클라이언트/서버 데이터베이스 어플리케이션의 제작 (I)

이번 장에서는 데이터베이스 어플리케이션을 원격 DBMS 를 이용하여 클라이언트/서버의 형태로 개발할 때 고려해야 할 사항 들에 대해서 2 장에 걸쳐서 알아보도록 하겠다.
델파이 3 부터는 클라이언트 데이터 세트를 이용한 멀티-tier 형태의 데이터베이스 어플리케이션 개발이 가능해졌기 때문에, 2-tier 형태의 클라이언트/서버 어플리케이션을 개발하는 방법에 대해서는 과거와 같은 중요성이 있다고 볼 수는 없다.
하지만, 아직도 많은 개발자 들이 이런 형태로 개발을 실제로 진행하고 있기 때문에 실제 개발에서 도움이 될 수 있는 정보를 제공하고자 한다.
먼저 RDBMS 를 이용하여 어플리케이션을 작성할 때 고려해야 할 주요 내용 들과 델파이 4 에서 새롭게 제공되는 SQL 빌더(과거의 비주얼 쿼리 빌더)의 사용 방법을 알아보고, 몇 가지 유용한 SQL 문장 들에 대한 소개와 클라이언트/서버 데이터베이스 어플리케이션을 사용할 때 유용하게 사용되는 캐쉬 업데이트(cached updates)에 대한 내용을 알아본다. 그리고, 다음 장에서는 주요 DBMS 의 특징과 데이터베이스 접속을 관리하는 방법에 대해서 알아보도록 할 것이다.

RDBMS 의 사용

RDBMS 를 사용하여 데이터베이스 어플리케이션을 개발하는 것은 각각의 데이터베이스의 구성 요소와 델파이의 구성 요소를 결합하는 것으로 서버 데이터베이스의 원하는 테이블과 저장 프로시저 등을 먼저 구성한 다음 델파이에서 이러한 환경에 접근하여 테이블을 조작하고 내용을 참조하는 작업을 하는 것이다. 이때 개발자가 고려해야할 것으로는 다음과 같은 것들이 있다.

1) 데이터베이스에 로그인
2) 트랜잭션 컨트롤의 사용
3) 특정 SQL 서버의 확장기능 사용
4) 만들어진 SQL 의 튜닝작업

물론 이외에도 초기 DB 설계 방법과 데이터웨어하우스 등 많은 작업이 필요하지만, 이 책에서 논의되는 내용은 델파이로 어플리케이션을 작성하는데 국한되므로 해당 내용만 다루도록 한다.

RDBMS 에서의 데이터 처리 요령

보통 개발자들은 검색 메소드로 SetRange, Locate, FindKey 등의 다양한 메소드를 사용하게 된다. 그렇지만, RDBMS 를 사용할 경우 이러한 작업은 많은 부하를 RDBMS 에 주게 되며 네트워크 트래픽도 발생한다. 더군다나 여러 개의 실행 모듈로 구분되는 프로그램을 제작할 경우 RDBMS 의 접속을 유지하기 위해 각각의 실행 모듈마다 TDatabase 컴포넌트를 사용하게 된다.
실제 RDBMS 의 접속은 상당한 시간을 소모하므로 이런 로그-인과 로그-아웃 작업은 많지 않은 것이 좋다. 그렇다고 이러한 접속을 계속 유지하는 것 또한 네트워크 트래픽과 서버의 부하에 영향을 주게 된다.
델파이 4 에서는 이를 해결하기 위해 다음과 같은 방법을 지원한다.

- 접속의 유지

원격 데이터 모듈(Remote DataModule)을 사용하여 하나의 클라이언트 시스템에서 하나의 접속을 사용하는 중간 연결 어플리케이션을 사용하고, 필요한 모듈은 해당 원격 리모트 데이터 모듈을 참고하는 TClientDataSet 이나 메소드로 동작하게 하면 접속을 적게 유지할 수 있으며, 작업 속도도 빨라진다. 더구나 TClientDataSet 의 Delta 내용을 적절하게 유지하고 캐쉬 업데이트를 사용하면 네트워크 트래픽을 많이 줄일 수 있다.

- 접속의 리소스 절약

MTS 와 CORBA 의 기능 중에 MTS 의 경우에는 MTS Pooling 과 세션 관리를 통하여 서버의 자원을 절약할 수 있으며, CORBA 의 경우에는 자원관리 기능을 이용하여 많은 사용자가 접근할 때 효과적으로 자원을 배분할 수 있다.

1. 많은 클라이언트의 요구를 처리하기 위한 여러 개의 어플리케이션 서버의 구축

델파이 3 부터 지원하기 시작한 MIDAS 와 델파이 4 의 MTS 의 분산객체 지원기능, CORBA 의 ORB 를 사용한 분산환경은 필요한 자원의 적절한 사용과 2 개 이상의 동일한 분산객체의 로드 밸런스(load balance)를 유지하여 전체 시스템의 성능을 보장한다.

2. 기본적인 SQL 의 튜닝작업

기본적인 SQL 문장을 기술하는 것에도 많은 신경을 써야 한다. 특히 테이블의 인덱스 작업과 재인덱스 작업은 중간중간 꼭 필요하다. 델파이에서는 어플리케이션 서버에서 쓰레드 등을 통한 모듈로 유휴 시간(idle time)을 적절하게 활용해서 이런 작업을 해 주어야 한다.

3. 데이터의 전반적인 연산작업은 RDBMS 에게 …

RDBMS 를 사용하는 것은 단순히 테이블의 내용을 클라이언트에서 보기위한 것이 아니다. 적절한 저장 프로시저와 트리거를 사용하여 RDBMS 의 기능을 100% 사용하게 하는 것이 중요하다. 예를 들어, 1 년 동안의 전체 통계를 낼 경우에 적절한 저장 프로시저를 사용하면 RDBMS 가 빠른 시간에 결과를 돌려줄 것이다.

SQL 작업 시의 주의할 내용과 참고사항

SQL 작업을 할 때 조금만 주의를 기울이면 많은 성능의 향상을 가져올 수 있다. 여기에 대해서 보다 자세하게 알아보도록 하자.

- 데이터의 구조와 비즈니스 어플리케이션의 구조, 규칙의 이해는 필수 !

개발자는 데이터베이스 내부의 데이터 크기와 분포를 반드시 알고 SQL 을 작성해야 한다. 또한, SQL 문장을 작성하기 전에 각 테이블의 연산방식과 구조에 대한 데이터 모델을 먼저 이해해야 한다. 이를 위해서 적극적으로 CASE 도구를 활용하여 비즈니스 관계의 데이터 객체의 구조를 문서화하는 작업이 필수적이다.

- 테스트는 실제 데이터를 가지고 할 것 !

많은 개발자가 보통 작업 시에는 많은 로드가 걸리는 실제 데이터 보다는 작게 표본화된 자료를 이용하여 프로그래밍을 하는 경우가 많다. 하지만, 데이터 100 개와 50000 개에서 동작하는 경우에 프로그래밍은 많은 차이를 보일 수 밖에 없다.
물론, 개발과정에서는 표본화된 자료로 작업할 수 있다. 그렇지만, 최소한 검사 단계에 이르면 반드시 실제 데이터 환경보다 더욱 많고 복잡한 상황 하에서 테스트해야 한다. 특히 표본자료에서 각각의 데이터의 분포도는 실제 데이터의 분포도와 유사한 구조를 가지는 것이 좋다.

