(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2014년07월03일
(11) 등록번호 10-1411083
(24) 등록일자 2014년06월17일

(51) 국제특허분류(Int. Cl.)
G06F 17/30 (2006.01)
(21) 출원번호 10-2008-7023105
(22) 출원일자(국제) 2007년03월22일
심사청구일자 2012년03월07일
(85) 번역문제출일자 2008년09월22일
(65) 공개번호 10-2009-0004881
(43) 공개일자 2009년01월12일
(86) 국제출원번호 PCT/US2007/007261
(87) 국제공개번호 WO 2007/112009
국제공개일자 2007년10월04일
(30) 우선권주장
11/725,206 2007년03월16일 미국(US)
60/785,672 2006년03월23일 미국(US)
(56) 선행기술조사문헌
US20050050068 A1*
*는 심사관에 의하여 인용된 문헌

(73) 특허권자
마이크로소프트 코포레이션
미국 워싱턴주 (우편번호 : 98052) 레드몬드 원 마이크로소프트 웨이
(72) 발명자
아디아, 아툴
미국 98052-6399 워싱턴주 레드몬드 원 마이크로소프트 웨이
블라켈리, 호세, 에이.
미국 98052-6399 워싱턴주 레드몬드 원 마이크로소프트 웨이
(뒷면에 계속)
(74) 대리인
제일특허법인

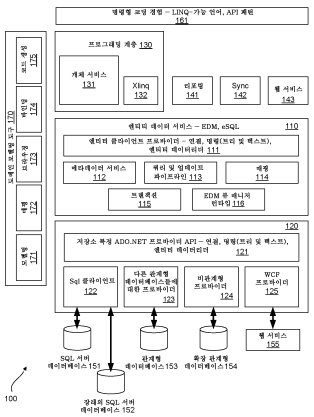
전체 청구항 수 : 총 10 항 심사관 : 이복현

(54) 발명의 명칭 **점진적 뷰 관리를 갖는 매핑 아키텍처**

(57) 요 약

애플리케이션에 의해 이용될 수 있는 데이터를 데이터베이스에서 지속되는 데이터에 매핑하기 위한 매핑 아키텍처를 포함하는 데이터 액세스 아키텍처가 제공된다. 이 매핑 아키텍처는 2가지 타입의 매핑 뷰(mapping views) — 쿼리들을 변환하는 데 도움이 되는 쿼리 뷰(query view) 및 업데이트들을 변환하는 데 도움이 되는 업데이트 뷰(update view) — 를 이용한다. 애플리케이션과 데이터베이스 간에 데이터를 변환하기 위해 점진적 뷰 관리(incremental view maintenance)가 이용될 수 있다.

대 표 도 - 도1



(72) 발명자

라슨, 퍼아케

미국 98052-6399 워싱턴주 레드몬드 원 마이크로소프트 웨이

멜니크, 세르게이

미국 98052-6399 워싱턴주 레드몬드 원 마이크로소프트 웨이

특허청구의 범위

청구항 1

애플리케이션에 데이터 서비스를 제공하는 방법으로서,

엔티티 스키마(entity schema)와 연관된 엔티티를 포함하는 엔티티 관계형 도메인(entity-relationship domain)의 개념을 포함하는 엔티티 데이터 모델(Entity Data Model, EDM)을 제공하는 단계;

선언적 언어(declarative language)를 사용하여 상기 엔티티 스키마 및 데이터베이스와 연관된 관계형 데이터베이스 스키마 사이의 양방향 매핑(bidirectional mapping)을 제공하는 단계 - 상기 매핑은 상기 엔티티 스키마 및 상기 관계형 데이터베이스 스키마에 대한 쿼리(query)의 형태로 표현됨 - ;

매핑 컴파일러를 사용하여 상기 엔티티 스키마, 상기 관계형 데이터베이스 스키마, 및 상기 매핑을 컴파일하여 쿼리 뷰(query view) 및 업데이트 뷰(update view)를 생성하는 단계 - 상기 쿼리 뷰는 상기 데이터베이스의 테이블의 형태를 사용해 상기 엔티티를 쿼리로서 나타내고, 상기 업데이트 뷰는 엔티티들의 형태를 사용해 상기 데이터베이스의 테이블을 나타내며, 상기 매핑 컴파일러는 모든 엔티티 데이터가 상기 데이터베이스로부터 손실 없이 유지(persist) 및 재생(reassemble) 가능하도록 보증함 - ;

사용자 쿼리 내의 쿼리 뷰를 언폴딩(unfolding)함으로써 데이터에 대한 액세스를 요청하는 상기 애플리케이션을 대신하여 상기 데이터베이스에 질의하는 것에 의해, 상기 엔티티 스키마를 대상으로 하는 상기 사용자 쿼리를 변환하는 단계; 및

상기 요청하는 애플리케이션을 대신하여 상기 데이터베이스를 업데이트하기 위해 업데이트 뷰에 뷰 관리 알고리즘을 적용하는 단계

를 포함하는, 애플리케이션에 데이터 서비스를 제공하는 방법.

청구항 2

제1항에 있어서,

상기 요청하는 애플리케이션으로부터, 프로그래밍 언어의 개체(object)를 수신하는 단계 - 상기 프로그래밍 언어의 개체는 상기 데이터베이스의 업데이트에 이용하기 위한 데이터를 포함함 - ;

상기 요청하는 애플리케이션으로부터, 생성(create), 삽입(insert), 업데이트(update), 또는 삭제(delete) 명령을 수신하는 단계 - 상기 생성, 삽입, 업데이트, 또는 삭제 명령은 상기 데이터베이스의 업데이트에 사용되는 데이터를 포함함 - ;

상기 요청하는 애플리케이션으로부터, 데이터 조작 언어(Data Manipulation Language, DML)의 식(expression)을 수신하는 단계 - 상기 식은 상기 데이터베이스의 업데이트에 사용되는 데이터를 포함함 -

중 적어도 하나를 포함하는

애플리케이션에 데이터 서비스를 제공하는 방법.

청구항 3

제2항에 있어서,

상기 생성, 삽입, 업데이트, 또는 삭제 명령을 수신하고, 상기 업데이트 뷰에 상기 뷰 관리 알고리즘을 적용하는 상기 단계는 상기 업데이트 뷰에 대한 델타 식을 생성하며, 상기 델타 식은 상기 업데이트 뷰 상에 점진적 업데이트(incremental update)를 실행하기 위해 필요한 식을 나타내며, 뷰 언폴딩을 사용하여 상기 델타 식을 쿼리 뷰와 조합하기 위해 사용하는 단계를 더 포함하는

애플리케이션에 데이터 서비스를 제공하는 방법.

청구항 4

삭제

청구항 5

삭제

청구항 6

삭제

청구항 7

삭제

청구항 8

제1항에 있어서,

상기 엔티티 스키마는 클래스(classes), 관계(relationships), 상속(inheritance), 집계(aggregation), 및 복합 타입(complex types)을 지원하는

애플리케이션에 데이터 서비스를 제공하는 방법.

청구항 9

삭제

청구항 10

삭제

청구항 11

애플리케이션에 데이터 서비스를 제공하기 위한 방법을 구현하는 컴퓨터에 의해 실행가능한 복수의 컴퓨터 실행가능 명령어들이 저장된 컴퓨터 판독가능 저장 매체로서, 상기 방법은,

엔티티 스키마와 연관된 엔티티를 포함하는 엔티티 관계형 도메인의 개념을 포함하는 엔티티 데이터 모델(EDM)을 제공하는 단계;

선언적 언어를 사용하여 상기 엔티티 스키마 및 데이터베이스와 연관된 관계형 데이터베이스 스키마 사이의 양방향 매핑을 제공하는 단계 - 상기 매핑은 상기 엔티티 스키마 및 상기 관계형 데이터베이스 스키마에 대한 쿼리의 형태로 표현됨 - ;

매핑 컴파일러를 사용하여 상기 엔티티 스키마, 상기 관계형 데이터베이스 스키마, 및 상기 매핑을 컴파일하여 쿼리 뷰 및 업데이트 뷰를 생성하는 단계 - 상기 쿼리 뷰는 상기 데이터베이스의 테이블의 형태를 사용해 상기 엔티티를 쿼리로서 나타내고, 상기 업데이트 뷰는 엔티티들의 형태를 사용해 상기 데이터베이스의 테이블을 나타내며, 상기 매핑 컴파일러는 모든 엔티티 데이터가 상기 데이터베이스로부터 손실 없이 유지 및 재생 가능하도록 보증함 - ;

사용자 쿼리 내의 쿼리 뷰를 언폴딩함으로써 데이터에 대한 액세스를 요청하는 상기 애플리케이션을 대신하여 상기 데이터베이스에 질의하는 것에 의해, 상기 엔티티 스키마를 대상으로 하는 상기 사용자 쿼리를 변환하는 단계; 및

상기 요청하는 애플리케이션을 대신하여 상기 데이터베이스를 업데이트하기 위해 업데이트 뷰에 뷰 관리 알고리즘을 적용하는 단계

를 포함하는

컴퓨터 판독가능 저장 매체.

