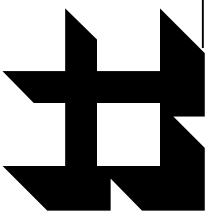


ГЛАВА 7

Работа с данными

Управление данными

Процесс управления данными включает в себя несколько действий:

- открытие необходимых баз данных и таблиц;
- установка буферизации и блокировок;
- поиск необходимых данных;
- изменение, добавление и удаление данных;
- отбор данных по заданным критериям;
- вывод на печать.

О печати данных вы узнаете из *главы 13*.

Открытие баз данных и таблиц

Язык Visual FoxPro разрабатывался как инструмент манипуляции данными, которые физически хранятся в файлах DBF. Это накладывает определенные особенности по работе с такими данными. Прежде всего, чтобы работать с какими-либо данными, необходимо получить информацию о том, где эти данные хранятся, и получить к ним доступ. В терминах Visual FoxPro обе эти операции означают то, что таблицу надо открыть.

Однако данные в VFP — это уже не просто одиночный файл DBF. Как правило, это уже комплекс взаимосвязанных файлов. Прежде всего, это структурный индексный файл CDX и файл для хранения memo-полей FPT. Но если таблица включена в базу данных, то это еще и файлы базы данных DBC, DCT, DCX. Именно поэтому и возникает необходимость в операции открытия. Нужно не просто получить доступ к одиночному файлу, но получить доступ к группе файлов, связанных между собой. Причем при дальнейших модификациях данных нужно организовать синхронное изменение в связанных файлах, если это необходимо.

Файлы DBF хранят информацию "построчно". Другими словами, то, что вы видите в окне **BROWSE**, когда просматриваете содержимое таблицы, это, по сути, и есть то, как данные хранятся физически. Не совсем, но очень близко. Поиск и модификация данных вручную в окне **BROWSE** — это практически и есть то, что будут делать различные команды Visual FoxPro, выполняя аналогичные действия. Даже если вы используете для модификации данных групповые команды, все равно изменения будут производиться по одной записи за одно действие: найти очередную запись, изменить очередную запись. Команда

```
OPEN DATABASE [FileName | ?] [EXCLUSIVE | SHARED] [NOUPDATE] [VALIDATE]
```

открывает базу данных. База данных не будет открыта, если она монопольно захвачена другим пользователем.

Параметры:

`FileName` — имя файла базы данных. Если путь к базе не указан, она ищется в каталоге по умолчанию, а затем по путям, установленным в программе или файле Config.fpw.

`EXCLUSIVE | SHARED` — открывает базу данных в режиме монопольного или разделяемого доступа.

`NOUPDATE` — запрещает изменение базы данных. Опция не влияет на изменение данных в таблицах базы данных. Для того чтобы запретить изменение данных в таблице, необходимо открыть таблицу командой `USE` с опцией `NOUPDATE`.

`VALIDATE` — включает режим проверки базы данных. Проверка заключается в контроле наличия всех таблиц и индексов на диске.

Открытие базы данных вовсе не означает, что входящие в ее состав таблицы откроются автоматически. Их тоже нужно открыть.

При открытии таблицы ее заголовок и первый блок записей переводится в *рабочую область* (Work Area), а указатель активной записи устанавливается на первую строку данных. Для каждой открытой таблицы в оперативной памяти выделяется своя рабочая область. Области нумеруются, начиная с 1; первые десять иногда обозначают буквами (A—J). Обозначение рабочих областей буквами осталось от первых версий FoxPro. Одновременно может быть открыто до 32 767 таблиц. На самом деле их гораздо больше, ведь 32 767 таблиц можно открыть в каждом сеансе данных. После запуска VFP активной является область с номером 1.

Когда в области открывается таблица, область получает *псевдоним* (Alias). Обычно alias совпадает с именем файла DBF. Однако это не всегда так. Дело в том, что в одном *сеансе данных* не может быть двух одинаковых alias. Поэтому если псевдоним таблицы не указан явно при ее открытии (опция `ALIAS` в команде `USE`, например), то Visual FoxPro самостоятельно назначит уникальный alias для таблицы, начав, разумеется, с alias, совпадающего с именем файла DBF, если это возможно.

Рабочая область может быть в активном или пассивном состоянии. Активной называется рабочая область, на которую направлены действия операторов VFP. В каждый момент времени активной может быть только одна рабочая область, все остальные — пассивны. Назначение активной области осуществляет команда `Select`.

Синтаксис команды:

```
SELECT {<номер области> | <псевдоним>}
```

Примеры см. в табл. 7.1.

Таблица 7.1. *Выбор рабочей области*

`SELECT 1`	Выбрать рабочую область с номером 1
`SELECT MyTable`	Выбрать рабочую область с alias MyTable
`SELECT 0`	Выбрать рабочую область со свободным номером

После того как область выбрана, можно открыть в ней таблицу. Таблицу можно открыть:

- в главном меню VFP: **File | Open | Table** или **Window | Data Session**;
- в менеджере проектов на вкладке **Database**;
- в командном окне, используя оператор `USE`.

Синтаксис команды:

```
USE [[DatabaseName!] TableName | SQLViewName | ?]
   [IN nWorkArea | cTableAlias] [ONLINE] [ADMIN] [AGAIN]
   [NOREQUERY [nDataSessionNumber]] [NODATA]
   [INDEX IndexFileList | ? [ORDER [nIndexNumber | IDXFileName
   | [TAG] TagName [OF CDXFileName] [ASCENDING | DESCENDING]]]]
   [ALIAS cTableAlias] [EXCLUSIVE] [SHARED] [NOUPDATE]
   [CONNSTRING cConnectionString | nStatementHandle ]
```

Параметры:

`[DatabaseName!] TableName | SQLViewName | ?` — указывает имя открываемой таблицы. Для того чтобы открыть таблицу, не входящую в текущую базу данных, укажите перед именем таблицы имя базы данных с последующим разделителем в виде восклицательного знака (!). `SQLViewName` указывает имя представления SQL, открываемого в текущей базе данных. `SQLViewName` может быть также именем автономного представления (offline view), созданного посредством функции `CREATEOFFLINE( )`.

`IN nWorkArea | cTableAlias` — указывает рабочую область, в которой открывается таблица. Указывает алиас (дополнительное имя) таблицы, которая будет открыта в рабочей области или таблицы, которая была открыта ранее. Если вы опустите опции `nWorkArea` и `cTableAlias`, таблица будет открыта в текущей рабочей области.

ONLINE — открывает автономное представление, созданное функцией CREATEOFFLINE( ). Вы можете указать имя автономного представления в параметре SQLViewName. Вы можете использовать функцию TABLEUPDATE( ) для обновления данных на сервере. Автономное представление должно быть открыто для монопольного использования. При открытии такого представления посредством команды USE, укажите в ней опцию EXCLUSIVE или выполните команду SET EXCLUSIVE ON. Вы не можете открывать или закрывать автономные представления, находясь внутри транзакций.

ADMIN — открывает автономное представление, созданное функцией CREATEOFFLINE( ), но не передает на сервер обновления, которые могут быть сделаны в этом представлении. Открытие автономного представления в режиме ADMIN позволяет вам делать любые изменения в представлении, не беспокоясь, что они будут переданы на сервер. Вы не можете открывать или закрывать автономные представления в режиме ADMIN, находясь внутри транзакции.

AGAIN — для открытия таблицы одновременно в нескольких рабочих областях вы можете выполнить следующее:

- выбрать другую рабочую область и применить команду USE с указанием имени таблицы и опцией AGAIN;
- применить команду USE с именем таблицы и опцией AGAIN, указав другую рабочую область в опции IN.

Когда вы повторно открываете таблицу в другой рабочей области, эта таблица принимает атрибуты из рабочей области, где она была открыта. Например, если таблица была открыта в первичной области только на чтение (read-only) или для монопольного использования, то после повторного открытия этой таблицы в другой рабочей области она также будет открыта только на чтение или монопольно.

NOREQUERY [nDataSessionNumber] — указывает, что данные из удаленного представления (remote SQL view) повторно не загружаются. Опция NOREQUERY допустима только для представлений SQL и обычно используется, когда вы повторно открываете представление с опцией AGAIN.

NODATA — загружает структуру представления SQL, не заполняя ее данными.

INDEX IndexFileList | ? — отображает диалог открытия файлов со списком доступных для выбора индексных файлов.

ORDER [nIndexNumber | IDXFileName — указывает, что файл типа IDX будет использован в качестве главного индексного файла.

ORDER [TAG TagName] [OF CDXFileName] — указывает главный индексный тег в составном индексном файле типа CDX.

ASCENDING | DESCENDING — указывает, что записи в таблице отображаются и обрабатываются в порядке возрастания/убывания индекса. Использование опции ASCENDING или DESCENDING не изменяет индексный файл или тег, а лишь определяет порядок, в котором будет осуществляться отображение и доступ к записям таблицы.

ALIAS cTableAlias — создает алиас для таблицы, который может быть использован вместо имени таблицы в командах и функциях.

EXCLUSIVE — открывает таблицу для монопольного использования в сети.

SHARED — открывает таблицу для совместного использования в сети.

NOUPDATE — запрещает внесение изменений в таблицу и в ее структуру.

CONNSTRING cConnectionString | nStatementHandle — указывает строку для соединения посредством источника данных ODBC. cConnectionString может содержать строку параметров, обычно передаваемых функцией SQLCONNECT( ) или SQLSTRINGCONNECT( ).

В простейшем случае таблицу можно открыть таким образом:

```
USE [<имя таблицы>][IN <номер области>|<псевдоним>]
```

При отсутствии опции In команда работает с текущей рабочей областью.

Примеры см. в табл. 7.2.

Таблица 7.2. *Способы выбора рабочей области*

SELECT 0 USE MyTable	Выбираем свободную рабочую область. В ней открываем таблицу MyTable
SELECT 5 USE MyTable	Выбираем рабочую область № 5. В ней открываем таблицу MyTable
USE MyTable IN 5 SELECT MyTable	Открываем таблицу в 5 области. Делаем эту рабочую область активной
USE Two_Table	В текущей рабочей области открываем таблицу Two_Table. Если в этой области была открыта другая таблица, она будет закрыта, а вместо нее открыта Two_Table
USE MyTable IN 0	Открывает таблицу в свободной области

Ссылка на нулевую рабочую область означает ссылку на первую свободную рабочую область. То есть рабочую область, которая в настоящий момент не связана ни с какой таблицей. Это позволяет открывать новые таблицы, не заботясь о том, не занята ли текущая рабочая область другой таблицей. То есть дается команда

```
USE MyTable IN 0
```

Указание нулевой рабочей области необходимо потому, что в противном случае таблица будет открыта в текущей рабочей области, одновременно закрыв таблицу, которая могла бы быть в ней открыта.

Иногда бывает необходимо открыть таблицу повторно. Цель такого действия — организация поиска в одной и той же таблице по разным индексам. Для реализации многократного открытия таблицы служит предложение AGAIN оператора USE.

```
USE table1 in 0
USE table1 in 0 AGAIN
```

Команда USE без имени таблицы освобождает текущую или указанную рабочую область, закрывая используемую в ней таблицу (табл. 7.3).

Таблица 7.3. *Закрытие таблицы в рабочей области*

USE	Закрывает таблицу в текущей рабочей области
USE IN MyTable	Закрывает таблицу с псевдонимом MyTable

Одновременно одной рабочей области может соответствовать только одна таблица, в то время как одной таблице может соответствовать несколько рабочих областей. Другими словами, одну и ту же таблицу можно одновременно открыть в нескольких рабочих областях, но в одной конкретной рабочей области может быть открыта только одна таблица.

Кроме понятия *"рабочая область"* (Work Area), в Visual FoxPro введено понятие *"сеанс данных"* (DataSession).