- RDBMS 고유의 기능에 능숙해야 한다.

SQL 문장은 기본적으로 파싱(parsing)이라는 단계를 거친다. 그러나, identical SQL 문(저장 프로시저 등)은 이러한 단계를 거치지 않으므로 동작 속도가 빨라진다. 쉽게 이야기해서 저장 프로시저 등을 적극적으로 활용해야 한다.

- 인덱스의 사용이 중요하다.

인덱스를 적절히 사용하면 상당한 수행 성능의 향상을 가져올 수 있다. 그렇지만, 지나친 사용은 되려 시스템의 성능을 떨어뜨릴 수도 있다. 인덱스와 SQL 문장을 사용함에 있어서 다음과 같은 것들을 주의해야 한다.

1. 사용자에게 가장 많이 사용되는 컬럼을 찾아내어, 이 컬럼의 인덱스를 생성한다.
2. Join 이 자주 되는 테이블은 뷰(View)를 가지는 것이 좋고, Join 되는 필드도 반드시 인덱스가 필요하다.
3. 각 레코드(행, row)에 적은 비율을 가지고 다양한 분포도를 가지는 컬럼도 인덱스가 필요하다.
4. SQL 문장을 만들 경우에 Where 절에 함수가 사용되는 컬럼에는 인덱스를 만들지 않는 것이 더 낫다.
5. 자주 변경되는 인덱스는 인덱스 재구성으로 인하여 효율성이 떨어지므로, 이 비율이 적절한 것이 좋다.
6. 인덱스는 유일(unique)한 것이 좋다. 특히 Primary Key 부분은 반드시 unique 한 인덱스가 필요하다. 그렇지만, foreign key 와 Where 절에 사용되는 컬럼은 non-unique 인데스를 사용하는 것이 낫다.
7. 충분히 SQL 문장을 다듬어야 한다. 특히, Where 절에서 작은 크기의 인덱스와 큰 크기의 인덱스, 유일한 인덱스, non-unique 한 인덱스 중에 어느 것을 먼저 수행하느냐에 따라 SQL 문장의 수행속도가 많이 달라진다.
8. 해당 SQL 문장의 최적화 작업을 이해해야 한다. 보통 SQL 은 RULE-BASED 나 COST-BASED 중의 하나로 구동된다. 보통 전통적인 RDBMS 의 경우 RULE-BASED 를 사용하였지만, 새로 출시된 RDB 의 경우에는 COST BASED 방식을 사용한다. 이 방식은 반드시 ANALYZE 스키마를 사용하여 데이터베이스의 이용 통계를 내고, 이 데이터를 데이터 사전 테이블에 기록하고, 이 자료를 COST BASES OPTIMIZER 를 통하여 사용하게 된다.
9. SQL 문장들 간의 여러 상관관계에 대해 전체 SQL 문장을 염두에 두어야 한다. 내가 만든 인덱스와 JOIN 이 다른 SQL 에 많은 영향을 줄 수도 있기 때문이다.
10. WHERE 절을 다시 한번 살펴보자. >, <, = 등의 연산은 A < B 의 경우 A 에 인덱스를 만들어 동작하게 된다. 그러나 NULL 값을 가지는 연산인 경우에는 A NOT IN (내용 1, 내용 2), A != 연산식, A LIKE '%패턴내용' 등을 사용하게 되는데 이때에는 인덱스가 동작하지 않는다. 그러므로, 이러한 연산은 피해야 한다.
11. HAVING 보다는 WHERE 를 적극적으로 사용하는 것이 좋다. 특히 인덱스가 걸려있는 컬럼에는 group by 나 having 절을 사용하지 않는 것이 좋다. 이 경우 인덱스가 동작하지 않기 때문이다. 예를 들어, 'SELECT 필드 A, SUM(필드 B) FROM 테이블 GROUP BY 필드 A HAVING 필드 A=100' 이라는 문장은 'SELECT 필드 A, SUM(필드 B)

FROM 테이블 WHERE 필드 A=100 GROUP BY 필드 A'로 변경하는 것이 낫다. 이렇게 사용하면 WHERE 절에 의하여 먼저 인덱스가 동작하기 때문에 인덱스를 사용할 수 있게 된다.

12. WHERE 절에는 먼저 인덱스가 있는 컬럼을 기술한다. 특히 복합 인덱스인 경우 선행 인덱스를 먼저 사용한다. .
13. 인덱스를 사용한 검색과 보통 검색의 차이를 알아보도록 한다. 테이블의 15% 이상을 검색하는 경우에는 오히려 그냥 검색하는 것이 인덱스를 사용한 검색보다 빠르다는 것을 알 수 있다. 이런 경우일 때에는 SQL 문장을 인덱스를 사용하지 않도록 구성하는 것이 좋다. 꼭, 인덱스를 사용하는 경우가 빠른 것은 아니다. 그 이유는 보통 인덱스를 사용한 검색에서는 검색된 하나의 레코드마다 다중의 논리 연산이 가해지기 때문이다. 그러나, 보통 검색인 경우에는 하나의 논리적인 블록마다 연산되는 모든 행을 사용하기 때문에 테이블에 접근하는 내용이 많은 경우에는 일반적인 보통 검색이 좋다.
14. 인덱스 검색의 경우 ORDER BY 절을 적극 활용하도록 한다. 이 절은 인덱스된 컬럼에 대해 검색을 가능하게 한다.
15. 가장 중요한 것은 사용하는 데이터를 이해하고 있어야 한다는 것이다. 데이터의 분포도의 이해는 다시 한번 말하지만 가장 중요한 점이다. 오라클 7.3 의 경우 이러한 데이터 분포를 이해하기 위한 HOSTOGRAMS 라는 기능이 있다.
16. SQL 문장을 만들 경우에 참조하는 테이블의 수는 작을수록 좋다.
17. 어쩔 수 없이 많은 테이블을 JOIN 하는 경우에는 JOIN 의 순서가 매우 중요하다. 다시 한번 이야기하지만, 적은 수의 행(레코드)를 가진 테이블과 분포도가 풍부한 테이블이 JOIN 의 처음에 기술되어야 한다.
18. SQL 문장은 단순해야 한다. 한 페이지를 넘어가는 SQL 문장은 각 RDBMS 의 SQL 최적화기를 멍청(?)하게 만든다. 가끔은 단순한 것이 복잡한 것보다 빠를 수 있다. 결론은, 각각의 SQL 문장을 테스트 해보아야 한다는 것이다. 임시 테이블의 경우 많은 테이블을 포함하는 경우에는 SQL 의 JOIN 을 나누어서 수행하는 것이 좋다. 복잡한 JOIN 인 경우에는 2~3 개 정도의 SQL 문장이 나을 수도 있다.
19. 하나의 결과를 얻기 위해 여러 형태의 SQL 문장을 기술할 수 있다. 연산자의 사용에 주의해야 하는데, 특히 NOT IN 과 같은 문장은 인덱스가 있다 하더라도 인덱스를 사용지 않는 보통 검색 작업을 취한다. 그러므로, NOT IN(SELECT)나 NOT EXISTS 보다는 MINUS 산술연산을 사용하는 것이 인덱스를 사용하기 때문에 유리하다.
20. WHERE 절에 OR 보다는 UNION 을 사용하는 것이 좋다.
21. ROWID 나 ROWNUM 을 사용할 수 있다면 적극적으로 사용하는 것이 좋다. 다만 이 값들은 언제나 같지 않기 때문에 이용할 때에는 ROWID 의 값을 리턴하지 않는 것이 좋다. 행은 ROWNUM 을 사용하는데, WHERE 절에 ROWNUM < 100 을 사용하면 100 개 이상의 행을 전달하지 않는다.
22. CURSOR 의 사용에 주의하라. 함축적으로 만들어진 CURSOR 는 여분의 fetch 를 만들