청구항 12

제11항에 있어서,

상기 방법은, 상기 요청하는 애플리케이션으로부터 프로그래밍 언어의 개체를 수신하는 단계를 더 포함하고, 상

기 프로그래밍 언어의 개체는 상기 데이터베이스의 업데이트에 이용하기 위한 데이터를 포함하는

컴퓨터 판독가능 저장 매체.

청구항 13

제11항에 있어서,

상기 방법은, 상기 요청하는 애플리케이션으로부터 생성(create), 삽입(insert), 업데이트(update), 또는 삭제(delete) 명령을 수신하는 단계를 더 포함하고, 상기 생성, 삽입, 업데이트, 또는 삭제 명령은 상기 데이터베이스의 업데이트에 이용하기 위한 데이터를 포함하는

컴퓨터 판독가능 저장 매체.

청구항 14

제11항에 있어서,

상기 방법은, 상기 요청하는 애플리케이션으로부터 데이터 조작 언어(Data Manipulation Language; DML)의 식(expression)을 수신하는 단계를 더 포함하고, 상기 식은 상기 데이터베이스의 업데이트에 이용하기 위한 데이터를 포함하는

컴퓨터 판독가능 저장 매체.

청구항 15

삭제

청구항 16

제11항에 있어서,

상기 업데이트 뷰에 뷰 관리 알고리즘을 적용하는 것은 상기 업데이트 뷰에 대한 델타 식(delta expression)을 생성하고, 상기 델타 식은 상기 업데이트 뷰 상에 점진적 업데이트를 실행하는데 필요한 식을 나타내며,

상기 방법은, 상기 델타 식을 쿼리 뷰와 조합하기 위해 뷰 언폴딩(view unfolding)을 이용하는 단계를 더 포함하는

컴퓨터 판독가능 저장 매체.

청구항 17

삭제

청구항 18

제11항에 있어서,

상기 엔티티 스키마는 클래스(classes), 관계(relationships), 상속(inheritance), 집계(aggregation), 및 복합 타입(complex types)을 지원하는

컴퓨터 판독가능 저장 매체.

청구항 19

삭제

청구항 20

삭제

명 세 서

배 경 기 술

[0001] 브리징(bridging) 애플리케이션 및 데이터베이스는 장기간 계속되는 문제이다. 1996년에, 캐리(Carey)와 드윗(Dewitt)은 개체 지향 데이터베이스(object-oriented databases) 및 지속성 프로그래밍 언어(persistent programming languages)를 포함하는 많은 기술들이 쿼리 및 업데이트 처리, 트랜잭션 처리량, 및 확장성(scalability)에서의 제한으로 인해 널리 용인되지 않은 이유를 개설(outline)하였다. 그들은 2006년에는 개체-관계형(object-relational; O/R) 데이터베이스들이 지배할 것이라고 추측하였다. 사실, DB2® 및 Oracle® 데이터베이스 시스템들은 종래의 관계형 엔진 위에 하드와이어드 O/R 매핑을 이용하는 기본 제공 개체 계층(built-in object layer)을 포함한다. 그러나, 이들 시스템에 의해 제공되는 O/R 특징들은 멀티미디어 및 공간 데이터(spaial data) 타입들을 제외하고는 기업 데이터(enterprise data)를 저장하는 데 거의 이용되지 않는 것으로 보인다. 그 이유들 중에는 데이터 및 벤더 독립성, 레거시 데이터를 이동시키는 비용, 비즈니스 로직이 중간 계층(middle tier) 대신에 데이터베이스 내에서 실행하는 경우 스케일-아웃(scale-out)의 어려움, 및 프로그래밍 언어와의 불충분한 통합 등이 있다.

[0002] 1990년대 중반 이후, 인터넷 애플리케이션들의 성장에 자극을 받아, 클라이언트 측 데이터 매핑 계층들(client-side data mapping layers)이 인기를 얻었다. 그러한 계층의 핵심 기능은 명시적 매핑에 의해 구동되는, 애플리케이션의 데이터 모델과 밀접하게 정렬된 데이터 모델을 익스포즈(expose)하는 업데이트 가능한 뷰(updatable view)를 제공하는 것이다. 이들 기능들을 제공하기 위해 많은 상업 제품들 및 오픈 소스 프로젝트들이 출현하였다. 사실상 모든 기업 프레임워크가 클라이언트 측 지속성 계층(client-side persistence layer)(예컨대, J2EE 내의 EJB)을 제공한다. ERP 및 CRM 애플리케이션 등의 대부분의 패키징된 비즈니스 애플리케이션들은 독점적 데이터 액세스 인터페이스(예컨대, SAP R/3 내의 BAPI)를 통합한다.

[0003] 하나의 널리 이용되는 오픈 소스인 Java® 용의 개체-관계형 매핑(Object-Relational Mapping; ORM) 프레임워크는 Hibernate®이다. 그것은 다수의 상속 매핑 시나리오, 낙관적인 동시성 제어(optimistic concurrency control), 및 포괄적인(comprehensive) 개체 서비스를 지원한다. Hibernate의 최신 릴리스는 자바 지속성 쿼리 언어(Java Persistence Query Language)를 포함하는 EJB 3.0 표준을 준수한다. 상업 측면에서, 인기있는 ORM들은 Oracle TopLink® 및 LLBLGen®을 포함한다. 후자는 .NET 플랫폼 상에서 실행된다. 이들 및 그 밖의 ORM들은 그들의 대상 프로그래밍 언어들의 개체 모델들과 단단히 결합된다.

[0004] BEA®는 최근에 AquaLogic Data Services Platform®(ALDSP)이라고 불리는 새로운 미들웨어 제품을 도입하였다. 그것은 애플리케이션 데이터를 모델링하기 위해 XML 스키마를 이용한다. XML 데이터는 데이터베이스들 및 웹 서비스들로부터 XQuery를 이용하여 어셈블된다. ALDSP의 런타임은 복수의 데이터 소스들 상에서의 쿼리들을 지원하고 클라이언트 측 쿼리 최적화(client-side query optimization)를 수행한다. 업데이트들은 XQuery 뷰들 상의 뷰 업데이트들로서 수행된다. 만일 업데이트가 고유 변환(unique traslation)을 갖고 있지 않다면, 개발자는 명령형 코드(imperative code)를 이용하여 업데이트 로직을 무효화(override)할 필요가 있다. ALDSP의 프로그래밍 표면은 서비스 데이터 개체(service data objects; SDO)에 기초한다.

[0005] 오늘날의 클라이언트 측 매핑 계층들은 매우 다양한 정도의 성능, 강건성(robustness), 및 소유권의 총 비용을 제공한다. 통상적으로, ORM들에 의해 이용되는 애플리케이션 및 데이터베이스 아티팩트들 간의 매핑은 막연한 의미를 갖고 케이스별 추리(case-by-case reasoning)를 구동한다. 시나리오-구동 구현은 지원되는 매핑들의 범위를 제한하고 종종 연장하기 곤란한 연약한 런타임(fragile runtime)을 초래한다. 극소수의 데이터 액세스 해법들이 데이터베이스 커뮤니티에 의해 개발된 데이터 변환 기법들을 이용하고, 종종 쿼리 및 업데이트 변환을 위한 특별 해법들(ad hoc solutions)에 의지한다.

[0006] 데이터베이스 연구는 지속성 계층들을 구축하기 위해 이용될 수 있는 많은 강력한 기법들을 제안하였다. 그럼에도 불구하고, 상당한 갭들이 존재한다. 가장 결정적인 것들 중에는 매핑을 통하여 업데이트들을 지원하는 것이 있다. 쿼리들에 비하여, 업데이트들이 다루기가 훨씬 더 어렵다. 왜냐하면 그것들은 매핑들에 걸쳐서 데이터 일관성을 유지할 필요가 있고, 비즈니스 룰을 트리거할 수도 있고, 등등의 이유가 있기 때문이다. 그 결과, 상업 데이터베이스 시스템들 및 데이터 액세스 제품들은 업데이트 가능한 뷰에 대한 매우 제한된 지원을 제공한다. 근래에, 연구원들은 양방향 변환(bidirectional transformations) 등의 대안적인 접근법들에 관심을 돌렸다.

[0007] 전통적으로, 개념적 모델링(conceptual modeling)은 데이터 베이스 및 애플리케이션 디자인, 리버스-엔지니어링(reverse-engineering), 및 스키마 변환(schema translation)에 제한되었다. 많은 디자인 도구들은 UML을 이용한다. 매우 최근의 개념적 모델링만이 산업-강도 데이터 매핑 해법들에 진입하기 시작하였다. 예를 들면, ALDSP 및 EJB 3.0. ALDSP 양쪽 모두에서의 엔티티들 및 관계 표면들(relationships surfaces)의 개념은 복잡

타입(complex-typed) XML 데이터의 위에 E-R-스타일 관계들을 오버레이하는 반면, EJB 3.0은 클래스 주석(class annotations)을 이용하여 개체들 간의 관계들을 특정하는 것을 허용한다.

