ГЛАВА 6. АНАЛИЗ СОВРЕМЕННОЙ ТЕХНОЛОГИИ РЕАЛИЗАЦИИ БАЗ ДАННЫХ. ЯЗЫКИ И СТАНДАРТЫ

6.1. Структура современной СУБД на примере Microsoft SQL Server

Для лучшего понимания принципов работы современных СУБД рассмотрим структуру одной из них, а именно Microsoft SQL Server. Несмотря на то, что каждая коммерческая СУБД имеет свои отличительные особенности, информации о том, как устроена какая-то из СУБД, обычно бывает достаточно для быстрого первоначального освоения другой СУБД. Итак, приступим к краткому обзору Microsoft SQL Server. За основу для настоящего рассмотрения возьмем Microsoft SQL Server 2000.

Краткий обзор возможностей Microsoft SQL Server был приведен в разделе, посвященном современным СУБД. В данном разделе рассмотрим основные моменты, связанные с архитектурой соответствующей базы данных.

6.1.1. Архитектура базы данных

Что понимается нами под архитектурой базы данных? В предыдущих главах достаточно подробно были рассмотрены вопросы проектирования и функционирования реляционных баз данных. Microsoft SQL Server – реляционная СУБД. В данном случае под архитектурой базы данных мы будем понимать ее основные составляющие и принципы их взаимодействия (имея в виду, что речь идет именно о реляционной базе данных). При этом архитектуру можно рассматривать на двух уровнях абстракции:

- Логический уровень (логическая архитектура базы данных) – данный уровень рассмотрения подразумевает изучение базы данных на уровне ее содержательных объектов. Здесь каждая СУБД имеет некоторые отличия, но они являются не очень значительными. Однако полезно знать, что у разных СУБД существенно отличаются механизмы перехода от логического к физическому уровню представления.
- Физический уровень (физическая архитектура базы данных). Понятно, что любая, сколь угодно сложная база данных на самом деле представляет собой набор файлов, хранящихся на жестком дис-

ке одного или нескольких компьютеров. Данный уровень рассмотрения подразумевает изучение базы данных на уровне файлов, хранящихся на жестком диске. Структура этих файлов – особенность каждой конкретной СУБД, в т.ч. и Microsoft SQL Server.

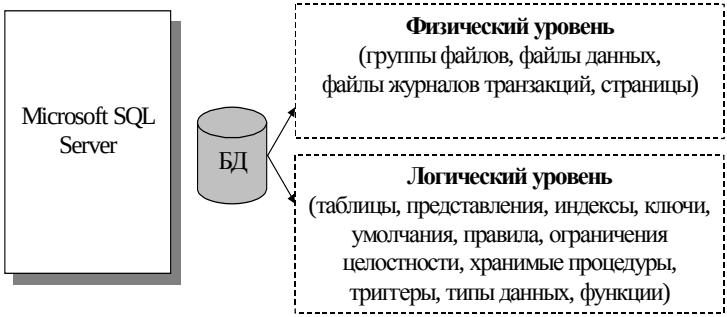

Рис. 40. Архитектура базы данных в Microsoft SQL Server 2000

Логический уровень

Рассмотрим логический уровень представления базы данных.

Запустим специальную утилиту Microsoft SQL Server Enterprise Manager, выберем одну из демонстрационных баз данных и посмотрим, что́ с точки зрения данной утилиты (а следовательно, с точки зрения СУБД) является ее основными компонентами (рис. 41).

Приведем краткие характеристики этих компонентов основных типов объектов базы данных с целью осознания ее типовой структуры.

Tables (Таблицы). Таблицы базы данных предназначены собственно для хранения данных. Подразделяются на две категории: User (пользовательские) и System (системные). Пользовательские таблицы хранят полезные данные из предметной области, системные – различную служебную информацию.

Views (Представления). По сути своей являются «виртуальными таблицами». С точки зрения пользователя, представление есть почти то же самое, что и таблица. На самом деле представление формируется на основе SQL-запроса SELECT, формируемого по обычным правилам. Таким образом, представление есть поименованный запрос SELECT.

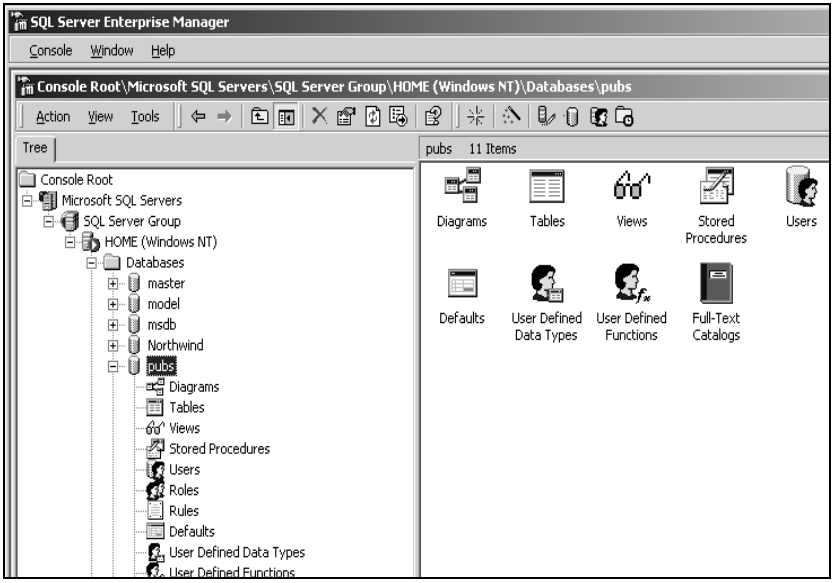

Рис. 41. База данных в Microsoft SQL Server 2000.
SQL Server Enterprise Manager

Indexes (Индексы). Индексы существуют для поддержания вместе с данными информации об их упорядоченности по различным критериям. Наличие информации об упорядоченности позволяет существенно повысить производительность некоторых операций, в частности поиска данных. Индексы существуют непосредственно вместе с таблицами и не имеют смысла сами по себе. Индексирование может быть выполнено по одному или нескольким столбцам. Индексирование может быть произведено в любой момент.

Diagrams (Диаграммы). Диаграммы представляют собой специальные визуальные средства изучения и описания структуры базы данных. Так, при помощи диаграмм вы можете изучать структуру таблиц и связи между ними, а также вносить в схему БД изменения.

Keys (Ключи). Так же как и индексы, ключи не существуют сами по себе. Ключ – фундаментальное понятие в области баз данных, является одним из типов ограничений целостности.

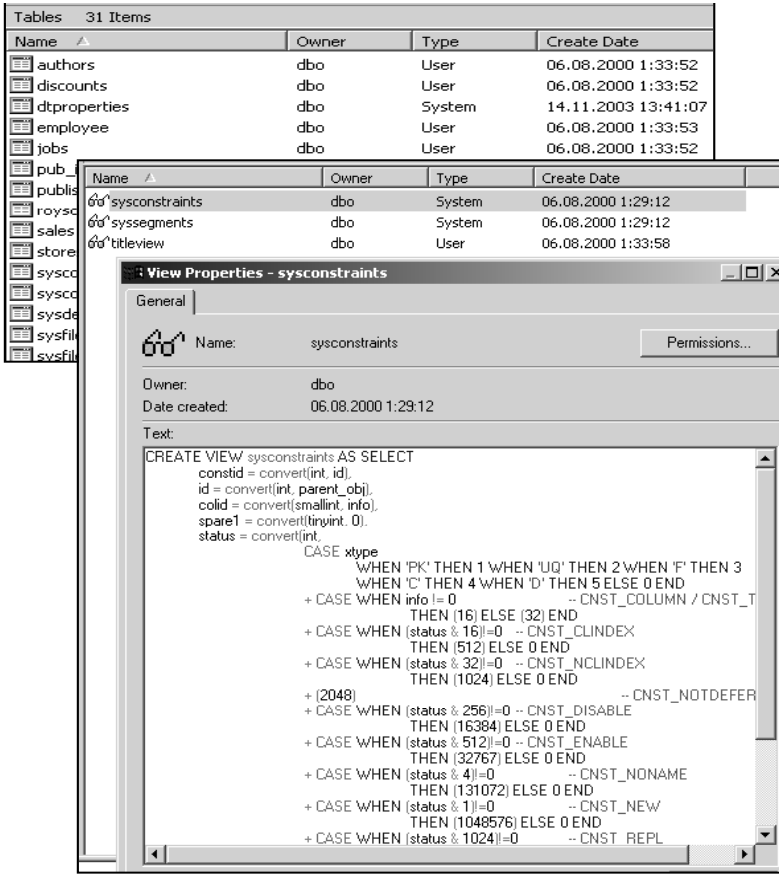

Рис. 42. Некоторые характеристики Tables и Views

Defaults (Умолчания). Умолчания не существуют отдельно от таблиц. Умолчания определяют, какие значения будут подставлены в поле данных при добавлении строки в таблицу в том случае, если значение не задано явно.

Rules (Правила). Правила – специальный механизм, предназначенный для установления ограничений на диапазон возможных значений поля таблицы или нового типа данных, определяемого пользователем.

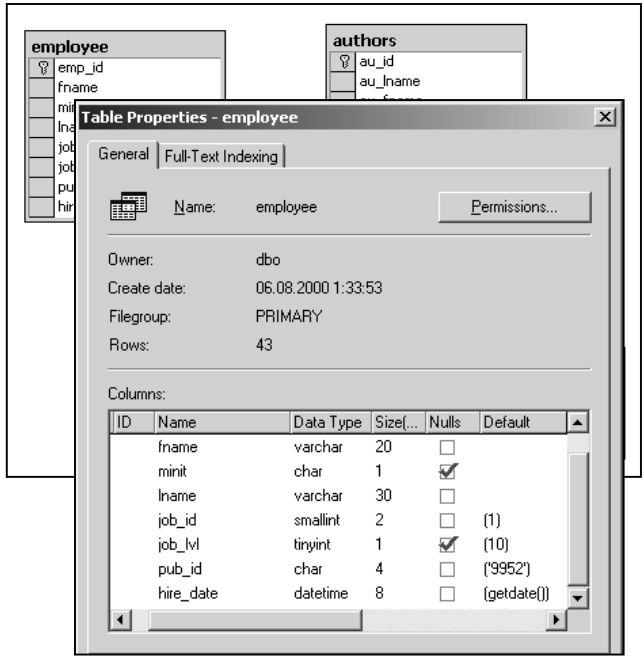

Рис. 43. Характеристика полей таблицы

Constraints (Ограничения целостности). Ограничения целостности, так же как и большинство объектов базы данных, не имеют смысла отдельно от таблиц. Ограничения целостности определяют диапазон возможных значений полей таблицы.

Stored Procedures (Хранимые процедуры). Хранимая процедура – набор команд SQL, сохраненных специальным способом в базе данных. Хранимые процедуры располагаются на сервере вместе с базой данных и могут запускаться различными пользователями, имеющими соответствующие права. При этом выполняется не один SQL-запрос, а все, что реализованы в теле процедуры.

Triggers (Триггеры). Триггеры не существуют отдельно от таблицы. Триггер – специальный вид хранимой процедуры, предназначенный для производства некоторых специальных действий при выполнении операций вставки, удаления, редактирования данных.

