Теория баз данных

Лекция 4. Язык SQL

Е. П. Моргунов
Сибирский федеральный университет
г. Красноярск
Институт космических и информационных технологий
emorgunov@mail.ru

Версия от 14.03.2018

4.1. Введение

- Язык SQL был разработан в лаборатории IBM Research в начале 1970-х годов. Первой серьезной реализацией этого языка был продукт-прототип System R компании IBM. Первой коммерческой реализацией СУБД, основанной на SQL, была СУБД Oracle (конец 1970-х гг.).
- Разработчики языка: **Donald D. Chamberlin и Raymond F. Boyce** (1947–1974).
- Стандарты языка SQL: SQL-86, SQL-89, SQL-92 (SQL2), SQL:1999 (SQL3), SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016.
- К. Дейт считает, что в наше время ни одна реальная СУБД не поддерживает стандарт полностью, а поддерживает «надмножество подмножества» стандарта: большинство СУБД не поддерживают некоторые средства, обусловленные стандартом, и в то же время предлагают другие средства, которые не определены этим стандартом.
- SQL — язык очень большого объема. Его спецификация содержит свыше 2000 страниц, не считая больше 300 страниц исправлений.
- В стандарте SQL есть противоречия.

4.1. Введение (продолжение)

- Некоторый язык принято называть **реляционно полным,** если он по своим возможностям, по крайней мере, не уступает реляционному исчислению.
- Реляционный язык может быть основан как на реляционной алгебре, так и на реляционном исчислении.
- Когда язык SQL только разрабатывался, предполагалось что он будет отличаться как от реляционной алгебры, так и от реляционного исчисления.
- На сегодняшний день ситуация складывается таким образом, что язык SQL в чем-то похож на реляционную алгебру, в чем-то на реляционное исчисление, а в чем-то отличается от них обоих.

4.1. Введение (продолжение)

- Фундаментальным объектом данных в языке SQL является не отношение, а скорее таблица, и таблицы SQL содержат (вообще говоря) *не множества*, а *мультимножества* строк (в мультимножествах допускаются повторения элементов). Таким образом, в языке SQL нарушается *информационный принцип.*
- Одно из следствий этого факта состоит в том, что основные операторы SQL являются не реляционными операторами в полном смысле этого слова, а аналогами реляционных операторов, предназначенных для работы с мультимножествами.
- Другим следствием является то, что теоремы и их побочные результаты, которые являются справедливыми в реляционной модели, не обязательно выполняются в языке SQL.

4.1. Введение (продолжение)

- В языке SQL вместо терминов *отношение* и *переменная отношения* используется термин **таблица,** а вместо терминов *кортеж* и *атрибут* — **строка** и **столбец.**
- В язык SQL не включен прямой аналог операции **реляционного присваивания.** Но эту операцию можно эмулировать, сначала удалив все строки из целевой таблицы, а затем выполнив для нее операции INSERT ... SELECT ...
- Стандарт SQL включает спецификации стандартного каталога, именуемого в нем **информационной** схемой. **Каталог** в языке SQL состоит из дескрипторов (метаданных) для отдельной базы данных, а схема состоит из дескрипторов той части базы данных, которая принадлежит отдельному пользователю. Каждый каталог должен содержать одну и только одну схему с именем INFORMATION_SCHEMA.

4.1. Введение (продолжение)

- Язык SQL первоначально разрабатывался конкретно как **подъязык** данных. Однако после включения в стандарт в конце 1996 года такого средства, как **постоянные хранимые модули SQL** (SQL Persistent Stored Modules — SQL/PSM), стандарт SQL стал полностью поддерживать все вычислительные конструкции. Сейчас в нем предусмотрены процедурные операторы, например, CALL, RETURN, SET, CASE, IF, LOOP, LEAVE, WHILE, REPEAT.
- Язык SQL — это непроцедурный язык, который является стандартным средством работы с данными во всех реляционных СУБД.
- Операторы (команды), написанные на этом языке, лишь указывают СУБД, какой результат должен быть получен, но не описывают процедуру получения этого результата. СУБД сама определяет способ выполнения команды пользователя.