Сеанс данных (DataSession) — это некоторая динамическая копия среды Visual FoxPro. Открывая среду Visual FoxPro, вы автоматически открываете *сеанс данных*, который автоматически же и завершается при выходе из Visual FoxPro. Но внутри собственно VFP вы имеете возможность сделать копию среды Visual FoxPro, используя так называемые *"частные сеансы данных"* (Private DataSession).

Что собственно дают эти *"частные сеансы данных"*? А дают они возможность симулировать внутри одного приложения VFP работу нескольких, независимых друг от друга пользователей.

В результате таблицы, открытые в одном сеансе данных, "не видны" в другом сеансе данных. И соответственно манипуляции (не все), производимые с таблицами в одном сеансе данных, не влияют на другие сеансы данных.

Самый распространенный случай применения **Private DataSession** — это одновременное открытие нескольких форм, рассматривающих один и тот же документ с разных сторон. Как правило, для этой цели между таблицами устанавливаются отношения с помощью команды SET RELATION. Если открыть подобные формы в одном сеансе данных, то зачастую это становится большой проблемой, поскольку в одной форме требуется наложить на таблицы одни индексы и связи, а в другой — другие.

Таблица может быть открыта как монопольно, или эксклюзивно (от англ. *Exclusive* — единственный, исключительный), так и в режиме разделяемого доступа (от англ. *Share* — разделять, совместно использовать). Режим доступа к таблице определяется предложением EXCLUSIVE | SHARED в операторе USE. Если же в команде USE это предложение опущено, то способ открытия таблицы зависит от установки SET EXCLUSIVE. Если один из пользователей уже открыл таблицу монопольно, то другие пользователи при попытке открыть эту таблицу получат сообщение об ошибке.

При выполнении команды USE имя таблицы может быть указано путем доступа к ней, например

```
USE Data\MyTable
```

или

```
USE C:\Myproject\Data\MyTable
```

Если путь к таблице не указан, таблица ищется сначала среди таблиц базы данных, а уже потом на диске в соответствии с установленными путями.

Общая концепция многопользовательского приложения

В настоящее время практически все приложения пишутся в расчете на многопользовательский доступ. Это значит, что с одними и теми же данными могут работать несколько пользователей в одно и то же время. Как следствие, возникает ряд специфических проблем при написании подобных приложений. Прежде всего, следует понять, что термин "одновременный доступ" — это иллюзия. На самом деле компьютер не изменяет данные одновременно. Вовсе нет. Он производит модификации данных строго по очереди. Даже если команды придут от двух разных пользователей одновременно, они все равно будут обработаны по очереди. Сначала команда одного пользователя, потом — другого. Весь вопрос в том, как эта очередь формируется. С точки зрения Visual FoxPro при модификации данных единая и неделимая единица информации — это одна запись таблицы. Именно запись, а не одно поле этой записи. Поэтому, чтобы избежать ситуации, когда один пользователь начал изменять запись, а второй пользователь тут же начал изменять те же данные, в момент начала модификации запись блокируется для изменений другим пользователем.

Эти блокировки FoxPro накладывает и снимает сам, автоматически, по мере необходимости. В большинстве случаев программисту нет надобности вмешиваться в этот процесс. Хотя для этого предусмотрены специальные команды и функции `RLOCK()`, `FLOCK()`, `UNLOCK` и т. п., но использовать их надо с крайней осторожностью и, по возможности, следует избегать.

Дело в том, что компьютер обрабатывает данные быстро, очень быстро. Значительно быстрее, чем человек их вводит. Это значит, что надо предоставить человеку как можно больше времени для процесса модификации данных, но если запись будет заблокирована, то пользователь ничего не сможет в ней изменить! Бесполезная трата рабочего времени пользователя.

Для преодоления этого противоречия было введено специальное понятие — *буферизация*. Пользователь производит изменения не напрямую в исходных данных, а в некоторой промежуточной структуре, которая называется "буфер". Хотя самому пользователю кажется, что он изменяет именно исходные данные. После того как пользователь закончит внесение изменений в эту промежуточную структуру, будет осуществлена операция сброса буфера в исходную таблицу.

Таким образом, пользователь неопределенно долгое время может изменять данные, но эти данные не будут заблокированы, поскольку физически изменения происходят не с ними, а в некотором буфере. А вот в момент сброса буфера будет осуществлена блокировка данных. Но, поскольку компьютер обрабатывает данные очень быстро, то

блокировка, последующая модификация и снятие блокировки произойдут очень быстро и незаметно для пользователя.

Правда, при такой организации работы возможен так называемый конфликт совместного доступа. Это когда два пользователя одновременно начали изменять одни и те же данные, а затем одновременно же попытались их записать. Чье изменение принять? Может быть, надо все-таки блокировать данные при начале изменения? Нет. Не надо. Данный вопрос разбивается на два: программный и организационный.

Программный вопрос заключается в том, чтобы подобная ситуация не привела к нарушению целостности базы данных. Ведь эти с точки зрения пользователя изменения были осуществлены одновременно. А с точки зрения программы сначала было произведено одно изменение, а потом — другое. Не одновременно, а последовательно. Все вполне логично и правильно.

Другое дело, что второй пользователь уверен в том, что он изменяет исходные данные, а, по сути, он изменяет изменения, внесенные первым пользователем. Вот это и есть организационный вопрос. А допустимо ли это? Надо ли выдать второму пользователю предупреждение вроде: пока вы вносили изменения, эти же данные были изменены другим пользователем. Писать поверх изменений первого пользователя или наоборот отказаться от своих собственных изменений?

Решение этого вопроса зависит от конкретной задачи. В FoxPro есть штатные инструменты определения подобной ситуации. Однако практика показывает, что если пользователю надо внести какие-либо изменения, он их внесет. Вне зависимости от грозных предупреждений и принудительных блокировок.

Но до сих пор рассматривался вопрос изменения одной записи в одной таблице. На практике зачастую необходимо изменить несколько взаимосвязанных записей в нескольких таблицах. Причем "одним пакетом". То есть либо изменились все, либо не изменился никто. Частичная модификация недопустима.

Буферизация эту проблему решить не может, поскольку сброс буфера идет по одной записи за раз. Это значит, что вполне возможна ситуация, когда буфер первой записи был успешно сброшен, а на второй произошла ошибка. Операция сброса буфера будет прервана, но первая-то запись уже обновилась.

Вот для решения задачи сброса "одним пакетом" было введено понятие — *транзакция*. Транзакция "фиксирует" состояние базы данных на момент начала транзакции. В случае успешного завершения всех изменений завершает транзакцию, а в случае ошибки возвращает базу данных в то состояние, в котором она была на момент начала транзакции. Причем в понятие "состояние базы данных" входит и состояние всех буферов таблиц. Это значит, что в "исходное состояние" (на момент начала транзакции) будут возвращены и буферы таблиц.

Буферизация

Буферизация используется в процессе модификации данных. Сам процесс модификации условно можно разбить на два этапа: внесение изменений в запись и сохранение

измененной записи в таблице. Для того чтобы реализовать разделение процесса внесения изменений и сохранения внесенных изменений, необходимо иметь некую промежуточную структуру, в которой и будет происходить модификация. А затем надо организовать сброс содержимого этой структуры в исходную таблицу.

В Visual FoxPro для создания такой промежуточной структуры введено понятие "буферизация". *Буферизацией* называют способ внесения изменений в таблицу, при котором изменения сначала производятся в памяти компьютера, а затем сбрасываются на диск. Возможна буферизация как таблицы, так и отдельной записи.

При строковой буферизации формируется буфер только одной изменяемой записи, а при табличной формируется буфер всех изменяемых записей таблицы. Блокируется не таблица, а все изменяемые записи.

Буферизация может быть жесткой (пессимистической) и мягкой (оптимистической). Пессимистическую буферизацию можно представить в виде документа, с которым работает пользователь. Если один пользователь уже работает с этим документом, то второму пользователю документ недоступен. Оптимистическая буферизация задействуется только в момент изменения записи.

Для использования буферизации необходимо сначала установить `SET MULTILOCKS ON`. Затем с помощью функции `CURSORSETPROP()` нужно установить один из методов буферизации:

```
lSuccess=CURSORSETPROP("Buffering", 5, "Customer")
```

Типы буферизации:

1. Отключает буферизацию строки или таблицы. Запись блокируется при начале редактирования, поля записываются при перемещении указателя записи, данные записываются как в более ранних версиях FoxPro.
2. Устанавливается пессимистическая буферизация строки. Записи блокируются при начале редактирования, а поля записываются при перемещении указателя записи. Запись блокируется одним пользователем, остальным запись доступна только в режиме чтения. Используя функцию `TABLEREVERT()`, можно аннулировать изменения, внесенные в текущую запись.
3. Устанавливается оптимистическая буферизация строки. Записи не блокируются при редактировании; Visual FoxPro пытается блокировать записи, когда они записываются на диск с помощью функции `TABLEUPDATE()`. Поскольку блокировка выполняется только в момент переноса изменений в таблицу, уменьшается время ограниченного доступа к записи.
4. Устанавливается пессимистическая буферизация таблицы. Записи блокируются на все время редактирования. Блокировка будет снята при выполнении функции `TABLEUPDATE()`.
5. Устанавливается оптимистическая буферизация таблицы. Записи блокируются только на время выполнения функции `TABLEUPDATE()`.

Блокировки

Для чего нужно блокировать записи? Предположим, один ваш пользователь в определенной записи меняет дату рождения лица, указанного в этой записи, а другой пользователь в это же время у этой же записи меняет адрес. Чья запись будет сохранена в таблице? Правильно, того пользователя, который сохранил ее позже.

Цель блокировки именно предотвратить вмешательство в модификацию другого пользователя.

Некоторые команды Visual FoxPro автоматически блокируют записи и таблицы во время своего выполнения. Список таких команд приводится в табл. 7.4.

Таблица 7.4. *Команды и функции, выполняющие автоматическую блокировку*

Команда или функция	Что блокируется
ALTER TABLE — SQL	Вся таблица
APPEND	Заголовок таблицы
APPEND BLANK	Заголовок таблицы
APPEND FROM	Заголовок таблицы
APPEND FROM ARRAY	Заголовок таблицы
APPEND MEMO	Текущая запись
BLANK	Текущая запись
BROWSE	Текущая запись
CHANGE	Текущая запись
CURSORSETPROP()	В зависимости от параметров
DELETE	Текущая запись
DELETE NEXT	Текущая запись
DELETE RECORD	Запись n
DELETE — SQL	Текущая запись
DELETE > 1 ЗАПИСИ	Вся таблица
EDIT	Текущая запись
GATHER	Текущая запись
INSERT	Вся таблица
INSERT — SQL	Заголовок таблицы
MODIFY MEMO	Текущая запись
RECALL	Текущая запись
RECALL NEXT	Текущая запись
RECALL RECORD	Запись n

RECALL > 1 ЗАПИСИ	Вся таблица
REPLACE	Текущая запись
REPLACE NEXT	Текущая запись
REPLACE RECORD	Запись n
REPLACE > 1 ЗАПИСИ	Вся таблица
TABLEUPDATE()	В зависимости от способа буферизации
UPDATE	Вся таблица
UPDATE – SQL	Блокирует только изменяемые записи

Существует ряд команд и функций Visual FoxPro, управляющих блокировкой записей и таблиц (или влияющих на блокировку). Хотя, конечно, по возможности использования команд прямой блокировки следует избегать. Их использование уместно в хранимых процедурах (триггерах, значениях по умолчанию), но не желательно в коде программы. К установкам, управляющим блокировкой, можно отнести SET REPROCESS TO:

```
SET REPROCESS TO nAttempts [SECONDS] [SYSTEM] | TO AUTOMATIC [SYSTEM]
```

Параметры:

nAttempts — определяет количество попыток блокировки данных, если первая попытка была неудачной. nAttempts может принимать значения от 0 до 32000. Если не установить в этом параметре разумное значение, можно бесконечно ждать окончания следующего цикла:

```
IF RLOCK()
 <...>
ENDIF
```

Этот цикл будет "висеть" для второго пользователя очень долго. Если nAttempts установлено в 0 и производится попытка блокировки, то число таких попыток бесконечно. При этом на экран выводится сообщение "Attempting to look... Press Escape to Cancel". Это сообщение исчезает только после снятия блокировки. Если при этом SET STATUS установлено в ON, на экране появляется сообщение "Waiting for lock". Если все попытки блокировки не имели успеха, то будет выполнена процедура ON ERROR (если она определена). Если же процедура обработки ошибки отсутствует, то появится системное сообщение "Record is in use by another".