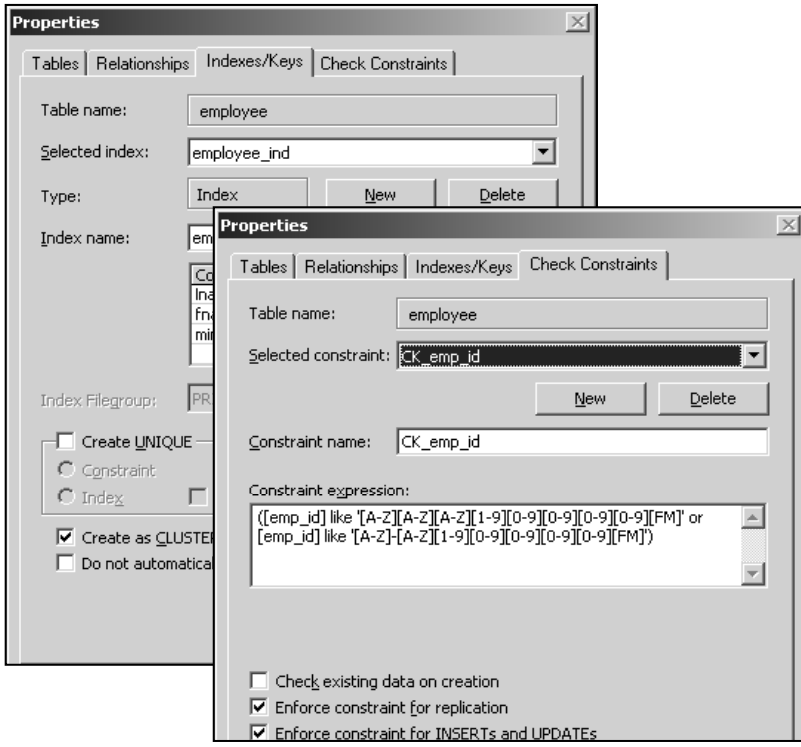

Рис. 44. Ограничения целостности

User-defined data types, UDDT (Определяемые пользователем типы данных). Предоставляет специальный аппарат для создания пользовательских типов данных.

User-defined functions, UDF (Определяемые пользователем функции). Функция, определяемая пользователем, представляет собой набор команд SQL, сохраненных в специальном виде. Наряду с пользовательскими функциями, существует некоторый набор стандартных функций Microsoft SQL Server, которыми можно пользоваться в работе с базой данных.

Users (Пользователи). Определяет список пользователей базы данных. Служит для работы механизма защиты данных.

Roles (Роли). Роли пользователей – важный механизм для организации разграничения доступа. Для базы данных могут быть определены различные роли (например, администратор). Далее они могут быть назначены пользователям.

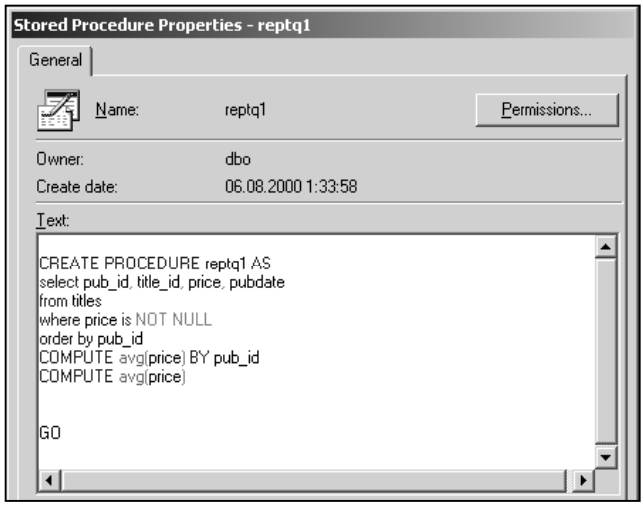

Рис. 45. Хранимые процедуры

Физический уровень

Файлы и группы файлов

Рассмотрим физический уровень представления базы данных [22].

На физическом уровне база данных в Microsoft SQL Server 2000 представляет собой набор файлов с данными и набор файлов, содержащий журнал транзакций. Каждая база данных состоит из собственного уникального набора файлов, что существенно повышает надежность системы (в случае случайной порчи файлов одной из баз, например вследствие частичной поломки жесткого диска, с другой ничего не произойдет).

Рассмотрим физическую структуру на уровне ее основных абстракций. Основными элементами структуры являются файлы и группы файлов. Все файлы делятся на два типа: файлы данных и файлы журнала транзакций.

- **Файлы данных** содержат собственно данные и различные объекты, входящие в логическую структуру базы данных, такие, как триггеры, хранимые процедуры и т.д.
- **Файлы журнала транзакций** содержат сведения о ходе выполнения транзакций. В них содержится информация о том, когда началась и закончилась транзакция, какие ресурсы, кем и когда были заблокированы или разблокированы и т.д.

Каждая база данных содержит как минимум два файла: файл с данными и файл журнала транзакций. Имеется возможность создания при необходимости новых файлов с данными и новых файлов для журнала транзакций. При этом можно указать, какие данные в каком файле хранить. Совершенно очевидно, что за всем этим многофайловым механизмом кроется определенный смысл. Он заключается в возможности оптимального распределения данных по файлам с целью повышения эффективности их обработки. Поиск подобного оптимального распределения – сложная практическая задача. Для ее однозначного решения трудно предложить какие-то общие рецепты, в каждом конкретном случае необходимо действовать по-своему. В принципе, для решения проблемы можно применить два подхода. Один из возможных подходов состоит в попытке построения теоретических оценок того, какое размещение является наилучшим в некотором смысле. На практике применение этого подхода сопряжено с большими трудностями. Второй подход заключается в проведении испытаний с различными распределениями данных по файлам с целью выявления наилучшего распределения. Этот подход также трудно претворить в жизнь из-за того, что на этапе разработки невозможно успешно смоделировать реальную ситуацию, в которой будет функционировать база данных. В итоге на практике применяются некоторые элементы обоих подходов. Так, например, очевидно, что при наличии нескольких жестких дисков разумно создать по крайней мере по одному файлу на каждом из них, разместив в них данные, которые обычно могут обрабатываться параллельно. Благоприятно скажется на производительности и размещение на разных носителях файлов с данными и журнала транзакций.

Файлы с данными бывают двух типов [22]:

- **Основной или главный файл** (**Primary File**) присутствует в любой базе данных. Если в базе данных всего один файл с данными, то он является главным. Главный файл содержит так называемые метаданные – данные о том, как устроена база данных (системные

таблицы и т.д.). Кроме системной информации, главный файл может хранить и пользовательские данные. Главный файл имеет расширение **mdf** (Master Data File).

- **Вторичный или дополнительный файл** (**Secondary File**) – дополнительный файл для хранения пользовательских данных. Не содержит системной информации. Такие файлы могут создаваться администратором по мере необходимости. Имеет расширение **ndf** (secoNdary Data File).

Файлы журнала транзакций бывают только одного типа – **файлы журнала транзакций** (Transaction Log File). Эти файлы содержат информацию о транзакциях. Имеют расширение **ldf** (Log Data File).

При необходимости администратор может изменить расширения файлов.

Для удобства администрирования и с целью повышения эффективности обработки файлы с данными могут объединяться в группы. По умолчанию существует одна группа Primary File Group – основная группа файлов и все файлы добавляются в нее. При необходимости возможно создание пользовательских групп файлов (User File Group). Существует также группа файлов по умолчанию (Default File Group), содержащая объекты базы данных, не приписанные явно ни к одной группе. Такая группа бывает только одна.

Microsoft SQL Server 2000 применяет специальные оптимизационные механизмы для улучшения ситуации с обработкой данных с применением групп файлов. Так, СУБД стремится к равномерному распределению данных по файлам в рамках одной группы. Умелое использование механизмов групп позволяет существенно повысить быстродействие.

Страницы и группы страниц

Для работы с файлами в Microsoft SQL Server активно применяется принцип постраничной организации файлов. В рамках этого принципа файл рассматривается как набор последовательно расположенных страниц с данными. Таким образом, страница является минимальным блоком (квантом) данных, с которым может идти работа (физической записью). Заметим, что размер страницы в Microsoft SQL Server составляет 8 Кб.

Нетрудно понять, в чем заключается смысл постраничной организации. Этот способ представления позволяет использовать разные алгоритмы буферизации и существенно повысить эффективность работы

с файлами. Однако постраничная организация применяется в SQL Server лишь для файлов с данными, т.к. именно они имеют очень сложную структуру и на работу с ними ложится основная нагрузка. Файлы журнала транзакций, напротив, устроены достаточно просто, т.к. содержат последовательно расположенные записи о происходящих транзакциях; здесь допустима работа с отдельными записями.

Страницы файлов данных могут использоваться не только для хранения собственно данных, но и для представления различной служебной информации. Так, страницы могут быть одного из следующих типов [22]:

- **Data**. Предназначены для хранения данных таблиц, кроме так называемых «тяжелых» данных (Image, Text, nText).
- **Index.** Предназначены для хранения индексов таблиц и представлений.
- **Text/Image**. Используются для хранения «тяжелых» данных Image, Text, nText, занимающих обычно более одной страницы. Для таких данных место сразу же выделяется постранично.
- **Global Allocation Map (GAM)**. Служебный тип страницы. Содержит информацию об использовании групп страниц.
- **Page Free Space (PFS)**. Содержит сведения о расположении свободного пространства в страницах файла. Своеобразная замена «списка свободных элементов».
- **Index Allocation Map (IAM)**. Содержит информацию о группах страниц, которые используются таблицами.

Каждая страница имеет некоторую внутреннюю архитектуру. Так, у любой страницы присутствует заголовок, содержащий служебную информацию. Структура заголовка одинакова для страниц разных типов.

Подобная страничная организация была бы не слишком удобной для хранения огромных объемов данных, а ведь современные серверные СУБД, и в частности Microsoft SQL Server, часто используются именно в таких целях. Для дополнительной оптимизации в SQL Server было введено понятие «группа страниц» (экстент). Экстент содержит 8 страниц и, следовательно, имеет размер 64 Кб. При необходимости выделить новую страницу создается сразу целый экстент. Наряду с различиями в типах страниц существуют аналогичные различия и в типах экстентов. Детальное описание структуры и типов экстентов можно найти, например, в [22]. Мы же в рамках данного пособия огра-

ничимся изложением общих принципов физической организации и перейдем к описанию организации БД на логическом уровне.

6.2. Программное окружение БД. Проблемы доступа к данным и обработки данных

6.2.1. Проблемы доступа к данным и обработки данных

В этом разделе мы перейдем к рассмотрению реляционных баз данных, функционирующих в рамках архитектуры «клиент – сервер». Краткая характеристика подобных баз данных была изложена в п. 1.6. В данном разделе описаны основные подходы, применяемые для решения проблемы организации данных и последующего доступа к ним.

Проблемы организации данных и доступа к ним имеют огромное значение. Существует ряд вопросов, которые неизбежно возникают у всех разработчиков баз данных на протяжении всего цикла создания продукта:

- Как создать таблицы (описать схему базы данных)?
- Как описать ограничения целостности?
- Как определить значения по умолчанию для полей?
- Что делать с неопределенными значениями (NULL)?
- Как добавить записи в таблицу?
- Как обновить записи?
- Как удалить записи?
- Как осуществить выборку нужных записей из базы данных?
- Как удалить часть записей?
- Как проконтролировать непротиворечивость данных после операций редактирования?
- Как осуществлять сортировки?
- Как осуществлять разграничение доступа?

