МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное

образовательное учреждение высшего образования

«Тульский государственный университет»

Институт прикладной математики и компьютерных наук

Кафедра «Вычислительная техника»

|  |
| --- |
| Утверждено на заседании кафедры  «Вычислительная техника»  «29» января 2019г., протокол № 6 |
| Заведующий кафедрой  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.Н. Ивутин |

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ**

**по выполнению лабораторных работ**

**по дисциплине (модулю)**

**«Базы данных»**

**основной профессиональной образовательной программы**

**высшего** **образования – программы бакалавриата**

по направлению подготовки

09.03.01 «Информатика и вычислительная техника»

с профилем

**«Электронно-вычислительные машины, комплексы, системы и сети»**

Формы обучения: очная, заочная

Идентификационный номер образовательной программы: 090301-02-18

Тула 2019 год

**Разработчик(и) методических указаний**

\_\_\_\_Набродова И.Н., доцент, к.т.н.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(ФИО, должность, ученая степень, ученое звание) (подпись)

**Оглавление**

[Лабораторная работа №1 3](#_Toc18416072)

[Лабораторная работа №2 6](#_Toc18416073)

[Лабораторная работа №3 14](#_Toc18416074)

[Лабораторная работа №4 18](#_Toc18416075)

[Лабораторная работа №5 22](#_Toc18416076)

[Лабораторная работа №6 27](#_Toc18416077)

[Лабораторная работа №7 31](#_Toc18416078)

[Лабораторная работа №8 37](#_Toc18416079)

[Лабораторная работа №9 45](#_Toc18416080)

[Лабораторная работа №10 56](#_Toc18416081)

[Лабораторная работа №11 62](#_Toc18416082)

[Лабораторная работа №12 73](#_Toc18416083)

[Лабораторная работа №13 84](#_Toc18416084)

[Лабораторная работа №14 94](#_Toc18416085)

[Лабораторная работа №15 99](#_Toc18416086)

[Лабораторная работа №16 104](#_Toc18416087)

[Лабораторная работа №17 110](#_Toc18416088)

[Лабораторная работа №18 114](#_Toc18416089)

[Лабораторная работа №19 118](#_Toc18416090)

**Лабораторная работа №1**

**Установка и настройка СУБД**

**1. Цель и задачи работы**

Целью данной лабораторной работы является получение навыком и настройки СУБД.

**2. Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

**3. Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

***Запуск базы данных***

PostgreSQL запускается автоматически сразу после инсталляции и после каждой системной перезагрузки. Таким образом, нет необходимости запускать базу данных, если предварительно Вы её не остановили.

Вы можете запустить базу данных непосредственно с рабочего стола или при помощи командной строки SQL.

Для запуска базы данных с рабочего стола:

Подключитесь к серверу PostgreSQL как администратор Windows - т.е., как пользователь, являющийся членом группы «Администраторы».

Перейдите на PostgreSQL 10, после чего нажмите Start, выберите нужную базу данных и дважды щелкните по ней.

Для запуска базы данных с использованием командной строки SQL, Вы должны запускать приложение командной строки с того же компьютера, на котором была установлена база данных PostgreSQL.

Чтобы запустить базу данных в командной строке SQL:

1.Подключитесь к серверу PostgreSQL под пользователем имеющим на это право.

2.Откройте терминальную сессию или командное окно, если оно еще не открыто.

3. На приглашение операционной системы введите следующую команду для запуска командной строки SQL и соединения с базой данных: psql -U postgres. В случае проблем с командой на macOS необходимо выполнить следующие действия в окне терминала:

* + - * + nano ~/.bash\_profile
        + Добавить в файл строку «PATH=/Application/Postgres.app/Contents/Version/10/bin:$PATH »
        + ~/.bash\_profile

4. Для выхода из базы данных введите команду -q.

***Остановка базы данных***

PostgreSQL останавливается автоматически одновременно с завершением работы компьютера, на котором она установлена.

Перед остановкой PostgreSQL рекомендуется убедиться, что все пользователи и приложения завершили свою работу и закрыли соединение с базой данных.

Если пользователи или приложения имеют открытые соединения на момент начала остановки базы данных, остановка произойдёт следующим образом:

Будет запрещено выполнение новых соединений и запрещён запуск новых транзакций;

Для всех неподтвержденных транзакций будет выполнен откат;

Позволяет всем существующим сеансам закончить работу в штатном режиме

Вы можете остановить базу данных как с рабочего стола, так и с помощью командной строки SQL.

Чтобы остановить базу данных с рабочего стола:

Подключитесь к серверу PostgreSQL как администратор Windows - т.е., как пользователь, являющийся членом группы «Администраторы».

Перейдите на PostgreSQL 10, после чего нажмите Stop.

Для остановки базы данных с использованием командной строки SQL, Вы должны запускать приложение командной строки с того же компьютера, на котором была установлена база данных PostgreSQL.

Чтобы завершить работу базы данных с помощью командной строки SQL:

Подключитесь к серверу PostgreSQL как администратор Windows - т.е., как пользователь, являющиеся членом группы «Администраторы».

Откройте терминальную сессию или командное окно, если оно еще не открыто.

На приглашение операционной системы введите следующую команду для запуска командной строки SQL и соединения с базой данных: sigterm

**Учетные записи пользователей**

Учётная запись пользователя состоит из имени пользователя и атрибутов пользователя, включающих:

1. Пароль для проверки прав доступа к базе данных.

2. Привилегии и роли.

3. Табличное пространство по умолчанию для объектов базы данных.

4. Временное табличное пространство по умолчанию для обработки запросов.

**5. Оборудование**

Персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

**6. Задание на работу**

1. Установите СУБД PostgreSQL 9.6 (или новее) и клиент управления СУБД pgAdmin4. Приведите скриншоты шаговой установки.

2. Подключитесь к базе данных через текстовый терминал (с помощью программы psql) от имени пользователя PostgreSQL и создайте базу данных. Приведите в отчете скриншоты терминала с выполненными командами и их результатом.

**7. Контрольные вопросы**

1. Каковы основные функции СУБД?
2. Назовите отличительные черты реляционных баз данных?
3. Что представляют собой базы данных?
4. Назовите преимущества архитектуры «клиент-сервер»?
5. Какие функции перешли к среднему уровню обработки данных в трехуровневой архитектуре «клиент-сервер»?
6. Определите роль сервера в двухуровневой архитектуре «клиент-сервер».

**Лабораторная работа №2**

**Создание таблиц**

1. **Цели и задачи работы**

Целью лабораторной работы является изучение и практическое применение функций для создания таблиц.

1. **Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

1. **Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

1. **Теоретические сведения**

Создание объектов базы данных осуществляется с помощью операторов языка запросов SQL.

Для создания таблицы базы данных (БД) используется команда CREATE TABLE. CREATE TABLE создаёт новую, изначально пустую таблицу в текущей базе данных. CREATE TABLE также автоматически создаёт составной тип данных, соответствующий одной строке таблицы. Таким образом, имя таблицы не может совпадать с именем существующего типа в этой же схеме.

Синтаксис команды CREATE TABLE:

CREATE TABLE name\_table (name\_columndata\_type,…);

Для удаления таблицы используется команда DROP TABLE. DROP TABLE удаляет таблицы из базы данных. Удалить таблицу может только её владелец, владелец схемы или суперпользователь. Чтобы опустошить таблицу, не удаляя её саму, вместо этой команды следует использовать [DELETE](https://postgrespro.ru/docs/postgresql/9.6/sql-delete) или [TRUNCATE](https://postgrespro.ru/docs/postgresql/9.6/sql-truncate).

Синтаксис команды DROP TABLE:

DROP TABLE name\_table;

Для модификации базы данных используется команда ALETR TABLE. ALTER TABLE меняет определение существующей таблицы. Например, можно добавлять или удалять столбцы, менять их размер, создавать или уничтожать индексы или переименовывать столбцы либо саму таблицу. Можно также изменять комментарий для таблицы и её тип.

Возможные синтаксисы команды ALTER TABLE:

* Запрос добавляющий столбец в конец таблицы ALTER TABLE table1 (column1 varchar(10));
* Изменение размера столбца ALTER TABLE table1 MODIFY (column1 varchar(20));
* Удаление столбца таблицы ALTER TABLE table1 DROP COLUMN column1;
* Изменение имени таблицы ALTER TABLE table1 RENAME TO table2;

При создании (изменении) таблицы могут быть определены ограничения на вводимые значения. В этом случае SQL будет отвергать любое из них при не соответствии заданным критериям. Ограничения могут быть статическими, ограничивающими значения или диапазон значений, вставляемых в столбец (CHECH, NOT NULL). Они могут иметь связь со всеми значениями столбца, ограничивая новые строки значениями, которые не содержатся в столбцах или их наборах (уникальные значения, первичные ключи). Ограничения могут также определяться связью со значениями, находящимися в другой таблице, допуская, например, вставку в столбец только тех значений, которые в данный момент содержатся так- же в другом столбце другой или этой же таблицы (внешний ключ). Эти ограничения носят динамический характер.

Существует два основных типа ограничений - ограничения на столбцы и ограничения на таблицу. Ограничения на столбцы (COLUMN CONSTRAINTS) применимы только к отдельным столбцам, а ограничения на таблицу (TABLE CONSTRAINTS) применимы к группам, состоящим из одного или более столбцов. Ограничения на столбец добавляются в конце определения столбца после указания типа данных и перед окончанием описания столбца (запятой). Ограничения на таблицу размещаются в конце определения таблицы, после определения последнего столбца.

Чтобы запретить возможность использования в поле NULL-значений, можно при создании таблицы командой CREATE TABLE указать для соответствующего столбца ключевое слово NOT NULL. Это ограничение применимо только к столбцам таблицы. NULL – это специальный маркер, обозначающий тот факт, что поле пусто.

Если при создании таблицы для столбца указывается ограничение UNIQUE, то база данных отвергает любую попытку ввести в это поле какой- либо строки значение, уже содержащееся в том же поле другой строки. Это ограничение применимо только к тем полям, которые были объявлены NOT NULL.

Можно сделать уникальными группу полей, указав UNIQUE в качестве ограничений таблицы. При объединении полей в группу важен порядок, в котором они указываются. Ограничение на таблицу UNIQUE является полезным, если требуется поддерживать уникальность группы полей. Ограничения уникальности гарантируют, что данные в определённом столбце или группе столбцов уникальны среди всех строк таблицы. Например, если в нашей базе данных не допускается, чтобы студент сдавал в один день больше одного экзамена, то можно в таблице объявить уникальной комбинацию значений полей STUDENT\_ID и EXAM\_DATE. Для этого следует создать таблицу EXAM\_MARKS таким способом.

CREATE TABLE EXAM\_MARKS

(EXAMID INTEGER NOT NULL,

STUDENTID INTEGER NOT NULL,

SUBJID INTEGER NOT NULL,

MARK CHAR(1),

EXAM\_DATEDATE NOTNULL,

UNIQUE (STUDENT\_ID, EXAM\_DATE)); Ограничениям таблиц можно присваивать уникальные имена. Преимущество явного задания имени ограничения состоит в том, что в этом случае при выдаче системой сообщения о нарушении установленного ограничения будет указано его имя, что упрощает обнаружение ошибок.

Для присвоения имени ограничению используется несколько изменённый синтаксис команд CREATE TABLE и ALTER TABLE.

Приведённый выше пример запроса изменяется следующим образом:

CREATE TABLE EXAM\_MARKS

(EXAMID INTEGER NOT NULL,

STUDENTID INTEGER NOT NULL,

SUBJID INTEGER NOT NULL,

MARK CHAR(1),

EXAM\_DATEDATE NOTNULL,

CONSTRAINT STUD\_SUBJ\_CONSTR

UNIQUE (STUDENT\_ID, EXAM\_DATE);

В этом запросе STUD\_SUBJ\_CONSTR - это имя, присвоенное указанному

ограничению таблицы. Первичные ключи таблицы - это специальные случаи комбинирования ограничений UNIQUE и NOT NULL. Первичные ключи имеют следующие особенности:

* таблица может содержать только один первичный̆ ключ;
* внешние ключи по умолчанию ссыпаются на первичный̆ ключ таблицы;
* первичный̆ ключ является идентификатором строк таблицы (строки, однако, могут идентифицироваться и другими способами).

Улучшенный вариант создания таблицы первичным ключом имеет теперь следующий вид:

CREATE TABLE STUDENT (STUDENTID INTEGER PRIMARY KEY,

SURNAME CHAR (25) NOT NULL,

NAME CHAR (10) NOT NULL,

STIPEND INTEGER,

KURS INTEGER,

CITY CHAR (15),

BIRTHDAY DATE,

UNIV\_ID INTEGER);

Ограничение PRIMARY KEY может также быть применено для нескольких полей, составляющих уникальную комбинацию значений – составнойпервичный ключ. Мы можем применить ограничение таблицы PRIMARY KEY, объявив пару EXAM\_ID и STUDENT\_ID первичным ключом таблицы:

CREATE TABLE NEW\_EXAM\_MARKS

(STUDENTID INTEGER NOT NULL,

SUBJID INTEGER NOT NULL,

MARK INTEGER,

DATA DATE,

CONSTRAINT EX\_PR\_KEY PRIMARY KEY(EXAM\_ID,STUDENT\_ID));

Ограничение CHECK позволяет определять условие, которому должно удовлетворять вводимое в поле таблицы значение, прежде чем оно будет принято. Любая попытка обновить или заменить значение поля такими, для которых предикат, задаваемый ограничением CHECK, имеет значение ЛОЖЬ, будет отвергаться.

Рассмотрим таблицу STUDENT. Значение столбца STIPEND в этой таблице STUDENT выражается десятичным числом. Наложим на значения этого столбца следующее ограничение – величина размера стипендии должна быть меньше 200.

Соответствующий запрос имеет следующий вид.

CREATE TABLE STUDENT

(STUDENTID INTEGER PRIMARY KEY,

SURNAME CHAR (25) NOT NULL,

NAME CHAR (10) NOT NULL,

STIPEND INTEGER CHECK (ST IPEND < 200),

KURS INTEGER,

CITY CHAR (15),

BIRTHDAY DATE,

UNIV\_ID INTEGER);

В SQL имеется возможность при вставке в таблицу строки, не указывая значений некоторого поля, определять значение этого поля по умолчанию. Наиболее часто используемым значением по умолчанию является NULL. Это значение принимается по умолчанию для любого столбца, для которого не было установлено ограничение NOT NULL. Ограничение NOT NULL просто указывает, что столбцу нельзя присваивать значение NULL. Пример синтаксиса:

Значение поля по умолчанию указывается в команде CREATE TABLE тем же способом, что и ограничение столбца, с помощью ключевого слова

DEFAULT <значение по умолчанию>.

Строго говоря, опция DEFAULT не имеет ограничительного свойства, так как она не ограничивает значения, вводимые в поле, а просто конкретизирует значение поля в случае, если оно не было задано.

Когда каждое значение, присутствующее в одном поле таблицы, представлено в другом поле другой или этой же таблицы, говорят, что первое поле ссылается на второе. Это указывает на прямую связь между значениями двух полей. Поле, которое ссылается на другое поле, называется внешним ключом, а поле, на которое ссылается другое поле, называется родительским ключом. Так что поле UNIV\_ID таблицы STUDENT – это внешний ключ (оно ссылается на поле другой таблицы), а поле UNIV\_ID таблицы UNIVERSITY, на которое ссылается этот внешний ключ – это родительский ключ.

На практике внешний ключ не обязательно может состоять только из одного поля. Подобно первичному ключу, внешний ключ может состоять из любого числа полей. Внешний ключ и родительский ключ, на который он ссылается, конечно же, должны быть определены на одинаковом множестве полей (по количеству полей, типам полей и порядку следования полей).

Для решения вопросов поддержания ссылочной целостности в SQL используется ограничение FOREIGN KEY. Назначение FOREIGN KEY – это ограничение допустимых значений поля множеством значений родительского ключа, ссылка на который указывается при описании данного ограничения FOREIGN KEY. Ограничение внешнего ключа указывает, что значения столбца (или группы столбцов) должны соответствовать значениям в некоторой строке другой таблицы. Это называется *ссылочной целостностью* двух связанных таблиц.

Проблемы обеспечения ссылочной целостности возникают как при вводе значений поля, являющегося внешним ключом, так и при модификации/удалении значений поля, на которое ссылается этот ключ (родительского ключа). Одно из действий ограничения FOREIGN KEY – это отклонение (блокировка) ввода значений внешнего ключа, отсутствующих в таблице с родительским ключом. Также это ограничение воздействует на возможность изменять или удалять значения родительского ключа.

Ограничение FOREIGN KEY используется в командах CREATE TABLE и ALTER TABLE при создании или модификации таблицы, которая содержит поле, которое требуется объявить внешним ключом. В команде указывается имя родительского ключа, на который имеется ссылка в ограничении FOREIGN KEY.

Синтаксис ограничения FOREIGN KEY имеет следующий вид:

FOREIGNKEY < список столбцов >

REFERENCES < родительская таблица > [<родительский ключ >];

В этом предложении список столбцов – это список из одного или более столбцов таблицы, которые будут созданы или изменены командами CREATE TABLE или ALTER TABLE (должны быть отделены друг от друга запятыми). Параметр родительская таблица - это имя таблицы, содержащейродительский ключ. Это, в частности, может быть и таблицей, которая создается или изменяется текущейкомандой. Параметр родительский ключ представляет собой список столбцов родительской таблицы, которые составляют собственно родительский ключ. Оба списка столбцов, определяющих внешний и родительский ключи, должны быть совместимы, а именно:

* содержать одинаковое число столбцов.
* последовательность (1-й, 2-й, 3-й и т.д.) столбцов списка внешнего ключа должны иметь типы данных и размеры, совпадающие с соответствующими (1-м, 2-м, 3-м и т.д.) столбцами списка родительского ключа.

При применении команды ALTER TABLE к какой-либо таблице для задания ограничения FOREIGN KEY, значения внешнего ключа этой таблицы и родительского ключа соответствующей таблицы должны находиться в состоянии ссылочной целостности. В противном случае команда будет отклонена.

Синтаксис команды ALTER TABLE в этом случае имеет следующий вид:

ALTER TABLE <имя таблицы>

ADD CONSTRAINT <имя ограничения>

FOREIGN KEY (< список столбцов внешнего ключа > )

REFERENCES < имя родительской таблицы >

[(< список столбцов родительского ключа >)];

Ограничение внешнего ключа может указываться не для всей таблицы, как это было показано выше, а непосредственно на соответствующий столбец таблицы. При таком варианте, называемом ссылочным ограничением столбца, ключевое слово FOREIGN KEY фактически не используется. Просто используется ключевое слово REFERENCES и далее указывается имя родительского ключа, подобно следующему примеру:

CREATE TABLE STUDENT

( STUDENTID INTEGER PRIMARY KEY,

SURNAME CHAR (25),

NAME CHAR (10),

STIPEND INTEGER,

KURS INTEGER,

CITY CHAR (15),

BIRTHDAY DATE,

UNIV\_ID INTEGER REFERENCES UNIVERSITY(UNIV\_ID)); Команда определяет поле STUDENT.UNIV\_ID как внешний ключ, использующий в качестве родительского ключа поле UNIVERSITY.UNIV\_ID, являющееся ключом таблицы UNIVERSITY.

Эта форма эквивалентна следующему ограничению таблицы STUDENT:

FOREIGN KEY (UNIV\_ID) REGERENCES UNIVERSITY (UNIV\_ID) или, вдругой записи,

CONSTRAINT UNIV\_FOR\_KEY FOREIGN KEY (UNIV\_ID) REFERENCES UNIVERSITY (UNIV\_ID).

Как уже говорилось, при использовании команд INSERT и UPDATE для модификации значений столбца, объявленного как внешний ключ, вновь вводимые значения должны уже быть обязательно представлены в фактически присутствующих значениях столбца, объявленного родительским ключом. При этом можно помещать в эти поля пустые (NULL) значения, несмотря на то, что значения NULL не допустимы в родительских ключах. Можно также удалять (DELETE) любые строки с внешними ключами из таблицы, в которой̆ эти ключи объявлены.

При необходимости модификации значений родительского ключа дело обстоит иначе. Использование команды INSERT, которая осуществляет ввод новой записи, не вызывает никаких особенностей, при которых возможно нарушение ссылочной целостности. Однако команда UPDATE, изменяющая значение родительского ключа и команда DELETE, удаляющая строку, содержащую такой̆ ключ, содержат возможность нарушения согласованности значений родительского и ссылающихся на него внешних ключей. Например, может возникнуть так называемая "висячая" ссылка внешнего ключа на несуществующее значение родительского ключа, что совершенно не допустимо. Чтобы при применении команд UPDATE и DELETE к полю, являющемуся родительским ключом, не нарушалась целостность ссылки, возможны следующие варианты действий̆.

* Любые изменения значений родительского ключа запрещаются и при по- пытке их совершения отвергаются (ограничение NO ACTION или RESTRICT). Эта спецификация действия применяется по умолчанию.
* Изменения значений родительского ключа разрешаются, но при этом автоматически осуществляется коррекция всех значений внешних ключей, ссылающихся на модифицируемое значение родительского ключа. Это называется каскадным изменением (ограничение CASCADE).
* Изменения значений родительского ключа разрешаются, но при этом соответствующие значения внешнего ключа автоматически удаляются, то есть за- меняются значением NULL (ограничение SET NULL).
* Изменения значений родительского ключа разрешаются, но при этом соответствующие значения внешнего ключа автоматически заменяются значением по умолчанию (ограничение SET DEFAULT).

При описании внешнего ключа должно указываться, какой из приведенных вариантов действий следует применять, причем в общем случае это должно быть указано раздельно для каждой из команд UPDATE и DELETE. В качестве примера использования ограничений, накладываемых на операции модификации родительских ключей, можно привести следующий запрос:

CREATE TABLE NEW\_EXAM\_MARKS

(STUDENTID INTEGER SUB J\_I D INTEGER NOT NULL,

MARK INTEGER,

DATA DATE,

CONSTRAINT EXAM\_PR\_KEY PRIMARY

REFERENCES SUBJECT,

CONSTRAINT STUDENT\_ID\_FOR\_KEY FOREIGN KEY (STUDENT\_ID) REFERENCES STUDENT ON UPDATE CAS CADE ON DELETE NO ACTION);

В этом примере при попытке изменения значения поля STUDENT\_ID таблицы STUDENT будет автоматически обеспечиваться каскадная корректировка этих значений в таблице EXAM\_MARKS. При изменении идентификатора студента STUDENT\_ID в таблице STUDENT сохранятся все ссылки на его оценки. Однако любая попытка удаления (DELETE) записи о студенте из таблицы STUDENT будет отвергаться, если в таблице EXAM\_MARKS существуют записи об оценках данного студента.

Более подробная информация: <https://www.postgresql.org/docs/9.5/tutorial-sql.html>

1. **Оборудование**

персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

1. **Задание на работу**

1. Напишите три SQL запроса, создающие таблицы в соответствии с вашим вариантом. В работе вы должны продемонстрировать использование различных типов данных: целочисленные, числа с плавающей точкой, числа с фиксированной точкой, строковые типы (с фиксированной и с переменной длиной строки), дата. Чем больше разных типов данных вы используете, тем лучше. Приведите в отчете скриншоты терминала с выполненными командами и их результатом.

2. С помощью pgAdmin заполните таблицы данными: в каждую таблицу добавьте 10 строк.

**7. Контрольные вопросы**

1. Создать базу данных basa1, причем для данных на приросте 10%, для журнала транзакций – на диске E один файл с начальным размером 50 Мб, но не более 100 Мб, с величиной прироста 10 Мб.
2. Создать базу данных basa1, причем для данных на диске D определить два файла с начальным размером по 100 Мб, но не более 500 Мб, с величиной прироста 10%, для журнала транзакций - на диске E один файл с начальным размером 50 Мб, но не более 100 Мб, с величиной прироста 10 Мб.
3. Создать базу данных basa1, причем для данных на диске D определить два файла с начальным размером по 100 Мб, но не более 500 Мб, с величиной прироста 10%, для журнала транзакций - на диске E один файл с начальным размером 50 Мб, но не более 100 Мб, с величиной прироста 10 %.
4. Создать базу данных basa1, причем для данных на диске D определить два файла с начальным размером по 100 Мб, но не более 500 Мб, с величиной прироста 10%, для журнала транзакций - на диске E один файл с начальным размером 50 Мб, но не более 100 Мб, с величиной прироста 10 Мб.

**Лабораторная работа №3**

**Операторы модификации данных**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение и практическое применение операторов модификации данных.

**2. Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

**3. Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

В SQL для выполнения операций ввода данных в таблицу, их изменения и удаления предназначены три команды языка манипулирования данными (DML).

Это команды – **INSERT** (вставить), **UPDATE** (обновить), **DELETE** (удалить).

Команда **INSERT** осуществляет вставку в таблицу новой строки. В простейшем случае она имеет следующий вид:

**INSERTINTO***<имя таблицы>***VALUES** (*<значение>*, *<значение>*, ...);

При такой записи указанные в скобках после ключевого слова **VALUES** значения вводятся в поля добавленной в таблицу новой строки в том порядке, в котором соответствующие столбцы указаны при создании таблицы, то есть в операторе **CREATE TABLE**.

Добавление одной строки в таблицу films:

**INSERT** INTO films VALUES ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');

В этом примере столбец len опускается и, таким образом, получает значение по умолчанию:

**INSERT** INTO films (code, title, did, date\_prod, kind)

VALUES ('T\_601', 'Yojimbo', 106, '1961-06-16', 'Drama');

В этом примере для столбца с датой задаётся указание DEFAULT, а не явное значение:

**INSERT** INTO films VALUES

('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');

**INSERT** INTO films (code, title, did, date\_prod, kind)

VALUES ('T\_601', 'Yojimbo', 106, DEFAULT, 'Drama');

Добавление строки, полностью состоящей из значений по умолчанию:

**INSERT** INTO films DEFAULT VALUES;

В случаях, когда необходимо ввести значения полей в порядке, отличном от порядка столбцов, заданного командой **CREATE TABLE**, или если требуется ввести значения не во все столбцы, то следует использовать следующую форму команды **INSERT**:

**INSERT INTO** STUDENT (STUDENT\_ID, CITY, SURNAME, NAME)

**VALUES** (101, ' Москва ',' Иванов ', ' Саша ');

Столбцам, наименования которых не указаны в приведенном в скобках списке, автоматически присваивается значение по умолчанию, если оно назначено при описании таблицы (команда **CREATE TABLE**), либо значение **NULL**.

Применение оператора **INSERT** с подзапросом позволяет загружать сразу несколько строк в одну таблицу, используя информацию из другой таблицы. В то время как оператор **INSERT**, использующий **VALUES** добавляет только одну строку, **INSERT** с подзапросом добавляет в таблицу столько строк, сколько подзапрос извлекает из другой таблицы. При этом количество и тип возвращаемых подзапросом столбцов должно соответствовать количеству и типу столбцов таблицы, в которую вставляются данные.

С помощью команды **INSERT** можно извлечь значение из одной таблицы и разместить его в другой, к примеру, запросом следующего вида:

**INSERT INTO** STUDENT 1

**SELECT**\* **FROM** STUDENT **WHERE** CITY = ^ Москва ';

При этом таблица STUDENT 1должна быть предварительно создана командой **CREATE TABLE** и иметь структуру, идентичную таблице STUDENT.

Команда **DELETE** удаляет из указанной таблицы строки, удовлетворяющие условию WHERE. Если предложение WHERE отсутствует, она удаляет из таблицы все строки, в результате будет получена рабочая, но пустая таблица.

Удаление всех фильмов, кроме мюзиклов:

**DELETE** FROM films WHERE kind <> 'Musical';

Очистка таблицы films:

**DELETE** FROM films;

Удаление завершённых задач с получением всех данных удалённых строк:

**DELETE** FROM tasks WHERE status = 'DONE' RETURNING \*;

Команда **UPDATE** позволяет изменять, то есть обновлять, значения некоторых или всех полей в существующей строке или строках таблицы.

**UPDATE** изменяет значения указанных столбцов во всех строках, удовлетворяющих условию. В предложении SET должны указываться только те столбцы, которые будут изменены; столбцы, не изменяемые явно, сохраняют свои предыдущие значения.

Изменить строки в таблице, используя информацию из других таблиц в базе данных, можно двумя способами: применяя вложенные запросы или указав дополнительные таблицы в предложении FROM. Выбор предпочитаемого варианта зависит от конкретных обстоятельств.

Изменение слова Drama на Dramatic в столбце kind таблицы films:

**UPDATE** films SET kind = 'Dramatic' WHERE kind = 'Drama';

Изменение значений температуры и сброс уровня осадков к значению по умолчанию в одной строке таблицы weather:

**UPDATE** weather SET temp\_lo = temp\_lo+1, temp\_hi = temp\_lo+15, prcp = DEFAULT WHERE city = 'San Francisco' AND date = '2003-07-03';

Более подробная информация: <https://www.postgresql.org/docs/9.5/tutorial-sql.html>

**5. Оборудование**

Персональный компьютер с установленной операционной системой WindowsXP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБДPostgreSQL.

**6. Задание на работу**

1. Предположим, что вам нужно вставить в таблицу строчку и значение одного из полей вставляемой строки вам еще не известно. Напишите запрос в двух вариантах: в одном укажите названия вводимых полей явно, в другом - не указывайте имена полей.

2. Напишите запрос, обновляющий значение двух полей одной строки.

3. Напишите запрос, удаляющий значение одной строки.

4. Напишите запрос, переносящий для нескольких строк значение одного поля в другое поле.

5. Напишите запрос, переносящий несколько строк одной таблицы в другую таблицу.

6. Продемонстрируйте, каким образом можно заполнять значение первичного ключа таблицы автоматически. Создайте секвенцию, определите значение, генерируемое секвенцией по умолчанию.

