파라독스 데이터베이스 파일의 이해
(Understanding Paradox Database File)

파라독스는 MS-DOS 가 사용되던 시절에 독자적인 사용자 인터페이스, 프로그래밍 언어, 파일 포맷으로 탄생했던 데이터베이스 프로그램이다. 이를 1993 년에 볼랜드에서 윈도우용 버전을 발표하면서 파일 포맷과 사용자 인터페이스를 분리하고, 이를 연결하기 위해 ODAPI(Object Database API)를 사용하도록 하였으며, ODAPI 가 IDAPI 를 거쳐 현재의 BDE 로 발전하기에 이른다.

이렇게 되는 동안 파라독스의 파일 포맷은 따로 분리되어 데스크 탑 데이터베이스 도구로 사용되었다. 최근에 볼랜드는 파라독스 데스크탑 데이터베이스 부분을 코렐 사에 매각하였고, 이것이 코렐의 오피스의 근간이 되었다. 아직도 파라독스 파일 포맷은 BDE 가 지원하는 가장 중요한 파일 포맷이다.

많은 델파이 개발자는 파라독스 테이블을 매일 사용한다. 그렇지만 여기에 대한 정보는 거의 없다고 해도 과언이 아니다. 아직도 파라독스 파일 포맷은 가장 견고하고 잘 설계된 파일 포맷이며, 데스크 탑 데이터베이스와 파일 서버 형태로 사용하는 데이터베이스로서는 가장 좋은 것으로 알려져 있다.

이번 장에서는 파라독스 테이블의 내부 구조에 대해서 살펴보고, 실제로 테이블에 필드를 추가, 삭제하거나 색인의 생성, 테이블의 pack, 패스워드 정의 등의 여러가지 작업을 할 때 내부적으로 어떻게 동작하는지에 대해서 알아보도록 한다. 이 내용은 인터넷에 공개된 문서를 바탕으로 작성한 것이다.

파일의 종류

마이크로소프트의 엑세스는 데이터베이스 전체를 하나의 파일로 정의하고 있다. .MDB 파일 안에는 테이블과 인덱스, 여러가지 관계와 리포트, 쿼리 심지어 코드까지 저장되어 있다. 이러한 저장 방식에는 Win32 의 구조화 저장 기법이 사용된다. 구조하 저장 기법에 대해서는 이 책의 다른 장에서 언급하므로 참고하기 바란다.

이에 비해 파라독스는 각 기능에 따라 다른 파일로 저장하고 있다. 데이터베이스는 연관이 있는 테이블의 시리즈를 담고 있는 서브 디렉토리이며, 테이블의 각각의 요소들은 분리된 파일에 저장된다. 이들은 하나의 패밀리(family)를 구성하며, 사용자가 파일의 이름을 정의하면 BDE 가 확장자를 조절한다. 파라독스 파일로 사용되는 것에는 다음과 같은 것들이 있다.

1. .DB: 테이블 정의와 메모/그래픽/BLOB 필드 데이터를 제외한 모든 데이터 포함
2. .MB: 메모/그래픽/BLOB 데이터
3. .PX: 테이블의 Primary 키에 대한 인덱스

4. .Xnn, .Ynn: Secondary 키에 대한 인덱스
5. .VAL: 테이블의 참조 무결성과 타당성 검사 등의 정의

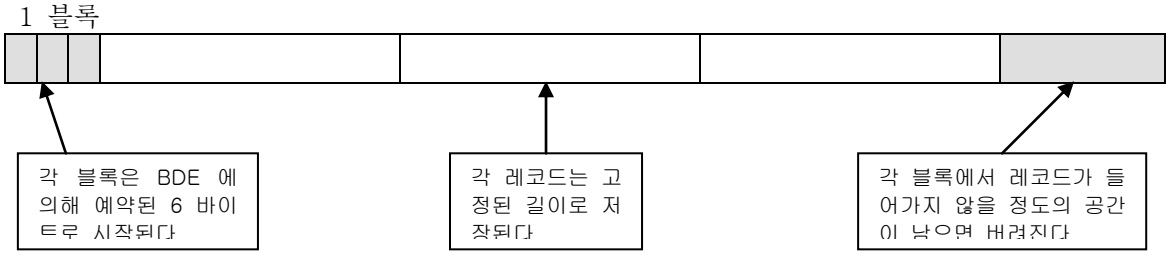
파라독스의 과거 버전과 데이터베이스 데스크 탑에서는 이들 외에 .TV, .FAM, .SET, .F, .R 등의 확장자를 가진 파일 들이 있다.
테이블을 복사할 때에는 이들을 모두 같이 복사해 주어야 한다. 그렇지 않으면 문제가 발생할 수도 있다.

테이블 포맷

내부적으로 파라독스 파일 포맷은 fixed-length record, VSAM-block, clustered-key 파일 포맷으로 알려져 있다. 이를 쉽게 풀어서 표현하면 다음과 같다.

1. 테이블 내에 고정된 길이를 차지하는 레코드에 저장된 데이터는 길이와 관계 없이 고정되게 저장된다. 예를 들어, 10 자 길이의 alphanumeric 필드에서 이 보다 짧은 길이의 값이 있어도 10 자 길이를 차지한다. (Fixed-length record)
2. 레코드는 물리적인 블록으로 이루어 지는데, 이들은 순차적으로 접근할 수도 있고 랜덤으로 접근할 수도 있다. 각각의 블록에는 하나 이상의 레코드가 존재하며, 레코드는 블록의 크기를 조절하게 만들지 못하므로 각 블록의 뒷 부분에는 레코드 하나가 못들어갈 크기 정도의 빈 공간이 존재하는 경우가 있다. (VSAM-block)
3. 각 블록 내부의 레코드 들은 primary 키 순서에 따라 저장된다. 이렇게 함으로써 primary 키가 사용될 때 접근 속도를 향상시킬 수 있다.