Понятно, что перечень этих вопросов не полон, но уже его достаточно для того, чтобы понять тот факт, что создание и обработка базы данных – сложная задача, требующая применения специальных технологий. Проанализировав список вопросов, можно прийти к выводу, что все они делятся на три категории:

- вопросы создания базы данных (создание таблиц, индексов, ограничений целостности);
- вопросы обеспечения безопасности и разграничения доступа;
- вопросы обработки данных (выборка, редактирование, удаление, добавление).

6.2.2. Навигационный подход

Первая категория вопросов может быть решена посредством создания в каждой конкретной СУБД некоторой утилиты, позволяющей пользователю осуществлять все необходимые действия. Однако что делать, если необходимо создать таблицу динамически во время работы программы? Как объяснить удаленному серверу, что ему нужно добавить в таблицу столбец?

Вторая категория вопросов может быть решена тем же способом, но остаются те же проблемы.

А что делать с третьей категорией вопросов? Очевидно, эти вопросы возникают постоянно при эксплуатации информационной системы. Пользователи постоянно что-то делают с данными. Рассмотрим простой пример. Пусть у нас есть таблица «Абитуриент», хранящая информацию следующего рода:
АБИТУРИЕНТ (Код абитуриента, Фамилия, Имя, Отчество, Номер аттестата, Дата выдачи аттестата).

Теперь мы хотим выполнить некоторый запрос к базе данных, результатом которого должны стать те строки таблицы «Абитуриент», для которых дата выдачи аттестата окажется больше 01.06.2003. Как алгоритмически организовать выполнение подобного запроса? Видимо, необходимо осуществить следующее:

1. Получаем доступ к таблице «Абитуриент» и устанавливаем указатель текущей строки на первую строку таблицы.
2. Анализируем поле «Дата выдачи аттестата» в текущей строке.
3. Если значение «Дата выдачи аттестата» > «01.06.2003», распечатываем на экране данные об абитуриенте.
4. Если таблица не кончилась, перемещаем указатель текущей строки на следующую строку и переходим к шагу 2, иначе заканчиваем работу.

Любой человек, знакомый с программированием, легко представит себе реализацию подобного алгоритма на любом языке программирования высокого уровня. Вот, в частности, пример реализации на Object Pascal:

```
Table.First;
while (not Table.Eof) do
begin
    if FieldByName(«Дата выдачи аттестата»).Value >
       «01.06.2003»
```

```
    then List.Add(FieldByName(«Фамилия»).AsString);
    Table.Next;
end;
```

Такой подход к обработке данных, ориентированный на последовательную работу с отдельными записями, называется **навигационным**.

Теперь представьте себе, что́ произойдет с текстом программы, если, например, в базе данных нужно найти всех абитуриентов из Нижегородской области, у которых средний бал по математическим дисциплинам за 3 семестра превысил 4, и упорядочить их по убыванию среднего балла. Очевидно, текст программы возрастет в объеме на порядок.

Теперь представьте, что необходимо предоставить пользователю базы данных интерфейс для осуществления подобных запросов. Можно ли заранее предугадать все запросы, потребность в которых может возникнуть и заранее запрограммировать их? Конечно, нет. С учетом того, что подавляющее большинство пользователей не владеет навыками программирования, это означает, что объем их действий будет ограничен рамками написанной программы, а именно, теми запросами, реализация которых предусмотрена при ее написании.

Но и это еще не все. Представьте себе, как будет функционировать весь этот механизм в рамках архитектуры «клиент – сервер». Каким образом клиентское приложение будет объяснять серверной части, что́ именно необходимо сделать с данными?

6.2.3. Подход, основанный на использовании интерпретируемых языков запросов

Для решения всех этих и некоторых других проблем в настоящее время используется альтернатива навигационному подходу – специальный интерпретируемый язык запросов SQL.

Язык SQL (Structured Query Language – структурированный язык запросов) применяется для общения пользователя с реляционной базой данных и состоит из трех частей [7]:

- DDL (Data Definition Language) – язык определения данных. Предназначен для создания базы данных (таблиц, индексов и т.д.) и редактирования ее схемы.
- DCL (Data Control Language) – язык управления данными. Содержит операторы для разграничения доступа пользователей к объектам базы данных.

- DML (Data Manipulation Language) – язык обработки данных. Содержит операторы для внесения изменений в содержимое таблиц базы данных.

Как видно из написанного выше, SQL решает все рассмотренные ранее вопросы, предоставляя пользователю достаточно простой и понятный механизм доступа к данным, не связанный с конструированием алгоритма и его описанием на языке программирования высокого уровня. Так, вместо указания того, **как** необходимо действовать, пользователь при помощи операторов SQL объясняет СУБД, **что** ему нужно сделать. Далее СУБД сама анализирует текст запроса и определяет, как именно его выполнять.

В архитектуре «клиент – сервер» язык SQL занимает очень важное место. Именно он используется как язык общения клиентского программного обеспечения с серверной СУБД, расположенной на удаленном компьютере. Так, клиент посылает серверу запрос на языке SQL, а сервер разбирает его, интерпретирует, выбирает план выполнения, выполняет запрос и отсылает клиенту результат.

Посмотрим, как выглядит запрос на языке SQL, решающий задачу о выборке абитуриентов по дате выдачи аттестата:

```
SELECT Фамилия
FROM  Абитуриенты
WHERE Дата выдачи аттестата > «01.06.2003»
```

У того, кто читает эти строки, может сложиться ложное впечатление, что мы хотим принизить значение языков программирования высокого уровня, выдвигая им некоторую альтернативу – язык SQL. Это не соответствует действительности. Понятно, что выполнение запроса все равно сводится к работе с отдельными записями, и от этого никуда не уйти. Однако важно понимать, что появление языка SQL привело к появлению некоторого нового уровня абстракции между пользователем и СУБД. Этот уровень находится ближе к пользователю, чем уровень программирования на языке высокого уровня, что снижает требования к квалификации пользователей. Второй существенный момент заключается в том, что многие вопросы, которые раньше все программистское сообщество многократно решало в силу своего понимания, зачастую дублируя действия друг друга, теперь решены в серверных СУБД, которая, к примеру, умеет самостоятельно выбирать лучший с

точки зрения быстродействия план выполнения запроса. Таким образом, отпала необходимость самостоятельного решения многих проблем, они уже решены в СУБД, а стандарт SQL предоставляет средства для доступа к возможностям СУБД.

6.3. Понятие языка SQL и его основные части

6.3.1. История возникновения и стандарты языка SQL

История возникновения языка SQL восходит к 1970 году [8], когда доктор Е.Ф. Кодд предложил реляционную модель в качестве новой модели базы данных. Для доказательства жизнеспособности новой модели данных внутри компании IBM был создан мощный исследовательский проект, получивший название System/R. Проект включал разработку собственно реляционной СУБД и специального языка запросов к базе данных. Так в начале 70-х годов появился первый исследовательский прототип реляционной СУБД. Для этого прототипа разрабатывались и опробовались разные языки запросов, один из которых получил название SEQUEL (Structured English Query Language). С момента создания и до наших дней этот язык претерпел массу изменений, но идеология и произношение названия остались неизменными (аббревиатура SQL иногда читается «Эс-Кю-Эль», а иногда «Сиквел»).

Период с 1979 года (окончание проекта System/R) до настоящего времени характеризуется развитием и совершенствованием языка SQL и его постоянно увеличивающейся ролью в индустрии, связанной с созданием и эксплуатацией баз данных. Совершенно очевидно, что язык никогда не получил бы мирового признания, если бы на него не было никаких стандартов. Стандартизация – важная часть технологических процессов конца XX века. Именно наличие разработанных и официально признанных стандартов позволило утвердиться многим современным технологиям (не только в индустрии разработки ПО, но и в многих других сферах человеческой деятельности). Как обстоит дело со стандартами языка SQL и их поддержкой в распространенных СУБД?

Когда ведут речь о стандартах в области, связанной с разработкой программного обеспечения, обычно подразумевают две организации:

- ANSI (American National Standards Institute) – Американский национальный институт стандартов;
- ISO (International Standards Organization) – Международную организацию по стандартизации.

Работа над официальным стандартом языка SQL началась в 1982 году [8] в рамках комитета ANSI. В 1986 году (обратите внимание, сколько времени ушло на разработку стандарта и согласование деталей!) был утвержден первый вариант стандарта ANSI, а в 1987 году этот стандарт был утвержден и ISO. В 1989 году стандарт претерпел незначительные изменения, но именно этот вариант получил название SQL-1 или SQL-89.

В чем особенность SQL-89? За время разработки стандарта (1982–1989 гг.) были созданы, представлены на рынке и активно использовались несколько различных СУБД, в которых в том или ином виде был реализован некоторый диалект языка SQL. С учетом того, что разработкой стандартов занимались те же люди, кто внедрял SQL в СУБД, стандарт SQL-89 представлял собой плод множества компромиссов, приведших к наличию в нем большого количества «белых пятен», т.е. мест, которые не были описаны, а отданы на усмотрение разработчиков диалекта. В результате чуть ли не все имеющиеся диалекты стали совместимыми со стандартом, но особой пользы это не принесло.

Следующая реализация стандарта была призвана решить эту проблему. В результате длительных обсуждений и согласований в 1992 году был принят новый стандарт ANSI SQL-2 или SQL-92. SQL-92 заполнил многие «белые пятна», впервые добавив в стандарт возможности, еще не реализованные в существующих коммерческих СУБД.

Работа над стандартизацией продолжается и сейчас. Конечно, SQL-92 не решил всех проблем, связанных с наличием нескольких диалектов языка. Одно только описание стандарта разрослось от ста страниц (SQL-89) до 600 (SQL-92). Кроме того, все разработчики как игнорировали, так и игнорируют некоторые положения стандарта, с одной стороны, отказываясь реализовывать некоторые его части и, с другой стороны, реализуя то, что отсутствует в стандарте. Однако не все так плохо, как может показаться. Несмотря на имеющиеся отличия, все коммерческие СУБД поддерживают некоторое ядро языка, описанное в стандарте SQL-92, одинаково. Отличий не очень много, они не носят слишком принципиального характера. Хотя каждая СУБД попрежнему поддерживает свой диалект языка. Среди этих диалектов есть некоторые более «уважаемые» многими разработчиками, чем непосредственно стандарт, например диалект компании IBM, родоначальника SQL. Компания IBM, выпуская СУБД DB2 и Informix, попрежнему удерживает существенные позиции на рынке серверных СУБД.

6.3.2. Достоинства языка SQL

Для ознакомления с достоинствами языка обратимся к соответствующей литературе [8]. Вот некоторые из них:

- межплатформенная переносимость;
- наличие стандартов;
- одобрение и поддержка компанией IBM (СУБД DB2);
- поддержка со стороны компании Microsoft (СУБД SQL Server, протокол ODBC и технология ADO);
- реляционная основа;
- высокоуровневая структура;
- возможность выполнения специальных интерактивных запросов;
- обеспечение программного доступа к базам данных;
- возможность различного представления данных;
- полноценность как языка, предназначенного для работы с базами данных;
- возможность динамического определения данных;
- поддержка архитектуры клиент/сервер;
- поддержка корпоративных приложений;
- расширяемость и поддержка объектно-ориентированных технологий;
- возможность доступа к данным в Интернете;
- интеграция с языком Java (протокол JDBC);
- промышленная инфраструктура.