7. Выполните запрос на вставку данных и получите только что вставленное значение первичного ключа (конструкция RETURNING ID) .

**7. Контрольные вопросы**

1. Даны таблицы Город и Разговор. Удалить данные об абонентах, суммарная продолжительность телефонных разговоров которых оказалась меньше 10 мин.

|  |  |
| --- | --- |
| CREATE TABLE Город  ( Код\_Города INT ,  Название VARCHAR(20) NOT NULL,  Тариф MONEY) | CREATE TABLE Разговор  (Код\_Разговора INT,  Код\_ГородаINTNOTNULL,  Фамилия VARCHAR(20),  Дата DATETIME,  Продолжительность INT) |

1. Даны таблицы Автор и Книга. Удалить сведения об авторах, издавших только одну книгу.

|  |  |
| --- | --- |
| CREATE TABLE Автор  ( Код\_Автора INT ,  Фамилия VARCHAR(50)) | CREATE TABLE Книга  ( Код\_Книги INT,  Название VARCHAR(50),  Цена MONEY,  Код\_Автора INT) |

1. Увеличить на 10% стоимость билетов, проданных в день вылета рейса.
2. Уменьшить на 50% продолжительность телефонных разговоров тех абонентов, которые за последний месяц сделали более 1000 звонков.
3. Даны таблицы

|  |  |
| --- | --- |
| CREATE TABLE Город  (Код\_Города INT,  Название VARCHAR(20) NOT NULL,  Тариф MONEY) | CREATE TABLE Разговор  (Код\_Разговора INT,  Код\_Города INT NOT NULL,  Фамилия IVARCHAR(20),  Дата DATETIME NOT NULL,  Продолжительность INT NOT NULL) |

Удалить информацию о городах, с которыми не было ни одного телефонного разговора.

1. Даны таблицы

|  |  |
| --- | --- |
| CREATE TABLE Блюдо  (Название\_блюдаVARCHAR(20) NOTNULL,  Время\_приготовления INT NOT NULL,  Общая\_калорийность INT NOT NULL,  Номер\_рецепта INT,  Повар VARCHAR(20),  Стоимость\_блюда MONEY) | CREATE TABLE Компонент  (Название\_компонента VARCHAR(20),  КалорийностьINTNOTNULL,  ЖирыINT,  БелкиINT,  БлюдоVARCHAR(20),  Углеводы INT,  Стоимость\_компонента MONEY NOT NULL) |

Удалить сведения о блюдах, стоимость которых меньше средней стоимости компонентов.

1. Даны таблицы:

|  |  |
| --- | --- |
| CREATE TABLE Автор  (Код\_Автора INT ,  Фамилия VARCHAR(50) NULL,  Имя VARCHAR(50) NULL,  Отчество VARCHAR(50) NULL,  Пол VARCHAR(50) NOT NULL ,  Дата\_рождения DATETIME ,  Телефон CHAR(9)) | CREATE TABLE Книга  (Код\_Книги INT,  Название VARCHAR(50) NOT NULL,  Цена MONEY,  Тематика VARCHAR(50) NOT NULL,  Издательство VARCHAR(50) NOT NULL,  Код\_Автора INT NOT NULL) |

Удалить сведения об авторах, чьи произведения не издаются.

**Лабораторная работа №4**

**Извлечение информации из таблиц**

1. **Цель и задачи работы**

Целью лабораторной работы является изучение и практическое применение возможностей SQL для извлечения информации из таблиц.

1. **Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

1. **Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

1. **Теоретические сведения**

Запрос - команда, которую вы даете вашей СУБД и которая сообщает ей чтобы она вывела определенную информацию из таблиц в память. Эта информация обычно посылается непосредственно на экран компьютера или терминала, которым вы пользуетесь, хотя, в большинстве случаев, ее можно также послать на принтер, сохранить в файле (как объект в памяти компьютера), или представить, как вводную информацию для другой команды или процесса.

В самой простой форме, команда SELECT просто инструктирует базу данных, чтобы извлечь информацию из таблицы. Например, можно вывести таблицу «Продавцов» напечатав следующее:

SELECTsnum, sname, sity, commFROMSalespeople;

Другими словами, эта команда просто выводит все данные из таблицы. Большинство программ будут также давать заголовки столбца как выше, а некоторые позволяют детальное форматирование вывода, но это уже вне стандартной спецификации. Имеется объяснение каждой части этой команды:

|  |  |
| --- | --- |
| SELECT | Ключевое слово, которое сообщает базе данных что эта команда – запрос. Все запросы начинаются этим словом, сопровождаемым пробелом. |
| snum  sname | Это – список столбцов из таблицы, которые выбираются запросом. Любые столбцы, не перечисленные здесь не будут включены в вывод команды. Это, конечно, не значит что они будут удалены или их информация будет стерта из таблиц, потому что запрос не воздействует на информацию в таблицах; он только показывает данные. |
| FROM | FROM - ключевое слово, подобно SELECT, которое должно Salespeople быть представлено в каждом запросе. Оно сопровождается пробелом и затем именем таблицы используемой в качестве источника информации. В данном случае – это таблица Продавцов (Salespeople). |
| ; | Точка с запятой используется во всех интерактивных командах SQL чтобы сообщать базе данных что команда заполнена и готова выполниться. В некоторых системах наклонна черта влево (\) в строке, является индикатором конца команды. |

Оператор SELECT имеет следующий вид:

SELECT [ ALL | DISTINCT [ ON ( выражение [, ...] ) ] ]

[ \* | выражение [ [ AS ] имя\_результата ] [, ...] ]

[ FROM элемент\_FROM [, ...] ]

[ WHERE условие ]

[ GROUP BY элемент\_группирования [, ...] ]

[ HAVING условие [, ...] ]

[ ORDERBY выражение ]

Формат запроса с использованием данного оператора:

SELECT список полей FROM список таблиц WHERE условия…

Основные ключевые слова, относящиеся к запросу SELECT:

WHERE — используется для определения, какие строки должны быть выбраны или включены в GROUP BY.

GROUP BY — используется для объединения строк с общими значениями в элементы меньшего набора строк.

HAVING — используется для определения, какие строки после GROUP BY должны быть выбраны.

ORDER BY— используется для определения, какие столбцы используются для сортировки результирующего набора данных.

Если вы хотите видеть каждый столбец таблицы, имеется необязательное сокращение, которое вы можете использовать. Звездочка (\*) может применяться для вывода полного списка столбцов следующим образом:

SELECT \* FROM Salespeople;

Условия использующие ключевое слово WHERE.

|  |  |
| --- | --- |
| Оператор | Описание |
| = | Отбираются значения равные указанному  SELECT \* FROMtopicsWHEREid\_author=4; |
| > | Отбираются значения больше указанного SELECT \* FROM topics WHERE id\_author>2; |
| < | Отбираются значения меньше указанного SELECT \* FROM topics WHERE id\_author<3; |
| >= | Отбираются значения большие и равные указанному SELECT \* FROM topics WHERE id\_author>=2; |
| <= | Отбираются значения меньшие и равные указанному SELECT \* FROM topics WHERE id\_author<=3; |
| != | Отбираются значения не равные указанному SELECT \* FROM topics WHERE id\_author!=1; |
| IS NOT NULL | Отбираются строки, имеющие значения в указанном поле SELECT \* FROM topics WHERE id\_author IS NOT NULL; |
| IS NULL | Отбираются строки, не имеющие значения в указанном поле SELECT \* FROM topics WHERE id\_author IS NULL; |
| BETWEEN (между) | Отбираются значения, находящиеся между указанными SELECT \* FROM topics  WHERE id\_author BETWEEN 1 AND 3; |
| IN (значение содержится) | Отбираются значения, соответствующие указанным SELECT \* FROM topics WHERE id\_author IN (1, 4); |
| NOT IN (значение не содержится) | Отбираются значения, кроме указанных SELECT \* FROMtopics  WHERE id\_author NOT IN (1, 4); |
| LIKE (соответствие) | Отбираются значения, соответствующие образцу SELECT \* FROM topics  WHERE topic\_name LIKE 'вел%'; |
| NOT LIKE  (не соответствие) | Отбираются значения, не соответствующие образцу SELECT \* FROM topics  WHERE topic\_name NOT LIKE 'вел%'; |

1. **Оборудование**

Персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

1. **Задание на работу**

1. Напишите запрос, выбирающий все поля и все строки из таблицы, используемой в данной лабораторной. При этом таблица должна содержать 10-15 строк. Приведите содержимое данной таблицы.

2. Напишите запрос, демонстрирующий выбор нескольких (не всех) полей таблицы с удалением дубликатов строк (DISTINCT).

2. Напишите запрос, демонстрирующий выбор всех полей (\*) и задание условий выборки в виде операций сравнения (>, <, =) и логических операций (AND, OR, NOT).

3. Напишите запрос, демонстрирующий работу конструкций IN, BETWEEN, IS NULL, IS NOT NULL.

4. Напишите запрос, демонстрирующий работу конструкции LIKE (с символами "%" и "\_").

5. Напишите запрос, демонстрирующий вычисление арифметических выражений как в условиях выборки (после WHERE), так и в списке выбора (после SELECT) с заданием имени для результата выражения.

Указания к выполнению заданий 1-5.

а) сформулируете смысл запроса на языке, понятном пользователю в данной предметной области, например, "Запрос, выбирающий данные о фамилии, имени и номере курса для студентов, получающих стипендию больше 1400"

б) приведите сам запрос

в) приведите результат выполнения запроса в виде скриншота. Данные (и параметры в условии выборки) должны быть подобраны таким образом, чтобы в результирующей выборке было 3-4 записи.

**7. Контрольные вопросы**

1. Дана таблица Рейс. Вывести в убывающем порядке список рейсов, вылетающих не позднее 1 апреля в Москву, Петербург или Самару, стоимость билета не более 1500 р.; в Саратов - не позднее 7 апреля, стоимость билета - от 500 до 800 р.
2. Дана таблица Город. Вывести в алфавитном порядке список городов Поволжского региона, в коде которых встречается цифра 9, а в названии города на втором месте стоит буква <д> или <ж>.
3. Дана таблица Автор. Вывести в алфавитном порядке фамилии авторов из Самары, в телефонном номере которых на первом или третьем месте стоит цифра от 5 до 8, а последними являются цифры 7 и 8.
4. Дана таблица Блюдо. Вывести в алфавитном порядке фамилии поваров, блюда которых относятся к десерту или выпечке, стоимость не превышает 50 руб., а калорийность не больше 300 ккал.
5. Дана таблица Рейс. Вывести список рейсов, продолжительность маршрутов которых не более 500 км и не менее 100 км, а стоимость билета - от 800 до 1500 руб.

**Лабораторная работа №5**

**Обобщение данных с помощью агрегатных функций**

1. **Цели и задачи работы**

Целью лабораторной работы является изучение и практическое применение методов обобщения данных с помощью агрегатных функций.

1. **Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

1. **Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

1. **Теоретические сведения**

Агрегирующие функции позволяют получать из таблицы сводную (агрегированную) информацию, выполняя операции над группой строк таблицы. Для задания в SELECT – запросе агрегирующих операций используются следующие ключевые слова:

|  |  |  |
| --- | --- | --- |
| **avg(*выражение*)** | smallint, int, bigint, real, double precision, numericили interval | арифметическое среднее для всех входных значений |
| **count(\*)** |  | количество входных строк |
| **count(*выражение*)** | any | количество входных строк, для которых значение *выражения* не равно NULL |
| **max(*выражение*)** | любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов | максимальное значение *выражения* среди всех входных данных |
| **min(*выражение*)** | любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов | минимальное значение *выражения* среди всех входных данных |
| **sum(*выражение*)** | smallint, int, bigint, real, double precision, numeric, interval или money | сумма значений *выражения* по всем входным данным |

Например, для определения среднего значения поля MARK (оценки) по всем записям таблицы EXAM \_ MARKS можно использовать запрос с функцией AVG следующего вида:

SELECT AVG(MARK) FROM EXAM\_MARKS; Для подсчета общего количества строк в таблице следует использовать

функцию COUNT со звёздочкой. SELECT COUNT (\*)

FROM EXAM \_ MARKS;

Аргументы DISTINCT и ALL позволяют, соответственно, исключать и включать дубликаты обрабатываемых функцией COUNT значений, при этом необходимо учитывать, что при использовании опции ALL значения NULL все равно не войдут в число подсчитываемых значений.

SELECT COUNT (DISTINCT SUBJID) FROM SUBJECT;

Предложение GROUP BY (ГРУППИРОВАТЬ ПО) позволяет группировать записи в подмножества, определяемые значениями какого-либо поля, и применять агрегирующие функции уже не ко всем записям таблицы, а раздельно к каждой сформированной группе.

Предположим, требуется найти максимальное значение оценки, полученной каждым студентом. Запрос будет выглядеть следующим образом:

SELECT STUDENT\_ID, MAX (MARK) FROM EXAM\_MARKS GROUP BY STUDENT\_ID;

Выбираемые из таблицы EXAM \_ MARKS записи группируются по значениям поля STUDENT \_ ID , указанного в предложении GROUP BY , и для каждой группы находится максимальное значение поля MARK . [Предложение GROUP BY](https://postgrespro.ru/docs/postgresql/9.6/sql-select#sql-groupby) группирует строки таблицы, объединяя их в одну группу при совпадении значений во всех перечисленных столбцах. Порядок, в котором указаны столбцы, не имеет значения. В результате наборы строк с одинаковыми значениями преобразуются в отдельные строки, представляющие все строки группы. Это может быть полезно для устранения избыточности выходных данных и/или для вычисления агрегатных функций, применённых к этим группам. В приведенном запросе рассматриваются группы записей, сгруппированные по идентификаторам студентов.

В конструкции GROUP BY для группирования может быть использовано более одного столбца. Например:

SELECT STUDENT\_ID,SUB J\_ID, MAX (MARK) FROM EXAM\_MARKS GROUPBY STUDENT\_ID, SUBJ\_ID;

В этом случае строки вначале группируются по значениям первого столбца, а внутри этих групп – в подгруппы по значениям второго столбца. Таким образом, GROUP BY не только устанавливает столбцы, по которым осуществляется группирование, но и указывает порядок разбиения столбцов на группы.

Следует иметь в виду, что в предложении GROUP BY должны быть указаны все выбираемые столбцы, приведенные после ключевого слова SELECT, кроме столбцов, указанных в качестве аргумента в агрегирующей функции.

Если таблица была сгруппирована с помощью GROUP BY, но интерес представляют только некоторые группы, отфильтровать их можно с помощью предложения HAVING, действующего подобно WHERE.

SELECT SUB J\_NAME, MAX(HOUR) FROM SUBJECT GROUP BY SUBJ\_NAME HAVING MAX(HOUR)>=72;

В условии, задаваемом предложением HAVING, указывают только поля или выражения, которые на выходе имеют единственное значение для каждой выводимой группы.

Влияние NULL – значений в функции count.

Если аргумент функции COUNT является константой или столбцом без пустых значений, то функция возвращает количество строк, к которым применимо определенное условие или группирование.

Если аргументом функции является столбец, содержащий пустое значение, то COUNT вернет число строк, не содержащих пустые значения, и к которым применимо определенное условие или группирование.

Если бы механизм NULL не был доступен, то неприменимые и отсутствующие значения пришлось бы исключать с помощью конструкции WHERE.

Поведение функции COUNT (\*) не зависит от пустых значений. Она возвратит общее количество строк в таблице.

Влияние NULL – значений в функции avg.

Среднее значение множества чисел равно сумме чисел, делённой на число элементов множества. Однако, если некоторые элементы пусты, то есть их значения неизвестны или не существуют, то деление на количество всех элементов множества приведет к неправильному результату.

Функция AVG вычисляет среднее значение всех известных значений множества элементов, то есть эта функция подсчитывает сумму известных значений и делит её на количество этих значений, а не на общее количество значений, среди которых могут быть NULL – значения. Если столбец состоит только из пустых значений, то функция AVG также возвратит NULL.

Более подробная информация: <https://www.postgresql.org/docs/9.5/tutorial-sql.html>

1. **Оборудование**

Персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

1. **Задания на работу**
2. Напишите запрос, выбирающий все поля и все строки из таблицы, используемой в данной лабораторной работе. При этом таблица должна содержать 10-15 строк. Приведите содержимое данной таблицы.
3. Напишите запрос, демонстрирующий возможности функций MAX и MIN.
4. Напишите запрос, демонстрирующий возможности функций AVG и SUM.
5. Напишите запрос, демонстрирующий работу конструкций COUNT, COUNT(\*) и COUNT(DISTINCT).
6. Напишите запрос, демонстрирующий работу конструкции GROUP BY
7. Напишите запрос, демонстрирующий работу конструкции HAVING.
8. Напишите запрос, демонстрирующий совместную работу конструкций HAVING и WHERE.

Указания к выполнению заданий 1-7.

а) сформулируете смысл запроса на языке, понятном пользователю в данной предметной области, например, «Запрос, выбирающий среднее значение стипендии по каждому курсу»

б) приведите сам запрос

в) приведите результат выполнения запроса. Данные (и параметры в условии выборки) должны быть подобраны таким образом, чтобы в результирующей выборке было 3-4 записи (или одна).

**7. Контрольные вопросы**

1. Определить количество и общую продолжительность разговоров для каждого региона, с городами которого осуществляли телефонную связь абоненты, чьи фамилии содержат слог <-ва->.
2. На какую сумму были проданы билеты на рейс до Москвы в день вылета?
3. В каком количестве и на какую сумму издавал свои книги автор Борисов в каждом издательстве?
4. Даны таблицы Город и Разговор.

|  |  |
| --- | --- |
| CREATE TABLE Город  ( Код\_Города INT ,  Название VARCHAR(20) NOT NULL,  Тариф MONEY,  Регион VARCHAR(20)) | CREATE TABLE Разговор  (Код\_Разговора INT ,  Код\_ГородаINTNOTNULL,  Фамилия VARCHAR(20),  Дата DATETIME NOT NULL,  Продолжительность INT NOT NULL) |

Рассчитать стоимость каждого телефонного разговора с Москвой

1. Даны таблицы Автор и Книга

|  |  |
| --- | --- |
| CREATE TABLE Автор  ( Код\_Автора INT ,  Фамилия VARCHAR(50) NULL) | CREATE TABLE Книга  ( Код\_Книги INT,  Название VARCHAR(50) NOT NULL,  Цена MONEY,  Издательство VARCHAR(50) NOT NULL,  Код\_Автора INT NOT NULL,  Количество INT) |

Книги каких авторов были проданы на сумму, превышающую 10000 руб.

1. Даны таблицы

|  |  |
| --- | --- |
| CREATE TABLE Город  ( Код\_Города INT ,  Название VARCHAR(20) NOT NULL,  Тариф MONEY,  Регион VARCHAR(20)) | CREATE TABLE Разговор  (Код\_Разговора INT,  Код\_Города INT NOT NULL,  Фамилия VARCHAR(20),  Дата DATETIME NOT NULL,  Продолжительность INT NOT NULL) |

Определить фамилии абонентов, общее время разговоров которых менее 10 мин, а общая стоимость оказалась больше 100 руб.

**Лабораторная работа №6**

**Запросы на объединение отношений**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение и практическое применение запросов на объединение отношений.

**2. Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

**3. Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

Можно поместить многочисленные запросы вместе и объединить их вывод используя предложение UNION. Предложение UNION объединяет вывод двух или более SQL запросов в единый набор строк и столбцов. Например, чтобы получить всех продавцов и заказчиков размещенных в Лондоне и вывести их как единое целое вы могли бы ввести:

SELECT snum, sname

FROM Salespeople

WHERE city = 'London'

**UNION**

SELECT cnum, cname

FROM Customers

WHERE city = 'London';

Как вы можете видеть, столбцы, выбранные двумя командами выведены так как если она была одна. Заголовки столбца исключены, потому что ни один из столбцов, выведенных объединением, не был извлечен непосредственно из только одной таблицы. Следовательно, все эти столбцы вывода не имеют никаких имен.

Кроме того, обратите внимание, что только последний запрос заканчивается точкой с запятой. Отсутствие точки с запятой дает понять SQL, что имеется еще одно или более запросов.

=============== SQL Execution Log ============

| |

| SELECT snum, sname |

| FROM Salespeople |

| WHERE city = 'London' |

| **UNION**  |

| SELECT cnum, cname |

| FROM Customers |

| WHERE city = 'London'; |

| ============================================= |

| |

| ----- -------- |

| 1001 Peel |

| 1004 Motika |

| 2001 Hoffman |

| 2006 Climens |

| |

=============================================

Когда два (или более) запроса подвергаются объединению, их столбцы вывода должны быть совместимы для объединения. Это означает, что каждый запрос должен указывать одинаковое число столбцов и в том же порядке что и первый, второй, третий, и так далее, и каждый должен иметь тип, совместимый с каждым. Значение совместимости типов - меняется. ANSI следит за этим очень строго и поэтому числовые пол должны иметь одинаковый числовой тип и размер, хотя некоторые имена, используемые ANSI для этих типов, являются - синонимами. Кроме того, символьные поля должны иметь одинаковое число символов.

Хорошо, что некоторые SQL программы обладают большей гибкостью чем это определяется ANSI. Типы, не определенные ANSI, такие как DATA и BINARY, обычно должны совпадать с другими столбцами такого же нестандартного типа. Длина строки также может стать проблемой. Большинство программ разрешают пол переменной длины, но они не обязательно будут использоваться с UNION. С другой стороны, некоторые программы (и ANSI тоже) требуют, чтобы символьные поля были точно равной длины. В этих вопросах вы должны проконсультироваться с документацией вашей собственной программы.

Более подробная информация: <https://www.postgresql.org/docs/9.5/sql.html>

**5. Оборудование**

персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

**6. Задание на работу**

1. Напишите запрос, демонстрирующий объединение (UNION) результатов двух запросов. Количество выбираемых столбцов и типы данных соответствующих столбцов обоих запросов должны совпадать.

2. Напишите запрос, демонстрирующий объединение (UNION) результатов трех запросов. Тип данных одного столбца у всех трех запросов должны быть разными (но приводимыми, например, целые числа, действительныечисла и строки). Эти типы данных должны быть указаны в дополнительном столбце результирующей выборки.

3. Дополните запрос из п.2 сортировкой по двум столбцам (сначала по возрастанию одного, затем по убыванию другого).

**7. Контрольные вопросы**

1. Вывести список авторов-женщин, работающих в жанре романа:

|  |  |
| --- | --- |
| CREATE TABLE Автор  (Код\_Автора INT ,  Фамилия VARCHAR(50),  Пол VARCHAR(50) NOT NULL ) | CREATE TABLE Книга  ( Код\_Книги INT,  Название VARCHAR(50) NOT NULL,  Тематика VARCHAR(50) NOT NULL,  Издательство VARCHAR(50) NOT NULL,  Код\_Автора INT NOT NULL) |

1. Даны таблицы Город и Разговор:

|  |  |
| --- | --- |
| CREATE TABLE Город  ( Код\_Города INT ,  Название VARCHAR(20) NOT NULL,  Тариф MONEY) | CREATE TABLE Разговор  (Код\_Разговора INT ,  Код\_ГородаINTNOTNULL,  Фамилия VARCHAR(20),  Дата DATETIME NOT NULL,  Продолжительность INT NOT NULL) |

Вывести список абонентов, которые говорили с Москвой в апреле.

1. Даны таблицы Рейс и Билет:

|  |  |
| --- | --- |
| CREATE TABLE Рейс  (Номер\_рейса INT,  Конечный\_пункт VARCHAR(30),  Дата\_вылета DATETIME) | CREATE TABLE БИЛЕТ  (Номер\_места CHAR(3),  Номер\_рейса CHAR(6),  Дата\_продажи DATETIME,  Фамилия\_пассажира VARCHAR(30)) |

Определить номера мест и дату продажи билетов на рейсы до Москвы с датой вылета 1 мая 2004 года

1. Даны таблицы Автор и Книга:

|  |  |
| --- | --- |
| CREATE TABLE Автор  (Код\_Автора INT ,  Фамилия VARCHAR(50),  Пол VARCHAR(50) NOT NULL ) | CREATE TABLE Книга  ( Код\_Книги INT,  Название VARCHAR(50) NOT NULL,  Тематика VARCHAR(50) NOT NULL,  Издательство VARCHAR(50) NOT NULL,  Код\_Автора INT NOT NULL) |

Вывести список авторов, работающих в жанре детектив.

1. Даны таблицы:

|  |  |
| --- | --- |
| CREATE TABLE Блюдо  (Название\_блюда VARCHAR(20) NOT NULL,  Время\_приготовления INT NOT NULL,  Общая\_калорийность INT NOT NULL,  Номер\_рецепта INT,  Повар VARCHAR(20),  Стоимость INT ) | CREATE TABLE Компонент  (Название\_компонента VARCHAR(20),  КалорийностьINTNOTNULL,  ЖирыINT,  БелкиINT,  БлюдоVARCHAR(20),  Углеводы INT,  Стоимость\_100\_грамм FLOAT NOT NULL) |

Сформировать список поваров, которые используют масло.

**Лабораторная работа №7**

**Запросы на соединение отношений**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение и практическое применение запросов на соединение отношений.

**2. Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

**3. Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

Если в операторе **SELECT** после ключевого слова **FROM** указывается не одна, а две таблицы, то в результате выполнения запроса, в котором отсутствует предложение **WHERE**, каждая строка одной таблицы будет соединена с каждой строкой второй таблицы. Такая операция называется *декартовым произведением* или *полным* (**CROSS**) *соединением* таблиц базы данных. Сама по себе эта операция не имеет практического значения, более того, при ошибочном использовании она может привести к неожиданным нештатным ситуациям, так как в этом случае в ответе на запрос количество записей будет равно произведению числа записей в соединяемых таблицах, то есть может оказаться чрезвычайно большим. Соединение таблиц имеет смысл тогда, когда соединяются *не все* строки исходных таблиц, а только те, которые интересуют пользователя. Такое ограничение может быть осуществлено с помощью использования в запросе соответствующего условия в предложении **WHERE**. Таким образом, SQL позволяет выводить информацию из нескольких таблиц, связывая их по значениям определенных полей.

Например, если необходимо получить фамилии студентов (таблица STUDENT) и для каждого студента - названия университетов (таблица UNIVERSITY), расположенных в городе, где живет студент, то необходимо получить все комбинации записей о студентах и университетах в обеих таблицах, в которых значение поля CITY совпадает. Это можно сделать с помощью следующего запроса.

**SELECT**STUDENT.SURNAME, UNIVERSITY.UNIV\_NAME, STUDENT.CITY

**FROM** STUDENT, UNIVERSITY

**WHERE** STUDENT.CITY= UNIVERSITY.CITY;

Возможные типы соединений с сопоставлениями строк:

**INNER JOIN**

Для каждой строки R1 из Tаблицы 1 в результирующей таблице содержится строка для каждой строки в Tаблицы 2, удовлетворяющей условию соединения с R1.

**LEFT OUTER JOIN**

Сначала выполняется внутреннее соединение (INNER JOIN). Затем в результат добавляются все строки из Tаблицы 1, которым не соответствуют никакие строки в Таблице2, а вместо значений столбцов Tаблицы 2 вставляются NULL. Таким образом, в результирующей таблице всегда будет минимум одна строка для каждой строки из Tаблицы 1.

**RIGHT OUTER JOIN**

Сначала выполняется внутреннее соединение (INNER JOIN). Затем в результат добавляются все строки из Tаблицы 2, которым не соответствуют никакие строки в Таблице1, а вместо значений столбцов Tаблицы 1 вставляются NULL. Это соединение является обратным к левому (LEFT JOIN): в результирующей таблице всегда будет минимум одна строка для каждой строки из Tаблицы 2.

**FULL OUTER JOIN**

Сначала выполняется внутреннее соединение. Затем в результат добавляются все строки из T1, которым не соответствуют никакие строки в T2, а вместо значений столбцов T2 вставляются NULL. И наконец, в результат включаются все строки из T2, которым не соответствуют никакие строки в T1, а вместо значений столбцов T1 вставляются NULL.

Соединение, использующее предикаты, основанные на равенствах, называется *эквисоединением*. Рассмотренный пример соединения таблиц относятся к виду так называемого *внутреннего* (**INNER**) *соединения*. При таком типе соединения соединяются только те строки таблиц, для которых является истинным предикат, задаваемый в предложении **ON** выполняемого запроса.

Приведенный выше запрос может быть записан иначе, с использованием ключевого слова **JOIN**.

**SELECT** STUDENT.SURNAME, UNIVERS ITY.UNIV\_NAME, STUDENT.CITY

**FROM** STUDENT **INNERJOIN** UNIVERSITY

**ON** STUDENT.CITY= UNIVERSITY.CITY;

Ключевое слово **INNER** в запросе может быть опущено, так как эта опция в операторе **JOIN** действует по умолчанию.

Рассмотренный выше случай полного соединения (декартова произведения таблиц) с использованием ключевого слова **JOIN** будет выглядеть следующим образом

**SELECT** \* **FROM** STUDENT **JOIN** UNIVERSITY;

что эквивалентно

**SELECT** \* **FROM** STUDENT, UNIVERSITY;

Информация в таблицах STUDENT и EXAM \_ MARKS уже связана посредством поля STUDENT \_ ID. В таблице STUDENT поле STUDENT \_ ID является первичным ключом, а в таблице EXAM \_ MARKS, ссылающимся на него внешним ключом. Состояние связанных таким образом таблиц называется состоянием ссылочной целостности. В данном случае ссылочная целостность этих таблиц подразумевает, что *каждому* значению поля STUDENT \_ ID в таблице EXAM \_ MARKS *обязательно* соответствует *такое же значение* поля STUDENT \_ ID в таблице STUDENT. Другими словами, в таблице EXAM \_ MARKS не может быть записей, имеющих идентификаторы студентов, которых нет в таблице STUDENT. Стандартное применение операции соединения состоит в извлечении данных в терминах этой связи.