이를 그림으로 정리하면 다음과 같다.



각 테이블은 최소한 2kb 크기의 헤더 파일로 시작되는데, 저장되어야 할 정보량에 따라서 2kb 의 배수의 크기가 될 수도 있다. 헤더에는 다음과 같은 정보 들이 담겨 있다.

1. Structure ID

패밀리 파일의 동기화를 위해 사용되는 것으로, BDE 가 패밀리 파일 중 하나를 열면 파일 헤더의 Structure ID 가 .DB 파일의 ID 와 동일한지 검사하게 된다. 이것이 다르면 .DB 파일이 다른 패밀리 멤버 들이 업데이트 되지 않았을 때, 따로 변경되었다는 의미이다. 그러므로, 테이블의 구조를 재조정하면 그 때마다 모든 패밀리 멤버는 refresh 되어야 한다.

2. 헤더의 크기 (Byte)
3. 레코드의 크기 (Byte)
4. 블록의 크기 (1k, 2k, 4k, 8k, 16k, 32k)
5. 테이블에 키가 있는지 여부
6. 테이블의 필드 수 (1~255)
7. 테이블에 키가 있으면 primary 키가 되는 필드의 번호
8. 필드의 종류, 크기의 배열
9. 필드 이름의 배열
10. 테이블의 데이터 블록의 수 (2 Byte)
11. 첫번째 데이터 블록의 포인터
12. 마지막 데이터 블록의 포인터
13. 테이블 내의 사용 가능한 블록의 수 (2 Byte)
14. 첫번째 사용 가능한 블록에 대한 포인터
15. 문자 번역 방법과 소트 순서 등을 정의하는 테이블 언어
16. AutoIncrement 필드의 최대값에 대한 현재 값
17. 테이블의 데이터를 암호화하는 마스터 패스워드, 헤더는 암호화 되어 있지 않기 때문에 이 패스워드는 안전하다고 말할 수 없다.
18. 테이블에 대한 부가 패스워드의 암호화된 리스트

데이터 블록

테이블을 처음 생성하면, 데이터 블록은 할당되지 않는다. 이때에는 테이블이 헤더로만 구성되는데, 첫번째 레코드를 추가함과 동시에 데이터 블록이 .DB 파일에 바로 추가된다.
데이터 블록의 크기는 1k, 2k, 4k, 8k, 16k, 32k 까지 가능하다. 디폴트로는 2k 가 설정되어 있지만, BDE Administrator 에 의해서 수정이 가능하다. 블록의 크기는 테이블 내에서 변화되지 않는다. 일단 테이블이 생성되면 BDE 는 블록 크기를 결정하고, 이 크기 만큼 .DB 파일 뒤에 부착시킨다. 데이터 블록의 수는 64k 로 한정되어 있기 때문에, 테이블의 최대 크기는 헤더 크기를 제외하고, 데이터 블록의 크기에 따라 1k 인 경우 64M, 2k 는 128M 이며, 32k 인 경우에는 최대 2048M(2G)까지 가능하다.
그렇지만, 실제로는 이러한 한계에 접근하기는 어렵다. 예를 들어, 100MB 가 넘는 테이블의 구조를 재조정할 경우 경우에 따라서는 이 작업이 몇 시간이 걸릴 수도 있다.

파라독스 파일의 데이터 블록에는 BDE 에 의해 사용되는 6 바이트가 있다. 이들은 3 개의 포인터로 이루어져 있으며, 각 포인터의 크기는 2 바이트이다. 이들의 의미는 다음과 같다.

1. 다음 논리 블록의 번호. 이러한 포인터의 시리즈를 forward chain 이라고 한다. BDE 는 블록 번호를 블록 크기로 곱하고, 여기에 헤더의 크기를 더해서 블록의 물리적인 시작 주소를 계산한다.
2. 이전 논리 블록의 번호. 이러한 포인터의 시리즈를 backward chain 이라고 한다.
3. 블록에 얼마나 많은 활성화된 레코드가 있는지 나타내는 카운터. 데이터 레코드는 언제나 블록의 처음 부분에서 시작하므로, 헤더에 저장되어 있는 레코드 크기를 이용해서 얼마나 많은 블록이 활성화된 데이터를 가지고 있는지 알 수 있다.

데이터 블록이 완전히 비어 있으면, 이 블록은 자유 블록(free block) chain 으로 옮겨 가게 된다. 이 chain 은 BDE 가 테이블에 또 다른 데이터 블록을 추가해야 할 때에 사용된다. 자유 블록에도 다른 블록과 마찬가지로 6 바이트의 포인터가 있으며, 3 번째 포인터가 나타내는 레코드 수는 0 으로 설정되어 있다.
BDE 는 가끔 레코드 크기를 계산할 때 반올림을 적용한다. 예를 들어 블록 크기가 4k(4096)인 경우, 키가 있는 테이블의 최대 레코드 크기를 (4096 – 6) / 3 으로 계산해서 1363 바이트로 생각할 수 있다. 그렇지만, BDE 는 1350 바이트를 최대 크기로 결정하고 있다. 만약 레코드의 크기가 1350 바이트가 넘으면 8k 블록을 사용한다.

레코드의 삽입과 삭제

레코드의 크기가 204 바이트인 키가 있는 테이블을 생각해 보자. 파라독스 테이블 내에서는 각 블록에 10 개의 레코드를 저장한다고 할 때 2k 크기의 블록을 사용한다. 이 경우 2040 바이트가 10 개의 레코드에 할당되고, 처음의 6 바이트가 블록에 대한 포인터로 사용되므로 단지 2 바이트만 버려진다.
여기서 이 테이블에 4 개의 레코드가 추가되면, BDE 는 블록에 있는 처음 4 개의 레코드 슬롯에 레코드를 저장하고, 레코드 카운터에 4 를 저장한다. 이때 블록의 구조는 다음과 같다.