SECONDS указывает, что nAttempts — это число в секундах. Число должно быть больше 0.

TO AUTOMATIC — задание этой опции аналогично nAttempts=0.

SYSTEM — позволяет применить SET REPROCESS к системной сессии данных, которая управляет файлами FoxUser, SCX и др.

Установка SET REPROCESS может быть выполнена в файле конфигурации Config.fpw, например:

```
REPROCESS=2
```

Кроме SET REPROCESS, используются еще некоторые установки:

- SET REFRESH TO — определяет интенсивность обновления данных;
- SET MULTILOCS — определяет возможность блокировки нескольких записей;
- SET LOCK — определяет возможность блокировки таблиц для команд:
  - AVERAGE
  - CALCULATE
  - COPY TO
  - COPY TO ARRAY
  - COUNT
  - DISPLAY (with a scope)
  - INDEX
  - JOIN (both files)
  - LABEL
  - LIST
  - REPORT
  - SORT
  - SUM
  - TOTAL

Функция

```
FLOCK ([nWorkArea | cTableAlias])
```

пытается заблокировать текущую или указанную таблицу. Возвращает "истину", если таблица заблокирована. Возвращает "ложь", если таблица заблокирована другим пользователем или просто-напросто не открыта в текущей рабочей области.

Разблокировать таблицу можно командой UNLOCK, или закрыть таблицу, или дать команду QUIT. При этом разблокировать таблицу может пользователь, выполнивший блокировку.

Функции

```
LOCK ([nWorkArea | cTableAlias]| [cRecordNumberList, nWorkArea | cTableAlias])
```

и

```
RLOCK ([nWorkArea | cTableAlias] | [cRecordNumberList, nWorkArea | cTableAlias])
```

пытаются заблокировать одну или более записей текущей или указанной таблицы.

Для блокировки нескольких записей вы можете использовать установки команды SET MULTILOCKS ON и указать соответственно номер рабочей области (nWorkArea) или имя алиаса (cTableAlias) для таблицы, записи которой должны быть заблокированы.

Вы можете также выполнять блокировку нескольких записей, перемещая указатель записи и вызывая последовательно функции RLOCK( ) или LOCK( ) для каждой записи.

В Visual FoxPro вы можете в качестве номера записи указывать 0. В этом случае блокируется заголовок таблицы.

Команда

```
UNLOCK [RECORD nRecordNumber] [IN nWorkArea | cTableAlias] [ALL]
```

снимает блокировку с записи или группы записей.

Параметры:

`RECORD nRecordNumber` — освобождает запись с номером `nRecordNumber`. Если номер записи равен нулю, то освобождается заголовок таблицы.

`IN nWorkArea | cTableAlias` — указание на снятие блокировки с таблицы в указанной рабочей области или с указанным псевдонимом.

`ALL` — освобождает все записи и все таблицы во всех рабочих областях.

Снятие блокировки: `UNLOCK [RECORD nRecordNumber] [IN nWorkArea | cTableAlias] [ALL]` (снятие должно производиться из того программного модуля, в котором производилась блокировка).

Функция

`SYS(2011)` —

определяет статус блокировки записи таблицы.

Функция

```
ISFLOCKED([nWorkArea | cTableAlias])
```

возвращает статус блокировки файла таблицы.

Функция возвращает `.T.` только на компьютере, с которого выполнена блокировка.

Функция

```
ISRLOCKED([nRecordNumber, [nWorkArea | cTableAlias]])
```

возвращает состояние блокировки записей. Функция возвращает `.T.` только на компьютере, с которого выполнена блокировка.

Функции работы с блокированными записями:

- `CURVAL()` — возвращает текущее значение поля таблицы;
- `GETFLDSTATE()` — определяет статус изменения поля(ей) записи таблицы;
- `GETNEXTMODIFIED()` — определяет номер следующей измененной записи в буферизованном курсоре `OLDVAL()` — определяет прежнее значение поля (до изменения), если изменение выполнено, но данные не обновлялись;
- `SETFLDSTATE()` — изменяет статус изменения полей записи;
- `TABLEREVERT()` — отменяет изменения, выполненные в буфере, но еще не сохраненные `TABLEUPDATE()` — пытается выполнить изменения, выполненные в буфере курсора.

Поиск данных

Организация поиска в таблице зависит от установки индекса. Если индекс отсутствует или не установлен, то придется организовать поиск в неупорядоченной таблице.

Такая ситуация вполне возможна в случае, если невозможно построить индексы для таблицы (например, проект разработан сторонним разработчиком, и вы по условиям технического задания не можете в нем ничего менять). Для поиска в неупорядоченной таблице существует ряд команд.

Команда

```
LOCATE [FOR lExpression1] [Scope] [WHILE lExpression2] [NOOPTIMIZE]
```

просматривает таблицу, пока не обнаружит первую запись, для которой выражение `lExpression1` оценивается в "истину". Если запись, удовлетворяющая условию, существует, то указатель будет установлен на эту запись, а функция `FOUND()` вернет `.T.` Совместно с `LOCATE` используется команда `CONTINUE`, которая позволяет продолжать поиск для нахождения следующих записей, удовлетворяющих условию.

Команда

```
LOOKUP (ReturnField, eSearchExpression, SearchedField [, cTagName])
```

ищет в таблице первую запись с полем, соответствующим указанному выражению.

И не только ищет, но и возвращает значение другого поля `ReturnField` найденной записи.

Поиск в индексированной таблице происходит гораздо быстрее, если индекс установлен по тому полю, по которому ведется поиск. Кроме того, индексы позволяют связывать таблицы, т. е. организовывать между ними отношения.

Команда

```
SEEK eExpression   [ORDER nIndexNumber | IDXIndexFileName
    | [TAG] TagName [OF CDXFileName]    [ASCENDING | DESCENDING]]
   [IN nWorkArea | cTableAlias]
```

ищет в таблице первое появление записи, индексный ключ которой соответствует общему выражению, затем перемещает указатель на соответствующую запись.

Если `SEEK` находит вхождение индексного ключа в записи, `RECNO( )` возвращает номер записи вхождения, `FOUND( )` возвращает истину (`.T.`), а `EOF( )` возвращает ложь (`.F.`).

Функция

```
SEEK(eExpression [, nWorkArea | cTableAlias   [, nIndexNumber | IDXIndexFileName
| cTagName]])
```

производит поиск записи в индексированной таблице, для которой индексный ключ соответствует выражению `eExpression`, при этом перемещая указатель на найденную запись. Использование функции `SEEK()` эквивалентно использованию команды `SEEK` и функции `FOUND()`. Функцию удобно применять в отчетах, этикетках.

Функция

```
INDEXSEEK eExpression [, lMovePointer [, nWorkArea | cTableAlias
   [, nIndexNumber | cIDXIndexFileName | cTagName]]])
```

ищет в индексированной таблице первую встретившуюся запись, чей индексный ключ соответствует указанному выражению, без перемещения указателя записи.

Навигационные команды

В Visual FoxPro доступ к данным осуществляется с помощью указателей. Использование указателей подразумевает, что для обработки конкретных данных в таблицах сначала по указателю находят необходимую запись, затем выполняют над ней нужные команды и, наконец, переходят к следующей записи.

Команды, использующие принцип перемещения по указателям, называются *навигационными*.

Перемещение по таблице

Команды

```
GO [RECORD] nRecordNumber [IN nWorkArea | IN cTableAlias]
GO TOP | BOTTOM [IN nWorkArea | IN cTableAlias]
GOTO [RECORD] nRecordNumber [IN nWorkArea | IN cTableAlias]
GOTO TOP | BOTTOM [IN nWorkArea | IN cTableAlias]
```

перемещают файловый указатель таблицы (либо текущей, либо указанной в `nWorkArea | IN cTableAlias`) на запись с заданным номером (`nRecordNumber`) или в начало (`GO TOP`) или в конец (`GO BOTTOM`) таблицы.

`RECORD nRecordNumber` — указывает на физический номер записи. Если номер превысит количество записей в таблице, то будет сгенерирована ошибка "Record is out of range".

`TOP` устанавливает указатель таблицы на первую запись. Если используется возрастающий индекс, то первой записью будет являться запись с наименьшим значением ключа, если убывающий — с наибольшим.

`GO BOTTOM` — устанавливает указатель на последнюю запись. Если используется возрастающий индекс, то последней является запись с наибольшим значением ключа.

Команда

```
SKIP [nRecords] [IN nWorkArea | cTableAlias]
```

перемещает указатель по таблице вперед или назад.

`nRecords` — количество записей, на которое перемещается указатель. Если число опущено, то указатель перемещается на следующую запись. Если число > 0, то перемещение следует к концу файла, если < 0, то к началу.

Добавление записей в таблицу

Добавить записи в таблицу можно различными способами, которые мы сейчас рассмотрим. Сразу после создания таблицы FoxPro предлагает ввести данные вручную.

Если вы отложили ввод данных на более позднее время, вы всегда сможете открыть таблицу в командном окне и дать команду `Append`

```
APPEND [BLANK] [IN nWorkArea | cTableAlias] [NOMENU]
```

Опции:

`BLANK` — добавляет одну пустую запись в конец таблицы;

`IN nWorkArea | cTableAlias` — для команды `Append` без опции `BLANK` работа происходит с текущей таблицей. Для `APPEND BLANK` запись добавляется в таблицу, которая указывается в опции;

`NOMENU` — удаляет меню **Table** из главного меню VFP. Не позволяет работать с записями таблицы из главного меню.

Если вы открыли таблицу и просматриваете ее, дав команду `BROWSE`, в главном меню FoxPro появляется пункт **Table**, в нем можно выбрать подпункт **Append New Record**, при этом в таблицу будет добавлена пустая запись. Если ваша таблица содержит поле типа **Integer** (**AutoInc**), то при добавлении записи она сразу же нумеруется.

Хотя для ввода пустой записи есть способ попроще: установите указатель записи на последнюю запись и нажмите комбинацию клавиш <Ctrl>+<Y>. Пустая запись будет добавлена, и вы сможете ее заполнить.

Физически запись всегда добавляется в конец файла. Нужный порядок следования обеспечивается главным индексом.

Теперь рассмотрим программные способы добавления записей в таблицу.

Самый простой способ состоит в том, что в таблицу добавляется пустая запись командой `APPEND BLANK`, а потом на экран выводится форма для ее заполнения.

```
USE SOTR
APPEND BLANK
DO FORM app_sotr
```

При этом редактируются поля уже введенной в таблицу записи.

Более популярный среди программистов способ — это редактирование переменных памяти. Получить список переменных памяти из таблицы можно командой `SCATTER MEMVAR`. Если же нам нужно получить список пустых переменных памяти, то используем команду `SCATTER MEMVAR BLANK`. После редактирования переменных той же формой, их можно сохранить командой `GATHER MEMVAR`.

Если в вашей таблице имеется поле типа `Integer` (`AutoInc`), то добавить в таблицу пустую запись можно следующим образом:

```
SELECT sotr
APPEND BLANK
replace fam WITH "Иванов";
  im WITH "Иван";
  ot WITH "Иванович"
BROWSE
```

Поле ID при этом будет заполнено автоматически.