어 낸다. DECLARE, OPEN, FETCH, CLOSE 문을 사용하면 개발자가 명시적으로 만드는 CURSOR 을 사용한다. 보통 DELETE, UPDATE, INSERT, SELECT 문을 사용하면 함축적인 CURSOR 을 사용하게 된다.

23. RDBMS 에 병렬 쿼리 옵션이 있다면 적극적으로 활용하라. 이 병렬 쿼리를 사용하면 보다 빠르게 SQL 문을 수행한다. 오라클 7 의 경우 전체 검색의 경우에만 병렬 쿼리가 동작한다. 그렇지만, 오라클 8 에서는 인덱스가 분할되어 있다면 indexed range 검색을 통한 쿼리도 병렬로 동작한다. 다만, 이 옵션은 다수의 디스크 드라이버나 SMP, MPP 시스템에서만 사용할 수 있다.
24. 네트워크 트래픽에 주의하라! 특히 한번에 처리되는 작업을 크게 하는 것이 빈번한 작업보다는 훨씬 좋다. 오라클의 경우 array processing 과 PL/SQL block 을 사용하면 SQL 문 하나로 많은 row 를 수행한다. INSERT 시에 배열을 사용하면 훨씬 빠른 작업을 할 수 있다.
25. 복잡한 SQL 문은 많은 네트워크 트래픽을 유도할 수 있다. 오라클의 경우 PL/SQL 의 블록내부에 존재한다면 전체 블록이 오라클 서버로 보내져 한번에 수행되고 결과를 빠르게 클라이언트에 전달합니다.

새롭게 변한 SQL 빌더의 사용

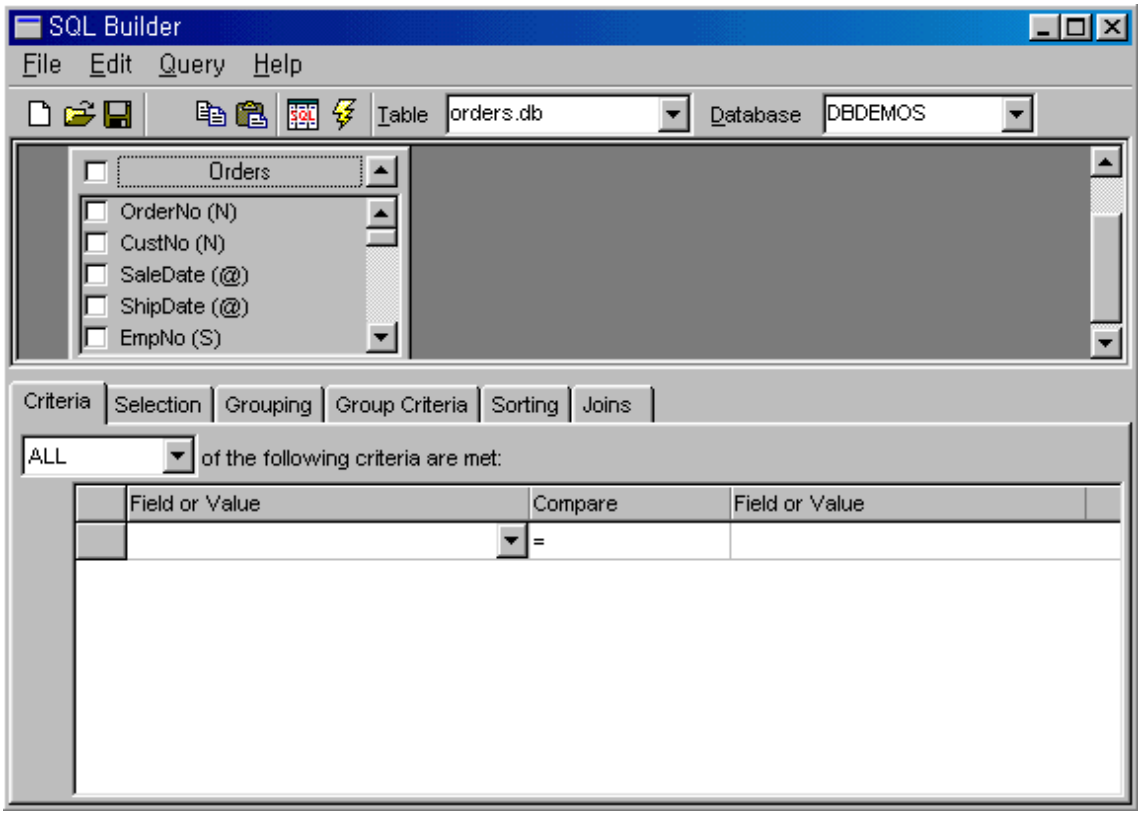
델파이 4 에서는 델파이 3 까지의 비주얼 쿼리 빌더(Visual Query Builder)가 한층 업그레이드된 SQL 빌더로 대체되었다. SQL 빌더를 처음 동작시키려면 디자인 시에 폼이나 데이터 모듈에 놓인 TQuery 컴포넌트를 선택하고 오른쪽 버튼을 클릭하면 컨텍스트 메뉴가 뜬다. 여기에서 필드 에디터와 SQL 탐색기, 그리고 SQL 빌더 등을 실행시킬 수 있다.

SQL 빌더를 이용해 시각적으로 대화식의 SQL 쿼리를 생성, 실행할 수 있다. 아주 복잡한 SQL 쿼리도 전문적인 지식 없이 간단하게 작성할 수 있는 것이 SQL 빌더의 장점이다. 심지어는 SQL 구문을 배우기 위한 하나의 도구로 사용할 수도 있을 정도이다. SQL 빌더를 이용해 SQL 쿼리의 결과를 보고, 이를 편집할 수 있다. SQL 빌더가 지원하는 쿼리는 기본적인 select 구문부터 계산 필드가 포함된 복잡한 다중 테이블의 join 을 비롯한 모든 구문을 만들어낼 수 있다.