4	A	B	C	D						

여기서 C 레코드를 삭제하면 BDE 는 레코드 D 를 C 의 자리로 옮겨 오게 되고, D 가 있었던 공간은 지워지지 않으므로 D 레코드의 데이터는 그대로 남아 있다. 그리고, 레코드 카운터는 3 으로 줄어든다. 이를 그림으로 표현하면 다음과 같다.

3	A	B	D	D						

여기서 추가되는 레코드가 있으면 과거의 D 레코드 자리는 다른 레코드로 차게 된다. 예를 들어, A 와 B 레코드 사이에 A1 레코드를 삽입하면 다음과 같이 변하게 된다.

4	A	A1	B	D						

여기에 E, F, G, H, I, J 라는 레코드 6 개를 추가하면 다음과 같이 데이터 블록이 모두 차게 된다.

10	A	A1	B	D	E	F	G	H	I	J

그런데, 여기서 K 레코드 하나를 더 추가하면 하나의 데이터 블록이 더 할당된다. 이때 BDE 는 이전 블록의 마지막 레코드를 새 블록으로 옮겨 오는데, 이렇게 해서 블록의 마지막 슬롯을 비워 둔다. 이렇게 하는 이유는 A 와 J 사이에 레코드가 삽입될 경우 새로운 블록을 필요없게 하기 위해서이다. K 라는 레코드를 하나 추가할 경우 다음과 같이 배열된다.

9	A	A1	B	D	E	F	G	H	I	J
10	J	K								

여기에 두개의 레코드를 E 와 F 레코드 사이에 추가한다고 가정하자. 이때 첫번째 데이터 블록에 사용 가능한 슬롯이 마지막에 하나 밖에 없으므로, E1 만 E 와 F 사이에 위치하게 되고, 마지막 슬롯에는 I 레코드가 들어가게 된다. 이 상황에서 E2 가 삽입되면 데이터 블록이 어쩔 수 없이 분리되어야 한다. 그렇지만, 블록의 마지막에서 삽입된 것이 아니므로 이러한 분리에는 삽입 지점(insertion point)에서 일어나게 되며, 이 삽입 지점 뒤에 위치하는 모든 레코드는 다음과 같이 새로운 데이터 블록으로 옮겨가게 된다.

6	A	A1	B	D	E	E1	F	G	H	I
2	J	K								
5	E2	F	G	H	I					

이 경우에 새로운 블록은 테이블의 마지막에 위치하게 되므로, BDE 는 이들 블록의 순서를 정확하게 유지하기 위해서 forward, backward chain 번호를 새로 매기게 된다. Forward chain 의 경우 1, 3, 2 의 순서를 가지게 되며, backward chain 의 경우 2, 3, 1 의 순서를 가지게 된다.

데이터베이스 데스크탑에서 테이블을 restructure 하면 BDE 는 빈 레코드 슬롯을 제거하지 않으며, 물리적으로 데이터 블록을 재배치하지도 않는다. 그러나 Pack Table 옵션을 선택

하면 테이블에서 사용되지 않는 공간을 모두 없애버리고 새롭게 테이블을 구성하는데, 이때 각각의 블록은 모두 레코드로 꽉 차게 되고, 자유 블록 chain 은 비워진다. 또한, forward chain 에 입각해서 모든 데이터 블록의 순서를 새롭게 구성한다.
이러한 방식은 키가 결정된 테이블에 적용되는 것이다. 키가 없는 테이블의 경우에는 특정 순서에 맞추어 데이터의 삽입이 결정될 필요가 없으므로, BDE 는 각 슬롯을 사용이 가능한 형태로 버려두지 않는다.

필드의 형

각 레코드는 1~255 필드로 구성된다. 파라독스에서는 17 가지 종류의 필드가 존재하는데, 이들에는 문자열을 나타내는 Alpha, Number, Money, LongInt, Data 등에서 부터 OLE, Binary 등까지 다양하게 존재한다.
이들의 크기는 대체로 필드를 정의할 때 결정되는데, Memo 형의 경우 BDE 는 지정된 처음 n 바이트는 .DB 파일에 저장되며, 나머지 내용은 .MB 파일에 저장된다. Formatted Memo, Graphic, OLE, Binary 형의 경우에도 메모 파일과 마찬가지로 저장되는데, 사실 .DB 파일에 저장되도록 지정하는 n 바이트는 거의 쓸모가 없기 때문에, 0 으로 지정하는 것이 가장 좋다. Memo 형의 경우에는 이와 달리 .DB 파일에 저장된 문자를 이용해서 검색을 할 때 사용될 수 있다.
델파이는 OLE, Binary, Byte 필드의 내용을 보여줄 수 있는 컴포넌트를 제공하지 않는다. 이들은 OLE 를 사용하는 어플리케이션이나 데이터 스트림을 이용해서 간접적으로 폼에 보여주게 된다. 그러므로, 이들 필드에 대해서는 약간의 코딩이 필요하다. 예를 들어, 실험용 기계에서 RS-232C 포트를 통해서 나온 데이터를 Byte 필드에 저장했다가 이를 스트림으로 읽어서 해석한 후 보여주는 등의 작업을 해 주어야 한다.

레코드와 테이블 크기 계산

다음과 같은 구조의 테이블이 있다고 하자. 이 테이블의 레코드의 크기와 테이블의 크기 등을 계산해 보자.