Можно добавлять записи, используя команду INSERT-SQL, что предпочтительнее:

```
SELECT sotr
INSERT INTO sotr (fam, im, ot) values ("Петров","Петр","Петрович")
```

Об SQL-командах вы узнаете чуть позже.

Команда

```
APPEND FROM ARRAY ArrayName [FOR lExpression]
   [FIELDS FieldList | FIELDS LIKE Skeleton | FIELDS EXCEPT Skeleton]
```

добавляет записи в таблицу из массива ArrayName. Количество добавляемых записей равно количеству строк массива. Если количество элементов строки массива больше, чем количество полей в таблице, лишние элементы отбрасываются без дополнительных сообщений. Если же количество элементов массива меньше, чем количество полей в таблице, лишние поля заполняются значениями по умолчанию.

Поля типа Memo, Blob и General командой APPEND FROM ARRAY игнорируются.

Запись в эти поля можно производить командой GATHER или APPEND GENERAL.

Добавить запись в таблицу, имеющую поле Integer (AutoInc), возможно, если установить предварительно SET AUTOINCERROR=OFF или установить

```
CURSORSETPROP("AutoIncError",.F.,"Sotr")
```

С появлением представлений (View) программисты все чаще используют их для добавления записей в таблицу. В этом случае создается представление (View), созданное View, добавляется в DataEnvironment создаваемой формы. Туда же добавляются и таблицы-источники, из которых создано View. Там же, в DataEnvironment формы, для добавленного Local View устанавливается свойство NoDataOnLoad = .T. Эта настройка означает, что Local View откроется, но без содержимого, только структура. В событии Init формы пишется такой код:

```
LOCAL MyVar
MyVar = 1
=ReQuery("MyView")
```

Этот код наполняет содержимым Local View.

В самой форме располагаются нужные объекты и в их свойстве ControlSource указывается ссылка на поля Local View.

Для сохранения внесенных изменений нужно дать команду TableUpdate().

Добавление данных в поля типа *Blob*, *Memo* и *General*

Эти три типа полей имеют свои особенности ввода. Пустое поле типа Memo обозначено как "memo", а заполненное — как "Memo", т. е. с прописной буквы. Аналогичное явление происходит с полями Blob и General. Для того чтобы открыть такое поле, нужно либо дважды щелкнуть по нему мышью, либо нажать комбинацию клавиш

<Ctrl>+<Home>. После этого поле готово для ввода любой текстовой информации. Сохранение информации происходит при нажатии комбинации клавиш <Ctrl>+<W?. Информацию в поле можно копировать.

Поле типа `General` может содержать OLE-объекты, например, рисунки, фотографии, таблицы Excel и т. д. В отличие от поля типа `Memo` при работе с полем типа `General` открывается приложение, с помощью которого создан OLE-объект. И если в этом приложении OLE-объект был изменен, мы немедленно увидим это и в поле типа `General`. Таким образом, мы получаем возможность отслеживать изменения, например, в документах Microsoft Word.

Изменение (редактирование) записей

Изменять уже введенные записи таблицы можно следующими командами (табл. 7.5).

Таблица 7.5. *Команды модификации данных*

Команда	Описание

`REPLACE FieldName1 WITH eExpression1 [ADDITIVE]`
`[, FieldName2 WITH eExpression2 [ADDITIVE]] ...`
`[Scope]`
`[FOR lExpression1] [WHILE lExpression2] [IN nWorkArea | cTableAlias]`
`[NOOPTIMIZE]`
	Обновляет записи в таблице новыми значениями

Таблица 7.5 *(окончание)*

Команда	Описание

`REPLACE FROM ARRAY`
`ArrayName [FIELDS FieldList] [Scope]`
`[FOR lExpression1] [WHILE lExpression2]`
`[NOOPTIMIZE]`
	Обновляет содержимое полей значениями из массива переменных памяти.
	Команда `REPLACE FROM ARRAY` игнорирует memo-поля и поля типа `General`. Чтобы импортировать данные в эти поля, пользуйтесь командами `GATHER` и `APPEND GENERAL`

`GATHER FROM`
`ArrayName` `| MEMVAR | NAME` *`ObjectName`*
`[FIELDS` *`FieldList`* `| FIELDS LIKE` *`Skeleton`* `| FIELDS EXCEPT` *`Skeleton`*`]`
`[MEMO]`
	Заменяет данные в текущей записи текущей выбранной таблицы данными, взятыми из массива, набора переменных или объекта

`BLANK [FIELDS FieldList] [DEFAULT [AUTOINC]]`
`[Scope] [FOR lExpression1]`
`[WHILE lExpression2] [NOOPTIMIZE]`
`[IN nWorkArea | cTableAlias]`
	Очищает данные в определенных полях текущей записи или из всех полей текущей записи, если команда задана без дополнительных аргументов

Пример

Нужно помнить, что команда REPLACE изменяет данные в текущей рабочей области, что сначала надо переместить указатель записи на ту таблицу и ту запись, содержимое которой предполагается изменить.

```
IF SEEK(otdel.id,"sotr")
    SELECT sotr
    REPLACE num_otdel  WITH otdel.id
endif
```

Удаление и восстановление записей

Для Visual FoxPro удаление — это не физическое удаление, а лишь установка метки на удаление. То есть удаленная запись все равно будет присутствовать в таблице, но с пометкой удаления. Спрятать все подобные записи можно при помощи настройки SET DELETED ON.

Физическое удаление записей происходит через команду PACK. Выполнение этой команды выносится в отдельную служебную процедуру. Ни в коем случае нельзя ее использовать в процессе работы приложения (табл. 7.6).

Таблица 7.6. *Команды удаления и восстановления записей*

Команда	Описание
DELETE [Scope] [FOR lExpression1] [WHILE lExpression2] [IN nWorkArea | cTableAlias] [NOOPTIMIZE]	Помечает записи для удаления

Таблица 7.6 *(окончание)*

Команда	Описание
RECALL [Scope] [FOR lExpression1] [WHILE lExpression2] [NOOPTIMIZE] [IN nWorkArea | cTableAlias]	Снимает отметку удаления записи в выбранной таблице; восстанавливает запись
PACK [MEMO | DBF] [Tablename] [IN nWorkarea | cTableAlias]	Навсегда удаляет все записи, помеченные для удаления в текущей таблице, и уменьшает размер memo-файла таблицы (fpt), который связан с таблицей. Таблица для этой операции должна быть открыта монопольно
ZAP [IN nWorkArea | cTableAlias]	Удаляет все записи из таблицы, оставляя только структуру таблицы. Записи, удаленные из таблицы командой ZAP, не подлежат восстановлению командой RECALL

Пример

```
IF SEEK(fio,"sotr")
    SELECT sotr
```

```
    DELETE
ENDIF
```

Циклы

Циклические команды тоже являются навигационными, поскольку перемещают указатель записи.

Команда

```
SCAN [NOOPTIMIZE] [Scope] [FOR lExpression1] [WHILE lExpression2]
   [Commands]
   [LOOP]
   [EXIT]
ENDSCAN
```

Выполняет последовательность операторов [Commands], перемещая указатель по текущей таблице.

LOOP — передает управление команде SCAN.

EXIT — организует выход из цикла SCAN... ENDSCAN.

VFP при достижении команды ENDSCAN автоматически выбирает таблицу, которая была текущей перед выполнением команды SCAN, поэтому команда SELECT <Table> не нужна.

Команда

```
DO WHILE lExpression
      [Commands]
   [LOOP]
   [EXIT]
ENDDO
```

выполняет группу команд, входящих в условный цикл. Сам по себе цикл DO WHILE никакие указатели не перемещает, для этого используется команда SKIP.

Пример 1 — организация цикла SCAN...ENDSCAN

```
SELECT vxod
GO top
SCAN
   IF SEEK(LEFT(vxod.avt,4),"ispspr")
       m.simv=SUBSTR(vxod.avt,5)
       replace vxod.avt WITH ispspr.cod+m.simv
   ELSE
       MESSAGEBOX("Не найден код автора записи")
   ENDIF
ENDSCAN
```

Пример 2 — те же действия, выполненные с помощью цикла DO WHILE ... ENDDO

```
SELECT vxod
```

```
GO top
DO WHILE ! EOF('vxod')
   IF SEEK(LEFT(vxod.avt,4),"ispspr")
       m.simv=SUBSTR(vxod.avt,5)
       replace vxod.avt WITH ispspr.cod+m.simv
   ELSE
       MESSAGEBOX("Не найден код автора записи")
   ENDIF
SKIP
ENDDO
```

Фильтры

Для того чтобы работать не со всей большой таблицей, а только с ее частью, можно установить фильтр, используя при этом некоторое условие.

Фильтр — это некое ограничение, накладываемое при перемещении указателя записи. Причем вычисление выражения фильтра происходит каждый раз при попытке переместить указатель. Логика примерно следующая: при подаче команды `SKIP` FoxPro перемещает указатель записи на следующую запись. Если эта запись не удовлетворяет условию фильтра, то FoxPro автоматически перемещает указатель на следующую запись. Опять проверяет условие фильтра. И так до тех пор, пока не будет найдена запись, удовлетворяющая условию фильтра или же не будет достигнут конец таблицы.

Другими словами, фильтр, это не что-то вычисленное один раз при подаче команды `SET FILTER`. Это условие, которые проверяется каждый раз при попытке перемещения на очередную запись. Именно поэтому, если условию фильтра не соответствует большое количество записей, то перемещение по такой таблице будет происходить чрезвычайно медленно. Просто слишком возрастет время поиска очередной записи, удовлетворяющей условию фильтра.

Имеет смысл использовать фильтр для отсечения небольшого количества записей. Если же надо отфильтровать (отбросить) много записей, то лучше использовать Local View или Select-SQL.

Для работы с фильтром служит команда

```
SET FILTER TO <условие>
```

Например, в таблице `SOTR` установим фильтр по номеру отдела.

```
USE sotr
SET FILTER TO otdel="08"
```

Но надо учитывать тот факт, что сам фильтр устанавливается быстро, а вот работать с ним на больших таблицах — медленно.

Реляционные команды

В Visual FoxPro существуют два метода обработки данных — навигационный и реляционный. Навигационные команды мы рассматривали выше, сейчас пришел черед реляционных.

Реляционные команды были разработаны компанией IBM в конце 80-х годов прошлого столетия. Первые SQL-команды были включены еще в FoxPro 2.*. С тех пор реляционные команды присутствуют в языке Visual FoxPro, а также регулярно модифицируются и добавляются. Основная реляционная команда — **Select SQL**. Скорость выполнения запросов SQL весьма высока. Но популярность реляционных команд обусловлена не только скоростью обработки данных, но и их универсальностью — они применимы как к файл-серверным информационным системам, так и к клиент-серверным. Рассмотрим основные реляционные команды подробнее. Главная из них — `SELECT – SQL`.

Команда `SELECT - SQL`

имеет следующий шаблон:

```
SELECT Select_List
   FROM Table_List
   [WITH (BUFFERING = lExpr)]
   [WHERE Conditions]
   [GROUP BY  Column_List]
   [UNION Clause]
   [ORDER BY Column_List]
   [INTO Clause | TO Clause]
   [PREFERENCE PreferenceName] [NOCONSOLE] [PLAIN] [NOWAIT]
```

и предназначена для выборки и отображения данных одной или более таблиц базы данных. Если `SET TALK = ON`, то по завершении команды отобразится время, затраченное на ее выполнение, и количество отобранных записей. Результат выполнения запроса — новая таблица, причем исходные таблицы остаются без изменений. Установка `SET FILTER` командой `SELECT – SQL` игнорируется.