4.1. Введение (продолжение)

- В языке SQL традиционно выделяются группа операторов определения данных (Data Definition Language — DDL), группа операторов манипулирования данными (Data Manipulation Language — DML) и группа операторов, управляющих привилегиями доступа к объектам базы данных (Data Control Language — DCL).
- К операторам DDL относятся команды для создания, изменения и удаления таблиц, представлений и других объектов базы данных.
- К операторам DML относятся команды для выборки строк из таблиц, вставки строк в таблицы, обновления и удаления строк.
- Тексты запросов и команд SQL не чувствительны к регистру символов (case insensitive). Например, ключевое слово FROM можно написать как from или From.
- Имена атрибутов, псевдонимов, таблиц и т. д. также не чувствительны к регистру символов. Регистр учитывается, только если идентификатор заключен в кавычки. Например, варианты имени столбца name и ‘Name’ будут восприниматься как различные.

4.2. Способы использования языка SQL

- Операторы языка SQL могут выполняться как **непосредственно** (т. е. интерактивно, с подключенного терминала), так и в виде части прикладной программы (т. е. операторы SQL могут быть **внедренными,** а значит, могут смешиваться с операторами базового языка этой программы).
- Фундаментальный принцип, лежащий в основе технологии внедрения операторов SQL**,** называется **принципом двухрежимности.** Он заключается в том, что *любое выражение SQL, которое можно использовать интерактивно, можно применять и путем внедрения в прикладную программу.* Конечно, существует множество различий в деталях между интерактивными операторами SQL и их внедренными аналогами. В частности, операции выборки требуют существенной дополнительной обработки в вычислительной среде базового языка.

4.2. Способы использования языка SQL (продолжение)

- Программный SQL может иметь следующие разновидности:

1) встроенный SQL – разновидность, наиболее широко применяемая в реляционных СУБД;
2) динамический SQL – усовершенствованная форма встроенного SQL; с его помощью создаются утилиты общего назначения для работы с базами данных;
3) SQL API – альтернативная разновидность программного SQL – интерфейс вызовов функций, применяемый в нескольких популярных СУБД.

4.2.1. Порядок обработка запросов в СУБД

- Выполняется синтаксический анализ SQL-запроса. СУБД разделяет запрос на отдельные слова, затем проверяет, правильно ли в запросе указана команда, используются ли допустимые предложения и т. д. На этом этапе обнаруживаются синтаксические ошибки.
- Осуществляется проверка семантической правильности SQL-запроса. Посредством обращения к системному каталогу выясняется, существуют ли в базе данных таблицы, указанные в инструкции, существуют ли указанные столбцы и являются ли их имена однозначными, имеет ли пользователь привилегии, необходимые для выполнения инструкции. На этом этапе обнаруживаются семантические ошибки.
- Выполняется планирование запроса – СУБД исследует возможные способы его выполнения. Выясняется, например, может ли быть использован индекс для ускорения поиска. После исследования возможных альтернатив СУБД выбирает одну из них.
- Запрос выполняется, и результаты возвращаются клиенту.

4.3. Встроенный SQL

- *Внедренные* операторы **SQL** предваряются инструкцией **EXEC SQL.**
- Операторы SQL могут включать ссылки на **базовые переменные** *(т.е.* переменные базового языка). Подобные ссылки должны иметь **префикс в виде двоеточия,** позволяющий отличить их от имен столбцов таблиц SQL
- Обратите внимание на конструкцию **INTO** оператора SELECT. Назначение этой конструкции — указать результирующие (целевые) переменные, в которых будут возвращены выбранные значения.
- Все базовые переменные, на которые ссылаются внедренные операторы SQL, должны быть определены в разделе объявлений **внедренного языка SQL,** который ограничивается операторами **BEGIN DECLARE SECTION И END DECLARE SECTION.**
- Каждая программа, содержащая внедренные операторы SQL, должна включать базовую переменную с именем **SQLSTATE.** После выполнения любого присутствующего в программе оператора SQL в эту переменную возвращается код состояния. В частности, код состояния 00000 означает, что оператор был выполнен успешно.