Чтобы получить список фамилий студентов с полученными ими оценками и идентификаторами предметов можно использовать следующий запрос:

**SELECT** SURNAME, MARK,SUBJ\_ID

**FROM** STUDENT, EXAM\_MARKS

**WHERE** STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID;

Тот же самый результат может быть получен при использовании в запросе для задания операции соединения таблиц ключевого слова **JOIN**. Запрос с оператором **JOIN**выглядитследующимобразом

**SELECT** SURNAME,MARK

**FROM** STUDENT **JOIN** EXAM\_MARKS

ON STUDENT. STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID;

Хотя выше речь шла о соединении двух таблиц, можно сформировать запросы путем соединения более чем двух таблиц.

Пусть требуется найти фамилии всех студентов, получивших неудовлетворительную оценку, вместе с названиями предметов обучения, по которым получена эта оценка.

**SELECT** SUB J\_NAME, SURNAME, MARK

**FROM** STUDENT, SUBJECT, EXAM\_MARKS

**WHERE** STUDENT.STUDENT\_ID= EXAM\_MARKS.STUDENT\_ID

**AND** SUBJECT. SUB J ID = EXAM MARKS. SUB J ID

**AND** EXAM \_ MARKS . MARK = 2;

To же самое с использованием опера- тора **JOIN**

**SELECT** SUBJ\_NAME, SURNAME, MARK

**FROM** STUDENT **JOIN** SUBJECT

**JOIN** EXAM\_MARKS ON STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID

**AND** SUB JECT.SUB J\_ID = EXAM\_MARKS.SUB J\_ID **AND** EXAM\_MARKS.MARK= 2;

Как отмечалось ранее, при использовании *внутреннего* (**INNER**) соединения таблиц соединяются только те их строки, в которых совпадают значения полей, задаваемые в предложении **WHERE** запроса. Однако во многих случаях это может привести к нежелательной потере информации. Рассмотрим еще раз приведенный выше пример запроса на выборку списка фамилий студентов с полученными ими оценками и идентификаторами предметов. При использовании, как это было сделано в рассматриваемом примере, внутреннего соединения в результат запроса не попадут студенты, которые еще не сдавали экзамены и которые, следовательно, отсутствуют в таблице EXAM \_ MARKS. Если же необходимо иметь записи об этих студентах в выдаваемом запросом списке, то можно присоединить сведения о студентах, не сдававших экзамен, путем использования оператора **UNION** с соответствующим запросом. Например, следующим образом:

**SELECT** SURNAME,**CAST** MARK **AS**CHAR(1), **CAST** SUBJ\_ID **AS** CHAR(IO)

**FROM**STUDENT, EXAM\_MARKS

**WHERE** STUDENT.STUDENT\_ID= EXAM\_MARKS.STUDENT\_ID

**UNIONSELECT** SURNAME,**CASTNULLAS**CHAR(l), **CASTNULLAS** CHAR(IO)

**FROM** STUDENT **WHERE** NOT EXIST

(**SELECT** \* **FROM**EXAM\_MARKS **WHERE**STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID) ;

(здесь функция преобразования типов **CAST** используется для обеспечения совместимости типов полей объединяемых запросов).

Нужный результат, однако, может быть получен и путем использования *внешнего соединения*, точнее одной из его разновидностей – *левого внешнего соединения*, с использованием которого запрос будет выглядеть следующим образом:

**SELECT** SURNAME,MARK**FROM** STUDENT **LEFTOUTER**

**JOIN** EXAM\_MARKS ON STUDENT.STUDENT\_ID = EXAM\_MARKS.STUDENT\_ID;

При использовании *левого* соединения расширение выводимой таблицы осуществляется за счет записей входной таблицы, имя которой указано *слева* от оператора **JOIN**.

Приведенный выше запрос может быть реализован и с применением *правого внешнего соединения*. Он будет иметь следующий вид

**SELECT** SURNAME,MARK

**FROM** EXAM\_MARKS

**RIGHT OUTER JOIN** STUDENT ON EXAM\_MARKS.STUDENT\_ID = STUDENT.STUDENT\_ID;

Здесь таблица STUDENT, за счет записей которой осуществляется расширение выводимой таблицы, стоит справа от оператора **JOIN**

Видно, что использование внешнего правого или левого соединения позволяет существенно упростить запрос, сделать его запись более компактной.

Иногда возникает необходимость включения в результат запроса записей из обеих (правой и левой) соединяемых таблиц, для которых не удовлетворяется условие соединения. Такое соединение называется *полным внешним соединением* и осуществляется указанием в запросе ключевых слов FULL OUTER JOIN или UNION JOIN.

Часто при получении информации из таблиц базы данных необходимо осуществлять соединение таблицы с ее же копией. Например, это требуется в случае, когда требуется найти фамилии студентов, имеющих одинаковые имена. При соединении таблицы с ее же копией вводят псевдонимы (алиасы) таблицы. Запрос для поиска фамилий студентов, имеющих одинаковые имена, выглядит следующим образом

**SELECT** FIRS Т . SURNAME, SECOND. SURNAME

**FROM** STUDENT FIRST, STUDENT SECOND

**WHERE**FIRST . NAME = SECOND . NAME

В этом запросе введены два псевдонима для одной таблицы STUDENT, что позволяет корректно задать выражение, связывающее две копии таблицы. Чтобы исключить повторения строк в выводимом результате запроса из-за повторного сравнения одной и той же пары студентов, необходимо задать порядок следования для двух значений так, чтобы одно значение было меньше, чем другое, что делает предикат асимметричным.

**SELECT** FIRS Т . SURNAME, SECOND. SURNAME

**FROM** STUDENT FIRST, STUDENT SECOND

**WHERE** FIRST.NAME= SECOND.NAME

**AND** FIRST.SURNAME< SECOND.SURNAME.

Более подробная информация: <https://www.postgresql.org/docs/9.5/sql.html>

**5. Оборудование**

Оборудование: персональный компьютер с установленной операционной системой WindowsXP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБДPostgreSQL.

**6. Задание на работу**

1. Напишите запрос, демонстрирующий соединение двух таблиц с помощью конструкции SELECT ... FROM TABLE1, TABLE2 WHERE... . Перепишите тот же запрос с помощью конструкции JOIN. Убедитесь, что результаты выполнения запросов одинаковы.

2. Напишите запрос, демонстрирующий смысл и назначение конструкции LEFT JOIN. Перепишите его с помощью конструкции RIGHT JOIN. Убедитесь, что результаты выполнения запросов одинаковы.

3. Напишите запрос, в котором таблица соединяется (JOIN) сама с собой.

4. Напишите запрос, в котором агрегация происходит по результату соединения таблиц. То есть, в запросе должны присутствовать агрегирующая функция (SUM, AVG, MAX, MIN или COUNT), GRUOP BY и HAVING, WHERE и JOIN (внутренний или внешний). Будьте внимательны к этому заданию, оно высоко оценивается.

**7. Контрольные вопросы**

1. Найти абонентов, которые звонят в Москву, но ни разу не звонили в Самару в мае:

|  |  |
| --- | --- |
| CREATE TABLE Город  ( Код\_Города INT ,  Название VARCHAR(20) NOT NULL,  Тариф MONEY) | CREATE TABLE Разговор  (Код\_Разговора INT ,  Код\_Города INT NOT NULL,  Фамилия VARCHAR(20),  Дата DATETIME NOT NULL,  Продолжительность INT NOT NULL) |

1. Вывести список авторов-женщин, работающих в жанре романа, но не в жанре фантастики:

|  |  |
| --- | --- |
| CREATE TABLE Автор  (Код\_Автора INT ,  Фамилия VARCHAR(50),  Пол VARCHAR(50) NOT NULL ) | CREATE TABLE Книга  ( Код\_Книги INT,  Название VARCHAR(50) NOT NULL,  Тематика VARCHAR(50) NOT NULL,  Издательство VARCHAR(50) NOT NULL,  Код\_Автора INT NOT NULL) |

1. Даны таблицы Город и Разговор:

|  |  |
| --- | --- |
| CREATE TABLE Город  ( Код\_Города INT ,  Название VARCHAR(20) NOT NULL,  Тариф MONEY) | CREATE TABLE Разговор  (Код\_Разговора INT ,  Код\_ГородаINTNOTNULL,  Фамилия VARCHAR(20),  Дата DATETIME NOT NULL,  Продолжительность INT NOT NULL) |

Вывести список абонентов, которые говорили с Москвой в апреле, но не с Тулой.

1. Даны таблицы Рейс и Билет:

|  |  |
| --- | --- |
| CREATE TABLE Рейс  (Номер\_рейса INT,  Конечный\_пункт VARCHAR(30),  Дата\_вылета DATETIME) | CREATE TABLE БИЛЕТ  (Номер\_места CHAR(3),  Номер\_рейса CHAR(6),  Дата\_продажи DATETIME,  Фамилия\_пассажира VARCHAR(30)) |

Определить номера мест и дату продажи билетов на рейсы до Москвы с датой вылета не позднее 1 мая 2004 года.

1. Даны таблицы:

|  |  |
| --- | --- |
| CREATE TABLE Блюдо  (Название\_блюда VARCHAR(20) NOT NULL,  Время\_приготовления INT NOT NULL,  Общая\_калорийность INT NOT NULL,  Номер\_рецепта INT,  Повар VARCHAR(20),  Стоимость INT ) | CREATE TABLE Компонент  (Название\_компонента VARCHAR(20),  КалорийностьINTNOTNULL,  ЖирыINT,  БелкиINT,  БлюдоVARCHAR(20),  Углеводы INT,  Стоимость\_100\_грамм FLOAT NOT NULL) |

Сформировать список поваров, которые используют масло, но не молоко.

**Лабораторная работа №8**

**Подзапросы**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение и практическое применение подзапросов.

**2. Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

**3. Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

SQL позволяет использовать одни запросы внутри других запросов, то есть вкладывать запросы друг в друга. Предположим, известна фамилия студента (**"Петров"**), но неизвестно значение поля STUDENT \_ ID для него. Чтобы извлечь данные обо всех оценках этого студента, можно записать следующий запрос:

**SELECT** \* **FROM** EXAM\_MARKS

**WHERE** STUDENT\_ID = (**SELECT** STUDENT\_ID **FROM** STUDENT SURNAME =' Петров ');

Как работает запрос SQL со связанным подзапросом?

• Выбирается строка из таблицы, имя которой указано во внешнем запросе.

• Выполняется подзапрос и полученное в результате его выполнения значение применяется для анализа этой строки в условии предложения **WHERE** внешнего запроса.

• По результату оценки этого условия принимается решение о включении или не включении строки в состав выходных данных.

• Процедура повторяется для следующей строки таблицы внешнего запроса.

Следует обратить внимание, что приведенный выше запрос корректен только в том случае, если в результате выполнения указанного в скобках подзапроса возвращается *единственное значение*. Если в результате выполнения подзапроса будет возвращено несколько значений, то этот подзапрос будет ошибочным. В данном примере это произойдет, если в таблице STUDENT будет несколько записей со значениями поля surname = 'Петров'.

В некоторых случаях для гарантии получения единственного значения в результате выполнения подзапроса используется **DISTINCT**. Одним из видов функций, которые автоматически *всегда* выдают в результате единственное значение для любого количества строк, являются агрегирующие функции.

Оператор **IN** также широко применяется в подзапросах. Он задает список значений, с которыми сравниваются другие значения для определения истинности задаваемого этим оператором предиката.

Данные обо всех оценках (таблица EXAM \_ MARKS) студентов из Воронежа можно выбрать с помощью следующего запроса:

**SELECT** \* **FROM** EXAM\_MARKS

**WHERE** STUDENT\_ID

**IN** (**SELECT** STUDENT\_ID **FROM** STUDENT **WHERE** CI TY = ' Воронеж ');

Подзапросы можно применять внутри предложения **HAVING**. Пусть требуется определить количество предметов обучения с оценкой, превышающей среднее значение оценки студента с идентификатором 301:

**SELECTCOUNT** (**DISTINCT** SUB JID) , MARK

**FROM** EXAM\_MARKS **GROUP BY** MARK

**HAVING** MARK >( **SELECT AVG** (**MARK**)

**FROM** EXAM\_MARKS **WHERE** STUDENT \_ ID = **301**);

**Формирование связанных подзапросов.** При использовании подзапросов во внутреннем запросе можно ссылаться на таблицу, имя которой указано в предложении **FROM** внешнего запроса. В этом случае такой *связанный* подзапрос выполняется по одному разу для *каждой* строки таблицы основного запроса.

**Пример:** выбрать сведения обо всех предметах обучения, по которым проводился экзамен 20 января1999 г.

**SELECT** \* **FROM** SUBJECT SU **WHERE'20/01/1999'**

**IN** (**SELECT** EXAM\_DATE **FROM** EXAM\_MARKS EX

**WHERE** SU.SUB J\_ID = EX.SUB J\_ID);

В некоторых СУБД для выполнения этого запроса, возможно, потребуется преобразование значения даты в символьный тип. В приведенном запросе SU и ЕХ являются псевдонимами (алиасами), то есть специально вводимыми именами, которые могут быть использованы в данном запросе вместо настоящих имен. В приведенном примере они используются вместо имен таблиц SUBJECT и EXAMJMARKS.

Эту же задачу можно решить с помощью операции соединения таблиц:

**SELECTDISTINCT** SU.SUB J\_ID,SUBJ\_NAME, HOUR, SEMESTER

**FROM** SUBJECT FIRST, EXAM\_MARKS SECOND

**WHERE** FIRST.SUBJ\_ID= SECOND.SUB J\_ID

**AND** SECOND.EXAM\_DATE = **'20/01/1999'**;

В этом выражении алиасами таблиц являются имена FIRST и SECOND.

Можно использовать подзапросы, связывающие таблицу со своей собственной копией. Например, надо найти идентификаторы, фамилии и стипендии студентов, получающих стипендию выше средней на курсе, на котором они учатся.

**SELECTDISTINCT** STUDENT\_ID,SURNAME, STIPEND

**FROM** STUDENT El

**WHERE** STIPEND > (**SELEC**T AVG(STIPEND)

**FROM** STUDENT E2 **WHERE**El . KURS = E 2. KURS );

Тот же результат можно получить с помощью следующего запроса:

**SELECTDISTINCT** STUDENT\_ID, SURNAME, STIPEND

**FROM** STUDENT El, (**SELECT** KURS, AVG(STIPEND)

**AS** AVG\_STIPEND **FROM** STUDENT E2

**GROUP BY** E2.KURS) E3

**WHERE** E1.STIPEND>AVG\_STIPENDANDE1.KURS=E3.KURS;

Обратите внимание - второй запрос будет выполнен гораздо быстрее. Дело в том, что в первом варианте запроса агрегирующая функция AVG выполняется над таблицей, указанной в подзапросе, для *каждой* строки внешнего запроса. В другом варианте вторая таблица (алиас Е2) обрабатывается агрегирующей функцией один раз, в результате чего формируется вспомогательная таблица (в запросе она имеет алиас ЕЗ), со строками которой затем соединяются строки первой таблицы (алиас Е1). Следует иметь в виду, что реальное время выполнения запроса в большой степени зависит от оптимизатора запросов конкретной СУБД.

Пусть, например, необходимо по данным из таблицы EXAM \_ MARKS определить сумму полученных студентами оценок (значений поля MARK), сгруппировав значения оценок по датам экзаменов и исключив те дни, когда число студентов, сдававших в течение дня экзамены, было меньше 10.

**SELECT** EXAM\_DATE, SUM(MARK)

**FROM** EXAM\_MARKS A **GROUP BY** EXAM DATE

**HAVING** 10 < (**SELECT**COUNT(MARK) **FROM** EXAM\_MARKS В

**WHERE** A.EXAM\_DATE = B.EXAM\_DATE);

Подзапрос вычисляет количество строк с одной и той же датой, совпадающей с датой, для которой сформирована очередная группа основного запроса.

Используемый в SQL оператор **EXISTS** (СУЩЕСТВУЕТ) генерирует значение истина или ложь, подобно булеву выражению. Используя подзапросы в качестве аргумента, этот оператор оценивает результат выполнения подзапроса как истинный, если этот подзапрос генерирует выходные данные, то есть в случае *существования* (возврата) хотя бы одного найденного значения. В противном случае результат подзапроса-ложный. Оператор **EXISTS** не может принимать значение **unknown** (неизвестно).

Пусть, например, нужно извлечь из таблицы EXAM MARKS данные о студентах, получивших хотя бы одну неудовлетворительную оценку.

**SELECT DISTINCT** STUDENT\_ID **FROM** EXAM\_MARKS A

**WHERE** EXISTS (**SELECT** \* **FROM** EXAM\_MARKS В

**WHERE** MARK < 3 **AND** B.STUDENT\_ID=A.STUDENT\_ID);

При использовании связанных подзапросов предложение **EXISTS** анализирует каждую строку таблицы, на которую имеется ссылка во внешнем запросе. Главный запрос получает строки-кандидаты на проверку условия. Для каждой строки-кандидата выполняется подзапрос. Как только подзапрос находит строку, где в столбце MARK значение удовлетворяет условию, он прекращает выполнение и возвращает значение **истина** внешнему запросу, который затем анализирует свою строку-кандидата.

Например, требуется получить идентификаторы предметов обучения, экзамены по которым сдавались не одним, а несколькими студентами:

**SELECT DISTINCT** SUBJID

**FROM** EXAM\_MARKS A **WHEREEXISTS** (**SELECT** \*

**FROM** EXAM\_MARKS В**WHERE** A.SUBJ\_ID =B.SUBJ\_ID

**AND** A.STUDENT\_ID<>B.STUDENT\_ID);

Часто **EXISTS** применяется с оператором **NOT** (по-русски **NOT EXISTS** интерпретируется, как "*не существует...*"). Если предыдущий запрос сформулировать следующим образом - найти идентификаторы предметов обучения, которые сдавались одним и только одним студентом (другими словами, для которых не существует другого сдававшего студента), то достаточно просто поставить **NOT** перед **EXISTS**.

Следует иметь в виду, что в подзапросе, указываемом в операторе **EXISTS**, *нельзя использовать агрегирующие функции*.

Возможности применения вложенных запросов весьма разнообразны. Например, пусть из таблицы STUDENT требуется извлечь строки для каждого студента, сдавшего более одного предмета.

**SELECT** \* **FROM** STUDENT FIRST

**WHERE EXISTS** (**SELECT** SUBJ\_ID **FROM** EXAM\_MARKS S E COND **GROUP BY** SUBJID

**HAVING**COUNT(SUBJID) **> 1 WHERE** FIRST.STUDENT\_ID= SECOND.STUDENT\_ID);

Операторы сравнения с множеством значений имеют следующий смысл. IN *Равно* любому из значений, полученных во внутреннем запросе. NOT IN *He равно* ни одному из значений, полученных во внутреннем запросе.

= ANY To же, что и IN . Соответствует логическому оператору OR .

> ANY , > = ANY

*Больше, чем* (либо *больше или равно*) любое полученное число. Эквивалент- но > или > = для самого меньшего полученного числа.

< ANY , < = ANY

*Меньше, чем* (либо *меньше или равно*) любое полученное число. Эквивалент < или < = для самого большего полученного числа.

= ALL

Равно всем полученным значениям. Эквивалентно логическому оператору AND .

> ALL , > = ALL

*Больше, чем* (либо *больше или равно*) все полученные числа. Эквивалент > или > = для самого большего полученного числа.

< ALL , < = ALL

*Меньше, чем* (либо *меньше или равно*) все полученные числа. Эквивалентно < или < =самого меньшего полученного числа.

Следует иметь в виду, что в некоторых СУБД поддерживаются не все из этих операторов.

Примеры запросов с использованием приведенных операторов.

Выбрать сведения о студентах, проживающих в городе, где расположен университет, в котором они учатся.

**SELECT** \* **FROM** STUDENT S

**WHERE** CITY **= ANY** (**SELECT** CITY **FROM** UNIVERSITY U

**WHERE** U.UNIV\_ID = S.UNIV\_ID);

Другой вариант этого запроса

**SELECT** \* **FROM** STUDENT S

**WHERE** CITYIN (**SELECT** CITY **FROM** UNIVERSITY U

**WHERE** U.UNIV\_ID = S.UNIV\_ID);

Выборка данных об идентификаторах студентов, у которых оценки превосходят величину, по крайней мере, одной из оценок, полученных ими же 6 октября 1999 года.

**SELECT DISTINCT** STUDENT\_ID

**FROM** EXAM\_MARKS**WHERE** MARK **> ANY** (**SELECT** MARK

**FROM** EXAM\_MARKS **WHERE** EXAMDATE = '06/10/1999');

Оператор **ALL**, как правило, эффективно используется с неравенствами, а не с равенствами, поскольку значение *равно всем*, которое должно получиться в этом случае в результате выполнения подзапроса, может иметь место, только если все результаты идентичны. Такая ситуация практически не может быть реализована, так как, если подзапрос генерирует множество различных значений, то никакое одно значение не может быть равно сразу всем значениям в обычном смысле. В SQL выражение **<> ALL** реально означает *не равно ни одному* из результатов подзапроса.

Подзапрос, выбирающий данные о названиях всех университетов с рейтингом более высоким, чем рейтинг любого университета в Воронеже:

**SELECT** \* **FROM** UNIVERSITY

**WHERE** RATING **> ALL** (**SELECT** RATING

**FROM** UNIVERSITY **WHERE** CI TY= ' Воронеж ');

В этом запросе вместо **ALL** можно также использовать **ANY**. (Проанализируйте, как в этом случае изменится смысл приведенного запроса?)

**SELECT** \* **FROM** UNIVERSITY

**WHERE** NOTRATING**>ANY** (**SELECT** RATING

**FROM** UNIVERSITY **WHERE**CITY = ' Воронеж ');

Необходимо иметь в виду, что при обработке NULL -значений следует учитывать различие реакции на них операторов **EXISTS**, **ANY** и **ALL** .

Когда правильный подзапрос не генерирует никаких выходных данных, оператор **ALL** автоматически принимает значение **истина**, а оператор **ANY** - значение **ложь**.

Запрос

**SELECT** \* **FROM** UNIVERSITY

**WHERE** RATING **> ANY** ( **SELECT** RATING

**FROM** UNIVERSITY **WHERE** CITY = ' New York');

не генерирует выходных данных (подразумевается, что в базе нет данных об университетах из города NewYork ), в то время как запрос

**SELECT** \* **FROM** UNIVERSITY

**WHERE** RATING **> ALL** (**SELECT** RATING

**FROM** UNIVERSITY **WHERE** CITY = ' New York');

полностью воспроизведет таблицу UNIVERSITY.

Использование NULL-значений создает определенные проблемы для рассматриваемых операторов. Когда в SQL сравниваются два значения, одно из которых NULL -значение, результат принимает значение UNKNOWN (неизвестно). Предикат UNKNOWN, так же как и FALSE -предикат, создает ситуацию, когда строка не включается в состав выходных данных, но результат при этом будет различен для разных типов запросов, в зависимости от использования в них ALL или ANY вместо EXISTS. Рассмотрим в качестве примера две реализации запроса: найти все данные об университетах, рейтинг которых меньше рейтинга любого университета в Москве.

1. **SELECT \* FROM** UNIVERSITY

**WHERE** RATING < ANY (**SELECT** RATING **FROM** UNIVERSITY

**WHERE** CITY= ' Москва ');

1. **SELECT \* FROM** UNIVERSITYA

**WHERE NOT EXISTS** (**SELECT \* FROM** UNIVERSITY В

**WHERE** A. RATING >= В.RATING **AND** B.CITY=' Москва ');

При отсутствии в таблицах NULL оба эти запроса ведут себя совершенно одинаково. Пусть теперь в таблице UNIVERSITY есть строка с NULL - значениями в столбце RATING. В версии запроса с ANY в основном запросе, когда выбирается поле RATING с NULL, предикат принимает значение UNKNOWN и строка не включается в состав выходных данных. Во втором же варианте запроса, когда NOT EXISTS выбирает эту строку в основном запросе, NULL -значение используется в предикате подзапроса, присваивая ему значение UNKNOWN. Поэтому в результате выполнения подзапроса не будет получено ни одного значения и подзапрос примет значение ложь. Это в свою очередь сделает NOT EXISTS истинным, и, следовательно, строка с NULL значением в поле RATING попадет в выходные данные. По смыслу запроса такой результат является неправильным, так как на самом деле рейтинг университета, описываемого данной строкой может быть и больше рейтинга какого-либо московского университета (он просто неизвестен). Указанная проблема связана с тем, что значение EXISTS всегда принимает значения истина или ложь, и никогда – UNKNOWN. Это является доводом для использования в таких случаях оператора ANY вместо EXISTS.

При отсутствии NULL-значений оператор EXISTS может быть использован вместо ANY и ALL. Также вместо EXISTS и NOT EXISTS могут быть использованы те же самые подзапросы, но с использованием COUNT (\*) в предложении SELECT. Например, запрос

**SELECT** \* **FROM** UNIVERSITY A

**WHERENOT EXISTS** (**SELECT**\* **FROM** UNIVERSITY В

**WHERE** A.RATING > = В.RAT ING AND B .CITY = ' Москва ');

может быть представлен и в следующем виде

**SELECT** \* FROM UNIVERSITY A

**WHERE** 1 > (**SELECT**COUNT(\*) **FROM** UNIVERSITY В

**WHERE** A.RATING > = В.RATING **AND** B.CITY= ' Москва ');

Более подробная информация: <https://www.postgresql.org/docs/9.5/sql.html>

**5. Оборудование**

Персональный компьютер с установленной операционной системой WindowsXP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБДPostgreSQL.

**6. Задание на работу**

1. Напишите пример запроса, в котором вместо любой из констант выражения, определяющего условие WHERE, используется скалярный подзапрос.

2. Напишите запрос с векторным подзапросом (ключевое слово IN)

3. Напишите два запроса с ключевыми словами ANY и ALL, делающие одно и то же.

4. Напишите пример запроса с табличным подзапросом (ключевое слово EXISTS).

5. Напишите пример запроса несколькими уровнями вложенности

6. Напишите пример запроса, в котором вместо таблицы, указываемой после ключевого слова FROM, используется подзапрос.

7. Напишите пример связанного подзапроса.

Указания к выполнению заданий 1-5.

а) сформулируйте смысл запроса на языке, понятном пользователю в данной предметной области, например

б) приведите сам запрос

в) приведите результат выполнения запроса. Данные должны быть подобраны таким образом, чтобы в результирующей выборке было 3-4 записи.

**7. Контрольные вопросы**

1. Даны таблицы

|  |  |
| --- | --- |
| CREATE TABLE Рейс  (Номер\_рейса INT,  Конечный\_пункт VARCHAR(30),  Дата\_вылета DATETIME,  Продолжительность\_маршрута INT,  Число\_билетов INT,  Стоимость MONEY) | CREATE TABLE БИЛЕТ  (Номер\_места CHAR(3),  Номер\_рейса CHAR(6),  Дата\_продажи DATETIME,  Стоимость MONEY,  Фамилия\_пассажира VARCHAR(20)) |

Вывести список пассажиров, не летающих в Самару.

1. Даны таблицы

|  |  |
| --- | --- |
| CREATE TABLE Автор  (Код\_Автора INT ,  Фамилия VARCHAR(50) NULL,  Имя VARCHAR(50) NULL,  Отчество VARCHAR(50) NULL,  Пол VARCHAR(50) NOT NULL ,  Дата\_рождения DATETIME ,  Телефон CHAR(9)) | CREATE TABLE Книга  ( Код\_Книги INT,  Название VARCHAR(50) NOT NULL,  Цена MONEY,  Тематика VARCHAR(50) NOT NULL,  Издательство VARCHAR(50) NOT NULL,  Код\_Автора INT NOT NULL,  Количество INT) |

Определить авторов, не печатающих свои книги в издательстве <АСТ>.

1. Даны таблицы

|  |  |
| --- | --- |
| CREATE TABLE Город  ( Код\_Города INT ,  Название VARCHAR(20) NOT NULL,  Тариф MONEY,  Регион VARCHAR(20)) | CREATE TABLE Разговор  (Код\_Разговора INT,  Код\_Города INT NOT NULL,  Фамилия IVARCHAR(20),  Дата DATETIME NOT NULL,  Продолжительность INT NOT NULL) |

Вывести список городов, телефонные тарифы которых выше среднего.

1. Даны таблицы

|  |  |
| --- | --- |
| CREATE TABLE Блюдо  (Название\_блюдаVARCHAR(20) NOTNULL,  Время\_приготовления INT NOT NULL,  Общая\_калорийность INT NOT NULL,  Номер\_рецепта INT,  Повар VARCHAR(20),  Стоимость MONEY ) | CREATE TABLE Компонент  (Название\_компонента VARCHAR(20),  КалорийностьINTNOTNULL,  ВесFLOAT,  БелкиINT,  БлюдоVARCHAR(20),  Углеводы INT,  Стоимость MONEY NOT NULL) |

Определить блюдо, которое можно приготовить быстрее всех остальных блюд.

1. Даны таблицы

|  |  |
| --- | --- |
| CREATE TABLE Рейс  (Номер\_рейса INT,  Конечный\_пункт VARCHAR(30),  Дата\_вылета DATETIME,  Продолжительность\_маршрута INT,  Число\_билетов INT,  Стоимость MONEY) | CREATE TABLE БИЛЕТ  (Номер\_места CHAR(3),  Номер\_рейса INT,  Дата\_продажи DATETIME,  Стоимость MONEY,  Фамилия\_пассажира VARCHAR(20)) |

Определить список пассажиров, покупающих билеты на самые дальние рейсы.