Оператор `SELECT` выполняется в следующем порядке:

1. Определяются имена используемых таблиц и представлений — предложение `FROM`.
2. Производится фильтрация строк объектов в соответствии с заданными условиями — предложение `WHERE`.
3. Образуются группы строк, имеющих одно и то же значение в заданном столбце (вложенность в порядке записи имен столбцов) — предложение `GROUP BY`.
4. Устанавливается, какие столбцы должны присутствовать в выходных данных — предложение `SELECT`.

5. Определяется упорядоченность результатов выполнения оператора — предложение ORDER BY.

Поскольку команда SELECT – SQL большая и довольно сложная, давайте рассмотрим каждое ее предложение по отдельности.

```
SELECT [Select_List]
```

Предложение SELECT определяет список столбцов результата выполнения запроса. В ней возможны следующие варианты:

SELECT ALL — в результате запроса будут отображены все записи.

SELECT DISTINCT — исключит из результата повторяющиеся записи. Опция не поддерживается в запросах, содержащих поля Blob, Memo и General. Выйти из ситуации можно, если использовать memo-поля в качестве параметров функций PADR() или ALLTRIM().

SELECT TOP nExpr — в результат включаются первые nExpr записей. В этом случае 1<nExpr>32767.

SELECT TOP nExpr PERCENT — в результат включаются первые nExpr процентов записей. В этом случае 0.01<nExpr>99.99.

При задании опции TOP необходимо задать и опцию ORDER BY.

SELECT Select_List — в этом случае в результат включаются перечисленные в списке Select_List поля, константы и выражения. Число элементов списка ограничено и может содержать не более 255 значений. Каждый элемент списка представляет собой колонку в результате, и каждой такой колонке можно присвоить свое собственное имя. Для этого необходимо использовать фразу AS Name.

Список Select_List может включать в себя:

- * — будут выбраны все поля таблиц-источников данных

  Пример

  ```
  SELECT * FROM Vxod
  ```

- Alias.Field [, Alias. Field1,...] — список полей таблиц-источников

  Пример

  ```
  SELECT Vxod.nvxod, Vxod.dreg FROM Vxod
  ```

- Константу — она будет выводиться в каждой строке результата

  Пример

  ```
  SELECT PI() AS PI, Table1.ID FROM Table1
  ```

- Выражение — может быть как пользовательской функций, так и агрегатной функцией VFP типа MAX(), COUNT(), SUM() и др. Список агрегатных функций приведен в табл. 7.7. Если вы не используете фразу AS Name в результате, то колонка с результатом вычислений по умолчанию будет названа Exp_1, Exp_2 и т. д.

  Пример 1 (агрегатная функция)

```
SELECT DISTINCT Sum(Table1.Sum) AS Summa FROM Table1
```

Пример 2 (пользовательская функция)

```
SELECT UDF() as FioDat FROM Table1
FUNCTION  UDF
     RETURN PADR(DTOC(Datr)+ALLTRIM(Fio),50)
ENDFUNC
```

Подзапрос — вложенная команда `SELECT – SQL`. Подзапрос заключается в круглые скобки. Вложенный подзапрос выглядит как

```
SELECT ... WHERE (SELECT... WHERE (SELECT... WHERE))
```

Пример

```
SELECT * FROM table1 WHERE Id NOT IN (SELECT Id FROM table2)
```

Таблица 7.7. *Список агрегатных функций*

Функция	Описание
`AVG(Table1.Field1)`	Среднее значение по столбцу `Table1.Field1` (только для числовых полей)
`COUNT(Table1.Field1) | CNT(Table1.Field1)`	Количество элементов столбца `Table1.Field1`
`MIN(Table1.Field1)`	Минимальное значение по столбцу
`MAX(Table1.Field1)`	Максимальное значение по столбцу
`SUM(Table1.Field1)`	Сумма значений по столбцу (только для числовых полей)

Фраза `FROM Table_List` задает список таблиц, из которых будет производиться выборка. Число таблиц не ограничено. `Table_List` может включать в себя:

- таблицу БД, свободную таблицу или существующее представление;
- псевдоним таблицы, устанавливаемый на время действия команды `Select` для квалификации столбцов с совпадающими именами.

`FROM FORCE` — указывает, что таблицы соединяются в порядке их появления в списке. Если фраза опущена, VFP пытается оптимизировать запрос.

`Table_List` содержит список таблиц-источников данных. Если вам надо явно указать имя базы данных, к которой относится таблица-источник, то используется такой синтаксис:

```
Database!TableName AS Local_Alias
```

Здесь:

`Database` — имя нетекущей базы данных;

`TableName` — имя таблицы, входящей в базу данных;

Local_Alias — локальный псевдоним таблицы. Если он задан, то его нужно использовать во всем запросе.

Пример

```
SELECT Table1.ID, T2.Summa From Table1, Table2 AS T2
```

(SubQuery) AS SubQuery_Alias — задает подзапрос. В опции FROM каждый подзапрос должен иметь псевдоним. Все подзапросы во FROM выполняются до оценки основного оператора SELECT.

Также в запросе может быть задана опция JOIN, задающая порядок объединения таблиц. В табл. 7.8 приведены типы опции JOIN.

Таблица 7.8. *Типы опции JOIN*

Тип опции JOIN	Описание
Inner Join	Inner Join определяет только те записи, которые соответствуют условию объединения. Этот тип является встроенным и наиболее употребительным
Left [Outer] Join	Left Outer Join определяет, что все записи в таблице, расположенной слева от JOIN, независимо от того, соответствуют ли они условию объединения, и только те записи из таблицы, расположенной справа от JOIN, которые соответствуют условию, включаются в результат
Right [Outer] Join	Right Outer Join определяет, что все записи в таблице, указанной справа от JOIN, независимо от того, соответствуют ли они условию, и только те записи в таблице, указанной слева от JOIN, которые соответствуют условию, включаются в результат
Full Join	Full Join определяет, что все записи, которые соответствуют и не соответствуют условию, включаются в результаты. Поле должно соответствовать тексту примера, символа в символ
Cross Join	Cross Join определяет, что каждая запись левой таблицы соответствует всем записям правой таблицы. Если WHERE-статья не существует, установка перекрестного соединения производит декартов результат. Декартов результат содержит набор всех возможных комбинаций каждого элемента одной таблицы в наборе со всеми элементами в другой таблицы

По умолчанию JOIN=Inner JOIN.

ON JoinCondition | FilterCondition — условие объединения двух таблиц и (или) фильтрации данных.

Условие объединения JoinCondition — это логическое выражение, которое может включать в себя (табл. 7.9):

Таблица 7.9. *Условия объединения Join*

Условие объединения	Описание
Сравнение двух полей различных таблиц	Field1 операция_сравнения Field2

Сравнение выражений с полями различных таблиц	`Table1.Field1+Table2.Field1` операция_сравнения `Table3.Field1`
Пользовательская функция (должна вернуть логическое значение)	`UDF(Tabl1.Field1, Table2.Field1)`

В качестве операции сравнения допустимы опции, представленные в табл. 7.10.

Таблица 7.10. *Опции, допустимые в "операции_сравнения"*

Опция и параметр	Описание
`Field1 LIKE cExpression`	Значение поля соответствует символьному выражению. Допустимы маски "%" и "_"
`Field1 IS NULL`	Значение поля соответствует `NULL`
`Field1 BETWEEN Beg_Range AND End_Range`	Значение поля находится между `Beg_Range` и `End_Range`
`Field1 IN (Value_Set)`	Значение поля является элементом списка значений `Value_Set`

Рассмотрим несколько примеров. Для них необходимы таблицы. Пусть это будут две таблицы — продуктов и поставщиков: **Prod** и **Post**.

Prod
ID_Pr C(4)
Naim_pr C(30)
Opis m(4)
ID_Post C(4)

Post
ID_Post C(4)
Name_pos C(30)
Adres C(100)
Tel C(10)

- Запрос с `INNER JOIN` — только те записи, которые соответствуют условию, включаются в результат. Этот тип является встроенным.

Выберем список продуктов по поставщикам:

```
SELECT TAlias1.name_pos, Talias2.naim_pr ;
FROM post AS TAlias1 ;
INNER JOIN prod AS TAlias2 ;
 ON TAlias1.ID_post = TAlias2.id_post ;
 ORDER BY TAlias1.name_pos ASC
```

Результат выполнения `SELECT` с `INNER JOIN` вы видите на рис. 7.1.

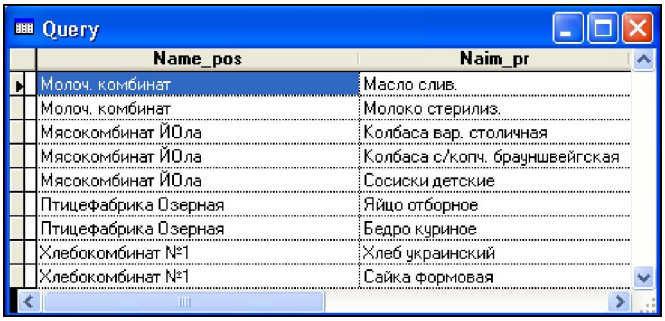


Рис. 7.1. Запрос с `INNER JOIN`

- Запрос с `LEFT JOIN` — все записи из левой таблицы `POST`, независимо от соответствия условию, и только те записи из правой таблицы `PROD`, которые соответствуют условию, включаются в результат.

Результаты запроса представлены на рис. 7.2.

Выберем все продукты одного поставщика:

```
SELECT post.name_pos, prod.* ;
FROM post;
LEFT JOIN prod ON Prod.ID_post=="0001";
where prod.id_post==post.id_post
```

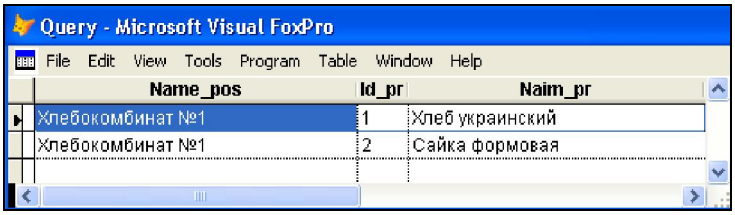
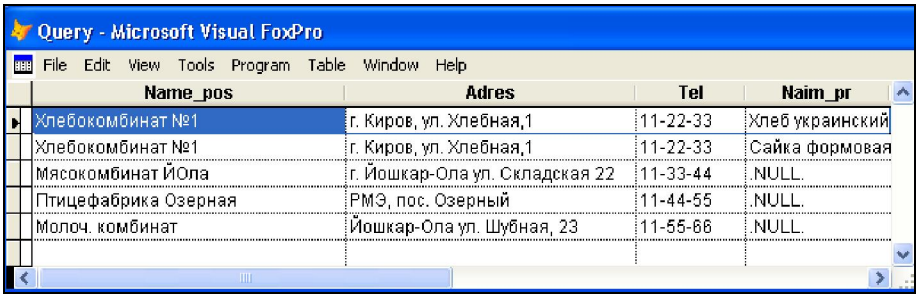


Рис. 7.2. Результат запроса с `LEFT JOIN`

- Запрос с `RIGHT JOIN`. Результаты запроса представлены на рис. 7.3.

