:00-1459. Занимает 12 байт.
- **interval**: представляет временной интервал. Занимает 16 байт.

Распространенные форматы дат:

- yyyy-mm-dd - 1999-01-08
- Month dd, yyyy - January 8, 1999
- mm/dd/yyyy - 1/8/1999

Распространенные форматы времени:

- hh:mi - 13:21
- hh:mi am/pm - 1:21 pm
- hh:mi:ss - 1:21:34

Логический тип

Тип **boolean** может хранить одно из двух значений: true или false.

Вместо true можно указывать следующие значения: TRUE, 't', 'true', 'y', 'yes', 'on', '1'.

Вместо false можно указывать следующие значения: FALSE, 'f', 'false', 'n', 'no', 'off', '0'.

Типы для представления интернет-адресов

- **cidr**: интернет-адрес в формате IPv4 и IPv6. Например, 192.168.0.1. Занимает от 7 до 19 байт.

- **inet**: интернет-адрес в формате cidr/y, где cidr это адрес в формате IPv4 или IPv6, а /y - количество бит в адресе (если этот параметр не указан, то используется 34 для IPv4, 128 для IPv6).
  Например, 192.168.0.1/24 или 2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128. Занимает от 7 до 19 байт.
- **macaddr**: хранит MAC-адрес. Занимает 6 байт.
- **macaddr8**: хранит MAC-адрес в формате EUI-64. Занимает 8 байт.

Геометрические типы

- **point**: представляет точку на плоскости в формате (x,y). Занимает 16 байт.
- **line**: представляет линию неопределенной длины в формате {A,B,C}. Занимает 32 байта.
- **lseg**: представляет отрезок в формате ((x1,y1),(x2,y2)). Занимает 32 байта.
- **box**: представляет прямоугольник в формате ((x1,y1),(x2,y2)). Занимает 32 байта.
- **path**: представляет набор соединенных точек. В формате ((x1,y1),...) путь является закрытым (первая и последняя точка соединяются линией) и фактически представляет многоугольник. В формате [(x1,y1),...] путь является открытым Занимает 16+16n байт.
- **polygon**: представляет многоугольник в формате ((x1,y1),...). Занимает 40+16n байт.
- **circle**: представляет окружность в формате <(x,y),r>. Занимает 24 байта.

Остальные типы данных

- **json**: хранит данные json в текстовом виде.
- **jsonb**: хранит данные json в бинарном формате.
- **uuid**: хранит универсальный уникальный идентификатор (UUID), например, a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11. Занимает 32 байта.
- **xml**: хранит даные в формате XML.

Примеры

Числовые типы

Имя	Размер	Описание	Диапазон	Пример
integer	2 байта	Отрицательное или положительное целое число.	от -2147483648 до +2147483647	516
numeric(m, n)	перемен ный	Вещественное число с фиксированной точностью. Точность для этого типа настраиваемая, это значит, что её можно указать с помощью точности (m) и масштаба (n).	131072 цифр до точки 16383 цифр после точки	12,12958

		Точность – общее количество цифр, масштаб – количество цифр после запятой. Например число 468,45 – NUMERIC(5, 2). Можно задать только точность, например NUMERIC(3), тогда масштаб станет равен 0. На самом деле точность и масштаб можно не задавать. В таком случае можно будет сохранить число с любой точностью и масштабом в рамках ограничений. Этот тип применяется для чисел, где важна точность, например для хранения денежных сумм.		
real	4 байта	Число с плавающей точкой. Как следствие точность меньше чем у **numeric**, зато вычисления происходят быстрее.	6 цифр после точки	56,13
serial	4 байта	Целое число из последовательности, то есть оно автоматически увеличивается для каждой новой строки. Проще говоря, создавая этот тип для столбца, вам не нужно	от 1 до 2147483647	50

		придумывать значения в нём. Так как за вас это сделает система. Этот тип гарантирует уникальность в рамках диапазона.		
money	8 байт	Тип для хранения денежной суммы.	от -92233720368547758.08 до +9223372036 8547758.07	100,50

CREATE TABLE test
(integer integer,
numeric numeric(4,3),
real real,
serial serial,
money money);

INSERT INTO test (integer, numeric, real, money) **VALUES**
(123, 1.343, 1233.2, 245.65);
INSERT 0 1

SELECT * **FROM** test;
integer | numeric | real | serial | money
---------+---------+--------+--------+----------
123 | 1.343 | 1233.2 | 1 | 245,65 □
(1 row)

Из вывода выше стоит отметить, что в примере выше мы не вставляли значение в колонку serial, система это проделала за нас. А также в колонке money при выводе мы видим знак денежной единицы.

Строки

Имя	**Описание**	**Пример**
varchar(n)	Строка переменной длины, но не длиннее “n”.	abc
char(n)	Строка фиксированной длины, длинной “n”.	abc

text	Строка неограниченной и переменной длины.	abcde

CREATE TABLE text
(varchar varchar(5),
char char(3),
text text);
INSERT INTO text (varchar, char, text) **VALUES**
('abc', 'abc', 'hello word'),
('abcde', 'abc', 'by-by');

SELECT * **FROM** text;
varchar | char | text
---------+------+------------
abc | abc | hello word
abcde | abc | by-by
(2 rows)

В результате в **varchar** можно записать любую строку не более 5 символов, в **char** строка должна быть ровно 3 символа, а в **text** можем записать строку любой длины.

Дата и время

Имя	**Размер**	**Описание**	**Диапазон**	**Точность**	**Пример**
timestamp	8 байт	Дата и время	от 4713 года до нашей эры до 294276 года нашей эры	1 микросекунда	2021-07-13 14:21:41
timestamptz	8 байт	Дата и время с часовым поясом	от 4713 года до нашей эры до 294276 года нашей эры	1 микросекунда	2021-07-13 14:21:41 +03
date	4 байта	Дата год-месяц-число	от 4713 года до нашей эры до 5874897 года нашей эры	1 день	2021-07-13
time	8 байт	Время	от 00:00:00 до 24:00:00	1 микросекунда	14:21:41

CREATE TABLE time

(timestamp timestamp,
timestamptz timestamptz,
date date,
time time**);**

SELECT now();
now

2021-07-13 14:21:41.406081+03

INSERT INTO time (timestamp, timestamptz, date, time) VALUES
('2021-07-13 14:21:41', '2021-07-13 14:21:41 +03', '2021-07-13', '14:21:41'**);**

INSERT INTO time (timestamp, timestamptz, date, time) VALUES
(now**()**, now**()**, now**()**, now**())**;

SELECT * FROM time;
timestamp | timestamptz | date | time
---+---+---+---
2021-07-13 14:21:41 | 2021-07-13 14:21:41+03 | 2021-07-13 | 14:21:41
2021-07-13 14:29:43.235957 | 2021-07-13 14:29:43.235957+03 | 2021-07-13 | 14:29:43.235957
(3 rows**)**

Функция **now()** возвращает текущее время в формате **timestamptz**, но помещая вывод этой функции в различные типы данных, возвращаемое значение подстраивается под конкретный тип данных.

Логический тип

Имя	**Размер**	**Описание**	**Диапазон**
boolean	1 байт	Истина или ложь	true или false

CREATE TABLE boolean
(name varchar(15),
date date,
attended boolean);

INSERT INTO boolean (name, date, attended) **VALUES**
('alex', '2021-06-15', true),
('bob', '2021-06-15', false);

SELECT * **FROM** boolean;
name | date | attended
---+---+---

```
alex | 2021-06-15 | t
bob | 2021-06-15 | f
(2 rows)
```

В примере выше в колонке **attended** (присутствовал) мы указываем:

- **true** – если человек в эту дату присутствовал где-то;
- **false** – если не присутствовал.

Существуют и другие типы, например для хранения двоичных данных, json, массивов, xml и другой информации.

Полная информация о типах данных в главе 8 документации: https://postgrespro.ru/docs/postgresql/14/datatype

ЛАБОРАТОРНАЯ РАБОТА 3.
SQL: основы языка определения данных

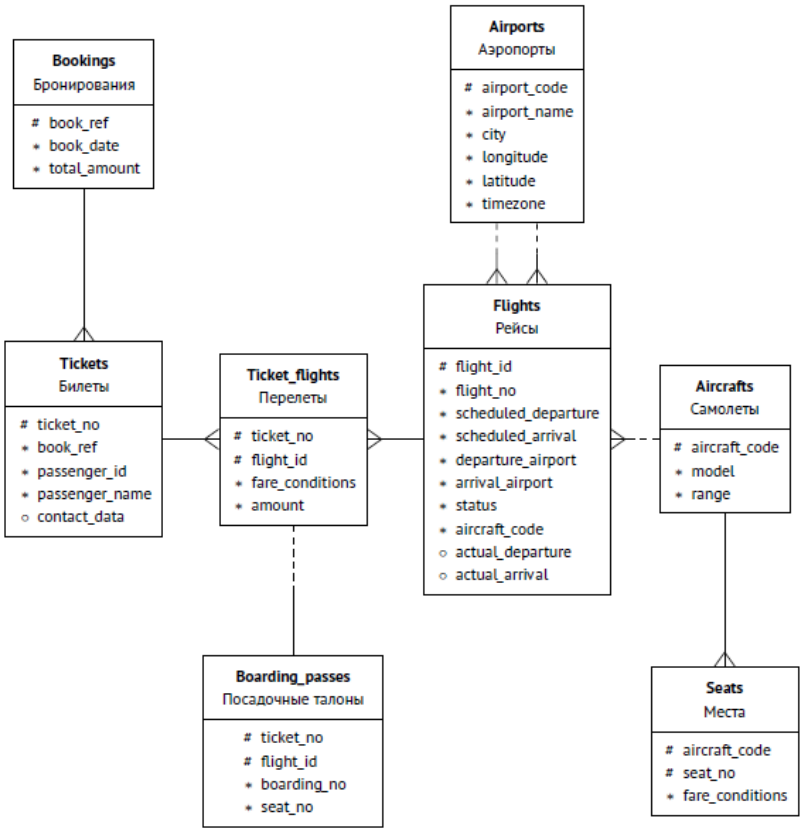
Продолжительность работы – 6 часов.
Основные понятия темы: операторы CREATE, ALTER и DROP.

Ход работы

1. Ознакомьтесь с материалами Главы 5 Основы языка определения данных (стр. 95-144) учебного пособия:

- Моргунов, Е. П. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. — СПб.: БХВ-Петербург, 2018. — 336 с.: ил.
- *см. файл «Моргунов_Учебник по PostgreSQL.psd»,*
- *оно же по ссылке: https://static-ru.insales.ru/files/1/526/10019342/original/786624396.pdf?1566571893)*

2. Используя консольный клиент psql **проделайте все упражнения из пунктов 5.2, 5.3, 5.4 и 5.5 со стр. 105-153** над таблицами БД «Авиаперевозки», созданными в ЛР1 и ЛР2.
Для этого вспомните описание предметной области и схему данных (стр.19-22 пособия).



3. **Выполните 18 заданий из пункта Контрольные вопросы и задания к Главе 5 (**стр. 133-144), кроме заданий со звездочкой* (их можно выполнить по желанию). Результаты каждой команды и всех запросов следует сохранять в файле отчета и проиллюстрировать скриншотами экрана.
4. Оформите отчет и защитите его у преподавателя.

В отчет включить:

- титульный лист;
- примеры из упражнений, иллюстрирующие **основные понятия**: схема базы данных, суррогатный ключ, уникальный ключ, значение по умолчанию, ограничение, внешний ключ, команда удаления таблицы, команды модификации таблицы команды создания и удаления представления, материализованное представление; смена текущей схемы базы данных, особенности выполнения операторов CREATE, ALTER и DROP)
- **ответы на контрольные вопросы** с 1 по 18 (номер вопроса, задание, текст команд и результаты их выполнения).

Теоретические сведения

Подмножество языка SQL, предназначенное для описания данных (SQL DDL), включает операторы CREATE, ALTER и DROP, а также GRANT и REVOKE, используемые для управления разграничением доступа и рассматриваемые в главе 5.

За ключевым словом, определяющим оператор, следует другое ключевое слово, определяющее тип объекта, к которому применяется оператор,—например CREATE TABLE является оператором создания таблиц. Далее обычно следует имя объекта базы данных и затем предложения, уточняющие, что именно делается при выполнении оператора. В этой ЛР рассматривается только работа с таблицами на уровне логической структуры, а управление структурами хранения обсуждается далее в других разделах курса.

В системе PostgreSQL используется термин «отношение», однако он означает не отношения в смысле теоретической реляционной модели, а понятие, включающее таблицы и представления в смысле модели данных SQL, а также некоторые другие виды объектов базы данных.

Простейшая форма оператора CREATE TABLE содержит имя создаваемой таблицы, за которым следует список определений атрибутов (колонок), разделяемых запятыми. Определение каждого атрибута содержит его наименование, тип и может содержать некоторые дополнительные указания, в частности ограничения целостности и значение, присваиваемое этому атрибуту при добавлении строки в таблицу, если оно не указано явно.

Например, таблица дисциплин может быть создана с помощью следующего оператора:

```
demo=# CREATE TABLE courses (
course_no varchar(30),
title
text,
credits
integer
);

CREATE TABLE
```

Последняя строка представляет собой ответ системы, подтверждающий выполнение оператора, и не является частью оператора.

Кроме определений атрибутов, список в команде CREATE TABLE может содержать ограничения целостности. Определение таблицы courses, приведенное выше, содержит не всю необходимую информацию. Более полный вариант оператора создания таблицы содержит определение первичного ключа и требование отсутствия неопределенных значений для атрибута, содержащего название курса:

```
demo=# CREATE TABLE courses (
course_no varchar(30),
title
text NOT NULL,
credits
integer,
CONSTRAINT course_pkey PRIMARY KEY (course_no)
);

CREATE TABLE
```

Заметим, что имена таблиц должны быть различны, поэтому выполнить оператор создания таблицы можно, только если таблицы с таким именем еще не существует.

Оператор DROP удаляет объект базы данных, заданный параметрами этого оператора. Например, для удаления таблицы courses можно было бы использовать следующий оператор:

```
demo=# DROP TABLE courses;

DROP TABLE
```

Оператор ALTER изменяет структуру или свойства существующего объекта. В частности, оператор ALTER TABLE можно использовать для добавления, переименования или удаления отдельных атрибутов таблицы, а также для добавления или удаления ограничений целостности.

В языке SQL предусмотрены следующие ограничения целостности:

- NOT NULL запрещает появление неопределенных значений атрибута.
- UNIQUE задает группу атрибутов, которые образуют возможный ключ отношения, т. е. для любых двух строк таблицы значения хотя бы одного из атрибутов, входящих в возможный ключ, должны различаться.

Очень часто такое ограничение состоит всего из одного атрибута.

PRIMARY KEY задает атрибут или группу атрибутов, которые используются как первичный ключ отношения. Обычно такое ограничение определяется для атрибутов, составляющих идентификатор сущности, и поэтому обычно эти атрибуты считаются неизменяемыми, однако язык SQL допускает изменение значений атрибутов первичного ключа. Очень часто первичный ключ состоит только из одного атрибута, хотя язык SQL позволяет создавать составные первичные ключи.

Важное отличие уникального ключа от первичного состоит в том, что для атрибутов, входящих в уникальный ключ, могут допускаться неопределенные значения (NULL), а для атрибутов первичного ключа — нет.

FOREIGN KEY задает группу атрибутов, значения которых должны совпадать созначениями атрибутов первичного или уникального ключа другого отношения, указанных в этом ограничении. Таким образом, это ограничение задает бинарные связи между сущностями.

CHECK задает произвольное условие на значения одного или нескольких атрибутов в одной строке таблицы.

Невозможность выполнения операторов DDL вызывает индикацию ошибки: например, попытка создать таблицу с именем, совпадающим с именем существующей таблицы в той же схеме, или попытка удаления несуществующего объекта. В системе PostgreSQL имеются условные формы операторов языка описания данных, проверяющие возможность выполнения.

Например, оператор

```
DROP TABLE IF EXISTS old_table;
```

удалит таблицу old_table, если таблица с таким именем существовала, и ничего не сделает в противоположном случае. Для оператора CREATE условие записывается как IF NOT EXISTS, поскольку выполнение этого оператора возможно, если объект не существует.

В программе psql можно получить список доступных таблиц, используя команду \d или \dt. Команда с указанием имени таблицы выводит ее описание.

Заполнение таблиц

Добавление новых данных производится оператором INSERT, содержащим два предложения. Первое из них (INTO) указывает, куда помещаются новые данные, и содержит имя таблицы, за которым может следовать список атрибутов, заключенный в круглые скобки. Второе предложение (VALUES) задает значения, которые добавляются в базу данных. В этом разделе мы используем только одну форму оператора INSERT, которая добавляет в базу данных одну строку. Значения атрибутов перечисляются в круглых скобках вслед за ключевым словом VALUES.

Следующие операторы заносят в базу данных строки созданной ранее таблицы courses:

```
demo=# INSERT INTO courses (course_no, title, credits)
VALUES ('CS301', 'Базы данных', 5);

INSERT 0 1

demo=# INSERT INTO courses (course_no, credits, title)
VALUES ('CS305', 10, 'Анализ данных');

INSERT 0 1
```

Ответ системы (второе число) показывает количество строк в таблице, которые были изменены, в данном случае — добавлены.

Если в предложении INTO указан список атрибутов, то и значения в предложении VALUES должны идти в том же порядке. В этом случае не важно, в каком порядке расположены атрибуты в определении таблицы.

Если в предложении INTO перечислены не все атрибуты, имеющиеся в таблице, то остальные получают значения по умолчанию, указанные в описании таблицы, или неопределенные значения (NULL), если значения в описании таблицы не заданы. В случае если список атрибутов в предложении INTO не указан, порядок значений в предложении VALUES должен совпадать с порядком в определении таблицы. Использование такой формы оператора INSERT делает код приложения зависимым от схемы базы данных: любое добавление или изменение порядка атрибутов потребует модификации кода приложения.

С другой стороны, если в коде приложений используется формат оператора INSERT, содержащий список атрибутов, то при добавлении новых атрибутов необходимо либо разрешать неопределенные значения, либо определять значения, присваиваемые по умолчанию, если приложение их не задает.

Если при выполнении оператора INSERT нарушаются ограничения целостности, то его выполнение заканчивается с ошибкой. В системе PostgreSQL в предложении ON CONFLICT можно указать действия, которые будут выполняться вместо индикации ошибки при нарушении ограничений уникальности (в том числе уникальности первичного ключа). Например, при попытке повторной вставки записи, в которой значение первичного или уникального ключа совпадает с уже имеющимся в таблице, можно вместо вставки выполнить изменение значений других атрибутов.

Обычно СУБД предоставляют средства для массовой загрузки данных. Это можно делать с помощью другой формы оператора INSERT, позволяющей включить данные, выбираемые из других таблиц (в этой форме вместо предложения VALUES записывается оператор SELECT).

