29장 데이터베이스 VCL의 확장

* BDE 사용법
* dBase 테이블
* 파라독스 테이블
* TDataSet의 확장

VCL의 데이터베이스 구조는 기본적으로 BDE(Borland Database Engine, 기능이 풍부하고 신뢰성도 높은 볼랜드의 데이터베이스 미들웨어이다)의 기능을 사용하도록 설계되어 있다. VCL의 데이터베이스 관련 컴포넌트들을 이용하면 서로 다른 여러 종류의 데이터베이스를 일관된 방식으로 조작할 수 있다. VCL은 신뢰성이나 처리 능력(scalability), 그리고 쉬운 사용법 같은 아주 중요한 장점들을 제공하지만, 특정 데이터베이스 고유의 기능이라던가 BDE가 VCL에게 제공하지 않는 기능들에 대해서는 취약한 편이다. 이 책의 앞부분에서 Win32 API를 직접 호출하는 방식으로 VCL 컴포넌트들의 기능을 확장하는 내용을 다룬 적이 있었는데, 이번 장은 그와 동일한 개념을 VCL 데이터베이스 컴포넌트와 BDE에 적용하겠다. 이번 장에서는 BDE의 기능을 직접 이용해서 VCL의 데이터베이스 관련 컴포넌트들의 기능을 확장하는 데 필요한 여러 가지 개념이나 기법들을 배울 수 있을 것이다.

BDE 사용법

BDE의 기능을 직접 호출하는 애플리케이션을 만들 때 반드시 기억해야 할 몇 가지 원칙들이 있다. 지금부터 델파이 애플리케이션에서 BDE API를 사용하는 데 필요한 여러 가지 정보들을 제시하겠다.

BDE 유닛

모든 BDE 함수나 데이터형, 상수들은 BDE 유닛에 정의되어 있다. BDE API를 사용하는 유닛의 uses절에는 반드시 BDE 유닛을 포함시켜야 한다. 그리고 BDE 유닛의 인터페이스 유닛은 \Delphi5\Doc 디렉토리의 BDE.INT이다. 이 파일을 BDE 함수나 레코드들에 대한 참조용으로 사용하면 된다.



BDE API에 대한 도움말은 BDE 디렉토리(기본 위치는 \Program Files\Borland\Common Files\BDE이다)의 BDE32.hlp이다. 이 도움말에는 모든 BDE API 함수들에 대한 설명뿐만 아니라 BDE에 대한 오브젝트 파스칼과 C의 훌륭한 예제들도 들어 있다.

Check() 함수

모든 BDE 함수들은 DBIRESULT형의 리턴값을 돌려준다. 이 값은 함수 호출의 성공 여부를 알려주는 것인데, 델파이는 이 값을 좀더 쉽게 처리하기 위한 함수인 Check() 라는 함수를 제공한다. Check() 함수는 DBIRESULT형이 값을 인자로 받아서 그 값에 해당하는 예외를 발생시킨다. 물론, 성공을 뜻하는 값이면 예외를 발생시키지 않는다. 다음은 BDE 함수 호출에 Check()를 사용하지 않는 예와 사용하는 예이다. 두 번째 예를 try..except/finally 블록으로 둘러싸면 좀더 완벽해 질 것이다.

```
// !! 좋지 않은 방식:
var
  Rez: DBIRESULT;
  A: array[0..dbiMaxUserNameLen] of char;
begin
  Rez := dbiGetNetUserName(A);      // BDE 함수를 호출
  if Rez <> DBIERR_NONE then        // 에러를 처리
      // 여기서 에러를 처리
  else begin
      // 작업을 계속 진행
  end;
end;

// !! 다음 방식을 권장한다.
var
  A: array[0..dbiMaxUserNameLen] of char;
begin
  { BDE 호출과 에러 처리를 한꺼번에 처리한다.
      에러가 있으면 예외가 발생된다. }
  Check(dbiGetNetUserName(A));
  // 작업을 계속 진행한다.
end;
```

커서와 핸들

BDE 함수들에는 커서(cursor)나 데이터베이스에 대한 핸들을 인자로 받아들이는 것들이 많이 있다. 간단히 말해서 커서 핸들이란 데이터셋 안의 특정 행(row, 레코드)을 뜻하는 BDE 객체이다. 커서 핸들의 데이터형은 hDBICur이다. 커서 핸들이라는

개념은 델파이가 사용하는 테이블이나 질의, 저장 프로시저의 "현재 레코드"라는 개념과 동일한 의미이다. TTable이나 TQuery, TStoredProc의 Handle 속성이 바로 커서 핸들이다. hDBICur형의 인자를 요구하는 BDE 함수를 사용할 때에는 TTable이나 TQuery, TStoredProc의 Handle 속성을 넣으면 된다.

일부 BDE 함수들은 데이터베이스에 대한 핸들도 필요로 한다. BDE 데이터베이스 핸들은 hDBIDb형으로 열려진 특정 데이터베이스를 뜻하는 객체이다. 열려진 특정 데이터베이스란, dDase나 파라독스의 경우에는 지역이나 네트웍상의 디렉토리를 뜻하며, SQL 서버 데이터베이스의 경우에는 서버 데이터베이스 파일을 뜻한다. TDatabase의 Handle 속성이 바로 데이터베이스 핸들이다. TDatabase 컴포넌트를 사용하지 않고 데이터베이스에 연결한 경우에는 TTabe이나 TQuery, TStoredProc 컴포넌트에 있는 DBHandle 속성을 사용하면 된다.

커서 동기화

앞에서 델파이 데이터셋은 현재 레코드라는 개념을 사용하는 반면, 데이터베이스를 실제로 조작하는 BDE는 커서라는 개념을 사용한다고 말했다. 델파이는 레코드들을 임시 버퍼, 즉 캐시에 저장하는 방식으로 수행 성능의 최적화하기 때문에 델파이의 현재 레코드가 가리키는 데이터와 BDE의 커서가 실제로 가리키는 데이터가 다를 수 있다. VCL의 데이터베이스 프레임웍을 그냥 사용하는 애플리케이션에서는 별 문제가 없겠지만, 커서를 인자로 요구하는 BDE 함수를 직접 호출해야 하는 애플리케이션에서는 문제가 생길 수 있다. 그런 경우에는 델파이의 커서 위치(즉, 현재 레코드)와 BDE의 커서를 동기화 시켜야 하는데, 다행히도 TDataSet을 상속하는 컴포넌트들에는 커서의 동기화를 수행해 주는 UpdateCursorPos()라는 메소드가 있다.

그리고 BDE 커서의 위치를 수정하는 BDE 함수를 호출한 후에도 BDE 커서와 VCL의 현재 레코드를 동기화시켜야 한다. 그렇게 하려면 BDE 호출이 끝난 직후 TDataSet 상속 컴포넌트의 CursorPosChanged() 메소드를 호출해야 한다. 다음은 커서 동기화 메소드들을 사용하는 방법을 보여주는 예이다.

```
procedure DoSomethingWithTable(T: TTable);
begin
  T.UpdateCursorPos;
  // 여기에서 커서의 위치를 변경하는 BDE 함수들을 호출한다.
  T.CursorPosChanged;
end;
```

dBase 테이블

dBase 테이블에는 델파이가 직접 지원하지 않는 여러 가지 유용한 기능들이 있다. 대표적인 것이 각 레코드에 대한 고유한 물리적 레코드 번호를 관리하는 기능이라든지, "잠정적 삭제(soft-delete. 테이블에서 레코드를 삭제하되 물리적으로 레코드를 제거하지는 않는 것)" 기능, 그리고 잠정적으로 삭제된 레코드를 다시 복구하거나 잠정적으로 삭제한 레코드들을 실제로 제거해서 테이블의 크기를 최적화하는 기능이다. 그럼 위에서 나열한 여러 기능들을 BDE 함수들을 통해서 실제로 사용하는 방법들을 살펴보고, 그 과정에서 TdBaseTable이라는 dBase 테이블에 특화된 TTable 상속 컴포넌트에 대해서도 설명하겠다.

물리적 레코드 번호

dBase 테이블은 테이블의 각 레코드에 대해 고유한 물리적 레코드 번호를 부여하고 관리한다. 물리적 레코드 번호란, 테이블의 시작 위치에 대한 레코드의 물리적 상대 위치를 뜻한다. 물리적 레코드 번호를 얻으려면 BDE의 dbiGetRecord() 함수를 호출해야 하는데, 이 함수의 선언은 다음과 같다.

```
function DbiGetRecord(hCursor: hDBICur; eLock: DBILockType;
      pRecBuff: Pointer; precProps: pRECProps): DBIResult stdcall;
```

hCursor는 커서 핸들이다. 일반적으로, 이 인자에는 TDataSet 상속 객체의 Handle 속성을 넣는다.

eLock은 레코드에 적용할 잠금의 종류인데, 다음과 같이 정의되어 있는 DBILockType 열거형의 값을 사용한다.

```
type
  DBILockType = (
    dbiNOLOCK,          // 잠그지 않음(기본값)
    dbiWRITELOCK,       // 쓰기 잠금
    dbiREADLOCK         // 읽기 잠금
  );
```

물리적 레코드 번호를 얻을 때에는 레코드의 내용을 수정할 필요가 없으므로 dbiNOLOCK을 사용하면 된다.

pRecBuff는 레코드 버퍼를 가리키는 포인터이다. 물리적 레코드 번호를 얻는다는 것은 레코드의 데이터가 아니라 레코드 자체의 정보를 얻기 위한 것이므로 이 인자에는 그냥 Nil을 넣으면 된다.

pRecProps는 RECProps 레코드에 대한 포인터인데, 이 구조체는 다음과 같이 정의되어 있다.

```
type
  pRECProps = ^RECProps;
  RECProps = packed record       // 레코드의 속성들
    iSeqNum         : Longint;   // 일련 번호를 지원하는 경우에만 유효함.
    iPhyRecNum      : Longint;   // 물리적 레코드 번호를 지원하는 경우에만
                                 // 유효함.
    iRecStatus      : Word;      // 레코드의 상태
    bSeqNumChanged  : WordBool;  // 사용되지 않음.
    bDeleteFlag     : WordBool;  // 잠정적 삭제를 지원하는 경우에만 유효함.
  end;
```

이 레코드를 통해서 다양한 정보를 얻을 수 있다. 지금 관심이 있는 것은 iPhyRecNum 필드이다. 이 필드는 dBase나 폭스프로 테이블에서만 의미가 있다.

그럼 현재 레코드의 물리적 레코드 번호를 돌려주는 TdBaseTable의 GetRecNum() 메소드를 살펴보자.

```
function TdBaseTable.GetRecNum: Longint;
{ 현재 레코드의 물리적 레코드 번호를 돌려준다. }
var
  RP: RECProps;
begin
  UpdateCursorPos;             // 델파이에서 BDE를 갱신한다.
  { 현재 레코드에 대한 정보를 얻는다. }
  Check(dbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
  Result := RP.iPhyRecNum;     // 물리적 레코드 번호를 돌려준다.
end;
```

삭제된 레코드들의 표시

dBase 테이블에서 잠정적으로 삭제된 레코드들을 보려면 dbiSetProp()라는 BDE 함수를 이용한다. 이 함수는 여러 가지 BDE 객체들의 다양한 속성들을 수정하는 데 사용하는 다용도 함수이다. 이 함수의 자세한 사용법은 BDE 도움말의 "properties-getting and setting" 부분을 참고하기 바란다. 이 함수는 다음과 같이 선언되어 있다.

```
function DbiSetProp (hObj: hDBIObj; iProp: Longint;
  iPropValue: Longint): DBIResult stdcall;
```

hObj 인자는 속성을 설정하고자 하는 BDE 객체의 핸들이다. 이번 경우에는 커서 핸들을 넣으면 된다.

iProp 인자는 설정하고자 하는 속성을 뜻하는 식별자이다. 이 인자에 사용할 수 있는 값들은 앞에서 이야기한 BDE 도움말 부분에 나와 있다. 삭제된 레코드들의 뷰를 활성화/비활성화시키는 데 관련된 값은 curSOFTDELETEON이다.

iPropValue에는 iProp 인자로 지정한 속성에 설정한 값을 넣는다. iProp에 curSOFTDELETEON을 지정한 경우, iPropValue에 0을 넣으면 삭제된 레코드들을 표시하지 않는 것이고, 1을 넣으면 표시하는 것이다.

다음 코드는 TdBaseTable의 SetViewDeleted() 메소드이다.

```
procedure TdBaseTable.SetViewDeleted(Value: Boolean);
{ 삭제된 레코드들의 표시 여부를 설정하는 메소드 }
begin
  { 테이블이 활성화되어 있어야 한다. }
  if Active and (FViewDeleted <> Value) then begin
    DisableControls;     // 컨트롤들이 깜박거리지 않게 한다.
    try
          { 잠정적 삭제 레코드들의 표시 여부를 설정한다. }
      Check(dbiSetProp(hdbiObj(Handle), curSOFTDELETEON, Longint(Value)));
    finally
      Refresh;           // 델파이쪽을 갱신한다.
      EnableControls;   // 컨트롤들을 다시 활성화시킨다.
    end;
    FViewDeleted := Value
  end;
end;
```

이 메소드는 우선 테이블이 열려 있는지 그리고 설정하고자 하는 값이 객체의 FViewDeleted 저장 필드에 이미 설정된 값과 같지는 않은지를 점검한다. 그런 다음, 테이블에 연결된 데이터 인식 컨트롤들이 깜박거리지 않도록 하기 위해 DisableControls()를 호출한다. 그리고 실제 작업을 수행하는 dbiSetProp()를 호출하는데, 이때 hDBICur형인 Handle 속성을 hDBIObj형으로 형변환시킨다는 점을 주목하기 바란다. hDBIObj는 모든 종류의 BDE 객체들에 사용할 수 있는 범용적인 핸들형이라고 생각하면 된다. dbiSetProp()를 호출한 다음에는 데이터셋을 갱신하고, 데이터 인식 컨트롤들도 다시 활성화시킨다.