또한, SQL 빌더를 이용하면 이종의 쿼리들을 사용할 수 있다. 즉, 서로 상이한 데이터베이스에 포함된 테이블들에 대한 쿼리가 가능하다는 말이다. 심지어 데이터베이스 서버가 다른 경우에도 작동한다. 이를 구현하기 위해서, 상이한 데이터베이스에 접근할 때 BDE 가 SQL 구문을 서버로 보내지 않고 로컬 SQL 로 처리하게 된다. 로컬 SQL 은 원래 dBASE 와 파라독스를 위해 지원되는 것인데, 쿼리의 해석과 데이터를 읽어오는 작업을 서버에 맡기지 않고 BDE 내부에서 처리하는 것이다.

- SQL 쿼리 텍스트 진입 윈도우 (SQL Query Text Entry Window)

SQL 쿼리 텍스트 진입 윈도우는 SQL 쿼리 구문을 텍스트로 입력하고 볼 수 있도록 지원되는 윈도우를 말한다. SQL 빌더에서 쿼리를 그래픽 인터페이스로 생성하면 이 윈도우를 통해 SQL 구문이 나타난다. 즉, SQL 빌더와 이 텍스트 입력 윈도우는 서로 연결되어 있다. 만약 텍스트 입력 윈도우에서 쿼리를 변경하면 그것이 SQL 빌더에도 나타난다. 이들을 서로 토글하기 위해서는 상단의 툴바 중에서 SQL 버튼을 클릭하면 된다.



- 쿼리 실행과 쿼리 결과 윈도우(Query Result Window)

F9 를 누르거나 툴바에서 번개 표시가 되어 있는 버튼을 누르면 쿼리가 실행된다.
이와 동시에 쿼리 결과 윈도우(Query Result Window)가 뜨고 그 결과가 그리드에 나타난다. 쿼리 결과 윈도우는 그리드 하나와 DBNavigator 하나로 구성된다.

- 테이블 pane

테이블 pane 은 SQL 구문에 사용된 각 테이블과 그 필드들을 표시하는 판이다. 특정 테이블을 불러오려면 SQL 빌더의 툴바에 나타난 테이블 드롭 다운 박스에서 한 테이블을 선택하면 된다. 물론 그에 앞서 Database 드롭 다운 박스에서 데이터베이스 앨리어스부터 먼

저 선택해야 한다.
테이블 pane 에 올려진 테이블을 변경하는 방법은 오른쪽 버튼을 클릭한 뒤, Edit Table Alias 메뉴를 선택하면 된다. 이렇게 하면, 테이블 이름이 편집가능 상태로 변하는데 여기에서 새로운 테이블 이름을 입력하면 된다.
테이블 왼쪽에 체크박스가 있는데, 이를 통해 어떤 필드를 쿼리의 select 문에 포함시키고, 어떤 필드를 포함시키지 않을 지를 지정할 수 있다. 파란색으로 체크된 필드만 쿼리 구문에 나타난다. 상단에 나타나는 테이블 이름에도 체크 박스가 있는데, 여기에 체크가 될 때에는 테이블 내의 모든 필드를 포함시킨다는 말이다. 이것을 끄면 기본적으로 모든 필드의 체크가 꺼진다. 이 상태에서 필요한 필드만 체크할 수 있다. 필요한 필드만 체크된 경우에는 테이블 이름의 체크박스에는 회색 체크가 나타난다.
테이블 pane 에서는 Join 된 필드끼리의 연결 상태도 보여주는데, 테이블 이름 사이의 가는 선이 연결 관계를 보여준다.
우상단에 있는 Minimize 버튼을 클릭하면 테이블 이름만 나타나고 필드들은 표시되지 않는다. 테이블끼리 조인 시키는 방법은 한 테이블의 필드를 드래그해서 다른 테이블의 필드 위에서 마우스 버튼을 떼는 것이다. 조인이 이루어지면 테이블끼리 굵은 선이 그어지며, from 절 안에 join 이 추가된다. 다음의 구문이 join 이 추가된 예이다.

```
select ... from INNER JOIN "PostCodeMaster.DB" PostcodeMaster
    ON  (PostcodeMaster.PostCode = Customermaster.PostCode),
    "CustomerClass.DB"   Customerclass
```

테이블 pane 은 SQL 빌더의 하단에 있는 Selection 탭과 연결해서 동작한다. 테이블 pane 에서 변경된 사항은 Selection 탭에 바로 적용되고, Selection 탭에서 변경된 사항은 테이블 pane 에 바로 적용된다.

- Where 절을 위한 Criteria 페이지

Criteria 페이지는 쿼리 결과 중에서 특정 Row 들만을 가져오기 위하여 where 절을 만드는 페이지이다. 여기에서의 결과를 Where 절에 추가시킨다. Join 문법과는 다르다는 것을 명심하자. 단지 Where 절을 위한 지원이다.

```
select ... from ...
where TableA.Field1 = TableB.Field1
```

여기에서 TableA.Field1 = TableB.Field1 과 같은 문장을 만들어내는 것이 Criteria 페이지의 역할이다. 이런 관계를 필요한 만큼 설정할 수 있다.

이 페이지의 상단에 보면 XXX of the following criteria met: 이라는 부분이 보인다. XXX 는 드롭 다운 콤보박스이다. 콤보박스에는 ALL, ANY, NONE, NOT ALL 중 하나를 선택할 수 있다. 'X = Y' 같은 criterition 이 A, B, C, ... 라고 가정하자. 이때, 각각의 의미는 다음과 같다.

ALL : A and B and C and ...
ANY : A or B or C or ...
NONE : not(A) or B or C or ...
NOTALL : not(A) and B and C and ...

criteria 는 표현식이나 EXISTS 절이 될 수 있다.

- Select 절을 위한 Selection 페이지

어떤 필드를 보여줄 것인가와 그 필드명이 어떻게 출력될 것인가를 지정해준다. 예를 들어,

```
select A, B, C, ... from ...
```

여기에서 A, B, C, ... 의 내용을 채우는 페이지다. Output Name 컬럼에 해당 필드의 출력명을 줄 수 있는데 한글도 사용할 수 있다. 이를 사용하면 DBGrid 등에서 한글 필드 명을 나타낼 수 있다. 필드의 추가는 드롭다운을 통해서 선택해도 되고 테이블 pane 에서 해당 필드를 드래그해서 빈 Row 에 놓으면 추가가 이루어진다. 또한 테이블 pane 에서 해당 필드를 드래그해 기존의 Row 에 놓으면 그 내용이 변경된다. 테이블 pane 과 Selection 페이지는 연결되어 있기 때문에, 한쪽의 내용을 변경하면 다른 쪽의 내용에 곧바로 적용된다.
상단에 Remove Duplicates 체크박스를 볼 수 있는데 이것을 클릭하면 동일한 Row 는 쿼리 결과에 나타나지 않는다. 즉, 이것을 SQL 문으로 표현하면,

```
select distinct A, B, C, ... from ...
```

과 같이 select 절에 distinct 키워드가 추가된다.
선택된 필드를 지우고자 하면, 오른쪽 버튼을 클릭한 뒤 Delete Row 메뉴를 선택하면 된다.
함수를 이용하고 싶은 경우에는 Field 컬럼에서 해당 필드를 함수로 묶어주면 된다.
예를 들어,

```
Sum(CustomerMaster.Count)
```

이렇게 하면 Summary 컬럼이 새로 생기면서,

SUM CustomerMaster.Count

와 같은 형태로 바뀐다. 직접 해보기 바란다.