В PostgreSQL предложение VALUES может содержать данные для нескольких строк таблицы. Кроме этого, в системе PostgreSQL имеется оператор COPY (не входящий в стандарт SQL), который может копировать данные из внешних файлов, находящихся в файловой системе того компьютера, на котором выполняется сервер базы данных, или наоборот — из базы данных во внешние файлы.

Модификация данных

Оператор UPDATE изменяет значения атрибутов в хранимой таблице. За ключевым словом UPDATE следуют имя таблицы и предложение SET, содержащее список имен изменяемых атрибутов из этой таблицы с указанием выражений, задающих для них новые значения. Далее следует предложение WHERE (такое же, как в операторе SELECT), которое определяет, какие строки будут обрабатываться данным оператором. Если предложение WHERE не указано, изменению подвергаются все строки таблицы.

Оператор UPDATE часто используется для изменения только одной строки таблицы, хотя SQL допускает указание условий, которым удовлетворяет несколькострок. Отметим также, что для изменения значения не требуется его предварительное считывание в приложение оператором SELECT: выражение, вычисляющее новое значение атрибута, может содержать имя этого атрибута, при

этом для вычисления нового значения будет использоваться старое. Новое значение может задаваться подзапросом, эта возможность более детально обсуждается ниже.

Например, оператор

```
demo=# UPDATE courses SET credits = credits + 1 WHERE course_no = 'CS305';

UPDATE 1
```

увеличивает на единицу значение одного атрибута в одной строке.

Оператор DELETE удаляет строки из хранимой таблицы, имя которой указывается в предложении FROM, а строки, подлежащие удалению, задаются предложением WHERE. Как и для операторов SELECT и UPDATE, отсутствие предложения WHERE означает, что должны быть обработаны (т. е. удалены) все строки таблицы. Так же как и оператор UPDATE, чаще всего оператор DELETE используется для удаления только одной строки.

Так, оператор

```
demo=# DELETE FROM courses WHERE course_no = 'CS305';

DELETE 1
```

удаляет одну строку таблицы (так как условие задает значение первичного ключа).

ЛАБОРАТОРНАЯ РАБОТА 4.
Запросы на языке SQL

Продолжительность работы – 8 часов.

Основные понятия темы: оператор SELECT, подзапросы, вложенные подзапросы, FROM, HAVING, рекурсивное табличное выражение, материализованное представление.

Ход работы

1. Ознакомьтесь с материалами Главы 6 Запросы (стр. 145-192) учебного пособия:
   - Моргунов, Е. П. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. — СПб.: БХВ-Петербург, 2018. — 336 с.: ил.
   - *или см. файл «Моргунов_Учебник по PostgreSQL.psd»,*
   - *или по ссылке: https://static-ru.insales.ru/files/1/526/10019342/original/786624396.pdf?1566571893)*
2. Используя таблицы базы данных «Авиаперевозки» (см. файл **demo_small.sql**) и консольный клиент psql исследуйте дополнительные возможности команды SELECT. Для этого проделайте **все упражнения из пунктов 6.1, 6.2, 6.3 со стр. 145-170.** Материал, касающийся *оконных функций (window functions)* на стр. 170-175 можно выполнить по желанию.
3. Изучите схему работы оператора SELECT (в пункете 6.4. Подзапросы), а затем выполните **упражнения со стр. 176-192**.
4. Выполните 18 заданий из пункта Контрольные вопросы и задания к Главе 6 (стр. 193-210), кроме заданий со звездочкой* (их можно выполнить по желанию). Результаты каждой команды и всех запросов следует сохранять в файле отчета и проиллюстрировать скриншотами экрана.
5. Оформите отчет и защитите его у преподавателя.

В отчет включить:

- титульный лист;
- примеры из упражнений, иллюстрирующие **основные понятия:**
- **ответы на контрольные вопросы** с 1 по 24 (номер вопроса, задание, текст команд и результаты их выполнения).

Теоретические сведения

4.1. Запросы в PostgreSQL

Рассмотрим, как извлекать нужные данные из БД. Процесс извлечения или команда извлечения данных из БД, называется *запросом*. В SQL для выполнения запроса используется команда SELECT. Команда SELECT имеет синтаксис:

[WITH *с_запросами*] SELECT *список_выборки* FROM *табличное_выражение* [*спецификация_сортировки*]

В следующих подразделах подробно описывается список выборки, табличное выражение и спецификация сортировки. Запросы с WITH обсуждаются в конце, так как они являются дополнительной возможностью SQL.
Простейший запрос выглядит так:
SELECT * FROM table1;
Эта команда извлекает из таблицы table1, все строки и все колонки. (Метод извлечения зависит от клиентского приложения. Например, программа psql покажет на экране таблицу, обрамлённую ASCII символами, в то время как клиентские библиотеки предложат функции для извлечения отдельных значений из результатов запроса.) Список выборки * означает все колонки, из представленного табличного выражения. Список выборки также может задавать подсписок доступных колонок или выполнять вычисления с использованием колонок. Например, если табдица table1 имеет колнки с именами a, b и c (и возможно ещё и другие) вы можете создать следующий запрос:
SELECT a, b + c FROM table1;
(предполагая, что b и c имеют числовой тип данных).
FROM table1 это простой вид табличного выражения: оно соответствует только одной таблице. Но табличные выражения могут быть и сложными конструкциями, которые базируются на таблицах, соединениях и подзапросах. Но вы также можете полностью опустить табличное выражение и использовать команду SELECT как калькулятор:
SELECT 3 * 4;
Такой способ полезен, если выражения в списке выбора возвращаеют разные результаты. Например, вы можете таким способом вызвать функцию:
SELECT random();

4.2. Табличные выражения
Табличное выражение предоставляет некую таблицу. Табличное выражение содержит предложение FROM, за которым необязательно следуют предложения WHERE, GROUP BY и HAVING. Простое табличное выражение просто указывает на какую-либо таблицу на диске, так называемую базовую таблицу, но для модификации и комбинирования базовых таблиц различными способами могут быть использованы более сложные выражения.
Необязательные предложения WHERE, GROUP BY и HAVING в табличном выражении, задают конвеер последовательных преобразований, выполняемых над таблицей, полученной из предложения FROM. Все эти преобразования создают виртуальную таблицу, которая предоставляет строки, которые затем передаются в список выборки для отбора строк на выходе запроса.
4.2.1. Предложение FROM
FROM Clause создаёт таблицу из одной или более других таблиц, которые задаются списком разделённых запятыми ссылок на таблицы:
FROM *ссылка_на_таблицу* [, *ссылка_на_таблицу* [, ...]]
Ссылка на таблицу может быть именем таблицы (возможно с указанием схемы) или производной таблицей, такой как подзапрос, соединием таблиц, или сложной комбинацией из них. Если в списке предложения FROM указано более одной ссылки на таблицу, они просто соединяются (см. ниже) в форму промежуточной виртуальной

таблицы, которая может затем быть преобразована с помощью предложений WHERE, GROUP BY, и HAVING, и в конце-концов стать результатом всего табличного выражения.

Когда ссылка на таблицу является родительской таблицей в иерархии наследования, данная ссылка на таблицу выдаёт строки не только этой таблицы, но и всех её таблиц-потомков, если перед именем таблицы не указано ключевое слово ONLY. Однако, ссылка на такую таблицу выдаёт только колонки, которые есть в этой таблице; любые другие колонки в таблицах-потомках игнорируются.

4.2.1.1. Соединённые таблицы

Соединённая таблица — это таблица, производная от двух других (реальных или в свою очередь производных) таблиц, полученая в соответствии с правилами определённого типа соединения. Доступные типы соединения: inner (внутреннее), outer (внешнее) и cross (перекрёсное).

Типы соединений

Перекрёстное соединение (cross join)

T1 CROSS JOIN *T2*

Для каждой комбинации строк из *T1* и *T2* (декартово произведение) соединённая таблица будет содержать строки, состоящие из всех колонок таблицы *T1*, за которым следуют все колонки из таблицы *T2*. Если таблицы имеют соответственно N и M строк, соединённая таблица будет иметь N * M строк.

FROM *T1* CROSS JOIN *T2* эквивалентно FROM *T1*, *T2*. Также это эквивалентно FROM *T1* INNER JOIN *T2* ON TRUE (см. ниже).

Квалифицированные соединения

T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN *T2* ON *логическое_выражение*

T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN *T2* USING (*список_колонок_для_соединения*)

T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN *T2*

Слова INNER и OUTER являются необязательными во всех формах. INNER применяется по умолчанию; LEFT, RIGHT и FULL неявно указывают на внешнее (outer) соединение.

Условие соединения задаётся в предложениях ON или USING, или неявно, с помощью слова NATURAL. Условие соединения определяет какие строки из двух исходных таблиц будут считаться "совпавшими", как ниже описывается в деталях.

Предложение ON является наиболее обычным способом задать условие соединения: в нём указывается логическое выражение того же типа, что используется в предложении WHERE. Пара строк из *T1* и *T2* считается совпавшей, если для неё, выражение, заданное в ON принимает значение истина.

USING имеет более краткую форму: в ней указывается разделённый запятыми список имён колонок, которые должны быть общими в соединяемых таблицах и условия соединения, задавая равенство каждой из этих пар колонок. Таким образом, вывод JOIN USING будет состоять из одной колонки для каждой, являющейся равной, пары входных колонок, за которыми следует остальные колонки из каждой таблицы. Так, USING (a, b, c) эквивалентно ON (t1.a = t2.a

AND t1.b = t2.b AND t1.c = t2.c) с тем исключением, что если ON используется там, где в результате будут две колонки a, b и c, то USING там, где будет только одна из них (и они будут выданы первыми, если используется SELECT *).

Наконец, NATURAL — это краткая форма USING: она формирует список USING, содержащий все имена колонок, которые есть в обеих входных таблицах. Как и с USING, эти колонки появляются в выходной таблице только один раз. Если общих колонок нет, NATURAL ведёт себя как CROSS JOIN.

Возможные типы квалифицированных соединений:

INNER JOIN

Для каждой строки R1 из таблицы T1, соединённая таблица будет иметь строку для каждой строки в T2, которая удовлетворяет условию соединения с R1.

LEFT OUTER JOIN

Сперва выполняется INNER JOIN. Затем, для каждой строки в T1, которая не удовлетворяет условию соединения с любой строкой в T2, добавляется соединённая строка с значениями NULL в колонках T2. Таким образом, соединённая таблица всегда имеет по крайней мере одну строку для каждой строки из T1.

RIGHT OUTER JOIN

Сперва выполняется INNER JOIN. Затем, для каждой строки в T2, которая не удовлетворяет условию соединения с любой строкой в T1, добавляется соединенная строка с значениями NULL в колонках T1. Этот тип соединения является обратным по отношению к LEFT JOIN: результирующая таблица всегда будет иметь хотя бы одну строку для каждой строки из T2.

FULL OUTER JOIN

Сперва выполняется INNER JOIN. Затем, для каждой строки в T1, которая не удовлетворяет условию соединения с любой строкой в T2, добавляется соединенная строка с значениями NULL в колонках T2. Также, для каждой строки T2, которая не удовлетворяет условию соединения с любой строкой в T1, добавляется соединённая строка с значениями NULL в колонках T1.

Соединения всех типов могуть быть сцеплены вместе или скомпанованы: одна или обе таблицы *T1* и *T2* могут быть соединяющимися таблицами. Для управления порядком выполнения соединений, вокруг JOIN могут использоваться круглые скобки. В отсутствие скобок, предложения JOIN компануются слева направо.

Чтобы проиллюстрировать всё вышеизложенное, предположим, что у нас есть две таблицы: t1:

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

и t2:

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

далее мы выполним разные виды соединений:

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
```

```
  1 | a    | xxx
  2 | b    |
  3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

Условие соединения, которое задаётся в ON, также может содержать условия, которые напрямую не относятся к соединению. Это может оказаться полезным для некоторых запросов, но нуждается в осторожном использовании. Например:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

Обратите внимание, что если поместить ограничение в предложение WHERE, то вы получите другой результат:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

Это потому, что ограничение помещённое в предложение ON обрабатывается *перед* соединением, в то время как ограничение, помещённое в предложение WHERE обрабатывается *после* соединения.

4.2.1.2. Псевдонимы таблиц и колонок

В сложных ссылках на таблицы, для дальнейших ссылок в остальной части запроса, могут быть использованы временные имена. Они называются *псевдонимами таблиц*.

Чтобы создать псевдоним таблицы, пишите

FROM *ссылка_на_таблицу* AS *псевдоним*

или

FROM *ссылка_на_таблицу псевдоним*
Ключевое слово AS является необязательным. *Псевдоним* может быть любым идентификатором.
Типичное применение псевдонимов таблиц состоит в назначении длинным именам таблиц коротких идентификаторов, чтобы сделать последующие предложения соединений более читабельными. Например:
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num;
Псевдоним даёт новое имя ссылающееся на ту же таблицу в текущем запросе — поэтому где-либо ещё в текущем запросе не разрешается ссылаться на таблицу по первоначальному имени. Таким образом, вот это неправильно:
SELECT * FROM my_table AS m WHERE my_table.a > 5; -- ошибка
Псевдонимы таблиц, в основном, используются для целей сокращения и лучшей читабельности текста запроса, но они бывают необходимы, когда необходим выполнить соединение таблицы с собой же, например:
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
Кроме того, псевдноним также требуется, если ссылка на таблицу является подзапросом (см. Section 4.2.1.3).
Скобки используются для того, чтобы избежать путаницы. В следующем примере, первый оператор назначает псевдоним b для второй таблицы my_table, а второй оператор назначает псевдоним результату соединения:
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
Другая форма псевдонимов таблиц даёт временные имена колонкам таблицы, точно также как и для самой таблицы:
FROM *ссылка_на_таблицу* [AS] *псевдоним* (*колонка1* [, *колонка2* [, ...]])
Вначале, указываются псевдонимы колонок, затем обычые колонки таблицы, оставшиеся колонки не переименовываются. Такой синтаксис особенно полезен для подзапросов соединений таблиц с самими собой.
Когда к результатам предложения JOIN применяется псевдоним, то этот псевдоним скрывает первоначальные имена внутри JOIN. Например:
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
является правильным SQL запросом, но:
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
неправильный запрос: псевдоним таблицы a не виден за пределами псевдонима c.
4.2.1.3. Подзапросы
Подзапросы, предоставляющие на выходе таблицу, должны быть заключены в круглые скобки и им *должен* быть назначен псевдоним таблицы. (См. Section 4.2.1.2.)
Например:
FROM (SELECT * FROM table1) AS alias_name
Этот пример эквивалентен FROM table1 AS alias_name. Когда подзапрос выполняет группировку или агрегирование, возникают более интересные случаи, которые не могут быть представлены как простое соединение таблиц.
Подзапрос также можте быть списком VALUES:
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
    AS names(first, last)

И снова требуется псевдоним таблицы. Назначение имён псевдонимов для колонок списка VALUES необзяталеьно, но является хорошим тоном. Подробности см. в Section 4.7.

4.2.1.4. Табличные функции

Табличные функции — это функции, которые возвращают список колонок либо базовых типов данных (скалярные типы), либо составных типов данных (колонки таблиц). Они используются в предложении FROM как обычные таблицы, представления или подзапросы. Колонки, возвращаемые табличными функциями могут быть включены в предложения SELECT, JOIN или WHERE точно таким же образом как и колонки таблицы, представления или подзапроса.

Если табличная функция возвращает базовый тип данных, то имя результирующей одиночной колонки совпадает с именем этой функции. Если функция возвращает составной тип, колонки результата получают такие же имена как отдельные атрибуты данного типа.

В предложении FROM табличная функция может получить псевдоним, но также может остаться без псевдонима. Если функция используется в предложении FROM без псевдонима, то в качестве имени таблицы-результата используется имя этой функции.

Несколько примеров:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

В некоторых случаях, полезно определить табличные функции, которые могут возвращать различные списки колонок, в зависимости от того как они вызываются. Чтобы реализовать такое поведение, табличная функция может быть объявлена как возвращающая псевдотип record. Когда такая функция используется в запросе, ожидаемая структура строки должна быть задана в самом запросе, так что система может знать как обработать и спланировать запрос. Рассмотрим пример:

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

Функция dblink (часть (модуля dblink) выполняет удалённый запрос. Она объявляется как возвращающая record и, таким образом, может быть использована для любых типов запроса. Фактический список колонок должен быть указан при вызове запроса, так что он известен обработчику запроса, например * должно быть развёрнуто в список.

4.2.2. Предложение WHERE

Синтаксис *WHERE Clause* следующий:

WHERE *условие_поиска*

где *условие поиска* является любым значением выражения (см. Section 4.2), которое возвращает значение логического типа boolean.

После того как выполнена обработка предложения FROM, каждая строка, полученной в итоге производной вируальной таблицы, проверяется согласно условию поиска. Если результатом этого условия является истина, то строка оставляется в виртуальной таблице, в противном случае (например, если результатом является ложь или NULL), строка отбрасывается. Условие поиска обычно использует по крайней мере одну колонку из таблицы, полученой в предложении FROM; этого не требуется, но в противном случае предложение WHERE будет фактически бесполезным.

Note: Условие соединения INNER JOIN модет быть написано либо в предложении WHERE, либо в предложении JOIN. Например, эти табличные выражения эквивалентны:

FROM a, b WHERE a.id = b.id AND b.val > 5

и:

FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5

или возможно даже:

FROM a NATURAL JOIN b WHERE b.val > 5

Какой из вышеперечисленных запросов использовать вам, в основном вопрос стиля. Синтаксис JOIN в предложении FROM, предположительно, не будет решением, переносимым на другие SQL СУБД, даже если они работают по стандарту SQL. Для OUTER JOIN в этом случае выбора нет: такие соединения должны выполняться в предложении FROM. Предложение ON/USING в OUTER JOIN *не* эквивалентно условию в WHERE, потому что результаты запроса будут разными.

Вот несколько примеров предложения WHERE:

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

fdt — это таблица, полученная в предложении FROM. Строки, которые не попадают под условие поиска в выражении WHERE, удаляются из fdt. Обратите внимание,

используйте скалярные позапросы как выражения значений. Как и другие запросы, подзапросы могут возвращать сложные табличные выражения. Также обратите внимание, как fdt используется в подзапросах. Полное имя c1 такое как fdt.c1 необходимо только если c1 также является именем колонки во входной таблице подзапроса. Но полное имя колонки обеспечивает однозначность, даже когда она не требуется. Этот пример показывает как пространство имён колонок внешнего запроса расширяет пространаство имён колонок во внутренних запросах.

4.2.3. Предложения GROUP BY и HAVING

После прохода через фильтр WHERE, полученная входная таблица может быть сгруппирована, используя предложение GROUP BY, а также из неё могут удалены строки, используя предложение HAVING.

SELECT *список_выборки*
FROM ...
[WHERE ...]
GROUP BY *ссылка_на_группирумую_колонку* [, *ссылка_на_группирумую_колонку*]...