데이터셋에 연결된 데이터 인식 컨트롤들의 작동을 유보시키기 위해 DisableControls()를 호출하는 경우, try..finally 블록을 이용해서 반드시 EnableControls()를 호출하도록 해야 한다. 그렇게 하면 예외가 발생하더라도 컨트롤들은 반드시 다시 활성화된다.

레코드의 삭제 여부 판별

삭제된 레코드들이 포함된 데이터셋을 다루는 경우, 현재 레코드가 삭제된 것인지 아닌지를 판단할 수 있어야 한다. dbiGetRecord()에 대한 설명 부분을 유심히 본 독자라면 레코드의 삭제 여부를 어떻게 알아내는지 이미 눈치를 챘을 것이다. 다음의 코드는 레코드의 삭제 여부를 판단하는 메소드인데, 앞에서 나온 GetRecNum() 메소드와 이 메소드의 차이는 RECProps 레코드의 iPhyRecNo 필드 대신 bDeletedFlag 필드의 값을 점검한다는 것뿐이다.

```
function TdBaseTable.GetIsDeleted: Boolean;
{ 현재 레코드가 잠정적으로 삭제되었는지를 뜻하는 부울값을 돌려준다. }
var
  RP: RECProps;
begin
  if not FViewDeleted then     // 삭제된 레코드를 표시하지 않는 경우에는
```

```
    Result := False           // 판단할 필요가 없다.
  else begin
    UpdateCursorPos;          // 델파이에서 BDE를 갱신한다.
    { 현재 레코드에 대한 정보를 얻는다. }
    Check(dbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
    Result := RP.bDeletedFlag;  // 레코드의 삭제 여부를 돌려준다.
  end;
end;
```

삭제된 레코드의 복구

지금까지 삭제된 레코드들을 표시하거나 레코드가 삭제되었는지를 알아내는 방법들을 이야기했으니 이제 잠정적으로 삭제된 레코드를 되살리는 방법을 이야기해야 하겠다. 삭제된 레코드를 복구하려면, 우선 현재 레코드가 삭제된 것인지 아닌지를 알아야 하는데, 그것은 이미 앞의 메소드로 구현했다. 그 다음은 레코드를 실제로 복구하는 방법이 필요한데, 다행히 BDE에는 dbiUndeleteRecord()라는 함수를 제공한다. 이름만 봐도 어떤 함수인지 알 수 있을 것이다. 이 함수의 선언은 다음과 같다.

```
function DbiUndeleteRecord (hCursor: hDBICur): DBIResult stdcall;
```

유일한 인자인 hCursor는 현재 데이터셋의 커서 핸들을 뜻한다. 다음은 이 함수를 이용해서 삭제된 레코드를 복구하는 TdBaseTable의 UndeleteRecord() 메소드이다.

```
procedure TdBaseTable.UndeleteRecord;
begin
  if not IsDeleted then
    raise EDatabaseError.Create('Record is not deleted');
  Check(dbiUndeleteRecord(Handle));
  Refresh;
end;
```

테이블의 패킹

잠정적으로 삭제된 레코드들을 실제로 제거하려면 테이블에 패킹(packing)이라는 작업을 수행해야 한다. BDE에는 패킹을 수행하는 dbiPackTable()이란 함수가 있는데, 그 함수의 선언은 다음과 같다.

```
function DbiPackTable (hDb: hDBIDb; hCursor: hDBICur;
       pszTableName: PChar; pszDriverType: PChar;
       bRegenIdxs: Bool): DBIResult stdcall;
```

hDb는 데이터베이스에 대한 핸들이다. 이 인자에는 TDataSet 상속 컴포넌트의 DBHandle 속성이나 TDatabase 컴포넌트의 Handle 속성을 넣는다.

hCursor는 커서 핸들이다. 이 인자에는 TDataset 상속 컴포넌트의 Handle 속성을 넣으면 된다. pszTableName과 pszDriverType 인자를 이용해서 테이블을 지정하는 경우에는 이 인자에 nil을 넣는다.

pszTableName은 테이블의 이름을 뜻하는 문자열을 가리키는 포인터이다.

pszDriverType은 테이블의 드라이버 종류를 뜻하는 문자열을 가리키는 포인터이다. hCursor에 Nil을 넣는 경우, 이 인자는 반드시 szdBase이어야 한다. 그러나 이 함수는 dBase 테이블만을 위한 것이기 때문에 이 인자를 직접 지정하는 경우는 드물다는 점을 지적하고 싶다.

bRegenIdxs 인자는 테이블에 관련된 유효하지 않은 모든 색인들을 다시 생성할 것인지를 뜻한다.

다음은 TdBaseTable 컴포넌트의 Pack() 메소드이다.

```
procedure TdBaseTable.Pack(RegenIndexes: Boolean);
{ 잠정적으로 삭제된 레코드들을 실제로 제거하기 위해 테이블을
  패킹한다. }
const
  SPackError = 'Table must be active and opened exclusively';
begin
  { 테이블이 활성화되어 있어야 하며, 독점 모드로 열려 있어야 한다. }
  if not (Active and Exclusive) then
    raise EDatabaseError.Create(SPackError);
```

```
  try
    { 테이블을 패킹한다. }
    Check(dbiPackTable(DBHandle, Handle, Nil, Nil, RegenIndexes));
  finally
    { BDE에서 델파이쪽을 갱신한다. }
    CursorPosChanged;
    Refresh;
  end;
end;
```

TdBaseTable 컴포넌트의 전체 소스는 잠시 후에 제시할 목록 29.1에 나와 있다.

파라독스 테이블

파라독스(Pardox) 테이블에는 잠정적 삭제 같은 유용한 기능은 없지만, 그래도 테이블 패킹이나 레코드 일련 번호 같은 개념을 지원한다. 그럼 TTable 컴포넌트를 상속해서 파라독스 테이블에 관련된 작업을 수행하는 TParadoxTable라는 클래스를 만들어 보자.

일련 번호

파라독스 테이블에는 dBase 테이블의 물리적 레코드 번호 같은 개념은 없다. 그 대신 테이블의 각 레코드마다 일련 번호(sequence number)를 매겨서 레코드들을 관리한다. 일련 번호는 물리적 레코드 번호와는 달리 테이블에 현재 적용된 색인에 따라 달라진다. 레코드의 일련 번호는 현재 색인에 기반해서 레코드들이 정렬된 순서에 의존하는 것이다.

BDE에서 레코드의 일련 번호를 얻을 때 사용하는 함수는 dbiGetSeqNo()인데, 이 함수의 선언은 다음과 같다.

```
function DbiGetSeqNo (hCursor: hDBICur; var iSeqNo: Longint): DBIResult
  stdcall;
```

hCursor는 파라독스 테이블의 커서 핸들이다. 그리고 이 함수를 호출하면 iSeqNo 인자에 현재 레코드의 레코드 일련 번호가 들어간다. 다음은 TParadoxTable 컴포넌트의 GetRecNum() 메소드이다.

```
function TParadoxTable.GetRecNum: Longint;
{ 현재 레코드의 일련 번호를 돌려준다. }
begin
  UpdateCursorPos;            // 델파이에서 BDE를 갱신한다.
  { 현재 레코드의 일련 번호를 Result에 넣는다. }
  Check(dbiGetSeqNo(Handle, Result));
end;
```

테이블 패킹

파라독스는 잠정적 삭제를 지원하지 않기 때문에 파라독스 테이블의 패킹은 dBase 테이블의 패킹과는 조금 다른 의미를 가진다. 파라독스 테이블에서 어떤 레코드가 삭제되면 그 레코드는 테이블에서 실제로 제거되나 그 레코드가 있던 자리는 그대로 비어 있게 된다. 즉, 테이블 파일 중간에 구멍이 생기는 것이다. 파라독스 테이블의 패킹이란, 그러한 구멍들을 제거해서 테이블의 크기를 압축하는 작업을 말한다.

dBase 테이블에서와는 달리 파라독스 테이블의 패킹을 수행하는 직접적인 BDE 함수는 없다. 대신 dbiDoRestructure()라는 함수를 호출해서 테이블을 재구축하는 과정에서 테이블의 패킹이 일어난다. dbiDoRestructure() 함수의 선언은 다음과 같다.

```
function DbiDoRestructure (hDb: hDBIDb; iTblDescCount: Word;
  pTblDesc: pCRTblDesc; pszSaveAs: PChar; pszKeyviolName:
  PChar; pszProblemsName: PChar; bAnalyzeOnly: Bool): DBIResult stdcall;
```

hDb는 데이터베이스의 핸들인데, 이 함수는 이미 열려진 테이블에 대해서는 작동하지 않는다. 그런데 문제는 델파이 TDataset 상속 컴포넌트의 경우 테이블을 열지 않는 이상 DBHandle 속성을 이용할 수가 없다는 점이다. 그러한 문제를 해결하려면 임시 데이터베이스를 만들어야 하는데, 이에 대해서는 잠시 후에 나올 TParadoxTable.Pack()의 코드에서 다시 설명하겠다.

iTblDescCount 인자는 테이블 서술자(decriptor)의 개수이다. 현재 버전의 BDE는 한 번의 함수 호출에 대해 하나의 테이블 서술자만을 지원하므로 이 인자에는 반드시 1을 넣어야 한다.

pTblDesc는 테이블 서술자를 뜻하는 CRTblDesc 레코드에 대한 포인터이다. 이 레코드에는 재구축하려는 테이블이 어떤 테이블인지에 대한 정보가 들어간다. 이 레코드의 정의는 다음과 같다.

```
type
  pCRTblDesc         = ^CRTblDesc;
  CRTblDesc = packed record     // 테이블 서술자를 생성/재구축
    szTblName      : DBITBLNAME; // 테이블 이름. 경로와 확장자가 포함될 수도 있음.
    szTblType      : DBINAME;   // 드라이버 종류(생략 가능)
    szErrTblName   : DBIPATH;   // 에러 테이블 이름(생략 가능)
    szUserName     : DBINAME;   // 사용자 이름(적용 가능한 경우)
    szPassword     : DBINAME;   // 패스워드(생략 가능)
    bProtected     : WordBool;  // szPassword로 제공된 마스터 패스워드
    bPack          : WordBool;  // 테이블을 패킹(재구축인 경우에만)
    iFldCount      : Word;      // 제공된 필드 정의 개수
    pecrFldOp      : pCROpType; // 필드 작업 배열
    pfldDesc       : pFLDDesc;  // 필드 서술자 배열
    iIdxCount      : Word;      // 제공된 색인 정의 개수
    pecrIdxOp      : pCROpType; // 색인 작업 배열
    pidxDesc       : PIDXDesc;  // 색인 서술자 배열
    iSecRecCount   : Word;      // 제공된 보안 정의 개수
    pecrSecOp      : pCROpType; // 보안 작업 배열
    psecDesc       : pSECDesc;  // 보안 서술자 배열
    iValChkCount   : Word;      // 유효성 점검 개수
    pecrValChkOp   : pCROpType; // 유효성 점검 작업 배열
    pvchkDesc      : pVCHKDesc; // 유효성 점검 서술자 배열
    iRintCount     : Word;      // 참조 무결성 지정자 개수
    pecrRintOp     : pCROpType; // 참조 무결성 배열
    printDesc      : pRINTDesc; // 참조 무결성 지정자 배열
    iOptParams     : Word;      // 추가적 인자들의 개수
    pfldOptParams  : pFLDDesc;  // 필드 서술자 배열
    pOptData       : Pointer;   // 추가적인 인자들
  end;
```

파라독스 테이블의 패킹을 수행하려면 szTblName, szTblType 그리고 bPack 필드의 값만 정확히 설정하면 된다.

다시 DbiDoRestructure()로 돌아가자. pszSaveAs 인자는 추가적인 문자열 포인터인데, 재구축된 테이블을 원본 테이블과는 다른 파일에 저장할 때 사용한다.

pszKeyvioName은 추가적인 문자열 포인터인데 재구축 과정에서 키 위반을 일으킨 레코드들을 보낼 테이블을 지정하기 위한 것이다.

pszProblemName은 추가적인 문자열 포인터로, 재구축 과정에서 문제를 일으킨 레코드들을 보낼 테이블을 지정하기 위한 것이다.

bAnalyzeOnly 인자는 사용되지 않는다.

다음 코드는 TPardoxTable의 Pack() 메소드이다. 이 메소드를 보면 CRTblDesc 레코드를 초기화하는 방법과 dbiOpenDatabase() 함수에 사용할 임시 데이터베이스를 만드는 방법을 알 수 있다. 그리고 finally 블록을 이용해서 임시 데이터베이스를 확실히 닫는다는 점도 기억해 둘 만하다.

```
procedure TParadoxTable.Pack;
var
  TblDesc: CRTblDesc;
  TempDBHandle: HDBIDb;
  WasActive: Boolean;
begin
  { TblDesc 레코드를 초기화 }
  FillChar(TblDesc, SizeOf(TblDesc), 0); // 0들로 채운다.
  with TblDesc do begin
    StrPCopy(szTblName, TableName);      // 테이블 이름을 설정
    szTblType := szPARADOX;              // 테이블 종류를 설정
    bPack := True;                       // 팩 플래그를 설정
  end;
  { 테이블의 활성화 상태를 보존하고, 패킹을 수행하기 위해 테이블을 닫는다. }
  WasActive := Active;
  if WasActive then Close;
  try
      { 임시 데이터베이스를 생성한다. 반드시 읽기-쓰기/독점 모드로 열어야 한다. }
```

```
    Check(dbiOpenDatabase(PChar(DatabaseName), Nil, dbiREADWRITE,
          dbiOpenExcl, Nil, 0, Nil, Nil, TempDBHandle));
    try
      { 테이블을 패킹한다. }
      Check(dbiDoRestructure(TempDBHandle, 1, @TblDesc, Nil, Nil, Nil,
            False));
    finally
      { 임시 데이터베이스를 닫는다. }
      dbiCloseDatabase(TempDBHandle);
    end;
  finally
    { 테이블의 활성화 상태를 복구한다. }
    Active := WasActive;
  end;
end;
```

목록 29.1은 TdBaseTable과 TParadoxTable 객체를 정의하는 DDGTbls.pas 유닛의 소스 코드이다.