[0008] 스키마 매핑 기법들은, Microsoft® BizTalk Server®, IBM® Rational Data Architect®, 및 ETL® 도구 등의 다수의 데이터 통합 제품들에서 이용된다. 이들 제품들은 개발자들이 데이터 변환들을 디자인하거나 또는 매핑들로부터 그것들을 컴파일하여 전자 상거래 메시지들을 변환하거나 데이터 웨어하우스(data warehouses)를 로딩하는 것을 허용한다.

[0009] <발명의 요약>

[0010] 애플리케이션에 의해 이용될 수 있는 데이터를 데이터베이스에서 지속되는 데이터에 매핑하기 위한 매핑 아키텍처를 포함하는 데이터 액세스 아키텍처의 구현 및 사용을 위한 시스템, 방법, 및 컴퓨터 판독가능 매체가 제공된다. 일 실시예에서, 상기 매핑 아키텍처는 2가지 타입의 매핑 뷰(mapping views) — 쿼리들을 변환하는 데 도움이 되는 쿼리 뷰(query view) 및 업데이트들을 변환하는 데 도움이 되는 업데이트 뷰(update view) — 를 이용한다. 애플리케이션과 데이터베이스 간에 데이터를 변환하기 위해 점진적 뷰 관리(incremental view maintenance)가 이용될 수 있다. 아래에서는 또 다른 양태들 및 실시예들이 설명된다.

발명의 상세한 설명

[0030] **신규 데이터 액세스 구조**

[0031] 일 실시예에서는, 혁신이 구현되어 이 섹션에서 설명되는 신규 데이터 액세스 아키텍처 — "엔티티 프레임워크(Entity Framework)" — 의 양태들을 통합할 수 있다. 그러한 엔티티 프레임워크의 일례는 MICROSOFT® 코퍼레이션에 의해 개발된 ADO.NET vNEXT® 데이터 액세스 아키텍처이다. 다음은 본 발명을 실시하는 데 필요한 고려되어야 하는 다수의 구현 특정 상세들(implementation-specific details)과 함께 ADO.NET vNEXT 데이터 액세스 아키텍처에 대한 일반적인 설명이다.

[0032] 개관(OVERVIEW)

[0033] 전통적인 클라이언트-서버 애플리케이션들은 그들의 데이터에 대한 쿼리 및 지속 연산(persistence operations)을 데이터베이스 시스템들에 위임한다. 데이터베이스 시스템은 행(rows) 및 테이블의 형태로 데이터에 작용하는 반면, 애플리케이션은 보다 고레벨의 프로그래밍 언어 구성들(constructs)(클래스, 구조 등)에 의하여 데이터에 작용한다. 애플리케이션과 데이터베이스 계층 간의 데이터 조작 서비스에서의 *임피던스 부정합(impedence mismatch)*은 전통적인 시스템들에서도 문제가 되었다. 서비스 지향 아키텍처(service-oriented architectures; SOA), 애플리케이션 서버 및 다중 계층(multi-tier) 애플리케이션의 출현으로, 프로그래밍 환경과 잘 통합되고 어떠한 계층에서도 동작할 수 있는 데이터 액세스 및 조작 서비스에 대한 요구가 엄청나게 증가하였다.

[0034] 마이크로소프트의 ADO.NET 엔티티 프레임워크는 추상화의 레벨을 관계형 레벨로부터 개념적(엔티티) 레벨로 끌어올림으로써, 애플리케이션들 및 데이터 중심(data-centric) 서비스들에 대한 임피던스 부정합을 현저히 감소시키는 데이터에 대한 프로그래밍을 위한 플랫폼이다. 아래에서는 엔티티 프레임워크, 전체 시스템 아키텍처, 및 기초가 되는(underlying) 기술들의 양태들에 대하여 설명한다.

[0035] 서론(INTRODUCTION)

[0036] 현대의 애플리케이션들은 모든 계층들에서 데이터 관리 서비스를 필요로 한다. 그것들은 구조화된(structured) 비즈니스 데이터(고객(Customers) 및 오더(Orders) 등)뿐만 아니라, 이메일, 캘린더, 파일, 및 문서 등의 반구조화된(semi-structured) 및 구조화되지 않은 콘텐츠를 포함하는 점점 더 풍부한 형태의 데이터를 처리할 필요가 있다. 이들 애플리케이션들은 복수의 데이터 소스들로부터의 데이터를 통합하는 것은 물론 보다 기민한 판정 수행 프로세스를 가능하게 하기 위해 이 데이터를 수집, 정화(cleanse), 변환 및 저장할 필요가 있다. 이들 애플리케이션들의 개발자들은 그들의 생산성을 증대시키기 위해 데이터 액세스, 프로그래밍 및 개발 도구들을 필요로 한다. 관계형 데이터베이스는 대부분의 구조화된 데이터에 대한 사실상의 저장소가 되었지만, 그러한 데이터베이스들에 의해 익스포즈(expose)되는 데이터 모델(및 성능들)과, 애플리케이션들이 필요로 하는 모델링 성능들 간에 부정합 — 잘 알려진 *임피던스 부정합* 문제 — 이 존재하기 쉽다.

[0037] 기업 시스템 디자인에서는 2개의 다른 인자들이 또한 중요한 역할을 한다. 첫째로, 애플리케이션들에 대한 데이터 표현은 기초가 되는 데이터베이스들의 데이터 표현과 상이하게 진화하는 경향이 있다. 둘째로, 많은 시스템들은 상이한 정도의 성능을 갖는 전혀 다른 데이터베이스 백엔드들(back-ends)로 구성된다. 중간 계층(mid-tier) 내의 애플리케이션 로직은 이들 차이를 조정하고 보다 균일한 데이터의 뷰를 제공하는 데이터 변환들의 책임이 있다. 이들 데이터 변환은 빠르게 복잡해진다. 특히 기초가 되는 데이터가 업데이트 가능할 필요가 있을 경우, 그것들을 구현하는 것은 어려운 문제이고 애플리케이션에 복잡성을 추가한다. 애플리케이션 개발의 상당한 부분 – 일부 경우에는 40%까지 – 이 이들 문제를 그럭저럭 극복하는 사용자 지정(custom) 데이터 액세스 로직을 작성하는 데 바쳐진다.

[0038] 데이터 중심 서비스들에 대해서도 동일한 문제가 존재하고 못지않게 심각하다. 쿼리, 업데이트, 및 트랜잭션 등의 전통적인 서비스들은 논리적 스키마(관계형) 레벨에서 구현되었다. 그러나, 복제 및 분석 등의 보다 새로운 서비스들의 대부분은 전형적으로 보다 고레벨의 개념적 데이터 모델과 관련된 아티펙트들에 최적으로 작용한다. 예를 들면, SQL SERVER® 리플리케이션(Replication)은 제한된 형태의 엔티티를 나타내기 위해 "논리적 레코드(logical record)"라 불리는 구조를 창안하였다. 유사하게, SQL 서버 리포팅 서비스(Server Reporting Services)는 의미 데이터 모델 언어(semantic data model language; SDML)라 불리는 엔티티 같은(entity-like) 데이터 모델 위에 리포트들을 구축한다. 이들 서비스의 각각은 개념적 *엔티티들*을 정의하고 그것들을 관계형 테이블들에 매핑하는 사용자 지정 도구들을 갖고 있다 – 따라서 고객(Customer) 엔티티가 정의되어 한쪽으로는 복제를 위해, 다른 쪽으로는 리포트 구축을 위해, 또 다른 쪽으로는 다른 분석 서비스를 위해 등등을 위해 매핑될 필요가 있을 것이다. 애플리케이션들과 마찬가지로, 각 서비스는 전형적으로 이 문제에 대한 사용자 지정 해법을 구축하는 것으로 끝나고, 결과적으로, 이들 서비스들 사이에는 코드 복제 및 제한된 상호 운용이 존재한다.

[0039] HIBERNATE® 및 ORACLE TOPLINK® 등의 개체-관계형 매핑(object-to-relational mapping; ORM) 기술들은 사용자 지정 데이터 액세스 로직에 대한 인기있는 대안이다. 데이터베이스와 애플리케이션들 간의 매핑들은 사용자 지정 구조(custom structure)로, 또는 스키마 주석들을 통하여 표현된다. 이들 사용자 지정 구조들은 개념적 모델과 유사해 보일 수 있지만, 애플리케이션들은 이 개념적 모델에 대하여 직접 프로그램할 수 없다. 그 매핑들은 데이터베이스와 애플리케이션 간의 독립성의 정도를 제공하지만, 동일한 데이터의 약간 상이한 뷰들을 가지고 복수의 애플리케이션들을 취급하는 문제(예컨대, 고객 엔티티의 상이한 *프로젝션들(projections)*을 보기를 원하는 2개의 애플리케이션을 생각해보자), 또는 보다 동적인 경향이 있는 서비스들의 필요의 문제(기초가 되는 데이터베이스가 보다 빠르게 진화할 수 있으므로, *선험적(a priori)* 클래스 생성 기법들은 데이터 서비스들에 대하여 잘 작용하지 않는다)는 이들 해법들에 의해 잘 처리되지 않는다.