필드 이름	종 류	크 기	비 고
Key	LongInt	4	Primary Key
ID	Alpha 8	8	ID
Password	Alpha 8	8	패스워드
Name	Alpha 10	10	이 름
Address	Alpha 100	100	주 소
BirthDay	Date	4	생년월일

이 경우에 레코드의 크기는 4+ 8+ 8+ 10+ 100+ 4 = 133 바이트가 된다. 그런데, 이 테이블은 키를 가지고 있으므로, BDE 는 최소한 3 개의 레코드를 가질 수 있는 가장 작은 블록 크기(1k 제외, 1k 는 과거 버전의 파라독스에서만 사용한다)를 사용한다. 이 경우에는 레코드 3 개를 합쳐도 399 바이트에 불과하므로 2k(2048) 바이트 크기의 블록이 데이터 블록으로 사용된다. 각 블록의 처음 6 바이트는 사용할 수 없으므로, 실제로 사용 가능한 공간은 2042 바이트가 된다.
그러므로, BDE 가 15 개의 레코드를 한 블록에 채워 넣으면 1995 바이트가 사용되고, 블록의 끝에 남는 47 바이트는 버려진다. 참고로, 테이블의 필드 구조를 바꿀 경우 3 바이트 크기까지의 필드를 추가하면 15 레코드이므로 블록에 45 바이트까지 가능하기 때문에, 테이블의 크기를 증가시키지 않아도 되지만, 4 바이트가 넘는 필드를 추가할 경우에는 47 바이트를 넘어버리기 때문에 15 레코드를 한 블록에 넣을 수 없게 되어, 테이블의 크기가 왕창 증가하게 된다.
테이블의 크기는 앞에서도 언급했지만 삽입과 삭제에 의해서 테이블의 크기가 변화할 수가 있다. 그러므로, 다음의 계산은 테이블을 pack 했다고 가정한 것으로 가장 작은 크기라고 말할 수 있다. 계산 방법은 단순히 레코드의 수를 15 로 나누어 올림하고, 각 블록 당 2k 이므로 2k 를 곱하면 얻을 수 있다. 그리고, 여기에 헤더에 해당되는 2k 를 더해주면 된다. 여기에 입각해서 계산을 해보면 100 레코드면 16,384 바이트가 되며, 10000 레코드면 1,368,064 바이트가 된다.

인덱스의 역할

원래의 관계형 모델에서는 인덱스라는 것이 존재하지 않는다. 인덱스는 파일 포맷의 일부로 수행속도를 향상시키기 위해 디자인 된 것이다.
인덱스란 정렬된 리스트의 일종으로, 파라독스 테이블에 대한 인덱스를 생성하면 BDE 는 각 레코드에 대한 필드의 값을 담은 독립된 파일을 생성한다. 이 값들은 문자의 순서나 숫자의 순서에 따라 정렬되어 있고, 테이블에서의 실제 위치를 가리키는 포인터를 가지고 있다.
예를 들어, 다음과 같은 테이블이 있다고 가정하자.

레코드 위치	필드의 내용	레코드 위치	필드의 내용
1	지훈	4	준완
2	현묵	5	대욱
3	종상	6	현호

이 테이블의 필드를 지정해서, 인덱스를 만들면 인덱스에는 다음과 같이 저장된다.

대욱	5	준완	4	현묵	2
종상	3	지훈	1	현호	6

어플리케이션에서 BDE 에 특정 값을 검색하라고 요청하면, 그 필드에 대한 인덱스가 있으면 BDE 는 정렬되어 있는 인덱스를 이용해서 값을 찾은 뒤, 포인터를 읽어서 실제로 레코드의 위치를 찾아낸다. 이때 정렬이 되어 있기 때문에 BDE 는 이진 검색 기법을 사용할 수가 있으므로, 순차적으로 검색하는 방법에 비해서 훨씬 효율과 수행속도가 빠르다.

1 차 인덱스 (Primary Index)

관계형 모델에서는 레코드의 유일성을 보장하기 위해서 하나 이상의 필드를 1 차 키(primary key)로 사용한다. 이 키는 테이블에서 하나만 설정할 수 있으며, 검색 작업 등을 할 때 빈번히 사용된다. 또한, 다른 테이블 과의 연결 시에 외부 키(foreign key)로 사용되기도 한다.
1 차 키가 이렇게 중요하기 때문에, BDE 는 1 차 키에 대한 인덱스를 생성하고 이를 1 차 인덱스라고 한다. 1 차 인덱스는 .PX 파일에 저장되며 테이블 내의 모든 레코드에 대한 엔트리를 포함하지 않는다는 측면에서 다른 인덱스와는 다르다.
앞에서도 설명했지만 1 차 키가 있는 모든 파라독스 테이블은 데이터 블록에 최소한 3 개의 레코드를 가지고 있어야 한다. 이 경우에 블록 내부에서는 모든 레코드는 1 차 키의 순서에 따라서 배열된다는 것은 이미 설명했다.
1 차 인덱스를 최소한의 크기로 유지하기 위해서, BDE 는 1 차 인덱스 파일에 각 블록의 첫 번째 레코드 값만을 저장한다. BDE 가 레코드를 검색할 때에는 검색하고자 하는 값을 넘지 않는 가장 큰 값을 가진 블록이 선택되며, 찾고자 하는 레코드는 그 블록에 존재하게 된다. 각 블록 내에서는 1 차 키에 따라 레코드가 순차적으로 배열되어 있으므로 여기에서 검색을 하면 된다. 달리 말하면 2 차례의 이진 트리 검색을 하게 되는 것이다.
이런 형태의 인덱스 방식의 잇점은 인덱스의 크기를 많이 줄일 수 있으며, 검색 속도를 향상시킬 수 있다는 것이다. 또한, 인접한 키 값의 레코드는 실제 물리적으로도 가깝게 위치하므로 디스크 접근 속도도 빠르다.