GROUP BY Clause используется для группировки вместе тех строк таблицы, которые имеют те же самые значения во всех перечисленных колонках. Порядок, в котором перечисляются колоки значения не имеет. Эффект группировки состоит в комбинировании каждого списка строк, имеющих общие значения колонок в одну сгруппированную строку, которая представляет все строки в данной группе. Это осуществляется исключением избыточности в выводе и/или подсчётом агрегатов, которые применяются к группам. Например:

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

Во втором запросе, мы не можем написать SELECT * FROM test1 GROUP BY x, потому что в нём нет ни одного значения в колонке y, которое может быть ассоциировано с каждой группой. Группируемые колонки могут быть перечислены в списке выбора, в случае, если они имеют хотя бы одно значение в каждой группе. Обычно, если таблица группируется, колонки, которые не перечислены в GROUP BY, не могут быть указаны нигде, кроме агрегирующего выражения. Пример агрегирующего выражения:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
```

```
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Здесь sum является агрегатной функцией, которая подсчитывает одно значение во всей группе. Больше информации о доступных агрегатных функциях можно найти в Section 9.18.

Tip: Группировка без агрегирующих выражений эффективно подсчитывает список отдельных значений в колонке. Это же может быть достигнуто, используя предложение DISTINCT (см. Section 4.3.3).

Здесь другой пример: он подсчитывает полные продажи для каждого продукта (вместо подсчёта полных продаж всех продуктов):

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

В этом примере, колонки product_id, p.name и p.price должны быть в предложении GROUP BY, поскольку в запросе они указываются в списке выборки (но см. ниже). Колонки s.units нет в списке GROUP BY, поскольку она используется только в агрегирующем выражении (sum(...)), которое представляет собой продажи продукта. Для каждого продукта, запрос возвращает суммарную строку о всех продажах этого продукта.

Если таблица products сделана так, что product_id является первичным ключём, то его должно быть достаточно для группировки по product_id в данном выше примере, так как name и price должны быть *функционально зависимы* от product_id и таким образом не должно быть двусмысленности в отношенни того какие name и price будут соответствовать каждой группе product_id.

В строгом SQL, GROUP BY может группировать только по колонкам исходной таблицы, но PostgreSQL расширяет поведение группировки, позволяя GROUP BY группировать по колонкам в списке выбора. Также позволяется группировка по значениям вместо имён колонок.

Если таблица была сгруппирована с помощью предложения GROUP BY, но интерес представляют только определённые группы, можно использовать предложение HAVING, которое очень похоже на предложение WHERE, но только для удаления групп из результата. Синтаксис такой:

SELECT *список_выбора* FROM ... [WHERE ...] GROUP BY ... HAVING *логическое_выражение*

Выражения в предложении HAVING могут использовать как сгруппированные выражения, так и несгруппированные выражения (которые необходимо обработать агрегатной функцией).

Ghbvth:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
```

```
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

Ещё один, более реалистичный пример:

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

В этом примере, предложение WHERE выбирает строки с колонками, которые не группируются (выражение истинно только для продаж за последние четыре недели), в то время как предложение HAVING ограничивает вывод сгруппированных результатов только для случаев, когда объём продаж превышает 5000. Обратите внимание, что агрегатные выражения могут разными во всех частях запроса.

Если запрос содержит вызовы агрегатный функций, но не содержит предложение GROUP BY, группировка всё-равно будет осуществлена: результатом будет одна сгруппированная строка (или не будет никаких строк, если данная сгруппированная строка будет исключена предложением HAVING). Тоже самое будет, если запрос содержит предложение HAVING, даже без каких-либо вызовов агрегатных функций GROUP BY.

4.3. Списки выбора

Как рассказывалось выше, табличное выражение в команде SELECT конструирует промежуточную виртуальную таблицу, которая возможно комбинирует в себе таблицы, представления, отдельные строки, группировки и т.д. Эта виртуальная таблица в конце концов передаётся на обработку *списка выбора*. Список выбора определяет, какие *колонки* из промежуточной таблицы в итоге будут выведены.

4.3.1. Элементы списка выбора

Простейший вид списка выбора это *, который говорит о выборе всех колонок, которые получились после работы табличного выражения. В другом случае, список выбора - это список разделённых запятыми значимых выражений (как определяется в Section 4.2). Например, это может быть списком имён колонок:

```
SELECT a, b, c FROM ...
```

Колонки с именами a, b и c либо актуальные имена колонок таблиц из предложения FROM или их псевдонимы, как рассказывалось в Section 4.2.1.2. Пространство имён доступное в списке выбора, является тем же, что и в предложении WHERE, за исключением использованных группировок, которые будут такими же как в предложении HAVING.

Если более чем в одной таблице есть колонка с тем же самым именем, перед именем колонки должно быть указано имя таблицы:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

При работе с несколькими таблицами, это может быть также полезно для запроса всех колонок в отдельной таблице:
SELECT tbl1.*, tbl2.a FROM ...
(См. также Section 4.2.2.)
Если соответствующее значимое выражение используется в списке выбора, оно концептуально добавляет новую виртуальную колонку к возвращаемой таблице. Значимое выражение выполняется один раз для каждой строки результата, с подстановкой значений из этой строки в каждую ссылку на колонку. Но выражения в списке выбора не должны указывать ни на какие колонки в табличном выражении предложения FROM; они, например, могут быть постоянными арифметическими выражениями.
4.3.2. Метки колонок
Записям в списке выбора могут быть назначены имена для дальнейшей обработки, такой как использование предложения ORDER BY или для отображения в клиентском приложении. Например:
SELECT a AS value, b + c AS sum FROM ...
Если не будет назначено никакого имени колонки в AS, то СУБД назначит имя колонки по умолчанию. Для простой ссылки на колонку, это имя будет таким же как у самой колонки. Для вызовов функций, это будет именем функции. Для сложных выражений, СУБД будет генерировать какое-либо простое имя.
Ключевое слово AS является необязательным, но только, если новое имя колонки не совпадает с другими ключевыми словами PostgreSQL (см. Appendix C). Чтобы избежать нежелательного совпадения с ключевым словом, вы можете заключить имя колонки в двойные кавычки. Например, VALUE является ключевым словом, так что такой запрос не работает:
SELECT a value, b + c AS sum FROM ...
а вот такой будет работать:
SELECT a "value", b + c AS sum FROM ...
Чтобы защитится от возможного совпадения с ключевыми словами, которые будут добавлены в будущем, всегда рекомендуется или писать AS или заключать имя колонки в двойные кавычки.
Note: Именование выходных колонок здесь отличается от того, что в предложении FROM (см. Section 4.2.1.2). Возможно переименовать одну и ту же колонку дважды, но имя, назначаемое в списке выбора является одним из тех, что передаются в него.
4.3.3. DISTINCT
После того как обработан список выбора, результирующая таблица может быть дополнительно обработана для исключения дублирующихся строк. Чтобы выполнить этоб непосредственно после SELECT пишется ключевое слово DISTINCT:
SELECT DISTINCT *список_выбора* ...
(Вместо ключевого слова DISTINCT может быть использовано ключевое слово ALL, которое задаёт поведение по умолчанию, обуславливающее выдачу всех строк.)
В общем случае, две строки считаются отличными друг от друга, если они различаются хотя бы одним значением колонки. Значения при сравнении NULL считаются равными.

В качестве альтернативы, какие строки должны считаться отличными друг от друга может определить указанное выражение:
SELECT DISTINCT ON (*выражение* [, *выражение* ...]) *список_выбора* ...
Здесь *выражение* является заданным значением выражения, которое применяется ко всем строкам. Список строк, которым будет соответствовать данное выражение, будет считаться дублирующим и при выводе результатов останется только первая строка этого списка. Обратите внимание, что какая именно строка будет "первой строкой" непредсказуемо, если результат запроса не сортируется по нужным колонкам, чтобы гарантировать уникальный порядок строк, полученных после фильтра DISTINCT. (Обработка DISTINCT ON происходит после сортировки ORDER BY.)
Предложение DISTINCT ON не является частью стандарта SQL и иногда считается плохим стилем, потому что потенциально может привести к неожиданным результатам. При разумном использовании GROUP BY и подзапросов в FROM, данное предложение может быть опущено, но часто оно является наиболее удобной альтернативой.
4.4. Комбинирование запросов
Результаты двух запросов могут быть скомбинированы, с помощью списочных операций объединения, пересечения и вычитания. Синтаксис следующий:
запрос1 UNION [ALL] *запрос2*
запрос1 INTERSECT [ALL] *запрос2*
запрос1 EXCEPT [ALL] *запрос2*
запрос1 и *запрос2* являются запросами, которые могут использовать любые из описанных выше возможностей. Списочные операции также могут быть вложенными или выстроенными в цепочку, например:
запрос1 UNION *запрос2* UNION *запрос3*
что выполняется как:
(*запрос1* UNION *запрос2*) UNION *запрос3*
UNION добавляет результат *запроса2* к результату *запроса1* (хотя сохранение порядка строк, в котором они возвращаются запросом не гарантируется). Таким образом, объединение исключает дублирующиеся строки из результата точно также как это делает DISTINCT, если только не используется UNION ALL.
INTERSECT возвращает все строки, которые есть как в результате *запроса1* так и в результате *запроса2*. Дублирующиеся строки исключаются, если только не используется INTERSECT ALL.
EXCEPT возвращает все строки, которые есть в результате *запроса1*, но которых нет в результате *запроса2*. (Иногда это называют также *разницей* между двумя запросами.) И снова, дублирующиеся строки исключаются, если только не используется EXCEPT ALL.
4.5. Сортировка строк
После того как запрос выдал итоговую результирующую таблицу (после обработки списка выбора), эта таблица по желанию может быть отсортирована. Если сортировка не задана, строки выдаются в несортированном виде. Фактически, сортировка в этом случае будет зависеть от плана сканирования и объединения и порядка расположения на диске, но вы не должны доверять этим соображениям. Определённый порядок вывода может быть гарантирован только, если явно задана и выполнена сортировка.

Предложение ORDER BY задаёт порядок сортировки:

SELECT *список_выбора*
FROM *табличное_выражение*
ORDER BY *выражение_сортировки1* [ASC | DESC] [NULLS { FIRST | LAST }]
[, *выражение_сортировки2* [ASC | DESC] [NULLS { FIRST | LAST }] ...]

Выражение(я) сортировки могут быть любыми выражениями, которые являются допустимыми в списке выбора. Например:

SELECT a, b FROM table1 ORDER BY a + b, c;

Когда задаётся более чем одно выражение сортировки, более поздние значения используются для сортировки строк, которые уже были отсортированы в соответствии с более ранними значениями. За каждым выражением может следовать необязательное ключевое слово ASC или DESC, чтобы указать направление сортировки: по возрастанию или по убыванию. По умолчанию используется ASC. Сортировка по возрастанию помещает меньшие значения вначале, где "меньшее" определяется согласно оператору <. Точно также, сортировка по убыванию определяется оператором >. [1]

Для указания порядка следования значений NULL при сортировке, могут быть использованы опции NULLS FIRST и NULLS LAST. По умолчанию, значения NULL сортируются так, как если бы любое значение было больше NULL; таким образом NULLS FIRST для DESC является порядком по умолчанию, в противном случае NULLS LAST.

Обратите внимание, что опции упорядочивания указываются независимо для каждой сортируемой колонки. Например, ORDER BY x, y DESC означает ORDER BY x ASC, y DESC, а не ORDER BY x DESC, y DESC.

Для обратной совместимости с версией стандарта SQL92, в *выражении_сортировки* могут быть метка или номер колонки в выводе:

SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;

Оба вышеуказанных примера, выполняют сортировку по первой колонке. Обратите внимание, что имя выходной колонки должно быть самостоятельным именем, оно не может быть использовано в выражении, например так *неправильно*:

SELECT a + b AS sum, c FROM table1 ORDER BY sum + c; -- ошибка

Это ограничение снижает путаницу. Путаница возможна, если элемент ORDER BY является простым именем, которое может совпадать либо с именем выводимой колонки, либо с колонкой из табличного выражения. В таких случаях используется выводимая колонка. Такое поведение может вызвать путаницу только, если вы используете AS для переименования выходной колонки, чтобы она совпадала с некоторыми другими именами колонок таблицы.

ORDER BY может применяться к результатам комбинирования UNION, INTERSECT или EXCEPT, но в этому случае допускается только сортировка по именам или номерам выходных колонок, а не по выражениям.

Notes

[1] Фактически, PostgreSQL использует *класс оператора B-tree по умолчанию* для типа данных указанного выражения сортировки, чтобы определить порядок сортировки при использовании ASC и DESC. По соглашению, типы данных установливаются так, чтобы операторы < и > отвечали порядку сортировки, но

разработчик пользовательского типа данных может выбрать что-либо другое.

4.6. LIMIT and OFFSET

Предложения LIMIT и OFFSET позволяют вам полчить только часть строк, сгенерированных запросом:

SELECT *список_выбора*
FROM *табличное_выражение*
[ORDER BY ...]
[LIMIT { *номер* | ALL }] [OFFSET *номер*]

Если задано количество для LIMIT, то будет выдано не более, чем указанное количество строк (но возможно меньшее, если сам запрос возвращает меньше строк). Указание LIMIT ALL аналогично отсутствию предложения LIMIT.

OFFSET говорит пропустить определённое количество строк перед началом вывода. OFFSET 0 аналогично отсутствию предложения OFFSET, LIMIT NULL это тоже самое, что и отсутствие предложения LIMIT. В случае указания как OFFSET так и LIMIT, перед выводом указанного в LIMIT количества строк, пропускается указанное в OFFSET количество строк.

При использовании LIMIT, важно использовать предложение ORDER BY, которое обеспечивает вывод строк в определённом порядке. В противном случае вы получите при выполнении запроса непредстказуемый набор строк. Вы можете спросить, а что будет если вы захотите получить десять строк, после пропуска двадцати? Если вы не указали ORDER BY, то какие это будут строки неизвестно.

Оптимизатор запроса при построении плана запроса берёт LIMIT в рассчёт, так что вы будете получать различные планы запросов (неустойчивые разные порядки следования строк) в зависимости от того какие значения вы укажете для LIMIT и OFFSET. Таким образом, использование различных значений в LIMIT/OFFSET для выбора различных подсписков из результата запроса *приведёт к получению противоречивых результатов*, если только для получения предсказумых результатов, вы не используете сортировку с ORDER BY. Такое поведение не является ошибочным; оно является закономерным следствием того факта, что SQL не обещает получения результатов запроса в одном и том же порядке, если для упорядочивания не осуществляется явно с помощью ORDER BY.

Строки, пропущенные предложением OFFSET должны подсчитываться внутри СУБД; таким образом большие значения OFFSET могут работать неэффективно.

4.4. Списки VALUES

VALUES предоставляет метод генерации "постоянной таблицы", которая может быть использована в запросе без фактического создания и размещения этой таблицы на диске. Синтаксис следующий:

VALUES (*выражение* [, ...]) [, ...]

Каждый заключённый в круглые скобки список генерирует строку в таблице. Все списки должны иметь то же самое количество элементов (т.е. количество колонок в таблице) и соответствующие элементы в каждом списке должны иметь совместимые типы данных. Фактический тип данных, назначаемый каждой колонке, определяется используя те же правила, что и для UNION (см. Section 10.5).

Пример:

VALUES (1, 'one'), (2, 'two'), (3, 'three');

вернёт таблицу с двумя колонками и тремя строками. Это эквивалентно:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

По умолчанию, PostgreSQL назначает колонкам таблицы VALUES имена column1, column2 и т.д. Эти имена колонок не оговариваются стандартом SQL и разные СУБД генерируют их по-разному, так что обычно, лучше перекрыть имена, присвоенные по умолчанию с помощью списка табличных псевдонимов.

Синтаксически, VALUES, за которым следуют списки выражений, считается эквивалентным:

SELECT *список_выбора* FROM *табличное_выражение*

и может появится везде, где может появляться SELECT. Например, вы можете использовать эту конструкцию как часть UNION или прикрепить к ней *спецификацию_сортировки* (ORDER BY, LIMIT, и/или OFFSET). VALUES наиболее часто используется как источник данных в команде INSERT и затем наиболее часто как подзапрос.

4.8. Запросы WITH (Общие табличные выражения)

WITH предоставляет способ написания вспомогательных операторов для использования в более больших запросах. Эти операторы, которые часто называются как Общие Табличные Выражения или CTE, могут быть задуманы как определяющие временные таблицы, которые существуют только для данного запроса. Каждый вспомогательный оператор в предложении WITH может затем быть подвергнут SELECT, INSERT, UPDATE или DELETE; а само предложение WITH прикрепляется к первичному оператору, которые также может быть одним из SELECT, INSERT, UPDATE или DELETE.

4.8.1. SELECT в WITH

Базовое значение SELECT в WITH должно разбивать сложные запросы на более простые части. Например:

```
WITH regional_sales AS (
        SELECT region, SUM(amount) AS total_sales
        FROM orders
        GROUP BY region
    ), top_regions AS (
        SELECT region
        FROM regional_sales
        WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
    )
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

запрос показывает общие продажи каждого продукта только в регионах с высоким объёмом продаж. Преложение WITH определяет два вспомогательных оператора с именами regional_sales и top_regions, где вывод regional_sales используется в top_regions, а вывод top_regions используется в первичном запросе SELECT. Этот пример может быть написан без WITH, но тогда потребуется два уровня вложенных под-SELECT'ов. Гораздо легче следовать вышеописанному методу.
Необязательный модификатор RECURSIVE изменяет WITH с явного синтаксического комфорта в возможность, выполняющую такие вещи, которые невозможны в стандарте SQL. Используя RECURSIVE, запрос WITH может ссылаться на свой собственный вывод. Простой пример такого запроса состоит в суммировании целых чисел от 1 до 100:

```
WITH RECURSIVE t(n) AS (
   VALUES (1)
 UNION ALL
   SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Общая форма рекурсивного запроса WITH всегда *нерекурсивный термин*, затем UNION (или UNION ALL), затем *рекурсивный термин*, где только рекурсивный термин может содержать ссылку на свой собственный вывод запроса. Такой запрос выполняется так:

Рекурсивное выполнение запроса

1. Выполняется не-рекурсивный термин. Для UNION (но не для UNION ALL), отбрасываются дублирующиеся строки. Включаются все оставшиеся строки из результата рекурсивного запроса и также размещаются во временную *рабочую таблицу*.
2. Пока рабочая таблица не окажется пустой, повторяются следующие шаги:
   a. Выполняется рекурсивный термин, подстановка текущего содержимого рабочей таблицы для рекурсивной ссылки на саму себя. Для UNION (но не для UNION ALL), отбрасываются дублирующиеся строки и строки, которые дублируют любые строки в предыдущих результатах. Включаются все оставшиеся строки из результата рекурсивного запроса и также размещаются во временную *промежуточную таблицу*.
   b. Замещается содержимое рабочей таблицы на содержимое промежуточной таблицы, затем промежуточная таблица очищается.