4.3. Встроенный SQL (продолжение)

- Каждая переменная базового языка должна иметь **тип данных,** соответствующий значениям, для хранения которых эта переменная используется. В частности, базовая переменная, используемая в качестве целевой (например, для хранения результатов операции SELECT), должна иметь тип данных, совместимый с типом выражения, значение которого присваивается этой целевой базовой переменной.
- Базовые переменные для столбцов таблиц SQL могут иметь те же имена, что и имена соответствующих столбцов.
- Как уже упоминалось, выполнение каждого оператора SQL, в принципе, должно сопровождаться проверкой значения, возвращаемого в переменной SQLSTATE. Для упрощения этого процесса предназначен оператор **WHENEVER,** который имеет следующий синтаксис.
  EXEC SQL WHENEVER *<condition> <action>*

4.3.1. Встроенный SQL (пример)

```
main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        int officenum;                  /* Номер офиса (задает
                                           пользователь) */
        char cityname[ 16 ];            /* Город */
        char regionname[ 11 ];          /* Регион */
        float targetval;                /* План продаж */
        float salesval;                 /* Объем продаж */
    exec sql end declare section;

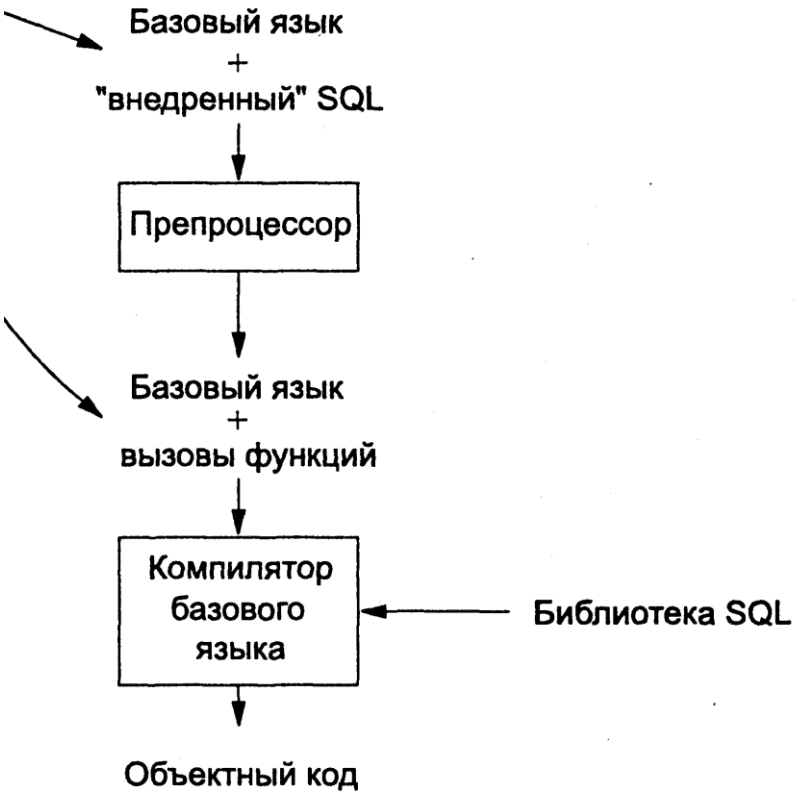

    /* Настройка обработки ошибок */
    exec sql whenever sqlerror goto query_error;
    exec sql whenever not found goto bad_number;

    /* Запрос номера офиса у пользователя */
    printf( " Введите номер офиса : " );
    scanf( "%d", &officenum );
```

4.3.1. Встроенный SQL (пример) (продолжение)

```
    /* Выполнение SQL-запроса */
    exec sql select city, region, target, sales
             from offices
             where office = :officenum
             into :cityname, :regionname,
                  :targetval, :salesval;
    /* Вывод результатов */
    printf( "Город %s\n", cityname );
    printf( "Регион: %s\n", regionname );
    printf( "План %f\n" , targetval );
    printf( "Продажи: %f\n", salesval );
    exit();
    query_error:
        printf( "Ошибка SQL: %s\n", sqlca.sqlstate ) ;|
        exit();
    bad_number:
       printf( "Неверный номер офиса.\n" );
       exit();
}
```

4.3.2. Обработка программного кода с инструкциями SQL



4.3.3. Встроенный SQL – преимущества

- Смешение инструкций SQL и языка программирования в исходном тексте программы является эффективным методом слияния двух языков. Принимающий язык обеспечивает **управление**, блочную структуру, использование переменных и функций ввода-вывода, а SQL – только **доступ к базе данных**.
- Применение препроцессора означает, что трудоемкая работа по синтаксическому анализу и оптимизации может быть выполнена на этапе разработки. Полученная в результате исполняемая программа эффективно использует ресурсы центрального процессора.
- Интерфейс программы к закрытым функциям СУБД во время выполнения полностью скрыт от прикладного программиста. Программист работает со встроенным SQL на уровне исходного текста и может не беспокоиться о других, более сложных, интерфейсах.