목록 29.1 Comps\DDGTbls.pas

```
unit DDGTbls;

interface

uses DB, DBTables, BDE;

type
  TdBaseTable = class(TTable)
  private
    FViewDeleted: Boolean;
    function GetIsDeleted: Boolean;
    function GetRecNum: Longint;
    procedure SetViewDeleted(Value: Boolean);
  protected
    function CreateHandle: HDBICur; override;
```

```
  public
    procedure Pack(RegenIndexes: Boolean);
    procedure UndeleteRecord;
    property IsDeleted: Boolean read GetIsDeleted;
    property RecNum: Longint read GetRecNum;
    property ViewDeleted: Boolean read FViewDeleted write SetViewDeleted;
  end;

  TParadoxTable = class(TTable)
  private
  protected
    function CreateHandle: HDBICur; override;
    function GetRecNum: Longint;
  public
    procedure Pack;
    property RecNum: Longint read GetRecNum;
  end;

implementation

uses SysUtils;

{ TdBaseTable }

function TdBaseTable.GetIsDeleted: Boolean;
{ 현재 레코드가 잠정적으로 삭제되었는지를 뜻하는 부울값을 돌려준다. }
var
  RP: RECProps;
begin
  if not FViewDeleted then     // 삭제된 레코드를 표시하지 않는 경우에는
    Result := False           // 판단할 필요가 없다.
 else begin
   UpdateCursorPos;            // 델파이에서 BDE를 갱신한다.
   { 현재 레코드에 대한 정보를 얻는다. }
   Check(dbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
   Result := RP.bDeletedFlag;     // 레코드의 삭제 여부를 돌려준다.
 end;
end;
```

```
function TdBaseTable.GetRecNum: Longint;
{ 현재 레코드의 물리적 레코드 번호를 돌려준다. }
var
  RP: RECProps;
begin
  UpdateCursorPos;            // 델파이에서 BDE를 갱신한다.
  { 현재 레코드에 대한 정보를 얻는다. }
  Check(dbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
  Result := RP.iPhyRecNum;     // 물리적 레코드 번호를 돌려준다.
end;

function TdBaseTable.CreateHandle: HDBICur;
{ 이 컴포넌트가 연결한 테이블이 dBase 테이블인지를 확인하도록 재정의된
  메소드이다. }
var
  CP: CURProps;
begin
  Result := inherited CreateHandle;        // 부모 메소드를 상속
  if Result <> Nil then begin
     { 커서 속성들을 얻은 다음, dBase 테이블이 아니면 예외를 발생시킨다. }
    Check(dbiGetCursorProps(Result, CP));
    if not (CP.szTableType = szdBase) then
      raise EDatabaseError.Create('Not a dBase table');
  end;
end;

procedure TdBaseTable.Pack(RegenIndexes: Boolean);
{ 잠정적으로 삭제된 레코드들을 실제로 제거하기 위해 테이블을 패킹한다. }
const
  SPackError = 'Table must be active and opened exclusively';
begin
  { 테이블이 활성화되어 있어야 하며, 독점 모드로 열려 있어야 한다. }
  if not (Active and Exclusive) then
    raise EDatabaseError.Create(SPackError);
  try
    { 테이블을 패킹한다. }
    Check(dbiPackTable(DBHandle, Handle, Nil, Nil, RegenIndexes));
  finally
```

```
    { BDE에서 델파이쪽을 갱신한다. }
    CursorPosChanged;
    Refresh;
  end;
end;

procedure TdBaseTable.SetViewDeleted(Value: Boolean);
{ 삭제된 레코드들의 표시 여부를 설정하는 메소드 }
begin
 { 테이블이 활성화되어 있어야 한다. }
  if Active and (FViewDeleted <> Value) then begin
    DisableControls;     // 컨트롤들이 깜박거리지 않게 한다.
    try
          { 잠정적 삭제 레코드들의 표시 여부를 설정한다. }
      Check(dbiSetProp(hdbiObj(Handle), curSOFTDELETEON, Longint(Value)));
    finally
      Refresh;           // 델파이쪽을 갱신한다.
      EnableControls;    // 컨트롤들을 다시 활성화시킨다.
    end;
    FViewDeleted := Value
  end;
end;

procedure TdBaseTable.UndeleteRecord;
begin
  if not IsDeleted then
    raise EDatabaseError.Create('Record is not deleted');
  Check(dbiUndeleteRecord(Handle));
  Refresh;
end;

function TParadoxTable.CreateHandle: HDBICur;
{ 이 컴포넌트가 연결한 테이블이 파라독스 테이블인지를 확인하도록 재정의된
  메소드이다. }
var
  CP: CURProps;
begin
  Result := inherited CreateHandle;         // 부모 메소드를 상속한다.
  if Result <> Nil then begin
```

```
      { 커서 속성들을 얻은 다음, 파라독스 테이블이 아니면
          예외를 발생시킨다. }
    Check(dbiGetCursorProps(Result, CP));
    if not (CP.szTableType = szPARADOX) then
      raise EDatabaseError.Create('Not a Paradox table');
  end;
end;

function TParadoxTable.GetRecNum: Longint;
{ 현재 레코드의 일련 번호를 돌려준다. }
begin
  UpdateCursorPos;            // 델파이에서 BDE를 갱신한다.
  { 현재 레코드의 일련 번호를 Result에 넣는다. }
  Check(dbiGetSeqNo(Handle, Result));
end;

procedure TParadoxTable.Pack;
var
  TblDesc: CRTblDesc;
  TempDBHandle: HDBIDb;
  WasActive: Boolean;
begin
  { TblDesc 레코드를 초기화 }
  FillChar(TblDesc, SizeOf(TblDesc), 0); // 0들로 채운다.
  with TblDesc do begin
    StrPCopy(szTblName, TableName);      // 테이블 이름을 설정
    szTblType := szPARADOX;              // 테이블 종류를 설정
    bPack := True;                       // 패킹을 수행하도록 설정
  end;
 { 테이블의 활성화 상태를 보존하고, 패킹을 수행하기 위해 테이블을 닫는다. }
  WasActive := Active;
  if WasActive then Close;
  try
     { 임시 데이터베이스를 생성한다. 반드시 읽기-쓰기/독점 모드로 열어야 한다. }
    Check(dbiOpenDatabase(PChar(DatabaseName), Nil, dbiREADWRITE,
          dbiOpenExcl, Nil, 0, Nil, Nil, TempDBHandle));
    try
      { 테이블을 패킹한다. }
      Check(dbiDoRestructure(TempDBHandle, 1, @TblDesc, Nil, Nil, Nil,
```

```
            False));
      finally
        { 임시 데이터베이스를 닫는다. }
        dbiCloseDatabase(TempDBHandle);
      end;
    finally
      { 테이블의 활성화 상태를 복구한다. }
      Active := WasActive;
    end;
  end;

  end.
```

TQuery 결과셋의 레코드 개수 제한

SQL 질의문의 조건을 느슨하게 만들면 질의 수행 결과로 생기는 결과셋에 많은 레코드들이 포함되게 된다. 결과셋이 크면 클수록 결과셋이 완성되기까지의 시간도 길어지며, 또한 네트웍의 부담도 가중된다. 그리고 사용자 입장에서는 멍하니 화면만을 쳐다보고 있어야 하는 시간이 길어진다는 의미도 된다. SQL 질의문은 꼭 필요한 데이터들만을 얻을 수 있도록 작성하는 것이 원칙이겠지만, 데이터베이스 애플리케이션을 만들다 보면 많은 개수의 레코드들이 포함되는 결과셋을 얻어야 하는 경우가 생길 수 있다. 그런 경우, 하나의 결과셋에 들어갈 수 있는 레코드 개수를 제한할 필요가 있는데, TQuery 자체에는 그러한 기능이 없다. 그러나 BDE 함수를 적절히 이용하면 결과셋의 크기를 제한할 수 있는 TQuery 상속 객체를 별로 어렵지 않게 작성할 수 있다.

레코드 개수를 제한할 때 사용하는 BDE 함수는 앞에서도 설명한 바 있는 다목적 함수 DbiSetProp()이다. 이 함수의 첫 번째 인자에는 질의의 커서 핸들을 넣어야 하며, 크기 제한이 목적인 경우, 두 번째 인자에는 curMAXROWS를 넣는다. 그리고 마지막 인자에는 제한하고자 하는 레코드 개수를 넣는다.

TQuery의 메소드들 중 DbiSetProp()을 사용하기에 가장 적합한 것은 질의가 열리면 즉시 호출되는 PrepareCursor() 메소드이다. PrepareCursor()는 질의가 열리면 즉시 호출된다. 목록 29.2는 TQuery를 상속하는 TRestrictedQuery 컴포넌트가 정의된 ResQuery.pas 유닛의 소스 코드이다.

목록 29.2 ResQuery.pas 유닛의 소스 코드

```
unit ResQuery;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
 Dialogs, DB, DBTables, BDE;

type
  TRestrictedQuery = class(TQuery)
  private
    FMaxRowCount: Longint;
  protected
    procedure PrepareCursor; override;
  published
    property MaxRowCount: Longint read FMaxRowCount write FMaxRowCount;
  end;

implementation

procedure TRestrictedQuery.PrepareCursor;
begin
  inherited PrepareCursor;
  if FMaxRowCount > 0 then
  begin
    Check(DbiValidateProp(hDBIObj(Handle), curMAXROWS, True));
    Check(DbiSetProp(hDBIObj(Handle), curMAXROWS, FMaxRowCount));
  end;
end;

 end.
```

이 컴포넌트로 수행한 질의의 결과로 생기는 결과셋의 레코드 개수는 MaxRowCount 속성에 설정된 값(0 이상이어야 함)보다 작게 된다. 그림 29.1은 레코드 개수를 3으로 제한한 질의를 SQL 모니터를 이용해서 확인한 모습이다.

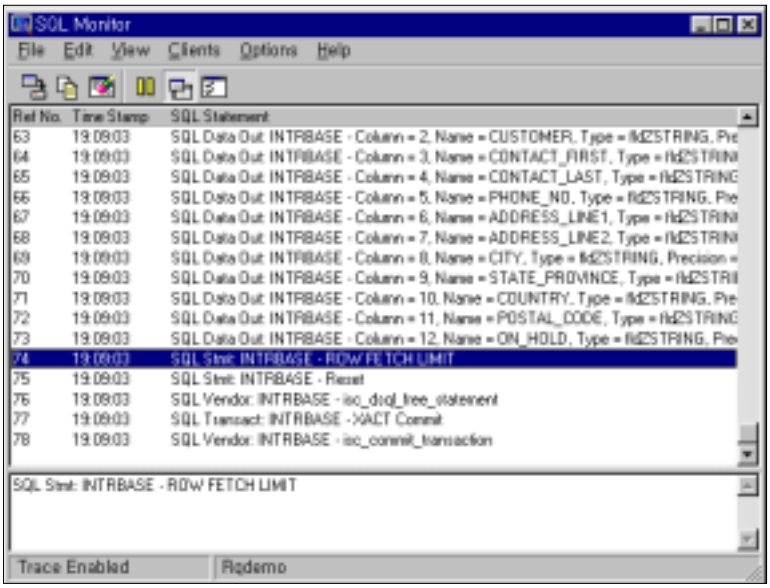


그림 29.1
SQL 모니터를 이용해서 크기 제한 질의를 확인

기타 BDE 관련 기법들

데이터베이스 애플리케이션 개발 과정에는 반복적으로 수행하는 작업들이 몇 가지 있다. 테이블에 대한 SQL 관련 Aggregate 함수의 수행이라든지, 테이블 복사 그리고 특정 세션에 대한 파라독스 사용자 목록을 얻는 작업이 좋은 예이다.

SQL Aggregate 함수들

일반적으로 말해서 SQL Aggregate 함수란 하나나 그 이상의 레코드들에 있는 하나나 그 이상의 필드들에 대해 연산 작업을 수행하는 SQL 함수들을 뜻한다. 여러 레코드들에 있는 여러 필드값들의 총합을 구하는 SUM()이라든지, 여러 레코드들에 있는 여러 필드값들의 평균을 구하는 AVG() 그리고 여러 레코드들에 있는 여러 필드값들의 최대값이나 최소값을 구하는 MAX()나 MIN() 등이 총계 함수이다.

델파이에서 Aggregate 함수를 사용하는 것은 좀 번거로운 편이다. 예를 들어서, TTable로 테이블을 연 경우 총계 함수를 사용하려면 TQuery를 만들고, 정확한 SQL 질의문을 작성하고, 질의를 수행하고, 질의로부터 결과를 얻는 과정을 거쳐야 한다.

이러한 과정을 자동화시켜 놓고 한 번에 처리할 수 있으면 무척 편할 것이다. 목록 29.3이 바로 그러한 작업을 하는 코드이다.