Note: Строго говоря, данный процесс является нерекурсивной итерацией, но RECURSIVE является терминологическим выбором комитета по стандартам SQL.
В данном выше примере, рабочая таблица на каждом шаге содержит только одну строку, в которую на успешных шагах попадают значения от 1 до 100. На 100-м шаге, вывода нет из-за преложения WHERE и таким образом, запрос завершается.
Рекурсивные запросы обычно используются при работе с иерархическими данными или данными, которые имеют древовидную структуру. Полезный пример такого запроса состоит в нахождении всех прямых и непрямых составных частей продукта, представленных как таблица, которая показывает только состав продукта из частей или одной части из других:

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
   SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
```

```
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

При работе с рекурсивными запросами, важно убедиться, что рекурсивная часть запроса не будет возвращать строки в определённый момент времени, в противном случае запрос станет бесконечным циклом. Иногда, используя UNION вместо UNION ALL этого можно добиться отбрасывая строки, которые дублируют выведенные ранее. Однако, часто цикл не выдаёт строк, которые дублируются полностью: это может быть необходимо, чтобы проверить только одно или несколько полей, чтобы увидеть, когда та же самая точка была достигнута ранее. Стандартный метод управления такими ситуациями состоит в подсчёте массива уже обработаных значений. Например, рассмотрим следующий запрос, который просматривает таблицу graph, используя поле link:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
        SELECT g.id, g.link, g.data, 1
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1
        FROM graph g, search_graph sg
        WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

Данный запрос зациклится, если отношения link содержат циклы. Поскольку мы требует вывода "depth", то простое изменение UNION ALL на UNION должно исключить зацикливание. Вместо этого, нам нужно определять достигли ли мы или нет той же строки снова, когда мы приходим к следующуму определённому пути поля link. Мы добавляем две колонки path и cycle в зацикливающийся запрос:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[g.id],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || g.id,
          g.id = ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Для предотвращения зацикливания, значение массива часто полезно для представления самого себя как "path", чтобы понять, что была достигнула любая отдельная строка.
В большинстве случаев, когда для определения зацикливания нужна проверка более чем одного поля, используйте массив строк. Например, если вам необходимо сравнить поля f1 и f2:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[ROW(g.f1, g.f2)],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || ROW(g.f1, g.f2),
          ROW(g.f1, g.f2) = ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Tip: Опускайте синтаксис ROW() в общем случае, где для определения зацикливания необходимо проверять только одно поле. Это позволит использовать простой массив, вместо массива с составным типом, что даёт преимущество в производительности.
Tip: Алгоритм выполнения рекурсивного запроса выводит результаты в порядке нахождения первых подходящих значений. Вы можете просматривать эти результаты в порядке вложенности, указав во внешнем запросе ORDER BY для колнки "path".
Полезная уловка для тестирования запросов, когда вы не знаете точно, может ли случиться зацикливание, состоит в добавлении LIMIT к родительскому запросу. Например, данный запрос без LIMIT зацикливался бы всегда:

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

Это работает, потому что текущая реализация PostgreSQL производит дальнейшее выполнение только в зависимости от того, как много строк запроса WITH фактически было получено родительским запросом. Использование данной уловки в продуктивных решениях не рекомендуентся, потому что другие СУБД могут работать по-другому. Также, такое обычно не работает, если вы делаете внешнюю сортировку результатов рекурсивного запроса или соединяете их с некоторой другой таблицей, потому что в таких случаях внешний запрос обычно в любом случае будет пытаться получить вывод от всех запросов WITH.
Полезное свойство запросов WITH заключается в том, что они выполняются только один раз при запуске родительского запроса, даже если они они вызываются из родительского запроса или вложеных WITH запросов более одного раза. Таким образом, затратные по ресусам вычисления, которые необходимы во многих случаях,

могут быть размещены внутри WITH запроса, чтобы избежать избыточной обработки. Другое возможное применение состоит в предотвращении нежелательного многократного выполнения функций, которое может вызвать посторонние эффекты. Однако, другая сторона монеты в том, что оптимизатор мало поможет в применении ограничений из родительского запроса к WITH запросу, как в случае обычного подзапроса. WITH запрос будет обычно выполнен как описано, без подавления строк, которые впоследствие могут быть отброшены родительским запросом. (Но, как говорилось выше, выполнение можно остановить раньше, если данный запрос будет ограничен указанием количества строк.)
Пример, данный выше показывает только как WITH будет использован с SELECT, но он может быть тем же способом прикреплён к INSERT, UPDATE или DELETE. В этом случае он фактически предоставляет временную таблицу(ы), которая может быть использована в главной команде.
4.8.2. Операторы, изменяющие данные в WITH
Вы можете использовать в WITH операторы, изменяющие данные (INSERT, UPDATE или DELETE). Это позволяет выполнять некоторые различные операции в одном запросе. Пример:

```
WITH moved_rows AS (
   DELETE FROM products
   WHERE
      "date" >= '2010-10-01' AND
      "date" < '2010-11-01'
   RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

Данный запрос фактически перемещает строки из products в products_log. DELETE в WITH удаляет заданные строки из products, возвращая их содержимое с помощью предложения RETURNING; и затем первичный запрос читает то что выводится и вставляет его в products_log.
Изящество данного выше примера в том, что предложение WITH прикрепляется к INSERT, вместо под-SELECT внутри INSERT. Это необходимо, потому что операторы, изменяющие данные разрешены только в выражениях WITH, которые прикрепляются к вышестоящему оператору. Однако, примененяются нормальные правила видимости WITH, так что становится возможным работать с выводом WITH из под-SELECTа.
Изменяющие данные операторы в WITH обычно имеют предложение RETURNING как видно в вышепоказанном примере. Вывод, осуществляемый предложением RETURNING, *не* является целевой таблицей оператора, изменяющего данные, а формирует временную таблицу, которая может использоваться в оставшейся части запроса. Если оператор, изменяющий данные в WITH опускает выражение RETURNING, то временная таблица не формирует и её нельзя использовать в оставшейся части запроса. Тем не менее, такие операторы будут запущены. Не очень полезный пример:

```
WITH t AS (
   DELETE FROM foo
```

```
)
DELETE FROM bar;
```

Данный пример должен удалить все строки из таблиц foo и bar. Кодичество затронутых запросом строк, выданных клиенту, должно включать только строки удалённые из bar.

Рекурсивные ссылки на себя в операторах, изменяющих данные, не разрешаются. В некоторых случаях возможно обойти это ограничение сославшись на вывод реурсивного WITH, например:

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

Данный запрос удалит все прямые и непрямые подчасти sub_part продукта product.

Операторы, изменяющие данные в WITH запускаются точно один раз и всегда, до полного завершения, независимо от того, будет ли первичный запрос читать всё (или ничего) из выводимого ими. Обратите внимание, что нужно различать правило для SELECT в WITH: как указано в предыдущем параграфе, выполнение SELECT осуществляется только когда первичный запрос затребует выводимое им.

Подоператоры в WITH запускаются конкуррентно с каждый другим и с основным запросом. Таким образом, когда в WITH используются запросы, изменяющие данные, порядок в котором фактически происходят заданные обновления является непредсказуемым. Все операторы запускаются в том же самом *снимке* (snapshot) (см. Chapter 13), так что они не могут "видеть" все другие изменения целевых таблиц. Это смягчает эффекты непредсказуемости фактического порядка обновления строк, и означает, что данные, возвращаемые RETURNING являются лишь способом связи изменений между разными подоператорами WITH и основным запросом. Вот пример этого:

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

внешний SELECT должен вернуть первоначальные цены (prices) перед выполнением UPDATE, в то время как

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

внешний SELECT должен вернуть обновлённые данные.

Попытка обновить ту же строку дважды в одном операторе не поддерживаются. Происходит только одно изменение, но надёжно предсказать каким будет это изменение не легко (и иногда невозможно). Это же самое касается также удаления строки, которая была уже обновлена в этом же операторе: выполняется только обновление. Таким образом вы должны обычно избегать попытки изменить одну строку дважды в одном операторе. В особенности избегать писать подоператоры WITH, которые могут влиять на те же строки, которые изменены основным или другим таким же оператором. Изменения, производимые такими операторами, будут непредсказуемы.
Как уже говорилось, любая таблица, используемая как целевая в WITH операторах, изменяющих данные, не должна иметь ни условных правил, ни правила ALSO, ни правила INSTEAD, которые расширяются до нескольких операторов.

ЛАБОРАТОРНАЯ РАБОТА 5
Проектирование реляционной базы данных

Продолжительность работы – 6 часов.
Цели:

- спроектировать БД в соответствии с вариантом согласно примеру, представленному в методическом указании;
- провести нормализацию отношений в БД (до 3 нормальной формы).

ЗАДАНИЕ

1) Описать предметную область.
   Опишите особенности работы системы. Определите группы пользователей, их основные информационные задачи и запросы к БД. Сформулируйте на естественном языке 10-12 типовых запросов для разных категорий пользователей: для получения ответа на простой вопрос, выполнение расчетов, отбор и сортировку данных, а также типовые задачи добавления, изменения или удаления данных в БД.
2) Выделить ключевые объекты системы.
3) Провести инфологическое проектирование.
   a. Составить и прокомментировать ER-диаграмму.
   b. Составить и прокомментировать уточненную ER-диаграмму (с атрибутами).
4) Провести логическое проектирование (разработать схемы отношений).
5) Провести нормализацию БД до 3 НФ и создать окончательную схему БД. После нормализации количество таблиц должно не превышать 7-8.
6) Описать ключевые ограничения.
7) Описать права доступа к каждой таблице для разных групп пользователей.
8) Подготовить ответы на контрольные вопросы.

Структура отчета:

1) титульный лист;
2) задание своего варианта;
3) описание процесса проектирования (инфологическое проектирование и ER-модель, см. пример из методических рекомендаций);
4) схемы полученных отношений;
5) ответы на контрольные вопросы.

Вариант предметной области и соответствующий набор данных можно выбрать на стр. **156-196** в пособии Гурвиц Г. А. Разработка реального приложения с использованием Microsoft Visual FoxPro 9 : учеб. пособие. – Хабаровск: Изд-во ДВГУПС, 2007. – 198 с.: ил.).

Для построения ER-диаграмм можно воспользоваться векторным графическим редактором, редактором диаграмм и блок схем – Microsoft Visio.

КОНТРОЛЬНЫЕ ВОПРОСЫ к ЛР 5

1. Что понимают под сущностью и какими свойствами должна обладать сущность? Поясните ответ примерами сущностей для предметной области своего варианта.
2. Приведите примеры связе между сущностями?
3. Что представляет собой реляционная таблица и какими базовыми свойствами она обладает?
4. Опишите домены атрибутов для предметной области своего варианта.
5. Что такое ключ? Какие они бывают? Что такое первичный ключ?
6. Перечислите и поясните этапы проектирования базы данных.
7. Что представляет собой логическая (даталогическая) модель предметной области? Чем она отличается от ER-диаграммы?
8. Как в базе данных реализуется связь типа 1:n (один-ко-многим) между отношениями? Приведите пример для своего варианта.
9. Что такое нормализация отношений? Кто разработал механизм нормализации реляционных отношений и в чем его назначение?
10. Что представляет собой декомпозиция схемы отношений?
11. Какое отношение находится в первой нормальной форме (1НФ)? Приведите пример нарушения 1НФ для предметной области своего варианта.
12. Что нужно проделать для приведения таблицы к 1НФ?
13. На чем основана 2НФ? Приведите пример нарушения 2НФ на примере предметной области своего варианта.
14. Что нужно проделать для приведения таблицы ко 2НФ?
15. Дайте понятие транзитивной зависимости. Приведите примеры функциональной зависимости атрибутов отношения для своего варианта.
16. Какое отношение находится в 3НФ? Приведите пример нарушения 3НФ на примере предметной области своего варианта.
17. Что нужно проделать для приведения таблицы к 3НФ? Приведите пример устранения функциональной зависимости атрибутов от части сложного ключа на примере предметной области своего варианта.

Для получения дополнительного балла:

18. Дайте определения Нормальной формы Бойса-Кодда (НФБК), 4НФ.
19. Приведите пример устранения нетривиальной многозначной зависимости для своего варианта.
20. Приведите примеры, когда предпочтительнее использовать функциональный подход к проектированию БД ("от задач"), а когда лучше применить предметный подход к проектированию БД ("от предметной области").

ЛАБОРАТОРНАЯ РАБОТА 6
Реализация проекта базы данных с многопользовательским доступом к данным в PostgreSQL

Продолжительность работы – 6 часов.
Цели:

- реализовать проект БД;
- разработать готовые запросы к БД.

ЗАДАНИЕ

1) В среде PostgreSQL создать таблицы БД, спроектированной в ЛР № 5
2) Заполнить таблицы данными (не менее 10-15 записей)
3) Создать 10-12 представлений (готовые запросы) к БД.
4) Назначение прав доступа
5) Создание триггеров

На дополнительный балл

6*) Создание индексов (с обоснованием необходимости и доказательством повышения скорости выполнения запросов.)

Структура отчета:

1) задание своего варианта и схема БД из ЛР № 5
2) набор команд создания базы данных
3) набор команд заполнения БД
4) набор команд создания запросов на SQL со скриншотами результатов работы запросов

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания по организации и проведению самостоятельной работы
по дисциплине
Базы данных
для студентов направления подготовки
02.03.01 Математика и компьютерные науки
направленность (профиль)
«Анализ данных и искусственный интеллект»

1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1. Общие положения

Проектирование базы данных (БД) является одной из наиболее сложных и ответственных задач, связанных с созданием автоматизированной информационной симтемы (АИС).

Проектирование БД подразумевает использование определённой технологии. Не следует начинать работу по реализации реляционной БД с создания таблиц. Получившаяся в ходе такого "проектирования" база данных будет ненадёжной, неэффективной и сложной в сопровождении. Исключением могут быть случаи простых предметных областей, которые можно отразить в реляционной базе данных, состоящей из 2-3 таблиц. Поэтому, если вас интересует результат, при создании базы данных необходимо придерживаться правил технологии проектирования БД.

Опишем вкратце процесс проектирования реляционной базы данных. (Более подробно этот процесс изложен в [2, 3]).

База данных – это, фактически, модель предметной области (ПрО). Значит, для создания БД надо сначала проанализировать ПрО и создать её модель (это называется **инфологическим проектированием**).

Основой для *анализа предметной области* служат документы, которые отражают ПрО, и информация, которую можно получить от специалистов этой предметной области в процессе общения с ними. Для анализа берутся те документы, которые имеют отношение к решаемой задаче. Изучение документов позволяет выявить объекты (сущности ПрО) и атрибуты сущностей – данные, которые должны храниться в БД. Из общения со специалистами необходимо извлечь сведения об особенностях ПрО, которые позволяют установить ограничения целостности, зависимости и связи между объектами (субъектами) предметной области. Также специалисты обладают знаниями о том, каковы алгоритмы обработки данных и какие задачи ставятся перед информационной системой.

Таким бразом, начать следует с анализа входных и выходных документов и опроса специалистов.

Модель ПрО может быть описана любым удобным для разработчика способом (словесное описание, набор формул, диаграмма потоков данных и т.п.). Но чаще всего при проектировании баз данных используется метод сущность–связь, и модель ПрО выполняется в виде ER–диаграммы (entity-relation diagram, диаграмма «сущность-связь»).

То-есть, следующий этап – создание ER–диаграммы.

После создания модели ПрО определяются **требования к операционной обстановке**: какое аппаратное и программное обеспечение необходимо для реализации БД и АИС в целом. Основные технические параметры (объём оперативной и дисковой памяти, наличие сетевой платы и др.) определяются исходя из планируемого объёма БД, режима работы (локальный или удалённый доступ) и требований к эффективности работы системы (например, ко времени реакции на запрос пользователя или к общей производительности БД). В

зависимости от планируемой нагрузки (интенсивности запросов) и требований к надёжности выбирается операционная система. Затем осуществляется **выбор СУБД**, под управлением которой будет работать создаваемая база данных.

В нашем курсе будем использовать PostgreSQL.

На следующем этапе – этапе **логического проектирования** – ER-диаграмма формальным способом преобразуется в схему реляционной базы данных (РБД). На основании схемы РБД и описания сущностей ПрО составляются отношения (таблицы) базы данных.

Потом выполняется **нормализация отношений**. Это необходимо сделать для того, чтобы исключить нарушения логической целостности данных и повысить надёжность и достоверность данных. В полученной нормализованной модели должны отсутствовать аномалии модификации, аномалии удаления и аномалии добавления. В отдельных случаях после нормализации может выполняться денормализация, но причина для этого может быть только одна: повышение эффективности выполнения критических запросов.

В результате всех этих операций создаётся **концептуальная схема БД** – основной технический документ для базы данных.

Далее, на этапе **физического проектирования** полученные отношения описываются на языке DDL (Data Definition Language) – языке определения данных, который поддерживается выбранной СУБД.

Также необходимо определить способы хранения данных (кластеризация, хеширование), способы доступа к данным (индексирование) и создать соответствующие индексы и кластеры (если нужно).

Если пользователей АИС можно разделить на группы по характеру решаемых задач, то для каждой группы создаётся свой набор прав доступа к объектам БД.

1.2. Последовательность проектирования базы данных

Итак, процесс проектирования включает в себя следующие шаги:

1. Определение задач, стоящих перед базой данных.
2. Сбор и анализ документов, относящихся к исследуемой предметной области.
3. Определение групп пользователей и перечня задач, стоящих перед каждой группой.
4. Описание особенностей ПрО, которые позволяют установить зависимости и связи между объектами (субъектами) предметной области.
5. Создание модели предметной области.
6. Выбор аппаратной и программной платформы для реализации БД.
7. Выбор СУБД (системы управления базой данных).
8. Создание логической схемы БД.
9. Создание схем отношений, определение типов данных атрибутов и ограничений целостности.
10. Нормализация отношений (до третьей или четвёртой нормальной формы).
11. Определение прав доступа пользователей к объектам БД.
12. Написание кода создания основных объектов базы данных на языке SQL в синтаксисе выбранной СУБД (пользователи, таблицы и др.).
13. Написание кода создания вспомогательных объектов базы данных (представления, индексы, триггеры, роли и т.д.).

Эти шаги можно объединить в 5 этапов:

1. Инфологическое проектирование (1-5).
2. Определение требований к операционной обстановке, в которой будет функционировать информационная система (6).
3. Выбор системы управления базой данных (СУБД) и других инструментальных программных средств (7).
4. Логическое проектирование БД (8-11).
5. Физическое проектирование БД (12-13).

На сегодняшний день не существует формальных способов моделирования реальности, но инфологический подход закладывает основы методологии проектирования базы данных как модели предметной области.

1.2.1. Инфологическое проектирование

Основными задачами этапа инфологического проектирования являются определение предметной области системы и формирование взгляда на неё с позиций сообщества будущих пользователей БД, т.е. информационно-логической модели ПрО.

Инфологическая модель ПрО представляет собой описание структуры и динамики ПрО, характера информационных потребностей пользователей в терминах, понятных пользователю и не зависимых от реализации БД. Это описание выражается в терминах не отдельных объектов ПрО и связей между ними, а их типов, связанных с ними ограничений целостности и тех процессов, которые приводят к переходу ПрО из одного состояния в другое.

Основными подходами к созданию инфологической модели предметной области являются [1]:

1. Функциональный подход к проектированию БД ("от задач").
2. Предметный подход к проектированию БД ("от предметной области").
3. Метод "сущность-связь" (entity–relation, ER–method).

Мы будем использовать метод "сущность–связь" как наиболее распространённый. Приведём основные термины, которыми мы будем пользоваться:

Сущность – это объект, о котором в системе будут накапливаться данные. Для сущности указывается название и тип (сильная или слабая). Сильные сущности существуют сами по себе, а существование слабых сущностей зависит от существования сильных.

Атрибут – свойство сущности. Различают:

1) *Идентифицирующие и описательные атрибуты*. Идентифицирующие позволяют отличить один экземпляр сущности от другого. Описательные атрибуты заключают в себе интересующие нас свойства сущности.
2) *Составные и простые атрибуты*. Простой атрибут имеет неделимое значение. Составной атрибут является комбинацией нескольких элементов, возможно, принадлежащих разным типам данных (ФИО, адрес и др.).
3) *Однозначные и многозначные атрибуты* (могут иметь соответственно одно или много значений для каждого экземпляра сущности). Например, дата рождения – это однозначный атрибут, а номер телефона – многозначный.

4) *Основные и производные атрибуты*. Значение основного атрибута не зависит от других атрибутов; значение производного атрибута вычисляется на основе значений других атрибутов. Например, возраст вычисляется на основе даты рождения и текущей даты.
5) *Обязательные и необязательные* (первые должны быть указаны при размещении данных в БД, вторые могут не указываться).

Для каждого атрибута необходимо определить название, указать тип данных и описать ограничения целостности – множество значений, которые может принимать данный атрибут.

Связь – это осмысленная ассоциация между сущностями. Для связи указывается название, тип (факультативная или обязательная), кардинальность (1:1, 1:n или m:n) и степень (унарная, бинарная, тернарная или n-арная).

На Рис. 1 приведены обозначения, которые мы будем использовать в ER-диаграммах.

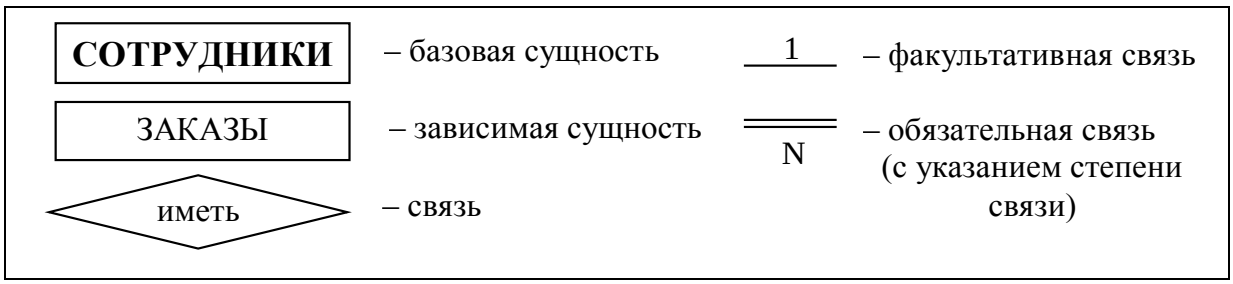


Рис. 1. Обозначения, используемые в ER-диаграммах

1.2.2. Определение требований к операционной обстановке

На этом этапе производится оценка требований к вычислительным ресурсам, необходимым для функционирования системы, определение типа и конфигурации конкретной ЭВМ, выбор типа и версии операционной системы. Объём вычислительных ресурсов зависит от предполагаемого объёма проектируемой базы данных и от интенсивности их использования. Если БД будет работать в многопользовательском режиме, то требуется подключение её к сети и наличие соответствующей многозадачной операционной системы [1].

1.2.3. Выбор СУБД и других программных средств

Выбор СУБД осуществляется на основании таких критериев, как тип модели данных и её адекватность потребностям рассматриваемой ПрО; характеристики производительности; набор функциональных возможностей; удобство и надежность СУБД в эксплуатации; стоимость СУБД и дополнительного программного обеспечения [1].

1.2.4. Логическое проектирование реляционной БД

На этапе логического проектирования разрабатывается логическая (концептуальная) структура БД. Для реляционной модели существуют формальные правила, которые позволяют преобразовать инфологическую модель ПрО в виде ER-диаграммы в логическую схему базы данных. Кроме получения схемы БД в целом на этом этапе выполняют создание схем

отношений и их нормализацию. Этот этап более подробно рассмотрен в п.2 "Пример проектирования реляционной базы данных".

1.2.5. Физическое проектирование БД

Этап физического проектирования заключается в определении схемы хранения, т.е. физической структуры БД. Схема хранения зависит от той физической структуры, которую поддерживает выбранная СУБД. Физическая структура БД, с одной стороны, должна адекватно отражать логическую структуру БД, а с другой стороны, должна обеспечивать эффективное размещение данных и быстрый доступ к ним. Результаты этого этапа документируются в форме схемы хранения на языке определения данных (DDL, Data Definition Language) выбранной СУБД. Принятые на этом этапе решения оказывают огромное влияние на производительность системы.

Одной из важнейших составляющих проекта базы данных является разработка средств защиты БД. Защита данных имеет два аспекта: защита от сбоев и защита от несанкционированного доступа. Для защиты от сбоев на этапе физического проектирования разрабатывается стратегия резервного копирования. Для защиты от несанкционированного доступа каждому пользователю доступ к данным предоставляется только в соответствии с его правами доступа, набор которых также является составной частью проекта БД.

Используемые термины

Атрибут — свойство некоторой сущности. Часто называется полем таблицы.
Домен атрибута — множество допустимых значений, которые может принимать атрибут.
Кортеж — конечное множество взаимосвязанных допустимых значений атрибутов, которые вместе описывают некоторую сущность (строка таблицы).
Отношение — конечное множество кортежей (таблица).
Схема отношения — конечное множество атрибутов, определяющих некоторую сущность. Иными словами, это структура таблицы, состоящей из конкретного набора полей.
Проекция — отношение, полученное из заданного путём удаления и (или) перестановки некоторых атрибутов.
Функциональная зависимость между атрибутами (множествами атрибутов) X и Y означает, что для любого допустимого набора кортежей в данном отношении: если два кортежа совпадают по значению X, то они совпадают по значению Y. Например, если значение атрибута «Название компании» — Canonical Ltd, то значением атрибута «Штаб-квартира» в таком кортеже всегда будет Millbank Tower, London, United Kingdom. Обозначение: {X} -> {Y}.
Нормальная форма — требование, предъявляемое к структуре таблиц в теории реляционных баз данных для устранения из базы избыточных функциональных зависимостей между атрибутами (полями таблиц).
Метод нормальных форм (НФ) состоит в сборе информации о объектах решения задачи в рамках одного отношения и последующей декомпозиции этого отношения на несколько взаимосвязанных отношений на основе процедур нормализации отношений.

Цель нормализации: исключить избыточное дублирование данных, которое является причиной аномалий, возникших при добавлении, редактировании и удалении кортежей(строк таблицы).
Аномалией называется такая ситуация в таблице БД, которая приводит к противоречию в БД либо существенно усложняет обработку БД. Причиной является излишнее дублирование данных в таблице, которое вызывается наличием функциональных зависимостей от не ключевых атрибутов.
Аномалии-модификации проявляются в том, что изменение одних данных может повлечь просмотр всей таблицы и соответствующее изменение некоторых записей таблицы.
Аномалии-удаления — при удалении какого либо кортежа из таблицы может пропасть информация, которая не связана на прямую с удаляемой записью.
Аномалии-добавления возникают, когда информацию в таблицу нельзя поместить, пока она не полная, либо вставка записи требует дополнительного просмотра таблицы.

1.3. Особенности проектирования реляционной базы данных

Проектирование реляционной базы данных проходит в том же порядке, что и проектирование БД других моделей данных, но имеет свои особенности.

Проектирование схемы БД должно решать задачи минимизации дублирования данных и упрощения процедур их обработки и обновления. При неправильно спроектированной схеме БД могут возникнуть аномалии модификации данных. Они обусловлены отсутствием средств явного представления типов множественных связей между объектами ПрО и неразвитостью средств описания ограничений целостности на уровне модели данных.

Для решения подобных проблем проводится **нормализация отношений.**

Механизм нормализации реляционных отношений разработал Э.Ф. Кодд (E.F. Codd). Этот механизм позволяет *по формальным признакам* любое отношение преобразовать к третьей нормальной форме.

Нормализация схемы отношения выполняется путём декомпозиции схемы. **Декомпозицией** схемы отношения R называется замена её совокупностью схем отношений A_i таких, что

$$R = \bigcup_i A_i ,$$

и не требуется, чтобы отношения A_i были непересекающимися по атрибутам.

Первая нормальная форма относится к понятию простого и сложного (составного или многозначного) атрибута (см. п.1.2.1).

Первая нормальная форма (1НФ).

Отношение приведено к 1НФ, если все его атрибуты простые.

Для того чтобы привести к 1НФ отношение, содержащее сложные атрибуты, нужно:

1) разбить составные атрибуты на простые,
2) построить декартово произведение всех многозначных атрибутов с кортежами, к которым они относятся.

Для идентификации кортежа в этом случае понадобится составной ключ, включающий первичный ключ исходного отношения и все многозначные атрибуты.

Вторая нормальная форма основана на понятии *функциональной зависимости*. Пусть X и Y – атрибуты некоторого отношения. Если в любой момент времени каждому значению X соответствует единственное значение Y, то говорят, что Y функционально зависит от X (X→Y). Атрибут X в функциональной зависимости X→Y называется *детерминантом* отношения.

В нормализованном отношении все неключевые атрибуты функционально зависят от ключа отношения. Неключевой атрибут функционально полно зависит от составного ключа, если он функционально зависит от ключа, но не находится в функциональной зависимости ни от какой части составного ключа.

Вторая нормальная форма (2НФ).

Отношение находится во 2НФ, если оно приведено к 1НФ и каждый неключевой атрибут функционально полно зависит от составного первичного ключа.

(Таким образом, если отношение в 1НФ имеет простой первичный ключ, оно сразу находится во второй нормальной форме).

Для того чтобы привести отношение ко 2НФ, нужно:

- построить его проекцию, исключив атрибуты, которые не находятся в функционально полной зависимости от составного первичного ключа;
- построить дополнительно одну или несколько проекций на часть составного ключа и атрибуты, функционально зависящие от этой части ключа.

Третья нормальная форма основана на понятии *транзитивной зависимости*. Пусть X, Y, Z – атрибуты некоторого отношения. При этом X→Y и Y→Z, но обратное соответствие отсутствует, т.е. Z не зависит от Y или Y не зависит от X. Тогда говорят, что Z транзитивно зависит от X (X→→Z).

Третья нормальная форма (3НФ).

Отношение находится в 3НФ, если оно находится во 2НФ и каждый неключевой атрибут нетранзитивно зависит от первичного ключа.

Для того чтобы привести отношение к 3НФ, нужно:

- построить проекцию, исключив транзитивно зависящие от первичного ключа атрибуты;
- построить дополнительно одну или несколько проекций на детерминанты исходного отношения и атрибуты, функционально зависящие от них.

Исключение составляют случаи, когда для транзитивной зависимости X→→Z (X→Y и Y→Z) либо Z зависит от Y, либо Y зависит от X, т.е. между атрибутами X и Y, например, существует связь 1:1. В такой ситуации декомпозиция отношения не производится.

Четвертая нормальная форма основана на понятии *многозначной зависимости*. Многозначная зависимость существует, если заданным значениям атрибута X соответствует множество, состоящее из нуля (или более) значений атрибута Y (X–»Y).

Различают тривиальные и нетривиальные многозначные зависимости. *Тривиальной* называется такая многозначная зависимость X–»Y, для которой $Y \subset X$ или $X \cup Y = R$, где R – рассматриваемое отношение. Тривиальная многозначная зависимость не нарушает 4НФ. Если хотя бы одно из двух этих условий не выполняется, то такая зависимость называется *нетривиальной*.

Четвертая нормальная форма (4НФ).

Отношение находится в 4НФ, если оно находится в 3НФ и в нём отсутствуют нетривиальные многозначные зависимости.

Для того чтобы привести отношение к 4НФ, нужно построить две или более проекции исходного отношения, каждая из которых содержит ключ и одну из многозначных зависимостей.

Формальное описание нормализации достаточно подробно изложено в [1], поэтому, чтобы не повторяться, будем рассматривать этот метод на примере проектирования конкретной базы данных. Нормализацию будем проводить в **два этапа**:

I. 1) Разобьем составные атрибуты на простые.
   2) Вынесем многозначные атрибуты в отдельные отношения.
II. Найдем транзитивные зависимости и вынесем их в отдельные отношения.

2. Пример проектирования реляционной базы данных

В качестве примера возьмем базу данных проектной организации. Основной вид деятельности такой организации – выполнение проектов по договорам с заказчиками.

2.1. Инфологическое проектирование

2.1.1. Анализ предметной области

База данных создаётся для информационного обслуживания руководства организации, руководителей проектов и участников проектов. БД должна содержать данные об отделах организации, сотрудниках и проектах.

В соответствии с предметной областью система строится с учётом следующих особенностей:

- Каждый сотрудник работает в определённом отделе, в каждом отделе могут работать несколько сотрудников.
- Каждый проект относится к определённому отделу, каждый отдел может отвечать за выполнение нескольких проектов.
- Каждый сотрудник может принимать участие в выполнении нескольких проектов, над каждым проектом может трудиться несколько сотрудников.
- Для каждого проекта назначается руководитель из числа сотрудников того отдела, к которому относится проект.
- Каждый проект должен быть выполнен в заданные сроки, каждый проект может состоять из нескольких этапов. Если проект состоит из одного этапа, то сроки его выполнения должны совпадать со сроками выполнения проекта в целом.