[0040] ADO.NET 엔티트 프레임워크는 애플리케이션들 및 데이터 중심 서비스들에 대한 임피던스 부정합을 현저히 감소시키는 데이터에 대한 프로그래밍을 위한 플랫폼이다. 그것은 적어도 다음의 점들에서 다른 시스템들 및 해법들과 다르다.

[0041] 1. 엔티티 프레임워크는 풍부한 개념적 데이터 모델(엔티티 데이터 모델, 즉 EDM), 및 이 모델의 인스턴스들에 작용하는 새로운 데이터 조작 언어(엔티티 SQL)을 정의한다. SQL과 마찬가지로, EDM은 값 기반(value-based)이다. 즉, EDM은 엔티티들의 거동(behaviors)(또는 방법)이 아니라 그의 구조적 양태들을 정의한다.

[0042] 2. 이 모델은 쿼리들 및 업데이트들에 대한 강력한 양방향(EDM – 관계형) 매핑들을 지원하는 미들웨어 매핑 엔진을 포함하는 런타임에 의해 *구체화(concrete)*된다.

[0043] 3. 애플리케이션들 및 서비스들은 값 기반 개념적 계층에 대하여, 또는 ORM 같은 기능을 제공하는 개념적(엔티티) 추상화 위에 계층화될 수 있는 프로그래밍 언어 특정 개체 추상화(programming-language-specific object abstractions)에 대하여 직접 프로그램할 수 있다. 값 기반 EDM 개념적 추상화는 개체들보다 애플리케이션들 및 데이터 중심 서비스들 사이에서 데이터를 공유하기 위한 보다 유연성 있는 기반(basis)이라고 생각된다.

[0044] 4. 마지막으로, 엔티티 프레임워크는 애플리케이션들에 대한 임피던스 부정합을 더 감소시키기 위해, 그리고 일부 시나리오에서는 완전히 제거하기 위해 쿼리 표현들을 이용하여 내재적으로 프로그래밍 언어들을 확장하는 마이크로소프트의 새로운 언어 통합 쿼리(Language Integrated Query; LINQ)를 이용한다.

[0045] ADO.NET 엔티티 프레임워크는 Microsoft.NET 프레임워크 등의 보다 큰 프레임워크에 통합될 수 있다.

[0046] 데이터 액세스 아키텍처에 대한 본 설명의 나머지는, ADO.NET 엔티티 프레임워크 실시예의 상황에서, 다음과 같이 편성된다. "동기(motivation)" 섹션은 엔티티 프레임워크에 대한 추가의 동기를 제공한다. "엔티티 프레임

워크(Entity Framework)" 섹션은 엔티티 프레임워크 및 엔티티 데이터 모델을 제공한다. "프로그래밍 패턴(Programming Patterns)" 섹션은 엔티티 프레임워크에 대한 프로그래밍 패턴들을 설명한다. "개체 서비스(Object Services)" 섹션은 개체 서비스 모듈을 개설한다. "매핑(Mapping)" 섹션은 엔티티 프레임워크의 매핑 컴포넌트들에 초점을 맞추고, "쿼리 처리(Query Processing)" 및 "업데이트 처리(Update Processing)" 섹션들은 쿼리들 및 업데이들이 처리되는 방법을 설명한다. "메타데이터(Metadata)" 및 "도구(Tools)"는 엔티티 프레임워크의 메타데이터 서브시스템 및 도구 컴포넌트들을 설명한다.

[0047] 동기(MOTIVATION)

[0048] 이 섹션은 애플리케이션들 및 데이터 중심 서비스들에 대하여 보다 고레벨의 데이터 모델링 계층이 필수적이 된 이유에 대하여 논의한다.

[0049] **데이터 애플리케이션에서의 정보 레벨**

[0050] 데이터베이스 디자인을 생성하기 위한 오늘날의 지배적인 정보 모델링 방법들은 정보 모델을 다음의 4가지 주요 레벨로 분해한다: 물리적, 논리적(관계형), 개념적, 및 프로그래밍/프리젠테이션.

[0051] *물리적(physical)* 모델은 메모리, 와이어 또는 디스크 등의 물리적 리소스들에서 데이터가 어떻게 *표현*되는지를 기술한다. 이 계층에서 논의되는 개념들의 어휘는 레코드 포맷, 파일 파티션 및 그룹, 힙(heaps), 및 인덱스를 포함한다. 물리적 모델은 전형적으로 애플리케이션에게 보이지 않는다 – 물리적 모델에 대한 변경들은 애플리케이션 로직에는 영향을 미치지 않겠지만, 애플리케이션 성능에는 영향을 미칠 수 있다.

[0052] *논리적(logical)* 데이터 모델은 대상 도메인의 완전하고 정확한 정보 모델이다. 관계형 모델은 가장 논리적인 데이터 모델들에 대한 선택의 표현이다. 이 논리적 레벨에서 논의되는 개념들은 테이블, 행(rows), 기본 키(primary-key)/외래 키(foreign-key) 제약들, 및 정규화(normalization)를 포함한다. 정규화는 데이터 일관성, 증가된 동시성, 및 보다 나은 OLTP 성능을 성취하는 데 도움이 되지만, 그것은 또한 애플리케이션들에 대한 상당한 도전 과제들을 도입한다. 논리적 레벨에서 정규화된 데이터는 종종 너무 조각화(fragment)되고 애플리케이션 로직은 복수의 테이블로부터의 행들을 애플리케이션 도메인의 아티팩트들과 더 많이 아주 닮은 보다 고레벨의 엔티티들로 어셈블할 필요가 있다.

[0053] *개념적(conceptual)* 모델은 문제의 도메인으로부터의 핵심 정보 엔티티들 및 그들의 관계들을 캡처한다. 잘 알려진 개념적 모델은 1976년에 피터 첸(Peter Chen)에 의해 도입된 엔티티-관계 모델(Entity-Relationship Model)이다. UML은 개념적 모델의 보다 최근의 예이다. 대부분의 애플리케이션들은 애플리케이션 개발 라이프 사이클의 초기에 개념적 디자인 단계(conceptual design phase)를 포함한다. 그러나, 불행히도, 개념적 데이터 모델 다이어그램들은 "벽에 밀어붙여진(pinned to a wall)" 상태에 머물러 시간이 흐름에 따라 점점 더 애플리케이션 구현의 실체로부터 해체된다. 엔티티 프레임워크의 중요한 목표는 (다음의 섹션에서 설명되는 엔티티 데이터 모델에 의해 구현되는) 개념적 데이터 모델을 데이터 플랫폼의 구체적인 프로그램 가능한 추상화로 만드는 것이다.

[0054] *프로그래밍/프리젠테이션(programming/presentation)* 모델은 개념적 모델의 엔티티들 및 관계들이 어떻게 주어진 과제에 기초하여 상이한 형태로 명시(manifest)(제시(present))될 필요가 있는지를 기술한다. 일부 엔티티들은 애플리케이션 비즈니스 로직을 구현하기 위해 프로그래밍 언어 개체들로 변환될 필요가 있고; 다른 것들은 웹 서비스 호출을 위해 XML 스트림으로 변환될 필요가 있고; 또 다른 것들은 사용자 인터페이스 데이터 바인딩을 위하여 리스트 또는 딕셔너리(dictionaries) 등의 메모리 내 구조들로 변환될 필요가 있다. 당연히, 보편적인 프로그래밍 모델 또는 프리젠테이션 형태는 없고, 따라서, 애플리케이션은 엔티티들을 다양한 프리젠테이션 형태들로 변환하기 위한 유연성 있는 메커니즘들을 필요로 한다.

[0055] 대부분의 애플리케이션들 및 데이터 중심 서비스들은 관계형 데이터베이스 스키마에서 오더(order)가 정규화될 수 있는 몇몇 테이블들에 관해서가 아니라, *오더(Order)* 등의 고레벨 개념들에 관하여 추론하기를 좋아할 것이다. 오더는 그 오더와 관련된 상태 또는 로직을 캡슐화(encapsulating)하는 비주얼 베이직(Visual Basic) 또는 C# 내의 클래스 인스턴스로서, 또는 웹 서비스와 통신하기 위한 XML 스트림으로서 프리젠테이션/프로그래밍 레벨에서 자신을 명시할 수 있다. 적당한 프리젠테이션 모델은 하나도 존재하지 않지만, 구체적인 개념적 모델을 제공하고, 그 후 다양한 프리젠테이션 모델들 및 다른 보다 고레벨의 데이터 서비스들로 및 그로부터의 유연성

있는 매핑을 위한 기반으로서 그 모델을 이용할 수 있는 데 있어서 가치가 존재한다.