4.3.4. Встроенный SQL – обработка ошибок

Для более гибкой обработки ошибок в интерфейсе встраиваемого SQL представлена глобальная переменная с именем sqlca (SQL Communication Area, Область сведений SQL), имеющая следующую структуру в СУБД PostgreSQL:

```
struct
{
   char sqlcaid[8];
   long sqlabc;
   long sqlcode;
   struct
   {
      int sqlerrml;
      char sqlerrmc[ SQLERRMC_LEN ];
   } sqlerrm;
   char sqlerrp[8];
   long sqlerrd[6];
   char sqlwarn[8];
   char sqlstate[5];
} sqlca;
```

4.3.4. Встроенный SQL – обработка ошибок (продолжение)

В стандарт была введена переменная SQLSTATE. Она состоит из двух частей:

- Двухсимвольный класс ошибки, который дает общую классификацию ошибки (например, «ошибка соединения», «недействительные данные» или «предупреждение»).
- Трехсимвольный подкласс ошибки, который определяет конкретный тип ошибки. Например, в классе «недействительные данные» могут быть такие подклассы ошибок, как «деление на нуль», «недействительное числовое значение» или «недействительные дата / время».

```
exec sql delete  from salesreps
            whe re quota < 150000;
if ( strcmp( sqlca.sqlstate, "00000" ) )
   goto  error_routine ;
. . .
error_routine:
   printf( "Ошибка SQL: %s\n", sqlca.sqlstate ) ;
   exit ( ) ;
```

4.3.5. Курсоры

- Проблема состоит в том, что операторы выборки в общем случае выбирают **не одну**, а **множество** строк, в то время как процедурные базовые языки обычно не приспособлены для выборки больше одной строки за одно обращение. Следовательно, необходимо создать своего рода «мост» между предусмотренными в языке SQL средствами выборки, позволяющими получать одновременно множество строк, и применяемыми в базовом языке средствами обработки, допускающими одновременное использование только одной строки. В качестве подобного «моста» используются **курсоры.**
- Курсор представляет собой своего рода *логический указатель,* который может использоваться в приложении для перемещения по набору строк, указывая поочередно на каждую из них и таким образом обеспечивая возможность адресации этих строк — по одной за один раз.
- Для этого во встроенном SQL для работы с курсорами добавляется несколько новых инструкций.

4.3.5. Курсоры (пример)

```
/* Определить курсор */
EXEC  SQL  DECLARE  X  CURSOR  FOR
           SELECT S.S#, S.SNAME, S.STATUS
           FROM S
           WHERE S.CITY = :Y
           ORDER BY Si ASC;

/* Выполнить запрос */
EXEC  SQL  OPEN X;
   DO <для всех строк таблицы S, доступных через курсор X>;
      /* Получить данные о следующем поставщике */
      EXEC  SQL  FETCH  X  INTO  :S#, :SNAME, :STATUS;
   END ;

/* Перевести курсор X в неактивное состояние */
EXEC  SQL  CLOSE  X;
```

4.4. Динамический SQL

- Концепция, лежащая в основе динамического SQL, проста: встроенная инструкция SQL не записывается в исходный текст программы. Вместо этого программа формирует текст инструкции во время выполнения в одной из своих областей данных, а затем передает сформированную инструкцию в СУБД для динамического выполнения.
- Как и следует ожидать, динамический SQL *менее эффективен*, чем статический. В статическом SQL синтаксический анализ SQL-команды, проверка, оптимизация и генерирование плана выполнения команды производятся на этапе компиляции. При использовании динамического SQL все этапы выполнения SQL-команды производятся во время исполнения программы.