Выберем всех поставщиков одного продукта

```
SELECT post.name_pos, post.adres, post.tel, prod.naim_pr ;
FROM prod;
RIGHT JOIN post ON Prod.ID_post=="0001" ;
AND prod.id_post==post.id_post
```



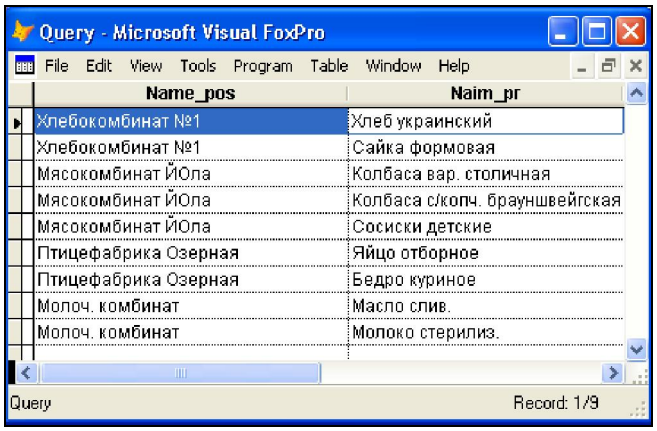
Name_pos	Adres	Tel	Naim_pr
Хлебокомбинат №1	г. Киров, ул. Хлебная,1	11-22-33	Хлеб украинский
Хлебокомбинат №1	г. Киров, ул. Хлебная,1	11-22-33	Сайка формовая
Мясокомбинат ЙОла	г. Йошкар-Ола ул. Складская 22	11-33-44	.NULL.
Птицефабрика Озерная	РМЭ, пос. Озерный	11-44-55	.NULL.
Молоч. комбинат	Йошкар-Ола ул. Шубная, 23	11-55-66	.NULL.

Рис. 7.3. Результат работы запроса с `RIGHT JOIN`

- Запрос с `FULL JOIN` включает в результат все записи, которые соответствуют и не соответствуют условию.

  Результат запроса представлен на рис. 7.4.

```
SELECT post.name_pos,prod.naim_pr ;
FROM prod;
FULL JOIN post ON prod.id_post==post.id_post
```



Name_pos	Naim_pr
Хлебокомбинат №1	Хлеб украинский
Хлебокомбинат №1	Сайка формовая
Мясокомбинат ЙОла	Колбаса вар. столичная
Мясокомбинат ЙОла	Колбаса с/копч. брауншвейгская
Мясокомбинат ЙОла	Сосиски детские
Птицефабрика Озерная	Яйцо отборное
Птицефабрика Озерная	Бедро куриное
Молоч. комбинат	Масло слив.
Молоч. комбинат	Молоко стерилиз.

Рис. 7.4. Результат работы запроса с `FULL JOIN`

Предложение `WITH (Buffering= lExpr)` — указывает, если `lExpr=.T.`, что запрос основывается на буферизованных данных (если они есть), или, если `lExpr=.F.`, что данные выбираются с диска. При этом в опции `FROM` надо писать псевдоним таблицы, буфер которой необходимо использовать. То есть ни в коем случае не указывать имя базы данных, путь доступа или расширение.

Предложение `WHERE Conditions` — задает условия объединения и/или отбора записей из таблиц-источников.

Предложение `GROUP BY Column_List` — список, задающий группировку строк результата по одному или нескольким столбцам.

Элементами списка могут быть:

- имена полей таблиц или подзапроса;
- поле результата запроса;
- числовое выражение, представляющее собой номер столбца с результатом запроса.

Список `Column_List` должен содержать все без исключения поля списка предложений `SELECT` и `HAVING`. Ознакомьтесь также с описанием команды `SET ENGINEBEHAVIOR`.

Следует упомянуть о группировке по условию. Пример такой группировки представлен на рис. 7.5.

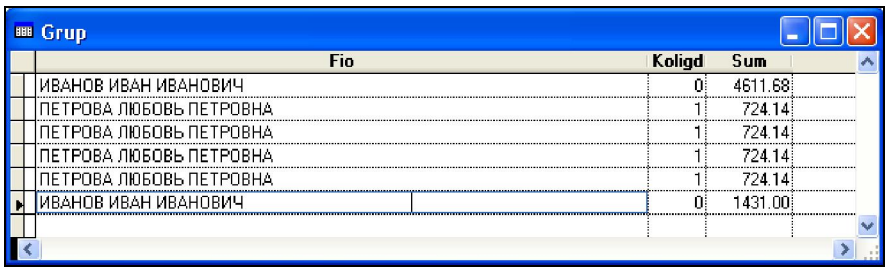


Рис. 7.5. Пример задания на группировку данных по условию

Постановка задачи такова: если поле `KOLIGD` имеет значение 0, то ВСЕ такие записи надо группировать (складывать), если же поле `KOLIGD` имеет значение, отличное от 0, то такие записи группировать не надо. Стандартный `Select-SQL` не может что-то группировать, а что-то не группировать. Он группирует все. Это противоречие можно разрешить, если для тех записей, которые группировать не надо, сформировать уникальное значение того поля, по которому происходит группировка. То есть такое значение, которое будет уникально в пределах *всей* таблицы. Не должно быть ни одной записи с повторяющимся значением в этом поле. В результате эти записи также будут сгруппированы, но, поскольку признак группировки уникален, то в эту группу попадет только одна запись. Именно это производится в следующем операторе `SELECT-SQL` (листинг 7.1).

```
SET ENGINEBEHAVIOR 70
SELECT ;
IIF(Koligd = 0, SPACE(10), SYS(2015)) as groupId, ;
Fio, Koligd, Sum(SUM);
FROM grup ;
GROUP BY ;
groupId, Fio
```

Для записей, имеющих значение `KOLIGD=0`, формируется повторяющееся, неуникальное значение поля **GroupID**, а для записей, имеющих значение `KOLIGD<>0`, формируется уникальное, не повторяющееся значение поля **GroupID**.

В результате суммируются только те записи, у которых значение поля **GroupID** повторяющееся. Чтобы не суммировать вообще все записи со значением `KOLIGD=0` в качестве признака группировки добавляется дополнительное поле, например, **FIO**.

Результат выполнения вы видите на рис. 7.6.

Groupid	Fio	Koligd	Sum_sum
	ИВАНОВ ИВАН ИВАНОВИЧ	0	6042.68
_1X10SOKW	ПЕТРОВА ЛЮБОВЬ ПЕТРОВНА	1	724.14
_1X10SOKW	ПЕТРОВА ЛЮБОВЬ ПЕТРОВНА	1	724.14
_1X10SOKW	ПЕТРОВА ЛЮБОВЬ ПЕТРОВНА	1	724.14
_1X10SOKW	ПЕТРОВА ЛЮБОВЬ ПЕТРОВНА	1	724.14

Рис. 7.6. Результат выполнения оператора `Select` с группировкой по условию

Предложение `HAVING` — задает условия, которым должна удовлетворять группа записей для включения в результат. Предложение `HAVING` особенно интересно при совместном применении `HAVING` и `GROUP BY` при вычислении агрегатных функций. В этом случае можно легко вычислить количество, сумму, среднее значение по полю таблицы.

Пример

```
SELECT TovarID, SUM(cena*kol) as summa FROM Table1 GROUP BY TovarID HAVING
SUM(cena*kol)>3000.
```

Предложение `UNION` используется при выборке из таблиц с совпадающими структурами для объединения отобранных строк. При этом происходит объединение двух запросов с исключением повторений. В объединяемых запросах число столбцов должно быть одинаковым. Директива `UNION ALL` объединит выборки без исключения повторов.

Предложение `ORDER BY` сортирует отобранные строки по перечисленным в списке столбцам с учетом их вложенности.

Предложение `INTO` указывает объект, принимающий результат запроса. В предложении могут быть заданы следующие опции:

```
ARRAY
CURSOR
TABLE | DBF
FILE
PRINTER
SCREEN
```

По умолчанию данные посылаются в курсор (INTO CURSOR) с одновременным отображением содержимого этого курсора в окне **BROWSE**.

Любая фраза, кроме SELECT и FROM, может быть опущена, но порядок оставленных в операторе фраз фиксирован.

В заключение статьи о команде SELECT – SQL хотелось бы предложить читателям список наиболее популярных команд SELECT:

- все записи из двух таблиц с одинаковой структурой

```
SELECT * FROM table1 UNION ALL  SELECT * FROM table2
```

- все записи из таблицы 1, которые есть в таблице 2

```
SELECT * FROM table1 WHERE ID IN (SELECT ID FROM table2)
```

- все записи из таблицы 1, которых нет в таблице 2

```
SELECT * FROM table1 WHERE ID NOT IN (SELECT ID FROM table2)
```

- все записи, которых нет или в таблице 1, или в таблице 2

```
SELECT * FROM table1 WHERE ID NOT IN (SELECT ID FROM table2);
UNION ALL  SELECT * FROM table2 WHERE ID NOT IN (SELECT ID FROM table1)
```

- выбрать все записи без дублирования

```
SELECT DISTINCT * FROM table1 INTO TABLE table2
```

Команда INSERT – SQL

Добавляет в конец таблицы новую запись, которая содержит указанные значения полей. Команда INSERT – SQL имеет три синтаксиса:

1. Используйте первый синтаксис, чтобы добавить указанные значения в указанные поля таблицы.

```
INSERT INTO TableName [(FieldName1 [, FieldName2, ...])]
   VALUES (eExpression1 [, eExpression2, ...])
```

2. Используйте второй синтаксис, чтобы вставить содержание элементов массива, переменной памяти, или свойства объекта, которые соответствуют именам полей в таблице.

```
INSERT INTO dbf_name FROM ARRAY ArrayName | FROM MEMVAR |
FROM NAME ObjectName
```

3. Используйте третий синтаксис, чтобы вставить строки из команды SQL – SELECT в указанные поля таблицы.

```
INSERT  INTO dbf_name [(FieldName1 [, FieldName2, ...])]
   SELECT SELECTClauses [UNION UnionClause SELECT SELECTClauses ...]
```

Примеры

1. ```
   INSERT INTO SOTR [ID, Fio]
         VALUES (m.Last+1, m.fio)
   ```
2. ```
   INSERT INTO Not_Found FROM ARRAY noFound
   ```
3. ```
   INSERT  INTO  All_Nalog    SELECT * from Nalog_2005 UNION ALL
          SELECT * FROM Nalog_2006
   ```

Команда UPDATE_SQL

служит для модификации (редактирования) данных в базе. Формат оператора:

```
UPDATE Target
SET Column_Name1 = eExpression1 [, Column_Name2 = eExpression2 ...]
 [FROM [FORCE] Table_List_Item [[, ...] | [JOIN [ Table_List_Item]]]
 WHERE FilterCondition1 [AND | OR FilterCondition2 ...]
```

Здесь:

`Target` — определяет имя базовой таблицы, курсора, псевдонима таблицы или курсора, или имя файла, где выполняются требуемые изменения;

фраза `SET` указывает, какие *столбцы* редактируются и какие *значения* в них будут занесены вместо прежних значений,

фраза `WHERE` указывает, какие строки будут заменены.

По умолчанию (при отсутствии `WHERE`) замена производится во всех строках.

В VFP при обновлении таблицы из открытой, но пассивной БД имя таблицы записывается с уточнением в виде DatabaseName!TableName.

В 9-ой версии Visual FoxPro поддерживаются подзапросы в секции SET, при этом в секции SET допустим только один подзапрос. Если в данной секции SET используется подзапрос, то использование WHERE недопустимо. Общая синтаксическая конструкция при этом выглядит следующим образом:

UPDATE … SET … (SELECT …) …

Пример 1

```
UPDATE MyTable ;
SET Cena =Cena * 1.2
```

Пример 2

```
UPDATE MyCursor1 SET field2=100+(SELECT field2 FROM MyCursor2 ;
 WHERE MyCursor2.field1=MyCursor1.field1) WHERE field1>5
```

Команда DELETE – SQL

Оператор `DELETE` предназначен для удаления данных из базы. Формат оператора:

```
DELETE [Target] FROM [FORCE] Table_List [[, Table_List ...] |
[JOIN [ Table_List]]]
```
```

```
[WHERE FilterCondition1 [AND | OR FilterCondition2 ...]]
```

По умолчанию — при отсутствии фразы `WHERE`, удаляются все строки; таблица становится пустой; в БД остается только описание ее структуры.