6.3.3. Разновидности SQL

Стандарты языка SQL регламентируют синтаксис операторов. Если посмотреть на операторы языка, становится понятно, что в отличие от «обычных» языков программирования в SQL отсутствует возможность объявления переменных, нет инструкции IF, нет цикла FOR и т.д. Одним словом, в таком виде язык годился исключительно для интерактивного режима работы с базой данных: пользователь вводит запрос – получает результат, вводит другой запрос – получает другой результат и т.д. Программирование на подобном языке представлялось крайне затруднительным. Естественно, дело этим не кончилось. Технологии продолжают двигаться вперед, и на настоящий момент известны следующие разновидности языка SQL:

- интерактивный SQL;
- программный (встроенный) SQL:
  - статический SQL;

– динамический SQL;
– API – интерфейсы вызова подпрограмм.

Разберем эти разновидности подробнее.

Интерактивный SQL будет рассмотрен в данном учебнике подробнее, чем программный. Для всех разновидностей SQL будут приведены основные идеи и рассмотрены ключевые концепции. Детальное рассмотрение статического, динамического SQL и различных API-интерфейсов (ODBC, JDBC, DB Library и др.) выходит за рамки нашего курса. Подробное изучение этих тем выносится на специальные курсы, посвященные данной проблематике.

6.4. Понятие интерактивного SQL. Элементы интерактивного SQL. Использование SQL для манипулирования данными

Итак, **интерактивный SQL** предусматривает непосредственную работу пользователя с базой данных по следующему алгоритму: используя прикладную программу (клиентское приложение) или стандартную утилиту, входящую в СУБД, пользователь:

- устанавливает соединение с БД (подтверждая наличие прав доступа);
- вводит текст SQL-запроса;
- запускает запрос на выполнение.

Текст запроса поступает в СУБД, которая:

- осуществляет синтаксический анализ запроса (проверяет, является ли запрос корректным);
- проверяет, имеет ли пользователь право выполнять подобный запрос (может быть, пользователь пытается что-то удалить, а права есть только на чтение);
- выбирает, каким образом осуществлять выполнение запроса – план выполнения запроса;
- выполняет запрос;
- результат выполнения отсылает пользователю.

В данном разделе кратко рассмотрим ту часть языка SQL, которая предназначена для манипулирования данными, находящимися в таблицах реляционной базы данных. Соответствующую подсистему SQL обычно называют DML (Data Manipulation Language) – язык манипулирования данными. Как было указано в предыдущих разделах, DML

содержит операторы для внесения изменений в содержимое таблиц базы данных.

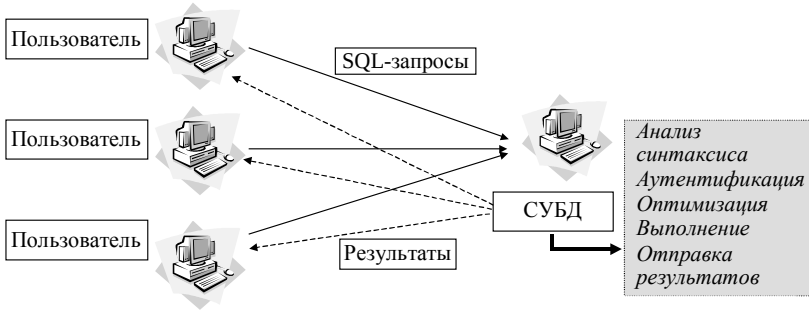

Рис. 46. Схема работы интерактивного SQL

В ходе рассмотрения будут представлены примеры SQL-запросов. Эти примеры предназначены для лучшего понимания изложения идеологии языка и синтаксиса операторов. Примеры основаны на базе данных, приведенной выше по тексту.

6.4.1. Использование SQL для выбора информации из таблицы

Простейший запрос

Вообще говоря, под запросом можно понимать команду, посылаемую пользователем системе управления базами данных для выполнения того или иного действия. С точки зрения русского языка это определение несколько отличается от того смысла, который люди, как правило, вкладывают в слово «запрос». Так, позже мы увидим, что далеко не все запросы связаны именно с извлечением какой-то информации из таблиц, однако в данном разделе рассматриваются именно такие запросы.

Простейший запрос – команда СУБД, которая указывает ей, что она должна выбрать определенные данные из одной конкретной таблицы. Для подобных запросов (и не только для них, но и для значительно более сложных) предназначена команда SELECT.

Команда SELECT

Простейший запрос на выборку может выглядеть следующим образом:

```
SELECT  *
FROM    entrant
```

В результате этого запроса будут выбраны все данные об абитуриентах, содержащиеся в таблице entrant.

Аналогичного результата можно достигнуть, явным образом перечислив через запятую названия всех столбцов таблицы после ключевого слова SELECT. Вот как это могло бы выглядеть:

```
SELECT  id, name, second_name,
        surname, birth_date, birth_place,
        distinction_id, citizen_id, education_id,
        sex, rem, original, mod_date, mod_user
FROM    entrant
```

Естественно, перечисление всех названий столбцов не имеет особого смысла и приносит только дополнительные неудобства по сравнению с использованием символа *. Кроме того, первый способ сработает правильно и после добавления новых столбцов в таблицу, чего не скажешь о втором.

Легко догадаться, что́ нужно сделать, если не все столбцы таблицы нужны в качестве результатов запроса:

```
SELECT  name, second_name, surname
FROM    entrant
```

Данный запрос приведет к включению в результат только трех столбцов таблицы entrant.

А что делать, если вам нужно переупорядочить столбцы в результате запроса? Решение следующее – необходимо поменять порядок их следования в первой части команды SELECT:

```
SELECT  surname, name, second_name
FROM    entrant
```

Рассмотрим еще один момент, связанный с простейшей формой оператора SELECT. Иногда в качестве результатов запроса вы получаете набор строк, некоторые из которых дублируются. Так, например, вы хотели бы получить список населенных пунктов, в которых родились абитуриенты. Как было показано выше, для этого достаточно применить следующий запрос:

```
SELECT  birth_place
FROM    entrant
```

Однако есть одна проблема: в качестве результата вы, скорее всего, получите список, содержащий одинаковые значения, например:

```
Нижний Новгород
Москва
Нижний Новгород
Нижний Новгород
Дзержинск
Нижний Новгород
Нижний Новгород
Козино
Нижний Новгород
Нижний Новгород
Нижний Новгород
Лукино
```

Обычно это не то, что требуется. Для получения нужного результата следует удалить из этого списка повторяющиеся города. Этот результат достигается при помощи ключевого слова DISTINCT:

```
SELECT DISTINCT birth_place
FROM    entrant
```

В результате выполнения запроса вы получите:

```
Нижний Новгород
Москва
Дзержинск
Козино
Лукино
```

Противоположность DISTINCT – ключевое слово ALL. Во всех предыдущих запросах мы ничего не указывали после SELECT и ALL подразумевалось по умолчанию.

Квалифицированный выбор – предложение WHERE. Начало использования

Используя перечисление имен столбцов непосредственно после ключевого слова SELECT, вы могли ограничить результат запроса лишь несколькими необходимыми столбцами. Аналогичные действия необходимо производить и для строк. Так, очень часто вам требуется выбрать из таблицы данные, удовлетворяющие определенным услови-

ям. Для реализации этой возможности в языке SQL существует ключевое слово WHERE. Ключевое слово WHERE пишется в команде SELECT после окончания секции FROM и содержит после себя логическое выражение, которое проверяется последовательно для каждой строки рассматриваемой таблицы.

Рассмотрим несколько несложных примеров запросов:

```
SELECT  name, second_name, surname
FROM    entrant
WHERE   birth_place = 'Нижний Новгород'
```

Этот запрос распечатает имена, отчества, фамилии все абитуриентов, родившихся в Нижнем Новгороде.

```
SELECT  name, second_name, surname
FROM    entrant
WHERE   birth_place <> 'Нижний Новгород'
```

Этот запрос распечатает имена, отчества, фамилии все абитуриентов, родившихся не в Нижнем Новгороде.

```
SELECT  name, second_name, surname
FROM    entrant
WHERE   birth_date > '01.01.1984'
```

Этот запрос распечатает имена, отчества, фамилии все абитуриентов, родившихся после 1 января 1984 года.

```
SELECT  name, second_name, surname
FROM    entrant
WHERE   birth_date <= '01.01.1986'
```

Этот запрос распечатает имена, отчества, фамилии все абитуриентов, родившихся до 2 января 1986 года.

Квалифицированный выбор – предложение WHERE. Использование реляционных и булевых операторов

Нетрудно видеть, что современные языки программирования высокого уровня предоставляют гораздо более мощные средства для составления условий, чем указано выше. Не остался в стороне и SQL. Так, SQL допускает формирование сложных составных условий с использованием реляционных и булевых операторов.

Под реляционными операторами в данном случае понимаются:

> – больше;
< – меньше;
>= – больше или равно;
<= – меньше или равно;
= – равно;
<> – не равно.

Примеры использования реляционных операторов мы рассмотрели выше.

Под булевыми операторами в данном случае понимают следующие:

AND – и;
OR – или;
NOT – не.

При помощи булевых операторов вы можете создавать составные условия, комбинируя различные ограничения. Рассмотрим примеры соответствующих запросов:

```
SELECT  name, second_name, surname
FROM    entrant
WHERE   (birth_place = 'Нижний Новгород') AND
        (birth_date <= '01.01.1986')
```

Этот запрос выведет в качестве результата данные (имена, отчества, фамилии) всех абитуриентов, которые родились в Нижнем Новгороде ранее 2 января 1986 года.

```
SELECT DISTINCT entrant_id
FROM     mark
WHERE   (mark >= 2)
    AND (mark <= 3)
```

Этот запрос выведет в качестве результата идентификаторы всех абитуриентов, которые получили на экзаменах хотя бы одну двойку или тройку.

```
SELECT  name, second_name, surname
FROM    entrant
WHERE   NOT (birth_place = 'Нижний Новгород')
```

Этот запрос выведет в качестве результата данные всех абитуриентов, которые родились не в Нижнем Новгороде.

Квалифицированный выбор – предложение WHERE. Использование специальных операторов

Наряду с рассмотренными операторами для составления условий, мало чем отличающимися от таких же операторов, существующих в языках программирования высокого уровня, в языке SQL (из-за специфики области применения) существуют ряд специальных операторов, которые, как правило, не имеют аналогов в других языках. Вот эти операторы:

IN – вхождение в некоторое множество значений;
BETWEEN – вхождение в некоторый диапазон значений;
LIKE – проверка на совпадение с образцом;
IS NULL – проверка на неопределенное значение.

Оператор IN используется для проверки вхождения в некоторое множество значений. Так, запрос

```
SELECT  DISTINCT entrant_id
FROM    mark
WHERE   mark IN (2,3)
```

выведет идентификаторы всех абитуриентов, получивших хотя бы одну двойку или тройку на экзаменах.

Того же результата можно добиться, используя оператор BETWEEN:

```
SELECT  DISTINCT entrant_id
FROM    mark
WHERE   mark BETWEEN 2 AND 3
```

Оператор LIKE применим исключительно к символьным полям и позволяет устанавливать, соответствует ли значение поля образцу. Образец может содержать специальные символы:

_ (символ подчеркивания) – замещает любой одиночный символ;
% (знак процента) – замещает последовательность любого числа символов.

Так, например, следующий запрос покажет данные всех абитуриентов, фамилии которых начинаются с буквы «А»:

```
SELECT  name, second_name, surname
FROM    entrant
WHERE   surname LIKE 'A%'
```

Оператор IS NULL позволяет обнаруживать в таблице неопределенные значения (напомним, что подобные значения называются NULL).