- Group by 절을 위한 Grouping 페이지

group by 는 쿼리 결과에서 특정 그룹의 Row 들을 묶어 하나의 summary Row 를 얻기 위한 SQL 키워드이다. 좌측의 Output Fields 리스트 박스에는 select 절에서 지정된 모든 필드가 올라온다. 이 필드들 중에서 하나를 선택하여 Add 버튼을 누르면 Grouped On 리스트 박스로 옮겨간다. 반대로 Remove 버튼을 누르면 Grouped On 리스트 박스에서 Ouput Fields 리스트 박스로 옮겨간다.

select .. from ... where ... group by A, B, C, ...

Grouped On 에 추가된 필드는 A, B, C, ... 과 같이 group by 절에 나타난다. 이를 통해 A, B, C, ... 필드에 동일한 값을 가진 Row 는 하나의 Row 로 쿼리 결과에 나타나게 된다. 여기서 주의할 것은 문법상 select 절에는 sum 등의 함수나 group by 절에 이용된 필드만 올 수 있다. 다른 필드의 값은 그룹화된 Row 들마다 각기 다를 수가 있고 그것을 하나의 Row 로 표현하는 것은 불가능하기 때문이다.

- Having 절을 위한 Grouping criteria 페이지

Having 절은 한마디로 어떤 그룹을 선택할 것인지를 지정하는 절이다. 해당 조건을 만족하는 그룹만 group by 를 위해 사용된다. SUM, COUNT 등의 함수를 이용하거나 그냥 SQL 표현식을 사용할 수도 있다. 어떤 형태가 이용될 것인지는 오른쪽 버튼을 클릭하여 SQL Expression, Simple Having Summary Expression, Two Summary Expression 중에서 하나를 선택하는 것으로 지정할 수 있다.
SQL Expression 은 SQL 표현식을 직접 기입하는 것이다. 즉, 다음의 예와 같다.

SUM (Qty * Price) > 1000

Simple Having Summary Expression 은 두 필드를 비교해 요약하는 것이다. 한 개의 함수와 비교할 두 개의 필드, 비교할 연산자를 선택한다.
Two Summary Expression 은 두 개의 summary 표현식을 비교하는 것이다. 두 개의 함수

와 두 개의 필드 그리고 연산자를 선택한다.
여기서도 Selection 페이지와 마찬가지로 ALL, ANY, NONE, NONE, NOT ALL 중 하나를 선택할 수 있다.

- Order by 절을 위한 Sorting 페이지

Order by 절에 사용될 필드들을 선택하는 페이지이다. Output Fields 에는 select 절에 포함된 필드 명들이 리스트 박스로 나타난다. Add 버튼을 누르면 Sorted By 리스트 박스로 옮겨지며, 반대로 Remove 버튼을 누르면 Sorted By 리스트 박스에서 지워진다.
각 필드 별로 정순정렬과 역순정렬을 지정할 수 있는데, A..Z 를 선택하면 정순, Z..A 를 선택하면 역순정렬이 이루어진다. 만들어진 쿼리의 결과는 다음과 같다.

```
select ... from ... where ... order by A desc, B, C, ...
```

위에서 order by A desc, B, C, ... 부분이 Sorting 페이지를 통해서 추가되는 부분이다. Z..A 를 선택하여, 역순정렬이 사용된 필드이다.

- From 절에서의 join 을 위한 Joins 페이지

다중 테이블 SQL 쿼리를 위해 이용되는 페이지이다. 2 개의 필드와 그 필드의 비교 연산자를 통해 SQL 쿼리에 join 을 추가한다.
상단에 각 필드별로 한 개 씩의 Inculde Unmatched Records 를 볼 수 있는데 이것이 체크되면 outer join 이 이루어지고, 체크되지 않으면 inner join 이 이루어진다. 첫 번째 것만 체크되면 left outer join 이 쿼리에 추가되고, 두 번째 것만 체크되면 right outer join 이 쿼리에 추가된다. 둘 다 체크된 경우 full outer join 이 쿼리에 추가된다. 아무것도 체크되지 않으면 inner join 으로 처리된다.
SQL1 표준은 outer join 을 지원하지 않는다. 그러므로, 몇몇 구형 SQL 서버들은 outer join 이라는 개념이 아예 없는 경우가 있으므로 주의해야 한다.
테이블 pane 에서 필드를 드래그하여 다른 테이블 pane 의 특정 필드 위에 올려 놓으면 join 이 자동으로 추가된다. 이 페이지에서 새로운 join 을 추가하는 방법은 드롭-다운 콤보 박스 중 <create a new join>을 선택 한 뒤에 필드와 연산자를 선택하면 된다.
Join 을 없애는 방법은 그리드에서 오른쪽 버튼을 눌러 Delete Row 를 Row 가 모두 빌 때까지 하면 자동으로 없어진다.

유용한 SQL 문장들

여기에 소개하는 몇 가지 SQL 문장 사용에 대한 팁들은 Inprise 에서 제공하고 있는 TI 들을 참고한 것임을 미리 밝혀 둔다.

- 계산된 컬럼의 정렬 (Sorting on a Calculated Column)

데이터 세트의 계산 결과에 대해서, 이를 바탕으로 정렬이 필요한 경우가 있다. 델파이 어플리케이션에서 SQL 을 사용하는 경우에 어떻게 하면 되는지 간단히 소개한다.
파라독스나 디베이스와 같은 로컬 SQL 의 경우에는 계산 필드(calculated field)가 AS 키워드를 사용한다. 이를 이용하면 계산 필드가 ORDER BY 와 같은 문장을 이용해서 이 필드를 키로 정렬하도록 설정할 수가 있다. 예를 들어, 다음의 SQL 문장을 살펴 보자.

```
SELECT I."PARTNO", I."QTY", (I."QTY' * 100) AS TOTAL
FROM "ITEMS.DB" I ORDER BY TOTAL
```

이 문장에 의해 계산 필드는 TOTAL 로 명명되며, 이 컬럼의 이름을 이용하여 ORDER BY 구문을 사용하면 정렬이 가능하다.
그런데, Interbase 같은 경우 이런 방법이 적용되지 않는데 이는 이들이 계산 필드를 사용할 때, 이름으로 적용하지 않고 필드의 위치를 이용하기 때문이다. 예를 들어, 다음의 SQL 문장은 월급 순으로 EMPLOYEE 테이블을 정렬하게 된다.

```
SELECT EMP_NO, SALARY, (SALARY / 12) AS MONTHLY
FROM EMPLOYEE ORDER BY 3
```

이 방법은 로컬 데이터베이스에서도 적용된다. 그러므로, ORDER BY 구문 뒤에 처음부터 필드의 위치를 지정하면 이들을 서로 적용하기가 쉬워질 것이다.

- SUBSTRING 함수의 사용법

SQL 함수인 SUBSTRING 은 대단히 유용하게 사용할 수 있는 함수이다. 많은 수의 SQL 서버와 로컬 데이터베이스에서 지원하지만, Interbase 에서는 지원하지 않는다. 문법은 다음과 같다.

```
SUBSTRING(<column> FROM <start> [, FOR <length>])
```

<column>은 문자열을 추출할 컬럼을 지정하는 것이며, <start>에는 문자를 추출할 시작점을 <length>는 추출할 문자열의 길이를 지정한다.