- Оклад сотрудника зависит от занимаемой должности, за участие в проектах сотрудник получает дополнительное вознаграждение.

– Виды участия сотрудников в проектах: руководитель, консультант, исполнитель.
– Каждый отдел занимает одно или несколько помещений (комнат), в каждом помещении может быть один или несколько стационарных телефонов.

Примечание. Описания особенностей ПрО должно быть достаточно для того, чтобы создать ER–диаграмму.

Для создания ER-модели необходимо выделить сущности предметной области и указать их атрибуты. **Идентифицирующие** атрибуты мы выделим полужирным шрифтом, *многозначные* – курсивом, составные подчеркнем:

1) **Отделы**. Атрибуты: **название**, **аббревиатура**, *комнаты*, *телефоны*.

2) **Сотрудники**. Атрибуты: ФИО, **паспортные данные**, дата рождения, пол, **ИНН** (индивидуальный номер налогоплательщика), **СНИЛС** (страховой номер индивидуального лицевого счета), *адреса*, *телефоны* (мобильный, рабочий, домашний), *данные об образовании* (вид образования (высшее, средне-специальное и т.д.), специальность, номер диплома, дата окончания учебного заведения), должность, оклад, **логин** (имя пользователя).

   Каждый сотрудник работает в определенном отделе, может руководить проектами и участвовать в их выполнении.

Примечания: 1. Логин потребуется нам для назначения дифференцированных прав доступа.
2. В нашем задании не предусмотрена полная информационная поддержка сотрудников отдела кадров, поэтому мы не будем отражать в БД такие сведения как переводы сотрудника с одной должности на другую, уходы в отпуска, больничные и т.п.

3) **Проекты**. Атрибуты: **номер договора; полное название проекта; сокращённое название проекта**; дата подписания договора; заказчик; контактные данные заказчика; дата начала проекта; дата завершения проекта; сумма по проекту; дата реальной сдачи проекта; сумма, полученная по проекту на текущую дату.

   За каждый проект отвечает определенный отдел, у каждого проекта есть руководитель из числа сотрудников этого отдела.

4) **Этапы проекта**. Атрибуты: номер по порядку, название, дата начала этапа, дата завершения этапа, *форма отчетности* (перечень технической документации), сумма по этапу, дата реальной сдачи этапа; сумма, полученная по этапу на текущую дату.

   Каждый этап относится к определенному проекту.

Обратите внимание! У сущности нет атрибутов других сущностей, это реализуется через связь. Связи как устойчивые ассоциации между сущностями являются неотъемлемой частью предметной области. Просто отражаются они не с помощью атрибутов сущностей, а с помощью специальных элементов – **связей.**

Например, каждый сотрудник работает в определенном отделе, но это не значит, что номер отдела является его личной характеристикой: это его связь с сущностью *Отделы*, которая, кстати, может измениться – сотрудник может перейти в другой отдел.

Метод сущность-связь не зависит от модели данных. На основе ER-диаграммы можно спроектировать не только реляционную БД, но и любую другую. Но только в реляционной базе данных связи реализуются с помощью атрибутов – так называемых внешних ключей. А в сетевой модели данных или в объектно-ориентированной, например, связь – это физическая ссылка от одной записи к другой, и никаких "чужих" атрибутов в этих записях нет.

Исходя из выявленных сущностей, построим ER–диаграмму (Рис. 2). Напомним, что пометки у линий отражают кардинальность связи: 1:1, 1:N и N:M.

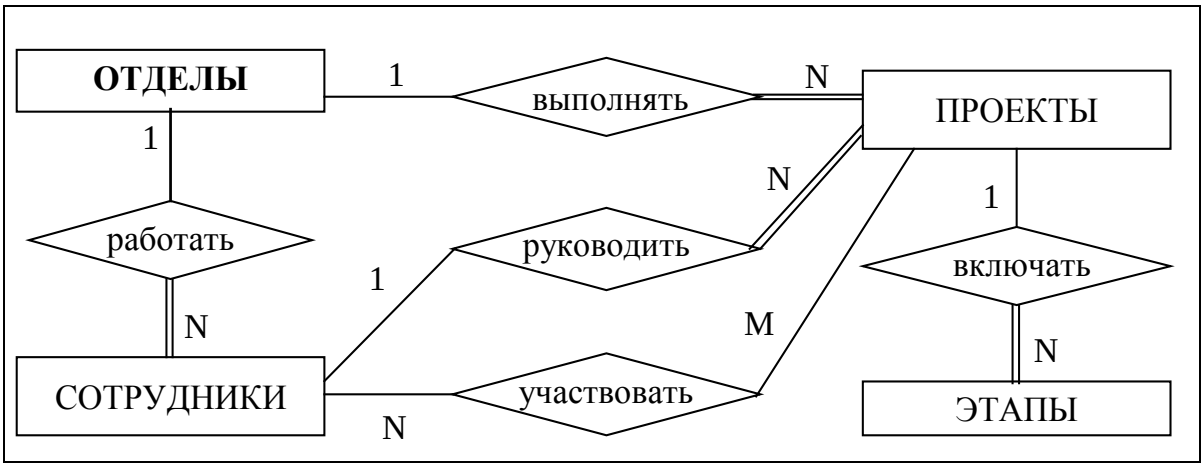


Рис. 2. ER–диаграмма ПрО «Проектная организация»

2.1.2. Анализ информационных задач и круга пользователей системы

Определим группы пользователей, их основные задачи и запросы к БД:

1. Руководитель организации:
   - заключение новых договоров;
   - назначение руководителей проектов;
   - получение списка всех участников проектов;
   - изменение должностных окладов;
   - получение полной информации о проектах;
   - внесение изменений в данные о проектах.
2. Заместитель руководителя организации:
   - изменение штатного расписания;
   - проверка и исправление данных о заключенных договорах;
   - формирование и актуализация справочных таблиц.
3. Руководитель проекта:
   - назначение участников проекта;
   - получение списка сотрудников, работающих над конкретным проектом;
   - получение полной информации о проекте, руководителем которого он является;

- получение сведений о сотрудниках, которые могут стать участниками проекта;
- определение размера дополнительного вознаграждения сотрудников по конкретному проекту;
- внесение изменений в данные об этапах проекта.

4. Сотрудники отдела кадров:
   - приём/увольнение сотрудников;
   - внесение изменений в данные о сотрудниках.
5. Бухгалтеры:
   - получение ведомости на выплату зарплаты.
6. Сотрудники – участники проектов:
   - просмотр данных о других участниках проекта;
   - просмотр данных о сроках сдачи проекта и форме отчётности.

2.2. Определение требований к операционной обстановке

Для выполнения этого этапа необходимо знать (хотя бы ориентировочно) объём работы организации (т.е. количество проектов и сотрудников), а также иметь представление о характере и интенсивности запросов.

Объём внешней памяти, необходимый для функционирования системы, складывается из двух составляющих: память, занимаемая модулями СУБД (ядро, утилиты, вспомогательные программы), и память, отводимая под данные ($М_Д$). Для реальных баз данных обычно наиболее существенным является $М_Д$.

На основе результатов анализа ПрО можно приблизительно оценить объём памяти, требуемой для хранения данных. Примем ориентировочно, что:

- одновременно осуществляется около десяти проектов, работа над проектом продолжается в среднем год (по 1К на каждый проект);
- каждый проект состоит в среднем из четырёх этапов (по 0,5К на этап);
- в компании работают 100 сотрудников (по 0,5К на каждого сотрудника);
- в выполнении каждого проекта в среднем участвуют 10 сотрудников (по 0,2К);
- объем технической документации составляет 100М на каждый проект.

Тогда объём памяти для хранения данных за первый год примерно составит:

$$М_д = 2(10*1+10*4*0,5+100*0,5+(10*10*0,2)+10*20000) \approx 1\ G,$$

Коэффициент 2 необходим для того, чтобы учесть необходимость выделения памяти под дополнительные структуры (например, индексы). Объём памяти будет увеличиваться ежегодно на столько же при сохранении объёма работы.

Требуемый объём оперативной памяти определяется на основании анализа интенсивности запросов и объёма результирующих данных. Для нашей БД требуемый объём памяти невелик, поэтому никаких специальных требований к объёму внешней и оперативной памяти компьютера не предъявляется.

2.3. Выбор СУБД и других программных средств

Анализ информационных задач показывает, что для реализации требуемых функций подходят почти все СУБД для ПЭВМ (PostgreSQL, MS

Access, Firebird, MySQL и др.). Все они поддерживают реляционную модель данных и предоставляют разнообразные возможности для работы с данными.

Объём внешней и оперативной памяти, требующийся для функционирования СУБД, обычно указывается в сопроводительной документации.

Для того чтобы в учебном примере не привязываться к конкретной СУБД, выполним описание логической схемы БД на SQL-92.

2.4. Логическое проектирование реляционной БД

2.4.1. Преобразование ER–диаграммы в схему базы данных

База данных создаётся на основании схемы базы данных. Преобразование ER–диаграммы в схему БД выполняется путем сопоставления каждой сущности и каждой связи, имеющей атрибуты, отношения (таблицы) БД. Связь типа 1:n (один-ко-многим) между отношениями реализуется через внешний ключ. Ключ вводится для дочернего отношения. Внешнему ключу должен соответствовать первичный или уникальный ключ основного (родительского) отношения.

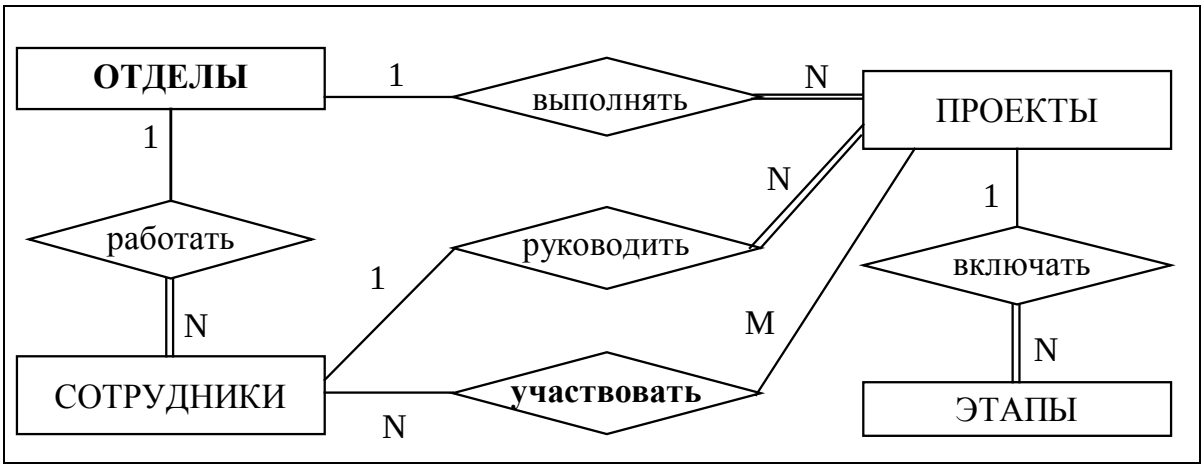


Рис. 3. ER–диаграмма ПрО «Проектная организация»

Связь *участвовать* между *ПРОЕКТАМИ* и *СОТРУДНИКАМИ* принадлежит к типу N:M (Рис. 3). Этот тип связи реализуется через вспомогательное отношение *Участие*, которое содержит комбинации первичных ключей соответствующих исходных отношений.

Более подробно о принципах преобразования ER-диаграммы в схему БД рассказано в [1]. Для схемы БД будем использовать обозначения, представленные на Рис. 4.

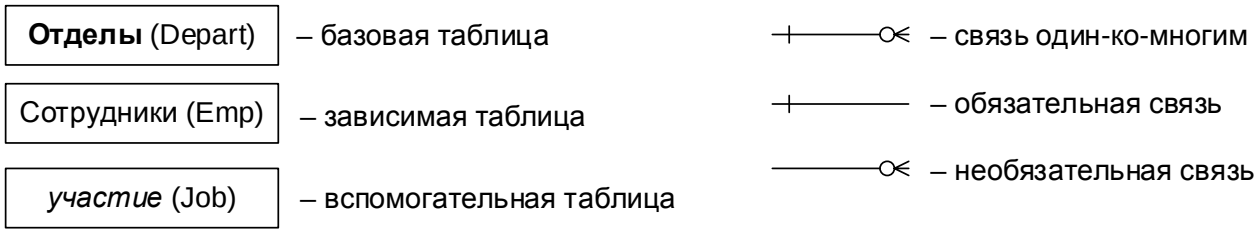


Рис. 4. Обозначения, используемые на схеме базы данных

Полученная схема реляционной базы данных (РБД) приведена на Рис. 5.

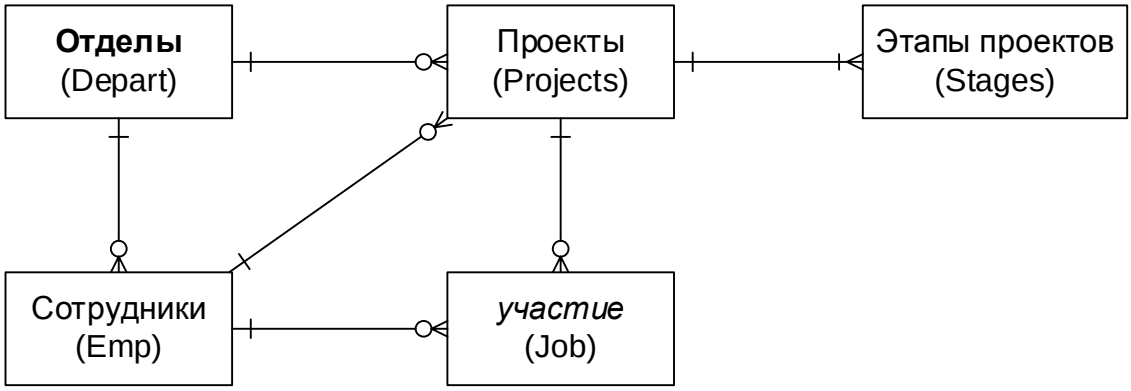


Рис. 5. Схема РБД, полученная из ER–диаграммы проектной организации

Бинарная связь между отношениями не может быть обязательной для обоих отношений. Такой тип связи означает, что, например, прежде чем добавить новый проект в отношение *ПРОЕКТЫ*, нужно добавить новую строку в отношение *ЭТАПЫ ПРОЕКТОВ*, и наоборот. Поэтому для такой связи необходимо снять с одной стороны условие обязательности. Так как все эти связи будут реализованы с помощью внешнего ключа, снимем условие обязательности связей для отношений, содержащих первичные ключи (Рис. 6). Кроме того, сотрудник может принимать участие не во всем проекте, а в его отдельных этапах; поэтому мы изменим связь между таблицами *ПРОЕКТЫ* и *УЧАСТИЕ* на связь между таблицами *ЭТАПЫ ПРОЕКТОВ* и *УЧАСТИЕ*.

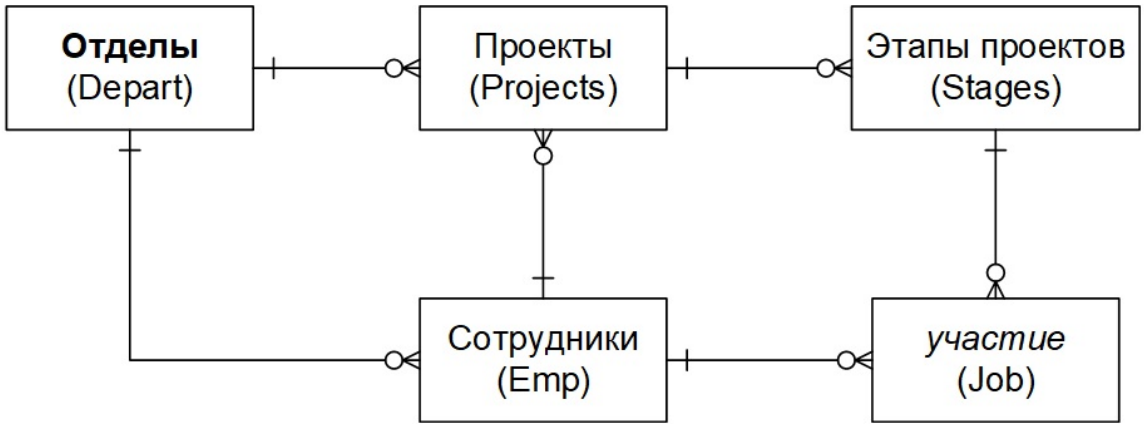


Рис. 6. Схема РБД без нереализуемых связей

2.4.2. Составление реляционных отношений

Каждое реляционное отношение соответствует одной сущности (объекту ПрО) и в него вносятся все атрибуты этой сущности. Для каждого отношения определяются первичный ключ и внешние ключи (в соответствии со схемой БД). В том случае, если базовое отношение не имеет потенциальных ключей, вводится ***суррогатный первичный ключ***, который не несёт смысловой нагрузки и служит только для идентификации записей.

Отношения приведены в **Таблица 1**-5. Для каждого отношения указаны атрибуты с их внутренним названием, типом и длиной. Типы данных обозначаются так: N – числовой, C – символьный тип фиксированной длины, V – символьный тип переменной длины, D – дата (этот тип имеет стандартную

длину, зависящую от СУБД, поэтому она не указывается). О правилах выбора типов данных подробно рассказано в [1].

Потенциальными ключами отношения *ОТДЕЛЫ* являются атрибуты *Аббревиатура* и *Название отдела*. Первый занимает меньше места, поэтому мы выбираем его в качестве первичного ключа.

Таблица 1. Схема отношения *ОТДЕЛЫ* (Departs)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Аббревиатура отдела	D_ID	C(10)	**первичный ключ**
Название отдела	D_NAME	V(100)	обязательное поле
Комнаты	D_ROOMS	V(20)	обязательное многозначное поле
Телефоны	D_PHONE	V(40)	обязательное многозначное поле

Потенциальными ключами отношения *СОТРУДНИКИ* являются поля *Паспортные данные*, *ИНН* и *СНИЛС*. Все они занимают достаточно много места, а паспортные данные кроме того могут меняться. Введём суррогатный первичный ключ *Номер* (номер сотрудника).

Таблица 2. Схема отношения *СОТРУДНИКИ* (Employees)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Номер	E_ID	N(4)	**суррогатный первичный ключ**
Фамилия, имя, отчество	E_NAME	V(50)	обязательное поле
Дата рождения	E_BORN	D	обязательное поле
Пол	E_GENDER	C(1)	обязательное поле, 'м' или 'ж'
Паспортные данные	E_PASP	V(50)	обязательное поле
ИНН	E_INN	C(12)	обязательное уникальное поле
СНИЛС	E_SNILS	C(14)	обязательное уникальное поле
Отдел	E_DEPART	C(10)	внешний ключ (к Departs)
Должность	E_POST	V(30)	обязательное поле
Оклад	E_SAL	N(8,2)	обязательное поле, > 4500 руб.
Данные об образовании	E_EDU	V(200)	обязательное многозначное поле
Адреса	E_ADDR	V(100)	многозначное поле
Телефоны	E_PHONE	V(30)	многозначное поле

Логин	E_LOGIN	V(30)	

Примечание. Суррогатный (искусственный) первичный ключ также может вводиться в тех случаях, когда потенциальный ключ имеет большой размер (например, длинная символьная строка) или является составным (не менее трёх атрибутов).

В отношении *ПРОЕКТЫ* три потенциальных ключа: *Номер проекта*, *Название проекта* и *Сокращённое названиие*. Меньше места занимает первый из них, но он малоинформативен. Зато сокращённое название, используемое в качестве внешнего ключа в других таблицах, позволит специалисту идентифицировать проект без необходимости соединения с отношением *ПРОЕКТЫ*.

Таблица 3. Схема отношения *ПРОЕКТЫ* (Projects)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Номер проекта	P_ID	N(6)	обязательное уникальное поле
Название проекта	P_TITLE	V(100)	обязательное поле
Сокращённое название	P_ABBR	C(10)	**первичный ключ**
Отдел	P_DEPART	C(10)	внешний ключ (к Departs)
Заказчик	P_COMPANY	V(40)	обязательное поле
Данные заказчика	P_LINKS	V(200)	обязательное поле
Руководитель	P_CHIEF	N(4)	внешний ключ (к Employees)
Дата начала проекта	P_BEGIN	D	обязательное поле
Дата окончания проекта	P_END	D	обязательное поле, больше даты начала проекта
Реальная дата окончания	P_FINISH	D	
Стоимость проекта	P_COST	N(10)	обязательное поле
Полученная сумма	P_SUM	N(10)	обязательное поле, значение по умолчанию – 0

Потенциальным ключом отношения *ЭТАПЫ* является комбинация внешнего ключа и номера этапа, а потенциальным ключом вспомогательного отношения *УЧАСТИЕ* является комбинация первых трёх полей этого отношения. Можно вообще не вводить первичный ключ для данных отношений, т.к. на них никто не ссылается. Но уникальность этих комбинации является в данном случае ограничением целостности данных, поэтому мы возьмём эти комбинации в качестве первичных ключей соответствующих отношений.

Таблица 4. Схема отношения *ЭТАПЫ ПРОЕКТА* (Stages)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*	
Проект	S_PRO	C(10)	внешний ключ (к Projects)	**составной первичный ключ**
Номер этапа	S_NUM	N(2)		
Название этапа	S_TITLE	V(200)	обязательное поле	
Дата начала этапа	S_BEGIN	D	обязательное поле	
Дата окончания этапа	S_END	D	обязательное поле, > даты начала	
Реальная дата окончания	S_FINISH	D	больше даты начала этапа	
Стоимость этапа	S_COST	N(10)	обязательное поле	
Полученная сумма по этапу	S_SUM	N(10)	обязательное поле, значение по умолчанию – 0	
Форма отчётности	S_FORM	V(300)	обязательное многозначное поле	

В состав формы отчётности входит техническая документация, которая занимает большой объём памяти. Её можно хранить в базе данных (типы LONG, BLOB), но мы будем хранить только ссылки на файлы с документами.

Таблица 5. Схема отношения *УЧАСТИЕ* (Job)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Проект	J_PRO	C(10)	составной внешний ключ (к Stages)
Номер этапа	J_NUM	N(2)	
Сотрудник	J_EMP	N(4)	внешний ключ (к Employees)
Роль	J_ROLE	V(20)	'консультант' или 'исполнитель'
Доплата	J_BONUS	N(2)	>=0, по умолчанию 0
Начало участия	J_BEGIN	D	обязательное поле
Окончание участия	J_END	D	

2.4.3. Нормализация полученных отношений (до 4НФ)

Механизм нормализации подразумевает определённую последовательность преобразования отношений к третьей нормальной форме. Мы не будем чётко придерживаться этой последовательности, т.к. она избыточна, и многозначные атрибуты сразу вынесем в отдельные отношения на первом же этапе.

Для приведения таблиц к **1НФ** разобьем сложные (составные) атрибуты на простые и вынесем многозначные атрибуты в отдельные отношения.

Примечание. В реальных БД составные атрибуты разбиваются на простые в следующих случаях:

а) необходимо установить ограничения целостности на отдельные части составного атрибута;

б) этого требует внешнее представление данных;

б) в запросах поиск может осуществляться по отдельной части атрибута.

ОТДЕЛЫ. Многозначные атрибуты *Комнаты* и *Телефоны* вынесем в отдельное отношение *КОМНАТЫ*. Так как в комнате может не быть стационарного телефона, первичный ключ отношения *КОМНАТЫ* не определен (ПК не может содержать null-значения), но на этих атрибутах можно определить составной уникальный ключ. После нормализации отношение *ОТДЕЛЫ* превратится в два отношения, которые приведены в Таблица 6-7.

Таблица 6. Схема отношения *ОТДЕЛЫ* (Departs)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Аббревиатура отдела	D_ID	V(12)	**первичный ключ**
Название отдела	D_NAME	V(100)	обязательное поле

Таблица 7. Схема отношения *КОМНАТЫ* (Rooms)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Отдел	R_DEPART	V(12)	внешний ключ (к Departs)
Номер комнаты	R_ROOM	N(4)	**составной уникальный ключ**
Телефон	R_PHONE	V(20)	

СОТРУДНИКИ. Разделим составной атрибут *Фамилия, имя, отчество* на два атрибута *Фамилия* и *Имя, отчество*, составной атрибут *Паспортные данные* на *Серия и номер паспорта* (уникальный), *Дата выдачи* и *Кем выдан*. Составной многозначный атрибут *Данные об образовании* вынесем в отдельное отношение и разделим на *Вид образования*, *Специальность*, *Номер диплома* и *Год окончания учебного заведения*.

Домашние и мобильные телефоны и адреса сотрудников вынесем в отношение *АДРЕСА-ТЕЛЕФОНЫ*. В отношении *АДРЕСА-ТЕЛЕФОНЫ* нет потенциальных ключей: оставим это отношение без первичного ключа, т.к. на него никто не ссылается. Что касается рабочих телефонов сотрудников, то один из этих номеров – основной – определяется рабочим местом сотрудника (мы учитываем только стационарные телефоны). Будем хранить этот номер в атрибуте *Рабочий телефон*. Наличие других номеров зависит от того, есть ли в том же помещении (комнате) другие сотрудники, имеющие стационарные телефоны. Добавим в отношение *СОТРУДНИКИ* атрибут *Номер комнаты*, чтобы можно было ссылаться на отношение *КОМНАТЫ*, а дополнительные номера телефонов сотрудника можно было вычислить из других кортежей с таким же номером комнаты. Связь между отношениями *СОТРУДНИКИ* и *КОМНАТЫ* реализуем через составной внешний ключ (*Номер комнаты*, *Рабочий телефон*).

Для приведения отношения ***СОТРУДНИКИ*** к **3НФ** найдем транзитивные зависимости. Атрибут *Оклад* зависит от атрибута *Должность*. Создадим

отношение *ДОЛЖНОСТИ*, перенесём в него атрибуты *Должность* и *Оклад*, а первичным ключом сделаем название должности (Таблица 8).

Таблица 8. Схема отношения *ДОЛЖНОСТИ* (Posts)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Название должности	P_POST	V(30)	**первичный ключ**
Оклад	P_SAL	N(8,2)	обязательное поле, > 12000 руб.

Таблица 9. Схема отношения *СОТРУДНИКИ* (Employees)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Идентификатор сотрудника	E_ID	N(4)	**суррогатный первичный ключ**
Фамилия	E_FNAME	V(25)	обязательное поле
Имя, отчество	E_LNAME	V(30)	обязательное поле
Дата рождения	E_BORN	D	обязательное поле
Пол	E_GENDER	C(1)	обязательное поле
Серия и номер паспорта	E_PASP	C(10)	обязательное уникальное поле
Когда выдан паспорт	E_DATE	D	обязательное поле
Кем выдан паспорт	E_GIVEN	V(50)	обязательное поле
ИНН	E_INN	C(12)	обязательное уникальное поле
СНИЛС	E_SNILS	C(14)	обязательное уникальное поле
Отдел	E_DEPART	V(12)	внешний ключ (к Departs)
Должность	E_POST	V(30)	внешний ключ (к Posts)
Номер комнаты	E_ROOM	N(4)	составной внешний ключ (к Rooms)
Рабочий телефон	E_PHONE	V(20)	
Логин	E_LOGIN	V(30)	

В отношениях ***СОТРУДНИКИ*** и ***ОБРАЗОВАНИЕ*** атрибуты (*Дата выдачи* и *Кем выдан*) и (*Номер диплома* и *Год окончания учебного заведения*) зависят не от первичного ключа, а от атрибутов соответственно *Номер паспорта* и *Специальность*. Но если мы выделим их в отдельные отношения, то получим связи типа 1:1. Следовательно, эта декомпозиция нецелесообразна.

Также мы создадим справочную таблицу ***ВИДЫ ОБРАЗОВАНИЯ***, Эта таблица обеспечит одинаковое написание значений соответствующего поля таблицы ***ОБРАЗОВАНИЕ***, и возможность добавлять и изменять их при необходимости. Отношения, получившиеся при вынесении многозначных полей из отношения ***СОТРУДНИКИ***, приведены в Таблица 10-12.

Таблица 10. Схема отношения *ВИДЫ ОБРАЗОВАНИЯ* (Types)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Название вида образования	T_NAME	V(20)	**первичный ключ**

Таблица 11. Схема отношения *ОБРАЗОВАНИЕ* (Edu)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Идентификатор сотрудника	U_ID	N(4)	внешний ключ (к Employees)
Вид образования	U_TYPE	V(20)	обязательное поле, внешний ключ (к Types)
Специальность	U_SPEC	V(40)	
Номер диплома	U_DIPLOM	V(15)	
Год окончания учебного заведения	U_YEAR	N(4)	обязательное поле

Таблица 12. Схема отношения *АДРЕСА-ТЕЛЕФОНЫ* (AdrTel)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Идентификатор сотрудника	A_ID	N(4)	внешний ключ (к Employees)
Адрес	A_ADDR	V(50)	
Телефон	A_PHONE	V(30)	

Отношение *АДРЕСА-ТЕЛЕФОНЫ* нарушает **4НФ**, т.к. не всякий телефон привязан к конкретному адресу (т.е. мы имеем две многозначных зависимости в одном отношении). Но выделять *Телефоны* в отдельное отношение не стоит, т.к. эти сведения носят справочный характер и не требуется их автоматическая обработка. Отношения *ОБРАЗОВАНИЕ* и *АДРЕСА-ТЕЛЕФОНЫ* не имеют потенциальных ключей, но мы не будем вводить суррогатные первичные ключи, т.к. на эти таблицы никто не ссылается. Но для таблицы АДРЕСА-ТЕЛЕФОНЫ введем ограничение, в соответствии с которым или адрес, или телефон должен быть указан.

ПРОЕКТЫ. Удалим вычислимый атрибут *Полученная сумма*, т.к. он является вычислимым – это сумма значений аналогичного атрибута из отношения ***ЭТАПЫ ПРОЕКТОВ***. Но атрибут *Стоимость проекта* оставим, т.к. она фигурирует в документации по проекту. Для обеспечения логической целостности данных необходимо предусмотреть в приложении проверку того, что сумма стоимостей по всем этапам совпадает с общей стоимостью проекта. Атрибут *Данные заказчика* зависит от атрибута *Заказчик*, а не от первичного ключа, поэтому их следует вынести в отдельное отношение *ЗАКАЗЧИКИ*. Но при этом первичным ключом нового отношения станет атрибут *Заказчик*, т.е. длинная символьная строка. Целесообразнее ввести суррогатный ПК. Так как с каждым заказчиком может быть связано несколько проектов, связь между отношениями ***ПРОЕКТЫ*** и ***ЗАКАЗЧИКИ*** будет 1:N и суррогатный ПК станет внешним ключом для отношения ***ПРОЕКТЫ*** (Таблица 13-14).

Таблица 13. Схема отношения *ЗАКАЗЧИКИ* (Clients)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Номер заказчика	C_ID	N(4)	**суррогатный первичный ключ**
Заказчик	C_COMPANY	V(40)	обязательное поле
Адрес заказчика	C_ADR	V(50)	обязательное поле
Контактное лицо	C_PERSON	V(50)	обязательное поле
Телефон	C_PHONE	V(30)	

Таблица 14. Схема отношения *ПРОЕКТЫ* (Projects)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Номер проекта	P_ID	N(6)	обязательное уникальное поле
Название проекта	P_TITLE	V(100)	обязательное поле
Сокращённое название	P_ABBR	C(10)	**первичный ключ**
Отдел	P_DEPART	V(12)	внешний ключ (к Departs)
Заказчик	P_COMPANY	N(4)	внешний ключ (к Clients)
Руководитель	P_CHIEF	N(4)	внешний ключ (к Employees)
Дата начала проекта	P_BEGIN	D	обязательное поле
Дата окончания проекта	P_END	D	обязательное поле, больше даты начала проекта
Реальная дата окончания	P_FINISH	D	больше даты начала проекта
Стоимость проекта	P_COST	N(10)	обязательное поле, > 0

***ЭТАПЫ ПРОЕКТОВ*.** Многозначный составной атрибут *Формы отчетности* вынесем в отдельное отношение *ОТЧЁТЫ* и разделим на поля *Номер ГОСТа*, *Название отчёта*, *Ссылка на файл отчёта*.

В отношении ***ОТЧЁТЫ*** атрибут *Название отчёта* зависит от *Номера ГОСТа*, поэтому вы создадим справочную таблицу *ФОРМЫ ОТЧЕТНОСТИ*, в которой *Номер ГОСТа* будет первичным ключом.

Таблица 15. Схема отношения *ФОРМЫ ОТЧЕТНОСТИ* (Forms)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Номер ГОСТа	F_GOST	V(20)	**первичный ключ**

Название	F_TITLE	V(150)	обязательное поле
Ссылка на файл с ГОСТом	F_FILE	V(250)	обязательное поле

Таблица 16. Схема отношения *ЭТАПЫ ПРОЕКТА* (Stages)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*	
Проект	S_PRO	C(10)	внешний ключ (к Projects)	**составной первичный ключ**
Номер этапа	S_NUM	N(2)		
Название этапа	S_TITLE	V(200)	обязательное поле	
Дата начала этапа	S_BEGIN	D	обязательное поле	
Дата окончания этапа	S_END	D	обязательное поле, больше даты начала этапа	
Реальная дата окончания	S_FINISH	D	больше даты начала этапа	
Стоимость этапа	S_COST	N(10)	обязательное поле	
Полученная сумма по этапу	S_SUM	N(10)	обязательное поле, значение по умолчанию – 0	

Таблица 17. Схема отношения *ОТЧЁТЫ* (Reports)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Проект	R_PRO	C(10)	составной внешний ключ (к Stages)
Номер этапа	R_NUM	N(2)	
Форма отчётности	R_FORM	V(20)	внешний ключ к Forms
Ссылка на файл с отчётом	R_FILE	V(250)	
Дата создания отчёта	R_DATE	D	

Таблица 18. Схема отношения *РОЛИ* (Roles)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Название роли участника	R_NAME	V(20)	**первичный ключ**

Таблица 19. Схема отношения *УЧАСТИЕ* (Job)

Содержание поля	*Имя поля*	*Тип, длина*	*Примечания*
Проект	J_PRO	C(10)	составной внешний ключ (к Stages)
Номер этапа	J_NUM	N(2)	
Сотрудник	J_EMP	N(4)	внешний ключ (к Employees)
Роль	J_ROLE	V(20)	внешний ключ (к Roles)

Начало участия	J_BEGIN	D	обязательное поле
Окончание участия	J_END	D	
Доплата	J_BONUS	N(2)	обязательное поле, по умолчанию 0

Схема базы данных после нормализации приведена на Рис. 7.

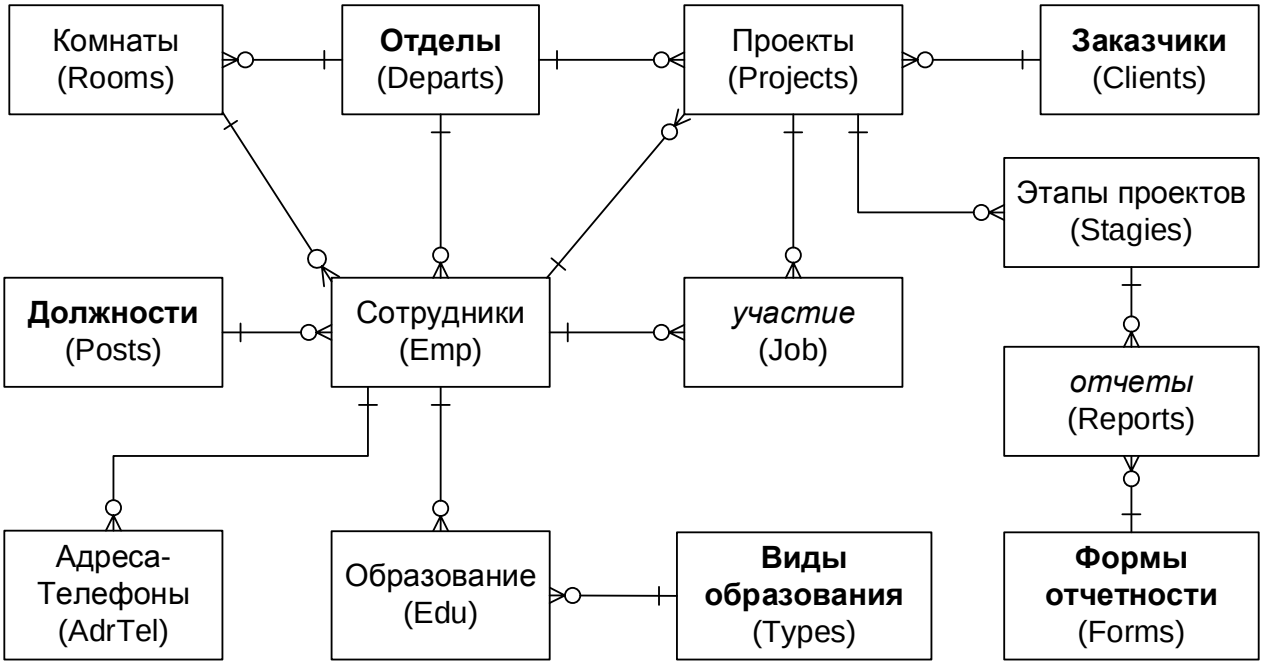


Рис. 7. Окончательная схема БД проектной организации

2.4.4. Определение дополнительных ограничений целостности

Перечислим ограничения целостности, которые не указаны в табл. 6–19.

1. В поле *Доплата* хранится величина доплаты сотруднику за участие в проекте (в процентах к его окладу). Значение поля больше либо равно 0.
2. Нумерация в поле *Номер этапа* начинается с 1 и является непрерывной для каждого проекта.
3. Дата начала первого этапа проекта должна соответствовать началу проекта в целом, дата завершения последнего этапа должна соответствовать завершению проекта в целом. Этапы не должны пересекаться по времени и между ними не должно быть разрывов.
4. Стоимость проекта должна быть равна сумме стоимостей всех этапов этого проекта. Вычислимые поля обычно не хранятся в БД. Но для проектов общая стоимость является базовым атрибутом, а стоимости этапов – зависимым, поэтому мы оставим это поле.
5. Периоды участия в проекте не должны выходить за границы периодов выполнения этапов проекта. Например, если сотрудник участвует в 1-м этапе проекта, то дата начала участия должна быть не меньше, чем дата начала 1-го этапа, а дата окончания – не больше, чем дата завершения этого этапа.

Ограничения 2-5 нельзя реализовать в схеме отношения. В реальных БД подобные ограничения целостности реализуются вручную или программно (через внешнее приложение или специальную процедуру контроля данных – триггер).

2.4.5. Описание групп пользователей и прав доступа

Опишем для каждой группы пользователей права доступа к каждой таблице (Таблица 21). Права доступа должны быть распределены так, чтобы для каждого объекта БД был хотя бы один пользователь, который имеет право добавлять и удалять данные из объекта. Используются следующие сокращения:

Таблица 20. Обозначения прав доступа

Обозначение	Расшифровка	Альтернативное обозначение
I (insert)	добавление данных	C (create)
S (select)	чтение данных	R (read)
U (update)	модификация данных	U (update)
D (delete)	удаление данных	D (delete)

Права руководителей проектов на изменение данных в таблицах *ЭТАПЫ ПРОЕКТОВ*, *УЧАСТИЕ* и *ОТЧЁТЫ* будут назначены через представления, т.к. руководитель проекта может изменять данные только о своих проектах. Описание представлений приведено в п.2.5.2. "Создание представлений (готовых запросов)". Права назначает администратор БД (или администратор безопасности, если система сложная и администраторов несколько).

Таблица 21. Права доступа к таблицам для групп пользователей

Таблицы	Группы пользователей (роли)					
	Руководитель организации	Зам. руково-дителя	Сотрудники отд. кадров	Руково-дители проектов	Бухгалтеры	Участники проектов
Отделы	S	SUID	SUI	S	S	S
Комнаты	S	S	SUID	S	S	S
Должности	SUID	SU	S		S	
Сотрудники	S	S	SUID	S	S	
Адреса-телефоны	S	S	SUID	S	S	
Виды образования	S	SIU	SUI	S		
Образование	S	SU	SUID	S		
Заказчики	SUID	SIU		S		
Проекты	SUID	SUI		SU	S	
Формы отчётности	SUID	SUI		SU		
Этапы проектов	SUID	SUI		S	S	
Отчёты	SUID	SU		S		
Роли	SUID	SUI		S		

Участие	S	S		S	S	

2.5. Реализация проекта базы данных

Приведём код создания БД (без привязки к конкретной реализации языка SQL).

2.5.1. Создание таблиц

1. Отношение Departs (отделы):