Так, например, следующий запрос покажет данные всех абитуриентов, которые не указали места своего рождения:

```
SELECT  name, second_name, surname
FROM    entrant
WHERE   birth_place IS NULL
```

Использование агрегатных функций. Простые запросы

Очень часто возникает необходимость произвести вычисление минимальных, максимальных или средних значений в столбцах. Так, например, вам может понадобиться вычислить средний балл или что-то другое. Для осуществления подобных вычислений в языке программирования высокого уровня вы предприняли бы некоторые действия, в частности организовали бы цикл, сосчитали сумму, потом разделили ее на количество слагаемых в сумме, получив значение среднего балла. SQL избавляет вас от необходимости программирования всей последовательности подобных действий. Для этого SQL предоставляет специальные агрегатные функции:

MIN – минимальное значение в столбце;
MAX – максимальное значение в столбце;
SUM – сумма значений в столбце;
AVG – среднее значение в столбце;
COUNT – количество значений в столбце, отличных от NULL.

Так, следующие запросы считают, соответственно, минимум, максимум, сумму, среднее среди всех баллов, полученных студентами на экзаменах. Количество оценок считает последний запрос.

```
SELECT  MIN(mark)
FROM    mark
SELECT  MAX(mark)
FROM    mark
SELECT  SUM(mark)
FROM    mark
SELECT  AVG(mark)
FROM    mark
SELECT  COUNT(mark)
FROM    mark
```

Заметим, что, изменив следующим образом текст последнего запроса, мы получим количество разных оценок, полученных на экзаменах:

```
SELECT  COUNT(DISTINCT mark)
FROM    mark
```

Использование агрегатных функций. Комбинирование с предложением WHERE

Дополнительно к описанным выше простым случаям, вы можете использовать агрегатные функции совместно с предложением WHERE:

```
SELECT  AVG(mark)
FROM    mark
WHERE   entrant_id = 100
```

Данный запрос вычислит средний балл студента с кодом 100 по результатам всех сданных им экзаменов.

```
SELECT  AVG(mark)
FROM    mark
WHERE   exam_id = 10
```

Данный запрос вычислит средний балл студентов по результатам сдачи экзамена с кодом 10.

Нетрудно видеть, что агрегатные функции, в сочетании с ограничением области данных при помощи предложения WHERE, представляют собой мощный аппарат, позволяющий ответить на многие реально возникающие вопросы путем применения SQL-запросов.

Использование агрегатных функций и группировка. Предложение GROUP BY

В дополнение к рассмотренным механизмам язык SQL предоставляет мощный аппарат для вычисления агрегатных функций не для всей таблицы результатов запроса, а для разных значений по группам.

Для этого в SQL существует специальная конструкция GROUP BY, предназначенная для указания того столбца, по значениям которого будет производиться группировка.

Так, например, мы можем вычислить средний балл по всем экзаменам для каждого студента. Для этого достаточно выполнить следующий запрос:

```
SELECT     entrant_id, AVG(mark)
FROM       mark
GROUP BY   entrant_id
```

Все это, как обычно, может быть совмещено с предложением WHERE. При этом, не вдаваясь в тонкости выполнения запроса внутри СУБД, вы можете считать, что сначала выполняется выборка тех строк таблицы, которые удовлетворяют условиям из предложения WHERE, а потом производится группировка и агрегирование.

Приведем запрос, который вычисляет средний балл по оценкам, полученным на экзамене с кодом 100, для каждого студента. Для этого достаточно выполнить следующий запрос:

```
SELECT     entrant_id, AVG(mark)
FROM       mark
WHERE      exam_id = 100
GROUP BY   entrant_id
```

Заметим, что группировка может производиться более чем по одному полю. Так, вы можете вычислить средний балл, введя дополнительное разграничение по дням, в которые вносилась информация в базу (считая, например, что эта дата совпадает с датой проведения экзамена):

```
SELECT     entrant_id, mod_date, AVG(mark)
FROM       mark
WHERE      exam_id = 100
GROUP BY   entrant_id, mod_date
```

Для запросов, содержащих секцию GROUP BY существует важное ограничение: такие запросы могут включать в качестве результата столбцы, по которым производится группировка, и столбцы, которые содержат собственно результаты агрегирования.

Использование агрегатных функций и группировка. Ограничение результатов запроса. Предложение HAVING

В языке SQL присутствуют средства, которые позволяют сделать следующее: допустим, вы хотите получить в качестве результатов запроса данные только тех абитуриентов, средний балл которых превышает 4. Как это сделать? Проблема в том, что стандарт языка не допускает использования агрегатных функций в предложении WHERE. Получается, что сначала необходимо выполнить запрос и сосчитать

среднее, а потом ограничить результат по условию «Среднее > 4». Для осуществления подобных действий стандартом языка предусмотрена специальная секция HAVING. Так, запрос может выглядеть следующим образом:

```
SELECT     entrant_id, mod_date, AVG(mark)
FROM       mark
WHERE      exam_id = 100
GROUP BY   entrant_id, mod_date
HAVING     AVG(mark) > 4
```

Форматирование вывода. Выражения в запросе. Упорядочение

Для того чтобы форматировать вывод, вы можете использовать различные возможности SQL. Так, например, допустимым является включение текста в запрос. Рассмотрим пример того, как это делается:

```
SELECT  'Средний балл=', AVG(mark)
FROM    mark
WHERE   exam_id = 10
```

В результате данного запроса вы увидите не просто некоторое число, а число, сопровожденное поясняющим текстом.

Наряду с обычной выборкой значений из столбцов таблицы имеется возможность вносить изменения в эти значения путем вычисления выражений.

Так, например, вы можете перейти от 5-балльной к 10-балльной системе, преобразовав значение среднего балла следующим образом:

```
SELECT  'Средний балл=', AVG(mark)*2
FROM    mark
WHERE   exam_id = 10
```

Следующий важный момент связан с осуществлением сортировки (упорядочения) данных. Для организации сортировок в стандарте SQL предусмотрена специальная секция ORDER BY. В этой секции вам необходимо указать, по какому столбцу (по каким столбцам) упорядочивать результаты запроса. При этом при использовании нескольких столбцов упорядочение будет производиться последовательно (по первому критерию, если значения совпали – по второму критерию и т.д.).

Следующий запрос распечатывает список абитуриентов, в котором фамилии следуют в алфавитном порядке:

```
SELECT   name, second_name, surname
FROM     entrant
ORDER BY surname
```

По умолчанию подразумевается, что сортировка производится по возрастанию. Если вам необходимо упорядочивать по убыванию, следует использовать указание «DESC»:

```
SELECT   name, second_name, surname
FROM     entrant
ORDER BY surname DESC
```

Для упорядочения по возрастанию присутствует указание «ASC», но его обычно не пишут явным образом.

При упорядочении по нескольким столбцам вы можете упорядочивать как по возрастанию, так и по убыванию, указывая соответствующую конструкцию после наименования каждого столбца:

```
SELECT   name, second_name, surname
FROM     entrant
ORDER BY surname DESC, name ASC
```

Наряду с упорядочиванием в обычных запросах стандарт SQL позволяет сортировать и запросы, содержащие агрегирование. В этом случае порядок следования секций следующий:

```
SELECT   entrant_id, mod_date, AVG(mark)
FROM     mark
WHERE    exam_id = 100
GROUP BY entrant_id, mod_date
HAVING   AVG(mark) > 4
ORDER BY entrant_id
```

6.4.2. Использование SQL для выбора информации из нескольких таблиц

До сих пор мы рассматривали ту часть SQL, которая касалась выбора информации из единственной таблицы. Естественно, возможности языка этим не ограничиваются. Так, вы можете запрашивать информацию из нескольких таблиц, реализуя описанные в соответствующем разделе учебника реляционные операции. Рассмотрим некоторые примеры того, как это делается. Стоит упомянуть, что полное рассмотрение темы выходит за рамки данного учебника. Подробно

этот вопрос можно изучить при помощи, например, [7, 8]. Мы рассмотрим некоторые несложные случаи, часто встречающиеся на практике, когда появляется необходимость выбора информации из нескольких таблиц сразу.

Как правило, в тех случаях когда возникает необходимость выбирать информацию из разных таблиц, они тем или иным образом связаны друг с другом, например отношениями один к многим или один к одному по некоторому полю.

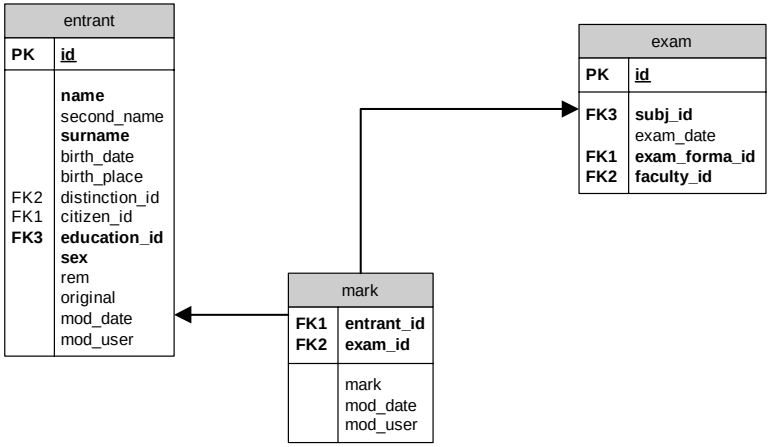

Рис. 47. Пример связанных таблиц

В примере (рис. 47) тоже присутствуют такие таблицы. Рассмотрим таблицы entrant, mark и exam.

Таблица mark (оценки) связана с таблицей exam (экзамены) по полям **id – exam_id**.

Таблица mark (оценки) связана с таблицей entrant (абитуриенты) по полям **id – entrant_id**.

Допустим, нам требуется распечатать список абитуриентов с оценками, которые они получили на экзаменах. Для этого необходимо выполнить следующий запрос:

```
SELECT  entrant.surname, mark.exam_id, mark.mark
FROM    entrant, mark
WHERE   entrant.id = mark.entrant_id
```

Отметим следующие изменения по сравнению с теми запросами, которые мы писали ранее:

1. В секции FROM указаны две таблицы.
2. Так как таблиц стало больше одной, появилась некоторая неоднозначность при упоминании полей. Так, во многих случаях неизвестно, из какой таблицы из списка FROM брать поле. Для устранения неоднозначности имена полей указываются с префиксом – именем таблицы. Имя таблицы от имени поля отделяется точкой.
3. В предложении WHERE указано условие соединения таблиц.

Предположим, нас не удовлетворяет тот факт, что мы видим код предмета, но не видим даты сдачи экзамена. Для того чтобы решить эту проблему, необходимо выполнить следующий запрос:

```
SELECT  entrant.surname, exam.exam_date, mark.exam_id,
        mark.mark
FROM    entrant, mark, exam
WHERE   (entrant.id = mark.entrant_id) AND
        (exam.id = mark.exam_id)
```

Нетрудно заметить, что использование префиксов-имен таблиц сильно загромождает запрос. Для того чтобы избежать подобного загромождения, используются псевдонимы. Так, мы можем переписать предыдущий запрос следующим образом:

```
SELECT  E.surname, X.exam_date, M.exam_id, M.mark
FROM    entrant E, mark M, exam X
WHERE   (E.id = M.entrant_id) AND
        (E.id = M.exam_id)
```

6.4.3. Использование SQL для вставки, редактирования и удаления данных в таблицах

Для добавления данных в таблицу в стандарте SQL предусмотрена команда INSERT. Синтаксис команды проще всего понять из следующего запроса:

```
INSERT INTO mark
VALUES (1, 2, 5, '01.08.2003', 11)
```

Данный запрос вставляет в таблицу mark строку, содержащую значения, перечисленные в списке VALUES.