2 차 인덱스 (Secondary index)

1 차 키 이외의 필드에 대한 작업을 많이 할 때에는 검색과 필터를 사용할 때의 수행속도 향상을 위해서 그 필드에 새롭게 인덱스를 걸어주기도 한다. 이러한 인덱스를 2 차 인덱스라고 한다. 2 차 인덱스는 하나 이상의 필드로 구성되며, 오름-내림차순 또는 혼합형을 선택할 수 있고, 대소문자를 가리거나, 중복의 허용 여부 등을 설정할 수 있도록 되어 있다.

이러한 2 차 인덱스는 .Xnn, .Ynn 이라는 확장자를 가지는 2 개의 파일에 저장된다. 이때 하나의 필드에 대해, 오름차순이며 대소문자를 가리는 non-unique 인덱스이면 nn 은 01 부터 FF 까지의 16 진수 문자인데, 테이블이 최고 255 개의 필드를 가질 수 있으므로 각 테이블의 모든 필드를 감당해낼 수 있게 된다.
그런데, 복합 필드를 인덱스로 사용하거나 내림차순이나 혼합형의 정렬 순서를 가지거나 대소문자를 가지는 경우, 또는 unique 인덱스인 경우에는 nn 이 G0 에서부터 하나씩 증가하면서 확장자를 가지게 된다. 예를 들어 이런 새로운 스타일의 인덱스가 처음 16 개까지는 확장자의 nn 이 G0 에서 GF 로 결정되며, 다음 인덱스는 H0 가 된다.

2 차 인덱스 파일의 구조

.Xnn 파일은 일종의 파라독스 테이블 구조이다. 이 테이블의 구조는 다음과 같다.

Secondary Key	Primary Key Columns			Hint

Secondary Key 필드는 실제로 인덱스된 값이 들어 있으며, 하나 이상의 필드가 인덱스된 경우에는 두 필드를 합친 값이 들어가 있다. 이때 Secondary Key 필드의 값은 유일하지 않다. 대소문자를 가리지 않는 인덱스의 경우에는 대문자로 변환되어 들어가게 된다. Primary Key Columns 필드에는 각 필드에 대한 인덱스 값을 찾을 수 있는 실제 테이블의 1 차 키가 저장된다. 마지막의 Hint 필드에는 이 1 차 키를 찾을 수 있는 물리적인 블록의 번호가 적혀있다.
.Xnn 파일이 일종의 테이블이므로 .DB 파일과 마찬가지로 1 차 키를 가지고 있다. 이때 Secondary Key 필드는 유일하지 않으므로, 원래 테이블의 1 차 키를 Xnn 테이블의 1 차 키로 사용한다. 그렇다면 이 1 차 키의 1 차 인덱스에 해당되는 파일은 무엇일까 ? 이것이 바로 .Ynn 파일이다. 즉, .DB 파일과 .PX 파일의 관계와 .Xnn, .Ynn 파일의 관계가 비슷한 것이다.
이렇게 복잡한 구조를 가지고 있으므로 .Xnn 파일은 .PX 파일에 비해서 훨씬 크기가 크다. 심한 경우에는 테이블의 각 레코드에 대한 엔트리를 포함하므로 .DB 파일 보다도 클 수가 있다. 그렇지만, .Ynn 파일의 크기는 작다.

2 차 인덱스의 동작

그러면, 예를 들어서 실제로 BDE 가 2 차 인덱스를 어떻게 사용하는지 알아보도록 하자. 설정된 2 차 인덱스는 TTable 컴포넌트에 의한 검색 뿐만 아니라 TQuery 컴포넌트에서 SQL 문장으로 실행해도 효과적으로 사용된다. 다음의 SQL 문장을 보자.

SELECT * FROM Customer WHERE Country = “Korea”

여기서 Country 필드는 1 차 키가 아니지만, 2 차 인덱스를 걸어놓았다고 하자. 그러면, BDE 는 SQL 문장을 실행하면서 2 차 인덱스를 사용한다. 이때 실제로 BDE 가 수행하는 작업은 다음과 같다.

1. .Ynn 파일을 열고 .PX 파일을 사용할 때처럼 이진 검색 기법을 이용하여 .Xnn 파일에서 “Korea” 엔트리가 들어있는 시작 블록을 찾는다.
2. .Xnn 파일에서 앞에서 찾은 시작 블록으로 가서 순차적으로 “Korea” 엔트리와 일치하는 레코드를 찾는다.
3. 레코드를 찾으면 Secondary Key 필드가 “Korea”가 아닌 레코드가 나올 때까지 순차적으로 레코드를 계속 읽는다.
4. 조건에 맞는 각 레코드의 Primary Key 정보와 Hint 필드에서 실제 테이블의 정보를 가져 온다.
5. BDE 는 테이블의 Hint 필드에 저장된 블록으로 가서 해당 레코드가 나올 때까지 순차적으로 검색한다. 이때 블록 안에서 레코드가 발견되지 않으면 .PX 파일에서 이진 검색을 하게 되고, 해당되는 블록을 찾으면 다시 .DB 파일의 블록으로 가서 레코드를 순차적으로 검색한다.
6. 찾은 레코드를 가져온다

타당성 검사 (Validity Check)

타당성 검사는 필드에 값을 추가하거나 수정할 때 BDE 에 의해서 수행되는 간단한 비즈니스 규칙(business rule)을 의미한다. 파라독스 파일 포맷에서 지원하는 타당도 검사 항목에는 다음과 같은 것들이 있다.