목록 29.3 SQL 총계 함수들의 자동화

```
type
  TSQLAggFunc = (safSum, safAvg, safMin, safMax);

const
  // SQL 총계 함수들
  SQLAggStrs: array[TSQLAggFunc] of string = (
    'select sum(%s) from %s',
    'select avg(%s) from %s',
    'select min(%s) from %s',
    'select max(%s) from %s');

function CreateQueryFromTable(T: TTable): TQuery;
// 주어진 테이블과 같은 데이터베이스와 세션에 연결된 TQuery 객체를 돌려준다.
begin
  Result := TQuery.Create(nil);
  try
    Result.DatabaseName := T.DatabaseName;
    Result.SessionName := T.SessionName;
  except
    Result.Free;
    Raise;
  end;
end;

function DoSQLAggFunc(T: TTable; FieldNames: string;
  Func: TSQLAggFunc): Extended;
begin
  with CreateQueryFromTable(T) do
  begin
    try
      SQL.Add(Format(SQLAggStrs[Func], [FieldNames, T.TableName]));
      Open;
      Result := Fields[0].AsFloat;
```

```
    finally
      Free;
    end;
  end;
end;

function SumField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safSum);
end;

function AvgField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safAvg);
end;

function MinField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safMin);
end;

function MaxField(T: TTable; Field: string): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safMax);
end;
```

목록을 보면 각각의 총계 함수에 해당하는 델파이 함수들이 정의되어 있는데, 그 함수들은 모두 DoSQLAggFun()이라는 함수를 호출한다. DoSQLAggFun() 함수는 일단 CreateQueryFromTable() 함수를 호출해서 T인자로 넘겨 받은 테이블과 동일한 데이터베이스와 세션을 가지는 TQuery 컴포넌트를 얻는다. 그런 다음, 그 TQuery 컴포넌트에 적절한 SQL 질의문을 설정해서 질의를 수행하고, 질의 결과를 호출한 쪽에 돌려준다.

빠른 테이블 복사

어떤 테이블의 복사본을 만드는 작업 역시 매우 반복적이다. Win32 API 함수 Copy() 파일을 이용해서 테이블 파일을 물리적으로 복사할 수도 있고, 아니면 TBatchMove 컴포넌트를 이용해서 TTable의 테이블을 다른 TTable로 복사할 수도 있다. 아니면 TTable의 BatchMove() 함수를 이용해서 복사를 할 수도 있다.

만일 다른 사용자나 프로세스가 해당 테이블 파일을 열어 놓은 상태이거나 테이블이 SQL 서버에 있는 데이터베이스 파일 안에 포함되어 있다면 CopyFile() API를 사용하는 방법은 적용할 수 없다. 그리고 테이블뿐만 아니라 그에 연관된 인덱스나 BLOB 또는 값 파일들까지 함께 복사해야 하는 경우 일일이 파일을 복사하는 것은 지겨울 뿐만 아니라 실수를 할 가능성이 높다. TBatchMove를 사용하면 파일 단위 복사의 단점을 극복할 수 있기는 하지만, 여러 개의 컴포넌트들을 다뤄야 한다는 문제도 있고, 파일 단위 복사보다 훨씬 느리다는 문제도 있다. TTable.BatchMove()는 TBatchMove 보다는 편리하지만, 그래도 느리다는 점은 마찬가지이다.

다행히도 BDE에는 사용하기도 쉽고, 속도도 빠른 테이블 복사 함수가 존재한다. 바로 DbiCopyTable()이라는 함수가 그것인데, 함수의 선언은 다음과 같다.

```
function DbiCopyTable (          { 한 테이블을 다른 테이블로 복사하는 함수 }
      hDb           : hDBIDb;    { 데이터베이스 핸들 }
      bOverWrite    : Bool;      { True이면 기존 파일을 덮어쓴다. }
      pszSrcTableName : PChar;   { 원본 테이블 이름 }
      pszSrcDriverType : PChar;  { 원본 드라이버 종류 }
      pszDestTableName : PChar   { 대상 테이블 이름 }
   ): DBIResult stdcall;
```

이 함수를 직접 호출하는 TTable 상속 컴포넌트는 없기 때문에 원본 테이블을 뜻하는 TTable형의 인자와 대상 테이블 이름을 뜻하는 문자열 인자를 받아서 DbiCopyTable() 함수를 호출하는 새로운 함수를 만들어 보았다. 다음이 바로 그 코드이다.

```
procedure QuickCopyTable(T: TTable; DestTblName: string; Overwrite:
Boolean);
// TTable T를 DestTblName으로 지정한 테이블로 복사한다.
// Overwrite가 True이면 기존 파일을 덮어 쓴다.
var
  DBType: DBINAME;
  WasOpen: Boolean;
  NumCopied: Word;
begin
  WasOpen := T.Active;        // 테이블 활성화 상태를 보존한다.
  if not WasOpen then T.Open; // 테이블이 열려 있도록 한다.
  // 드라이버 종류를 뜻하는 문자열을 얻는다.
  Check(DbiGetProp(hDBIObj(T.Handle), drvDRIVERTYPE, @DBType,
    SizeOf(DBINAME), NumCopied));
  // 테이블을 복사한다.
  Check(DbiCopyTable(T.DBHandle, Overwrite, PChar(T.TableName), DBType,
    PChar(DestTblName)));
  T.Active := WasOpen;        // 활성화 상태를 복구한다.
end;
```



지역 데이터베이스(파라독스, dBase, 액세스, 폭스프로 등)의 경우, 위의 함수는 색인이나 BLOB 파일 같은 테이블에 관련된 모든 파일들도 대상 테이블에 복사한다. SQL 서버 데이터베이스에 있는 테이블의 경우 위의 함수는 테이블만 복사하며, 테이블에 필요한 색인이나 기타 요소들은 복사하지 않는다.

파라독스 세션의 사용자 목록 얻기

파라독스 테이블을 사용하는 경우, 테이블을 현재 사용하고 있는 사용자가 누구인지 알고 싶을 때가 있을 것이다. 현재 사용자를 알아낼 때에는 BDE API 함수 DbiOpenUserList()를 이용한다. 이 함수는 현재 세션의 사용자 목록에 대한 BDE 커서를 제공한다. 다음은 DbiOpenUserList() 함수로 사용자 목록을 얻는 프로시저의 코드이다.

```
procedure GetPDoxUsersForSession(Sess: TSession; UserList: TStrings);
// UserList를 비우고 Sess로 지정된 세션의 네트웍 파일을 공유하는 모든
// 사용자들을  UserList에 추가한다.
// Sess가 nil이면 기본 네트웍 파일을 이용해서 사용자들을 가려낸다.
var
  WasActive: Boolean;
  SessHand: hDBISes;
  ListCur: hDBICur;
  User: UserDesc;
begin
  if UserList = nil then Exit;
  WasActive := False;
  UserList.Clear;
  if Assigned(Sess) then
  begin
    WasActive := Sess.Active;
    if not WasActive then Sess.Open;
    Check(DbiStartSession(Nil, SessHand, PChar(Sess.NetFileDir)));
  end
  else
    Check(DbiStartSession(Nil, SessHand, Nil));
  try
    Check(DbiOpenUserList(ListCur));
       try
      while DbiGetNextRecord(ListCur, dbiNOLOCK, @User, Nil) = DBIERR_NONE
               do UserList.Add(User.szUserName);
       finally
           DbiCloseCursor(ListCur);  // "사용자 목록 테이블" 커서를 닫는다.
  finally
    DbiCloseSession(SessHand);
    if Assigned(Sess) then Sess.Active := WasActive;
  end;
end;
```

DbiOpenUserList() 함수에서 재미있는 점은 이 함수가 다른 모든 BDE 테이블 커서와 동일한 방식으로 조작할 수 있는 사용자 목록 테이블 커서를 생성한다는 점이다. 그런 커서가 존재하기 때문에 DbiGetNextRecord()를 반복적으로 호출하면 사용자 목록

테이블의 모든 사용자 레코드들을 하나씩 점검할 수 있다. 사용자 목록 테이블의 레코드 버퍼는 USerDesc 레코드(구조체)의 형식을 따르는데, USerDesc 레코드는 BDE 유닛에 다음과 같이 정의되어 있다.

```
type
  pUSERDesc = ^USERDesc;
  USERDesc = packed record                { 사용자 정보}
    szUserName      : DBIUSERNAME;        { 사용자 이름 }
    iNetSession     : Word;               { 네트웍 수준의 세션 번호 }
    iProductClass   : Word;               { 사용자의 제품 등급 }
    szSerialNum     : packed array [0..21] of Char; { 일련 번호 }
  end;
```

DbiGetNextRecord()를 호출하면 User라는 UserDesc 레코드에 다음 사용자의 정보가 채워진다. User 레코드를 채운 다음에는 User 레코드의 szUserName 필드값을 UserList라는 TStrings에 추가한다.



GetPDoxUsersForSession()의 try..finally 블록들을 주목하기 바란다. 이 블록들을 통해서 세션과 커서에 관련된 자원들을 확실하게 해제한다.

데이터 인식 VCL 컨트롤 작성

21장, "델파이 사용자 컴포넌트 만들기"와 22장, "고급 컴포넌트 기법"에서 컴포넌트 작성 기법이나 방법론에 대해서 자세히 설명했다. 그러나 21장과 22장에서는 데이터 인식(data-aware) 컨트롤의 작성에 대해서는 다루지 않았다. 데이터 인식 컨트롤의 작성은 일반 VCL 컨트롤의 작성과 크게 다르지 않지만, 그래도 몇 가지 기억해 둘 사항들이 있다. 다음은 데이터 인식 컴포넌트의 네 가지 주요 특징들을 정리한 것이다.

- 데이터 인식 컨트롤은 내부에 데이터 연결 객체를 포함한다. 데이터 연결 객체란, TDataLink의 상속 객체로 TDataSource와의 통신을 제어하는 역할을 담당한다. 데이터셋의 단일 필드에 연결하는 데이터 인식 컨트롤의 경우, 보통 TFieldDataLink라는 TDataLink 상속 객체를 사용한다. 데이터 인식 컨트롤은 필드나 레코드가 변경되면 발생하는 데이터 연결 객체의 OnDataChange 이벤트를 처리해야 한다.
- 데이터 인식 컨트롤은 CM_GETDATALINK 메시지를 처리해야 한다. 일반적으로, 데이터 인식 컨트롤이 이 메시지를 받으면 메시지의 Result 필드에 자신의 데이터 연결 객체를 설정한다.
- 데이터 인식 컨트롤은 TDataSource형의 속성을 제공해야 한다. 데이터 인식 컨트롤과 데이터셋을 연결하려면 중간에 TDataSource 컴포넌트가 존재해야 하는데, 그 컴포넌트를 지정하기 위한 속성이 필요한 것이다. 속성의 이름으로는 DataSource를 사용하는 것이 일반적인 관례이다. 단일 필드에 연결하는 컨트롤이라면 자신이 연결할 필드 이름을 뜻하는 문자열 속성도 제공해야 한다. 보통 그런 속성의 이름으로는 DataField를 사용한다.
- 데이터 인식 컨트롤은 TComponent의 Notification() 메소드를 재정의(override)해야 한다. 컨트롤에 연결된 TDataSource 컴포넌트가 폼에서 삭제되는 경우를 처리하기 위한 것이다.

WAV를 재생하는 데이터 인식 컨트롤

그럼 간단한 데이터 인식 컨트롤 예제를 살펴보자. 목록 29.4는 DBSound.pas 유닛의 소스 코드이다. 이 유닛은 데이터셋의 BLOB 필드로부터 얻은 WAV 데이터를 재생하는 TDBWavPlayer 컴포넌트를 정의한다.

목록 29.4 DBSound.pas 유닛의 소스 코드

```
unit DBSound;

interface

uses Windows, Messages, Classes, SysUtils, Controls, Buttons, DB,
 DBTables, DbCtrls;

type
  EDBWavError = class(Exception);
```

```
  TDBWavPlayer = class(TSpeedButton)
  private
    FAutoPlay: Boolean;
    FDataLink: TFieldDataLink;
    FDataStream: TMemoryStream;
    FExceptOnError: Boolean;
    procedure DataChange(Sender: TObject);
    function GetDataField: string;
    function GetDataSource: TDataSource;
    function GetField: TField;
    procedure SetDataField(const Value: string);
    procedure SetDataSource(Value: TDataSource);
    procedure CMGetDataLink(var Message: TMessage); message
          CM_GETDATALINK;
    procedure CreateDataStream;
    procedure PlaySound;
    procedure Notification(AComponent: TComponent; Operation: TOperation);
          override;
 protected
     procedure Click; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Click; override;
    property Field: TField read GetField;
  published
    property AutoPlay: Boolean read FAutoPlay write FAutoPlay default
          False;
    property ExceptOnError: Boolean read FExceptOnError write
          FExceptOnError;
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write
          SetDataSource;
  end;

implementation

uses MMSystem;

const
```

```
  // 에러 문자열들
  SNotBlobField = '필드 "%s" 는 blob 필드가 아님';
  SPlaySoundErr = '연주 시도 에러';

constructor TDBWavPlayer.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);            // 기본 작업을 수행
  FDataLink := TFieldDataLink.Create;   // 필드 데이터 연결 객체를 생성
  FDataLink.OnDataChange := DataChange; // 데이터 변경 통지를 설정
  FDataStream := TMemoryStream.Create;  // 작업용 메모리 스트림을 생성
end;

destructor TDBWavPlayer.Destroy;
begin
  FDataStream.Free;
  FDataLink.Free;
  FDataLink := Nil;
  inherited Destroy;
end;

procedure TDBWavPlayer.Click;
begin
  inherited Click; // 기본 작업을 수행
  PlaySound;       // 음성을 재생
end;

procedure TDBWavPlayer.CreateDataStream;
// BLOB 필드의 WAV 파일로부터 메모리 스트림을 생성
var
  BS: TBlobStream;
begin
  // BLOB 필드인지 확인
  if not (Field is TBlobField) then
    raise EDBWavError.CreateFmt(SNotBlobField, [DataField]);
  // BLOB 스트림을 생성
  BS := TBlobStream.Create(TBlobField(Field), bmRead);
  try
    // BLOB 스트림을 메모리 스트림에 복사
    FDataStream.SetSize(BS.Size);
    FDataStream.CopyFrom(BS, BS.Size);
```

```
  finally
    BS.Free;  // BLOB 스트림을 해제
  end;
end;

procedure TDBWavPlayer.PlaySound;
// 메모리 스트림에 있는 WAV 데이터를 재생
begin
  // 데이터셋과 필드에 제대로 연결되어 있는지 확인
  if (DataSource <> nil) and (DataField <> '') then
  begin
    // 데이터 스트림이 생성되어 있는지 확인
    if FDataStream.Size = 0 then CreateDataStream;
    // 메모리 스트림의 WAV 데이터를 재생. 에러가 있으면 예외를 발생시킨다.
    if (not MMSystem.PlaySound(FDataStream.Memory, 0, SND_ASYNC or
          SND_MEMORY))
      and FExceptOnError then
      raise EDBWavError.Create(SPlaySoundErr);
  end;
end;

procedure TDBWavPlayer.DataChange(Sender: TObject);
// FFieldDataLink의 OnChange 이벤트 처리 메소드
begin
  // 이전의 WAV 파일에 할당된 메모리를 해제
  with FDataStream do if Size <> 0 then SetSize(0);
  // 자동 재생으로 되어 있으면 즉시 음성을 재생한다.
  if FAutoPlay then PlaySound;
end;

procedure TDBWavPlayer.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  // 폼에서 TDatasource가 삭제된 경우의 처리
  if (Operation = opRemove) and (FDataLink <> nil) and
    (AComponent = DataSource) then DataSource := nil;
end;

function TDBWavPlayer.GetDataSource: TDataSource;
```

```
begin
  Result := FDataLink.DataSource;
end;

procedure TDBWavPlayer.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
  if Value <> nil then Value.FreeNotification(Self);
end;

function TDBWavPlayer.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TDBWavPlayer.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

function TDBWavPlayer.GetField: TField;
begin
  Result := FDataLink.Field;
end;

procedure TDBWavPlayer.CMGetDataLink(var Message: TMessage);
begin
  Message.Result := Integer(FDataLink);
end;

end.
```