```
create table departs (
    d_id        varchar(12)  primary key,
    d_name      varchar(100)        not null);
```

2. Отношение Rooms (комнаты):

```
create table rooms (
    d_depart    varchar(12)  not null
    constraint fk_rooms_departs references departs(d_id),
    r_room      numeric(4)   not null,
    r_phone     varchar(20),
    constraint uniq_room_phone unique(r_room, r_phone));
```

3. Отношение Posts (должности):

```
create table posts (
    p_post      varchar(30)  primary key,
    p_salary    numeric(8,2)        not null,
    constraint check_salary check(p_salary>=12000));
```

4. Отношение Employees (сотрудники):

```
create table employees (
    e_id        numeric(4)   primary key,
    e_fname     varchar(25)  not null,
    e_lname     varchar(30)  not null,
    e_born      date         not null,
    e_sex char          not null
        constraint check_sex check(e_sex in ('ж','м')),
    e_pasp      char(10)            not null
        constraint uniq_pasp unique,
    e_date      date         not null,
    e_given     varchar(50)  not null,
    e_inn char(12)           not null
        constraint uniq_inn unique,
    e_snils     char(14)            not null
        constraint uniq_snils unique,
    e_depart    varchar(12)  not null
        constraint fk_emp_departs references departs,
    e_post      varchar(30)  not null
        constraint fk_emp_posts references posts,
    e_room      numeric(4)   not null,
    e_phone     varchar(20),
    e_login     varchar(30),
    constraint fk_emp_rooms foreign key(e_room,e_phone)
```

```
        references rooms(r_room,r_phone));
```

(Если внешний ключ ссылается на первичный ключ отношения, его можно не указывать, как в случае ссылок на Departs и Posts).

5. Отношение Types (виды образования):

```
create table types (
    t_name      varchar(20)  primary key);
```

6. Отношение Edu (образование):

```
create table edu (
    u_id   numeric(4)  not null
           constraint fk_edu_emp references employees,
    u_type      varchar(20)  not null
           constraint fk_edu_types references types,
    u_spec      varchar(40),
    u_diplom varchar(15),
    u_year      numeric(4)   not null);
```

7. Отношение AdrTel (адреса-телефоны):