**Лабораторная работа №9**

**Представления**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение и практическое представлений.

**2. Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

**3. Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

До сих пор речь шла о таблицах, обычно называемых базовыми таблицами. Это - таблицы, которые содержат данные. Однако имеется и другой вид таблиц, называемый **VIEW** или **ПРЕДСТАВЛЕНИЯ**. Таблицы-представления не содержат никаких собственных данных. Фактически *представление* — *это именованная таблица, содержимое которой является результатом запроса, заданного опри описании представления*. Причем данный запрос выполняется всякий раз, когда таблица-представление становится объектом команды SQL. Вывод запроса при этом в каждый момент становится содержанием представления. Представления позволяют:

• ограничивать число столбцов, из которых пользователь выбирает или в которые вводит данные;

• ограничивать число строк, из которых пользователь выбирает или в которые вводит данные;

• выводить дополнительные столбцы, преобразованные из других столбцов базовой таблицы;

• выводить группы строк таблицы.

Благодаря этому представления дают возможность гибкой настройки выводимой из таблиц информации в соответствии с требованиями конкретных пользователей, позволяют обеспечивать защиту информации на уровне строк и столбцов, упрощают формирование сложных отчетов и выходных форм.

Представление определяется с помощью команды **CREATE VIEW** (**СОЗДАТЬ ПРЕДСТАВЛЕНИЕ**). Например:

**CREATE VIEW** MOSCSTUD AS

**SELECT** \* **FROM** STUDENT **WHERE** CITY = ' Москва ';

Данные из базовой таблицы, предъявляемые пользователю в представлении, зависят от условия (предиката), описанного в **SELECT** -запросе при определении представления.

В созданную в результате приведенного выше запроса таблицу- представление MOSC \_ STUD передаются данные из базовой таблицы STUDENT, но не все, а только записи о студентах, для которых значение поля CITY равно 'Москва'. К таблице MOSC \_ STUD можно теперь обращаться с помощью запросов так же, как и к любой другой таблице базы данных. Например, запрос для просмотра представления MOSC \_ STUD имеет вид:

**SELECT** \* **FROM** MOSC \_ STUD;

Различают представления таблиц и представления столбцов. В простейшем представлении таблиц выбираются все строки и столбцы базовой таблицы.

**CREATE VIEW** NEW\_STUD\_TAB AS **SELECT** \* **FROM** STUDENT;

Такое представление, по сути, эквивалентно применению синонима, но является менее эффективным, поэтому применяется редко.

В простейшем виде представление столбцов выбирает все строки и столбцы, подобно представлению таблиц; кроме того, в качестве имен столбцов применяются псевдонимы. Например:

**CREATE VIEW** NEW\_STUDENT

(NEW\_STUDENT\_ID,NEW\_SURNAME,

NEW\_NAME, NEW\_STIPEND, NEW\_KURS,NEW\_CITY,

NEW\_BIRTHDAY, NEW\_UNIV\_ID)

AS **SELECT** STUDENT\_ID, SURNAME, NAME,STIPEND, KURS,

CITY, BIRTHDAY, UNIV\_ID **FROM** STUDENT;

Представление столбцов является простым способом организации общей таблицы для группы пользователей или прикладных задач, которые используют собственные имена полей и таблицы.

Данные, предъявляемые пользователю через представление, могут изменяться с помощью команд модификации DML, но при этом фактическая модификация данных будет осуществляться не в самой виртуальной таблице- представлении, а будет перенаправлена к соответствующей базовой таблице. Например, запрос на обновление представления NEW \_ STUDENT

**UPDATE** NEW \_ STUDENT **SET**CITY =' Москва '

**WHERE** STUDENT \_ ID = 1004;

эквивалентен выполнению команды **UPDATE** над базовой таблицей STUDENT.

Следует, однако, обратить внимание на то, что в общем случае, из-за того, что обычно в представлении отображаются данные из базовой таблицы в преобразованном или усеченном виде, применение команд модификации к таблицам-представлениям имеет некоторые особенности, рассматриваемые ниже.

Данный вид представлений ограничивает число столбцов базовой таблицы, к которым возможен доступ. Например, представление

**CREATE VIEW** STUD AS

**SELECT** STUDENT\_ID, SURNAME, CITY **FROM** STUDENT;

дает доступ пользователю к полям STUDENT \_ ID, SURNAME , CITY базовой таблицы STUDENT , полностью скрывая от него как содержимое, так и сам факт наличия в базовой таблице полей NAME , STIPEND , KURS , BIRTHDAY и UNIV \_ ID .

Представления, как уже отмечалось выше, могут изменяться с помощью команд модификации DML, но при этом модификация данных будет осуществляться не в самой таблице-представлении, а в соответствующей базовой таблице. В связи с этим, с представлениями, маскирующими столбцы, функции вставки и удаления работают несколько иначе, чем с обычными таблицами. Оператор **INSERT**, примененный к представлению, фактически осуществляет вставку строки в соответствующую базовую таблицу, причем во все столбцы этой таблицы независимо от того, видны они пользователю через представление или скрыты от него. В связи с этим, в столбцах, не включенных в представление, устанавливается **NULL** – значение или значение по умолчанию. Если не включенный в представление столбец имеет опцию **NOT NULL**, то генерируется сообщение об ошибке.

Любое применение оператора **DELETE** удаляет строки базовой таблицы независимо от их значений.

Представления могут также ограничивать доступ к строкам. Охватываемые представлением строки базовой таблицы задаются условием(предикатом) в конструкции **WHERE** при описании представления. Доступ через представление возможен только к строкам, удовлетворяющим условию. Например, представление

**CREATEVIEW** MOSCSTUD AS

**SELECT** \* **FROM** STUDENT **WHERE** CITY = ' Москва ';

Показывает пользователю только те строки таблицы STUDENT, для которых значение поля CITY равно 'Москва'.

Каждая включенная в представление строка доступна для вывода, обновления и удаления. Любая допустимая для основной таблицы строка вставляется в базовую таблицу независимо от ее включения в представление. При этом может возникнуть проблема, состоящая в том, что значения, введенные пользователем в базовую таблицу через представление, значений, будут отсутствовать в представлении, оставаясь при этом в базовой таблице. Рассмотрим такой случай:

**CREATEVIEW** HIGH\_RATING AS

**SELECT** \* **FROM** UNIVERSITY **WHERE** RAT ING = 300;

Это представление является обновляемым. Оно просто ограничивает доступ пользователя к определенным столбцам и строкам в таблице UNIVERSITY. Предположим, необходимо вставить с помощью команды **INSERT** следующую строку:

**INSERT INTO** HIGH \_ RATING

**VALUES** (180,'Новый университет', 200, 'Воронеж');

Команда **INSERT** допустима в этом представлении. С помощью представления HIGH \_ RATING строка будет вставлена в базовую таблицу UNIVERSITY. Однако, после появления этой строки в базовой таблице, из самого представления она исчезнет, поскольку значение поля RATING неравно 300, и, следовательно, эта строка не удовлетворяет условию предложения **WHERE** для отбора строк в представление. Для пользователя такое исчезновение только что введенной строки является неожиданным. Действительно, не понятно, почему после ввода строки в таблицу ее нельзя увидеть и, например, тут же удалить. Тем более, что пользователь вообще может не знать - работает он в данный момент с базовой таблицей или с таблицей-представлением.

Аналогичная ситуация возникнет, если в какой-либо существующей записи представления HIGH \_ RATING изменить значение поля RATING назначение, отличное от 300.

Подобные проблемы можно устранить путем включения в определение представления опции **WITH CHECK OPTION**. Эта опция распространяет условие **WHERE** для запроса на операции обновления и вставки в описание представления. Например:

**CREATE VIEW** HI GH\_RATUNG AS

**SELECT** \* **FROM** UNIVERSITY

**WHERE** RATING = 300 **WITH CHECK OPTION**;

В этом случае вышеупомянутые операции вставки строки или коррекции поля RATING будет отклонены.

Опция **WITH CHECK OPTION** помещается в определение представления, а не в команду **DML**, так что все команды модификации в представлении будут проверяться. Рекомендуется использовать эту опцию во всех случаях, когда нет причины разрешать представлению помещать в таблицу значения, которые в нем самом не могут быть видны.

Рассмотренная выше проблема возникает и при вставке строк в представление с предикатом, использующим поля базовой таблицы, не присутствующие в самом представлении. Например, рассмотримпредставление:

**CREATE VIEW** MOSCSTUD AS

**SELECT** STUDENT\_ID, SURNAME, STIPEND

**FROM** STUDENT **WHERE** CITY = ' Москва ';

Видно, что в данное представление не включено поле CITY таблицы STUDENT.

Что будет происходить при попытках вставки строки в это представление? Так как мы не можем указать значение CITY в представлении как значение по умолчанию (ввиду отсутствия в нем этого поля), то этим значением будет **NULL**, и оно будет введено в поле CITY базовой таблицы STUDENT (считаем, что для этого поля опция **NOT NULL** не используется). Так как в этом случае значение поля CITY базовой таблицы STUDENT не будет равняться значению 'Москва', вставляемая строка будет исключена из самого представления и, поэтому, не будет видна пользователю. Причем так будет происходить для любой вставляемой в представление MOSC \_ STUD строки. Другими словами, пользователь вообще не сможет видеть строки, вводимые им в это представление. Данная проблема не решается и в случае, если в определение представления будет добавлена опция **WITH CHECK OPTION**

**CREATE VIEW** MOSCSTUD AS

**SELECT** STUDENT\_ID,SURNAME, STIPEND

**FROM** STUDENT **WHERE** CITY = ' Москва ' **WITH CHECK OPTION**;

Таким образом, в определенном указанными способами представлении, можно модифицировать значения полей или удалять строки, но нельзя вставлять строки. Исходя из этого, рекомендуется даже в тех случаях, когда этого не требуется по соображениям полезности (и даже безопасности) информации, при определении представления включать в него все поля, на которые имеется ссылка в предикате. Если эти поля не должны отображаться в выводе таблицы, всегда можно исключить их уже в запросе к представлению. Другими словами, можно было бы определить представление MOSC \_ STUD подобно следующему:

**CREATEVIEW** MOSCSTUD AS

**SELECT** \* **FROM** STUDENT **WHERE** CITY= ' Москва ' **WITH CHECK OPTION**;

Эта команда заполнит в представлении поле CITY одинаковыми значениями, которые можно просто исключить из вывода с помощью другого запроса уже к этому сформированному представлению, указав в запросе только поля, необходимые для вывода.

**SELECT** STUDENT\_ID,SURNAME, STIPEND **FROM** M0SC\_STUD;

Создание представлений с использованием агрегированных функций и предложения **GROUP BY** является удобным инструментом для непрерывной обработки и интерпретации извлекаемой информации. Предположим, необходимо следить за количеством студентов, сдающих экзамены, количеством сданных экзаменов, количеством сданных предметов, средним баллом по каждому предмету. Для этого можно сформировать следующее представление

**CREATE VIEW** TOTALDAY AS

**SELECT** EXAM\_DATE, COUNT(DISTINCT SUBJ\_ID) AS

SUBJ\_CNT, COUNT(STUDENT\_ID) AS STUD\_CNT,

COUNT(MARK) ASMARK\_CNT, AVG(MARK) AS MARK\_AVG, SUM(MARK) AS MARK\_SUM

**FROM** EXAM\_MARKS **GROUPBY** EXAM\_DATE;

Теперь требуемую информацию можно увидеть с помощью простого запроса к представлению:

**SELECT** \* **FROM** TOTALDAY;

нескольких таблиц (базовых и/или других представлений) в одну большую виртуальную таблицу.

Такое решение имеет ряд преимуществ:

• представление, объединяющее несколько таблиц, может использоваться как промежуточный макет при формировании сложных отчетов, скрывающий детали объединения большого количества исходных таблиц.

• предварительно объединенные поисковые и базовые таблицы обеспечивают наилучшие условия для транзакций, позволяют использовать компактные схемы кодов, устраняя необходимость написания для каждого отчета длинных объединяющих процедур.

• позволяет использовать при формировании отчетов более надежный модульный подход.

• предварительно объединенные и проверенные представления уменьшают вероятность ошибок, связанных с неполным выполнением условий объединения.

Можно, например, создать представление, которое показывает имена и названия сданных предметов для каждого студента:

**CREATE VIEW** STUDSUBJ AS

**SELECT** A.STUDENT\_ID, C.SUBJ\_ID, A.SURNAME, C.SUBJ\_NAME

**FROM** STUDENT A, EXAM\_MARKS B, SUBJECT С

**WHERE** A.STUDENT\_ID = B.STUDENT\_ID

**AND** B.SUBJ\_ID = C.SUBJ\_ID;

Теперь все предметы студента или всех студентов для каждого предмета можно выбрать с помощью простого запроса. Например, чтобы увидеть все предметы, сданные студентом Ивановым, подается запрос: SELECT SUBJ \_ NAME FROMSTUD \_ SUBJ WHERE SURNAME = 'Иванов';

При создании представлений могут также использоваться и подзапросы, включая связанные подзапросы. Предположим, предусматривается премия для тех студентов, которые имеют самый высокий балл на любую заданную дату. Получить такую информацию можно с помощью представления:

**CREATE VIEW** ELITE \_ STUD AS

**SELECT** B.EXAM\_DATE, A.STUDENT\_ID, A. SURNAME

**FROM** STUDENT A, EXAM\_MARKS В

**WHERE** ASTUDENT\_ID = B.STUDENT\_ID

**AND** B.MARK = (SELECT MAX(MARK)

**FROM** EXAM\_MARKS С

**WHERE** C.EXAM\_DATE = B.EXAM\_DATE);

Если, с другой стороны, премия будет назначаться только студенту, который имел самый высокий балл и не меньше 10-ти раз, то необходимо использовать другое представление, основанное на первом:

**CREATE VIEW** BONUS AS

**SELECT** DISTINCTSTUDENT\_ID, SURNAME

**FROM** ELITE\_STUDA **WHERE** 10 < = (**SELECT** COUNT (\*)

**FROM** ELITE\_STUD В**WHERE** ASTUDENT\_ID =B.STUDENT\_ID);

Извлечение из этой таблицы записей о студентах, которые будут получать премию, выполняется простым запросом:

**SELECT** \* **FROM** BONUS;

Имеются некоторые виды запросов, не допустимые в определениях представлений. Одиночное представление должно основываться на одиночном запросе, поэтому **UNION** и **UNION ALL** в представлениях не разрешаются. Предложение **ORDER BY** также никогда не используется в определении представлений. Представление является реляционной таблицей-отношением, поэтому его строки по определению являются неупорядоченными.

Синтаксис удаления представления из базы данных подобен синтаксису удаления базовых таблиц:

DROP VIEW < имя представления >

Как уже говорилось, использование команд модификации языка SQL - INSERT (ВСТАВИТЬ), UPDATE (ЗАМЕНИТЬ), и DELETE (УДАЛИТЬ)- применительно для представлений имеет ряд особенностей. В дополнение к аспектам, рассмотренным выше, следует отметить, что не все представления могут модифицироваться.

Если команды модификации могут выполняться в представлении, то представление является обновляемым (модифицируемым); в противном случае оно предназначено только для чтения при запросе. Каким образом можно определить, является ли представление модифицируемым? Критерии обновляемости представления можно сформулировать следующим образом.

• Представление строится на основе одной и только одной базовой таблицы.

• Представление должно содержать первичный ключ базовой таблицы.

• Представление недолжно иметь никаких полей, которые представляют собой агрегирующие функции.

• Представление недолжно содержать DISTINCT в своем определении.

• Представление недолжно использовать GROUP BY или HAVING в своем определении.

• Представление недолжно использовать подзапросы.

• Представление может быть использовано в другом представлении, но это представление должно быть также модифицируемыми.

• Представление не должно использовать в качестве полей вывода константы или выражения значений.

Суть этих ограничений в том, что обновляемые представления фактически подобны окнам в базовых таблицах. Они показывают информацию из базовой таблицы, ограничивая определенные ее строки (использованием соответствующих предикатов) или специально именованные столбцы (с исключениями). Но при этом представления выводят значения без их обработки с использованием агрегирующих функций и группировки. Они также не сравнивают строки таблиц друг с другом (как это имеет место в объединениях и подзапросах, или при использовании DISTINCT).

Различия между модифицируемыми(обновляемыми) представлениями и представлениями "только для чтения' не случайны. Обновляемые представления в основном используются аналогично базовым таблицам. Пользователи могут даже не знать, является ли запрашиваемый ими объект базовой таблицей или представлением. Это превосходный механизм защиты для скрытия частей таблицы, которые являются конфиденциальными или не предназначены данному пользователю.

Не модифицируемые представления, с другой стороны, позволяют более рационально получать и переформатировать данные. С их помощью формируются библиотеки сложных запросов, которые могут затем использоваться в запросах для получения информации самостоятельно (например, в объединениях). Эти представления могут также иметь значение при решении задач защиты и безопасности данных. Например, можно предоставить некоторым пользователям возможность получения агрегатных данных (таких, как усредненное значение оценки студента), не показывая конкретных значений оценок и, тем более, не позволяя их модифицировать.

Относительно использования предложения **WITH CHECK OPTION** следует отметить, что в стандарте SQL это предложение не предусматривает каскадного изменения, то есть оно применяется только в представлениях, в которых оно определено, но не распространяется на другие представления, основанные на этом представлении. Например, в предыдущем примере

**CREATE VIEW** HIGH\_RATING AS

**SELECT** UNIV\_ID, RATING **FROM** UNIVERSITY

**WHERE** RAT ING >=400 **WITH CHECK OPTION**;

Попытка вставить или обновить значения поля RATING, отличные от 400, будет отвергнута, поскольку присутствует указание **WITH CHECK OPTION**. Однако, если создается второе представление (с тем же содержанием), основанное на первом:

**CREATEVIEW** MYRATING AS

**SELECT** \* **FROM** HIGH\_RATING;

То ввод в поле RATING с помощью нижеприведенного запроса значений, отличающихся от 400, уже не будет отвергнуто как ошибочное. То есть следующий запрос

**UPDATE** MYRATING **SET** RATING = 200 **WHERE** UNIV\_ID = 18;

не будет отвергнут как не корректный, и, после его выполнения, строки с обновленными данными исчезнут из как из представления MYRATING, так и из представления HIGH \_ RATING .

Предложение **WITH CHECK OPTION** просто гарантирует, что любое обновление в представлении осуществляется в соответствии со значениями, указанными именно для этого представления. Обновление других представлений, базирующихся на первом текущем, при этом допустимым, если эти представления не защищены предложениями **WITH CHECK OPTION**, заданными именно для них. Предложения **WITH CHECK OPTION** проверяют предикаты только того представления, в котором они содержатся. Приэто мне является выходом из положения и создание представления MYRATING с помощью запроса

**CREATE VIEW** MYRATING AS

**SELECT** \* **FROM** HIGH\_RATING **WITH CHECK OPTION**;

Более подробная информация: <https://www.postgresql.org/docs/9.5/sql.html>

**5. Оборудование**

Персональный компьютер с установленной операционной системой WindowsXP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБДPostgreSQL.

**6. Задание на работу**

Часть А

1. Создайте три обновляемых представления (по одному представлению для каждой из ваших сущностей).

2. Дайте для каждого представления понятную формулировку его смысла.

3. Проверьте, что эти представления обновляемые.

Часть Б

1. Создайте пять не обновляемых представлений. Каждое представление должно быть не обновляемым по какой-то одной из следующих причин:

• Представление строится на основе нескольких таблиц.

• Представление не содержит первичного ключа базовой таблицы.

• Представление содержит агрегирующие функции.

• Представление содержит DISTINCT в своем определении.

• Представление использует подзапросы.

2. Дайте для каждого представления понятную формулировку его смысла.

3. Проверьте, что все эти представления не обновляются.

Часть В

1. Создайте материализованное представление.

2. Измените данные в исходной таблице.

3. Добейтесь, чтобы данные изменились также и в материализованном представлении.

**7. Контрольные вопросы**

1. Дано представление.

*CREATE VIEW view4*

*AS*

*SELECT Книга.Название, Книга.Издательство,*

*Автор.Фамилия*

*FROM Автор INNERJOIN Книга ON*

*Автор.Код\_Автора = Книга.Код\_Автора*

Предпринимается попытка удалить из представления запись.

*DELETE FROM view4*

*WHERE Фамилия='Л.Толстой'*

Какой будет результат выполнения этой команды?

1. Дано представление.

*CREATE VIEW view5*

*AS*

*SELECT Билет.Номер\_места, Билет.Номер\_рейса,*

*Рейс.Конечный\_пункт*

*FROM Билет INNER JOIN Рейс ON*

*Билет.Номер\_рейса = Рейс.Номер\_рейса*

Предпринимается попытка удалить из представления запись.

*DELETE FROM view5*

*WHERE Конечный\_пункт ='Москва'*

Какой будет результат выполнения этой команды?

1. Дано представление.

*CREATE VIEW view3*

*AS*

*SELECT Название, Тариф, Тариф\*0.05 AS Налог*

*FROM Город*

*WHERE Регион='Поволжье'*

Предпринимается попытка удалить из представления запись.

*UPDATE view3 SET Тариф=Тариф\*1.5*

*WHERE Тариф>10*

Какой будет результат выполнения этой команды?

1. Дано представление.

*CREATE VIEW view4*

*AS*

*SELECT Книга.Название, Книга.Издательство,*

*Автор.Фамилия*

*FROM Автор INNERJOIN Книга ON*

*Автор.Код\_Автора = Книга.Код\_Автора*

Предпринимается попытка удалить из представления запись.

*DELETE FROM view4*

*WHERE Фамилия='Л.Толстой'*

Какой будет результат выполнения этой команды?

1. Дано представление.

*CREATE VIEW view1*

*AS*

*SELECT Название, Цена, Цена\*Количество AS Стоимость*

*FROM Книга*

*WHERE Издательство='Мир'*

Предпринимается попытка удалить из представления запись.

*UPDATE view1 SET Цена=Цена\*1.5*

*WHERE Цена>50*

Какой будет результат выполнения этой команды?

1. Дано представление.

*CREATE VIEW*

*AS*

*SELECT Город.Название, Разговор.Фамилия,*

*Разговор.Продолжительность*

*FROM Город INNER JOIN Разговор ON*

*Город.Код\_Города = Разговор.Код\_Города;*

Предпринимается попытка удалить из представления запись.

*DELETE FROM VIEW*

*WHERE Название='Самара'*

Какой будет результат выполнения этой команды?

1. Дано представление.

*CREATE VIEW view4*

*CREATE VIEW view4*

*AS*

*SELECT Регион, Avg(Тариф) AS Средн\_Тариф*

*FROM Город*

*GROUP BY Регион*

*HAVING Регион='Поволжье'*

Предпринимается попытка удалить из представления запись.

*DELETE FROM view4*

*WHERE Регион='Урал'*

Какой будет результат выполнения этой команды?

**Лабораторная работа №10**

**Триггеры**

#### 1. Цель и задачи работы

Целью лабораторной работы является изучение и практическое применение триггеров.

#### 2.Порядок выполнения работы

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

#### 3. Оформление отчета

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

#### 4. Теоретические сведения

Триггер базы данных соотносится с таблицей, представлением, схемой или базой данных. Триггер представляет собой процедуру, которая срабатывает автоматически, когда с определенным объектом происходит определенное событие. Рассмотрим триггеры, присоединенные к таблицам, которые возбуждаются при выполнении команд INSERT, DELETE или UPDATE. Триггер компилируется и записывается в словарь данных. Триггеры позволяют реализовать сложный алгоритм для проверки вводимых или редактируемых данных, выполнять определенные действия, сопутствующие указанной команде. Например, при продаже определенного товара по какой-то накладной, количество данного товара на определенном складе должно быть уменьшено на проданное количество. Триггеры особенно полезны, когда потребуется предотвратить появление неверных записей в таблицах, выполнить проверку допустимости значений столбца или обеспечить безопасность доступа. Другой областью применения триггеров является отслеживание модификаций таблиц базы данных. При любом изменении таблицы можно сохранять информацию о том, кто его выполнил, в какой момент времени, тип модификации и другие данные в специальной контрольной таблице.

Триггеры могут срабатывать до, после или вместо операции DML Триггер, разработанный для срабатывания до наступления события, может собрать данные о строке до начала модификации, триггер, срабатывающий после события, имеет доступ к новым значениям строки, а триггер типа «INSTEAD OF» полезен, если требуется запретить какое-то действие по модификации таблицы.

Триггер может быть создан таким образом, что он будет запускаться каждый раз для каждой строки, участвующей в модификации. Такой триггер называется обработчиком событий на уровне строк. При этом в команде триггера должна присутствовать опция «FOR EACH ROW». Если указанная фраза отсутствует, это означает, что триггер сработает один раз. Такой триггер называется обработчиком событий на уровне выражений.

#### *Команда создания триггера*

Общий вид команды создания триггера:

CREATE TRIGGER имя\_триггера

BEFORE| AFTER |INSTEAD OF

DELETE [OR INSERT [OR UPDATE [OF списокстолбцов]]] ON

имя\_таблицы

[FOR EACH ROW] [ WHEN (условие)]

[ DECLARE /\* переменные, константы, курсоры и т.п. \*/]

BEGIN

/\* блок PL/SQL \*/END;

Предложение триггера определяет:

1-тип команды SQL, при выполнении которой запускается триггер (DELETE, INSERT или UPDATE). В спецификацию предложения триггера могут быть включены одна, две или все три эти команды.

#### *Список столбцов для UPDATE*

Если триггер должен реагировать на команду UPDATE, то в спецификацию триггера может быть включен необязательный список столбцов. При наличии списка столбцов данный триггер запускается по команде UPDATE лишь тогда, когда она обновляет один из перечисленных столбцов. Если список столбцов отсутствует, то триггер возбуждается при обновлении любого столбца ассоциированной таблицы. Список столбцов не может быть специфицирован для предложений триггера INSERT или DELETE

#### *Опция WHEN*

В определение триггера строки может быть включено необязательное ограничение триггера с помощью условия, указанного в опции WHEN. Выражение в опции WHEN, если она присутствует, вычисляется для каждой строки, затрагиваемой триггером. Если результат выражения для строки дает TRUE, то тело триггера исполняется для этой строки. Однако если это выражение вычисляется для строки как FALSE или NULL, то тело триггера не исполняется для этой строки.

#### *Тело триггера*

Тело триггера - это блок PL/SQL, который может содержать предложения SQL и PL/SQL. Эти предложения исполняются тогда, когда сработала соответствующая команда, запускающая триггер, и ограничение триггера (если оно есть) вычислено как TRUE. Для триггеров строк тело триггера имеет некоторые специальные конструкции, которые могут быть включены в код этого блока PL/SQL: корреляционные имена, условные предикаты INSERTING, DELETING и UPDATING.

#### *Доступ к значениям столбцов в триггерах строки*

Внутри тела триггера строк, код PL/SQL и предложения SQL имеют доступ как к старым, так и к новым значениям столбцов текущей строки, затрагиваемой предложением триггера. Для каждого столбца модифицируемой таблицы определены два **корреляционных имени**: одно для старого(old), другое – для нового значения столбца (new). В зависимости от типа предложения триггера, то или иное корреляционное имя может быть лишено смысла.

#### *Условные предикаты*

Если триггер может быть возбужден более чем одним типом предложения DML (например, "INSERT OR DELETE OR UPDATE OF fio"), то в теле триггера можно использовать условные предикаты INSERTING, DELETING и UPDATING, для того чтобы выполнять различные участки кода в зависимости от типа команды, запустившей триггер. Предположим, что предложение триггера определено следующим образом:

INSERT OR UPDATE OF fio ON manager

***Создание и использование триггера BEFORE***

Для таблиц, в которых используются последовательности для создания полей-счетчиков, очень удобно использовать триггер типа BEFORE для команды INSERT. Данный триггер будет формировать поле-счетчик, а в команде добавления новой записи, это поле вообще не нужно будет указывать. Тем самым разработчик будет избавлен от необходимости вспоминать название последовательности каждый раз, как требуется добавить новую запись. Например, для таблицы manager мы создали последовательность man\_km, которую будем использовать в теле триггера. Кроме того, предусмотрим в триггере исправление возможной ошибки при вводе значения поля fio с маленькой буквы. Триггер, срабатывающий до выполнения команды INSERT, может иметь следующий вид:

CREATE OR REPLACE TRIGGER bi\_trg\_manager

BEFORE INSERT ON manager FOR EACH ROW

BEGIN

SELECT m\_km.nextvalINTO :new.kod\_men FROM DUAL;

:new.fio:=INITCAP(:new.fio);

END;

***Создание и использование триггера AFTER***

Триггер AFTER срабатывает после наступления события. При этом старое значение изменяемого поля можно получить из переменной с именем :old.имя\_поля, а новое – из переменной :new.имя\_поля. Обе переменные доступны при операции UPDATE, переменная :new.имя\_поля – при операции INSERT, а переменная :old.имя\_поля – при операции DELETE.

Триггер, который запускается после выполнения команды UPDATE, будет реагировать на попытку изменить любое поле таблицы manager. При изменении поля kod\_men, которое формируется с помощью последовательности и является первичным ключом, должно возбудиться исключение, которое выведет сообщение о том, что данное поле менять нельзя, и транзакция будет отменена. При любых других изменениях будет добавлена соответствующая запись в таблицу man\_hist.

CREATE OR REPLACE TRIGGER au\_trg\_manager

AFTER UPDATE ON manager

FOR EACH ROWDECLAREupd\_kod EXCEPTION;BEGIN

IF UPDATING('kod\_men') THEN

RAISE upd\_kod;

ELSE

INSERT INTO man\_hist VALUES (user, sysdate, :old.fio,:new.fio,:old.oklad,:new.oklad, :old.kod\_men);

END IF;

EXCEPTION

WHEN upd\_kod THEN

RAISE\_APPLICATION\_ERROR(-20005,'Код менеджера изменять нельзя');END;

***Ограничения при создании табличных триггеров***

Триггеры, разработанные для таблицы, не могут использовать следующие команды: COMMIT, ROLLBACK, SAVEPOINT, SET TRANSATION. Процедуры и функции, которые вызываются в триггере, также не должны содержать внутри себя указанные команды.