이 컴포넌트는 TSpeedButton 컴포넌트를 상속한 것으로 사용자가 버튼을 클릭하면 데이터베이스의 BLOB 필드에 있는 WAV 데이터를 재생한다. AutoPlay 속성이 True로 되어 있으면 사용자가 테이블에서 현재 레코드를 이동할 때마다 해당 레코드의 WAV 데이터가 재생된다. AutoPlay 속성을 True로 하는 경우 버튼은 필요가 없으므로 컴포넌트의 Visible 속성을 False로 해서 버튼이 폼에 나타나지 않도록 하는 것이 합리적이다.

FDataLink.OnChange 이벤트의 처리 메소드인 DataChange()에서는 TBlobStream을 이용해서 BLOB 필드의 데이터를 뽑아낸 다음, BLOB 스트림을 메모리 스트림인 FDataStream에 복사한다. 메모리 스트림에 WAV 데이터가 들어간 후에는 언제라도 Win32 API 함수인 PlaySound()를 이용해서 WAV를 재생할 수 있다.

TDataSet의 확장

VCL 데이터베이스 프레임웍의 일원인 TDataSet은 BDE 이외의 데이터 원본을 다룰 수 있는 능력을 가지고 있다.

델파이 3, 4 시절의 TDataSet

델파이 3에서 새로운 구조가 도입되었는데, 새 구조의 핵심 개념은 TDataSet 클래스를 추상 클래스로 만들고 모든 VCL 데이터셋 클래스의 기반 클래스로 사용한다는 것이다. 그리고 BDE 중심의 데이터셋 기능은 새로 도입된 TBDEDataSet 클래스가 담당하게 되었다. 그림 29.2는 델파이 3과 4에서 사용했던 BDE 중심의 데이터셋 구조이다.

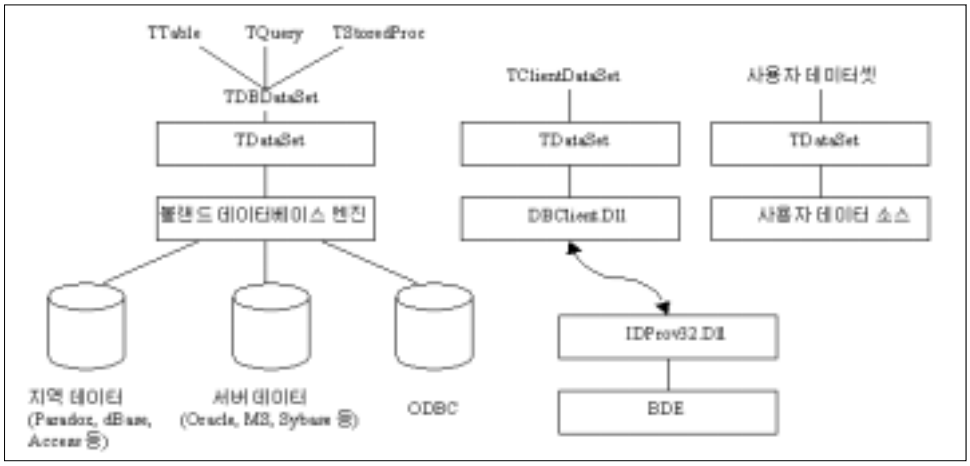


그림 29.2
델파이 3과 4의 VCL 데이터셋 구조

TDataSet이 BDE에만 종속되는 것이 아니기 때문에 BDE 이외의 데이터 원본을 다루기 위한 데이터셋 컴포넌트를 만든다는 것은 TDataSet의 상속 컴포넌트를 만든다는 것과 동일한 의미가 되었다.

델파이 5의 경우

델파이 3, 4의 혁신적인 BDE 개선 구조에도 불구하고, 여전히 산재하는 문제는 OLEDB, ADO와의 연계 문제였다. 볼랜드 자체 데이터베이스 서버인 인터베이스 서버 조차 BDE를 통해야만 접속이 가능했었다. 그러나 이러한 데이터베이스들에 대해 좀더 융통성을 주고자 델파이 5에서는 아예 BDE가 필요없는 컴포넌트를 직접 제공하게 되었다. 그림 29.3은 새 구조를 도식화한 그림이다.

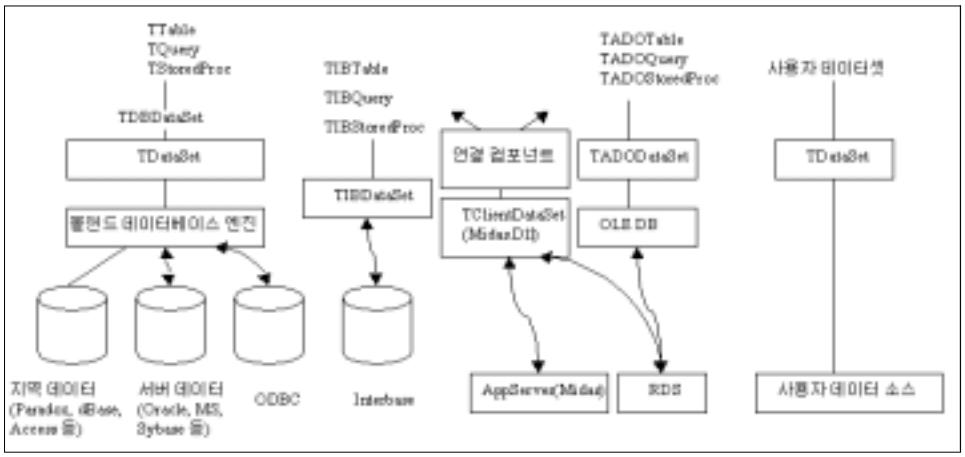


그림 29.3
델파이 5와 VCL 데이터셋 구조

VCL의 상속 구조가 프로그래밍을 얼마나 편하게 만드는지를 이해하고 있는 독자라면 델파이 3과 4의 VCL 데이터셋 구조에 비해 델파이 5의 데이터셋 구조가 얼마나 다양하고 복잡하면서도 융통성 있고 편리한 것인지를 실감할 수 있을 것이다. 가장 중추적인 역할을 하는 것은 물론 TClientDataSet(Midas.Dll) 부분이다. 이 부분은 MIDAS 뿐만 아니라 인터베이스, ADO, 순수 클라이언트/서버까지도 모두 통합할 수 있는 분산 데이터셋 객체이다.

Tip

델파이의 예제 중에는 TDataSet 상속 컴포넌트 작성에 관련된 것이 두 개 있는데, 하나는 매우 간단하고 하나는 매우 복잡하다. 간단한 예제는 \Delphi 5\Demo\Db\TextData 디렉토리의 TextData 유닛에 있는 TTextDataSet 클래스이다. 이 예제는 TStringList를 단일 필드 데이터셋으로 캡슐화한 것이다. 복잡한 예제란 VCL 소스 디렉토리의 DbTables 유닛에 있는 TBDEDataSet 클래스 자체를 말한다. 이 클래스는 VCL의 데이터셋 구조를 BDE에 연결시키는 역할을 한다.

TDataSet 상속 컴포넌트의 작성

대부분의 데이터셋 컴포넌트는 TTextDataSet 보다는 복잡하고, TBDEDataSet 보다는 간단할 것이다. 그럼 데이터셋 컴포넌트 작성을 설명하기 위해서 오브젝트 파스칼의 레코드 파일(file of로 정의하는 정형 파일. 이에 대해서는 12장, "파일 다루기"를 참고하기 바란다)을 데이터 원본으로 취급하는 TDataSet의 상속 컴포넌트를 예제로 들겠다. 다음은 이번 예제에서 사용할 레코드와 파일 형태이다.

```
type

  // Name 필드에 사용할 char 배열
  TNameStr = array[0..31] of char;

  // "테이블" 구조를 뜻하는 레코드
  PDDGData = ^TDDGData;
  TDDGData = record
    Name: TNameStr;
    Height: Double;
    ShoeSize: Integer;
  end;

  // "테이블"의 데이터를 담을 오브젝트 파스칼의 레코드 파일
  TDDGDataFile = file of TDDGData;
```

오브젝트 파스칼의 레코드 파일은 정보를 그냥 저장하기에는 편하고 효율적이지만, 파일 중간에 레코드를 삽입하거나 레코드를 삭제하기에는 불편하다. 그래서 두 종류의 파일을 이용해서 하나의 "테이블" 정보를 관리하도록 한다. 첫 번째 파일은 실제 데이터를 저장하는 데이터 파일로, 이 파일의 형태는 오브젝트 파스칼의 레코드 파일이다. 두 번째 파일은 색인 파일로, 이 파일의 형태는 데이터 파일의 각 레코드 위치를 뜻하는 정수들의 목록이다. 즉, 데이터 파일의 물리적 레코드 위치와 데이터셋에서의 레코드 위치는 동일하지 않을 수도 있다는 뜻이 된다. 데이터셋에서의 레코드 위치는 색인 파일에 저장된 레코드 순서에 따른다. 색인 파일의 첫 번째 정수는 데이터 파일의 첫 번째 레코드의 실제 위치를 뜻하며, 색인 파일의 두 번째 정수는 데이터 파일의 두 번째 레코드의 실제 위치를 뜻하는 식으로 진행된다.

이번 예제에서 설명할 TDataSet 상속 컴포넌트의 이름은 TDDGDataSet이며, 이 컴포넌트는 위에서 말한 레코드 파일 구조를 사용한다.

TDataSet의 추상 메소드들

TDataSet은 추상(abstract) 클래스이므로 그 자체로는 아무런 기능도 하지 못한다. TDataSet에 생명을 불어넣기 위해서는 TDataSet을 상속한 클래스가 TDataSet의 추상 메소드들을 재정의(override)해야 하는 것이다. 적어도 23개의 TDataSet 추상 메소드들은 필수적으로 재정의해야 하며, 그 이외에도 여러 가지 추상 메소드들이 있다. TDataSet의 추상 메소드들을 기능상으로 분류하자면 레코드 버퍼 메소드들, 레코드 이동 메소드들, 북마크 메소드들, 편집 메소드들, 기타 메소드들, 그리고 추가적인 메소드들로 나뉘어진다.

다음 목록은 Db.pas 유닛에 있는 TDataSet의 정의에서 필요한 부분만 간추린 것이다. 이해를 돕기 위해서 각각의 메소드들을 분야별로 모아 두었다.

```
TDataSet = class(TComponent)
    {...}
protected

{ 레코드 버퍼 메소드들 }
  function AllocRecordBuffer: PChar; virtual; abstract;
  procedure FreeRecordBuffer(var Buffer: PChar); virtual; abstract;
  procedure InternalInitRecord(Buffer: PChar); virtual; abstract;
  function GetRecord(Buffer: PChar; GetMode: TGetMode; DoCheck:
         Boolean): TGetResult; virtual; abstract;
```

```
    function GetRecordSize: Word; virtual; abstract;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; virtual;
           override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); virtual;
           abstract;
  { 북마크 메소드들 }
    function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; virtual; abstract;
    procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); virtual;
           override;
    procedure GetBookmarkData(Buffer: PChar; Data: Pointer); virtual;
             abstract;
    procedure SetBookmarkData(Buffer: PChar; Data: Pointer); virtual;
             abstract;
    procedure InternalGotoBookmark(Bookmark: Pointer); virtual; abstract;
    procedure InternalSetToRecord(Buffer: PChar); virtual; abstract;
  { 레코드 이동 메소드들 }
    procedure InternalFirst; virtual; abstract;
    procedure InternalLast; virtual; abstract;
  { 편집 메소드들 }
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean);
           virtual; abstract;
    procedure InternalDelete; virtual; abstract;
    procedure InternalPost; virtual; abstract;
  { 기타 메소드들 }
    procedure InternalClose; virtual; abstract;
    procedure InternalHandleException; virtual; abstract;
    procedure InternalInitFieldDefs; virtual; abstract;
    procedure InternalOpen; virtual; abstract;
    function IsCursorOpen: Boolean; virtual; abstract;
  { 추가적인 메소드들 }
    function GetRecordCount: Integer; virtual;
    function GetRecNo: Integer; virtual;
    procedure SetRecNo(Value: Integer); virtual;
  { ... }
  end;
```

레코드 버퍼 메소드들

TDataSet 상속 컴포넌트를 만들려면 레코드 버퍼를 다루는 여러 메소드들을 재정의(override)해야 한다. 레코드 버퍼의 복잡한 내부적인 구현은 VCL이 대부분 처리해 주기 때문에 버퍼에 무엇을 저장할 것인지 아니면 여러 버퍼 사이에서 데이터를 어떻게 이동할 것인지 등만 처리하면 된다. TDataSet의 상속 컴포넌트는 반드시 북마크를 구현해야 하므로 이번 예제에서는 레코드 데이터와 함께 북마크 정보도 레코드 버퍼에 저장한다. 북마크 정보에 이용할 레코드(구조체)의 정의는 다음과 같다.

```
type
  // TDataSet 북마크를 지원하기 위한 북마크 정보 구조체
  PDDGBookmarkInfo = ^TDDGBookmarkInfo;
  TDDGBookmarkInfo = record
    BookmarkData: Integer;
    BookmarkFlag: TBookmarkFlag;
  end;
```

BookmarkData 필드에는 데이터 파일의 특정 레코드 위치가 저장된다. BookmarkFlag 필드는 버퍼가 유효한 북마크를 담고 있는지를 판단하기 위한 것이다. 데이터셋이 BOF나 EOF를 넘어가면 이 필드에는 특별한 값이 들어간다.