예를 들어, 다음과 같은 문장은 COMPANY 라는 컬럼의 2 번째 문자에서부터 3 문자를 가져온다.

```
SUBSTRING(COMPANY FROM 2 FOR 3)
```

SUBSTRING 함수는 SELECT 문에 의해 필드 리스트를 지정하는 부분이나, WHERE 절에서 값을 비교할 때 유용하게 사용할 수 있다. 당연한 이야기 이지만 SUBSTRING 함수는 문자열 형의 컬럼에서만 사용할 수 있다.
다음 문장은 COMPANY 컬럼에서 앞의 3 자를 추출해서 SS 라는 계산 컬럼(calculated column)으로 지정하게 된다.

```
SELECT (SUBSTRING(C.”COMPANY” FROM 1 FOR 3)) AS SS
FROM “CUSTOMER.DB” C
```

다음 문장은 COMPANY 컬럼의 2~3 번째 문자가 ‘an’인 모든 레코드를 뽑는다.

```
SELECT C.”COMPANY” FROM “CUSTOMER.DB” C
WHERE SUBSTRING(C.”COMPANY” FROM 2 FOR 2) = “an”
```

Interbase 의 경우, SUBSTRING 함수와 비슷한 기능을 하는 SQL 문장을 작성하기 위해서는, LIKE 연산자를 사용할 수 있다. 다음 문장은 EMPLOYEE 테이블의 LAST_NAME 필드의 값의 2~3 번째 문자가 “an”인 모든 레코드를 뽑는다.

```
SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEE
WHERE LAST_NAME LIKE “_an%”
```

파라미터가 있는 질의의 작성

쿼리 문장에 변화를 줄 수 있도록 하기 위해서는 파라미터를 사용할 수 있는 SQL 문장을 이용해야 한다. 다음의 SQL 문장을 살펴 보자.

```
SELECT TEST.”FNAME”, TEST.”Salary of Employee” FROM TEST
WHERE TEST.”Salary of Employee” > :Val
```

이 문장에서 변수 이름으로 Val 을 설정한 것이다. 이제는 TQuery 객체의 Params 프로퍼티를 이용해서 이들 변수의 내용에 접근할 수 있다. 이 변수의 내용을 TEdit 박스를 이용

해서 입력하게 하려면 다음과 같이 하면 된다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Query1 do
  begin
    Close;
    ParamByName('Val').AsInteger := StrToInt(Edit1.Text);
    Open;
  end;
end;
```

LIKE 문을 같이 사용하면 더욱 유용하게 사용할 수 있다. 예를 들어 다음가 같은 SQL 문장이 있다고 하자.

```
SELECT * FROM CUSTOMER
WHERE Company LIKE :CompanyName
```

이 문장에서 CompanyName 이 파라미터 변수가 된다. 이를 이용하여 다음과 같이 코딩을 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Query1 do
  begin
    Close;
    ParamByName('CompanyName').AsString := Edit1.Text + '%';
    Open;
  end;
end;
```

참고로, ParamByName 을 사용하여 파라미터 변수 이름을 쓰는 것은, Params[Parameter Number]를 사용하는 것과 동일하다. 예를 들어, 앞의 코드에서 CompanyName 이 첫번째 파라미터라면 다음과 같이 쓸 수도 있다.

```
Params[0].AsString := Edit1.Text + '%';
```

캐쉬 업데이트(Cached update)의 활용

캐쉬 업데이트는 기본적으로 트랜잭션에 걸리는 시간과 네트워크 트래픽을 줄이려는 목적으로 사용된다. 이런 효과가 있지만, 모든 데이터베이스 어플리케이션에 적용하는 것은 좋지 않다. 캐쉬 업데이트의 사용을 결정할 때 고려해야할 사항은 크게 3 가지가 있다.

1. 로컬에 복사된 데이터를 편집하는 동안 다른 어플리케이션이 서버의 실제 데이터에 접근하여 데이터를 변경할 수 있다.
2. 로컬에 복사된 데이터의 편집된 사항이 서버에 적용되기 전에는 다른 어플리케이션이 변화된 사항을 알 수 없다.
3. 쿼리에 기초한 데이터 세트에 캐쉬 업데이트를 적용할 때에는 업데이트 객체를 필요로 한다.

- 캐쉬 업데이트 과정의 이해

캐쉬 업데이트를 수행하는 동안 실제로 다음과 같은 과정을 거치게 된다.

1. 캐쉬 업데이트를 가능하게 한다. 이렇게 하면 서버의 데이터를 읽기전용으로 가져오는 트랜잭션이 시작된다. 이 데이터의 로컬 복사본이 메모리에 저장되며 데이터 컨트롤에 디스플레이되고 편집이 가능해진다.
2. 레코드의 로컬 복사본이 편집된다. 이 때 원본 레코드와 편집된 동작이 메모리에 기록된다.
3. 사용자가 레코드를 스크롤할 때마다 읽기전용 트랜잭션을 이용해 필요한 레코드를 추가로 가져온다.
4. 편집된 레코드를 데이터베이스에 적용하거나 변화를 취소한다. 데이터베이스에 레코드가 기록될 때마다 OnUpdateRecord 이벤트가 발생한다. 이 때 에러가 발생하면 OnUpdateError 이벤트가 발생하여 에러를 수정하고, 업데이트를 계속할 수 있게 하는 방법을 제공한다. 업데이트가 끝나면 로컬 캐쉬에 저장되어 있던 변화사항이 제거된다.

- 업데이트 컴포넌트에 대한 SQL 문장 제작

업데이트 컴포넌트에 대한 SQL 문장을 제작하려면, 먼저 TUpdateSQL 컴포넌트를 데이터 모듈이나 폼에 위치시키고, 업데이트 컴포넌트의 이름을 데이터 세트의 UpdateObject 프로퍼티에 기록한다. 그리고, 업데이트 컴포넌트를 더블 클릭하면 업데이트 SQL 에디터를 띄울 수 있는데, 이 에디터에서 SQL 문장을 편집하면 된다.

업데이트 SQL 에디터에는 2 개의 페이지가 있다. 처음 에디터를 띄우면 Options 페이지를 볼 수 있다. Table Name 콤보 박스에서 업데이트할 테이블을 고른 후, Update Fields 리스트 박스에서 업데이트할 컬럼을 선택한다. 또한, Key Fields 리스트 박스에서는 업데이트시 키로 사용할 컬럼을 고른다. 파라독스, 디베이스, 폭스 프로의 경우에는 선택한 컬럼이 반드시 인덱스가 존재하는 컬럼이어야 한다. 만약 데이터베이스 서버가 필드 이름에 따옴표를 요구하면 Quote Field Names 체크 박스를 선택한다. 그리고 나서 Generate SQL 을 클릭하면 업데이트 컴포넌트의 ModifySQL, InsertSQL, DeleteSQL 프로퍼티와 연관된 SQL 문장이 생성된다.
만들어진 SQL 문장을 보고, 편집하려면 SQL 페이지를 선택한다.