Если вы не хотите указывать значение какого-то поля, можете присвоить ему NULL:

```
INSERT INTO mark
VALUES (1, 2, 5, '01.08.2003', NULL)
```

В случае если вы считаете необходимым использование для некоторых полей значений по умолчанию, SQL позволяет явно указать, какие поля необходимо заполнить вашими данными, а какие – значениями по умолчанию:

```
INSERT INTO mark (entrant_id, exam_id, mark)
VALUES (1, 2, 5)
```

Кроме явного указания значений, вы также можете вставить в таблицу данные, представляющие собой результат выполнения другого запроса. Так, пусть у вас есть вспомогательная таблица entrant_temp с полями id, surname:

```
INSERT INTO entrant_temp (surname)
        SELECT      E.surname
        FROM        entrant E
        ORDER BY    E.surname
```

Для удаления данных из таблицы существует команда DELETE:

```
DELETE
FROM    entrant_temp
```

Этот запрос удаляет все данные из таблицы entrant_temp.

Вы можете ограничить диапазон удаляемой информации следующим образом:

```
DELETE
FROM    entrant_temp
WHERE   surname > 'И'
```

Для обновления данных в таблице существует команда UPDATE. Так, запрос

```
UPDATE  mark
SET     mark = '5'
```

производит обновление таблицы mark, меняя в ней значения всех оценок на «5».

При необходимости, вы можете ограничить состав обновляемых записей. Например, это может быть сделано следующим образом:

```
UPDATE  mark
SET     mark = '5'
WHERE   mark = '4'
```

При помощи этого запроса вы поднимете балл на 1 у тех абитуриентов, кто получил на экзамене «4».

Итак, для изменения данных, содержащихся в таблицах, предназначены команды SQL INSERT, DELETE и UPDATE.

6.4.4. Язык SQL и операции реляционной алгебры

Язык SQL является средством выражения мощного математического аппарата теории множеств и реляционной алгебры. В данном разделе рассматривается связь операторов языка SQL с операциями реляционной алгебры и теории множеств.

Операция объединения

Средствами языка SQL операция объединения представляется следующим образом:

```
SELECT *
FROM A
UNION
SELECT *
FROM B
```

Операция разности

Средствами языка SQL операция разности представляется следующим образом:

```
SELECT *
FROM A
EXCEPT
SELECT *
FROM B
```

Операция проекции

```
SELECT Field_i1, … , Field_in
FROM A
```

Операция выборки (селекции)

```
SELECT *
FROM A
WHERE (<condition>)
```

Операция пересечения

```
SELECT *
FROM A
INTERSECT
SELECT *
FROM B
```

Операция соединения, эквисоединения

```
SELECT A.Field1, … , A.Fieldn, B.Field1, … , B.Fieldm
FROM A, B
WHERE (A.Fieldi Θ B.Field1)
```

Если Θ – операция «=», то это эквисоединение.

Операция естественного соединения

Пусть есть отношения A (X_1, … , X_n, A_1, …, A_m) и B (X_1, … , X_n, B_1, …, B_r).

```
SELECT A.X1, … , A.Xn, A.A1, … , A.Am,  B.B1, … , B.Br
FROM A, B
WHERE (A.X1 = B.X1) AND … AND (A.Xn = B.Xn)
```

6.5. Программный (встроенный) SQL

Программный (**встроенный**) **SQL** (Embedded SQL) предназначен для того, чтобы встраивать SQL-запросы в прикладную программу. Представьте себе, что вы – программист, разрабатывающий приложение для работы с базами данных. Совершенно, очевидно, что для вас является неприемлемой интерактивная форма работы. Вместо этого вам необходимо каким-то образом встроить SQL-запросы в программу, написанную, скажем, на языке C++. Но как это сделать?

- Как объяснить компилятору C++, что часть вашей программы – это не операторы C++, а операторы какого-то SQL?
- Как соединить возможности языка программирования высокого уровня (переменные, ветвления, циклы) и возможности SQL (запросы на языке, близком к естественному, где уже не на програм-

миста, а на СУБД ложится труд по их оптимизации и выбору плана выполнения)?

Эти и некоторые другие проблемы решаются несколькими способами. Эти способы частично описаны и в стандарте SQL-92. На настоящий момент используются три варианта программного SQL: статический, динамический и основанный на различных API. Посмотрим, что представляет собой каждый вариант.

6.5.1. Статический SQL

Статический SQL – разновидность программного SQL, предназначенная для встраивания SQL-операторов в текст программы на языке программирования высокого уровня.

Рассмотрим еще раз алгоритм выполнения SQL-запросов в интерактивном режиме работы. Легко видеть, что пользователь вынужден ожидать результатов выполнения запроса в течение всего времени работы алгоритма. Если через некоторое время пользователю снова нужно будет выполнить тот же самый запрос, СУБД вновь проделает те же самые действия, что и при предыдущем обращении. Налицо некоторое несовершенство механизма:

- одни и те же этапы выполняются каждый раз заново для одинаковых запросов;
- СУБД не может обрабатывать интерактивные запросы с опережением.

Решение подобных проблем очевидно – часть действий по обработке запроса необходимо выполнять один раз, сохранять результат в некотором виде, а потом использовать столько раз, сколько необходимо. Эта идея является одной из основных идей программного SQL. Итак, программный, а в частности и статический, SQL позволяет:

- использовать операторы интерактивного SQL в тексте программы на языке программирования высокого уровня;
- наряду с операторами интерактивного SQL использовать новые специальные конструкции, дополняющие SQL и увеличивающие его возможности;
- для передачи параметров в запрос использовать в тексте запроса переменные, объявленные в программе;
- для возврата в программу результатов запроса использовать специальные конструкции, отсутствующие в интерактивном SQL;
- осуществлять компиляцию запросов совместно с программой, обеспечивая впоследствии согласованную работу программы и

СУБД. Заранее (на этапе компиляции) выполнять действия по анализу и оптимизации запросов, экономя время, затрачиваемое на этапе выполнения программы.

Рассмотрим два основных этапа, связанных с работой статического SQL, – компиляция программы и работа (выполнение) программы.

Схема компиляции и сборки программы выглядит следующим образом (рис. 48):

- Программа, включающая операторы языка программирования высокого уровня (ЯПВУ) вместе с операторами SQL, подается на вход специального препроцессора, который выделяет из нее части, связанные с SQL.
- Вместо инструкций встроенного SQL препроцессор подставляет вызовы специальных функций СУБД. Библиотеки таких функций для связи с языками программирования существуют для всех распространенных серверных СУБД. Стоит особо отметить, что эти библиотеки имеют «закрытый» интерфейс, т.е. создатели могут менять его по своему усмотрению, соответственно обновив препроцессор. Все это говорит о том, что программист не должен вмешиваться в этот процесс.
- Сами инструкции SQL препроцессор выделяет в отдельный файл.
- Программа поступает на вход обычного компилятора языка программирования, после чего получаются объектные модули. Далее эти объектные модули вместе с библиотеками СУБД собираются в один исполняемый модуль – приложение.
- Наряду с этими операциями происходит работа с файлом, содержащим SQL-инструкции. В литературе этот модуль часто носит название «модуль запросов к базе данных» (Database Request Module, DBRM) [8]. Обработку этого модуля осуществляет специальная утилита, которая обычно носит название BIND. Для каждой инструкции SQL утилита выполняет следующие действия:
  - осуществляет синтаксический анализ запроса (проверяет, является ли запрос корректным);
  - проверяет, существуют ли в базе данных те объекты, на которые ссылается запрос;
  - выбирает, каким образом осуществлять выполнение запроса – план выполнения запроса;
- Все планы выполнения запросов сохраняются в СУБД для последующего использования.

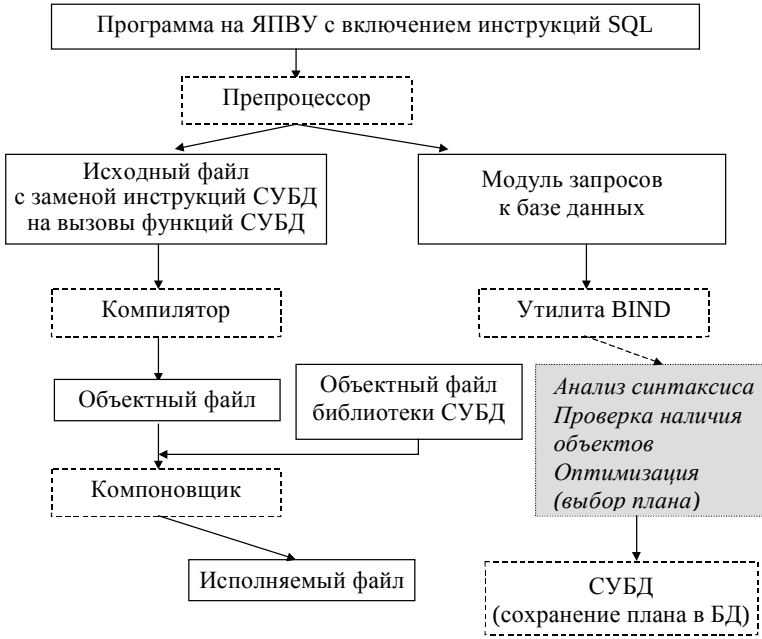

Рис. 48. Схема компиляции программы с встроенными инструкциями статического SQL

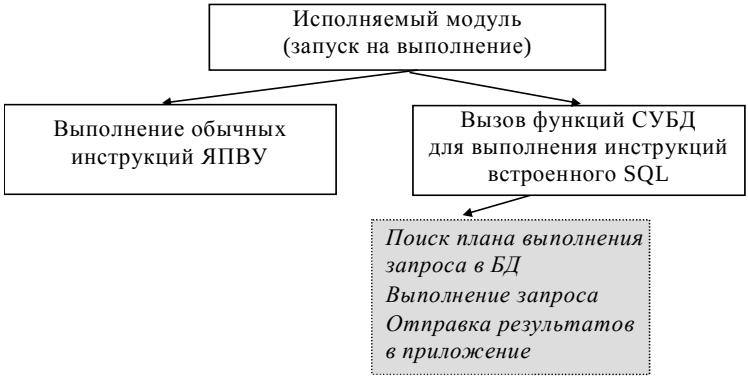

Рис. 49. Схема выполнения программы с встроенными инструкциями статического SQL

Схема выполнения программы выглядит следующим образом (рис. 49):

- Программа запускается на выполнение обычным образом.
- При необходимости выполнить запрос программой осуществляется вызов специальной функции СУБД, которая отыскивает уже сформированный ранее план выполнения запроса.
- СУБД выполняет запрос в соответствии с выбранным планом. Результат выполнения запроса поступает в приложение.