1. 값이 요구되는지 여부 (Null 허용 여부) : Required
2. 필드의 최대값 : Maximum
3. 필드의 최소값 : Minimum
4. 레코드가 삽입될 때 디폴트 값 : Default
5. 포맷 마스크 : Picture

필드에 따라서 이러한 타당성 검사를 못하는 경우가 있는데, 많은 필드가 이들 모두를 지원한다. 그런데 AutoIncrement 형의 필드는 Minimum 만을 지원하며, 각종 Blob 필드 들은 Required 만 지원한다. Logical 형의 필드는 Required 와 Default 를 지원한다.

이러한 타당성 검사를 이용하면 어플리케이션 레벨에서가 아니라 데이터베이스 레벨에서 비즈니스 규칙을 정의할 수 있으며, 이로 인해 특정 어플리케이션에 종속적이지 않은 비즈니스 규칙을 활용할 수 있게 된다.
이러한 타당성 검사에 대한 값들은 .VAL 파일에 저장된다. 그런데, 이 파일에 대한 이진 파일 포맷이 공개되지 않았으므로 직접 수정은 불가능하다.

참조 무결성 (Referential Integrity)

테이블에서 하나 이상의 필드가 다른 테이블의 1 차 키를 참조하고 있을 때, 이런 필드를 외부 키(foreign key)라고 한다. 외부 키는 데이터베이스 내에 있는 다양한 테이블 간의 데이터 연관성을 유지하는데 매우 중요한 역할을 한다. 참조 무결성 규칙은 두 개의 테이블이 1 차 키와 외부 키의 관계로 연결되어 있을 때, 하나의 테이블의 모든 외부 키의 값은 다른 테이블의 1 차 키의 값과 반드시 일치해야 한다는 것이다.
이러한 참조 무결성은 각 테이블에 작업을 할 때 깨질 수 있는 상황들이 발생한다. 특히 데이터를 삭제하거나 업데이트할 때가 문제가 된다. 이럴 때 참조 무결성을 지키기 위한 방법으로는 크게 세가지가 있다. 첫째 방법은 변화가 일어난 부분을 데이터베이스 내부에서 모두 적용시키는 방법이다. 즉, 필드 값이 삭제된 경우 연결된 모든 테이블의 해당 레코드를 삭제하거나, 데이터 값을 변경한다. 이를 'cascade' 라고 한다. 두번 째로 참조가 되고 있는 레코드의 삭제나 업데이트를 금지하는 경우이다. 이를 'prohibit' 이라고 한다. 마지막 방법은 연결된 값을 null 로 설정하는 경우이다. 이를 'null' 이라고 한다.
파라독스 파일 포맷에서는 이들 방법 중에서 몇 가지 방법 만을 사용할 수 있도록 되어 있다. 데이터를 업데이트할 경우에는 cascade 와 prohibit 중에서 하나를 선택할 수 있다. 또한, 데이터를 삭제할 때에는 prohibit 만을 지원한다. 이것의 의미는 개발자가 반드시 연결된 1 차 키가 있는 레코드를 삭제할 때, 이와 연결된 외부 키가 있으면 직접 이들 레코드를 삭제한 뒤에야 삭제가 가능하다는 것이다.
파라독스 파일 포맷의 가장 부족한 한계점을 지적한다면 바로 이와 같은 참조 무결성에 대한 불완전한 지원을 꼽을 수 있다.
이러한 참조 무결성에 대한 정보도 .VAL 파일에 저장된다.

암호화된 테이블

파라독스 파일 포맷에는 패스워드를 통하지 않고는 테이블을 보지 못하게 하는 암호화 기능을 가지고 있다. 파라독스 테이블에는 테이블에 조작을 하기 위해서 여러 단계의 패스워드 레벨을 요구한다.
암호화를 하기 위해서는 마스터 패스워드를 정해야 한다. 데이터베이스 데스크탑의 Create/Restructure 대화상자에서 이를 설정할 수 있다. 패스워드는 128 자까지 입력할 수

있으나, 처음 31 자만 의미를 가지는데, 대소문자를 가린다. 이 패스워드는 테이블의 소유자를 정의한다. 이 패스워드를 알아야 부가적인 패스워드를 바꿀 수 있는 권한이 있다.
부가 패스워드(auxiliary password)는 마스터 패스워드를 입력한 후에야 정의할 수 있다.
부가 패스워드를 입력하면 테이블과 필드에 대한 접근 권한을 일일이 정할 수 있는데, 테이블에 대한 접근 권한에는 다음과 같은 것들이 있다.

권 한	설 명
Read Only	테이블을 볼 수는 있지만 데이터나 구조는 수정할 수 없음
Update	테이블을 볼 수 있고, non-key 필드의 수정은 가능하지만, 레코드를 추가, 삭제하거나 키 필드의 수정은 불가능하다.
Data Entry	테이블을 볼 수 있고, non-key 필드의 수정, 레코드 삽입이 가능하지만, 레코드 삭제나 키 필드의 수정은 불가능하다.
Insert & Delete	레코드를 삽입, 삭제, 수정할 수는 있지만, 테이블의 구조를 바꿀 수는 없다.
All	마스터, 부가 패스워드를 바꾸는 것을 제외한 모든 권한을 가짐