4.4. Динамический SQL (продолжение)

- **Выполнение операторов без набора результатов (PostgreSQL)**
- ```
  EXEC SQL BEGIN DECLARE SECTION;
      const char *stmt = "CREATE TABLE test1 (...);";
  EXEC SQL END DECLARE SECTION;

  EXEC SQL EXECUTE IMMEDIATE :stmt;
  ```

- EXECUTE IMMEDIATE можно применять для SQL-операторов, которые не возвращают набор результирующих строк (например, команды DDL, команды INSERT, UPDATE, DELETE). Выполнять операторы, которые получают данные, (например, SELECT) таким образом нельзя.
  ```

4.4. Динамический SQL (продолжение)

- **Выполнение оператора с входными параметрами (PostgreSQL)**
- Более эффективно выполнять произвольный оператор SQL можно, *подготовив его один раз*, а затем запуская подготовленный оператор столько, сколько нужно. Также можно подготовить обобщённую версию оператора, а затем выполнять специализированные его версии, подставляя в него параметры. Подготавливая оператор, поставьте знаки вопроса там, где позже хотите подставить параметры. Например:
- ```
  EXEC SQL BEGIN DECLARE SECTION;
      const char *stmt = "INSERT INTO test1 VALUES( ?, ? );";
  EXEC SQL END DECLARE SECTION;
  EXEC SQL PREPARE mystmt FROM :stmt;
      . . .
  EXEC SQL EXECUTE mystmt USING 42, 'foobar';
  ```
- Когда подготовленный оператор больше не нужен, его следует освободить:
- `EXEC SQL DEALLOCATE PREPARE` *имя*`;`
  ```

4.4. Динамический SQL (продолжение)

- **Выполнение оператора с набором результатов (PostgreSQL)**
- Для выполнения оператора SQL с одной строкой результата можно применить команду EXECUTE. Чтобы сохранить результат, добавьте предложение INTO.
- ```
  EXEC SQL BEGIN DECLARE SECTION;
      const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
      int v1, v2;
      VARCHAR v3[50];
  EXEC SQL END DECLARE SECTION;
  EXEC SQL PREPARE mystmt FROM :stmt;
      . . .
  EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
  ```
- Команда EXECUTE может содержать предложение INTO и/или предложение USING, либо не содержать ни того, ни другого.
  ```

4.4. Динамический SQL (продолжение)

- **Выполнение оператора с набором результатов (PostgreSQL)**

1. Программа создает в буфере строку инструкции SQL, как и в случае с инструкцией EXECUTE IMMEDIATE. Любая константа в тексте инструкции может быть заменена вопросительным знаком; это говорит о том, что значение константы будет предоставлено позднее. Вопросительный знак называется *маркером параметра*, хотя очень часто используется термин «*заполнитель*».
2. Инструкция PRE PARE дает команду СУБД произвести синтаксический анализ инструкции, проверить ее правильность, оптимизировать и создать план выполнения (это первый шаг взаимодействия с СУБД). СУБД присваивает некоторое значение переменным SQLCODE / SQLSTATE, чтобы сообщить о любых ошибках, обнаруженных в инструкции, и сохраняет план выполнения. Обратите внимание на то, что, когда СУБД выполняет инструкцию PREPARE, она не выполняет соответствующий план.
3. Если программе требуется выполнить подготовленную ранее инструкцию, она передает в СУБД инструкцию EXECUTE вместе со значениями всех маркеров параметров (это второй шаг взаимодействия с СУБД). СУБД подставляет значения параметров, исполняет ранее подготовленный план и присваивает код завершения переменным SQLCODE / SQLSTATE .
4. Программа может многократно выполнять инструкцию EXECUTE, всякий раз изменяя значения параметров. С УБД просто повторяет второй шаг, поскольку первый уже выполнен, и его результат – план выполнения – остается корректен.

4.4. Динамический SQL (продолжение)

- **Выполнение оператора с набором результатов (PostgreSQL)**
- Если ожидается, что запрос вернёт более одной строки результата, следует применять курсор, как показано в следующем примере.
-