В триггерах запрещено объявлять переменные типа LONG или LONG ROW. Следующие ограничения связаны с понятиями «мутирующей таблицы» и таблицы с ограничениями целостности типа PRIMARY KEY, UNIQUE, FOREIGN KEY.

Мутирующей является таблица, которая в данный момент модифицируется одной из DML команд (INSERT, DELETE или UPDATE). Таблица, которая может модифицироваться за счет каскадного ограничения целостности типа DELETE CASCADE, также является мутирующей.

В триггере типа «FOR EACH ROW» нельзя ни читать, ни изменять данные мутирующей таблицы внутри триггера. Эта ошибка выявляется только во время выполнения триггера, а не на этапе компиляции. Исключением является только триггеры типа BEFORE INSERT и AFTER INSERT, которые позволяют считывать и изменять данные с помощью корреляционных переменных :new и :old при условии добавления единственной строки.

В теле триггера запрещено изменять любые ключи (primary, unique, foreign) таблиц, поддерживающих ограничения ссылочной целостности. Другие поля данных таблиц могут быть модифицированы триггером.

#### 5. Оборудование

персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

#### 6. Задание на работу

1. Создайте: триггер для INSERT, триггер для DELETE, триггер для UPDATE.

Один из них должен быть BEFORE, один – AFTER, один – INSTEAD OF. Два из них должны быть определены уровне строк (FOR EACH ROW), один – на уровне оператора.

Какой-то из триггеров должен проверять возможность действия (вызывать исключение в зависимости от определенных условий), какой-то – модифицировать ту же таблицу, для которой создается триггер, оставшийся должен быть определен ДЛЯ ПРЕДСТАВЛЕНИЯ и модифицировать таблицу, на базе которой построено это представление.

В каком-то из триггеров должна использоваться псевдострокаnew, в каком-то – псевдострокаold.

1. Приведите:

* исходное содержимое таблиц
* тексты триггеров
* формулировки того, что делает триггер
* операторы, проверяющие действия триггеров
* содержимое таблиц после этих операторов, с выделением (шрифтом или цветом) того, что изменилось.

#### 7. Контрольные вопросы

1. Дана таблица Книга. Разработать триггер, который выполняется вместо изменения цены одной книги в этой таблице. Изменение цены выполнить только в том случае, если книга издается в издательстве 'Мир'.

CREATE TRIGGER trig\_upd

ON Книга INSTEAD OF UPDATE

AS

Напишите операторы для продолжения текста триггера.

1. Даны таблицы Город и Разговор. Пусть они не связаны внешним ключом. Для обеспечения целостности данных создать триггер, обрабатывающий удаление информации о некотором городе из таблицы Город. При этом в таблице Разговор необходимо также удалить записи обо всех телефонных разговорах с удаленным городом.

CREATE TRIGGER trig\_del ON Город FOR DELETE

AS

Напишите операторы для продолжения текста триггера.

1. Даны таблицы Рейс и Билет. Пусть они не связаны внешним ключом. Для обеспечения целостности данных создать триггер, обрабатывающий удаление записи о некотором рейсе из таблицы Рейс. При этом в таблице Билет необходимо его номер заменить на номер запасного рейса 111 для билетов, проданных на удаленный рейс.

CREATE TRIGGER trig\_del ON Рейс FOR DELETE

AS

Напишите операторы для продолжения текста триггера.

1. Даны таблицы Книга и Автор. Пусть они не связаны внешним ключом. Для обеспечения целостности данных создать триггер, обрабатывающий удаление информации о некотором авторе из таблицы Автор. При этом необходимо запретить удаление, если общий тираж упомянутого автора больше 1000 экземпляров..

CREATE TRIGGER trig\_del ON Автор FOR DELETE

AS

Напишите операторы для продолжения текста триггера.

1. Дана таблица Разговор. Разработать триггер, который выполняется вместо изменения продолжительности одного разговора. Выполнить изменение продолжительности только в том случае, если общая продолжительность телефонных соединений абонента, в разговоре которого меняется продолжительность, не превышает 300 мин.

CREATE TRIGGER trig\_upd

ON Разговор INSTEAD OF UPDATE

AS

Напишите операторы для продолжения текста триггера.

1. Дана таблица Билет. Разработать триггер, который выполняется вместо изменения стоимости одного билета. Изменение стоимости билета выполнить только в том случае, если на этот рейс продано не менее 20 билетов.

CREATE TRIGGER trig\_upd

ON Билет INSTEAD OF UPDATE

AS

Напишите операторы для продолжения текста.

# Лабораторная работа №11

**Хранимые процедуры**

#### 1. Цель и задачи работы

Целью лабораторной работы является изучение и практическое применение хранимых процедур.

#### 2.Порядок выполнения работы

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

#### 3. Оформление отчета

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

#### 4. Теоретические сведения

PL/SQL – это язык, структурированный блоками. Это значит, что основные единицы (процедуры, функции и анонимные блоки), составляющие программу PL/SQL, являются логическими БЛОКАМИ, которые могут содержать любое число вложенных в них подблоков.

Структура блока имеет вид: [DECLARE -- описание переменных, констант и пользовательских типов данных] BEGIN -- операторы SQL -- управляющие операторы PL / SQL [EXCEPTION -- действия, выполняемые при возникновении ошибки] END; Программы PL/SQL могут быть неименованными (анонимными блоками), но чаще всего используются именованные программы: процедуры, функции, пакеты и триггеры.

**Циклы**

**Простой цикл.**Синтаксис:

LOOP*оператор 1*;. . .EXIT [WHEN*условие*];

END LOOP;

Простой цикл бесконечен. Завершить цикл можно оператором EXIT.**Цикл FOR.** Синтаксис:

FOR*индекс*IN[REVERSE]*нижняя\_граница.верхняя\_граница*LOOP*оператор 1*;

. . .

ENDLOOP;*индекс*– неявно описанная целая переменная, значение которой автоматически увеличивается или уменьшается на 1 при каждом выполнении цикла, пока не будет достигнута верхняя граница. Объявлять индекс не нужно!!! REVERSE – вызывает уменьшение индекса.

Внутри цикла FOR к индексу цикла можно обращаться как к константе. Поэтому индекс может встречаться в выражениях, но ему нельзя присваивать значений, как показывает следующий пример:

FOR ctr IN 1..10 LOOP

...

IF NOT finished THEN

INSERT INTO ... VALUES (ctr, ...); -- законноfactor := ctr \* 2; -- законно

...ELSEctr := 10; -- незаконно

END IF;

ENDLOOP;

**Цикл WHILE.** Цикл WHILE используется для повторения последовательности операторов в течение всего времени, пока значение условия равно TRUE. Синтаксис:

WHILE*условие*LOOP*оператор 1;*. . .ENDLOOP;Пример:WHILEtotal<= 25000 LOOP

...

SELECT sal INTO salary FROM empWHERE ...total := total + salary;

END LOOP;

Циклы могут быть вложены один в другой на несколько уровней.

Управление курсорами

***Явные курсоры***

Множество строк, возвращаемых запросом (активное множество), может состоять из нуля, одной или нескольких строк, в зависимости от того, сколько строк удовлетворяют вашим поисковым условиям. Когда запрос возвращает несколько строк, вы можете явно определить курсор для обработки этих строк.

Вы определяете курсор в декларативной части блока PL/SQL, подпрограммы или пакета путем задания его имени и специфицирования запроса. После этого вы манипулируете курсором при помощи трех команд: OPEN, FETCH и CLOSE.

Прежде всего вы инициализируете курсор предложением OPEN, которое идентифицирует активное множество. Затем с помощью предложения FETCH вы извлекаете первую строку. Вы можете повторять FETCH неоднократно, пока не будут извлечены все строки. После обработки последней строки вы освобождаете курсор предложением CLOSE.

Вы можете обрабатывать параллельно несколько запросов, объявив и открыв несколько курсоров.

#### *Объявление курсора*

Ссылки в перед недопустимы в PL/SQL. Поэтому вы должны объявить курсор, прежде чем сможете ссылаться на него в других предложениях. Объявляя курсор, вы даете ему имя и ассоциируете его с конкретным запросом. В следующем примере объявляется курсор с именем c1:

DECLARE

CURSORc 1ISSELECTename ,deptnoFROMempWHEREsal> 2000;

...

BEGIN...

Имя курсора - это необъявленный ранее идентификатор, а не переменная PL/SQL; его можно использовать только для обращения к запросу. Вы не можете присваивать значений имени курсора или использовать его в выражениях.

#### *Параметризованные курсоры*

Курсоры могут принимать параметры, как показывает следующий пример. Параметр курсора может появляться в запросе всюду, где допускается появление константы.

CURSOR c1 (median IN NUMBER) IS

SELECT job, ename FROM emp WHER sal> median;

Для объявления формальных параметров курсора используется синтаксис:

CURSOR имя[ (параметр [, параметр, ...]) ] IS

SELECT...

гдепараметр, в свою очередь, имеет следующий синтаксис:имя\_переменной [IN] тип\_данных [{:= | DEFAULT} значение]

Как показывает следующий пример, вы можете инициализировать параметры курсора умалчиваемыми значениями. Таким способом вы можете передавать курсору различное число фактических параметров, принимая или перекрывая умалчиваемые значения по своему желанию. Более того, вы можете добавлять в курсор новые формальные параметры без необходимости отыскивать и исправлять все обращения к курсору в тексте программы.

DECLARE

CURSOR c1

(low INTEGER DEFAULT 0,high INTEGER DEFAULT 99) IS SELECT ...

...

Сфера параметров курсора локальна в этом курсоре, что означает, что к этим параметрам можно обращаться лишь в запросе, который участвует в объявлении курсора. Значения параметров курсора используются ассоциированным запросом в момент открытия курсора.

Открытие курсора

Открытие курсора предложением OPEN исполняет предложение SELECT и идентифицирует АКТИВНОЕ МНОЖЕСТВО, т.е. все строки, удовлетворяющие поисковым условиям запроса. Для курсоров, объявленных с фразой FOR UPDATE, предложение OPEN также осуществляет блокировку этих строк. Пример предложения OPEN:

OPEN c1;

Предложение OPEN не извлекает строк активного множества. Для этого используется предложение FETCH.

Передача параметров

Курсору могут быть переданы параметры при открытии. Например, при объявлении курсора

CURSOR c1 (my\_ename CHAR, my\_commNUMBER) IS SELECT ...

любое из следующих предложений открывает этот курсор:

OPEN c1('ATTLEY', 300);

OPENc1(employee\_name, 150);

OPEN c1('THURSTON', my\_comm);

В последнем примере переменная, специфицированная в предложении OPEN, имеет такое же имя, что и параметр в объявлении курсора.

Когда идентификатор my\_comm используется в объявлении курсора, он обозначает формальный параметр курсора. Когда этот же идентификатор используется вне объявления курсора, он обозначает переменную PL/SQL с этим именем. Однако, для ясности, рекомендуется использовать уникальные идентификаторы.

Если вы не хотите принять умалчиваемые значения, каждому формальному параметру в объявлении курсора следует сопоставить соответствующий фактический параметр в предложении OPEN.

Формальные параметры, объявленные с умалчиваемым значением, могут и не иметь соответствующих им фактических параметров. В этом случае они просто принимают свое умалчиваемое значение во время выполнения OPEN. Не забывайте, что формальные параметры курсора должны быть параметрами IN, так что они не могут возвращать значений фактическим параметрам.

Извлечение данных из курсора

Предложение FETCH извлекает очередную строку из активного множества. При каждом выполнении FETCH курсор продвигается к следующей строке в активном множестве. Пример предложения FETCH:

FETCHc1 INTO my\_empno, my\_ename, my\_deptno;

Для каждого значения столбца, извлекаемого запросом, ассоциированного с курсором, в списке INTO должна быть соответствующая переменная, имеющая совместимый с этим столбцом тип данных. Обычно вы используете неиндивидуальные переменные, а запись:

...

OPEN c1;

LOOP

FETCH c1 INTO my\_record;

EXITWHENc 1%NOTFOUND ;

-- обработатьизвлеченные данные

END LOOP;

Любые переменные в фразе WHERE запроса, ассоциированного с курсором, вычисляются лишь в момент открытия курсора. Как показывает следующий пример, запрос может обращаться к переменным PL/SQL внутри своей сферы:

DECLARE

my\_salemp.sal%TYPE; my\_jobemp.job%TYPE; factor INTEGER := 2;

CURSOR c1 IS

SELECT factor\*sal FROM empWHERE job = my\_job; BEGIN

...

OPEN c1; -- здесь factor равен 2

LOOP

FETCH c1 INTO my\_sal;

EXIT WHEN c1%NOTFOUND;

...

factor := factor + 1; -- не окажет влияния на FETCH

END LOOP ;

END ;

В этом примере каждое извлекаемое значение sal будет умножаться на 2, несмотря на то, что значение factor увеличивается после каждой операции FETCH. Чтобы изменить активное множество или значения переменных в запросе, вы должны закрыть и заново открыть курсор с новыми значениями входных параметров.

Однако для каждой операции FETCH на одном и том же курсоре вы можете использовать собственный список INTO. Каждая FETCH извлекает строку и присваивает значения своим переменным INTO, как показывает следующий пример: DECLARE

CURSOR c1 IS SELECT ename FROM emp;name1 emp.ename%TYPE;name2 emp.ename%TYPE;name3 emp.ename%TYPE;

BEGIN

OPEN c1;

FETCH c1 INTO name1; --извлекаетпервуюстроку

FETCH c1 INTO name2; -- извлекает вторую строку

FETCH c1 INTO name3; -- извлекает третью строку

...

CLOSEc 1;

END ;

Если вы выдаете FETCH, но в активном множестве больше нет строк, то значения переменных в списке INTO не определены.

Закрытие курсора

Предложение CLOSE деактивирует курсор, и активное множество становится неопределенным. Пример предложения CLOSE:

CLOSE c1;

После того, как курсор закрыт, вы можете снова открыть его.

Атрибуты явного курсора

Каждый курсор, явно объявленный вами, имеет четыре атрибута: % NOTFOUND, % FOUND, % ROWCOUNT и % ISOPEN. Атрибуты позволяют вам получать полезную информацию о выполнении многострочного запроса. Для обращения к атрибуту просто присоедините его имя к имени курсора. Атрибуты явного курсора можно использовать в процедурных предложениях, но не в предложениях SQL.

Использование %NOTFOUND

Когда курсор открыт, строки, удовлетворяющие ассоциированному запросу, идентифицированы и образуют активное множество. Эти строки извлекаются операцией FETCH по одной за раз. Если последняя операция FETCH вернула строку, %NOTFOUND дает FALSE.

Если последняя операция FETCH не смогла вернуть строку (так как активное множество исчерпано), %NOTFOUND дает TRUE. Операция FETCH должна в конце концов исчерпать активное множество, так что, когда это происходит, никакого исключения не возбуждается.

В следующем примере вы используете %NOTFOUND, чтобы выйти из цикла, когда FETCH не сможет вернуть строку:

LOOP

FETCH c1 INTO my\_ename, my\_deptno;EXIT WHEN c1%NOTFOUND;

...

END LOOP;

Перед первой операцией FETCH атрибут % NOTFOUND дает NULL . Поэтому, если FETCH ни разу не выполнится успешно, вы никогда не выйдете из этого цикла. Причина в том, что предложение EXIT WHEN выполняется только в том случае, когда условие WHEN дает TRUE. Поэтому для безопасности вы можете предпочесть такой вариант предложения EXIT:

EXIT WHEN c1%NOTFOUND OR c1%NOTFOUNDIS NULL;

Вы можете открыть несколько курсоров, а затем использовать %NOTFOUND, чтобы проверять, в каких курсорах еще есть строки..

Использование %FOUND

%FOUND логически противоположен атрибуту %NOTFOUND. После открытия явного курсора, но до первой операции FETCH, %FOUND дает NULL. Впоследствии он дает TRUE, если последняя операция FETCH вернула строку, или FALSE, если последняя операция FETCH не смогла извлечь строку, так как больше нет доступных строк.

Следующий пример использует %FOUND, чтобы выбрать одно из двух альтернативных действий:

LOOP

FETCH c1 INTO my\_ename, my\_deptno;

IF c1%FOUND THEN --извлечение успешно

INSERT INTO temp VALUES (...);

ELSE

EXIT;

...

END LOOP;

Вы можете открыть несколько курсоров, а затем использовать %FOUND, чтобы проверять, в каких курсорах еще есть строки.

#### *Использование %ROWCOUNT*

Когда вы открываете курсор, его атрибут %ROWCOUNT обнуляется. Перед первой операцией FETCH %ROWCOUNT возвращает 0.Впоследствии, ROWCOUNT возвращает число строк, извлеченных операциями FETCH из активного множества на данный момент. Это число увеличивается, если последняя FETCH вернула строку.

Следующий пример использует %ROWCOUNT, чтобы предпринять определенные действия, если выбрано более10 строк:

LOOP

FETCH c1 INTO my\_ename, my\_deptno;

IFc 1%ROWCOUNT > 10THEN--выбрано больше 10 строк

...

END IF;

END LOOP;

Вы можете открыть несколько курсоров, а затем использовать

%ROWCOUNT, чтобы проверять, сколько строк извлечено из каждого курсора.

Использование %ISOPEN

%ISOPEN дает TRUE, если явный курсор открыт, и FALSE в противном случае. Следующий пример использует %ISOPEN для выбора действия:

IF с1%ISOPEN THEN -- курсор открыт

...

ELSE -- курсор закрыт, открыть его

OPENc 1;

ENDIF ;

Курсорные циклы FOR

Курсорный цикл FOR неявно объявляет свой индекс цикла как запись типа %ROWTYPE, открывает курсор, в цикле извлекает строки из активного множества в поля записи, и закрывает курсор, когда все строки обработаны или когда вы выходите из цикла.

В большинстве ситуаций, которые требуют явного курсора, текст программы может быть упрощен при использовании "курсора в цикле FOR", заменяющего команды OPEN, FETCH и CLOSE. Курсор в цикле FOR: - неявно объявляет индекс цикла записью, поля которой соответствуют столбцам (псевдонимам) предложения SELECT ... из описания курсора; - передает параметры курсора (если они есть) и открывает курсор; - выбирает в цикле строки из полученного набора в индекс цикла (поля записи); – закрывает курсор после обработки всех строк набора или досрочному выходу из него с помощью команд EXIT или GOTO. Синтаксис курсора в цикле FOR имеет вид:

FOR var\_rec\_name IN cursor\_name [(value [,value]...) ] LOOPТЕЛОЦИКЛАEND LOOP,

где – var\_rec\_name индекс цикла, в котором при первом прохождении цикла хранится первая строка набора, при втором прохождении цикла - вторая строка и т.д.; – список значений ("value") используется для передачи параметров курсора (он заменяет в данном случае список из команды OPEN); - ТЕЛО ЦИКЛА содержит нужные строки повторяющейся части программы, в которых используются переменные с именами var\_rec\_name.column\_name, а column\_name имя столбца из перечня столбцов предложения SELECT в описании курсора.

Рассмотрим следующий блок PL/SQL, который анализирует данные, собранные в ходе лабораторных экспериментов, и помещает результаты во временную таблицу. Переменная c1rec, используемая как индекс в курсорном цикле FOR, неявно объявляется как запись, хранящая все элементы данных, возвращаемые одной операцией FETCH для курсора c1. Вы обращаетесь к элементам данных, хранящимся в полях записи, используя квалифицированные ссылки.

DECLARE

result temp.col1%TYPE;

CURSOR c1 IS

SELECT n1, n2, n3 FROM data\_table

WHERE exper\_num = 1; BEGIN

FOR c1rec IN c1 LOOP

/\* вычислить и сохранить результаты \*/

result := c 1 rec . n 2 / ( c 1 rec . n 1 + c 1 rec . n 3);

INSERT INTO temp VALUES (result, NULL, NULL);

END LOOP ;

COMMIT ;

END;

То же самое без использования цикла FOR : DECLARE

num1 data\_table.n1%TYPE; -- Объявить переменные num2 data\_table.n2%TYPE; -- с теми же типами , num3 data\_table.n3%TYPE; -- что и столбцы таблицы

result temp.col1%TYPE;

CURSOR c1 IS

SELECT n1, n2, n3 FROMdata\_table

WHERE exper\_num = 1;

BEGIN

OPEN c1;

LOOP

FETCH c1 INTOnum1, num2, num3;

EXIT WHEN c1%NOTFOUND;

-- условиеc1%NOTFOUNDбудетравноTRUE,

-- когда FETCH необнаружит больше строк

/\* вычислить и сохранить результаты \*/ result := num 2/( num 1 + num 3);

INSERT INTO temp VALUES (result,NULL, NULL);

END LOOP;

CLOSE c1;

COMMIT;

END;

#### *Процедуры*

Процедура – это подпрограмма, которая выполняет специфическое действие. Вы пишете процедуры, используя синтаксис

PROCEDUREимя [ (параметр [, параметр, ...]) ] IS

[локальные объявления]BEGIN

исполняемые предложения

[EXCEPTIONобработчики исключений]END [имя];где каждый "параметр" имеет следующий синтаксис:

имя\_перем [IN | OUT | IN OUT] тип\_данных [{:= | DEFAULT} знач]

Каждый параметр сопровождается режимом и типом. В качестве режима можно задавать: IN (только чтение), OUT (только запись) и INOUT (чтение и запись). Замечание: в описании типа нельзя задавать длину. Например: CHAR(10)

или VARCHAR(20) не допустимые описания, а CHAR или VARCHAR – допустимые. Количество символов (длина) будет определена при вызове процедуры по фактическому параметру (аргументу) функции.

Процедура начинается с ключевых слов CREATE PROCEDURE, за которыми следует ее имя и список параметров. В качестве ключевого слова (описателя) вместо CREATE может использоваться ORREPLACE. Преимущество использования этого ключевого слова в том, что если процедура с каким-то именем уже определена, то новое определение с тем же именем не вызовет ошибки. С другой стороны, предыдущее определение процедуры с аналогичным именем заменится новым определением, и старая процедура перестанет существовать.

Процедура имеет две части: спецификацию и тело. Спецификация процедуры начинается с ключевого слова PROCEDURE и заканчивается именем процедуры или списком параметров. Объявления параметров необязательны. Если процедура не принимает параметров, скобки также не кодируются.

Тело процедуры начинается с ключевого слова IS и заканчивается ключевым словом END, за которым может следовать имя процедуры.

Тело процедуры состоит из трех частей: декларативной части, исполняемой части и необязательной части обработки исключений.

**Пример.**Процедура raise\_salary, которая увеличивает жалованье сотрудника:

CREAT OR REPLACE PROCEDUREraise\_salary (emp\_id NUMBER, increase REAL) IS current\_salary REAL; salary\_missing EXCEPTION;

BEGIN

SELECT sal INTO current\_salaryFROMemp

WHERE empno =emp\_id;

IF current\_salary IS NULL THEN

RAISE salary\_missing;

ELSE

UPDATE emp SET sal = sal +increase

WHERE empno = emp\_id;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

INSERT INTO emp\_auditVALUES(emp\_id, 'No such number'); WHEN salary\_missing THEN

INSERT INTO emp\_auditVALUES(emp\_id, 'Salary is null');

END raise \_ salary ;

#### *Функции*

Функция – это подпрограмма, которая вычисляет значение. Функции структурируются так же, как и процедуры, с той разницей, что функции содержат фразу RETURN. Вы пишете функции, используя синтаксис

FUNCTIONимя [ (аргумент [, аргумент, ...]) ]RETURN тип\_данныхIS[локальные\_объявления]BEGIN

исполняемые предложения

[EXCEPTIONобработчики исключений]END [имя]; где каждый "аргумент" имеет следующий синтаксис: имя\_перем [IN | OUT | IN OUT] тип\_данных [{:= | DEFAULT} знач]

В исполняемой части процедуры должно встретиться хотя бы одно предложение RETURN.

Процедура начинается с ключевых слов CREATE PROCEDURE, за которыми следует ее имя и список параметров. В качестве ключевого слова (описателя) вместо CREATE может использоваться ORREPLACE. Преимущество использования этого ключевого слова в том, что если процедура с каким-то именем уже определена, то новое определение с тем же именем не вызовет ошибки. С другой стороны, предыдущее определение процедуры с аналогичным именем заменится новым определением, и старая процедура перестанет существовать.

**Пример:**

функция balance возвращает баланс заданного бухгалтерского счета:

CREATE OR REPLACE FUNCTION balance(acct\_id NUMBER)RETURN REAL ISacct\_bal REAL;

BEGIN

SELECT bal INTO acct\_bal FROM acctsWHEREacctno = acct\_id;

RETURN acct\_bal;

END balance;

##### **5. Оборудование**

персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

##### **6. Задание на работу**

1. Создайте функцию, возвращающую количество записей в таблице по указанному значению внешнего ключа. Входной параметр - значение ключа.
2. Создайте процедуру, изменяющую поле строки. Входные параметры - первичный ключ строки и новое значение изменяемого поля.
3. Создайте новую таблицу TEMP для занесения временных данных. Напишите процедуру (используя явный курсор) для выборки первых пяти записей какой-либо таблицы, соответствующих определенному критерию, и занесения их в таблицу TEMP.
4. Приведите:

* смысл процедуры или функции на языке, понятном пользователю в данной предметной области.
* оператор, создающий данную процедуру или функцию
* состояние данных до выполнения процедуры или функции
* оператор, запускающий процедуру или функцию
* состояние данных после выполнения процедуры или функции

##### **7. Контрольные вопросы**

1. В базе данных basa1 имеется таблица Книга. Содержащуюся в ней информацию необходимо прочитать из базы данных basa2. В какой базе данных должна быть создана процедура, и как можно ее выполнить?
2. В базе данных basa1 имеется таблица Город. Из базы данных basa2 необходимо выполнить действие по увеличению тарифов для всех городов на 15%. В какой базе данных должна быть создана процедура и как можно ее выполнить?
3. В базе данных basa1 имеется таблица Блюдо. Из базы данных basa2 необходимо выполнить действие по увеличению стоимости всех блюд на 50%. В какой базе данных должна быть создана процедура и как можно ее выполнить?
4. В базе данных basa1 имеется таблица Рейс. Из базы данных basa2 необходимо выполнить действие по увеличению стоимости билетов на все рейсы на 25%. В какой базе данных должна быть создана процедура и как можно ее выполнить?
5. В базе данных basa1 имеется таблица Книга. Из базы данных basa2 необходимо выполнить действие по увеличению цены всех книг на 10%. В какой базе данных должна быть создана процедура и как можно ее выполнить?
6. В базе данных basa1 имеется таблица Рейс. Содержащуюся в ней информацию необходимо прочитать из базы данных basa2. В какой базе данных должна быть создана процедура, и как можно ее выполнить?

**Лабораторная работа №12**

**Индексирование**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение и практическое применение индексирования.

**2. Порядок выполнения работы**

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

**3. Оформление отчета**

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

Индексы – это традиционное средство увеличения производительности БД. Используя индекс, сервер баз данных может находить и извлекать нужные строки гораздо быстрее, чем без него. Однако с индексами связана дополнительная нагрузка на СУБД в целом, поэтому применять их следует обдуманно.

Предположим, что у нас есть такая таблица:

CREATE TABLE test1 (

id integer,

content varchar

);

и приложение выполняет много подобных запросов:

SELECT content FROM test1 WHERE id = *константа*;

Если система не будет заранее подготовлена, ей придётся сканировать всю таблицу test1, строку за строкой, чтобы найти все подходящие записи. Когда таблица test1 содержит большое количество записей, а этот запрос должен вернуть всего несколько (возможно, одну или ноль), такое сканирование, очевидно, неэффективно. Но если создать в системе индекс по полю id, она сможет находить строки гораздо быстрее. Возможно, для этого ей понадобится опуститься всего на несколько уровней в дереве поиска.

Задача программиста баз данных – заранее определить, какие индексы будут полезны.

Создать индекс для столбца id рассмотренной ранее таблицы можно с помощью следующей команды:

CREATE INDEX test1\_id\_index ON test1 (id);

Имя индекса test1\_id\_index может быть произвольным, главное, чтобы оно позволяло понять, для чего этот индекс.

Для удаления индекса используется команда DROP INDEX. Добавлять и удалять индексы можно в любое время.

Когда индекс создан, никакие дополнительные действия не требуются: система сама будет обновлять его при изменении данных в таблице и сама будет использовать его в запросах, где, по её мнению, это будет эффективнее, чем сканирование всей таблицы. Возможно, придётся только периодически запускать команду ANALYZE для обновления статистических данных, на основе которых планировщик запросов принимает решения.

Индексы могут быть полезны также при выполнении команд UPDATE и DELETE с условиями поиска. Кроме того, они могут применяться в поиске с соединением. То есть, индекс, определённый для столбца, участвующего в условии соединения, может значительно ускорить запросы с JOIN.

Создание индекса для большой таблицы может занимать много времени. По умолчанию PostgreSQL позволяет параллельно с созданием индекса выполнять чтение (операторы SELECT) таблицы, но операции записи (INSERT, UPDATE и DELETE) блокируются до окончания построения индекса. Для производственной среды это ограничение часто бывает неприемлемым. Хотя есть возможность разрешить запись параллельно с созданием индексов.

После создания индекса система должна поддерживать его в состоянии, соответствующем данным таблицы. С этим связаны неизбежные накладные расходы при изменении данных. Таким образом, индексы, которые используются в запросах редко или вообще никогда, должны быть удалены.

PostgreSQL поддерживает несколько типов индексов: B-дерево, хеш, GiST, SP-GiST, GIN и BRIN. Для разных типов индексов применяются разные алгоритмы, ориентированные на определённые типы запросов. По умолчанию команда CREATE INDEX создаёт индексы типа B-дерево, эффективные в большинстве случаев.