이번 예제에 나온 북마크와 레코드 버퍼의 구현은 레코드 파일(file of record)을 사용하는 이번 예제에 한정된 것이다. 다른 형태의 데이터를 다루는 TDataSet 상속 컴포넌트를 만드는 경우에는 다른 방식으로 북마크와 레코드 버퍼를 구현해야 할 것이다. 예를 들어서, TDataSet 컴포넌트로 캡슐화하려면 데이터 원본 자체에 북마크 기능이 포함되어 있을 수도 있다.

레코드 버퍼 관련 메소드들을 살펴보기 전에, 우선 TDDGDataSet 클래스의 생성자부터 살펴보자.

```
constructor TDDGDataSet.Create(AOwner: TComponent);
begin
  FIndexList := TIndexList.Create;
  FRecordSize := SizeOf(TDDGData);
  FBufferSize := FRecordSize + SizeOf(TDDGBookmarkInfo);
  inherited Create(AOwner);
end;
```

이 생성자는 세 가지 일을 한다. 우선, TIndexList 객체를 생성한다. 이 객체는 데이터셋의 레코드 순서를 관리할 색인 파일을 다루기 위한 색인 목록이다. 그리고 FRecordSize와 FBufferSize라는 저장 필드를 초기화한다. FRecordSize에는 데이터 레코드의 크기를 넣고, FBufferSize에는 데이터 레코드의 크기와 북마크 정보 레코드의 크기를 합한 것, 즉 레코드 버퍼의 전체 크기를 넣는다. 마지막으로, TDataSet의 Create()를 호출해서 기본적인 초기화 작업을 수행한다.

다음은 TDDGDataSet이 재정의하는 TDataSet의 레코드 버퍼 관련 메소드들이다. GetFieldData()를 제외한 모든 메소드들은 TDataSet에서 추상(abstract) 클래스로 선언되어 있다.

```
    function AllocRecordBuffer: PChar; override;
    procedure FreeRecordBuffer(var Buffer: PChar); override;
    procedure InternalInitRecord(Buffer: PChar); override;
    function GetRecord(Buffer: PChar; GetMode: TGetMode;
      DoCheck: Boolean): TGetResult; override;
    function GetRecordSize: Word; override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
```

AllocRecordBuffer()

AllocRecordBuffer() 메소드는 하나의 레코드 버퍼에 대한 메모리를 할당하는 작업을 해야 한다. 이번 예제에서는 AllocMem() 함수를 이용해서 레코드 데이터와 북마크 정보를 모두 담을 수 있는 크기의 메모리를 할당한다.

```
function TDDGDataSet.AllocRecordBuffer: PChar;
begin
  Result := AllocMem(FBufferSize);
end;
```

FreeRecordBuffer()

FreeRecordBuffer()는 AllocRecordBuffer() 메소드에서 할당한 메모리를 해제하는 역할을 해야 한다. 이번 예제에서는 FreeMem()을 이용해서 메모리를 해제한다.

```
procedure TDDGDataSet.FreeRecordBuffer(var Buffer: PChar);
begin
  FreeMem(Buffer);
end;
```

InternalInitRecord()

InternalInitRecord() 메소드는 레코드 버퍼를 초기화하는 역할을 해야 한다. 이 메소드에서 기본 필드값을 설정한다든지 레코드 버퍼 데이터의 특별한 초기화 작업을 수행하면 된다. 이번 예제에서는 그냥 레코드 버퍼를 0으로 채운다.

```
procedure TDDGDataSet.InternalInitRecord(Buffer: PChar);
begin
  FillChar(Buffer^, FBufferSize, 0);
end;
```

GetRecord()

GetRecord() 메소드의 기본적인 기능은 데이터셋의 이전, 현재, 다음 레코드에 대한 레코드 데이터를 돌려주는 것이다. 이 메소드의 리턴값은 TGetResult형인데, 이 열거형은 Db 유닛에 다음과 같이 정의되어 있다.

```
  TGetResult = (grOK, grBOF, grEOF, grError);
```

각 열거값의 의미는 한 눈에 알 수 있을 것이다. grOK는 성공을 뜻하며, grBOF와 grEOF는 데이터셋이 각각 BOF와 EOF 위치를 넘어섰다는 뜻이다. 그리고 grError는 에러가 발생했다는 뜻이다.

이번 예제에서 구현한 GetRecord()는 다음과 같다.

```
function TDDGDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var
  IndexPos: Integer;
begin
 if FIndexList.Count < 1 then
    Result := grEOF
  else begin
    Result := grOk;
    case GetMode of
      gmPrior:
        if FRecordPos <= 0 then
        begin
          Result := grBOF;
          FRecordPos := -1;
        end
        else
          Dec(FRecordPos);
      gmCurrent:
        if (FRecordPos < 0) or (FRecordPos >= RecordCount) then
           Result := grError;
      gmNext:
        if FRecordPos >= RecordCount-1 then
          Result := grEOF
        else
          Inc(FRecordPos);
    end;
    if Result = grOk then
    begin
      IndexPos := Integer(FIndexList[FRecordPos]);
      Seek(FDataFile, IndexPos);
      BlockRead(FDataFile, PDDGData(Buffer)^, 1);
      with PDDGBookmarkInfo(Buffer + FRecordSize)^ do
      begin
        BookmarkData := FRecordPos;
```

```
        BookmarkFlag := bfCurrent;
      end;
    end
    else if (Result = grError) and DoCheck then
      DatabaseError('No records');
  end;
end;
```

FRecordPos 필드는 데이터셋 안에서 현재 레코드의 위치를 뜻한다. GetMode 인자에 따라(이전, 현재, 다음) 이 FRecordPos 필드의 값이 증가되거나 감소된다. FRecordPos에 유효한 레코드 번호가 들어 있는 경우, FRecordPos의 값은 FIndexList의 배열 색인 값으로 사용된다. 색인 목록인 FIndexList에는 데이터 파일에 있는 레코드의 위치가 들어 있다. 이 메소드는 데이터 파일에서 FIndexList가 가리키는 레코드를 읽은 후 Buffer 인자에 그 레코드의 값을 넣는다.

그리고 예외에 관련된 부분이 있다. DoCheck 인자가 True이며, 리턴값으로 grError를 돌려줘야 하는 상황이면 이 메소드 안에서 예외를 발생시킨다.

GetRecordSize()

GetRecordSize() 메소드는 레코드 버퍼에 담긴 레코드 데이터 부분의 크기를 바이트 단위로 돌려줘야 한다. 레코드 버퍼 전체의 크기가 아니라 데이터 부분만의 크기를 돌려준다는 점을 명시하기 바란다. 이번 예제에서는 FRecordSize 필드의 값을 돌려주도록 재정의했다.

```
function TDDGDataSet.GetRecordSize: Word;
begin
  Result := FRecordSize;
end;
```

GetFieldData()

GetFieldData() 메소드는 현재 활성화된 레코드 버퍼(ActiveBuffer 속성이 가리키는 것)에서 필드 데이터를 복사해서 필드 버퍼(Buffer 인자)에 넣는 작업을 해야 한다. Move() 프로시저를 사용하면 간단하게 해결할 수 있다. 어떤 필드 데이터를 복사해야 하는지는 Field 인자의 Index나 Name 속성을 보고 판단한다. 그리고 ActiveBuffer

에는 레코드 데이터 전체가 들어 있기 때문에 ActiveBuffer에서 필드에 해당하는 부분만을 정확히 Buffer 인자에 복사하도록 해야 한다는 점을 주의하기 바란다.

```
function TDDGDataSet.GetFieldData(Field: TField; Buffer: Pointer): Boolean;
begin
  Result := True;
  case Field.Index of
    0:
      begin
        Move(ActiveBuffer^, Buffer^, Field.Size);
        Result := PChar(Buffer)^ <> #0;
      end;
    1: Move(PDDGData(ActiveBuffer)^.Height, Buffer^, Field.DataSize);
    2: Move(PDDGData(ActiveBuffer)^.ShoeSize, Buffer^, Field.DataSize);
  end;
end;
```

계산 필드(calculated field)나 필터 같은 기능을 지원해야 한다면 이 메소드와 SetFieldData() 메소드가 훨씬 더 복잡해질 것이다.

SetFieldData()

SetFieldData()가 해야 하는 일은 GetFieldData()의 반대에 해당한다. SetFieldData()는 필드 버퍼의 데이터를 활성화된 레코드 버퍼에 복사해야 한다. 다음은 이번 예제에서 재정의한 SetFieldData()인데, GetFieldData()를 뒤집어 놓은 것이라고 할 수 있다.

```
procedure TDDGDataSet.SetFieldData(Field: TField; Buffer: Pointer);
begin
  case Field.Index of
    0: Move(Buffer^, ActiveBuffer^, Field.Size);
    1: Move(Buffer^, PDDGData(ActiveBuffer)^.Height, Field.DataSize);
    2: Move(Buffer^, PDDGData(ActiveBuffer)^.ShoeSize, Field.DataSize);
  end;
  DataEvent(deFieldChange, Longint(Field));
end;
```

데이터를 복사한 다음에는 DataEvent() 메소드를 호출해서 필드가 변경되었음을 알린다. 결과적으로는 필드에 대한 OnChange 이벤트가 발생한다.

북마크 관련 메소드들

TDataSet의 상속 컴포넌트는 북마크 기능을 지원해야 한다. 다음은 이번 예제에서 북마크를 지원하기 위해 재정의하는 TDataSet의 추상 메소드들이다.

```
function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
override;
procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure InternalGotoBookmark(Bookmark: Pointer); override;
procedure InternalSetToRecord(Buffer: PChar); override;
```

이번 예제에서는 위의 메소드들이 주로 레코드 버퍼의 끝 부분에 붙어 있는 북마크 정보를 다루는 작업을 한다.

GetBookmarkFlag()와 SetBookmarkFlag()

북마크 플래그란, 특정 레코드가 데이터셋의 처음 또는 마지막 위치에 있는지를 판단하기 위한 것이다. GetBookmarkFlag()와 SetBookmarkFlag()는 북마크 플래그를 다루는 메소드인데, TDDGDataSet에서는 레코드 버퍼에 있는 북마크 정보 레코드를 이용해서 북마크 플래그를 얻거나 설정한다.

```
function TDDGDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
  Result := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag;
end;


procedure TDDGDataSet.SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag := Value;
end;
```

GetBookmakrData()와 SetBookmakrData()

GetBookmarkData()와 SetBookmarkData()는 현재 레코드를 이동하지 않고 레코드의 북마크 데이터를 다루기 위한 것이다. 이번 예제에서는 앞에서와 마찬가지로 레코드 버퍼의 북마크 정보 레코드의 북마크 데이터를 읽거나 쓰도록 재정의했다.

```
procedure TDDGDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PInteger(Data)^ := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData;
end;

procedure TDDGDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData := PInteger(Data)^;
end;
```

InternalGotoBookmark()

InternalGotoBookmark() 메소드는 현재 레코드를 Bookmark 인자로 지정된 위치로 옮기는 작업을 해야 한다. 이번 예제 TDDGDataSet의 경우에는 북마크 데이터가 레코드 위치를 뜻하므로 다음과 같이 간단하게 재정의할 수 있다.

```
procedure TDDGDataSet.InternalGotoBookmark(Bookmark: Pointer);
begin
  FRecordPos := Integer(Bookmark);
end;
```

InternalSetToRecord()

InternalSetToRecord()는 InternalGotoBookmark()와 비슷하지만, 북마크값이 아니라 레코드 버퍼를 인자로 받는다. 이 메소드는 Buffer 인자로 지정된 레코드로 데이터셋의 레코드 위치를 이동시켜야 한다. TDDGDataSet의 경우, 레코드 버퍼에는 북마크 정보가 들어 있으며, 북마크 데이터가 레코드 위치를 뜻하므로 다음과 같이 간단하게 구현할 수 있다.

```
procedure TDDGDataSet.InternalSetToRecord(Buffer: PChar);
begin
  // 북마크 데이터는 레코드 위치를 뜻한다.
  FRecordPos := PDDGBookmarkInfo(Buffer + FRecordSize)^.Bookmarkdata;
end;
```

레코드 이동 메소드들

데이터셋을 처음 레코드나 마지막 레코드로 이동시키는 기능을 구현하려면 다음과 같이 TDataSet의 추상 메소드들을 재정의해야 한다.

```
procedure InternalFirst; override;
procedure InternalLast; override;
```

TDDGDataSet에서 이 메소드들을 구현하는 것은 그리 어렵지 않다. InternalFirst()에서는 FRecordPos의 값을 -1(BOF를 넘어섰다는 뜻)로 설정하면 되고, InternalLast()에서는 FRecordPos의 값을 레코드 전체 개수로 설정하면 된다. 레코드 색인은 0부터 시작하므로 레코드 개수는 마지막 색인 번호보다 1 큰 값(이 값은 EOF를 넘어섰다는 뜻이다)이 된다.

```
procedure TDDGDataSet.InternalFirst;
begin
  FRecordPos := -1;
end;


procedure TDDGDataSet.InternalLast;
begin
  FRecordPos := FIndexList.Count;
end;
```

편집 메소드들

레코드를 편집, 추가, 삽입, 삭제하려면 다음과 같이 TDataSet의 추상 메소드들을 재정의해야 한다.

```
procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
procedure InternalDelete; override;
procedure InternalPost; override;
```

InternalAddRecord()

InternalAddRecord()는 데이터셋에 레코드를 추가하거나 삽입하는 작업을 수행해야 한다. Buffer 인자는 데이터셋에 추가될 레코드 버퍼를 가리키며, Append 인자는 추가인지 삽입인지를 뜻하는 것이다. Append가 True이면 데이터셋의 끝에 레코드가 추가되며, False이면 현재 레코드 위치에 레코드가 삽입된다.