[0056] **애플리케이션들 및 서비스들의 진화**

[0057] 10-20년 전에 데이터 기반 애플리케이션들은 전형적으로 데이터 단일체(data monoliths); 그 논리적 스키마 레벨에서 데이터베이스 시스템과 상호작용한 동사-목적어 함수들(verb-object functions)(예컨대, create-order, update-customer)에 의해 팩터링되는 논리를 갖는 닫힌 시스템(closed systems)으로서 구조화되었다. 몇몇 현저한 경향들은 현대의 데이터 기반 애플리케이션들이 오늘날 팩터링되고 전개되는 방향을 구체화하였다. 이들 중 주요한 것들은 개체 지향 팩터링(object-oriented factoring), 서비스 레벨 애플리케이션 구성(service level application composition), 및 보다 고레벨의 데이터 중심 서비스(higher level data-centric services)이다. 개념적 엔티티들은 오늘날의 애플리케이션들의 중요한 부분이다. 이들 엔티티들은 각종의 표현들에 매핑되고 각종의 서비스들에 바운딩되어야 한다. 정확한 표현 또는 서비스 바인딩은 하나도 존재하지 않는다: XML, 관계형 및 개체 표현들은 모두 중요하지만, 단 하나도 모든 애플리케이션들에 대하여 충분하지 않다. 그러므로, 보다 고레벨의 데이터 모델링 계층을 지원하고, 또한 복수의 프리젠테이션 계층들이 플러그인되는 것을 허용하는 프레임워크가 요구되고 있다 – 엔티티 프레임워크는 이들 요건을 충족시키려고 한다.

[0058] 데이터 중심 서비스들도 유사한 방식으로 진화해오고 있다. 20년 전에 "데이터 플랫폼"에 의해 제공된 서비스들은 최소한이었고 RDBMS 내의 논리적 스키마에 집중하였다. 이들 서비스는 쿼리 및 업데이트, 원자성 트랜잭션(atomic transactions), 및 백업 및 로드/익스트랙트(load/extract) 등의 벌크 연산(bulk operations)을 포함하였다.

[0059] SQL 서버 자체는 전통적인 RDBMS로부터 개념적 스키마 레벨에서 실현된 엔티티들에 걸쳐서 다수의 높은 가치의 데이터 중심 서비스들을 제공하는 *완전한 데이터 플랫폼(complete data platform)*으로 진화하고 있다. SQL 서버 제품 내의 몇몇 보다 고레벨의 데이터 중심 서비스들 – 둘만을 지명하여 리플리케이션(Replication), 리포트 빌더(Report Builder) – 은 점점 더 개념적 스키마 레벨에서 그들의 서비스를 제공하고 있다. 현재, 이들 서비스들 각각은 개념적 엔티티들을 기술하고 그것들을 기초가 되는 논리적 스키마 레벨에 매핑하기 위한 개별 도구를 갖고 있다. 엔티티 프레임워크의 목표는 이들 서비스들 전부가 공유할 수 있는 공통의 보다 고레벨의 개념적 추상화를 제공하는 것이다.

[0060] 엔티티 프레임워크(THE ENTITY FRAMEWORK)

[0061] 본 명세서에서 설명되는 엔티티 프레임워크 전에 존재한 마이크로소프트의 ADO.NET는 애플리케이션들이 데이터 저장소들에 접속하여 그 안에 포함된 데이터를 다양한 방법으로 조작할 수 있게 한 데이터 액세스 기술이었다. 그것은 Microsoft.NET 프레임워크의 일부였고 그것은 .NET 프레임워크 클래스 라이브러리의 나머지와 고도로 통합되었다. 이전의 ADO.NET 프레임워크는 다음의 2개의 주요 부분들을 가졌다: *프로바이더(providers)* 및 *서비스(services)*. ADO.NET *프로바이더*들은 특정 데이터 저장소들에게 말하는 방법을 아는 컴포넌트들이다. 프로바이더들은 다음의 3개의 핵심 기능 부분들로 구성된다: *연결(connections)*)은 기초가 되는 데이터 소스에의 액세스를 관리하고; *명령(commands)*은 데이터 소스에 대하여 실행될 명령(쿼리, 프로시저 호출(procedure call) 등)를 나타내고; *데이터 리더(data readers)*는 명령 실행의 결과를 나타낸다. ADO.NET *서비스*들은 오프라인 데이터 프로그래밍 시나리오를 가능하게 하는 DataSet 등의 프로바이더-중립 컴포넌트들을 포함한다. (DataSet는 데이터 소스에 관계없이 일관성 있는 관계형 프로그래밍 모델을 제공하는 데이터의 메모리 상주 표현이다.)

[0062] **엔티티 프레임워크 – 개관**

[0063] ADO.NET 엔티티 프레임워크는 기존의 현존하는 ADO.NET *프로바이더* 모델 상에 구축되고, 다음의 기능을 추가한다:

[0064] 1. 개념적 스키마들을 모델링하는 데 도움이 되는, 새로운 개념적 데이터 모델인 *엔티티 데이터 모델(Entity Data Model*; EDM).

[0065] 2. 상이한 프로바이더들과 통신하기 위한 쿼리의 프로그램적 표현(정규 명령 트리(canonical command trees)), 및 EDM의 인스턴스들을 조작하기 위한 새로운 데이터 조작 언어(data manipulation language; DML)인 *엔티티*

SQL.

[0066] 3. 개념적 스키마와 논리적 스키마 간의 매핑을 정의하는 능력.

[0067] 4. 개념적 스카마에 대한 ADO.NET 프로바이더 프로그래밍.

[0068] 5. ORM 같은 기능을 제공하는 개체 서비스 계층.

[0069] 6. .NET 언어로부터의 개체들로서의 데이터에 대하여 프로그램하는 것을 용이하게 하는 LINQ 기술과의 통합.

[0070] **엔티티 데이터 모델**

[0071] 엔티티 데이터 모델(EDM)은 풍부한 데이터 중심 애플리케이션의 개발을 허용한다. 그것은 E-R 도메인으로부터의 개념들을 이용하여 고전적인 관계형 모델을 확장한다. 여기서 제공되는 예시적인 실시예에서, EDM 내의 조직 개념들은 엔티티들 및 관계들을 포함한다. *엔티티(entities)*는 ID(identity)를 갖는 최상위 레벨 항목들을 나타내는 반면, *관계(Relationships)*는 2 이상의 엔티티들을 *관련(relate)*시키는(즉, 그들 간의 관계를 기술하는) 데 이용된다.

[0072] 일 실시예에서, EDM은 C#(CLR)와 같이 *개체/참조 기반(object/reference-based)*이기보다는, 관계형 모델(및 SQL)과 같이 *값 기반(value-based)*이다. 몇몇 개체 프로그래밍 모델들은 EDM의 위에 용이하게 계층화될 수 있다. 유사하게, EDM은 지속을 위해 하나 이상의 DBMS 구현들에 매핑될 수 있다.

[0073] EDM 및 엔티티 SQL은 데이터 플랫폼을 위한 보다 풍부한 데이터 모델 및 데이터 조작 언어를 나타내며 CRM 및 ERP 등의 애플리케이션들, 리포팅(Reporting), 비즈니스(Business), 인텔리전스(Intelligence), 복제(Replication) 및 동기화(Synchronization) 등의 데이터 집약적(data-intensive) 서비스, 및 데이터 집약적 애플리케이션들이 그들의 필요에 보다 근접하는 의미 및 구조의 레벨에서 데이터를 모델링 및 조작하는 것을 가능하게 하도록 의도되어 있다. 이제 EDM에 관한 다양한 개념들을 설명한다.

[0074] *EDM 타입(EDM Types)*

[0075] *EntityType*은 엔티티의 구조를 기술한다. 엔티티는 엔티티의 구조를 기술하는 0개 이상의 속성(특성, 필드)을 가질 수 있다. 또한, 엔티티 타입은 *키(key)* – 엔티티들의 컬렉션 내의 엔티티 인스턴스를 고유하게 식별하는 값을 갖는 속성들의 세트 – 를 정의해야 한다. EntityType은 다른 엔티티 타입(또는 서브타입)으로부터 파생할 수 있다 – EDM은 단일 상속 모델을 지원한다. 엔티티의 속성들은 단순(simple) 또는 복잡(complex) 타입들일 수 있다. *SimpleType*은 스칼라(또는 원자성) 타입(예컨대, 정수(integer), 문자열(string))을 나타내는 반면, *ComplexType*은 구조화된 속성들(예컨대, 어드레스)을 나타낸다. ComplexType은 그 자신이 스칼라 또는 복잡 타입 속성들일 수 있는 0개 이상의 속성들로 구성된다. *RelationshipType*은 2개(또는 그 이상)의 엔티티 타입들 간의 관계를 기술한다. *EDM 스키마*들은 타입들에 대한 그룹핑 메커니즘을 제공한다 – 타입들은 스키마에서 정의되어야 한다. 타입 이름과 조합된 스키마의 네임스페이스는 특정 타입을 고유하게 식별한다.