```
EXEC SQL BEGIN DECLARE SECTION;
    char dbaname[ 128 ];
    char datname[ 128 ];
    char *stmt = "SELECT u.usename as dbaname, d.datname "
                 " FROM pg_database d, pg_user u "
                 " WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;
EXEC SQL CONNECT TO testdb AS con1 USER testuser;
EXEC SQL PREPARE stmt1 FROM :stmt;
EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
while ( 1 )
{
    EXEC SQL FETCH cursor1 INTO :dbaname, :datname;
    printf( "dbaname = %s, datname = %s\n", dbaname, datname );
}
EXEC SQL CLOSE cursor1;
EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

4.5. Сравнение статического и динамического SQL

- **Простота**. Статический SQL относительно прост; даже самый сложный его элемент – курсоры – можно легко освоить, вспомнив концепцию файлового ввода-вывода. Динамический SQL довольно сложен; в нем осуществляется динамическое формирование инструкций, используются структуры данных переменной длины, выполняется распределение памяти и решаются другие сложные задачи.
- **Производительность**. Во время компиляции программы, использующей статический SQL, создается план выполнения всех встроенных инструкций; инструкции динамического SQL компилируются непосредственно на этапе выполнения. В результате производительность статического SQL, как правило, намного выше, чем динамического. Производительность динамического SQL существенно зависит от дизайна приложения; минимизация накладных расходов на компиляцию позволяет достичь производительности статического SQL.
- **Гибкость**. Динамический SQL дает программе возможность решать на этапе выполнения, какие конкретно инструкции SQL она будет выполнять. Статический SQL требует, чтобы все инструкции SQL были написаны заранее, на этапе создания программы; тем самым он ограничивает гибкость программы.

4.6. Интерфейс SQL/CLI

- CLI означает Call-Level Interface.
- Интерфейс прикладного программирования. При этом подходе программа взаимодействует с СУБД с применением набора функций, называемого интерфейсом прикладного программирования (Application Program Interface – API). Вызывая функции API, программа передает в СУБД инструкции SQL и получает обратно результаты запросов. В этом случае специальный препроцессор не требуется.
- Пользователям предоставляется специальная библиотека функций, представляющая собой интерфейс прикладного программирования (API) между приложениями и СУБД. Прикладная программа вызывает функции SQL API для передачи в СУБД инструкций SQL и для получения от СУБД результатов запросов и служебной информации.

4.6.1. Интерфейс SQL/CLI (порядок работы)

1. Программа получает доступ к базе данных путем вызова одной или нескольких API-функций, подключающих программу к СУБД, а зачастую - и к конкретной базе данных или схеме.
2. Для пересылки инструкции SQL в СУБД программа формирует инструкцию в виде текстовой строки (зачастую в переменной языка программирования) и затем передает эту строку в качестве параметра при вызове API-функции.
3. Программа вызывает API-функции для проверки состояния переданной в СУБД инструкции и для обработки ошибок.
4. Если инструкция SQL представляет собой запрос на выборку, то при вызове API-функций для получения результатов запроса программа записывает последние в свои переменные; обычно за один вызов возвращается одна строка или один столбец данных.
5. Свое обращение к базе данных программа заканчивает вызовом API-функции, отключающей ее от СУБД.

4.6.2. Библиотека libpq в PostgreSQL

- libpq — это интерфейс PostgreSQL для программирования приложений на языке C. Библиотека libpq содержит набор функций, используя которые, клиентские программы могут передавать запросы серверу PostgreSQL и принимать результаты этих запросов.
- libpq также является базовым механизмом для нескольких других прикладных интерфейсов PostgreSQL, включая те, что написаны для C++, Perl, Python, Tcl и ECPG.
- Чтобы собрать (то есть, скомпилировать и скомпоновать) программу, использующую libpq, нужно проделать следующие действия:
- Включить заголовочный файл libpq-fe.h:
  #include <libpq-fe.h>
- Скомпилировать программу:
  cc -c -I/usr/local/pgsql/include testprog.c
- При компоновке окончательной программы добавить параметр -lpq, чтобы была подключена библиотека libpq, а также параметр -L*каталог*, указывающий на каталог, в котором находится libpq
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq

4.6.2. Библиотека libpq в PostgreSQL (пример)

```
/*
 * Программа: test.c
 * Проверка возможности доступа к базе данных с помощью
 * библиотеки libpq
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "libpq-fe.h"

static void exit_nicely( PGconn *conn )
{
  PQfinish( conn );
  exit( 1 );
}

int main( int argc, char **argv )
{
  const char *conninfo;/* параметры для соединения с базой
                          данных */
  PGconn     *conn;    /* дескриптор соединения с базой данных */
  PGresult   *res;     /* результат выполнения SQL-запроса */
  int        nFields;  /* число полей в выборке */
  int        i, j;
```