```
create table adrtel (
    a_id   numeric(4)  not null
           constraint fk_adrtel_emp references employees,
    a_adr varchar(100),
    a_phone     varchar(20),
    constraint check_adrtel check(a_adr is not null
           or a_phone is not null));
```

8. Отношение Clients (заказчики):

```
create table clients (
    c_id   numeric(4)  primary key,
    c_company varchar(40)  not null,
    c_adr varchar(50)  not null,
    c_person varchar(50)      not null,
    c_phone     varchar(30));
```

9. Отношение Projects (проекты):

```
create table projects (
    p_id   numeric(6)  not null
           constraint uniq_p_id unique,
    p_title     varchar(100)        not null,
    p_abbr      char(10)            primary key,
    p_depart    varchar(12)  not null
           constraint fk_pro_departs references departs,
    p_company numeric(4)   not null
           constraint fk_pro_clients references clients,
    p_chief     numeric(4)  not null
           constraint fk_pro_emp references employees,
    p_begin     date           not null,
    p_end       date           not null,
    p_finish date,
    p_cost      numeric(10)         not null,
```

```
    constraint check_pro_cost check(p_cost>0),
    constraint check_pro_end_begin check (p_end>p_begin),
    constraint check_pro_finish check (p_finish is null
        or p_finish>p_begin));
```

10.Отношение Forms (формы отчетности):

```
create table forms (
    f_gost      varchar(20)  primary key,
    f_title     varchar(150)     not null,
    f_file varchar(250));
```

11.Отношение Stages (этапы проектов):

```
create table stages (
    s_pro char(10)          not null
        constraint fk_stages_projects references projects,
    s_num       numeric(2),
    s_title     varchar(200)     not null,
    s_begin     date        not null,
    s_end       date        not null,
    s_finish date,
    s_cost      numeric(10)      not null,
    s_sum       numeric(10)      not null,
    primary key(s_pro, s_num),
    constraint check_stages_cost check(s_cost>0),
    constraint check_stages_end_begin check(s_end>s_begin),
    constraint check_stages_finish check(s_finish is null
        or s_finish>s_begin));
```

12.Отношение Reports (отчеты):

```
create table reports (
    r_pro char(10)          not null,
    r_num       numeric(2)  not null,
    r_form      varchar(100)     not null
    constraint fk_reports_forms references forms,
    r_file varchar(250),
    r_datedate,
    constraint fk_reports_stages foreign key(r_pro, r_num)
        references stages);
```

13.Отношение Roles (роли участников):

```
create table roles (
    r_name      varchar(20)  primary key);
```

14.Отношение Job (участие):

```
create table job (
    j_pro char(10)          not null,
    j_num       numeric(2)    not null,
    j_emp       numeric(4)  not null
        constraint fk_job_emp references employees,
    j_role      varchar(20)  not null
```

```
        constraint fk_job_roles references roles,
    j_begin     date        not null,
    j_end date,
    j_bonus     number(2)   default 0 not null,
    constraint fk_job_stagse foreign key(j_pro, j_num)
        references stages,
    constraint check_job_end_begin check (j_end>j_begin),
    constraint check_job_bonus check(j_bonus>=0));
```

Начальный состав данных для справочных таблиц:

1) Таблица *Виды образования*:

```
insert into Types values('начальное');
insert into Types values('среднее');
insert into Types values('средне-специальное');
insert into Types values('высшее');
```

2) Таблица *Роли*:

```
insert into Roles values ('исполнитель');
insert into Roles values ('консультант');
```

3) Таблица *Формы Отчетности* [6]:

```
insert into Forms(f_gost, f_title) values('ГОСТ 19.005-85', 'ЕСПД. Р-схемы
    алгоритмов и программ. Обозначения условные графические и правила
    выполнения');
insert into Forms(f_gost, f_title) values('ГОСТ 19.201-78', 'ЕСПД. Техническое
    задание. Требования к содержанию и оформлению');
insert into Forms(f_gost, f_title) values('ГОСТ 19.202-78', 'ЕСПД.
    Спецификация. Требования к содержанию и оформлению');
insert into Forms(f_gost, f_title) values('ГОСТ 19.301-79', 'ЕСПД. Программа и
    методика испытаний. Требования к содержанию и оформлению');
insert into Forms(f_gost, f_title) values('ГОСТ 19.401-78', 'ЕСПД. Текст
    программы. Требования к содержанию и оформлению');
insert into Forms(f_gost, f_title) values('ГОСТ 19.402-78', 'ЕСПД. Описание
    программы');
insert into Forms(f_gost, f_title) values('ГОСТ 19.404-79', 'ЕСПД.
    Пояснительная записка. Требования к содержанию и оформлению');
insert into Forms(f_gost, f_title) values('ГОСТ 19.501-78', 'ЕСПД. Формуляр.
    Требования к содержанию и оформлению');
insert into Forms(f_gost, f_title) values('ГОСТ 19.502-78', 'ЕСПД. Описание
    применения. Требования к содержанию и оформлению');
insert into Forms values('ГОСТ 19.503-79', 'ЕСПД. Руководство системного
    программиста. Требования к содержанию и оформлению');
insert into Forms(f_gost, f_title) values('ГОСТ 19.504-79', 'ЕСПД. Руководство
    программиста. Требования к содержанию и оформлению');
insert into Forms(f_gost, f_title) values('ГОСТ 19.505-79', 'ЕСПД. Руководство
    оператора. Требования к содержанию и оформлению');
insert into Forms(f_gost, f_title) values('ГОСТ 19.508-79', 'ЕСПД. Руководство
    по техническому обслуживанию. Требования к содержанию и
    оформлению');
```

```
insert into Forms(f_gost, f_title) values('ГОСТ 19.701-90', 'ЕСПД. Схемы
    алгоритмов, программ, данных и систем. Обозначения условные и
    правила выполнения');
```

2.5.2. Создание представлений (готовых запросов)

Запрос – это объект БД, позволяющий производить основные операции по обработке данных: сортировку, фильтрацию, объединение данных из разных источников, преобразование данных – и сохранять результаты с некоторым именем, чтобы в дальнейшем использовать их по мере необходимости. Главное предназначение запросов — это отбор данных на основании заданных условий. Запрос объединяет в себе возможности, предоставляемые сортировкой и фильтрацией.

Особенность запросов состоит в том, что они черпают данные из базовых таблиц и создают на их основе временную таблицу. Применение запросов позволяет избежать дублирования данных в таблицах и обеспечивает максимальную гибкость при поиске и отображении данных в базе данных.

Все запросы делятся на две группы: **запросы-выборки, запросы-действия.**

Запросы-выборки осуществляют выборку данных из таблиц в соответствии с заданными условиями. К этой группе запросов относятся следующие.

1. Запрос к связанным таблицам — позволяет производить выборку данных из связанных таблиц.
2. Перекрестный запрос — отображает итоговые данные с группировкой их по горизонтали и вертикали, выводя результаты их обработки в виде таблиц.
3. Запрос с параметром — позволяет пользователю задать критерий отбора, введя нужный параметр при вызове запроса.
4. Запрос с вычисляемым полем — позволяет рассчитать данные на основе других полей из той же строки запроса.
5. Запрос с критерием поиска — позволяет производить отбор записей в соответствии с заданным критерием поиска.
6. Запрос с итогами — производит математические вычисления и выдает результат.

Запросы-действия позволяют модифицировать данные в таблицах: удалять, обновлять, добавлять записи. К этой группе запросов относятся следующие.

1. Запросы на создание таблицы создают таблицы на основании данных, содержащихся в результирующем множестве запроса.
2. Запросы на добавление записей позволяют добавлять в таблицу записи, создаваемые запросом.
3. Запросы на обновление изменяют значения существующих полей в соответствии с заданным критерием.

4. Запросы на удаление удаляют записи из одной или нескольких таблиц одновременно.

Приведём примеры нескольких **готовых запросов (представлений)**:

1. **Список всех текущих проектов** (current_date – функция, возвращающая текущую дату, определена в СУБД PostgreSQL; в других системах аналогичная функция может называться по-другому, например, sysdate в Oracle, getdate() в Transact-SQL, now() в MS Access, currdate() в MySQL):

```
create view curr_projects as
    select    *
      from    projects
     where    current_date between p_begin and p_end;
```

2. **Список всех текущих этапов проектов**:

```
create view curr_stages as
    select    *
      from    stages
     where    current_date between s_begin and
       COALESCE(s_end, current_date);
```

Функция COALESCE() возвращает первый аргумент, отличный от null. В данном случае функция вернет текущую дату, если дата завершения участия сотрудника в проекте не определена.

3. **Определение суммы по текущим проектам, полученной на текущую дату**:

```
create or replace view summ (title, cost, total) as
    select    p_title, p_cost, sum(s_sum)
      from    curr_projects, stages
     where    p_abbr=s_pro
     group by p_title, p_cost;
```

4. **Список сотрудников, участвующих в текущих проектах**:

```
create view participants (project, name, role) as
    select    p_abbr, e_fname||' '||e_lname, 'руководитель'
      from    curr_projects, employees
     where    p_chief=e_id
    union all
    select    j_pro, e_fname||' '||e_lname, j_role
      from    employees, job
      where   e_id=j_emp and current_date between j_begin
      and COALESCE(j_end, current_date)
     order by 1, 3 desc;
```

5. **Список рабочих телефонов сотрудников**:

```
create or replace view worktel (name, room, phone) as
    select    e_fname||' '||e_lname, e_room, e_phone
      from    employees
     order by 1;
```

6. **Форма отчётности и сроки выполнения текущих этапов по проектам**:

```
create or replace view curr_reports as
    select    s_pro, s_num, s_title, s_begin, s_end, r_form,
```

```
        r_file, r_date
    from   stages, reports
    where s_pro=r_pro and s_num=r_num and current_date
        between s_begin and COALESCE(s_finish, s_end)
  order by 1, 2;
```

7. **Данные о проектах (будут доступны только руководителям проектов):**

```
create or replace view my_projects as
    select    *
      from   projects p
      where exists (select * from employees e
          where e.e_id=p.p_chief and e.e_login=user);
```

Функция user возвращает имя пользователя, выполняющего текущий запрос. Таким образом, **права доступа к этому предсталвению можно дать всем сотрудникам,** но каждый пользователь получит данные только о тех проектах, руководителем которых является. Используя аналогичный способ, можно ограничить участника проекта данными только о сотрудниках тех проектов, в которых он сам участвует.

8. **Данные о текущих этапах проектов (доступны руководителям проектов)**:

```
create or replace view my_stages as
    select    s.*
      from   curr_stages s
      where exists (select *
          from employees e, projects p
          where e.e_id=p.p_chief and e.e_login=user
          and s.s_pro=p.p_abbr);
```

9. **Данные об участниках текущих проектов**:

```
create or replace view my_staff as
    select    j.*
      from   job j
      where exists (select *
          from employees e, curr_projects p
          where e.e_id=p.p_chief and e.e_login=user
          and j.j_pro=p.p_abbr);
```

Обратите внимание: таблицы *УЧАСТИЕ* и *ПРОЕКТЫ* не связаны напрямую внешним ключом; но наличие в таблице *УЧАСТИЕ* поля "Аббревиатура проекта" (j_pro) позволяет соединять эти таблицы непосредственно, без обращения к таблице *ЭТАПЫ ПРОЕКТА*.

10. **Данные о других участниках проекта (доступны участникам проектов):**

```
create or replace view my_emps as
    select    je.j_pro, e.e_fname||' '||e.e_lname e_name,
     e_depart, e_post, e_phone, e_room
      from   employees e, job je
      where e.e_id=je.j_emp and exists (select *
          from job jm, employees m
```

```
where m.e_id=jm.j_emp and
m.e_login=user and je.j_pro=jm.j_pro);
```

Для работы с этими представлениями соответствующим пользователям нужно определить права доступа к представлениям (Таблица 22).

Таблица 22. Права доступа к представлениям

Представления	Группы пользователей (роли)		
	Руководители организации	Руководители проектов	Все сотрудники (PUBLIC)
Текущие проекты (curr_projects)	S	S	
Текущие этапы (curr_stages)	S	S	
Сумма по текущим проектам (summ)	S	S	
Рабочие телефоны (worktel)			S
Участники проектов (participants)			S
Отчетность (curr_reports)			S
Проекты для руководителя (my_projects)			SIUD
Стадии проектов (my_stages)			SIUD
Участники проектов (my_staff)			SIUD
Участники проектов (my_emps)			S

2.5.3. Назначение прав доступа

Права доступа пользователей предоставляются с помощью команды GRANT. Рассмотрим для примера права сотрудника отдела кадров (роль ***ok_user***). Права доступа к отношениям Departs и Rooms могут быть описаны следующим образом:

```
grant select, insert, update, delete on departs to ok_user;
grant select, insert, update, delete on rooms to ok_user;
```

Права доступа руководителей проектов (роль staff) к представлению summ могут быть описаны следующим образом:

```
grant select on summ to staff;
```

Права доступа всех сотрудников к представлению my_projects могут быть описаны следующим образом:

```
grant select,insert,update,delete on my_projects to PUBLIC;
```

Если сотрудник не является руководителем проекта, он не получит данных через это представление и не сможет воспользоваться правами доступа к нему.

Права доступа участников проекта к представлению my_emps могут быть описаны следующим образом:

```
grant select on my_emps to PUBLIC;
```

Если сотрудник не является участником проекта, он не получит данных через это представление и не сможет воспользоваться правами доступа к нему.

2.5.4. Создание триггеров

Периоды участия в проекте не должны выходить за границы периодов выполнения этапов проекта. Например, если сотрудник участвует в 1-м этапе проекта, то дата начала участия должна быть не меньше, чем дата начала 1-го этапа, а дата окончания – не больше, чем дата завершения этого этапа. Реализуем эту проверку с помощью триггера уровня строки:

```
create or replace function check_period() returns trigger as $$
declare
        cnt integer;
begin
        select count(*) into cnt
                from  stages s
                where s_pro = new.j_pro and
                        s_num = new.j_num and
                        new.j_begin between s_begin and s_end and
                        (new.j_end is null or
                        new.j_end between s_begin and s_end);
        if cnt=0 then
                raise exception 'Период участия сотрудника с номером % не
                совпадает с периодом выполнения этапа', new.j_emp;
        else return NEW;
        end if;
end;
$$ LANGUAGE plpgsql;
```

После создания триггерной функции можно создать сам триггер:

```
create trigger tr_check_period
        before insert or update on Job
        for each row
        execute procedure check_period();
```

Примечание. Не забудьте проверить работоспособность создаваемых вами триггеров с помощью ввода команд, которые являются событиями триггера.

2.5.5. Создание индексов

Анализ готовых запросов показывает, что для повышения эффективности работы с данными необходимо создать индексы для всех внешних ключей (в тех системах, которые не создают такие индексы автоматически). Приведём примеры создания индексов:

```
create index ind_e_posts on employees(e_post);
create index ind_p_chief on projects(p_chief);
create index ind_e_room2 on employees(e_room, e_phone);
```

Также есть запросы, в которых учитывается роль участника проекта; добавим это поле к составному индексу на поля *Сотрудник* и *Роль*. Возможно, полезным окажется составной индекс по ФИО сотрудника. А для ускорения

поиска текущих участников проекта нужен индекс по полям "начало" и "окончание участия" в таблице Участие. Создадим эти индексы:

```
create index ind_j_role on job(j_emp, j_role);
create index ind_e_name on employees(e_lname, e_fname);
create index ind_e_period on job(j_begin, j_end);
```

2.5.6. Разработка стратегии резервного копирования

Интенсивность обновления разработанной базы данных низкая, поэтому для обеспечения сохранности вполне достаточно проводить полное резервное копирование БД раз в день (перед окончанием рабочего дня). Для разработанной БД нет необходимости держать сервер включенным круглосуточно, поэтому можно создать соответствующее задание операционной системы, которое будет автоматически запускаться перед выключением сервера.

Заключение

Процесс проектирования БД сопряжен со многими сложностями.
Во-первых, необходимо разработать начальный набор элементов данных и записей, которые могут быть нормализованы. Затем необходимо оценить факторы, влияющие на нормализацию:

1. Однозначные и многозначные факты.
2. Зависимость от всего ключа.
3. Независимые и зависимые факты.
4. Наличие взаимных ограничений.
5. Наличие неуникальных или неособых представлений.

И, наконец, необходимо оценить желательность нормализации с точки зрения ее влияния на производительность приложений поиска.

Учебно-методическое обеспечение дисциплины

Перечень основной литературы:

1. Кузнецов С. Д. Введение в реляционные базы данных: учебное пособие / С. Д. Кузнецов. - 4-е изд. - Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2025. - 247 с. - ISBN 978-5-4497-0902-8. - Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. - URL: https://www.iprbookshop.ru/146337.html.
2. Швецов В.И. Базы данных Электронный ресурс: учебное пособие / В.И. Швецов. - Базы данных. - Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. - 218 с. - Книга находится в базовой версии ЭБС IPRbooks.
3. Мамедли Р. Э. Базы данных: лабораторный практикум / Р. Э. Мамедли. - Нижневартовск: Нижневартовский государственный университет, 2021. - 160 с. - ISBN 978-5-00047-586-7. - Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. - URL: https://www.iprbookshop.ru/118977.html

Перечень дополнительной литературы:

1. Базы данных и экспертные системы: учебное пособие / сост.: А.Н. Макоха, И.А. Журавлёва. – Ставрополь: Изд-во СКФУ, 2020. – 248 с.
2. Грошев А.С. Основы работы с базами данных Электронный ресурс: учебное пособие / А.С. Грошев. - Основы работы с базами данных. - Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. - 255 с. - Книга находится в базовой версии ЭБС IPRbooks.
3. Моргунов, Е. П. PostgreSQL. Основы языка SQL учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. - СПб.: БХВ-Петербург, 2018. - 336 с.: ил.
4. Новиков Б. А. Основы технологий баз данных: учеб. пособие / Б. А. Новиков, Е. А. Горшкова, Н. Г. Графеева; под ред. Е. В. Рогова. - 2-е изд. - М.: ДМК Пресс, 2020. - 582 с. экземпляров неограничено
5. Тарасов С. В. СУБД для программиста. Базы данных изнутри / С. В. Тарасов. - Москва: СОЛОН-Пресс, 2024. - 320 с. - ISBN 978-2-7466-7383-0. - Текст: электронный // Цифровой образовательный ресурс IPR SMART: [сайт]. - URL: https://www.iprbookshop.ru/142024.html.

Перечень ресурсов информационно-телекоммуникационной сети «Интернет», необходимых для освоения дисциплины

1. https://www.postgresql.org/PostgreSQL: реляционная база данных с открытым исходным кодом
2. http://citforum.ru/database/sqlbook/index.shtml С.Д. Кузнецов Введение в стандарты языка баз данных SQL.
3. https://www.youtube.com/playlist?list=PLaFqU3KCWw6JhHBp07QSu9uE8zahhKnTn Администрирование PostgreSQL10. Базовый курс.