필드에 관한 권한의 종류에는 None, Read Only, All 의 세 가지가 있는데, None 은 필드를 볼 수가 없기 때문에, 그 필드의 데이터는 숨겨진다. Read Only 는 필드의 내용을 볼 수는 있지만 수정이 불가능한 것이고, All 은 수정까지 모두 가능한 권한이다.
마스터 패스워드는 암호화된 형식으로 테이블의 헤더에 저장되며, 부가 패스워드는 마스터 패스워드를 이용해서 암호화하여 테이블의 헤더에 저장된다.
실제로 패스워드는 버퍼를 이용해서 저장된다. BDE 를 이용해서 파라독스 테이블을 열게 되면, 디폴트 세션이 생성된다. 이러한 세션에 접근하려면 TSesstion 컴포넌트의 프로퍼티, 메소드, 이벤트를 사용한다. 세션 객체는 하나 이상의 데이터베이스에 가상 연결을 하고, 각 세션은 데이터베이스에 대해 다른 사용자로 취급된다. 개발자는 추가적인 TSession 객체를 추가하고, 이를 이용해서 가상 유저를 관리할 수 있다.
BDE 는 패스워드에 대한 버퍼를 세션에 저장한다. 이 버퍼는 파라독스 테이블에서 사용되는 모든 패스워드를 담을 수 있을 정도로 큰데, 하나의 세션에 모두 25 개까지 담을 수 있다. 만약 사용자가 패스워드를 입력하면, 이것이 버퍼에 추가된다. 마찬가지로 어플리케이션에서 코드로 패스워드를 추가해도 같은 버퍼에 추가된다. 만약 패스워드를 버퍼에 두 차례 추가하면, 두 개가 모두 버퍼에 추가되기 때문에 한 개의 인스턴스가 제거되어도 여전히 암호화된 테이블에 접근할 수 있다.
어플리케이션에서 파라독스 테이블에 접근할 필요가 있으면, BDE 는 버퍼에 있는 패스워드 목록과 테이블의 헤더에 있는 패스워드를 비교해서 권한을 설정하게 된다. 이때 여러 개의 패스워드를 이용할 경우 상위의 권한을 얻게 된다.
암호화 테이블을 사용하면 BDE 의 작업 속도가 약 10~15% 정도 속도가 느려진다. 이는

BDE 가 데이터에 접근할 때 마다 암호화와 해독 알고리즘을 적용하기 때문이다.
또한, 암호화 테이블은 잘 압축이 되지 않는데, 이는 정상적인 파라독스 테이블의 경우에는 반복되는 데이터가 많은데 비해, 암호화 테이블에는 빈 공간과 반복되는 바이트가 거의 없기 때문이다.

BDE 와 네트워크 잠금 (Network Locking)

BDE 는 파일을 이용해서 파라독스 테이블과 레코드 잠금에 대해서 관리를 하게 된다. 이 파일들은 필요할 때 BDE 에 의해서 생성되며, 자동으로 관리된다.
PDOXUSRS.NET 파일은 파라독스 네트워크 컨트롤 파일로, 네트워크의 사용자에 대한 여러가지 관리에 관여하게 된다. 파라독스 테이블을 네트워크 상에서 접근하게 되는 모든 어플리케이션에서는 이 파일을 참조하게 되며, 전체 네트워크에는 이 파일이 하나만 존재한다. 이 파일의 위치는 IDAPI.CFG 파일에 저장되어 있으며, 이 파일은 각 사용자에 의해 관리되거나 네트워크 상에서 공유된다. 이를 설정하려면 BDE Administrator 를 실행하여 Drivers 페이지에 있는 파라독스 드라이버에 대한 설정에서 Net Dir 을 이용해서 정의할 수 있다. 이 디렉토리를 네트워크 컨트롤 디렉토리라고도 한다.
PDOXUSRS.NET 파일이 지정된 위치에서 발견되지 않으면, BDE 에 의해 자동으로 생성된다. 이 파일에서는 최고 300 명의 가상 유저의 관리가 가능하다. 델파이 어플리케이션에서 PDOXUSRS.NET 파일의 위치는 TSession.NetFileDir 프로퍼티를 이용해서 파악하거나 설정할 수 있다. 만약 다른 PDOXUSRS.NET 파일에 의해 관리하려 하면 새로운 세션 객체를 생성해야 한다.
디렉토리에 있는 테이블들에 대한 접근 권한 관리에는 PDOXUSRS.LCK 파일이 이용된다. 이 파일은 파라독스 테이블이 위치한 각 디렉토리에 BDE 에 의해 첫번째 사용자가 테이블에 접근할 때 생성된다. 이 파일에는 다음과 같은 세 종류의 파일이 있다.

1. 공유된 디렉토리에서는 테이블과 레코드에 대한 잠금(locking) 정보를 가지고 있다. 즉, 어떤 사용자와 세션이 잠금을 걸고 있으며 그 종류가 무엇인지를 저장하고 있다. 사용자가 테이블에 잠금을 걸려고 하면, 이 파일에서 잠금 정보를 확인하고 잠금이 존재하지 않으면 새로운 잠금을 설정하고 이를 파일에 기록한다.
2. 공유되지 않는 디렉토리에서는 이 파일에 특정 플래그가 설정되어 있어서, BDE 가 다른 사용자나 세션이 파라독스 테이블에 접근하지 못하도록 한다. 이러한 디렉토리에는 특정 사용자만 접근하게 하는데, TSession.PrivateDir 프로퍼티에서 이 디렉토리를 설정할 수 있다.
3. 공유가 되지만, 읽기 전용인 디렉토리에서는 일단 잠금이 설정되면 다른 잠금이 설정될 수 없으므로, 이 디렉토리의 모든 테이블에 접근할 수 없게 된다. 이를 잘 활용하면 작업의 수행속도를 증진시킬 수 있다.