TDDGDataSet의 경우, 이 메소드는 데이터 파일의 끝을 찾아서 거기에 레코드 데이터를 기록하고, 색인 파일에는 Append 인자에 따라 적절히 색인 번호를 추가 또는 삽입한다.

```
procedure TDDGDataSet.InternalAddRecord(Buffer: Pointer; Append: Boolean);
var
  RecPos: Integer;
begin
  Seek(FDataFile, FileSize(FDataFile));
  BlockWrite(FDataFile, PDDGData(Buffer)^, 1);
  if Append then
  begin

    FIndexList.Add(Pointer(FileSize(FDataFile) - 1));
    InternalLast;
  end
  else begin
    if FRecordPos = -1 then RecPos := 0
    else RecPos := FRecordPos;
    FIndexList.Insert(RecPos, Pointer(FileSize(FDataFile) - 1));
  end;
  FIndexList.SaveToFile(FIdxName);
end;
```

InternalDelete()

InternalDelete() 메소드는 데이터셋에서 현재 레코드를 삭제하는 작업을 해야 한다. TDDGDataSet의 경우, 데이터 파일의 중간에서 레코드를 직접 삭제하기가 좀 어렵기 때문에 색인 목록에서만 현재 레코드를 삭제한다. 따라서, 데이터 파일 안에는 삭제된 레코드가 그대로 남아 있게 된다.

```
procedure TDDGDataSet.InternalDelete;
begin
  FIndexList.Delete(FRecordPos);
  if FRecordPos >= FIndexList.Count then Dec(FRecordPos);
end;
```



위의 코드를 보면 알겠지만, 레코드를 삭제한다고 해서 데이터 파일의 크기가 줄어들지는 않는다(dBase 파일과 비슷하다). 따라서, TDDGDataSet이 좀더 실용적인 것이 되려면 파일에 있는 삭제된 레코드들을 실제로 제거하는 파일 패킹 메소드를 추가해야 하는데, 그 부분은 독자에게 숙제로 맡기겠다.

InternalPost()

InternalPost() 메소드는 TDataSet.Post()가 호출한다. 이 메소드는 활성화된 레코드 버퍼의 데이터를 데이터 원본(이번 예제의 경우 데이터 파일)에 실제로 기록하는 작업을 수행해야 한다. 다음은 TDDGDataSet이 재정의한 InternalPost()인데, InternalAddRecord()와 거의 비슷하다.

```
procedure TDDGDataSet.InternalPost;
var
  RecPos, InsPos: Integer;
begin
  if FRecordPos = -1 then
```

```
    RecPos := 0
  else begin
    if State = dsEdit then RecPos := Integer(FIndexList[FRecordPos])
    else RecPos := FileSize(FDataFile);
  end;

  Seek(FDataFile, RecPos);
  BlockWrite(FDataFile, PDDGData(ActiveBuffer)^, 1);
  if State <> dsEdit then
  begin
    if FRecordPos = -1 then InsPos := 0
    else InsPos := FRecordPos;
    FIndexList.Insert(InsPos, Pointer(RecPos));
  end;
  FIndexList.SaveToFile(FIdxName);
end;
```

기타 메소드들

반드시 재정의해야 하는 TDataSet의 메소드들 중 특정 분야에 속하지 않는 것들을 기타 메소드라고 칭하겠다. 대부분 일반적인 관리 작업에 관련된 것들인데, 다음의 메소드들이 기타 메소드들이다.

```
procedure InternalClose; override;
procedure InternalHandleException; override;
procedure InternalInitFieldDefs; override;
procedure InternalOpen; override;
function IsCursorOpen: Boolean; override;
```

InternalClose()

InternalClose()는 TDataSet.Close()가 호출한다. 이 메소드에서는 InternalOpen()에서 할당된 자원이나 데이터셋을 사용하는 과정에서 할당된 자원들을 깨끗이 해제시켜야 한다. TDDGDataSet에서는 데이터 파일을 닫고, 색인 목록을 디스크에 있는 색인 파

일에 기록한다. 그리고 FRecordPos를 BOF에 해당하는 값으로 설정한 후 데이터 파일 레코드를 0으로 채운다.

```
procedure TDDGDataSet.InternalClose;
begin
  if TFileRec(FDataFile).Mode <> 0 then
    CloseFile(FDataFile);
  FIndexList.SaveToFile(FIdxName);
  FIndexList.Clear;
  if DefaultFields then
    DestroyFields;
  FRecordPos := -1;
  FillChar(FDataFile, SizeOf(FDataFile), 0);
end;
```

InternalHandleException()

컴포넌트 자신이 스트림으로부터 읽히거나 스트림에 기록하는 도중 예외가 발생하면 InternalHandleException()이 호출된다. 특별한 예외 처리를 하지 않는 한 다음과 같이 구현하는 것이 일반적이다.

```
procedure TDDGDataSet.InternalHandleException;
begin
  // 일반적으로는 이런 식으로 구현하면 된다.
  Application.HandleException(Self);
end;
```

InternalInitFieldDefs()

InternalInitFieldDefs() 메소드는 데이터셋에 들어 있는 필드를 정의해야 한다. 필드를 정의한다는 것은 TFieldDef 객체를 생성한다는 것이다. TFieldDef 객체 생성자의 Owner 인자에 TDataSet의 FieldDefs 속성을 넣고 나머지 인자들에 필드 정의 정보들을 넣어서 호출하면 된다. TDDGDataSet의 경우, 세 개의 TFieldDef 객체들을 생성한다.

```
procedure TDDGDataSet.InternalInitFieldDefs;
begin
  // 데이터 레코드를 구성하는 각 필드에 해당하는 TFieldDef 객체들을 생성
  FieldDefs.Clear;
  TFieldDef.Create(FieldDefs, 'Name', ftString, SizeOf(TNameStr), False, 1);
  TFieldDef.Create(FieldDefs, 'Height', ftFloat, 0, False, 2);
  TFieldDef.Create(FieldDefs, 'ShoeSize', ftInteger, 0, False, 3);
end;
```

InternalOpen()

InternalOpen() 메소드는 TDataSet.Open()이 호출한다. 이 메소드에서는 컴포넌트가 연결할 데이터 원본을 열고, 내부 저장 필드들이나 속성들을 초기화하고, 필요한 경우에는 필드 정의들을 생성해서 필드 정의들을 데이터에 연결하는 등의 작업을 수행해야 한다. TDDGDataSet이 경우에는 데이터 파일을 열고, 색인 파일에서 색인 목록을 읽어 들인 후 FRecordPost 저장 필드와 BookmarkSize 속성을 초기화하고, 필드 정의들을 생성하고 연결한다. 또한, 데이터 파일이나 색인 파일이 만들어져 있지 않은 경우에는 디스크에 새 파일들을 만들 것인지를 사용자에게 묻고, 응답에 따라 적절한 처리를 한다.

```
procedure TDDGDataSet.InternalOpen;
var
  HFile: THandle;
begin
  // 데이터 파일과 색인 파일이 존재하는지 확인
  FIdxName := ChangeFileExt(FTableName, feDDGIndex);
  if not (FileExists(FTableName) and FileExists(FIdxName)) then
  begin
    if MessageDlg('Table or index file not found.  Create new table?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    begin
      HFile := FileCreate(FTableName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating table file');
      FileClose(HFile);
```

```
      HFile := FileCreate(FIdxName);

      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating index file');
      FileClose(HFile);
    end
    else
      DatabaseError('Could not open table');
  end;
  // 데이터 파일을 연다.
  FileMode := fmShareDenyNone or fmOpenReadWrite;
  AssignFile(FDataFile, FTableName);
  Reset(FDataFile);
  try
    FIndexList.LoadFromFile(FIdxName); // 색인 파일에서 색인 목록을 읽는다.
    FRecordPos := -1;                 // 레코드 위치를 BOF 이전 위치로 한다.
    BookmarkSize := SizeOf(Integer);   // 북마크 크기를 설정한다.
    InternalInitFieldDefs;             // FieldDef 객체들을 초기화한다.
      // 기본 필드 객체들이 만들어져 있지 않으면 TField 객체들을 생성한다.
    if DefaultFields then CreateFields;
    BindFields(True);                 // FieldDefs를 실제 데이터에 연결한다.
  except
    CloseFile(FDataFile);
    FillChar(FDataFile, SizeOf(FDataFile), 0);
    raise;
  end;
end;
```



InternalOpen()에서 할당한 자원들은 반드시 InternalClose()에서 해제시켜 줘야 한다.

IsCursorOpen()

데이터셋이 열리는 도중, 즉 데이터셋이 아직 활성화되지 않은 상태에서 데이터를 조작할 수 있는지 없는지를 알고 싶을 때 TDataSet은 내부적으로 IsCursorOpen()을 호출한다. TDDGDataSet에서는 데이터셋이 열리는 도중에 데이터를 사용할 일이 없으므로 데이터 파일이 열려 있는 경우에만 True를 돌려준다.

```
function TDDGDataSet.IsCursorOpen: Boolean;
begin
  // 데이터 파일이 열려 있는 경우에만 "커서"가 열린 것으로 한다.
  // FDataFile의 Mode 속성이 0이 아니면 파일이 열려 있는 것이다.
  Result := TFileRec(FDataFile).Mode <> 0;
end;
```



위의 메소드에서 레코드 파일(file of record)이나 비정형 파일(untyped file)을 TFileRec로 형변환시키는 부분을 주목하기 바란다. TFileRec로 형변환을 하면 파일에 대한 저수준 정보를 얻을 수 있다. 자세한 내용은 델파이 도움말을 참고하기 바란다.

레코드 번호에 관련된 추가적인 메소드들

TDBGrid는 데이터셋의 커서 위치를 인식하고 그에 맞게 스크롤을 구현하는 기능이 있다. 그러한 기능을 지원하려면 다음 세 메소드를 적절히 재정의해야 한다.

```
function GetRecordCount: Integer; override;
function GetRecNo: Integer; override;
procedure SetRecNo(Value: Integer); override;
```

TDDGDataSet에서는 이러한 메소드들을 구현하는 것이 별로 어렵지 않지만, 특정한 상황에서는 TDBGrid의 그러한 기능을 지원하는 것이 실용적이지 못하거나 아예 불가능할 수도 있다. 예를 들어서, 아주 큰 데이터를 다루는 경우 레코드 개수를 센다는 것 자체가 매우 비효율적이며, SQL 서버를 다루는 경우에는 레코드 개수라는 것 자체를 얻을 수 없을 수가 있다.

TDDGDataSet의 경우에는 별로 어렵지 않게 위의 메소드들을 구현할 수 있다. 다음과 같다.

```
function TDDGDataSet.GetRecordCount: Integer;
begin
  Result := FIndexList.Count;
end;

function TDDGDataSet.GetRecNo: Integer;
begin
  UpdateCursorPos;
  if (FRecordPos = -1) and (RecordCount > 0) then
    Result := 1
  else
    Result := FRecordPos + 1;
end;

procedure TDDGDataSet.SetRecNo(Value: Integer);
begin
  if (Value >= 0) and (Value <= FIndexList.Count-1) then
  begin
    FRecordPos := Value - 1;
    Resync([]);
  end;
end;
```

TDDGDataSet

목록 29.5는 TDDGDataSet 컴포넌트를 정의하는 DDG_DS.pas 유닛의 소스 코드이다.