B-деревья могут работать в условиях на равенство и в проверках диапазонов с данными, которые можно отсортировать в некотором порядке. Точнее, планировщик запросов PostgreSQL может задействовать индекс-B-дерево, когда индексируемый столбец участвует в сравнении с одним из следующих операторов:

|  |
| --- |
| < |
| <= |
| = |
| >= |
| > |

При обработке конструкций, представимых как сочетание этих операторов, например BETWEEN и IN, так же может выполняться поиск по индексу-B-дереву. Кроме того, такие индексы могут использоваться и в условиях IS NULL и IS NOT NULL по индексированным столбцам.

Также оптимизатор может использовать эти индексы в запросах с операторами сравнения по шаблону LIKE и ~, если этот шаблон определяется константой и он привязан к началу строки – например, col LIKE 'foo%' или col ~ '^foo', но не col LIKE '%bar'. Но если ваша база данных использует не локаль C, для поддержки индексирования запросов с шаблонами вам потребуется создать индекс со специальным классом операторов. Индексы-B-деревья можно использовать и для ILIKE и ~\*, но только если шаблон начинается не с алфавитных символов, то есть символов, не подверженных преобразованию регистра.

B-деревья могут также применяться для получения данных, отсортированных по порядку. Это не всегда быстрее простого сканирования и сортировки, но иногда бывает полезно.

Хеш-индексы работают только с простыми условиями равенства. Планировщик запросов может применить хеш-индекс, только если индексируемый столбец участвует в сравнении с оператором =. Создать такой индекс можно следующей командой:

CREATE INDEX *имя* ON *таблица* USING HASH (*столбец*);

Операции с хеш-индексами в настоящее время не проходят через WAL, так что после аварийной остановки базы данных может потребоваться перестроить хеш-индексы командой REINDEX. Кроме того, изменения в хеш-индексах после начальной копии не переносятся при потоковой или файловой репликации, так что в последующих запросах они будут давать неправильные ответы. По этим причинам настоятельно рекомендуется не использовать их.

GiST-индексы представляют собой не просто разновидность индексов, а инфраструктуру, позволяющую реализовать много разных стратегий индексирования. Как следствие, GiST-индексы могут применяться с разными операторами, в зависимости от стратегии индексирования (класса операторов). Например, стандартный дистрибутив PostgreSQL включает классы операторов GiST для нескольких двумерных типов геометрических данных, что позволяет применять индексы в запросах с операторами:

<<

&<

&>

>>

<<|

&<|

|&>

|>>

@>

<@

~=

&&

Классы операторов GiST, включённые в стандартный дистрибутив. В коллекции contrib можно найти и другие классы операторов GiST, реализованные как отдельные проекты.

GiST-индексы также могут оптимизировать поиск «ближайшего соседа», например такой:

SELECT \* FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;

который возвращает десять расположений, ближайших к заданной точке. Возможность такого применения индекса опять же зависит от класса используемого оператора.

Индексы SP-GiST, как и GiST, предоставляют инфраструктуру, поддерживающие различные типы поиска. SP-GiST позволяет организовывать на диске самые разные несбалансированные структуры данных, такие как деревья квадрантов, k-мерные и префиксные деревья. Например, стандартный дистрибутив PostgreSQL включает классы операторов SP-GiST для точек в двумерном пространстве, что позволяет применять индексы в запросах с операторами:

<<

>>

~=

<@

<^

>^

Классы операторов SP-GiST, включённые в стандартный дистрибутив. GIN-индексы представляют собой инвертированные индексы, в которых могут содержаться значения с несколькими ключами, например массивы. Подобно GiST и SP-GiST, индексы GIN могут поддерживать различные определённые пользователем стратегии и в зависимости от них могут применяться с разными операторами. Например, стандартный дистрибутив PostgreSQL включает классы операторов GIN для одномерных массивов, что позволяет применять индексы в запросах с операторами:

<@

@>

=

&&

Классы операторов GIN, включённые в стандартный дистрибутив. В коллекции contrib можно найти и другие классы операторов GIN, реализованные как отдельные проекты.

BRIN-индексы (сокращение от Block Range indexes, Индексы зон блоков) хранят обобщённые сведения о значениях, находящихся в физически последовательно расположенных блоках. Подобно GiST, SP-GiST и GIN, индексы BRIN могут поддерживать определённые пользователем стратегии, и в зависимости от них применяться с разными операторами. Для типов данных, имеющих линейный порядок сортировки, записям в индексе соответствуют минимальные и максимальные значения данных в столбце для каждой зоны блоков, что позволяет поддерживать запросы со следующими операторами:

<

<=

=

>=

>

Классы операторов BRIN, включённые в стандартный дистрибутив.

Индексы можно создавать и по нескольким столбцам таблицы. Например, если у вас есть таблица:

CREATE TABLE test2 (

major int,

minor int,

name varchar

);

(предположим, что вы поместили в неё содержимое каталога /dev) и вы часто выполняете запросы вида:

SELECT name FROM test2 WHERE major = *константа* AND minor = *константа*;

тогда имеет смысл определить индекс, покрывающий оба столбца major и minor. Например:

CREATE INDEX test2\_mm\_idx ON test2 (major, minor);

В настоящее время составными могут быть только индексы типов B-дерево, GiST, GIN и BRIN. Число столбцов в индексе ограничивается 32 (этот предел можно изменить при компиляции PostgreSQL).

Составной индекс-B-дерево может применяться в условиях с любым подмножеством столбцов индекса, но наиболее эффективен он при ограничениях по ведущим (левым) столбцам. Точное правило состоит в том, что сканируемая область индекса определяется условиями равенства с ведущими столбцами и условиями неравенства с первым столбцом, не участвующим в условии равенства. Ограничения столбцов правее них также проверяются по индексу, так что обращение к таблице откладывается, но на размер сканируемой области индекса это уже не влияет. Например, если есть индекс по столбцам (a, b, c) и условие WHERE a=5 AND b>=42 AND c<77, индекс будет сканироваться от первой записи a=5 и b=42 до последней с a=5. Записи индекса, в которых c >= 77, не будут учитываться, но, тем не менее, будут просканированы. Этот индекс в принципе может использоваться в запросах с ограничениями по b и/или c, без ограничений столбца a, но при этом будет просканирован весь индекс, так что в большинстве случаев планировщик предпочтёт использованию индекса полное сканирование таблицы.

Составной индекс GiST может применяться в условиях с любым подмножеством столбцов индекса. Условия с дополнительными столбцами ограничивают записи, возвращаемые индексом, но в первую очередь сканируемая область индекса определяется ограничением первого столбца. GiST-индекс будет относительно малоэффективен, когда первый его столбец содержит только несколько различающихся значений, даже если дополнительные столбцы дают множество различных значений.

Составной индекс GIN может применяться в условиях с любым подмножеством столбцов индекса. В отличие от индексов GiST или B-деревьев, эффективность поиска по нему не меняется в зависимости от того, какие из его столбцов используются в условиях запроса.

Составной индекс BRIN может применяться в условиях запроса с любым подмножеством столбцов индекса. Подобно индексу GIN и в отличие от B-деревьев или GiST, эффективность поиска по нему не меняется в зависимости от того, какие из его столбцов используются в условиях запроса. Единственное, зачем в одной таблице могут потребоваться несколько индексов BRIN вместо одного составного индекса — это затем, чтобы применялись разные параметры хранения pages\_per\_range.

При этом, разумеется, каждый столбец должен использоваться с операторами, соответствующими типу индекса; ограничения с другими операторами рассматриваться не будут.

Составные индексы следует использовать обдуманно. В большинстве случаев индекс по одному столбцу будет работать достаточно хорошо и сэкономит время и место. Индексы по более чем трём столбцам вряд ли будут полезными, если только таблица не используется крайне однообразно.

Помимо простого поиска строк для выдачи в результате запроса, индексы также могут применяться для сортировки строк в определённом порядке. Это позволяет учесть предложение ORDER BY в запросе, не выполняя сортировку дополнительно. Из всех типов индексов, которые поддерживает PostgreSQL, сортировать данные могут только B-деревья – индексы других типов возвращают строки в неопределённом, зависящем от реализации порядке.

Планировщик может выполнить указание ORDER BY, либо просканировав существующий индекс, подходящий этому указанию, либо просканировав таблицу в физическом порядке и выполнив сортировку явно. Для запроса, требующего сканирования большой части таблицы, явная сортировка скорее всего будет быстрее, чем применение индекса, так как при последовательном чтении она потребует меньше операций ввода/вывода. Важный особый случай представляет ORDER BY в сочетании с LIMIT *n*: при явной сортировке системе потребуется обработать все данные, чтобы выбрать первые n строк, но при наличии индекса, соответствующего столбцам в ORDER BY, первые *n* строк можно получить сразу, не просматривая остальные вовсе.

По умолчанию элементы B-дерева хранятся в порядке возрастания, при этом значения NULL идут в конце. Это означает, что при прямом сканировании индекса по столбцу x порядок оказывается соответствующим указанию ORDER BY x (или точнее, ORDER BY x ASC NULLS LAST). Индекс также может сканироваться в обратную сторону, и тогда порядок соответствует указанию ORDER BY x DESC (или точнее, ORDER BY x DESC NULLS FIRST, так как для ORDER BY DESC подразумевается NULLS FIRST).

Вы можете изменить порядок сортировки элементов B-дерева, добавив уточнения ASC, DESC, NULLS FIRST и/или NULLS LAST при создании индекса; например:

CREATE INDEX test2\_info\_nulls\_low ON test2 (info NULLS FIRST);

CREATE INDEX test3\_desc\_index ON test3 (id DESC NULLS LAST);

Индекс, в котором элементы хранятся в порядке возрастания и значения NULL идут первыми, может удовлетворять указаниям ORDER BY x ASC NULLS FIRST или ORDER BY x DESC NULLS LAST, в зависимости от направления просмотра.

У вас может возникнуть вопрос, зачем нужны все четыре варианта при создании индексов, когда и два варианта с учётом обратного просмотра покрывают все виды ORDER BY. Для индексов по одному столбцу это и в самом деле излишне, но для индексов по многим столбцам это может быть полезно. Рассмотрим индекс по двум столбцам (x, y): он может удовлетворять указанию ORDER BY x, y при прямом сканировании или ORDER BY x DESC, y DESC при обратном. Но вполне возможно, что приложение будет часто выполнять ORDER BY x ASC, y DESC. В этом случае получить такую сортировку от простого индекса нельзя, но можно получить подходящий индекс, определив его как (x ASC, y DESC) или (x DESC, y ASC).

Очевидно, что индексы с нестандартными правилами сортировки весьма специфичны, но иногда они могут кардинально ускорить определённые запросы. Стоит ли вводить такие индексы, зависит от того, как часто выполняются запросы с необычным порядком сортировки.

При простом сканировании индекса могут обрабатываться только те предложения в запросе, в которых применяются операторы его класса и объединяет их AND. Например, для индекса (a, b) условие запроса WHERE a = 5 AND b = 6 сможет использовать этот индекс, а запрос WHERE a = 5 OR b = 6 – нет.

К счастью, PostgreSQL способен соединять несколько индексов (и в том числе многократно применять один индекс) и охватывать также случаи, когда сканирования одного индекса недостаточно. Система может сформировать условия AND и OR за несколько проходов индекса. Например, запрос WHERE x = 42 OR x = 47 OR x = 53 OR x = 99 можно разбить на четыре сканирования индекса по x, по сканированию для каждой части условия. Затем результаты этих сканирований будут логически сложены (OR) вместе и дадут конечный результат. Другой пример – если у нас есть отдельные индексы по x и y, запрос WHERE x=5 AND y=6 можно выполнить, применив индексы для соответствующих частей запроса, а затем вычислив логическое произведение (AND) для найденных строк, которое и станет конечным результатом.

Выполняя объединение нескольких индексов, система сканирует все необходимые индексы и создаёт в памяти *битовую карту* расположения строк таблицы, которые удовлетворяют условиям каждого индекса. Затем битовые карты объединяются операциями AND и OR, как того требуют условия в запросе. Наконец система обращается к соответствующим отмеченным строкам таблицы и возвращает их данные. Строки таблицы просматриваются в физическом порядке, как они представлены в битовой карте; это означает, что порядок сортировки индексов при этом теряется и в запросах с предложением ORDER BY сортировка будет выполняться отдельно. По этой причине, а также потому, что каждое сканирование индекса занимает дополнительное время, планировщик иногда выбирает простое сканирование индекса, несмотря на то, что можно было бы подключить и дополнительные индексы.

В большинстве приложений (кроме самых простых) полезными могут оказаться различные комбинации индексов, поэтому разработчик баз данных, определяя набор индексов, должен искать компромиссное решение. Иногда оказываются хороши составные индексы, а иногда лучше создать отдельные индексы и положиться на возможности объединения индексов. Например, если типичную нагрузку составляют запросы иногда с условием только по столбцу x, иногда только по y, а иногда по обоим столбцам, вы можете ограничиться двумя отдельными индексами по x и y, рассчитывая на то, что при обработке условий с обоими столбцами эти индексы будут объединяться. С другой стороны, вы можете создать один составной индекс по (x, y). Этот индекс скорее всего будет работать эффективнее, чем объединение индексов, в запросах с двумя столбцами, но он будет практически бесполезен для запросов с ограничениями только по y, так что одного этого индекса будет недостаточно. Выигрышным в этом случае может быть сочетание составного индекса с отдельным индексом по y. В запросах, где задействуется только x, может применяться составной индекс, хотя он будет больше и, следовательно, медленнее индекса по одному x. Наконец, можно создать все три индекса, но это будет оправдано, только если данные в таблице изменяются гораздо реже, чем выполняется поиск в таблице, при этом частота запросов этих трёх типов примерно одинакова. Если запросы какого-то одного типа выполняются гораздо реже других, возможно лучше будет оставить только два индекса, соответствующих наиболее частым запросам.

Индексы также могут обеспечивать уникальность значения в столбце или уникальность сочетания значений в нескольких столбцах.

CREATE UNIQUE INDEX *имя* ON *таблица* (*столбец* [, ...]);

В настоящее время уникальными могут быть только индексы-B-деревья.

Если индекс создаётся как уникальный, в таблицу нельзя будет добавить несколько строк с одинаковыми значениями ключа индекса. При этом значения NULL считаются не равными друг другу. Составной уникальный индекс не принимает только те строки, в которых все индексируемые столбцы содержат одинаковые значения.

Когда для таблицы определяется ограничение уникальности или первичный ключ, PostgreSQL автоматически создаёт уникальный индекс по всем столбцам, составляющим это ограничение или первичный ключ (индекс может быть составным). Такой индекс и является механизмом, который обеспечивает выполнение ограничения.

Хотя индексы в PostgreSQL не требуют какого-либо обслуживания или настройки, это не избавляет от необходимости проверять, как и какие индексы используются на самом деле в реальных условиях. Узнать, как отдельный запрос использует индексы, можно с помощью команды EXPLAIN. Также возможно собрать общую статистику об использовании индексов на работающем сервере.

Вывести универсальную формулу, определяющую, какие индексы нужно создавать, довольно сложно, если вообще возможно. В предыдущих разделах рассматривались некоторые типовые ситуации, иллюстрирующие подходы к этому вопросу. Часто найти ответ на него помогают эксперименты. Ниже приведены ещё несколько советов:

* Всегда начинайте исследование с [ANALYZE](https://postgrespro.ru/docs/postgresql/9.5/sql-analyze). Эта команда собирает статистические данные о распределении значений в таблице, которые необходимы для оценивания числа строк, возвращаемых запросов. А это число, в свою очередь, нужно планировщику, чтобы оценить реальные затраты для всевозможных планов выполнения запроса. Не имея реальной статистики, планировщик будет вынужден принять некоторые значения по умолчанию, которые почти наверняка не будут соответствовать действительности. Поэтому понять, как индекс используется приложением без предварительного запуска ANALYZE, практически невозможно.
* Используйте в экспериментах реальные данные. Анализируя работу системы с тестовыми данными, вы поймёте, какие индексы нужны для тестовых данных, но не более того.

Особенно сильно искажают картину очень маленькие наборы тестовых данных. Тогда как для извлечения 1000 строк из 100000 может быть применён индекс, для выбора 1 из 100 он вряд ли потребуется, так как 100 строк скорее всего уместятся в одну страницу данных на диске и никакой другой план не будет лучше обычного сканирования 1 страницы.

Тем не менее, пока приложение не эксплуатируется, создавать какие-то тестовые данные всё равно нужно, и это нужно делать обдуманно. Если вы наполняете базу данных очень близкими, или наоборот, случайными значениями, либо добавляете строки в отсортированном порядке, вы получите совсем не ту статистику распределения, что дадут реальные данные.

* Когда индексы не используются, ради тестирования может быть полезно подключить их принудительно. Для этого можно воспользоваться параметрами выполнения, позволяющими выключать различные типы планов. Например, выключив наиболее простые планы: последовательное сканирование (enable\_seqscan) и соединения с вложенными циклами (enable\_nestloop), вы сможете заставить систему выбрать другой план. Если же система продолжает выполнять сканирование или соединение с вложенными циклами, вероятно, у неё есть более серьёзная причина не использовать индекс; например, индекс может не соответствовать условию запроса.
* Если система начинает использовать индекс только под принуждением, тому может быть две причины: либо система права и применять индекс в самом деле неэффективно, либо оценка стоимости применения индекса не соответствует действительности. В этом случае вам следует замерить время выполнения запроса с индексами и без них. В анализе этой ситуации может быть полезна команда EXPLAIN ANALYZE.
* Если выясняется, что оценка стоимости неверна, это может иметь тоже два объяснения. Общая стоимость вычисляется как произведение цены каждого узла плана для одной строки и оценки избирательности узла плана. Цены узлов при необходимости можно изменить параметрами выполнения. С другой стороны, оценка избирательности может быть неточной из-за некачественной статистики. Улучшить её можно, настроив параметры сбора статистики.

Более подробная информация: [PostgreSQL : Документация: 9.5: Индексы](https://postgrespro.ru/docs/postgresql/9.5/indexes.html)

**5. Оборудование**

Персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

**6. Задание на работу**

1. Создайте таблицу для экспериментов. В таблице должно обязательно быть два целочисленных поля. Первое поле будет использоваться в качестве ключа поиска. Это может быть номер зачетной книжки, табельный номер сотрудника, номер заказа, и т.п. Второе поле - информационное – нужная нам информация (например – зарплата, стипендия, средний балл, сумма заказа). Оставьте эту таблицу пустой.
2. Напишите процедуру, заполняющую эту таблицу случайными данными (например, используйте формулу псевдослучайной последовательности*x*[*i + 1*]*= x*[*i*] \* 1103515245 + 12345). Реализуйте ее так, чтобы в таблице появилось сто-двести строк с одинаковыми значениями столбца (но не добавляйте эти строки одновременно - физически они должны появиться на разных станицах. Проще всего это сделать, разделяя значение псевдослучайной последовательности по модулю. Например, если вы хотите добавить 150 тысяч строк, разделите по модулю значение псевдослучайной последовательности на 100, тогда у вас появится где-то по 150 строк с одинаковыми значениями). Запустите ее. Засеките время ее выполения - количество данных должно быть достаточным, чтобы это время стало ощутимым. Повторите это пять раз (перед каждым разом очищая таблицу), чтобы оценить, сколько времени занимает выполнение процедуры в среднем.
3. Напишите процедуру, тестирующую время выборки данных. Объявите в ней числовую переменную. В процедуре должен быть цикл, в теле которого выбирается запись по случайному ключу, а значение информационного поля добавляется к объявленной числовой переменной. Например, вам нужно найти общую сумму заказов со случайными номерами. Запустите эту процедуру и измерьте время ее выполнения (повторите пять раз).
4. Проиндексируйте ключевое поле и повторите пункты 2 и 3. Сравните измеренное время с результатами, полученными ранее. Время должно отличаться! Если во всех экспериментах оно стремится к нулю, значит у вас недостаточно итераций циклов и недостаточно данных. В этом случае сделайте необходимые изменения и повторите весь комплекс экспериментов.

**6. Контрольные вопросы**

1. Дать определение индексу БД.
2. Перечислите основные типы индексов.
3. Как индексы могут влиять на производительность?
4. Напишите примеры использования оператора CREATE INDEX.
5. Напишите примеры использования оператора DROP INDEX.

**Лабораторная работа №13**

**Ограничения целостности**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение ограничений целостности.

#### 2. Порядок выполнения работы

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

#### 3. Оформление отчета

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

#### 4. Теоретические сведения

Типы данных сами по себе ограничивают множество данных, которые можно сохранить в таблице. Однако для многих приложений такие ограничения слишком грубые. Например, столбец, содержащий цену продукта, должен, вероятно, принимать только положительные значения. Но такого стандартного типа данных нет. Возможно, вы также захотите ограничить данные столбца по отношению к другим столбцам или строкам. Например, в таблице с информацией о товаре должна быть только одна строка с определённым кодом товара.

Для решения подобных задач SQL позволяет вам определять ограничения для столбцов и таблиц. Ограничения дают вам возможность управлять данными в таблицах так, как вы захотите. Если пользователь попытается сохранить в столбце значение, нарушающее ограничения, возникнет ошибка. Ограничения будут действовать, даже если это значение по умолчанию.

***Ограничения-проверки***

Ограничение-проверка – наиболее общий тип ограничений. В его определении вы можете указать, что значение данного столбца должно удовлетворять логическому выражению (проверке истинности). Например, цену товара можно ограничить положительными значениями так:

CREATE TABLE products (

product\_no integer,

name text,

price numeric **CHECK (price > 0)**

);

Как вы видите, ограничение определяется после типа данных, как и значение по умолчанию. Значения по умолчанию и ограничения могут указываться в любом порядке. Ограничение-проверка состоит из ключевого слова CHECK, за которым идёт выражение в скобках. Это выражение должно включать столбец, для которого задаётся ограничение, иначе оно не имеет большого смысла.

Вы можете также присвоить ограничению отдельное имя. Это улучшит сообщения об ошибках и позволит вам ссылаться на это ограничение, когда вам понадобится изменить его. Сделать это можно так:

CREATE TABLE products (

product\_no integer,

name text,

price numeric **CONSTRAINT positive\_price** CHECK (price > 0)

);

То есть, чтобы создать именованное ограничение, напишите ключевое слово CONSTRAINT, а за ним идентификатор и собственно определение ограничения (если вы не определите имя ограничения таким образом, система выберет для него имя за вас).

Ограничение-проверка может также ссылаться на несколько столбцов. Например, если вы храните обычную цену и цену со скидкой, так вы можете гарантировать, что цена со скидкой будет всегда меньше обычной:

CREATE TABLE products (

product\_no integer,

name text,

price numeric CHECK (price > 0),

discounted\_price numeric CHECK (discounted\_price > 0),

**CHECK (price > discounted\_price)**

);

Первые два ограничения определяются похожим образом, но для третьего используется новый синтаксис. Оно не связано с определённым столбцом, а представлено отдельным элементом в списке. Определения столбцов и такие определения ограничений можно переставлять в произвольном порядке.

Про первые два ограничения можно сказать, что это ограничения столбцов, тогда как третье является ограничением таблицы, так как оно написано отдельно от определений столбцов. Ограничения столбцов также можно записать в виде ограничений таблицы, тогда как обратное не всегда возможно, так как подразумевается, что ограничение столбца ссылается только на связанный столбец. Ранее приведённый пример можно переписать и так:

CREATE TABLE products (

product\_no integer,

name text,

price numeric,

CHECK (price > 0),

discounted\_price numeric,

CHECK (discounted\_price > 0),

CHECK (price > discounted\_price)

);

Или даже так:

CREATE TABLE products (

product\_no integer,

name text,

price numeric CHECK (price > 0),

discounted\_price numeric,

CHECK (discounted\_price > 0 AND price > discounted\_price)

);

Это дело вкуса.

Ограничениям таблицы можно присваивать имена так же, как и ограничениям столбцов:

CREATE TABLE products (

product\_no integer,

name text,

price numeric,

CHECK (price > 0),

discounted\_price numeric,

CHECK (discounted\_price > 0),

**CONSTRAINT valid\_discount** CHECK (price > discounted\_price)

);

Следует заметить, что ограничение-проверка удовлетворяется, если выражение принимает значение true или NULL. Так как результатом многих выражений с операндами NULL будет значение NULL, такие ограничения не будут препятствовать записи NULL в связанные столбцы. Чтобы гарантировать, что столбец не содержит значения NULL, можно использовать ограничение NOT NULL, описанное в следующем разделе.

***Ограничения NOT NULL***

Ограничение NOT NULL просто указывает, что столбцу нельзя присваивать значение NULL. Пример синтаксиса:

CREATE TABLE products (

product\_no integer **NOT NULL**,

name text **NOT NULL**,

price numeric

);

Ограничение NOT NULL всегда записывается как ограничение столбца и функционально эквивалентно ограничению CHECK (*имя\_столбца* IS NOT NULL). Хотя у такой записи есть недостаток – назначить имя таким ограничениям нельзя.

Естественно, для столбца можно определить больше одного ограничения. Для этого их нужно просто указать одно за другим:

CREATE TABLE products (

product\_no integer NOT NULL,

name text NOT NULL,

price numeric NOT NULL CHECK (price > 0)

);

Порядок здесь не имеет значения, он не обязательно соответствует порядку проверки ограничений.

Для ограничения NOT NULL есть и обратное: ограничение NULL. Оно не означает, что столбец должен иметь только значение NULL, что конечно было бы бессмысленно. Суть же его в простом указании, что столбец может иметь значение NULL (это поведение по умолчанию). Ограничение NULL отсутствует в стандарте SQL и использовать его в переносимых приложениях не следует. Однако некоторые пользователи любят его использовать, так как оно позволяет легко переключать ограничения в скрипте. Например, вы можете начать с:

CREATE TABLE products (

product\_no integer NULL,

name text NULL,

price numeric NULL

);

и затем вставить ключевое слово NOT, где потребуется.

***Ограничения уникальности***

Ограничения уникальности гарантируют, что данные в определённом столбце или группе столбцов уникальны среди всех строк таблицы. Ограничение записывается так:

CREATE TABLE products (

product\_no integer **UNIQUE**,

name text,

price numeric

);

в виде ограничения столбца и так:

CREATE TABLE products (

product\_no integer,

name text,

price numeric,

**UNIQUE (product\_no)**

);

в виде ограничения таблицы.

Чтобы определить ограничение уникальности для группы столбцов, запишите его в виде ограничения таблицы, перечислив имена столбцов через запятую:

CREATE TABLE example (

a integer,

b integer,

c integer,

**UNIQUE (a, c)**

);

Такое ограничение указывает, что сочетание значений перечисленных столбцов должно быть уникально во всей таблице, тогда как значения каждого столбца по отдельности не должны быть (и обычно не будут) уникальными.

Можно назначить уникальному ограничению имя обычным образом:

CREATE TABLE products (

product\_no integer **CONSTRAINT must\_be\_different** UNIQUE,

name text,

price numeric

);

При добавлении ограничения уникальности будет автоматически создан уникальный индекс-B-дерево для столбца или группы столбцов, перечисленных в ограничении. Условие уникальности, распространяющееся только на некоторые строки, нельзя записать в виде ограничения уникальности, однако такое условие можно установить, создав уникальный [частичный индекс](https://postgrespro.ru/docs/postgrespro/9.5/indexes-partial).

Вообще говоря, ограничение уникальности нарушается, если в таблице оказывается несколько строк, у которых совпадают значения всех столбцов, включённых в ограничение. Однако два значения NULL при сравнении никогда не считаются равными. Это означает, что даже при наличии ограничения уникальности в таблице можно сохранить строки с дублирующимися значениями, если они содержат NULL в одном или нескольких столбцах ограничения. Это поведение соответствует стандарту SQL, но мы слышали о СУБД, которые ведут себя по-другому. Имейте в виду эту особенность, разрабатывая переносимые приложения.

***Первичные ключи***

Ограничение первичного ключа означает, что образующий его столбец или группа столбцов может быть уникальным идентификатором строк в таблице. Для этого требуется, чтобы значения были одновременно уникальными и отличными от NULL. Таким образом, таблицы со следующими двумя определениями будут принимать одинаковые данные:

CREATE TABLE products (

product\_no integer UNIQUE NOT NULL,

name text,

price numeric

);

CREATE TABLE products (

product\_no integer **PRIMARY KEY**,

name text,

price numeric

);

Первичные ключи могут включать несколько столбцов; синтаксис похож на запись ограничений уникальности:

CREATE TABLE example (

a integer,

b integer,

c integer,

**PRIMARY KEY (a, c)**

);

При добавлении первичного ключа автоматически создаётся уникальный индекс-B-дерево для столбца или группы столбцов, перечисленных в первичном ключе, и данные столбцы помечаются как NOT NULL.

Таблица может иметь максимум один первичный ключ (ограничений уникальности и ограничений NOT NULL, которые функционально почти равнозначны первичным ключам, может быть сколько угодно, но назначить ограничением первичного ключа можно только одно). Теория реляционных баз данных говорит, что первичный ключ должен быть в каждой таблице.

Первичные ключи полезны и для документирования, и для клиентских приложений. Например, графическому приложению с возможностями редактирования содержимого таблицы, вероятно, потребуется знать первичный ключ таблицы, чтобы однозначно идентифицировать её строки. Первичные ключи находят и другое применение в СУБД; в частности, первичный ключ в таблице определяет целевые столбцы по умолчанию для сторонних ключей, ссылающихся на эту таблицу.

***Внешние ключи***

Ограничение внешнего ключа указывает, что значения столбца (или группы столбцов) должны соответствовать значениям в некоторой строке другой таблицы. Это называется *ссылочной целостностью* двух связанных таблиц.