- 이미 존재하는 레코드의 업데이트

이미 존재하는 레코드를 업데이트할 때에는 업데이트 객체의 ModifySQL 프로퍼티를 이용하여 SQL UPDATE 구문을 생성한다. 이 때 UpdateMode 프로퍼티를 설정하여 델파가 업데이트할 레코드를 찾는 방법을 지정할 수 있다. 다음 테이블에 UpdateMode 프로퍼티에 가능한 값을 나열하였다.

값	의 미
upWhereAll(디폴트)	UPDATE 문장의 WHERE 절에 지정된 값에 정확하게 해당되는 레코드만 업데이트 한다.
upWhereKeyOnly	지정된 키 컬럼에 기초한 레코드를 업데이트 한다.
upWhereChange	키 컬럼과 수정된 컬럼들만 업데이트 한다.

- 캐쉬 업데이트의 적용

데이터 세트가 캐쉬 업데이트 모드에 있으면 데이터의 변화가 어플리케이션에서 메소드를 호출하지 않으면 데이터베이스에 기록되지 않는다.

> 참고:
> SQL 쿼리에 의해 가져온 레코드 세트는 라이브 결과 세트(live result set)를 지원하지 않는다. 이 때 업데이트를 하려면 반드시 TUpdateSQL 객체를 사용해야 한다. 업데이트에 테이블의 결합(join)이 포함되면 반드시 각각의 테이블당 하나의 TUpdateSQL 객체를 지원해야 하며, OnUpdateRecord 이벤트 핸들러를 이용해 이 객체들이 업데이트할 수 있도록 해야 한다.

- 데이터 세트 컴포넌트 메소드를 사용한 캐쉬 업데이트의 적용

각각의 데이터 세트에서 ApplyUpdates 와 CommitUpdates 메소드를 사용하여 업데이트가 가능하다. 이렇게 데이터 세트 레벨에서 직접 캐쉬 업데이트를 하면 데이터베이스 트랜잭션을 이용해야 한다. 다음의 코드는 CustomerQuery 데이터 세트의 업데이트를 적용하는 예이다.

```
procedure ApplyButtonClick(Sender: TObject);
begin
  with CustomerQuery do
  begin
    Database1.StartTransaction;
    try
      if not IsSQLBased and (TransIsolation <> tiDirtyRead) then   //파라독스, 디베이스 ... ?}
          TransIsolation := tiDirtyRead;              //그렇다면 TransIsolation 은 tiDirtyRead!
      ApplyUpdates;                                   //데이터베이스에 업데이트 기록을 시도
      Database1.Commit;                               //성공하면 변화를 commit!
    except
      Database1.Rollback;                             //실패하면 롤백!
      raise;                                         //예외 발생으로 CommitUpdates 의 호출을 막는다.
    end;
    CommitUpdates;                                    //성공하면 내부 캐쉬의 내용을 지운다.
  end;
end;
```

ApplyUpdates 를 호출할 때 예외가 발생하면 데이터베이스 트랜잭션은 롤백된다. 위에서 raise 문장을 사용하는 것은 뒤에 나오는 CommitUpdates 문장의 호출을 막기 위한 것인데, CommitUpdates 가 호출되지 않으면 업데이트의 내부 캐쉬가 지워지지 않기 때문에 에러 상황에 대한 조작을 하고 업데이트를 다시 시도할 수 있다.

- 데이터베이스 컴포넌트 메소드를 이용한 캐쉬 업데이트

하나 이상의 데이터 세트에 대해 데이터베이스 컴포넌트의 ApplyUpdates 메소드를 이용해 캐쉬 업데이트의 적용이 가능하다. 다음 코드는 버튼 클릭시 업데이트 내용을 CustomersQuery 데이터 세트에 적용한다.

```
procedure ApplyButtonClick(Sender: TObject);
begin
```

```
  if not IsSQLBased and (TransIsolation <> tiDirtyRead) then
    TransIsolation := tiDirtyRead;
  Database1.ApplyUpdates([CustomersQuery]);
end;
```

이 메소드는 트랜잭션을 시작하고, 캐쉬 업데이트 내용을 데이터베이스에 기록한다. 성공적이면 트랜잭션을 commit 하고, 로컬 캐쉬의 업데이트 내용을 지운다. 성공적이지 못하면 롤백하고 캐쉬 업데이트의 내용을 변화시키지 않는다. 이 때 데이터 세트의 OnUpdateError 이벤트를 통해 에러 처리를 할 수 있다.
데이터베이스 컴포넌트의 ApplyUpdates 메소드는 데이터베이스와 연관된 여러 데이터 세트 컴포넌트를 업데이트할 수 있는 잇점이 있다. 다음 코드는 마스터/디테일 형의 2 개의 테이블을 업데이트 한다.

```
 if not IsSQLBased and (TransIsolation <> tiDirtyRead) then
   TransIsolation := tiDirtyRead;
 Database1.ApplyUpdates([MasterTable, DetailTable]);
```

- 마스터/디테일 테이블의 업데이트

마스터/디테일 테이블에 업데이트를 적용할 때에는 삭제된 레코드를 다룰 때만 제외하고는 먼저 마스터 테이블을 업데이트하고 디테일 테이블을 업데이트 한다. 복잡한 마스터/디테일 관계를 가지고 있는 경우에도 기본적인 원칙은 같다.
마스터/디테일 테이블을 업데이트할 때 데이터 세트와 데이터베이스 컴포넌트 레벨에서 업데이트하는 방법이 있다. 이때 보다 명확하게 조절을 하기 위해서는 데이터 세트 레벨에서 업데이트하는 것이 좋다. 다음 코드는 2 개의 마스터/디테일 테이블을 캐쉬 업데이트하는 예이다.

```
Database1.StartTransaction;
try
  Master.ApplyUpdates;
  Detail.ApplyUpdates;
  Database1.Commit;
except
  Database1.Rollback;
  raise;
end;
```

```
Master.CommitUpdates;
Detail.CommitUpdates;
```

- 캐쉬 업데이트의 취소

캐쉬 업데이트를 취소하는 방법은 다음의 3 가지가 있다.

1. 모든 업데이트를 취소하고, 캐쉬 업데이트를 사용하지 않으려면 CachedUpdates 프로퍼티를 False 로 설정한다.
2. 캐쉬 업데이트를 계속 사용하되, 지금까지의 업데이트를 취소하려면 CancelUpdates 메소드를 호출한다.
3. 현재 레코드에 가해진 업데이트를 취소하려면 RevertRecord 메소드를 호출한다.

- 캐쉬된 레코드의 되살리기

삭제된 레코드는 일단 캐쉬에 저장되기 때문에 이를 되살리는 방법이 있다. 이렇게 하려면 일단 UpdateRecordTypes 프로퍼티를 삭제된 레코드를 가리키도록 설정하고 RevertRecord 를 호출하면 된다. 다음 코드는 삭제된 레코드를 모두 되살린다.

```
procedure UndeleteAll(DataSet: TDataSet);
begin
  with DataSet do
  begin
    UpdateRecordTypes := [rtDeleted];
    try
      First;
      while not EOF do
          RevertRecord;
    finally
      UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
    end;
  end;
end;
```

- 업데이트 상태의 검사

업데이트 상태의 검사는 OnUpdateRecord 와 OnUpdateError 이벤트에서 가장 빈번하게 사용하게 된다. 이때 UpdateStatus 프로퍼티를 사용하게 되는데, 그 값은 다음과 같다.