Table_List может иметь следующие синтаксисы:

[*DatabaseName* !] *Table* [[AS] *Local_Alias*]

DatabaseName ! определяет имя базы данных, содержащей таблицу, если таблица находится вне текущей базы данных. Если таблица находится вне текущей базы данных, необходимо включать имя базы данных. Используйте восклицательный знак (!) как разделитель после имени базы данных и перед именем таблицы.

Table определяет имя таблицы или курсора, из которых вы хотите извлечь данные. Если нет открытых таблиц, Visual FoxPro отображает диалоговое окно Open, так что вы можете определить местонахождение файла. После того, как таблица открывается, она остается открытой и после завершения запроса.

Local_Alias определяет временное имя для таблицы, указанной в *Table* . Если вы определяете локальный псевдоним, используйте этот локальный псевдоним вместо имени таблицы в любом месте инструкции **DELETE** . Псевдоним может представлять таблицу или курсор.

JOIN обеспечивает возможность определить одну или более вторичных таблиц. Нет никакого жестко заданного предела количества таблиц, псевдонимов, или предложений **JOIN** в инструкции **DELETE** .

(*Subquery*) AS *Subquery_Alias*

Subquery определяет инструкцию **SELECT** в пределах другой инструкции **SELECT,** например:

```
SELECT * FROM table1 WHERE table1.iid IN ;
 (SELECT table2.itable1id FROM table2 WHERE table2.iID IN ;
 (SELECT table3.itable2id FROM table3 WHERE table3.cValue = "value"))
```

Примеры

- очистить таблицу "`MyTable`":

```
DELETE FROM MyTable
```

- удалить из таблицы "`MyTable`" записи, в которых количество оставшихся продуктов менее трех килограммов:

```
DELETE FROM MyTable;
WHERE  Ost < 3
```

VFP полностью поддерживает оператор DELETE языка SQL. При его выполнении строки таблицы физически не удаляются, но помечаются на удаление. Физическое удаление помеченных строк — сжатие файла, выполняет оператор PACK. Область действия оператора PACK — активная таблица.

Использование транзакций

Команда SQL, которая воздействует на содержание или структуру базы данных, не обязательно будет необратимой. Вы можете определить после окончания ее действия, останутся ли изменения сделанные данной командой или группой команд постоянными в базы данных, или они будут возвращены в исходное состояние. С этой целью команды обрабатываются группами, называемыми *транзакциями*.

По завершении транзакции (как только будет встречена команда END TRANSACTION) данные либо фиксируются в таблицах, либо восстанавливаются исходные данные командой ROLLBACK.

Функции и команды, обеспечивающие поддержку транзакций:

- BEGIN TRANSACTION — открывает блок транзакций;
- END TRANSACTION — закрывает блок транзакций сохраняя все изменения текущего блока;
- ROLLBACK — отменяет все изменения в текущем блоке транзакций;
- TXNLEVEL() — определяет уровень вложенности транзакций (0 — нет транзакций [0:5]).

Нужно иметь в виду, что транзакции VFP поддерживает только локальные данные, помещенные в VFP базу данных. Для обеспечения транзакций внешних данных следует использовать средства сервера данных.

Вовлеченные в транзакции данные не доступны даже на чтение, пока транзакция не завершится. Поэтому следует стремиться свести к минимуму время выполнения транзакции.

Функции и команды, не поддерживающие транзакции:

- APPEND PROCEDURES
- CLEAR ALL
- COPY INDEXES
- COPY PROCEDURES
- CREATE SQL VIEW
- CREATE TRIGGER
- DELETE DATABASE
- DELETE TRIGGER
- MODIFY DATABASE
- MODIFY PROCEDURE

Функции и команды, которые не могут быть выполнены над таблицей, вовлеченной в транзакции:

- ALTER TABLE
- CREATE TABLE

- CURSORSETPROP()
- DELETE TAG
- INDEX
- INSERT
- MODIFY STRUCTURE
- PACK
- REINDEX
- TABLEREVERT()
- ZAP

Команда

BEGIN TRANSACTION

начинает транзакцию. Команда поддерживается для таблиц, входящих в базу данных.

Возможно применение транзакций для свободных таблиц и курсоров при успешном выполнении функции MAKETRANSTABLE(). Число вложений транзакции — 5.

Поскольку записи, участвующие в транзакции, недоступны остальным пользователям, следует минимизировать время ее выполнения. Все индексные файлы должны быть закрыты, допустимо лишь открытое состояние CDX-файлов.

Команда

ENDTRANSACTION

завершает текущую транзакцию и сохраняет все изменения в таблицах. По завершении транзакции верхнего уровня данные записываются на диск.

Команда

ROLLBACK

производит откат транзакции, т. е. отменяет все изменения, внесенные в таблицы во время транзакции. Все блокировки записей, внесенные во время транзакции, снимаются после выполнения ROOLBACK.

Пример

```
BEGIN TRANSACTION
  TRY
    INSERT INTO Rschet (Nschet , User_ID, published) ;
      VALUES ;
      (m.Nschet, m.User_ID, DATETIME())
    INSERT INTO Summa (Sum, MESSAGE, User_ID, published ) ;
      VALUES ;
      (m.Sum, m.lcMess, m.User_ID, DATETIME())
END TRANSACTION
  CATCH TO oException
    ROLLBACK
  FINALLY
  ENDTRY
```

Данный пример использует команду TRY...CATCH...FINALLY для обработки ошибок. Об обработке ошибок в Visual FoxPro рассказывается в *главе 12*.

Если ваши данные хранятся в серверных базах данных (имеются в виду данные на SQL-сервере, DB2-сервере и прочих подобных), то все обновления данных на сервере реализуются в виде транзакций. SQL-сервер поддерживает три вида транзакции:

1. Явные транзакции (Explicit Transaction), в которых используются команды:
   - `BEGIN TRANSACTION` — начать транзакцию;
   - `COMMIT TRANSACTION` — зафиксировать данные в случае удачного окончания транзакции;
   - `ROLLBACK TRANSACTION` — для отката в случае неудачного окончания транзакции.
2. Неявные транзакции (Implicit Transaction), в которых транзакция устанавливается с помощью команды `SET IMPLICIT_TRANSACTIONS ON`. В этом случае при выдаче некоторых команд сервера (`SELECT`, `UPDATE`, `OPEN` и др.) выдается команда `BEGIN TRANSACTION`, но по окончании команды автоматического завершения (отката) транзакции не следует. Вам необходимо самостоятельно дать команду `COMMIT TRANSACTION` или `ROLLBACK TRANSACTION`.
3. Автоматически фиксируемые транзакции (AutoCommit Transaction) используются при выполнении команд `DELETE`, `UPDATE`, `INSERT`.

   Применяются автоматически. Это значит, что при использовании этих команд нет необходимости использовать `BEGIN TRANSACTION` и `COMMIT TRANSACTION`.

События базы данных

События контейнера базы данных (DBC) происходят, когда выполняются действия в базе данных. Вы можете добавить в эти события процедурный код, который будет исполнен при наступлении событий.

В приведенной ниже таблице описываются доступные события контейнера базы данных — DBC (табл. 7.11).

Таблица 7.11. *События базы данных*

Событие	Описание
`DBC_Activate`	Происходит, когда открытая база данных становится текущей базой данных
`DBC_Deactivate`	Происходит, когда база данных перестает быть текущей базой данных
`DBC_BeforeAddRelation`	Происходит до ввода отношения в базу данных
`DBC_AfterAddRelation`	Происходит после успешного ввода отношения в базу данных
`DBC_BeforeAddTable`	Происходит до ввода таблицы в открытую базу данных
`DBC_AfterAddTable`	Происходит после ввода таблицы в открытую базу данных
`DBC_BeforeAppendProc`	Происходит до завершения операции `APPEND PROCEDURES` в текущие хранимые процедуры контейнера базы данных DBC
`DBC_AfterAppendProc`	Происходит после завершения операции `APPEND PROCEDURES`

DBC_BeforeCloseTable	Происходит до закрытия таблицы или представления
DBC_AfterCloseTable	Происходит после закрытия таблицы или представления
DBC_BeforeCopyProc	Происходит до начала операции копирования процедуры в текущем DBC
DBC_AfterCopyProc	Происходит после завершения операции копирования процедуры в текущем DBC

Таблица 7.11 *(продолжение)*

Событие	Описание
DBC_BeforCreateConnection	Происходит до создания соединения
DBC_AfterCreateConnection	Происходит после создания соединения
DBC_BeforeCreateOffline	Происходит до перевода представления в автономный режим — offline
DBC_AfterCreateOffile	Происходит после завершения функции CREATEOFFLINE( )
DBC_BeforeCreateTable	Происходит до создания таблицы в открытой базе данных
DBC_AfterCreateTable	Происходит после создания таблицы в открытой базе данных
DBC_BeforeCreateView	Происходит до создания представления
DBC_AfterCreateView	Происходит после создания представления
DBC_BeforeDBGetProp	Происходит до запуска DBGETPROP( )
DBC_AfterDBGetProp	Происходит после завершения DBGETPROP( )
DBC_BeforeDBSetProp	Происходит до запуска DBSETPROP( ) или когда изменено свойство "Event"
DBC_AfterDBSetProp	Происходит после завершения DBSETPROP( ) или когда изменено свойство "Event"
DBC_BeforeDeleteConnection	Происходит до удаления соединения
DBC_AfterDeleteConnection	Происходит после удаления соединения
DBC_BeforeDropOffline	Происходит до возврата представления в online
DBC_AfterDropOffline	Происходит после завершения DROPOFFLINE( )
DBC_BeforeDropRelation	Происходит до удаления отношения из базы данных
DBC_AfterDropRelation	Происходит после успешного удаления отношения из базы данных
DBC_BeforeDropTable	Происходит до удаления таблицы из DBC и удаления с диска
DBC_AfterDropTable	Происходит после удаления таблицы из DBC и удаления с диска
DBC_BeforeDropView	Происходит до удаления представления из DBC

DBC_AfterDropView	Происходит после удаления представления из DBC
DBC_BeforeModifyConnection	Происходит до изменения соединения
DBC_AfterModifyConnection	Происходит после изменения соединения
DBC_BeforeModifyProc	Происходит до изменения DBC-хранимой процедуры в открытом DBC
DBC_BeforeModifyProc	Происходит до изменения DBC-хранимой процедуры в открытом DBC

Таблица 7.11 *(окончание)*

Событие	**Описание**
DBC_AfterModifyProc	Происходит после изменения и закрытия файла хранимых процедур для DBC
DBC_BeforeModifyTable	Происходит до изменения таблицы
DBC_AfterModifyTable	Происходит после изменения таблицы
DBC_BeforeModifyView	Происходит до изменения представления
DBC_AfterModifyView	Происходит после изменения представления
DBC_BeforeOpenTable	Происходит до открытия таблицы или представления
DBC_AfterOpenTable	Происходит после открытия таблицы или представления
DBC_BeforeRemoveTable	Происходит до удаления таблицы из DBC
DBC_AfterRemoveTable	Происходит после удаления таблицы из DBC
DBC_BeforeRenameConnection	Происходит до переименования соединения
DBC_AfterRenameConnection	Происходит после переименования соединения
DBC_BeforeRenameTable	Происходит до переименования таблицы
DBC_AfterRenameTable	Происходит после переименования таблицы
DBC_BeforeRenameView	Происходит до переименования представления
DBC_AfterRenameView	Происходит после переименования представления
DBC_BeforeValidateData	Происходит до исполнения VALIDATE DATABASE. Возврат .F. предотвращает DBC от проверки на действительность
DBC_AfterValidateData	Происходит после завершения VALIDATE DATABASE
DBC_ModifyData	Происходит незамедлительно после Modify Data
DBC_OpenyData	Происходит, когда база данных открывается или когда дана команда MODIFY DATABASE в отношении закрытой базы данных
DBC_CloseData	Закрывает открытую базу данных
DBC_PackData	Происходит до исполнения PACK DATABASE

Все указанные в таблице события база данных распознает. События могут быть обработаны процедурами, заголовки которых жестко определены, а сами процедуры разрабатываются программистами. Тексты процедур размещаются в хранимых процедурах. События будут распознаваться, если значение свойства базы данных `DBCEvents=.T.`.