Пусть у вас уже есть таблица продуктов, которую мы неоднократно использовали ранее:

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name text,

price numeric

);

Предположим, что есть таблица с заказами этих продуктов. Мы хотим, чтобы в таблице заказов содержались только заказы действительно существующих продуктов. Поэтому определим в ней ограничение внешнего ключа, ссылающееся на таблицу продуктов:

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

product\_no integer **REFERENCES products (product\_no)**,

quantity integer

);

С таким ограничением создать заказ со значением product\_no, отсутствующим в таблице products (и не равным NULL), будет невозможно.

В такой схеме таблицу orders называют *подчинённой* таблицей, а products – *главной*. Соответственно, столбцы называют так же подчинённым и главным (или ссылающимся и целевым).

Предыдущую команду можно сократить так:

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

product\_no integer **REFERENCES products**,

quantity integer

);

то есть, если опустить список столбцов, внешний ключ будет неявно связан с первичным ключом главной таблицы.

Внешний ключ также может ссылаться на группу столбцов. В этом случае его нужно записать в виде обычного ограничения таблицы. Например:

CREATE TABLE t1 (

a integer PRIMARY KEY,

b integer,

c integer,

**FOREIGN KEY (b, c) REFERENCES other\_table (c1, c2)**

);

Естественно, число и типы столбцов в ограничении должны соответствовать числу и типам целевых столбцов.

Ограничению внешнего ключа можно назначить имя стандартным способом.

Таблица может содержать несколько ограничений внешнего ключа. Это полезно для связи таблиц в отношении многие-ко-многим. Скажем, у вас есть таблицы продуктов и заказов, но вы хотите, чтобы один заказ мог содержать несколько продуктов (что невозможно в предыдущей схеме). Для этого вы можете использовать такую схему:

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name text,

price numeric

);

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

shipping\_address text,

...

);

CREATE TABLE order\_items (

product\_no integer REFERENCES products,

order\_id integer REFERENCES orders,

quantity integer,

PRIMARY KEY (product\_no, order\_id)

);

В последней таблице первичный ключ покрывает внешние ключи.

Внешние ключи запрещают создание заказов, не относящихся ни к одному продукту. Но что делать, если после создания заказов с определённым продуктом мы захотим удалить его? При попытке удаления продукта, на который ссылаются заказы (через таблицу order\_items), мы запрещаем эту операцию. Если же кто-то попытается удалить заказ, то удалится и его содержимое:

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name text,

price numeric

);

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

shipping\_address text,

...

);

CREATE TABLE order\_items (

product\_no integer REFERENCES products **ON DELETE RESTRICT**,

order\_id integer REFERENCES orders **ON DELETE CASCADE**,

quantity integer,

PRIMARY KEY (product\_no, order\_id)

);

Ограничивающие и каскадные удаления – два наиболее распространённых варианта. RESTRICT предотвращает удаление связанной строки. NO ACTION означает, что если зависимые строки продолжают существовать при проверке ограничения, возникает ошибка (это поведение по умолчанию). (Главным отличием этих двух вариантов является то, что NO ACTION позволяет отложить проверку в процессе транзакции, а RESTRICT — нет.) CASCADE указывает, что при удалении связанных строк зависимые от них будут так же автоматически удалены. Есть ещё два варианта: SET NULL и SET DEFAULT. При удалении связанных строк они назначают зависимым столбцам в подчинённой таблице значения NULL или значения по умолчанию, соответственно. Заметьте, что это не будет основанием для нарушения ограничений. Например, если в качестве действия задано SET DEFAULT, но значение по умолчанию не удовлетворяет ограничению внешнего ключа, операция закончится ошибкой.

Аналогично указанию ON DELETE существует ON UPDATE, которое срабатывает при изменении заданного столбца. При этом возможные действия те же, а CASCADE в данном случае означает, что изменённые значения связанных столбцов будут скопированы в зависимые строки.

Обычно зависимая строка не должна удовлетворять ограничению внешнего ключа, если один из связанных столбцов содержит NULL. Если в объявление внешнего ключа добавлено MATCH FULL, строка будет удовлетворять ограничению, только если все связанные столбцы равны NULL (то есть при разных значениях (NULL и не NULL) гарантируется невыполнение ограничения MATCH FULL). Если вы хотите, чтобы зависимые строки не могли избежать и этого ограничения, объявите связанные столбцы как NOT NULL.

Внешний ключ должен ссылаться на столбцы, образующие первичный ключ или ограничение уникальности. Таким образом, для связанных столбцов всегда будет существовать индекс (определённый соответствующим первичным ключом или ограничением), а значит проверки соответствия связанной строки будут выполняться эффективно. Так как команды DELETE для строк главной таблицы или UPDATE для зависимых столбцов потребуют просканировать подчинённую таблицу и найти строки, ссылающиеся на старые значения, полезно будет иметь индекс и для подчинённых столбцов. Но это нужно не всегда, и создать соответствующий индекс можно по-разному, поэтому объявление внешнего ключа не создаёт автоматически индекс по связанным столбцам.

***Ограничения-исключения***

Ограничения-исключения гарантируют, что при сравнении любых двух строк по указанным столбцам или выражениям с помощью заданных операторов, минимум одно из этих сравнений возвратит false или NULL. Записывается это так:

CREATE TABLE circles (

c circle,

EXCLUDE USING gist (c WITH &&)

);

При добавлении ограничения-исключения будет автоматически создан индекс того типа, который указан в объявлении ограничения.

**5. Задание на работу**

1. Напишите SQL-команду (**ALTER TABLE**), добавляющую в любую из ваших таблиц следующие ограничения целостности: **NOT NULL, UNIQUE, PRIMARY KEY**.
2. Напишите SQL-команду, добавляющую в таблицу (дочернюю), связанную с первой (базовой), ограничение **FOREIGN KEY**. Вы должны сделать это в трех вариантах, по-разному описывая действие при операции с первичным ключом базовой таблицы.
3. Добавьте в таблицу ограничение **CHECK**.
4. Приведите пример динамического контроля целостности.

Для каждого пункта необходимо привести:

1. запрос, добавляющий ограничение;
2. формулировку смысла ограничения (в понятиях предметной области);
3. запрос, модифицирующий данные некорректно (не соответствуя ограничениям), и ответ сервера БД на этот запрос;
4. запрос, модифицирующий данные корректно, и ответ сервера на этот запрос.

**5. Оборудование**

Персональный компьютер с установленной операционной системой WindowsXP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБДPostgreSQL.

**6. Контрольные вопросы**

1. Что понимается под ограничением целостности базы данных?
2. Какие виды ограничений целостности данных Вам известны?
3. Как осуществляются непосредственно проверяемые ограничения целостности?
4. Как осуществляются откладываемые ограничения целостности?
5. Что понимается под ссылочной целостностью данных?
6. Что означает ограничение целостности CHECK? Когда условие ограничения CHECK считается выполненным (удовлетворяется)? Можно ли определить ограничение CHECK над несколькими столбцами в синтаксисе ограничения столбца? Можно ли задать несколько ограничений CHECK для одного столбца? Проверяет ли PostgreSQL тот факт, что множество ограничений CHECK, сформулированных для одного столбца, являются взаимно исключающими? Можно ли определить ограничение CHECK над столбцами различных таблиц?

**Лабораторная работа №14**

**Привилегии**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение и практическое применение привилегий.

#### 2. Порядок выполнения работы

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

#### 3. Оформление отчета

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

#### 4. Теоретические сведения

Команда GRANT имеет две основные разновидности: первая назначает права для доступа к объектам баз данных (таблицам, столбцам, представлениям, сторонним таблицам, последовательностям, базам данных, обёрткам сторонних данных, сторонним серверам, функциям, процедурным языкам, схемам или табличным пространствам), а вторая назначает одни роли членами других. Эти разновидности во многом похожи, но имеют достаточно отличий, чтобы рассматривать их отдельно.

***GRANT для объектов баз данных***

Эта разновидность команды GRANT даёт одной или нескольким ролям определённые права для доступа к объекту базы данных. Эти права добавляются к списку имеющихся, если роль уже наделена какими-то правами.

Также можно дать роли некоторое право для всех объектов одного типа в одной или нескольких схемах. Эта функциональность в настоящее время поддерживается только для таблиц, последовательностей и функций (но заметьте, что указание ALL TABLES распространяется также на представления и сторонние таблицы).

Ключевое слово PUBLIC означает, что права даются всем ролям, включая те, что могут быть созданы позже. PUBLIC можно воспринимать как неявно определённую группу, в которую входят все роли. Любая конкретная роль получит в сумме все права, данные непосредственно ей и ролям, членом которых она является, а также права, данные роли PUBLIC.

Если указано WITH GRANT OPTION, получатель права, в свою очередь, может давать его другим. Без этого указания распоряжаться своим правом он не сможет. Группе PUBLIC право передачи права дать нельзя.

Нет необходимости явно давать права для доступа к объекту его владельцу (обычно это пользователь, создавший объект), так как по умолчанию он имеет все права (однако владелец может лишить себя прав в целях безопасности).

Право удалять объект или изменять его определение произвольным образом не считается назначаемым; оно неотъемлемо связано с владельцем, так что отозвать это право или дать его кому-то другому нельзя (однако похожий эффект можно получить, управляя членством в роли, владеющей объектом). Владелец также неявно получает право распоряжения всеми правами для своего объекта.

PostgreSQL по умолчанию назначает группе PUBLIC определённые права для некоторых типов объектов. Для таблиц, столбцов, последовательностей, обёрток сторонних данных, сторонних серверов, больших объектов, схем или табличных пространств PUBLIC по умолчанию никаких прав не имеет. Для других типов объектов PUBLIC имеет следующие права по умолчанию: CONNECT и TEMPORARY (создание временных таблиц) для баз данных; EXECUTE – для функций, USAGE – для языков и типов данных (включая домены). Владелец объекта, конечно же, может отозвать (с помощью REVOKE) как явно назначенные права, так и права по умолчанию (для максимальной безопасности команду REVOKE нужно выполнять в транзакции, создающей объект; тогда не образуется окно, в котором другой пользователь смог бы обратиться к объекту). Кроме того, эти изначально назначаемые права по умолчанию можно изменить, воспользовавшись командой [ALTER DEFAULT PRIVILEGES](https://postgrespro.ru/docs/postgresql/9.6/sql-alterdefaultprivileges).

Все возможные права перечислены ниже:

**SELECT**

Позволяет выполнять [SELECT](https://postgrespro.ru/docs/postgresql/9.6/sql-select) для любого столбца или перечисленных столбцов в заданной таблице, представлении или последовательности. Также позволяет выполнять [COPY](https://postgrespro.ru/docs/postgresql/9.6/sql-copy) TO. Помимо того, это право требуется для обращения к существующим значениям столбцов в [UPDATE](https://postgrespro.ru/docs/postgresql/9.6/sql-update) или [DELETE](https://postgrespro.ru/docs/postgresql/9.6/sql-delete). Для последовательностей это право позволяет пользоваться функцией currval. Для больших объектов это право позволяет читать содержимое объекта.

**INSERT**

Позволяет вставлять строки в заданную таблицу с помощью [INSERT](https://postgrespro.ru/docs/postgresql/9.6/sql-insert). Если право ограничивается несколькими столбцами, только их значение можно будет задать в команде INSERT (другие столбцы получат значения по умолчанию). Также позволяет выполнять [COPY](https://postgrespro.ru/docs/postgresql/9.6/sql-copy) FROM.

**UPDATE**

Позволяет изменять (с помощью [UPDATE](https://postgrespro.ru/docs/postgresql/9.6/sql-update)) данные во всех, либо только перечисленных, столбцах в заданной таблице (на практике для любой нетривиальной команды UPDATE потребуется и право SELECT, так как она должна обратиться к столбцам таблицы, чтобы определить, какие строки подлежат изменению, и/или вычислить новые значения столбцов). Для SELECT ... FOR UPDATE и SELECT ... FOR SHARE также необходимо это право как минимум для одного столбца, помимо права SELECT. Для последовательностей это право позволяет пользоваться функциями nextval и setval. Для больших объектов это право позволяет записывать данные в объект или обрезать его.

**DELETE**

Позволяет удалять строки из заданной таблицы с помощью [DELETE](https://postgrespro.ru/docs/postgresql/9.6/sql-delete) (на практике для любой нетривиальной команды DELETE потребуется также право SELECT, так как она должна обратиться к столбцам таблицы, чтобы определить, какие строки подлежат удалению).

**TRUNCATE**

Позволяет опустошить заданную таблицу с помощью [TRUNCATE](https://postgrespro.ru/docs/postgresql/9.6/sql-truncate).

**REFERENCES**

Это право требуется для создания ограничений внешнего ключа, как для ссылающихся, так и для целевых столбцов. Дать это право можно для всех, либо только для некоторых столбцов таблицы.

**TRIGGER**

Позволяет создавать триггеры в заданной таблице.

**CREATE**

Для баз данных это право позволяет создавать схемы в заданной базе.

Для схем это право позволяет создавать новые объекты в заданной схеме. Чтобы переименовать существующий объект, необходимо быть владельцем этого объекта ииметь это право для схемы, содержащей его.

Для табличных пространств это право позволяет создавать таблицы, индексы и временные файлы в заданном табличном пространстве, а также создавать базы данных, для которых это пространство будет основным (учтите, что когда это право отзывается, существующие объекты остаются в прежнем расположении).

**CONNECT**

Позволяет пользователю подключаться к указанной базе данных. Это право проверяется при установлении соединения (в дополнение к условиям, определённым в конфигурации pg\_hba.conf).

**TEMPORARY TEMP**

Позволяет создавать временные таблицы в заданной базе данных.

**EXECUTE**

Позволяет выполнять заданную функцию и применять любые определённые поверх неё операторы. Это единственный тип прав, применимый к функциям (этот синтаксис распространяется и на агрегатные функции).

**USAGE**

Для процедурных языков это право позволяет создавать функции на заданном языке. Это единственный тип прав, применимый к процедурным языкам.

Для схем это право даёт доступ к объектам, содержащимся в заданной схеме (предполагается, что при этом имеются права, необходимые для доступа к самим объектам). По сути это право позволяет субъекту «просматривать» объекты внутри схемы. Без этого разрешения имена объектов всё же можно будет узнать, например, обратившись к системным таблицам. Кроме того, если отозвать это право, в существующих сеансах могут оказаться операторы, для которых просмотр имён объектов был выполнен ранее, так что это право не позволяет абсолютно надёжно перекрыть доступ к объектам.

Для последовательностей это право позволяет использовать функции currval и nextval.

Для типов и доменов это право позволяет использовать заданный тип или домен при создании таблиц, функций или других объектов схемы (это право не ограничивает общее «использование» типа, например, обращение к значениям типа в запросах. Не имея этого права, субъект лишается только возможности создавать объекты, зависящие от заданного типа. Основное предназначение этого права в том, чтобы ограничить круг пользователей, способных создавать зависимости от типа, которые могут впоследствии помешать владельцу типа изменить его).

Для обёрток сторонних данных это право позволяет субъекту создавать определения сторонних серверов, использующих заданную обёртку сторонних данных.

Для серверов это право позволяет субъекту создавать определения сторонних таблиц на заданном сервере, а также создавать, изменять и удалять сопоставление для собственного имени пользователя, связанное с этим сервером.

**ALL PRIVILEGES**

Даёт целевой роли все права сразу. Ключевое слово PRIVILEGES является необязательным в PostgreSQL, хотя в строгом SQL оно требуется.

Более подробная информация: [PostgreSQL : Документация: 9.6: GRANT](https://postgrespro.ru/docs/postgresql/9.6/sql-grant.html)

**5. Оборудование**

Персональный компьютер с установленной операционной системой WindowsXP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБДPostgreSQL.

**6. Задание на работу**

1. Создайте учетные записи (в PostgreSQL они называются «роли входа») двух пользователей по одному на каждую локальную область из вашего варианта и две учетные записи помощников администратора.
2. На учетных записях помощников администратора продемонстрируйте работу привилегий на доступ к схеме – **USAGE, CREATE**. Две из указанных системных привилегий должны быть у обоих пользователей, однако администратор выдает их только одному пользователю (с возможностью наследованной передачи – конструкция **WITH GRANT OPTION**).

На учетных записях пользователей проверьте работу привилегий на доступ к объекту – **SELECT, DELETE, UPDATE, INSERT**, а также **UPDATE** для определенного столбца. Покажите, как работает конструкция **WITH GRANT OPTION** в данном случае.

Работа каждой привилегии должна быть продемонстрирована следующим образом:

а) выполнить SQL-команду, требующую привилегии. Убедиться, что ее выполнение вызывает ошибку.

б) выдать привилегию

в) выполнить команду из пункта «а» и убедиться, что все заработало.

**7. Контрольные вопросы**

1. Каким объектам могут быть предоставлены системные привилегии?
2. Укажите все способы наделения пользователя привилегиями. Какой пользователь может назначить другому пользователю роль?
3. Как полностью/частично аннулировать пользователю его привилегии? Какой пользователь может это сделать?
4. Приведите примеры использования оператора GRANT средствами встроенного SQL.
5. Приведите примеры использования оператора REVOKE средствами встроенного SQL

**Лабораторная работа №15**

**Роли**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение и практическое применение ролей.

#### 2.Порядок выполнения работы

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

#### 3. Оформление отчета

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

#### 4. Теоретические сведения

Большое число пользователей, статус которых требует различных привилегий для доступа к ресурсам базы данных, создает значительный объем рутинной работы администратору. *Роль* – это поименованный набор привилегий, который может быть предоставлен пользователю или другой роли. Роль не является объектом какой-либо схемы.

Описание привилегий, характерных для той или иной роли, готовится заранее. При регистрации нового пользователя в системе администратор выполняет только предоставление пользователю привилегий конкретной роли. При необходимости изменить привилегии конкретному приложению достаточно изменить только привилегии соответствующей роли. Все пользователи, отображенные на эту роль, автоматически получат измененные привилегии. Дополнительное преимущество использования ролей состоит в том, что роль может динамически разрешаться или запрещаться (например, при запуске приложения).

***GRANT для ролей***

Эта разновидность команды GRANT включает роль в члены одной или нескольких других ролей. Членство в ролях играет важную роль, так как права, данные роли, распространяются и на всех её членов.

С указанием WITH ADMIN OPTION новоиспечённый член роли сможет, в свою очередь, включать в члены этой роли, а также исключать из неё другие роли. Без этого указания обычные пользователи не могут это делать. Считается, что роль не имеет права WITH ADMIN OPTION для самой себя, но ей позволено управлять своими членами из сеанса, в котором пользователь сеанса соответствует данной роли. Суперпользователи баз данных могут включать или исключать любые роли из любых ролей. Роли с правом CREATEROLE могут управлять членством в любых ролях, кроме ролей суперпользователей.

В отличие от прав, членство в ролях нельзя назначить группе PUBLIC. Заметьте также, что эта форма команды не принимает избыточное слово GROUP.

Для лишения субъектов прав доступа применяется команда [REVOKE](https://postgrespro.ru/docs/postgresql/9.6/sql-revoke).

Начиная с PostgreSQL версии 8.1, концепции пользователей и групп объединены в единую сущность, названную ролью. Таким образом, теперь нет необходимости добавлять ключевое слово GROUP, чтобы показать, что субъект является группой, а не пользователем. Слово GROUP всё ещё принимается этой командой, но оно лишено смысловой нагрузки.

Пользователь может выполнять SELECT, INSERT и подобные команды со столбцом таблицы, если он имеет такое право для данного столбца или для всей таблицы. Если назначить пользователю требуемое право на уровне таблицы, а затем отозвать его для одного из столбцов, это не даст эффекта, которого можно было бы ожидать: операция с правами на уровне столбцов не затронет право на уровне таблицы.

Если назначить право доступа к объекту (с помощью GRANT) попытается не владелец объекта, команда завершится ошибкой, если пользователь не имеет никаких прав для этого объекта. Если же пользователь имеет какие-то права, команда будет выполняться, но пользователь сможет давать другим только те права, которые даны ему с правом передачи. Формы GRANT ALL PRIVILEGES будут выдавать предупреждение, если у него вовсе нет таких прав, тогда как другие формы будут выдавать предупреждения, если пользователь не имеет прав распоряжаться именно правами, указанными в команде (эти утверждения применимы и к владельцу объекта, но ему разрешено распоряжаться всеми правами, поэтому такие ситуации невозможны).

Следует отметить, что суперпользователи баз данных могут обращаться к любым объектам, вне зависимости от наличия каких-либо прав. Это сравнимо с привилегиями пользователя root в системе Unix. И так же, как root, роль суперпользователя следует использовать только когда это абсолютно необходимо.

Если суперпользователь решит выполнить команду GRANT или REVOKE, она будет выполнена, как если бы её выполнял владелец заданного объекта. В частности, права, назначенные такой командой, будут представлены как права, назначенные владельцем объекта (если так же установить членство в роли, оно будет представлено как назначенное самой ролью).

GRANT и REVOKE также могут быть выполнены ролью, которая не является владельцем заданного объекта, но является членом роли-владельца, либо членом роли, имеющей права WITH GRANT OPTION для этого объекта. В этом случае права будут записаны как назначенные ролью, которая действительно владеет объектом, либо имеет право WITH GRANT OPTION. Например, если таблица t1 принадлежит роли g1, членом которой является u1, то u1 может дать права на использование t1 роли u2, но эти права будут представлены, как назначенные непосредственно ролью g1. Отозвать эти права позже сможет любой член роли g1.

Если роль, выполняющая команду GRANT, получает требуемое право по нескольким путям членства, какая именно роль будет выбрана в качестве назначающей право, не определено. Если это важно, в таких случаях рекомендуется воспользоваться командой SET ROLE и переключиться на роль, которую хочется видеть в качестве выполняющей GRANT.

При назначении прав для доступа к таблице они автоматически не распространяются на последовательности, используемые этой таблицей, в том числе, на последовательности, связанные со столбцами SERIAL. Права доступа к последовательностям нужно назначать отдельно.

Чтобы получить информацию о существующих правах, назначенных для таблиц и столбцов, воспользуйтесь командой \dp в [psql](https://postgrespro.ru/docs/postgresql/9.6/app-psql). Например:

=> \dp mytable

Access privileges

Schema | Name | Type | Access privileges | Column access privileges

--------+---------+-------+-----------------------+--------------------------

public | mytable | table | miriam=arwdDxt/miriam | col1:

: =r/miriam : miriam\_rw=rw/miriam

: admin=arw/miriam

(1 row)

Записи, выводимые командой \dp, интерпретируются так:

имя\_роли=xxxx -- права, назначенные роли

=xxxx -- права, назначенные PUBLIC

r -- SELECT ("read", чтение)

w -- UPDATE ("write", запись)

a -- INSERT ("append", добавление)

d -- DELETE

D -- TRUNCATE

x -- REFERENCES

t -- TRIGGER

X -- EXECUTE

U -- USAGE

C -- CREATE

c -- CONNECT

T -- TEMPORARY

arwdDxt -- ALL PRIVILEGES (все права для таблиц; для других объектов другие)

\* -- право передачи заданного права

/yyyy -- роль, назначившая это право

В примере выше показано, что увидит пользователь miriam, если создаст таблицу mytable и выполнит:

GRANT SELECT ON mytable TO PUBLIC;

GRANT SELECT, UPDATE, INSERT ON mytable TO admin;

GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam\_rw;

Для других объектов есть другие команды \d, которые также позволяют просмотреть назначенные права.

Если столбец «Права доступа» для данного объекта пуст, это значит, что для объекта действуют стандартные права (то есть столбец прав содержит NULL). Права по умолчанию всегда включают все права для владельца и могут также включать некоторые права для PUBLIC в зависимости от типа объекта, как разъяснялось выше. Первая команда GRANT или REVOKE для объекта приводит к проявлению записи прав по умолчанию (например, {miriam=arwdDxt/miriam}), а затем изменяет эту запись в соответствии с заданным запросом. Подобным образом, строки, показанные в столбце «Права доступа к столбцам», выводятся только для столбцов с нестандартными правами доступа. (Заметьте, что в данном контексте под «стандартными правами» всегда подразумевается встроенный набор прав, предопределённый для типа объекта. Если с объектом связан набор прав по умолчанию, полученный после изменения в результате ALTER DEFAULT PRIVILEGES, изменённые права будут всегда выводиться явно, показывая эффект команды ALTER.)

Право распоряжения правами, которое имеет владелец, не отмечается в выводимой сводке. Знаком \* отмечаются только те права с правом передачи, которые были явно назначены кому-либо.

Более подробная информация: [PostgreSQL : Документация: 9.6: GRANT](https://postgrespro.ru/docs/postgresql/9.6/sql-grant.html)

#### 5. Оборудование

Оборудование: персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

**6. Задание на работу**

1. Создайте три роли (которые в PostgreSQL называются «групповые роли»).

Первая роль – общая. Назначьте этой роли привилегии, необходимые обеим группам пользователей в вашей системе.

Две другие – специальные роли, соответствующие группам пользователям (локальным областям) вашей базы данных. Эти роли должны унаследовать привилегии общей роли с добавлением специальных привилегий, характерных для соответствующей группы пользователей.

Назначьте созданные специальные роли пользователям.

2. Приведите SQL запросы на создание ролей, назначение привилегий, присвоение ролей пользователям, а также экспериментально дать ответы на следующие вопросы:

- что будет, если назначить роль пользователю, а затем изменить состав привилегий в данной роли? Отразится ли это на возможностях пользователя?

- что будет, если назначить специальную роль пользователю, а затем исключить из специальной роли общую роль?

- что будет, если отобрать у пользователя привилегию, которая была передана ему неявно через роль.

**7. Контрольные вопросы**

1. Понятие «Роль». Область действия роли.
2. Как можно создать, удалить учетную запись пользователя, редактировать ее параметры?
3. Укажите все способы наделения пользователя привилегиями. Какой пользователь может назначить другому пользователю роль?
4. Как полностью/частично аннулировать пользователю его привилегии? Какой пользователь может это сделать?
5. Приведите примеры использования оператора CONNECT средствами встроенного SQL.

**Лабораторная работа №16**

**Избирательный доступ к данным**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение избирательного доступа к данным.

#### 2.Порядок выполнения работы

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

#### 3. Оформление отчета

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

В дополнение к стандартной системе привилегий SQL, доступной через GRANT, таблицы могут иметь политики безопасности строк, которые ограничивают для каждого пользователя, Какие строки могут быть возвращены обычными запросами или вставлены, обновлены или удалены командами изменения данных. Эта функция также известна как безопасность на уровне строк. По умолчанию в таблицах нет политик, поэтому, если пользователь имеет права доступа к таблице в соответствии с системой привилегий SQL, все строки в ней одинаково доступны для запроса или обновления.

Когда безопасность строк включена в таблице (ALTER TABLE ... ENABLE ROW LEVEL SECURITY), все обычные права доступа к таблице для выбора строк или изменения строк должны быть разрешены политикой безопасности строк (однако владелец таблицы обычно не подпадает под действие политики безопасности строк). Если для таблицы не существует политики, используется политика по умолчанию, то есть строки не отображаются и не могут быть изменены. Операции, применяемые ко всей таблице, такие как усечение и ссылки, не подлежат защите строк.

Политики безопасности строк могут быть специфичными для команд, ролей или для обоих типов. Можно указать политику для применения ко всем командам, а также для SELECT, INSERT, UPDATE, или DELETE. Данной политике может быть назначено несколько ролей, и применяются обычные правила членства и наследования ролей.

Чтобы указать, какие строки являются видимыми или изменяемыми в соответствии с политикой, необходимо выражение, возвращающее логический результат. Это выражение будет вычисляться для каждой строки до любых условий или функций, поступающих из запроса пользователя (единственным исключением из этого правила являются leakproof функции, которые гарантированно не пропускают информацию; оптимизатор может выбрать применение таких функций перед проверкой безопасности строк). Строки, для которых выражение не возвращает true не будет обработано. Отдельные выражения могут быть заданы для обеспечения независимого контроля над строками, которые являются видимыми, и строками, которые могут быть изменены. Выражения политики выполняются как часть запроса и с привилегиями пользователя, выполняющего запрос, хотя функции определения безопасности могут использоваться для доступа к данным, недоступным вызывающему пользователю.

Суперпользователи и роли с атрибутом BYPASSRLS всегда обходят систему безопасности строк при обращении к таблице. Владельцы таблиц обычно также обходят безопасность строк, хотя владелец таблицы может выбрать защиту строк с помощью ALTER TABLE ... ENABLE ROW LEVEL SECURITY.

Включение и отключение защиты строк, а также добавление политик в таблицу всегда является привилегией только владельца таблицы.

Политики создаются с помощью команды [CREATE POLICY](https://www.postgresql.org/docs/9.5/sql-createpolicy.html), изменяются с помощью команды [ALTER POLICY](https://www.postgresql.org/docs/9.5/sql-alterpolicy.html)и удаляются с помощью команды [DROP POLICY](https://www.postgresql.org/docs/9.5/sql-droppolicy.html). Чтобы включить или отключить защиту строк для данной таблицы, используйте команду [ALTER TABLE](https://www.postgresql.org/docs/9.5/sql-altertable.html).

Каждая политика имеет имя, и для таблицы можно определить несколько политик. Поскольку политики зависят от таблицы, каждая политика для таблицы должна иметь уникальное имя. В разных таблицах могут быть политики с одинаковыми именами.

Когда несколько политик применяются к данному запросу, они объединяются с помощью OR, чтобы строка была доступна, если это позволяет какая-либо политика. Это аналогично правилу, согласно которому данная роль имеет привилегии всех ролей, членом которых она является.