[0076] *EDM 인스턴스 모델*

[0077] 엔티티 인스턴스들(또는 단지 엔티티들)은 *EntitySet*에 논리적으로 포함된다. Entityset는 균질의 엔티티들의 컬렉션(homogeneous collection of entities)이다. 즉, EntitySet 내의 모든 엔티티들은 동일한(또는 파생된) EntityType의 것이어야 한다. EntitySet는 데이터베이스 테이블과 개념적으로 유사한 반면, 엔티티는 테이블의 행(row)과 유사하다. 엔티티 인스턴스는 정확히 하나의 엔티티 세트에 속해야 한다. 유사한 방식으로, 관계 인스턴스들은 *RelationshipSet*에 논리적으로 포함된다. RelationshipSet의 정의는 관계의 범위를 정한다. 즉, 그것은 관계에 참여하는 엔티티 타입들의 인스턴스들을 보유하는 EntitySet들을 식별한다. RelationshipSet는 데이터베이스 내의 링크 테이블(link-table)과 개념적으로 유사하다. SimpleType 및 ComplexType은 EntityType의 속성들로서만 인스턴스화될 수 있다. *EntityContainer*는 EntitySet들 및 RelationshipSet들의 논리적 그룹핑이다 – 이는 어떻게 스키마(Schema)가 EDM 타입들에 대한 그룹핑 메커니즘인지와 유사하다.

[0078] *예시적인 EDM 스키마*

[0079] 샘플 EDM 스키마가 아래 제시되어 있다:

```
<?xml version="1.0" encoding="utf-8"?>
<Schema Namespace="AdventureWorks" Alias="Self" …>
  <EntityContainer Name="AdventureWorksContainer">
    <EntitySet Name="ESalesOrders"
               EntityType="Self.ESalesOrder" />
    <EntitySet Name="ESalesPersons"
               EntityType="Self.ESalesPerson" />
    <AssociationSet Name="ESalesPersonOrders"
           Association="Self.ESalesPersonOrder">
      <End Role="ESalesPerson"
           EntitySet="ESalesPersons" />
      <End Role="EOrder" EntitySet="ESalesOrders" />
    </AssociationSet>
  </EntityContainer>
```

[0080]

```
  <!-- Sales Order Type Hierarchy-->
  <EntityType Name="ESalesOrder" Key="Id">
    <Property Name="Id" Type="Int32"
              Nullable="false" />
    <Property Name="AccountNum" Type="String"
              MaxLength="15" />
  </EntityType>
  <EntityType Name="EStoreSalesOrder"
              BaseType="Self.ESalesOrder">
    <Property Name="Tax" Type="Decimal"
              Precision="28" Scale="4" />
  </EntityType>
```

[0081]

```
  <!-- Person EntityType -->
  <EntityType Name="ESalesPerson" Key="Id">
    <!-- Properties from SSalesPersons table-->
    <Property Name="Id" Type="Int32"
              Nullable="false" />
    <Property Name="Bonus" Type="Decimal"
              Precision="28" Scale="4" />
    <!-- Properties from SEmployees table-->
    <Property Name="Title" Type="String"
              MaxLength="50" />
    <Property Name="HireDate" Type="DateTime" />
    <!-- Properties from the SContacts table-->
    <Property Name="Name" Type="String"
              MaxLength="50" />
    <Property Name="Contact" Type="Self.ContactInfo"
              Nullable="false" />
  </EntityType>
```

[0082]

```
  <ComplexType Name="ContactInfo">
    <Property Name="Email" Type="String"
              MaxLength="50" />
    <Property Name="Phone" Type="String"
              MaxLength="25" />
  </ComplexType>
  <Association Name="ESalesPersonOrder">
    <End Role="EOrder" Type="Self.ESalesOrder"
         Multiplicity="*" />
    <End Role="ESalesPerson" Multiplicity="1"
         Type="Self.ESalesPerson" />
  </Association>
</Schema>
```

[0083]

[0084] **고레벨 아키텍처**

[0085] 이 섹션은 ADO.NET 엔티티 프레임워크의 아키텍처를 개설한다. 그의 주요 기능 컴포넌트들이 도 1에 도시되어

있고 다음을 포함할 수 있다.

[0086] 데이터 소스 특정 프로바이더(Data source-specific providers). 엔티티 프레임워크(100)는 ADO.NET 데이터 프로바이더 모델 상에 구축된다. SQL 서버(151, 152), 관계형(relational) 소스(153), 비관계형(non-relational)(154), 및 웹 서비스(155) 소스 등의 몇몇 데이터 소스들에 대한 특정 프로바이더들(122-125)이 있다. 이 프로바이더들(122-125)은 저장소 특정(store-specific) ADO.NET 프로바이더 API(121)로부터 호출될 수 있다.

[0087] EntityClient 프로바이더. EntityClient 프로바이더(111)는 구체적인 개념적 프로그래밍 계층을 표현한다. 이는 데이터가 EDM 엔티티들 및 관계들에 의하여 액세스되고 엔티티 기반 SQL 언어(엔티티 SQL)를 사용하여 쿼리/업데이트되는 새로운 값 기반 데이터 프로바이더이다. EntityClient 프로바이더(111)는 메타데이터 서비스(112), 쿼리 및 업데이트 파이프라인(113), 트랜잭션 서포트(115), 뷰 매니저 런타임(116), 및 플랫 관계형 테이블들(flat relational tables) 상에서 업데이트 가능한 EDM 뷰를 지원하는 뷰 매핑 서브시스템(114)을 또한 포함할 수 있는 엔티티 데이터 서비스(Entity Data Services)(110)의 일부를 형성한다. 테이블들과 엔티티들 간의 매핑은 매핑 명세 언어(mapping specification language)에 의하여 선언적으로 특정된다.

[0088] 개체 서비스들 및 그 밖의 프로그래밍 계층들. 엔티티 프레임워크(100)의 개체 서비스(Object Services) 컴포넌트(131)는 엔티티들에 대한 풍부한 개체 추상화, 이들 개체들에 대한 풍부한 서비스 세트를 제공하고, 애플리케이션들이 잘 알려진 프로그래밍 언어 구성들을 이용하여 명령형 코딩 경험(imperative coding experience)(161)에서 프로그램하는 것을 허용한다. 이 컴포넌트는 개체들(변경 추적, ID 확인(identity resolution))에 대한 상태 관리 서비스들을 지원하고, 개체들 및 관계들을 탐색 및 로딩하기 위한 서비스들을 지원하고, Xlinq(132) 등의 컴포넌트들을 이용하여 LINQ 및 엔티티 SQL을 통한 쿼리들을 지원하고, 개체들이 업데이트되고 지속될 수 있게 한다.

[0089] 엔티티 프레임워크는 130과 유사한 복수의 프로그래밍 계층들이 EntityClient 프로바이더(111)에 의해 익스포즈된 값 기반 엔티티 데이터 서비스 계층(110)에 플러그온되는 것을 허용한다. 개체 서비스(131) 컴포넌트는 CLR 개체들을 표면화시키고, ORM 같은 기능을 제공하는 하나의 그러한 프로그래밍 계층이다.

[0090] 메타데이터 서비스(112) 컴포넌트는 엔티티 프레임워크(110), 및 엔티티 프레임워크 상의 애플리케이션들의 디자인 타임 및 런타임 필요들에 대한 메타데이터를 관리한다. EDM 개념들(엔티티, 관계, EntitySet, RelationshipSet), 저장소 개념들(테이블, 열(columns), 제약(constraints)), 및 매핑 개념들과 연결된 모든 메타데이터는 메타데이터 인터페이스를 통하여 익스포즈된다. 메타데이터 컴포넌트(112)는 또한 모델-구동 애플리케이션 디자인을 지원하는 도메인 모델링 도구들 간의 링크로서 기능한다.

[0091] 디자인 및 메타데이터 도구. 엔티티 프레임워크(100)는 모델-구동 애플리케이션 개발을 가능하게 하기 위해 도메인 디자이너들(170)과 통합한다. 도구들은 EDM 디자인 도구, 모델링 도구(171), 매핑 디자인 도구(172), 브라우징 디자인 도구(173), 바인딩 디자인 도구(174), 코드 생성 도구(175), 및 쿼리 모델러(query modelers)를 포함한다.

[0092] 서비스. 리포팅(141), 동기화(142), 웹 서비스(143) 및 비즈니스 분석 등의 풍부한 데이터 중심 서비스들이 엔티티 프레임워크(100)를 이용하여 구축될 수 있다.

[0093] 프로그래밍 패턴

[0094] LINQ와 함께 ADO.NET 엔티티 프레임워크는 애플리케이션 코드와 데이터 간의 임피던스 부정합을 현저히 감소시킴으로써 애플리케이션 개발자 생산성을 증대시킨다. 이 섹션에서는 논리적, 개념적 및 개체 추상화 계층들에서의 데이터 액세스 프로그래밍 패턴들의 진화를 설명한다.