4.6.2. Библиотека libpq в PostgreSQL (пример) (продолжение)

```
/*
   * если пользователь передает параметр в командной строке,
   * то нужно использовать его в качестве строки conninfo;
   * если параметр в командной строке не передан,
   * то по умолчанию для подключения к базе данных
   * используется строка "dbname=test user=postgres"
  */
  if ( argc > 1 )
    conninfo = argv[ 1 ];
  else
    conninfo = "dbname=test user=postgres";

  /* подключаемся к базе данных */
  conn = PQconnectdb( conninfo );

  /* проверяем успешность подключения */
  if ( PQstatus( conn ) != CONNECTION_OK )
  {
    fprintf( stderr,
             "Подключиться к базе данных не удалось: %s",
             PQerrorMessage( conn ) );
    exit_nicely( conn );
  }
```

4.6.2. Библиотека libpq в PostgreSQL (пример) (продолжение)

```
/* функция PQexec исполняет SQL-запрос, переданный ей
   в виде символьной строки */
res = PQexec( conn, "SELECT * FROM Students" );
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
  fprintf( stderr, "Запрос SELECT неудачен: %s",
           PQerrorMessage( conn ) );
  PQclear( res );
  exit_nicely( conn );
}

nFields = PQnfields( res ); /* количество полей в выборке */

/* сначала выведем имена полей для полученной выборки
   (фактически это имена полей таблицы Students) */
for ( j = 0; j < nFields; j++ )
{
  /* для поля "Фамилия, имя, отчество" выделим больше места
     (функция PQfname возвращает имя поля по его порядковому
     номеру в выборке) */
  if ( strcmp( PQfname( res, j ), "name" ) == 0 )
    printf( "%-30s", PQfname( res, j ) );
  else
    printf( "%-10s", PQfname( res, j ) );
}

printf( "\n\n" );
```

4.6.2. Библиотека libpq в PostgreSQL (пример) (продолжение)

```
  /* выведем данные из полученной выборки
     (функция PQntuples возвращает число строк в выборке) */
  for ( i = 0; i < PQntuples( res ); i++ )
  {
    for ( j = 0; j < nFields; j++ )
    {
      if ( strcmp( PQfname( res, j ), "name" ) == 0 )
        /* функция PQgetvalue возвращает значение поля номер i
           из строки номер j */
        printf( "%-30s", PQgetvalue( res, i, j ) );
      else
        printf( "%-10s", PQgetvalue( res, i, j ) );
    }
    printf( "\n" );
  }

  /* освободим память, которую занимала структура res,
     как только эта структура больше не нужна */
  PQclear( res );

  /* закроем соединение с базой данных */
  PQfinish( conn );

  return 0;
}
```

4.7. Порядок обработки запроса SELECT (на примере СУБД PostgreSQL)

- В запросе ключевые слова следуют в таком порядке: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY.

1. Выполняются все запросы в списке WITH. По сути они формируют временные таблицы, к которым затем можно обращаться в списке FROM. Запрос в WITH выполняется только один раз, даже если он фигурирует в списке FROM неоднократно.
2. Вычисляются все элементы в списке FROM. (Каждый элемент в списке FROM представляет собой реальную или виртуальную таблицу.) Если список FROM содержит несколько элементов, они объединяются перекрёстным соединением.
3. Если указано предложение WHERE, все строки, не удовлетворяющие условию, исключаются из результата.
4. Если присутствует указание GROUP BY либо в запросе вызываются агрегатные функции, вывод разделяется по группам строк, соответствующим одному или нескольким значениям, а затем вычисляются результаты агрегатных функций. Если добавлено предложение HAVING, оно исключает группы, не удовлетворяющие заданному условию.