В качестве простого примера рассмотрим, как создать политику для отношения «учетная запись», чтобы разрешить доступ к строкам только членам роли «менеджеры» и только строкам их учетных записей:

CREATE TABLE accounts (manager text, company text, contact\_email text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account\_managers ON accounts TO managers

USING (manager = current\_user);

CREATE POLICY user\_policy ON users

USING (user\_name = current\_user);

Ограничение относится к строкам, выбранных командой (так менеджер не может выбрать, обновить или удалить существующие строки, принадлежащие другой менеджер) и строк, измененных команд (так строки, принадлежащие другой менеджер не может быть создан посредством вставки или обновления).

Если роль не указана или используется специальное имя пользователя PUBLIC, политика применяется ко всем пользователям системы. Чтобы разрешить всем пользователям доступ только к их собственной строке в таблице users, можно использовать простую политику:

CREATE POLICY user\_sel\_policy ON users

FOR SELECT

USING (true);

CREATE POLICY user\_mod\_policy ON users

USING (user\_name = current\_user);

Чтобы использовать другую политику для строк, добавляемых в таблицу по сравнению с видимыми строками, можно объединить несколько политик. Эта пара политик позволит всем пользователям просматривать все строки в таблице users, но изменять только свои собственные:

CREATE POLICY user\_sel\_policy ON users

FOR SELECT

USING (true);

CREATE POLICY user\_mod\_policy ON users

USING (user\_name = current\_user);

В команде SELECT эти две политики объединяются с помощью OR, при этом результирующий эффект заключается в том, что могут быть выбраны все строки. В других типах команд применяется только вторая политика, так что эффекты остаются прежними.

Безопасность строк также можно отключить с помощью команды ALTER TABLE. Отключение защиты строк не удаляет политики, определенные в таблице; они просто игнорируются. Затем все строки таблицы становятся видимыми и изменяемыми в соответствии со стандартной системой привилегий SQL.

Проверки ссылочной целостности, такие как ограничения по уникальному или первичному ключу и ссылки на внешний ключ, всегда обходят защиту строк для обеспечения целостности данных. При разработке схем и политик уровня строк необходимо соблюдать осторожность, чтобы избежать утечек информации «скрытого канала» через такие проверки ссылочной целостности.

В некоторых контекстах важно убедиться, что безопасность строк не применяется. Например, при резервном копировании это может привести к катастрофическим последствиям, если безопасность строк приведет к тому, что некоторые строки будут опущены из резервной копии. В такой ситуации параметр конфигурации row\_security можно отключить . Это само по себе не обходит безопасность строк; то, что он делает, вызывает ошибку, если результаты любого запроса будут отфильтрованы политикой. Затем можно исследовать и устранить причину ошибки.

В приведенных выше примерах выражения политики учитывают только текущие значения в строке для доступа или обновления. Это самый простой и наиболее эффективный случай; когда это возможно, лучше всего разрабатывать приложения безопасности строк для работы таким образом. Если необходимо проконсультироваться с другими строками или другими таблицами для принятия решения о политике, это можно сделать с помощью sub-SELECTs или функций, содержащих SELECTs, в выражениях политики. Однако имейте в виду, что такой доступ может создать условия гонки, которые могут позволить утечку информации, если не принять меры предосторожности. В качестве примера рассмотрим следующую структуру таблицы:

-- definition of privilege groups

CREATE TABLE groups (group\_id int PRIMARY KEY,

group\_name text NOT NULL);

INSERT INTO groups VALUES

(1, 'low'),

(2, 'medium'),

(5, 'high');

GRANT ALL ON groups TO alice; -- alice is the administrator

GRANT SELECT ON groups TO public;

-- definition of users' privilege levels

CREATE TABLE users (user\_name text PRIMARY KEY,

group\_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES

('alice', 5),

('bob', 2),

('mallory', 2);

GRANT ALL ON users TO alice;

GRANT SELECT ON users TO public;

-- table holding the information to be protected

CREATE TABLE information (info text,

group\_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES

('barely secret', 1),

('slightly secret', 2),

('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- a row should be visible to/updatable by users whose security group\_id is

-- greater than or equal to the row's group\_id

CREATE POLICY fp\_s ON information FOR SELECT

USING (group\_id <= (SELECT group\_id FROM users WHERE user\_name = current\_user));

CREATE POLICY fp\_u ON information FOR UPDATE

USING (group\_id <= (SELECT group\_id FROM users WHERE user\_name = current\_user));

-- we rely only on RLS to protect the information table

GRANT ALL ON information TO public;

Теперь предположим, что alice хочет изменить «слегка секретную» информацию, но решает, что mallory не следует доверять новому содержанию этой строки, поэтому она делает:

BEGIN;

UPDATE users SET group\_id = 1 WHERE user\_name = 'mallory';

UPDATE information SET info = 'secret from mallory' WHERE group\_id = 2;

COMMIT;

Это выглядит безопасным; нет окна, в котором mallory должна видеть строку «секрет от mallory». Однако здесь есть расовые условия. Если mallory одновременно делает, скажем,

SELECT \* FROM information WHERE group\_id = 2 FOR UPDATE;

и ее транзакция в режиме Read COMMITTED, она может видеть «секрет от mallory». Это происходит, если ее транзакция достигает информационной строки сразу после alice. Он блокирует ожидание транзакции Алисы для фиксации, а затем извлекает обновленное содержимое строки благодаря предложению FOR UPDATE. Однако он не извлекает обновленную строку для неявного выбора от пользователей, поскольку этот подвыборка не была для обновления; вместо этого строка пользователей считывается с моментальным снимком, сделанным в начале запроса. Поэтому выражение политики проверяет старое значение mallory's уровень привилегий и позволяет ей видеть обновленную строку.

Существует несколько способов обойти эту проблему. Один простой ответ – использовать SELECT ... FOR SHARE in sub-SELECTs Однако это требует предоставления привилегий обновления в таблице ссылок (здесь пользователи) для затронутых пользователей, что может быть нежелательным. Кроме того, одновременное использование блокировок общего доступа к строкам в ссылочной таблице может создать проблемы с производительностью, особенно если ее обновления происходят часто. Другим решением, практичным, если обновления ссылочной таблицы происходят нечасто, является монопольная блокировка ссылочной таблицы при ее обновлении, чтобы никакие параллельные транзакции не могли проверять старые значения строк. Или можно просто дождаться завершения всех параллельных транзакций после фиксации обновления ссылочной таблицы и внесения изменений, зависящих от новой ситуации безопасности.

Более подробная информация: [PostgreSQL: Documentation: 9.5: Row Security Policies](https://www.postgresql.org/docs/9.5/static/ddl-rowsecurity.html)

#### 5. Оборудование

Оборудование: персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

**6. Задание на работу**

В этой работе должен быть реализован принцип защиты данных на уровне строк. Каждая строке и каждому пользователю назначается метка. Пользователь может получить строку таблицы только если его метка и метка строки соответствуют.

Создайте:

1. Таблицу меток.

2. Таблицу соответствия имени пользователя и метки.

3. Дополнительной столбец в защищаемой таблице, содержащий метку строки.

4. Представление, выбирающее для пользователя только принадлежащих ему данных. Для этого необходимо использовать ключевое слово USER (оно содержит имя ткущего подключенного пользователя).

5. Триггер INSERT для представления, заполняющий дополнительный столбец в защищаемой таблице в зависимости от метки текущего пользователя.

6. А потом сделайте то же самое с использованием встроенной в PostgreSQL+ защиту на уровне строк.

В работе приведите:

1. Запросы на создание необходимых объектов.

2. Проверочные запросы от имени двух разных пользователей.

**7. Контрольные вопросы**

1. Какие подходы к вопросу обеспечения безопасности данных поддерживаются в современных СУБД.
2. К какой группе по умолчанию относится вновь созданный пользователь?
3. Какой оператор используется для предоставления привилегий? Приведите примеры.
4. Какой оператор используется для отмены ранее назначенных привилегий? Приведите примеры.
5. Что такое ограничения полей, ограничение таблиц? Как они используются и для чего?

**Лабораторная работа №17**

**Транзакции и блокировки**

**1. Цель и задачи работы**

Целью лабораторной работы является изучение понятия целостности базы данных, выполнение транзакций.

#### 2.Порядок выполнения работы

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

#### 3. Оформление отчета

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

PostgreSQL предоставляет разработчикам богатый набор средств для управления конкурентным доступом к данным. Внутри он поддерживает целостность данных, реализуя модель MVCC (Multiversion Concurrency Control, Многоверсионное управление конкурентным доступом). Это означает, что каждый SQL-оператор видит снимок данных (версию базы данных) на определённый момент времени, вне зависимости от текущего состояния данных. Это защищает операторы от несогласованности данных, возможной, если другие конкурирующие транзакции внесут изменения в те же строки данных, и обеспечивает тем самым изоляцию транзакций для каждого сеанса баз данных. MVCC, отходя от методик блокирования, принятых в традиционных СУБД, снижает уровень конфликтов блокировок и таким образом обеспечивает более высокую производительность в многопользовательской среде.

Основное преимущество использования модели MVCC по сравнению с блокированием заключается в том, что блокировки MVCC, полученные для чтения данных, не конфликтуют с блокировками, полученными для записи, и поэтому чтение никогда не мешает записи, а запись чтению. PostgreSQL гарантирует это даже для самого строгого уровня изоляции транзакций, используя инновационный уровень изоляции SSI (Serializable Snapshot Isolation, Сериализуемая изоляция снимков).

Для приложений, которым в принципе не нужна полная изоляция транзакций и которые предпочитают явно определять точки конфликтов, в PostgreSQLтакже есть средства блокировки на уровне таблиц и строк. Однако при правильном использовании MVCC обычно обеспечивает лучшую производительность, чем блокировки. Кроме этого, приложения могут использовать рекомендательные блокировки, не привязанные к какой-либо одной транзакции.

Стандарт SQL определяет четыре уровня изоляции транзакций. Наиболее строгий из них – сериализуемый, определяется одним абзацем, говорящем, что при параллельном выполнении несколько сериализуемых транзакций должны гарантированно выдавать такой же результат, как если бы они запускались по очереди в некотором порядке. Остальные три уровня определяются через описания особых явлений, которые возможны при взаимодействии параллельных транзакций, но не допускаются на определённом уровне. Как отмечается в стандарте, из определения сериализуемого уровня вытекает, что на этом уровне ни одно из этих явлений не возможно. (В самом деле – если эффект транзакций должен быть тем же, что и при их выполнении по очереди, как можно было бы увидеть особые явления, связанные с другими транзакциями?)

Стандарт описывает следующие особые условия, недопустимые для различных уровней изоляции:

«грязное» чтение: транзакция читает данные, записанные параллельной незавершённой транзакцией.

неповторяемое чтение: транзакция повторно читает те же данные, что и раньше, и обнаруживает, что они были изменены другой транзакцией (которая завершилась после первого чтения).

фантомное чтение: транзакция повторно выполняет запрос, возвращающий набор строк для некоторого условия, и обнаруживает, что набор строк, удовлетворяющих условию, изменился из-за транзакции, завершившейся за это время.

аномалия сериализации: результат успешной фиксации группы транзакций оказывается несогласованным при всевозможных вариантах исполнения этих транзакций по очереди.

Уровни изоляции транзакций, описанные в стандарте SQL и реализованные в PostgreSQL, описываются в Таблице 1.

Таблица 1. Уровни изоляции транзакций

| Уровень изоляции | «Грязное» чтение | Неповторяемое чтение | Фантомное чтение | Аномалия сериализации |
| --- | --- | --- | --- | --- |
| Read uncommited (Чтение незафиксированных данных) | Допускается, но не в PG | Возможно | Возможно | Возможно |
| Read committed (Чтение зафиксированных данных) | Невозможно | Возможно | Возможно | Возможно |
| Repeatable read (Повторяемое чтение) | Невозможно | Невозможно | Допускается, но не в PG | Возможно |
| Serializable (Сериализуемость) | Невозможно | Невозможно | Невозможно | Невозможно |

В PostgreSQL вы можете запросить любой из четырёх уровней изоляции транзакций, однако внутри реализованы только три различных уровня, то есть режим Read Uncommitted в PostgreSQL действует как Read Committed. Причина этого в том, что только так можно сопоставить стандартные уровни изоляции с реализованной в PostgreSQL архитектурой многоверсионного управления конкурентным доступом.

В этой таблице также показано, что реализация Repeatable Read в PostgreSQL не допускает фантомное чтение. Стандарт SQL допускает возможность более строгого поведения: четыре уровня изоляции определяют только, какие особые условия не должны наблюдаться, но не какие *обязательно должны*. Поведение имеющихся уровней изоляции подробно описывается в следующих подразделах.

Для выбора нужного уровня изоляции транзакций используется команда [SET TRANSACTION](https://postgrespro.ru/docs/postgresql/9.5/sql-set-transaction).

Для управления параллельным доступом к данным в таблицах PostgreSQL предоставляет несколько режимов явных блокировок. Эти режимы могут применяться для блокировки данных со стороны приложения в ситуациях, когда MVCC не даёт желаемый результат. Кроме того, большинство команд PostgreSQL автоматически получают блокировки соответствующих режимов, защищающие от удаления или изменения задействованных таблиц, несовместимого с характером выполняемой команды. (Например, TRUNCATE не может безопасно выполняться одновременно с другими операциями с этой таблицей, так что во избежание конфликта эта команда получает исключительную блокировку для данной таблицы.)

Список текущих активных блокировок на сервере можно получить, прочитав системное представление [pg\_locks](https://postgrespro.ru/docs/postgresql/9.5/view-pg-locks).

Более подробная информация: [PostgreSQL : Документация: 9.5: Управление конкурентным доступом](https://postgrespro.ru/docs/postgresql/9.5/mvcc.html)

#### 5. Оборудование

персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

**6. Задание на работу**

1. Продемонстрируйте работу команд COMMIT, ROLLBACK, SAVEPOINT, ROLLBACK TO.

2. Подключитесь к СУБД двумя клиентами. Продемонстрируйте работу всех уровней изоляции. В работе должны быть показаны явные отличия этих уровней изоляции.

3. Покажите, как работают блокировки уровня таблиц и уровня строк.

4. Вызовите deadlock (взаимную блокировку)

Работу следует выполнять в вашей предметной области. Для каждого примера вы должны дать понятное описание на языке предметной области: что за пользователи подключены к системе и что они хотят сделать.

**7. Контрольные вопросы**

1. Дайте понятие транзакции.
2. Для чего используется ограничения целостности БД?
3. Назовите условия для обеспечения контроля целостности?
4. Какие инструкции языка SQL используются для выполнения транзакций?
5. Опишите модель автоматического выполнения транзакций.
6. Опишите модель управляемого выполнения транзакций.

**Лабораторная работа №18**

**Аудит**

**1. Цель и задачи работы**

Целью работы является практическое знакомство с аудитом и получение навыков по детальной настройке аудита и мониторингу событий.

#### 2.Порядок выполнения работы

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

#### 3. Оформление отчета

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

Ниже приведен пример универсальной триггерной функции, используемой для записи изменений в таблицы журнала аудита. Он будет записывать старые и новые записи, затронутую таблицу, пользователя, внесшего изменения, и временную метку для каждого изменения.

*-- create a schema named "audit"*

**CREATE** schema audit;

**REVOKE** **CREATE** **ON** schema audit **FROM** public;

**CREATE** **TABLE** audit.logged\_actions (

schema\_name text **NOT** **NULL**,

**TABLE\_NAME** text **NOT** **NULL**,

user\_name text,

action\_tstamp **TIMESTAMP** **WITH** **TIME** zone **NOT** **NULL** **DEFAULT** **CURRENT\_TIMESTAMP**,

action TEXT **NOT** **NULL** **CHECK** (action **IN** ('I','D','U')),

original\_data text,

new\_data text,

query text

) **WITH** (fillfactor=100);

**REVOKE** **ALL** **ON** audit.logged\_actions **FROM** public;

*-- You may wish to use different permissions; this lets anybody*

*-- see the full audit data. In Pg 9.0 and above you can use column*

*-- permissions for fine-grained control.*

**GRANT** **SELECT** **ON** audit.logged\_actions **TO** public;

**CREATE** **INDEX** logged\_actions\_schema\_table\_idx

**ON** audit.logged\_actions(((schema\_name||'.'||**TABLE\_NAME**)::TEXT));

**CREATE** **INDEX** logged\_actions\_action\_tstamp\_idx

**ON** audit.logged\_actions(action\_tstamp);

**CREATE** **INDEX** logged\_actions\_action\_idx

**ON** audit.logged\_actions(action);

*--*

*-- Now, define the actual trigger function:*

*--*

CREATE OR REPLACE FUNCTION audit.if\_modified\_func() RETURNS TRIGGER AS $body$

DECLARE

v\_old\_data TEXT;

v\_new\_data TEXT;

BEGIN

*/\* If this actually for real auditing (where you need to log EVERY action),*

*then you would need to use something like dblink or plperl that could log outside the transaction,*

*regardless of whether the transaction committed or rolled back.*

*\*/*

*/\* This dance with casting the NEW and OLD values to a ROW is not necessary in pg 9.0+ \*/*

IF (TG\_OP = 'UPDATE') THEN

v\_old\_data := ROW(OLD.\*);

v\_new\_data := ROW(NEW.\*);

INSERT INTO audit.logged\_actions (schema\_name,table\_name,user\_name,action,original\_data,new\_data,query)

VALUES (TG\_TABLE\_SCHEMA::TEXT,TG\_TABLE\_NAME::TEXT,session\_user::TEXT,substring(TG\_OP,1,1),v\_old\_data,v\_new\_data, current\_query());

RETURN NEW;

ELSIF (TG\_OP = 'DELETE') THEN

v\_old\_data := ROW(OLD.\*);

INSERT INTO audit.logged\_actions (schema\_name,table\_name,user\_name,action,original\_data,query)

VALUES (TG\_TABLE\_SCHEMA::TEXT,TG\_TABLE\_NAME::TEXT,session\_user::TEXT,substring(TG\_OP,1,1),v\_old\_data, current\_query());

RETURN OLD;

ELSIF (TG\_OP = 'INSERT') THEN

v\_new\_data := ROW(NEW.\*);

INSERT INTO audit.logged\_actions (schema\_name,table\_name,user\_name,action,new\_data,query)

VALUES (TG\_TABLE\_SCHEMA::TEXT,TG\_TABLE\_NAME::TEXT,session\_user::TEXT,substring(TG\_OP,1,1),v\_new\_data, current\_query());

RETURN NEW;

ELSE

RAISE WARNING '[AUDIT.IF\_MODIFIED\_FUNC] - Other action occurred: %, at %',TG\_OP,now();

RETURN NULL;

END IF;

 EXCEPTION

WHEN data\_exception THEN

RAISE WARNING '[AUDIT.IF\_MODIFIED\_FUNC] - UDF ERROR [DATA EXCEPTION] - SQLSTATE: %, SQLERRM: %',SQLSTATE,SQLERRM;

RETURN NULL;

WHEN unique\_violation THEN

RAISE WARNING '[AUDIT.IF\_MODIFIED\_FUNC] - UDF ERROR [UNIQUE] - SQLSTATE: %, SQLERRM: %',SQLSTATE,SQLERRM;

RETURN NULL;

WHEN OTHERS THEN

RAISE WARNING '[AUDIT.IF\_MODIFIED\_FUNC] - UDF ERROR [OTHER] - SQLSTATE: %, SQLERRM: %',SQLSTATE,SQLERRM;

RETURN NULL;

END;

$body$

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = pg\_catalog, audit;

*--*

*-- To add this trigger to a table, use:*

*-- CREATE TRIGGER tablename\_audit*

*-- AFTER INSERT OR UPDATE OR DELETE ON tablename*

*-- FOR EACH ROW EXECUTE PROCEDURE audit.if\_modified\_func();*

*--*

Существует ограничение на то, что вы можете сделать с триггерами для аудита в Pg. Однако большинство вещей, которые вы не можете проверить с помощью триггера, все еще можно проследить в системных журналах и, что более важно, может и должно быть вне разрешений обычных пользователей.

Если ваше производственное приложение входит в базу данных с ролью, не имеющей прав суперпользователя или прав CREATEUSER CREATEDB, если эта роль не владеет таблицами, и если вы убедитесь, что у нее есть только минимальные права, необходимые для предоставления ей, все будет в порядке.

Этот триггер аудита не фиксирует действие выбора. Единственный способ аудита SELECT в PostgreSQL ­– через системные журналы, так как триггеры SELECT не поддерживаются, а аудит SELECT не будет работать без автономных транзакций в триггерах, чтобы предотвратить потерю данных аудита при откате.

Более подробная информация: [Audit trigger - PostgreSQL wiki](https://wiki.postgresql.org/wiki/Audit_trigger), [Audit trigger 91plus - PostgreSQL wiki](https://wiki.postgresql.org/wiki/Audit_trigger_91plus).

#### 5. Оборудование

персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

**6. Задание на работу**

1. Реализуйте два варианта аудита событий в базе данных на основе двух представленных ссылок.

2. Разберитесь с описанием. Приведите последовательность выполняемых операций по созданию системы. Покажите на примерах все реализованные вами возможности аудита.

3. Дайте каждому примеру краткий, но емкий комментарий.

Эта работа, как и все предыдущие, должна выполняться в вашей предметной области!

**7. Контрольные вопросы**

1. Зачем используется аудит?
2. Когда пользователям следует подвергаться аудиту?
3. Как могут быть проконтролированы пользователи?
4. Какие проблемы могут возникнуть с производительностью и сложностью?
5. Для чего используется временная метка?

**Лабораторная работа №19**

**Древовидные структуры**

**1. Цель и задачи работы**

Целью работы является практическое знакомство с древовидными структурами.

#### 2.Порядок выполнения работы

- ознакомится с теоретическими сведениями;

- выполнить задание;

- оформить отчет;

- ответить на контрольные вопросы, заданные преподавателем.

#### 3. Оформление отчета

Отчет должен содержать: титульный лист, цель работы, описание пунктов выполнения лабораторной работы в соответствии с заданием, ответы на контрольные вопросы и выводы по работе.

**4. Теоретические сведения**

Иерархические запросы (ИЗ) в PostgreSQL реализовано на базе стандратной SQL clause WITH. Не рекурсивный WITH позволяет удешевить повторяющиеся подзапросы, разделить сложный запрос на несколько меньших, является удобным так сказать ярлыком для обращения к подзапросу и само по себе удобно в плане экономии времени при написании кода. В примере ниже удалось избежать использования подзапроса в WHERE за счет применения временой таблицы top\_regions сформированой специально для этого запроса.

1. WITH regional\_sales AS (
2. SELECT region, SUM(amount) AS total\_sales
3. FROM orders
4. GROUP BY region
5. ), top\_regions AS (
6. SELECT region
7. FROM regional\_sales
8. WHERE total\_sales > (SELECT SUM(total\_sales)/10 FROM regional\_sales)
9. )
10. SELECT region,
11. product,
12. SUM(quantity) AS product\_units,
13. SUM(amount) AS product\_sales
14. FROM orders
15. WHERE region IN (SELECT region FROM top\_regions)
16. GROUP BY region, product;

Добавление необязательного оператора RECURSEVE позволяет запросу в Postgre обращаться к своим же выходным данным. Алгоритм запроса должен состоять из двух частей, первая часть это основа, обычно возвращающий одну строку с исходной точкой иерархии или части иерархии. То есть место в иерархии откуда будет начат отсчет (например, корень), и вторя рекурсивная часть которая будет связываться с временной таблицей которую мы объявили после WITH. Объединяются первая и вторая части оператором UNION или UNION ALL. Создадим таблицу в которой будет описана структура одной компании:

CREATE TABLE KPO

(

"ID" character varying(55),

"DESCRIPTION" character varying(255),

"PARENT" character varying(55

) ;

после внесения туда данных:

Select \* from kpo

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **PARENT** |
| == ===== | ================================ | ======= |
| KPO | KARACHAGANAK PETROLEUM OPERATING | {null} |
| AKSAY | AKSAY | KPO |
| URALSK | KPO | KPO |
| LONDON | LONDON | KPO |
| KPC | KPC | AKSAY |
| U2 | UNIT-2 | AKSAY |
| U3 | UNIT-3 | AKSAY |
| PROD | PRODACTION | KPC |
| MAINT | MAINTENANCE | AKSAY |
| CMMS | CMMS TEAM | MAINT |

Теперь сам рекурсивный запрос:

1. WITH RECURSIVE temp1 ( "ID","PARENT","DESCRIPTION",PATH, LEVEL ) AS (
2. SELECT T1."ID",T1."PARENT", T1."DESCRIPTION", CAST (T1."ID" AS VARCHAR (50)) asPATH, 1
3. FROM KPO T1 WHERE T1."PARENT" IS NULL
4. union
5. select T2."ID", T2."PARENT", T2."DESCRIPTION", CAST ( temp1.PATH ||'->'|| T2."ID" ASVARCHAR(50)) ,LEVEL + 1
6. FROM KPO T2 INNER JOIN temp1 ON( temp1."ID"= T2."PARENT")      )
7. select \* from temp1 ORDER BY PATH LIMIT 100

Первая часть (строки 2-3) возвращает во временную таблицу первую строку в данном случае корневую запись нашей структуры, от которой будет начинаться отсчет в нашей иерархии. Вторая часть (строки 4-5) добавляет в эту же временную таблицу записи связанные с уже содержащейся в temp1 строкой через JOIN (ID = PARENT) и так до конца пока все листья нашего ROOTa не окажутся в temp1.

Так же в данном примере была сымитирована Ораколавская функция sys\_connect\_by\_path.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **«ID»** | **«PARENT»** | **«DESCRIPTION»** | **«path»** | **«level»** |
| KPO |  | KARACHAGANAK PETROLEUM OPERATING | KPO | 1 |
| AKSAY | KPO | AKSAY | KPO->AKSAY | 2 |
| KPC | AKSAY | KPC | KPO->AKSAY->KPC | 3 |
| PROD | KPC | PRODAUCTION | KPO->AKSAY->KPC->PROD | 4 |
| MAINT | AKSAY | MAINTENANCE | KPO->AKSAY->MAINT | 3 |
| CMMS | MAINT | CMMS TEAM | KPO->AKSAY->MAINT->CMMS | 4 |
| U2 | AKSAY | UNIT-2 | KPO->AKSAY->U2 | 3 |
| U3 | AKSAY | UNIT-3 | KPO->AKSAY->U3 | 3 |
| LONDON | KPO | LONDON | KPO->LONDON | 2 |
| URALSK | KPO | URALSK | KPO->URALSK | 2 |

В Postgre нет встроенной проверки на зацикливание, поэтому если данные получили от тех, кто занимался непосредственно созданием структуры в Excel, то необходимо проверить эту структуру на целостность. Иногда достаточно использовать UNION вместо UNION ALL, но это только иногда. Если в первой части задали отправную точку в иерархии, и если даже где-то в иерархии есть обрывы в принципе запустив вышеупомянутый квери ошибки не будет, просто строки «отщипенцы» будут проигнорированы. Но нам же надо знать где ошибка, и реализовать это можно внедрив дополнительную проверку перед выполнением UNION.

1. WITH RECURSIVE temp1 ( "ID","PARENT","DESCRIPTION",PATH, LEVEL, cycle ) AS (
2. SELECT T1."ID",T1."PARENT", T1."DESCRIPTION", cast (array[T1."ID"] as varchar(100)[]) , 1 , FALSE
3. FROM KPO T1
4. union all
5. select T2."ID", T2."PARENT", T2."DESCRIPTION", cast(temp1.PATH || T2."ID" asvarchar(100) []) ,LEVEL + 1 ,
6. T2."ID" = any (temp1.PATH)
7. FROM KPO T2 INNER JOIN temp1 ON( temp1."ID"= T2."PARENT") AND NOT CYCLE     )
9. select \* from temp1 WHERE CYCLE IS TRUE LIMIT 100;

Здесь создается такое же поле Path но уже все «предшествующие родители» организованны в массиве, что дает сравнивать каждый новый “ID” на дубликат, и если в массиве уже есть такая запись тогда во временную таблицу строка заносится с флагом и в следующий проход уже не используют эту строку для поиска потомков, благодаря этому избегается зацикливание (union all… WHERE … AND NOT CYCLE).

Более подробная информация: [Рекурсивные запросы в PostgreSQL (WITH RECURSIVE) / Хабрахабр](https://habrahabr.ru/post/269497/), [Запросы WITH (Общие табличные выражения) | PostgreSQL](http://postgresql.ru.net/manual/queries-with.html), [Рекурсивные (Иерархические) запросы в PostgreSQL / Хабрахабр](https://habrahabr.ru/post/73700/), [Nested Sets + PostgreSQL TRIGGER / Хабрахабр](https://habrahabr.ru/post/63416/), http://www.ibase.ru/treedb/

#### 5. Оборудование

персональный компьютер с установленной операционной системой Windows XP/7/8, браузер (Например, InternetExplorer, GoogleChrome, Opera), СУБД PostgreSQL.

**6. Задание на работу**

1. Опишите, для чего в вашей предметной области могла бы понадобиться древовидная структура.
2. Создайте древовидную структуру (таблицу, содержащую внешний ключ, ссылающийся на эту же таблицу). Наполните ее данными. Создайте рекурсивный вопрос, выводящий данные в соответствии с их иерархией. Проанализируйте зависимость времени выполнения запроса от количества записей в таблице (определите для вашего компьютера, при каком количестве записей запрос будет выполняться 0,1 секунду, 1 секунду и 10 секунд).
3. Создайте структуру для хранения древовидных данных типа Nested Sets и все необходимые триггеры. Покажите на примерах, как работает эта структура.
4. Проанализируйте зависимость времени добавления новых данных от числа строк в таблице (определите для вашего компьютера, при каком количестве записей добавление новых данных будет выполняться 0.1 секунду, 1 секунду и 10 секунд).

**7. Контрольные вопросы**

1. Как представить дерево данных в Postgres?

2. Как эффективно получить произвольный узел и всех его потомков (и потомков потомков)?

3. Как выглядит общая схема рекурсивного запроса?