Пример

```
PROCEDURE DBC_AfterCloseTable
nzap1=RECNO("users")
IF ! USED("users")
        USE data\users ORDER user IN 0
ELSE
    SELECT users
    SET ORDER TO user IN users
endif
GO nzap1
ENDPROC
```

Локальное представление данных

Чем же хороши представления? Тем, что они хранятся в базе данных, и их можно использовать для обновления данных в таблицах. Это, безусловно, их достоинство.

Но их "фишка" не столько в этом, сколько в том, что с ними можно работать как с обычными таблицами. С некоторой долей приближения можно считать `Local View` исходной таблицей. Это не просто обычный запрос Select-SQL, но запрос, обернутый в некий класс-оболочку, предоставляющий дополнительные возможности.

Создание представления (View) с помощью конструктора представлений

Для того чтобы создать представление, база данных должна быть открыта. Существуют два способа создать представление:

- из главного меню FoxPro: **File** | **New** | **View**;
- из командного окна FoxPro командой `CREATE VIEW`.

При этом запускается конструктор представлений **View Designer** (рис. 7.7).

В своей основе, `Local View` — это обычная команда Select-SQL. Основная масса закладок дизайнера **View** предназначена для конструирования выражения `Select-SQL`. На вкладке **Fields** отображается список полей, доступных для создания представления. Чтобы скопировать в представление все поля, мы можем:

- дважды щелкнуть на звездочке в верхней части окна **Available fields**;
- нажать кнопку **Add All**;

♦ перетащить поля в окно **Selected fields**.

Для удаления ошибочно выбранных полей используйте кнопки **Remove** и **Remove All**.

Обратите внимание на кнопку **Properties**. Она позволяет делать дополнительные настройки для выбранных полей. В этом окне можно задать правила контроля данных, выбрать формат ввода данных, ввести комментарии для каждого поля.

Во вкладке **Join** устанавливается объединение, если в представлении участвуют несколько таблиц.

Вкладка **Filter** позволяет вводить условия отбора записей. Они потом займут свое место в качестве параметра `WHERE` оператора `SQL – SELECT`.

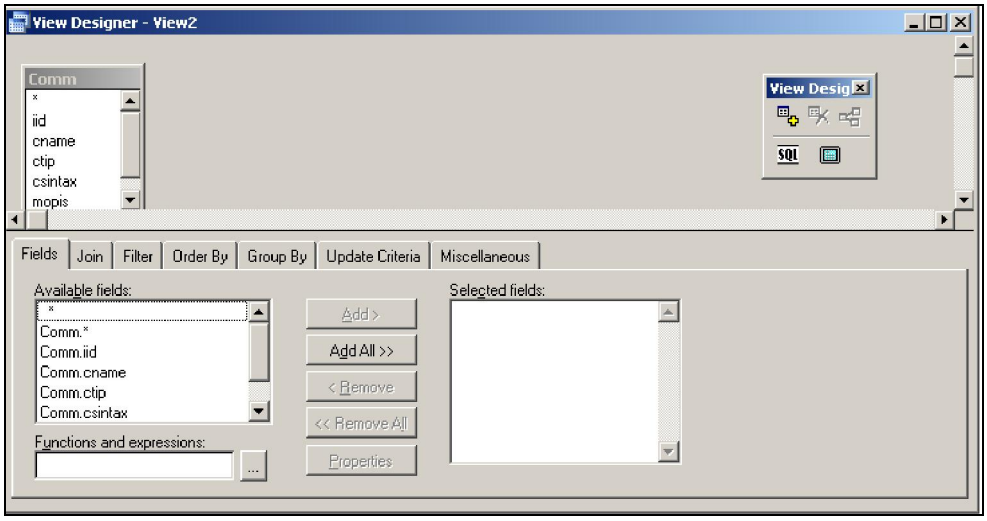


Рис. 7.7. Конструктор представлений

Во вкладке **Order By** устанавливается порядок сортировки записей. Для установки порядка нужно выбрать поле (или поля) в порядке иерархии сортировки и перетащить их в окно **Ordering criteria**.

Вкладка **Group By** позволяет произвести группировку записей по некоторому критерию, например, вычислить некую сумму. Вкладка **Update Criteria** посвящена настройкам, которые предоставляют возможность "сбросить" изменения, внесенные `Local View` в исходные таблицы. Здесь определяется как сама возможность такого "сброса", так и правила, по которым следует сопоставлять записи `Local View` и исходной таблицы. Смысл в том, что `Local View` — это все-таки другая таблица, не таблица-источник. Поэтому, чтобы сбросить изменения, сделанные во **View** в исходную таблицу, надо иметь однозначное соответствие — какая запись **View** какой записи таблицы-источника соответствует. Физически все настройки `Local View` сохраняются

в контейнере базы данных (файл DBC). Это значит, что вне контейнера базы данных создать Local View невозможно.

Любое **View** открывается в режиме оптимистической буферизации, по умолчанию — оптимистической буферизации строк, но можно переключить и в режим оптимистической буферизации таблиц. Отключить режим буферизации для **View** или установить для них пессимистический режим буферизации — невозможно. Пока **View** находится в режиме буферизации строк, его можно индексировать как обычную таблицу, через команду INDEX. Однако сохранить такой индекс невозможно. Необходимо при новом открытии **View** заново создавать для него индекс. Для использования буферизации необходимо сначала установить SET MULTILOCKS ON. Затем с помощью функции CURSORSETPROP() нужно установить один из методов буферизации.

Если в момент открытия (или перезапроса) Local View была установлена настройка SET DELETED OFF, то записи, помеченные как удаленные, будут выбраны во **View**, но уже без метки на удаление. Поэтому при работе с **View** надо особо тщательно следить за настройкой SET DELETED.

Создание представления (View) программным способом

Создать представление можно программно. Для создания используют команду

```
CREATE [SQL] VIEW [ViewName] [REMOTE]
   [CONNECTION ConnectionName [SHARE] | CONNECTION DataSourceName]
   [AS SQLSELECTStatement]
```

Если не указано предложение AS SQLSELECTStatement, то будет открыт проектировщик. Если же предложение указано, будет создано представление.

Поскольку представление создается на основе имеющейся базы данных, то при отсутствии открытой текущей базы данных VFP генерирует ошибку.

Параметры

ViewName — имя представления.

REMOTE — указывает, что используются удаленные таблицы.

Если отсутствует предложение CONNECTION, то открывается диалоговое окно для создания соединения.

CONNECTION ConnectionName — именованное соединение, используемое при открытии представления. Если указано предложение CONNECTION, то создается удаленное представление даже при опущенном REMOTE. Опция SHARE позволяет использовать представление другими представлениями.

1. Пример создания локального представления:

```
SET DEAFAULT TO d:\PROJECTS\DELO
CLOSE DATABASES
OPEN DATABASE data\Delo
CREATE SQL View lVxod AS Select * FROM Vxod
```

```
BROWSE
```

2. Пример создания удаленного представления:

```
CREATE SQL VIEW rView CONNECTION Fox AS  SELECT * FROM ISxod
WHERE Data_reg =DATE()
```

Функция

`REFRESH ([nRecords [, nRecordOffset]] [, cTableAlias | nWorkArea])`

восстанавливает данные представления, возвращая количество обновленных записей.

`nRecords` — число обновляемых записей.

`nRecordsOffset` — число записей, расположенных между первой обновляемой записью и текущей записью.

`CtableAlias | nWorkArea` — рабочая область представления.

Команды

```
DELETE VIEW ViewName и
DROP VIEW      ViewName
```

удаляют представление из текущей базы данных.

При этом база данных должна быть открыта монопольно.

Команда

`MODIFY VIEW` *`ViewName`* `[REMOTE][NOWAIT]`

отображает конструктор представлений, где вы можете изменять параметры SQL-представления.

Команда

`RENAME VIEW ViewName1 TO ViewName2`

переименовывает представление. База данных при этом должна быть открыта монопольно.

Функция

`REQUERY ([nWorkArea | cTableAlias])`

повторно загружает данные для представления. Возвращает 1, если данные обновлены, и 0 — в противном случае.

Функция

`ISMEMOFETCHED cFieldName | nFieldNumber [, nWorkArea | cTableAlias])`

определяет, выполнена ли выборка полей типа memo для режима "отложенной выборки полей `Memo`" (delayed memo fetch). Возвращает `.T.` в случае, если выполнено. В отличие от `REFRESH()` функция обновляет все поля представления.

Функция

`CREATEOFFLINE (ViewName [, cPath])`

переводит существующее представление в режим OffLine. При этом представление уже не может передавать изменения источнику данных.

Представление OffLine открывается командой USE. Когда представление открыто, вы можете добавлять данные, записи или изменять значения полей. Однако нельзя использовать команды CREATE TRIGGER, INSERT, PACK или ZAP для представления в режиме OffLine. После завершения необходимых изменений данных вы можете обновить данные в источнике (на сервере), для этого нужно открыть рассматриваемое представление OffLine при помощи команды USE, указывая дополнительную опцию ONLINE.

Вы не сможете обновить данные на сервере до тех пор, пока представление не будет открыто командой USE с опцией ONLINE.

Для перевода представление из режима OffLine в режим OnLine используется функция DROPOFFLINE( ).

Что такое *Rushmore*?

Впервые технология Rushmore представлена компанией Microsoft в FoxPro 2.0. Она состоит в том, чтобы использовать стандартные индексы для ускорения доступа к данным. При использовании этой технологии выполнение команды SEEK на больших таблицах происходит в сотни и тысячи раз быстрее обычного. Как же Rushmore удается достичь такого быстродействия? Для поиска нужной записи используется бинарный поиск. При этом используются индексы типа Regular, Candidate и Primary, потому что они имеют ссылку на каждую запись. Метод Rushmore требует, чтобы поисковое значение в точности совпадало с индексным ключом. Еще одно требование Rushmore — отсутствие логических операторов в индексе. Метод Rushmore подключается, если SET OPTIMIZE установлено в значение ON, причем оптимизация распространяется на все команды. Команда выполняется без оптимизации в случае использования опции NOOPTIMIZE.

Оптимизации подвержены команды, содержащие предложение FOR, и когда используемые условные выражения представлены в терминах, определяющих выражений индексов.

В SQL-командах для успешной оптимизации не требуется предварительного открытия ни таблиц, ни индексов. SQL самостоятельно решает вопрос о необходимости тех или иных индексов и, если они не существуют, создает их временными самостоятельно.

- Функция SYS(3054) позволяет контролировать режим оптимизации: None, Partial, Full.
- Определяя, допустима ли оптимизация в FOR-выражениях, VFP тратит некоторое время. Можно сэкономить это время, отключив оптимизацию посредством параметра NOOPTIMIZE или команды SET OPTIMIZE OFF.

Вот наиболее частые ошибки, при которых оптимизация перестает работать:

- не совпадает `Set Collate` (в индексах таблицы используется Russian, а системная установка — Machine);
- заведен индекс с условием `FOR`.

Выражение для поиска не совпадает с индексным. Часто допускают такую ошибку. У таблицы есть индекс по `UPPER(cField1)`. Для поиска пустого значения используют условие `EMPTY(cField1) = .T.`, что не совпадает с индексным выражением. Для того чтобы поиск был оптимизированным, выражение должно выглядеть как `UPPER(cField1) = SPACE(N)`, где `N` — длина поля `cField1`.

Полная оптимизация не всегда приводит к повышению быстродействия. Иногда наблюдается прямо противоположный эффект.