[0095] 샘플 AdventureWorks 데이터베이스에 기초하여 다음의 관계형 스키마 프래그먼트를 생각해보자. 이 데이터베이스는 도 2에 도시된 것과 같은 관계형 스키마를 따를 수 있는 SContacts(201), SEmployees(202), SSalesPerson(203), 및 SSalesOrders(204) 테이블들로 이루어진다.

```
SContacts (ContactId, Name, Email, Phone)
SEmployees (EmployeeId, Title, HireDate)
SSalesPersons (SalesPersonId, Bonus)
SSalesOrders (SalesOrderId, SalesPersonId)
```

[0096]

[0097] 어떤 날짜 전에 고용된 판매원의 이름 및 고용 날짜를 얻기 위한 애플리케이션 코드 프래그먼트를 생각해보자(아래에 제시됨). 이 코드 프래그먼트에는 답변할 필요가 있는 비즈니스 질문과 거의 관계가 없는 4개의 주요 단점들이 있다. 첫째로, 쿼리가 영어로 매우 간결하게 진술될 수 있다 하더라도, SQL 문(statement)은 꽤 장황하고 SContacts, SEmployees, SSalesPerson, 및 SSalesOrders 테이블들로부터의 적당한 열(column)들을 수집하기 위해 필요한 다중 테이블 결합(multi-table join)을 공식화(formulate)하기 위해 개발자가 정규화된 관계형 스키마를 알고 있을 것을 요구한다. 또한, 기초가 되는 데이터베이스 스키마들에 대한 임의의 변경은 아래의 코드 프래그먼트에서의 대응하는 변경을 필요로 할 것이다. 둘째로, 사용자는 데이터 소스에의 명시적 연결(explicit connection)을 정의해야 한다. 셋째로, 반환된 결과들은 강 타입(strongly typed)이 아니므로, 현존하지 않는 열 이름들에 대한 임의의 언급은 쿼리가 실행된 후에만 캡처될 것이다. 넷째로, SQL 문은 명령(Command) API에게 강한 속성이고 그것의 공식화에서의 어떠한 오류든 실행시에만 캡처될 것이다. 이 코드는 ADO.NET 2.0을 이용하여 작성되어 있지만, 그 코드 패턴 및 그의 단점들은 ODBC, JDBC, 또는 OLE-DB 등의 임의의 다른 관계형 데이터 액세스에도 적용된다.

```
void EmpsByDate(DateTime date) {
using( SqlConnection con =
  new SqlConnection (CONN_STRING) ) {
    con.Open();
    SqlCommand cmd = con.CreateCommand();
    cmd.CommandText = @"
    SELECT SalesPersonID, FirstName, HireDate
    FROM SSalesPersons sp
      INNER JOIN SEmployees e
      ON sp.SalesPersonID = e.EmployeeID
      INNER JOIN SContacts c
      ON e.EmployeeID = c.ContactID
    WHERE e.HireDate < @date";
    cmd.Parameters.AddWithValue("@date",date);
```

[0098]

```
    DbDataReader r = cmd.ExecuteReader();
    while(r.Read()) {
      Console.WriteLine("{0:d}:\t{1}",
         r["HireDate"], r["FirstName"]);
} } }
```

[0099]

[0100] 이 샘플 관계형 스키마는 도 3에서 도시된 EDM 스키마를 통하여 개념적 레벨에서 캡처될 수 있다. 그것은 SContacts(201), SEmployees(202), SSalesPerson(203), 및 SSalesOrders(204) 테이블들의 프래그먼트를 추출해내는 엔티티 타입 ESalesPerson(302)을 정의한다. 그것은 또한 EStoreOrder(301) 및 ESalesOrder(303) 엔티티 타입들 간의 상속 관계도 캡처한다.

[0101] 개념적 계층에서의 등가의 프로그램은 다음과 같이 작성된다:

```
void EmpsByDate (DateTime date) {
using( EntityConnection con =
  new EntityConnection (CONN_STRING) ) {
    con.Open();
    EntityCommand cmd = con.CreateCommand();
    cmd.CommandText = @"
      SELECT VALUE sp
      FROM ESalesPersons sp
      WHERE sp.HireDate < @date";
    cmd.Parameters.AddWithValue ("date",
        date);
    DbDataReader r = cmd.ExecuteReader(
              CommandBehavior.SequentialAccess);
    while (r.Read()) {
      Console.WriteLine("{0:d}:\t{1}",
        r["HireDate"]], r["FirstName"])
} } }
```

[0102]

[0103] 이 SQL 문은 상당히 단순화되었다 – 사용자는 더 이상 정확한 데이터베이스 레이아웃에 관하여 알지 않아도 된다. 또한, 애플리케이션 로직은 기초가 되는 데이터베이스 스키마에 대하나 변경들로부터 분리될 수 있다. 그러나, 이 프래그먼트는 여전히 강 기반(strong-based)이고, 여전히 프로그래밍 언어 타입-체킹의 이익을 얻지 못하고, 약 타입(weakly typed) 결과들을 반환한다.

[0104] 엔티티들 주위에 씬 개체 랩퍼(thin object wrapper)를 추가하고 C#에서의 언어 통합 쿼리(Language Integrated Query; LINQ) 확장들을 이용함으로써, 다음과 같이 임피던스 부정합이 없는 등가의 함수를 작성할 수 있다:

```
void EmpsByDate(DateTime date) {
  using (AdventureWorksDB aw =
    new AdventureWorksDB()) {
    var people = from p in aw.SalesPersons
              where p.HireDate < date
              select p;
    foreach (SalesPerson p in people) {
      Console.WriteLine("{0:d}\t{1}",
           p.HireDate, p.FirstName);
} } }
```

[0105]

[0106] 이 쿼리는 단순하고; 애플리케이션은 기초가 되는 데이터베이스 스키마에 대한 변경들로부터 (크게) 분리되고; 쿼리는 C# 컴파일러에 의해 완전히 타입-체크된다. 쿼리들에 더하여, 개체들과 상호작용하여 그 개체들에 대하여 생성(Create), 판독(Read), 업데이트(Update) 및 삭제(Delete)(CRUD) 연산들을 수행할 수 있다. 이들에 대한 예는 업데이트 처리 섹션에서 설명한다.

[0107] 개체 서비스(OBJECT SERVICES)

[0108] 개체 서비스 컴포넌트는 개념적(엔티티) 계층 상의 프로그래밍/프리젠테이션 계층이다. 그것은 프로그래밍 언어와 값 기반 개념적 계층들 간의 상호작용을 용이하게 하는 몇 개의 컴포넌트들을 수용한다. 프로그래밍 언어 런타임(예컨대, NET, 자바)마다 하나의 개체 서비스가 존재할 것이라고 예상된다. 만일 그것이 .NET CLR을 지원하도록 디자인되어 있다면, 임의의 .NET 언어에서의 프로그램들이 엔티티 프레임워크와 상호작용할 수 있다. 개체 서비스는 다음의 주요 컴포넌트들로 구성된다:

[0109] *ObjectContext* 클래스는 데이터베이스 연결(database connection), 메타데이터 작업 영역(metadata workspace), 개체 상태 관리자(object state manager), 및 개체 구체화기(object materializer)를 수용한다.

이 클래스는 엔티티 SQL 또는 LINQ 구문에서 쿼리들의 공식화를 가능하게 하는 개체 쿼리 인터페이스 *ObjectQuery*<T>를 포함하고, 강 타입 개체 결과들을 *ObjectReader*<T>로서 반환한다. ObjectContext는 또한 프로그래밍 언어 계층과 개념적 계층 간의 쿼리 및 업데이트(즉, *SaveChange*들) 개체 레벨 인터페이스들을 익스포즈한다. *개체 상태 관리자*는 3개의 주요 기능: (a) ID 확인(identity resolution)을 제공하는, 쿼리 결과들을 캐싱하고, 겹치는 쿼리 결과들로부터의 개체들을 병합하는 정책들을 관리하는 기능, (b) 메모리 내 변경들(in-memory changes)을 추적하는 기능, 및 (c) 업데이트 처리 인프라에 입력되는 변경 목록을 구성하는 기능을 갖는다. 개체 상태 관리자는 캐시 내의 각 엔티티의 상태 – (캐시로부터) *분리(detached)*, *추가(added)*, *불변(unchanged)*, *수정(modified)*, 및 *삭제(deleted)* – 를 유지하고 그들의 상태 변화를 추적한다. *개체 구체화기(object materializer)*는 쿼리 및 업데이트 동안에 개념적 계층으로부터의 엔티티 값들과 대응하는 CRL 개체들 간의 변환을 수행한다.

[0110]

[0111] 매핑

[0112] 일 실시예에서, ADO.NET 엔티티 프레임워크 등의 범용 데이터 액세스 계층의 백본(backbone)은 애플리케이션 데이터와 데이터베이스에 저장된 데이터 간의 관계를 확립하는 *매핑(mapping)*일 수 있다. 애플리케이션은 개체 또는 개념적 레벨에서 데이터를 쿼리 및 업데이트하고 이들 연산들은 매칭에 의하여 저장소로 변환된다. 임의의 매핑 해법에 의해 다루어져야 하는 다수의 기술적 도전 과제들이 있다. 특히 선언적 데이터 조작이 요구되지 않는 경우에, 관계형 테이블 내의 각 행을 개체로서 익스포즈하기 위해 일대일 매핑을 이용하는 ORM을 구축하는 것은 비교적 수월하다. 그러나, 보다 복잡한 매핑, 세트 기반 연산(set-based operations), 성능, 다중-DBMS-벤더 지원(multi-DBMS-vendor support), 및 기타 요건들이 개입함에 따라서, 특별 해법들은 빠르게 감당할 수 없게 된다.