Основные **команды** статического SQL

EXEC SQL	Спецификатор, указывающий, что следующая за ним инструкция является инструкцией встроенного SQL
;	В языке С – признак окончания инструкции встроенного SQL
DECLARE TABLE	Объявляет таблицу, которая потом будет использоваться в инструкциях встроенного SQL
SQLCODE	Переменная для обработки ошибок
SQLSTATE	Переменная для обработки ошибок
GET DIAGNOSTICS	Инструкция для обработки ошибок
WHENEVER **SQLERROR** **SQLWARNING** **NOT FOUND** **GOTO** **CONTINUE**	Набор совместно используемых инструкций для упрощения обработки ошибок
BEGIN DECLARE SECTION **END DECLARE SECTION**	Инструкции для определения области, в которой будут объявлены переменные, впоследствии используемые в запросах SQL
INTO	Используется в операторе SELECT для указания переменной, в которую необходимо поместить результат выполнения запроса

DECLARE CURSOR	**Курсор** – специальный инструмент, предназначенный для обработки результатов запроса, содержащих более одной строки. Работа с курсором похожа на работу с файлами. Данная инструкция служит для создания курсора и связывания его с конкретным запросом
OPEN	Команда, открывающая курсор и побуждающая СУБД начать выполнение запроса. Устанавливает курсор перед первой строкой результата запроса
FETCH	Команда, перемещающая указатель текущей строки (курсор) на следующую строку. В некоторых СУБД и стандарте SQL-92 реализованы разные формы команды FETCH, перемещающие курсор на произвольную строку результатов запроса
CLOSE	Закрывает курсор и прекращает доступ к результатам запроса

Использование описанной выше схемы компиляции/сборки/выполнения программы позволяет:

- использовать SQL совместно с программой на языке программирования высокого уровня;
- заранее осуществлять проверку синтаксиса запросов и оптимизацию их выполнения (выбор плана). Понятно, что проверка синтаксиса выполняется быстро, но выбор плана выполнения – весьма трудоемкая процедура. Тот факт, что она выполняется один раз на этапе компиляции, позволяет говорить о существенном уменьшении накладных расходов.

Однако статическая разновидность программного SQL имеет некоторые существенные ограничения. Так, переменные в запросах могут использоваться только в тех местах, где в запросах обычно стоят константы. Например, вы не можете параметризовать имя таблицы, из которой производится выборка, а также названия столбцов. В результате мы приходим к следующему выводу.

При использовании статического варианта вложенного (программного) SQL необходимо на этапе написания программы точно знать состав запросов, которые необходимо будет выполнять в программе.

Во многих случаях это ограничение является существенным. Для его устранения была введена новая разновидность программного SQL – динамический SQL. Рассмотрим кратко основные идеи динамического SQL.

6.5.2. Динамический SQL

Динамический SQL – разновидность программного SQL, предназначенная для встраивания SQL-операторов в текст программы на языке программирования высокого уровня, допускающая динамическое формирование и выполнение запросов во время работы программы.

История возникновения динамического SQL во многом связана с компанией IBM, внедрившей этот мощный инструмент в свою СУБД DB2. Стандарты SQL, в частности SQL-1, не поддерживали динамического SQL. Лишь в 1992 году в стандарт SQL-2 были включены спецификации динамического SQL. Для того чтобы использовать этот инструментарий в своих программах, необходимо представлять основные концепции, основные идеи, заложенные в динамический SQL, и принципы его функционирования. Именно это и является предметом рассмотрения в данном разделе.

Теоретики считают основной концепцией динамического SQL следующее утверждение: 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 [8].

Вспомним, как работал статический SQL: схема использования подразумевала два этапа – компиляцию программы и выполнение программы. При этом на этап компиляции ложилась основная нагрузка в том смысле, что он решал вопросы проверки, разбора и оптимизации запросов, поскольку было известно, что́ именно разбирать и оптимизировать. Совершенно очевидно, что подобную двухэтапную схему нельзя реализовать для динамического SQL, ведь на этапе компиляции программы еще неизвестно, что́ именно нужно разбирать и оптимизировать.

Каковы последствия этого изменения? Любой механизм имеет свои достоинства и недостатки.

Достоинство динамического SQL в том, что он позволяет формировать запрос к базе данных во время работы программы, реагируя на те или иные произошедшие события. Такая возможность является жизненно важной для клиент-серверной и трехзвенной архитектур, в которых структура базы данных и деловые правила имеют тенденцию к изменению, что требует определенной гибкости при организации процесса обработки данных.

Недостаток динамического SQL – низкая производительность по сравнению со статическим. Действительно, многие из операций, выполнявшихся на этапе компиляции один раз, теперь выполняются непосредственно во время работы программы. В связи с этим там, где только возможно, представляется правильным рекомендовать использование статической разновидности SQL, применяя аппарат динамического SQL там, где это действительно необходимо.

Рассмотрим схему функционирования динамического SQL. Схема предусматривает одноэтапное и двухэтапное выполнение инструкций.

Одноэтапное выполнение инструкций осуществляется командой EXECUTE IMMEDIATE.

Рис. 50. Схема выполнения программы со встроенными инструкциями динамического SQL с применением одноэтапной схемы

Схема выполнения инструкции подразумевает:

- динамическое формирование команды SQL в строковом виде во время работы программы;

- передачу строкового вида инструкции в СУБД при помощи команды EXECUTE IMMEDIATE;
- выполнение инструкции системой управления БД, включающее синтаксический анализ, проверку параметров, оптимизацию (выбор плана) и выполнение этого плана.

Основные проблемы одноэтапной схемы заключаются в том, что она не позволяет выполнять инструкции SELECT (ибо нет средств для возврата в приложение результатов запроса) и приводит к нерациональному расходованию вычислительных ресурсов (т.к. при повторном выполнении той же инструкции вновь будет затрачено время на все те же действия по ее интерпретации и выполнению).

Двухэтапное выполнение инструкций основано на следующем соображении: скорее всего, команда динамического SQL в таком виде, как она поступает на выполнение, будет выполняться неоднократно. При этом могут меняться какие-то конкретные детали. А это значит, что инструкцию можно параметризовать. Использование параметризованных инструкций позволяет сделать схему выполнения двухэтапной, разделив процесс на «подготовку инструкции» и «выполнение инструкции».

Рис. 51. Схема выполнения программы со встроенными инструкциями динамического SQL с применением двухэтапной схемы

На этапе подготовки можно осуществить синтаксический анализ инструкции, интерпретировать ее и подготовиться к выполнению, выбрав план выполнения.

На этапе выполнения СУБД подставляет значения параметров (полученные из программы) и использует сформированный ранее план выполнения для достижения результата.

При этом реализуется идея однократного выполнения тех действий, которые можно выполнить один раз. Так, подготовленная один раз инструкция может быть выполнена десятки раз с разными параметрами.

Основные **команды** динамического SQL

EXECUTE IMMEDIATE	Немедленное выполнение инструкции
PREPARE	Подготовка инструкции к выполнению
EXECUTE	Выполнение подготовленной ранее инструкции
DESCRIBE	Специальная команда, участвующая при возврате результата выполнения инструкций динамического SQL
DECLARE CURSOR	Разновидность инструкции DECLARE CURSOR, применявшейся ранее в рамках статического SQL, содержащая вместо запроса его имя (связанное с запросом при помощи инструкции PREPARE)
OPEN **FETCH** **CLOSE**	Разновидности инструкций для работы с курсором в динамическом SQL

6.5.3. Интерфейсы программирования приложений (API). DB-Library, ODBC, OCI, JDBC

Как замечено выше, программный SQL отличается от обычной, интерактивной формы наличием некоторых специальных инструкций, а также механизмом трансляции и выполнения запросов. Таким образом, для применения программного SQL в тексте своих программ программистам необходимо ознакомиться с некоторым специфическим набором инструкций. Стоит заметить, что в разных СУБД эти наборы

инструкций, вообще говоря, могут несколько отличаться друг от друга. В результате возникает некоторая проблема, связанная с непереносимостью программы.

Наряду с описанным выше механизмом существует и активно применяется еще один подход, связанный с наличием специальных интерфейсов – API (application programming interface – интерфейс программирования приложений). Эти API представляют собой библиотеки функций, разработанные для обеспечения связи прикладной программы с СУБД посредством выполнения SQL-запросов. Прикладная программа вызывает специальные функции библиотеки для передачи в СУБД SQL-запроса в текстовом виде и для получения результатов выполнения запросов, а также различной служебной информации.

Применение подобного подхода приводит к тому, что программистам более не требуется изучать специальные наборы инструкций SQL, а необходимо лишь изучить специальную библиотеку функций. С учетом того, что механизм использования API является широко используемым и стандартным подходом (чего только стоит использование мощного аппарата Windows API), для специалистов нет ничего нового в изучении еще одной библиотеки, в данном случае – для общения с СУБД.

Кроме этого, программа, содержащая вызовы некоторых функций специализированной библиотеки, ничем не отличается по схеме компиляции и выполнения от обычной программы. Так, подобная программа не требует применения специализированного препроцессора с механизмом раздельной компиляции.

Может показаться, что подход, связанный с использованием библиотек API, является наиболее прогрессивным, на самом же деле такой вывод вряд ли верен. Так, на настоящий момент очень активно используются оба подхода. В каждом из них существуют свои достоинства и недостатки и границы разумной применимости. Как обычно, выбор того, каким из подходов воспользоваться, лежит на административной группе проекта (менеджерах проекта), которая принимает решения в зависимости от особенностей конкретной задачи, имеющегося времени, специалистов и, иногда, финансовых ресурсов. В данном разделе мы рассмотрим подход, основанный на интерфейсе программирования приложений.

Посмотрим, как работают прикладные программы, использующие различные API. Как можно заметить, принцип работы от одной библиотеки к другой не претерпевает существенных изменений. Схема

работы приложения совместно с SQL API выглядит следующим образом [8]:

- программа получает доступ к базе данных путем вызова одной или нескольких API-функций, подключающих программу к СУБД и к конкретной базе данных;
- для пересылки инструкций SQL в СУБД программа формирует инструкцию в виде текстовой строки и затем передает эту строку в качестве параметра при вызове API-функции;
- программа вызывает выполнение API-функции для проверки состояния переданной в СУБД инструкции и обработки ошибок;
- если инструкция SQL представляет собой запрос на выборку, то, вызывая API-функции, программа записывает результаты запроса в свои переменные; обычно за один вызов возвращается одна строка или столбец данных;
- свое обращение к базе данных программа заканчивает вызовом API-функции, отключающей ее от СУБД.

Из имеющихся SQL API на настоящий момент выделилось несколько библиотек, «стандартных» в том смысле, что они активно применяются множеством разработчиков по всему миру. Данное пособие не является подробным руководством по всем этим библиотекам. Более того, для их профессионального освоения необходимо серьезное изучение соответствующей литературы. В рамках данного курса мы лишь приведем обзор этих библиотек, рассмотрим основные заложенные в них идеи. Подробно эти SQL API описаны, например, в книге Дж. Гроффа и П. Вайнберга «Энциклопедия SQL» [8].

6.5.3.1. Библиотека DB-Library

Библиотека DB-Library реализует интерфейс программирования приложений для совместной работы с известной СУБД Microsoft SQL Server. Данная библиотека является весьма обширной и содержит более 100 функций. Основными из них являются:

– dblogin(); dbopen() – подключение к БД;
– dbopen(); dbexit() – установка/разрыв соединения с БД;
– dbcmd() – передача инструкции (пакета инструкций) SQL в СУБД в текстовом виде;
– dbsqlexec() – требование к СУБД выполнить текущий пакет инструкций;
– dbcancel() – прекращение выполнения пакета инструкций SQL;

– dbresults() – получение результатов выполнения очередной инструкции SQL в текущем пакете;
– dbbind(), dbdata(), dbnextrow(), dbnumcols(), dbdatlen() и др. – обработка результатов запросов на выборку данных.