디렉토리 잠금은 보통 PARADOX.DRO 파일을 사용하는데, 이 파일을 생성할 때에는 다음과 같은 코드를 사용한다.

```
var
  Lock_Path: Array [0..DbiMaxTblNameLen] of Char;
begin
  StrPCopy(Lock_Path, 'C:₩Path₩PARADOX.DRO');
  Check(DbiAcqPersistTableLock(Database1.Handle, Lock_Path, szPARADOX));
end;
```

DbiAcqPersistTableLock() 함수는 존재하지 않는 테이블에 까지 디렉토리에 대한 잠금을 걸 수 있으며, 이를 해제할 때에는 DbiRelPersisTableLock() 함수를 사용한다.
이와 같이 추가적인 파일을 이용해서 잠금을 제어하게 되면, 추가적인 정보를 에러 상황에서 사용할 수 있다. 예를 들어, 테이블에 접근할 때 잠금이 존재할 경우 BDE 를 이용해서 잠금을 걸고 있는 사용자 등을 알아낼 수 있다.

테이블 잠금 (Table Lock)

파라독스 테이블에 대한 잠금에는 Exclusive Lock, Write Lock, Read Lock, Open Lock 의 4 가지 종류가 있다. Exclusive Lock 은 테이블에 대해서 잠금을 건 사용자만 접근할 수 있는 경우로 보통 테이블을 생성하거나, 구조를 바꿀 때 사용된다. Write Lock 은 데이터를 읽을 수는 있지만 이를 수정하지 못하게 하는 잠금으로, 테이블을 복사할 때 쓰는 쪽의 테이블에는 Write Lock 이 걸려 있어야 한다. Write Lock 은 한 테이블에 하나만 걸 수 있다. Read Lock 은 다른 사용자가 Read Lock 을 걸고 있지 않을 경우에 테이블에 쓰기를 할 수 있다. 모든 사용자가 데이터를 읽을 수 있다는 점에서는 Write Lock 과 같지만, 이 경우에는 한 테이블에 대해 여러 개의 Read Lock 이 걸릴 수 있으며, 다른 사용자가 Read Lock 을 걸고 있다면 쓰기를 할 수 없다. 마지막으로 Open Lock 은 최하 레벨의 잠금으로 다른 사용자도 테이블에 대한 여러가지 권한을 가질 수 있다. 단지 Exclusive Lock 이 설정된 테이블에만 접근할 수 없다.
이들 잠금에 대한 상호작용은 다음과 같다. 각 잠금에서 허용되는 잠금에는 * 표시를 하였다.

	Exclusive Lock	Write Lock	Read Lock	Open Lock
Exclusive Lock				
Write Lock				*

Read Lock			*	*
Open Lock		*	*	*

부연해서 설명하면 Open Lock 은 여러 명의 유저가 동시에 테이블을 볼 때 서로 가질 수 있다. 그리고, Read Lock 역시 여러 명이 동시에 걸 수 있는데, 이 경우에는 각각의 Read Lock 이 다른 사용자가 테이블에 데이터를 쓸 수 없게 한다. Write Lock 은 쓰기 권한을 잠금을 건 사용자에게만 부여하지만, 다른 사람들은 Open Lock 을 통해 데이터를 볼 수 있다.
델파이에서는 테이블을 공유 모드(Share Mode)나 전용 모드(Exclusive Mode)로 열 수 있으며, 이는 TTable.Exclusive 프로퍼티를 가지고 조절할 수 있다. 이 프로퍼티의 디폴트 값은 False 이며 이 경우 테이블은 Open Lock 으로 열리게 된다. 그리고, 테이블이 열리기 전에 True 로 설정될 경우에는 Exclusive Lock 을 걸게 되므로 다른 어플리케이션에서 테이블에 접근을 할 수 없다.
그리고, TTable.Lock 메소드를 이용해서 잠금을 걸 수 있는데 여기에서 사용할 수 있는 LockType 은 [liReadLock, ltWriteLock] 중의 하나이다. 이미 걸려있는 잠금을 해제할 때에는 TTable.UnLock 메소드를 사용한다.
어떤 경우에는 테이블 잠금을 걸 수 없는 경우가 있는데, 다음과 같은 경우가 있다.

1. 디렉토리가 로컬 하드 디스크에 있으면서 공유되지 않는 경우
2. 사용자의 Private Directory 에 위치하여 다른 사용자가 이 디렉토리에 접근할 수 없는 경우
3. 디렉토리가 CD-ROM 같이 읽기 전용 장치에 있는 경우
4. 테이블의 파일 속성이 읽기 전용일 때
5. PDOXUSRS.LCK 파일이 디렉토리 잠금으로 지정한 경우

레코드 잠금 (Record Lock)

사용자가 테이블에 있는 레코드를 편집하기 시작하면, 파라독스는 각 잠금 파일에 레코드 잠금을 걸기 시작한다. 그렇지만 보통의 경우에는 테이블 잠금은 Open Lock 을 계속 유지하고 있게 된다. 사용자가 새로운 레코드나 변경된 레코드를 post 하는 도중에는 테이블의 잠금이 Open Lock 에서 Write Lock 으로 변경된다.
BDE 는 한번에 테이블 당 255 레코드까지 잠금을 걸 수 있다. 이러한 잠금은 모두 exclusive 형태이기 때문에 오직 하나의 유저만 레코드를 편집할 수 있으며, 다른 유저들은 레코드가 편집되는 상태를 볼 수는 있게 된다.

정 리 (Summary)

이번 장에서는 델파이가 기본적으로 사용하는 파라독스 데이터베이스 파일에 대한 포맷과 파라독스의 구조를 살펴 보았다. 직접적으로 쓸 수 있는 정보는 아니지만, 데이터베이스 파일을 이해하고, 이들이 어떻게 관리되는지를 이해하는 데에는 커다란 도움이 되었을 것으로 믿는다.

다음 장에서는 멀티-tier 어플리케이션을 작성하는 방법을 알아볼 것이다.