4.7. Порядок обработки запроса SELECT (продолжение)

5. Вычисляются фактические выходные строки по заданным в SELECT выражениям для каждой выбранной строки или группы строк.
6. SELECT DISTINCT исключает из результата повторяющиеся строки. SELECT DISTINCT ON исключает строки, совпадающие по всем указанным выражениям. SELECT ALL (по умолчанию) возвращает все строки результата, включая дубликаты.
7. Операторы UNION, INTERSECT и EXCEPT объединяют вывод нескольких команд SELECT в один результирующий набор. Оператор UNION возвращает все строки, представленные в одном, либо обоих наборах результатов. Оператор INTERSECT возвращает все строки, представленные строго в обоих наборах. Оператор EXCEPT возвращает все строки, представленные в первом наборе, но не во втором. Во всех трёх случаях повторяющиеся строки исключаются из результата, если явно не указано ALL. Чтобы явно обозначить, что выдаваться должны только неповторяющиеся строки, можно добавить избыточное слово DISTINCT. Заметьте, что в данном контексте по умолчанию подразумевается DISTINCT, хотя в самом SELECT по умолчанию подразумевается ALL.

4.7. Порядок обработки запроса SELECT (продолжение)

8. Если присутствует предложение ORDER BY, возвращаемые строки сортируются в указанном порядке. В отсутствие ORDER BY строки возвращаются в том порядке, в каком системе будет проще их выдать.
9. Если указано предложение LIMIT (или FETCH FIRST) либо OFFSET, оператор SELECT возвращает только подмножество строк результата.
10. Если указано FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE или FOR KEY SHARE, оператор SELECT блокирует выбранные строки, защищая их от одновременных изменений.

4.8. Полезные ссылки

- Моргунов, Е. П. Технологии разработки программ в среде операционных систем Linux и FreeBSD. Вводный курс [Текст] : учеб. пособие / Е. П. Моргунов, О. Н. Моргунова. – Красноярск, 2018. – 207 с. http://www.morgunov.org/programming.html
- Bruce Momjian http://momjian.us/main/presentations/sql.html

Литература

1. **Гарсиа-Молина, Г. Системы баз данных. Полный курс : пер. с англ. / Гектор Гарсиа-Молина, Джеффри Ульман, Дженнифер Уидом. – М. : Вильямс, 2003. – 1088 с.**
2. **Грофф, Дж. SQL. Полное руководство : пер. с англ. / Джеймс Р. Грофф, Пол Н. Вайнберг, Эндрю Дж. Оппель. – 3-е изд. – М. : Вильямс, 2015. – 960 с.**
3. **Дейт, К. Дж. Введение в системы баз данных : пер. с англ. / Крис Дж. Дейт. – 8-е изд. – М. : Вильямс, 2005. – 1328 с.**
4. **Коннолли, Т. Базы данных. Проектирование, реализация и сопровождение. Теория и практика : пер. с англ. / Томас Коннолли, Каролин Бегг. – 3-е изд. – М. : Вильямс, 2003. – 1436 с.**
5. Кузнецов, С. Д. Основы баз данных : учеб. пособие / С. Д. Кузнецов. – 2-е изд., испр. – М. : Интернет-Университет Информационных Технологий ; БИНОМ. Лаборатория знаний, 2007. – 484 с.
6. Лузанов, П. PostgreSQL для начинающих / П. Лузанов, Е. Рогов, И. Лёвшин ; Postgres Professional. – М., 2017. – 146 с.
7. Моргунов, Е. П. Язык SQL. Базовый курс : учеб.-практ. пособие. / Е. П. Моргунов ; под ред. Е. В. Рогова, П. В. Лузанова ; Postgres Professional. – М., 2017. – 257 с.
8. **PostgreSQL [Электронный ресурс] : официальный сайт / The PostgreSQL Global Development Group. – http://www.postgresql.org.**
9. Postgres Professional [Электронный ресурс] : российский производитель СУБД Postgres Pro : официальный сайт / Postgres Professional. – http://postgrespro.ru.

Задание

Для выполнения практических заданий необходимо использовать книгу:

Моргунов, Е. П. Язык SQL. Базовый курс : учеб.-практ. пособие / Под ред. Е. В. Рогова, П. В. Лузанова ; Postgres Professional. – М., 2017. – 257 с. https://postgrespro.ru/education/books/sqlprimer

1. Изучить материал главы 6. Запросы к базе данных выполнять с помощью утилиты psql, описанной в главе 2, параграф 2.2.