Логика работы прикладной программы, обрабатывающей данные, хранящиеся в базе данных под управлением Microsoft SQL Server, выглядит следующим образом:

- при помощи указанных выше функций (dblogin(), dbopen()) прикладная программа формирует сведение об авторизации и пытается установить соединение с СУБД;
- при помощи СУБД программа открывает конкретную базу данных, с которой будет происходить работа (dbopen());
- при помощи специальной функции (dbcmd()) программа передает в СУБД текст SQL-инструкции, которую далее необходимо будет выполнить; в библиотеке DB-Library поддерживается так называемый пакетный режим работы. Данный режим подразумевает возможность создания пакетов инструкций. Так, вызывая функцию dbcmd() несколько раз, вы можете передать в СУБД текст нескольких команд SQL, которые впоследствии будут выполнены как одна команда;
- используя функцию dbsqlexec(), программа вызывает выполнение инструкций, переданных ранее при помощи вызовов функций dbcmd();
- вызывая функцию dbresults(), программа может определить, удалось ли СУБД выполнить очередную инструкцию (как правило, число вызовов dbresults() соответствует числу инструкций в очередном пакете);
- в случае если мы имеем дело с запросом, возвращающим набор строк в качестве результата (запросом на выборку), программа при помощи вызовов функции dbbind() осуществляет связывание каждого поля результатов запроса с некоторой областью оперативной памяти. Далее при помощи функции dbnextrow() программа выполняет переход к следующей строке результатов запроса, что приводит к помещению в буфер новых данных;
- при помощи функции dbexit() программа разрывает соединение с базой данных.

На самом деле, несмотря на кажущуюся простоту, библиотека DB-Library представляет собой большой и сложный механизм. Так, в библиотеке предусмотрены специальные механизмы обработки ошибок,

разные способы передачи результатов выполнения запросов в прикладную программу и т.д.

6.5.3.2. Протокол ODBC

ODBC (Open Database Connectivity – открытый доступ к базам данных) – разработанный компанией Microsoft универсальный интерфейс программирования приложений для доступа к базам данных [8].

Основной целью разработки протокола ODBC считается стандартизация механизмов взаимодействия с различными СУБД. Основная проблема, связанная с разработкой приложений, взаимодействующих с базами данных на основе специальных SQL API, состояла в том, что каждая СУБД имела собственный программный интерфейс доступа, каждый из них имел свои особенности и функционировал не совсем так, как другие. В связи с этим разработка приложения существенно зависела от используемой СУБД. Компания Microsoft сделала важный шаг для решения этой проблемы. Основная идея заключалась в разработке универсального интерфейса на уровне семейства операционных систем Windows, который мог бы быть поддержан в разных СУБД.

Рассмотрим кратко структуру программного обеспечения ODBC [8]:

- **интерфейс вызовов функций ODBC**: это так называемый верхний уровень ODBC, содержащий API, который и используется непосредственно приложениями. Данный API реализован в виде библиотеки динамической компоновки Dll и входит в состав операционной системы Windows;
- **драйверы ODBC**: это так называемый нижний уровень ODBC, содержащий набор драйверов для СУБД, поддерживающих протокол ODBC. В рамках технологии для каждой СУБД может быть разработан соответствующий ODBC-драйвер, который будет являться промежуточным звеном между прикладной программой и API, транслируя вызовы функций API в вызовы внутренних специализированных функций СУБД. Таким образом решается проблема стандартизации. Для многих современных СУБД существуют специализированные драйверы ODBC, отдельно устанавливаемые в операционную систему;
- **диспетчер драйверов ODBC:** данный программный механизм представляет средний уровень ODBC, управляя процессом загрузки необходимых драйверов.

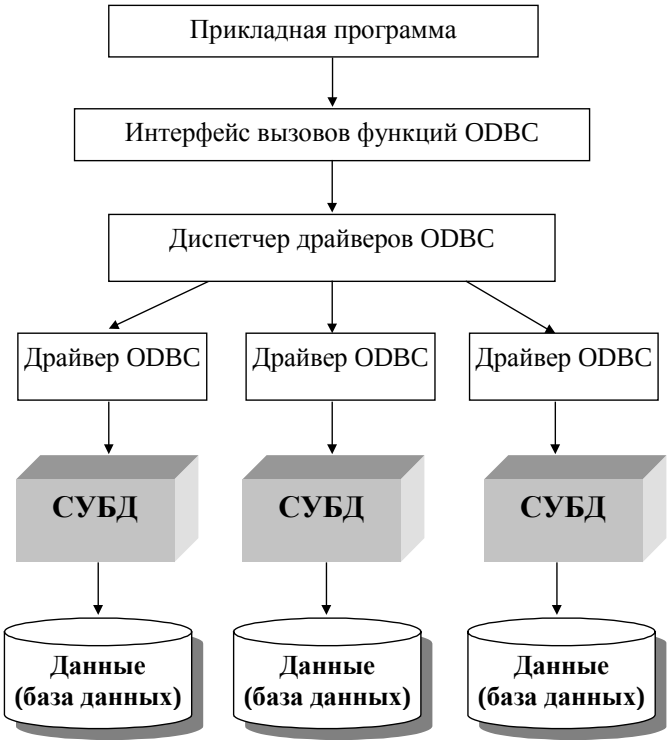

Рис. 52. Схема выполнения программы с использованием протокола ODBC для доступа к данным

6.5.3.3. Протокол OCI

Интерфейс вызовов Oracle [8] – Oracle Call Interface – OCI – специальный инструмент, представляющий собой программный интерфейс в Oracle.

Данный интерфейс появился еще в ранних версиях Oracle и продолжает свое существование по сей день, несмотря на успешное восхождение ODBC. Более того, при переходе от версии к версии OCI продолжает развиваться. Так, при переходе от версии 7 к версии 8 в OCI были внесены существенные изменения. По сути своей функции OCI очень похожи на функции динамического SQL, особенно в той части, которая касается старой версии OCI. В новую версию были вне-

сены изменения, многие из которых идеологически заимствованы из ODBC. Интерфейс OCI подробно описан в литературе и может быть изучен самостоятельно.

6.5.3.4. Протокол JDBC

JDBC (Java Database Connectivity) представляет собой API для выполнения SQL-запросов к базам данных из программ, написанных на языке Java [8].

Рассмотрим кратко историю возникновения и основные принципы JDBC. Более полное описание можно обнаружить в соответствующей литературе, например [8].

Исторически, языки C/C++ долго являлись доминирующими на рынке разработки коммерческого программного обеспечения. Много копий сломано, в том числе и в литературе, в дискуссиях на тему «Какой язык лучше?». Очевидно, что само понятие «лучше» существенно зависит от предметной области, решаемой задачи, имеющихся ресурсов и т.д. В рамках данного пособия мы не ставим своей задачей проведение сравнительного анализа различных языков программирования (интересующиеся могут обратиться, например, к работам Гради Буча [2] и других ведущих теоретиков), однако то, что языки C/C++ доминируют на рынке разработки ПО, является фактом, который на настоящий момент трудно оспорить.

По этой причине долгое время все разрабатываемые дополнительные средства программирования (такие, как, например, SQL API) ориентировались в первую очередь именно на C/C++. Однако с развитием глобальных сетей, в частности Интернета, и всех сопутствующих технологий стали появляться новые языки, специально предназначенные для работы в новых условиях. Одним из таких языков является язык программирования Java. Имея стандартный набор операторов, будучи весьма похожим на язык C++, Java быстро завоевал большую популярность на рынке разработки приложений для Интернета.

Если обратиться к части данного пособия, содержащей описание основных архитектур, можно увидеть, что Интернет-приложения занимают существенное место на рынке, работая в рамках 2-, 3- и многозвенной архитектуры. При этом значение языка Java как средства создания приложений, работающих с базами данных, существенно возрастает. Именно это и явилось одной из основных причин разработки нового программного интерфейса – JDBC. Первоначально интерфейс JDBC был разработан компанией Sun Microsystems, в настоящий мо-

мент этот API поддерживается всеми ведущими коммерческими СУБД.

Известно несколько различных версий JDBC. Так, версия 1.0 содержала некоторые средства доступа к данным:

- диспетчер драйверов (для подключения к разным СУБД);
- механизм управления сеансами (для одновременной работы с несколькими СУБД);
- механизм передачи инструкций SQL на выполнение в СУБД;
- механизм работы с курсорами (для передачи результатов выполнения запросов из СУБД в приложение).

Этот перечень определенным образом напоминает аналогичный функциональный аппарат протокола ODBC.

Версия JDBC 2.0 содержит существенные отличия. Так, вследствие увеличения возможностей интерфейса было проведено его идеологическое разделение на две основные части: Core API (основные возможности) и Extensions API (так называемые расширения).

В литературе указаны следующие основные параметры JDBC [8]:

- **Пакетные операции**. Программа на Java может осуществить обновление базы данных в пакетном режиме, т.е. одна функция JDBC может добавить в базу данных несколько записей, что положительно сказывается на производительности программ.
- **Курсоры с произвольным доступом**. В JDBC 2.0 существует средство, позволяющее перемещаться по результатам запроса произвольным образом.
- **Обновляемые курсоры.** В JDBC 2.0 курсоры, наряду с функцией возврата результата запроса, выполняют еще и роль обновления базы данных. Обновления производятся при добавлении или изменении одной из строк в результатах запроса.
- **Организация связного пула.** Несколько программ на языке Java могут пользоваться совместным доступом к базе данных, уменьшая затраты на подключения к базе данных и отключения от нее.

Данный перечень можно продолжить (распределенные транзакции, поддержка JNDI и т.д.).

Версия JDBC 3.0 появилась совсем недавно и содержит такие новации, как объектно-реляционные расширения SQL и улучшенные механизмы обработки транзакций.

Архитектура JDBC берет свое начало от ODBC и в существенной части повторяет ее (рис. 53).

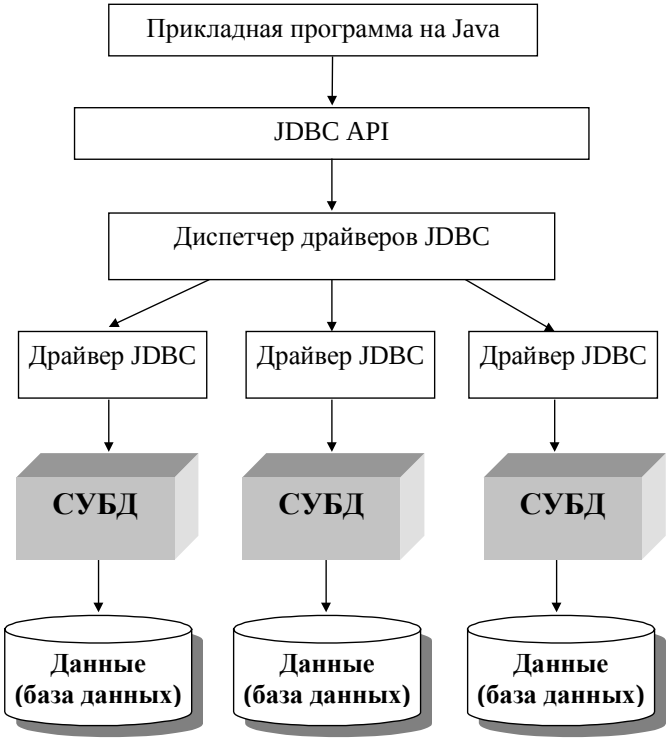

Рис. 53. Схема выполнения программы на Java с использованием протокола JDBC для доступа к данным

В отличие от ODBC, драйверы JDBC подразделяются на четыре типа. Основные отличия между этими типами связаны с местонахождением API СУБД (на клиентской или серверной СУБД) и способом доступа к базе данных (через собственный API СУБД или через ODBC).