목록 29.5 Coms\DDG_DS.pas 유닛의 소스 코드

```
unit DDG_DS;

interface

uses Windows, Db, Classes, DDG_Rec;

type

  // TDataSet 북마크를 지원하기 위한 북마크 정보 레코드(구조체)
  PDDGBookmarkInfo = ^TDDGBookmarkInfo;
  TDDGBookmarkInfo = record
    BookmarkData: Integer;
    BookmarkFlag: TBookmarkFlag;
  end;

  // 데이터 파일의 레코드 위치를 관리하기 위한 색인 목록
  TIndexList = class(TList)
  public
    procedure LoadFromFile(const FileName: string); virtual;
    procedure LoadFromStream(Stream: TStream); virtual;
    procedure SaveToFile(const FileName: string); virtual;
    procedure SaveToStream(Stream: TStream); virtual;
  end;

  // 레코드 파일 형태의 데이터셋을 위한 TDataset 상속 클래스
  TDDGDataSet = class(TDataSet)
  private
    function GetDataFileSize: Integer;
  public
    FDataFile: TDDGDataFile;
    FIdxName: string;
    FIndexList: TIndexList;
    FTableName: string;
```

```
  FRecordPos: Integer;
  FRecordSize: Integer;
  FBufferSize: Integer;
  procedure SetTableName(const Value: string);
protected
  { 반드시 재정의해야 하는 메소드들 }
  // 레코드 버퍼 관련 메소드들
  function AllocRecordBuffer: PChar; override;
  procedure FreeRecordBuffer(var Buffer: PChar); override;
  procedure InternalInitRecord(Buffer: PChar); override;
  function GetRecord(Buffer: PChar; GetMode: TGetMode;
    DoCheck: Boolean): TGetResult; override;
  function GetRecordSize: Word; override;
  procedure SetFieldData(Field: TField; Buffer: Pointer); override;
  // 북마크 관련 메소드들
  procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
  function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
  procedure InternalGotoBookmark(Bookmark: Pointer); override;
  procedure InternalSetToRecord(Buffer: PChar); override;
  procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
        override;
  procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
  // 레코드 이동 관련 메소드들
  procedure InternalFirst; override;
  procedure InternalLast; override;
  // 편집 관련 메소드들
  procedure InternalAddRecord(Buffer: Pointer; Append: Boolean);
        override;
  procedure InternalDelete; override;
  procedure InternalPost; override;
  // 기타 메소드들
  procedure InternalClose; override;
  procedure InternalHandleException; override;
  procedure InternalInitFieldDefs; override;
  procedure InternalOpen; override;
  function IsCursorOpen: Boolean; override;
  { 반드시 재정의하지는 않아도 되는 메소드들 }
  function GetRecordCount: Integer; override;
  function GetRecNo: Integer; override;
```

```
    procedure SetRecNo(Value: Integer); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean;
          override;

    // 추가적인 프로시저
    procedure EmptyTable;
  published
    property Active;
    property TableName: string read FTableName write SetTableName;
    property BeforeOpen;
    property AfterOpen;
    property BeforeClose;
    property AfterClose;
    property BeforeInsert;
    property AfterInsert;
    property BeforeEdit;
    property AfterEdit;
    property BeforePost;
    property AfterPost;
    property BeforeCancel;
    property AfterCancel;
    property BeforeDelete;
    property AfterDelete;
    property BeforeScroll;
    property AfterScroll;
    property OnDeleteError;
    property OnEditError;

    // 추가적인 속성들
    property DataFileSize: Integer read GetDataFileSize;
  end;

procedure Register;

implementation
```

```
uses BDE, DBTables, SysUtils, DBConsts, Forms, Controls, Dialogs;

const
  feDDGTable = '.ddg';
  feDDGIndex = '.ddx';
  // 파일이 잠기지 않는다는 점을 주의하기 바람.

{ TIndexList }

procedure TIndexList.LoadFromFile(const FileName: string);
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmOpenRead or fmShareDenyWrite);
  try
    LoadFromStream(F);
  finally
    F.Free;
  end;
end;

procedure TIndexList.LoadFromStream(Stream: TStream);
var
  Value: Integer;
begin
  while Stream.Position < Stream.Size do
  begin
    Stream.Read(Value, SizeOf(Value));
    Add(Pointer(Value));
  end;
  ShowMessage(IntToStr(Count));
end;

procedure TIndexList.SaveToFile(const FileName: string);
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmCreate or fmShareExclusive);
  try
```

```
    SaveToStream(F);
  finally
    F.Free;
  end;
end;

procedure TIndexList.SaveToStream(Stream: TStream);
var
  i: Integer;
  Value: Integer;
begin
  for i := 0 to Count - 1 do
  begin
    Value := Integer(Items[i]);
    Stream.Write(Value, SizeOf(Value));
  end;
end;

{ TDDGDataSet }

constructor TDDGDataSet.Create(AOwner: TComponent);
begin
  FIndexList := TIndexList.Create;
  FRecordSize := SizeOf(TDDGData);
  FBufferSize := FRecordSize + SizeOf(TDDGBookmarkInfo);
  inherited Create(AOwner);
end;

destructor TDDGDataSet.Destroy;
begin
  inherited Destroy;
  FIndexList.Free;
end;

function TDDGDataSet.AllocRecordBuffer: PChar;
begin
  Result := AllocMem(FBufferSize);
end;
procedure TDDGDataSet.FreeRecordBuffer(var Buffer: PChar);
```

```
begin
  FreeMem(Buffer);
end;

procedure TDDGDataSet.InternalInitRecord(Buffer: PChar);
begin
  FillChar(Buffer^, FBufferSize, 0);
end;

function TDDGDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var
  IndexPos: Integer;
begin
 if FIndexList.Count < 1 then
    Result := grEOF
  else begin
    Result := grOk;
    case GetMode of
      gmPrior:
        if FRecordPos <= 0 then
        begin
          Result := grBOF;
          FRecordPos := -1;
        end
        else
          Dec(FRecordPos);
      gmCurrent:
        if (FRecordPos < 0) or (FRecordPos >= RecordCount) then
           Result := grError;
      gmNext:
        if FRecordPos >= RecordCount-1 then
          Result := grEOF
        else
          Inc(FRecordPos);
    end;
    if Result = grOk then
    begin
      IndexPos := Integer(FIndexList[FRecordPos]);
```

```
      Seek(FDataFile, IndexPos);
      BlockRead(FDataFile, PDDGData(Buffer)^, 1);
      with PDDGBookmarkInfo(Buffer + FRecordSize)^ do
      begin
        BookmarkData := FRecordPos;
        BookmarkFlag := bfCurrent;
      end;
    end
    else if (Result = grError) and DoCheck then
      DatabaseError('No records');
  end;
end;

function TDDGDataSet.GetRecordSize: Word;
begin
  Result := FRecordSize;
end;

function TDDGDataSet.GetFieldData(Field: TField; Buffer: Pointer):
 Boolean;
begin
  Result := True;
  case Field.Index of
    0:
      begin
        Move(ActiveBuffer^, Buffer^, Field.Size);
        Result := PChar(Buffer)^ <> #0;
      end;
    1: Move(PDDGData(ActiveBuffer)^.Height, Buffer^, Field.DataSize);
    2: Move(PDDGData(ActiveBuffer)^.ShoeSize, Buffer^, Field.DataSize);
  end;
end;

procedure TDDGDataSet.SetFieldData(Field: TField; Buffer: Pointer);
begin
  case Field.Index of
    0: Move(Buffer^, ActiveBuffer^, Field.Size);
    1: Move(Buffer^, PDDGData(ActiveBuffer)^.Height, Field.DataSize);
    2: Move(Buffer^, PDDGData(ActiveBuffer)^.ShoeSize, Field.DataSize);
```

```
  end;
  DataEvent(deFieldChange, Longint(Field));
end;

procedure TDDGDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PInteger(Data)^ := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData;
end;

function TDDGDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
  Result := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag;
end;

procedure TDDGDataSet.InternalGotoBookmark(Bookmark: Pointer);
begin
  FRecordPos := Integer(Bookmark);
end;

procedure TDDGDataSet.InternalSetToRecord(Buffer: PChar);
begin
  // 북마크 데이터는 레코드 위치를 뜻한다.
  FRecordPos := PDDGBookmarkInfo(Buffer + FRecordSize)^.Bookmarkdata;
end;

procedure TDDGDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData := PInteger(Data)^;
end;

procedure TDDGDataSet.SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag := Value;
end;

procedure TDDGDataSet.InternalFirst;
begin
  FRecordPos := -1;
end;
```

```
procedure TDDGDataSet.InternalInitFieldDefs;
begin
  // 데이터 레코드를 구성하는 각 필드에 해당하는 TFieldDef 객체들을 생성
  FieldDefs.Clear;
  TFieldDef.Create(FieldDefs, 'Name', ftString, SizeOf(TNameStr), False,
     1);
  TFieldDef.Create(FieldDefs, 'Height', ftFloat, 0, False, 2);
  TFieldDef.Create(FieldDefs, 'ShoeSize', ftInteger, 0, False, 3);
end;

procedure TDDGDataSet.InternalLast;
begin
  FRecordPos := FIndexList.Count;
end;

procedure TDDGDataSet.InternalClose;
begin
  if TFileRec(FDataFile).Mode <> 0 then
    CloseFile(FDataFile);
  FIndexList.SaveToFile(FIdxName);
  FIndexList.Clear;
  if DefaultFields then
    DestroyFields;
  FRecordPos := -1;
  FillChar(FDataFile, SizeOf(FDataFile), 0);
end;

procedure TDDGDataSet.InternalHandleException;
begin
  // 일반적으로는 이런 식으로 구현하면 된다.
  Application.HandleException(Self);
end;

procedure TDDGDataSet.InternalDelete;
begin
  FIndexList.Delete(FRecordPos);
  if FRecordPos >= FIndexList.Count then Dec(FRecordPos);
end;
```

```
procedure TDDGDataSet.InternalAddRecord(Buffer: Pointer; Append: Boolean);
var
  RecPos: Integer;
begin
  Seek(FDataFile, FileSize(FDataFile));
  BlockWrite(FDataFile, PDDGData(Buffer)^, 1);
  if Append then
  begin
    FIndexList.Add(Pointer(FileSize(FDataFile) - 1));
    InternalLast;
  end
  else begin
    if FRecordPos = -1 then RecPos := 0
    else RecPos := FRecordPos;
    FIndexList.Insert(RecPos, Pointer(FileSize(FDataFile) - 1));
  end;
  FIndexList.SaveToFile(FIdxName);
end;

procedure TDDGDataSet.InternalOpen;
var
  HFile: THandle;
begin
  // 데이터 파일과 색인 파일이 존재하는지 확인
  FIdxName := ChangeFileExt(FTableName, feDDGIndex);
  if not (FileExists(FTableName) and FileExists(FIdxName)) then
  begin
    if MessageDlg('Table or index file not found.  Create new table?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    begin
      HFile := FileCreate(FTableName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating table file');
      FileClose(HFile);
      HFile := FileCreate(FIdxName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating index file');
      FileClose(HFile);
```

```
    end
    else
      DatabaseError('Could not open table');
  end;
  // 데이터 파일을 연다.
  FileMode := fmShareDenyNone or fmOpenReadWrite;
  AssignFile(FDataFile, FTableName);
  Reset(FDataFile);
  try
    FIndexList.LoadFromFile(FIdxName); // 색인 파일에서 색인 목록을 읽는다.
    FRecordPos := -1;                 // 레코드 위치를 BOF 이전 위치로 한다.
    BookmarkSize := SizeOf(Integer);   // 북마크 크기를 설정한다.
    InternalInitFieldDefs;             // FieldDef 객체들을 초기화한다.
     // 기본 필드 객체들이 만들어져 있지 않으면 TField 객체들을 생성한다.
    if DefaultFields then CreateFields;
    BindFields(True);                 // FieldDefs를 실제 데이터에 연결한다.
  except
    CloseFile(FDataFile);
    FillChar(FDataFile, SizeOf(FDataFile), 0);
    raise;
  end;
end;

procedure TDDGDataSet.InternalPost;
var
  RecPos, InsPos: Integer;
begin
  if FRecordPos = -1 then
    RecPos := 0
  else begin
    if State = dsEdit then RecPos := Integer(FIndexList[FRecordPos])
    else RecPos := FileSize(FDataFile);
  end;
  Seek(FDataFile, RecPos);
  BlockWrite(FDataFile, PDDGData(ActiveBuffer)^, 1);
  if State <> dsEdit then
  begin
    if FRecordPos = -1 then InsPos := 0
    else InsPos := FRecordPos;
```

```
    FIndexList.Insert(InsPos, Pointer(RecPos));
  end;
  FIndexList.SaveToFile(FIdxName);
end;

function TDDGDataSet.IsCursorOpen: Boolean;
begin
  // 데이터 파일이 열려 있는 경우에만 "커서"가 열린 것으로 한다.
 // FDataFile의 Mode 속성이 0이 아니면 파일이 열려 있는 것이다.
  Result := TFileRec(FDataFile).Mode <> 0;
end;

function TDDGDataSet.GetRecordCount: Integer;
begin
  Result := FIndexList.Count;
end;

function TDDGDataSet.GetRecNo: Integer;
begin
  UpdateCursorPos;
  if (FRecordPos = -1) and (RecordCount > 0) then
    Result := 1
  else
    Result := FRecordPos + 1;
end;

procedure TDDGDataSet.SetRecNo(Value: Integer);
begin
  if (Value >= 0) and (Value <= FIndexList.Count-1) then
  begin
    FRecordPos := Value - 1;
    Resync([]);
  end;
end;

procedure TDDGDataSet.SetTableName(const Value: string);
begin
  CheckInactive;
  FTableName := Value;
```

```
  if ExtractFileExt(FTableName) = '' then
    FTableName := FTableName + feDDGTable;

  FIdxName := ChangeFileExt(FTableName, feDDGIndex);

end;

procedure Register;
begin
  RegisterComponents('DDG', [TDDGDataSet]);
end;


function TDDGDataSet.GetDataFileSize: Integer;
begin
  Result := FileSize(FDataFile);
end;

procedure TDDGDataSet.EmptyTable;
var
  HFile: THandle;
begin
  Close;

  DeleteFile(FTableName);
  HFile := FileCreate(FTableName);
  FileClose(HFile);

  DeleteFile(FIdxName);
  HFile := FileCreate(FIdxName);
  FileClose(HFile);

  Open;
end;

 end.
```



그림 29.4
사용자 데이터셋으로 구현한 애플리케이션 실행 화면

요약

이번 장에서는 델파이 데이터베이스 애플리케이션에서 VCL이 기본적으로 제공하지 못하는 기능들을 사용하는 방법을 설명했다. 또한, 델파이 애플리케이션에서 BDE 함수들을 직접 호출하는 데 필요한 몇 가지 규칙이나 과정들에 대해서도 이야기했다. 그리고 dBase 테이블이나 파라독스 테이블에 맞게 TTable을 개선해 보았다. 마지막 부분에서는 새로운 형식의 데이터셋을 다루기 위한 TDataSet 상속 컴포넌트를 만들어 보았다. 다음 장 “웹 브로커를 이용한 웹 애플리케이션 작성”에서는 웹 애플리케이션의 서버쪽 애플리케이션을 만드는 방법과 웹 클라이언트에게 실시간으로 데이터를 제공하는 방법에 대해서 이야기하겠다.

쉬어가는 페이지