값	의 미
usUnmodified	레코드의 변화가 없음
usModified	레코드가 수정 되었음
usInserted	새로운 레코드
usDeleted	레코드가 삭제 되었음

다음 코드는 레코드가 수정된 경우 UpdateStatus 프로퍼티를 이용하여 OnCalcFields 이벤트 핸들러에서 필드에 별표('*')를 보여주는 예제이다.

```
procedure TForm.CalcFields(DataSet: TDataSet);
begin
  if DataSet.UpdateStatus <> usUnmodified then
    Table1ModifiedRow.Value := '*'
  else
    Table1ModifiedRow.Value := Null;
end;
```

주의: 레코드의 UpdateStatus 가 usModified 이면 데이터 세트의 각 필드의 OldValue 프로퍼티를 이용하여 이전 값을 알아낼 수 있다.

- 필드의 OldValue, NewValue 프로퍼티의 사용

캐쉬 업데이트를 사용할 때 각 레코드의 원래 값은 TField 객체의 읽기전용 프로퍼티인 OldValue 에 담겨 있다. 바뀐 값은 TField 의 NewValue 프로퍼티에 저장된다. 이 값들은 OnUpdateError 와 OnUpdateRecord 이벤트 핸들러에서 유용하게 쓰일 수 있다.
이 값들을 이용해 에러를 일으킨 원인을 알아내고 이를 수정할 수 있다. 다음 코드는 월급(salary) 필드를 한번에 25% 이상 인상할 수 없도록 제한한다.

```
var
  SalaryDif: Integer;
  OldSalary: Integer;
begin
  OldSalary := EmpTabSalary.OldValue;
```

```
  SalaryDif := EmpTabSalary.NewValue - OldSalary;
  if SalaryDif / OldSalary > 0.25 then
  begin
    {인상 폭이 너무 크면 25% 까지 감소시킨다.}
    EmpTabSalary.NewValue := OldSalary + OldSalary * 0.25;
    UpdateAction := uaRetry;
  end
  else
    UpdateAction := uaSkip;
end;
```

앞의 예에서 NewValue 는 25% 인상으로 제한되고 이 값을 가지고 업데이트를 다시하게 된다.

- OnUpdateRecord 이벤트 핸들러의 제작

데이터 세트의 OnUpdateRecord 이벤트 핸들러의 기본 골격은 다음과 같다.

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  {업데이트 코드...}
end;
```

DataSet 파라미터는 업데이트할 데이터 세트를 가리키며, UpdateKind 파라미터는 업데이트의 type, UpdateAction 파라미터는 업데이트할 것인지를 결정한다. 디폴트 값은 uaFail 이다. 업데이트하는 동안 문제가 없으면 이벤트 핸들러에서 빠져 나가기 전에 이 값을 uaApplied 로 설정하고, 특정 레코드를 업데이트 하지 않으려면 uaSkip 으로 설정하면 캐쉬에 적용되지 않은 변화가 그대로 남아있게 된다. 이 파라미터들 외에 필드 컴포넌트의 OldValue 와 NewValue 프로퍼티가 유용하게 쓰일 수 있다.
주의할 점은, OnUpdateRecord 이벤트에서는 OnUpdateError, OnCalcFields 이벤트 핸들러와 마찬가지로 현재 레코드를 변화시키는 메소드를 호출하면 안된다는 것이다.
OnUpdateRecord 는 필요조건은 아니지만 보통 하나 이상의 업데이트 컴포넌트를 사용한다. 다음의 코드는 업데이트를 위해 테이블 컴포넌트를 이용한다.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
```

```
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue,
DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField', DataSet.Fields[0].OldValue, []) then
      case UpdateKind of
      ukModify:
        begin
          Edit;
          UpdateTable.Fields[1].Value := DataSet.Fields[1].Value;
          Post;
        end;
      ukDelete: DeleteRecord;
      end;
  UpdateAction := uaApplied;
end;
```

보통은 OnUpdateRecord 이벤트 핸들러에서 2 개 이상의 업데이트 컴포넌트를 사용한다.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  EmployeeUpdateSQL.Apply(UpdateKind);
  JobUpdateSQL.Appply(UpdateKind);
  UpdateAction := uaApplied;
end;
```

주의:
만약 데이터 세트의 UpdateObject 프로퍼티에 업데이트 컴포넌트를 지정하고, OnUpdateRecord 이벤트 핸들러를 제작하면 지정된 업데이트 컴포넌트는 이벤트 핸들러에서 Apply 메소드를 호출하지 않으면 동작하지 않는다.

- 캐쉬 업데이트 에러 처리

레코드가 처음 캐쉬되어 캐쉬 업데이트를 적용하려고 할 때 시간 차이가 있기 때문에 그동안 다른 어플리케이션에서 데이터베이스의 레코드가 바뀌었을 수가 있다. 이 때 BDE 는 업데이트를 적용할 때 이를 에러로 처리한다.
데이터 세트 컴포넌트의 OnUpdateError 이벤트는 이런 에러를 포함한 에러 처리를 가능하게 해준다. 만약 이 이벤트 핸들러를 작성하지 않으면 업데이트는 실패한다.
이때 주의할 점은 현재 레코드를 변화시키는 메소드(Next, Prior 등)을 호출하면 무한 루프에 빠지게 된다는 점이다.
OnUpdateError 이벤트 핸들러의 코드 골격은 다음과 같다.

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
        UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  {업데이트 코드를 적는다.}
end;
```

- 에러 메시지의 처리

E 파라미터는 보통 EDBEngineError type 이다. 에러 처리에서 여기서 에러 메시지를 알아낼 수 있다. 다음 코드는 라벨에 에러 메시지를 보여준다.

```
ErrorLabel.Caption := E.Message;
```

이 파라미터는 업데이트 에러의 원인을 알아내는 데에도 유용하다. 다음의 코드는 업데이트 에러가 키 위반(key violation)에 의한 것인지 알아보고 UpdateAction 파라미터를 uaSkip 으로 설정한다.

```
{실제 사용시 uses 절에 'Bde'를 추가해야 함}
if E is EDBEngineError then
  with EDBEngineError(E) do
  begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip  {키 위반이면 이 레코드를 건너 뛴다.}
    else
      UpdateAction := uaAbort {원인을 모르므로 업데이트를 취소 !}
 end;
```

정 리 (Summary)

이번 장에서는 클라이언트/서버 모델을 이용하여 데이터베이스 프로그래밍을 할 때 유용하게 사용될 수 있는 SQL 빌더의 사용 방법과 유용한 SQL 문장과 테크닉, 마지막으로 캐쉬 업데이트를 활용하여 프로그래밍을 하는 방법에 대해서 알아보았다.
다음 장에서는 실제 DBMS 의 종류와 그 특징 들에 대해서 알아보고, 이들의 접속 관리를 어떻게 할 것인지에 대한 내용을 알아보도록 할 것이다.