[0113] **문제: 매핑에 의한 업데이트**

[0114] 매핑에 의하여 데이터에 액세스하는 문제는 "뷰들(views)"에 의하여 모델링될 수 있다. 즉, 클라이언트 계층 내의 개체들/엔티티들은 테이블 행들에 걸쳐서 풍부한 뷰들로서 간주될 수 있다. 그러나, 제한된 클래스의 뷰들만이 업데이트 가능하다는 것은 잘 알려져 있다. 예를 들어, 상업 데이터베이스 시스템들은 조인(joins) 및 유니언(unions)을 포함하는 뷰들에서 복수의 테이블에 대한 업데이트를 허용하지 않는다. 매우 단순한 뷰들에서조차 고유 업데이트 변환을 찾아내는 것은 뷰에 의한 업데이트 거동(update behavior)의 고유의 미명세(under-specification)로 인해 거의 불가능하다. 연구에 따르면 뷰들로부터 업데이터 의미들을 빼내는 것은 어렵고 상당한 사용자 전문 지식을 필요로 할 수 있다. 그러나, 매핑 구동 데이터 액세스를 위하여, 뷰에 대한 모든 업데이트마다의 명확히 정의된 변환이 존재하는 것이 유리하다.

[0115] 또한, 매핑 구동 시나리오에서, 업데이트 가능성 요건은 단일 뷰를 넘어선다. 예를 들면, 고객(Customer) 및 오더(Order) 엔티티들을 조작하는 비즈니스 애플리케이션은 2개의 뷰에 대한 동작들을 효과적으로 수행한다. 때때로 몇 개의 뷰들을 동시에 업데이트하는 것에 의해서만 일관성 있는 애플리케이션 상태가 성취될 수 있다. 그러한 업데이트들의 케이스별(case-by-case) 변환은 업데이트 로직의 조합적 폭발(combinatorial explosion)을 가져올 수 있다. 그의 구현을 애플리케이션 개발자들에게 위임하는 것은 만족스럽지 못하다. 왜냐하면 그것은 그들이 데이터 액세스의 가장 복잡한 부분들 중 하나를 수동으로 다룰 것을 요구하기 때문이다.

[0116] **ADO.NET 매핑 접근법**

[0117] ADO.NET 엔티티 프레임워크는 상기 도전적 과제들을 처리하려고 하는 혁신적인 매핑 아키텍처를 지원한다. 이는 다음의 아이디어를 이용한다.

[0118] 1. 명세(Specification): 매핑들은 명확히 정의된 의미를 갖고 광범위한 매핑 시나리오들을 비전문가 사용자들의 능력 범위 내에 있게 하는 *선언적 언어(declarative language)*를 이용하여 특정된다.

[0119] 2. 컴파일(Compilation): 매핑들은 런타임 엔진에서 쿼리 및 업데이트 처리를 구동하는, 쿼리 및 업데이트 뷰(query and update views)라 불리는, *양방향 뷰들(bidirectional views)*로 컴파일된다.

[0120] 3. 실행: 업데이트 변환은 *구체화된 뷰 관리(materialized view maintenance)*인 강건한 데이터베이스 기술

(robust database technology)을 이용하는 일반적인 메커니즘을 이용하여 행해진다. 쿼리 변환은 뷰 언폴딩(view unfolding)을 이용한다.

[0121] 이 새로운 매핑 아키텍처는 원칙에 의한 미래 보장 방식(principled, future-proof way)으로 강력한 다량의 매핑 구동 기술들을 구축하는 것을 가능하게 한다. 또한, 그것은 즉각적인 실제 관련의 흥미로운 연구 방향들을 전개한다. 다음의 하위 섹션들은 매핑들의 명세 및 컴파일을 설명한다. 실행은 아래에서 쿼리 처리 및 업데이트 처리 섹션들에서 고려된다. 본 명세서에서 제공된 예시적인 매핑 아키텍처들의 추가의 양태들 및 실시예들도 아래에서 "추가의 양태들 및 실시예들"이라는 표제가 붙은 섹션에서 설명된다.

[0122] **매핑의 명세**

[0123] 매핑은 매핑 프래그먼트들의 세트를 이용하여 특정된다. 각 매핑 프래그먼트는 $Q_{Entities} = Q_{Tables}$이라는 형태의 제약이고 여기서 $Q_{Entities}$는 (애플리케이션 측의) 엔티티 스키마 상의 쿼리이고 Q_{Tables}는 (저장소 측의) 데이터베이스 스키마 상의 쿼리이다. 매핑 프래그먼트는 엔티티 데이터의 일부가 어떻게 관계형 데이터의 일부에 대응하는지를 기술한다. 즉, 매핑 프래그먼트는 다른 프래그먼트들과 관계없이 이해될 수 있는 명세의 기본 단위이다.

[0124] 설명을 위해, 도 4의 샘플 매핑 시나리오를 생각해보자. 도 4는 엔티티 스키마(좌측)와 데이터 스키마(우측) 간의 매핑을 예시한다. 매핑은 XML 파일 또는 그래픽 도구를 이용하여 정의될 수 있다. 엔티티 스키마는 본 명세서의 엔티티 데이터 모델 섹션에서의 것에 대응한다. 저장소 측에는 4개의 테이블, 즉 SSalesOrders, SSalesPersons, SEmployees, 및 SContacts가 있다. 엔티티 스키마 측에는 2개의 엔티티 세트, 즉, ESalesOrder와 ESalesPersons, 및 하나의 연결 세트(association set), ESalesPersonOrders가 있다.

[0125] 매핑은 도 5에 도시된 바와 같이 엔티티 스키마 및 관계형 스키마 상의 쿼리들에 의하여 표현된다.

[0126] 도 5에서, 프래그먼트 1은 ESalesOrders 내의 정확한 타입 ESalesOrder의 모든 엔티티들에 대한 (Id, AccountNum) 값들의 세트가 IsOnline이 참(true)인 SSalesOrders 테이블로부터 검색된 (SalesOrderId, AccountNum) 값들의 세트와 동일하다는 것을 말한다. 프래그먼트 2도 유사하다. 프래그먼트 3은 연결 세트 RSalesPersonOrders를 SSalesOrders 테이블에 매핑하고 각 연결 항목이 이 테이블 내의 각 행에 대한 기본 키(primary key), 외래 키(foreign key)에 대응한다는 것을 말한다. 프래그먼트 4, 5, 및 6은 ESalesPerson 엔티티 세트 내의 엔티티들이 3개의 테이블 SSalesPersons, SContacts, SEmployees에 걸쳐서 분할된다는 것을 말한다.

[0127] **양방향 뷰(Bidirectional Views)**

[0128] 매핑들은 런타임을 구동하는 양방향 엔티티 SQL 뷰들로 컴파일된다. *쿼리 뷰들(query views)*은 테이블들에 의하여 엔티티들을 표현하는 반면, *업데이트 뷰들(update views)*은 엔티티들에 의하여 테이블들을 표현한다.

[0129] 업데이트 뷰들은 다소 반직관적(counterintuitive)이다. 왜냐하면 그것들은 가상 구성들(virtual constructs)에 의하여 지속 데이터를 특정하지만, 나중에 알 수 있는 바와 같이, 그것들은 세련된 방식으로 업데이트들을 지원하기 위해 이용될 수 있기 때문이다. 생성된 뷰들은 명확히 정의된 의미에서 매핑을 '고려(respect)'하고 다음의 속성들을 갖는다(프리젠테이션은 약간 단순화된다는 것에 유의한다 – 특히, 지속 상태는 가상 상태에 의해 완전히 결정되지 않는다):

[0130] Entities = QueryViews(Tables)

[0131] Tables = UpdateViews(Entities)

[0132] Entities = QueryViews(UpdateViews(Entities))

[0133] 마지막 조건은 *라운드트리핑 조건(roundtripplng criterion)*으로, 이것은 모든 엔티티 데이터가 지속될 수 있고 무손실 방식으로 데이터베이스로부터 리어셈블(reassemble)될 수 있다는 것을 보증한다. 엔티티 프레임워크에 포함된 매핑 컴파일러는 생성된 뷰들이 라운드트리핑 조건을 만족시키는 것을 보증한다. 그것은 입력 매핑으로부터 그러한 뷰들이 생성될 수 없는 경우에는 오류를 일으킨다.

[0134] 도 6은 도 5의 매핑을 위한 매핑 컴파일러에 의해 생성된 양방향 뷰들 – 쿼리 및 업데이트 뷰들 – 을 보여준

다. 일반적으로, 뷰들은, 요구된 데이터 변환들을 명시적으로 특정하기 때문에, 입력 매핑보다 상당히 더 복잡할 수 있다. 예를 들면, QV_1에서 ESalesOrders 엔티티 세트는 SSalesOrders 테이블로부터 구성되고, 따라서 ESalesOrder 또는 EStoreSalesOrder는 IsOnline 플래그가 참(true)인지의 여부에 따라서 인스턴스화된다. 관계형 테이블들로부터 ESalesPersons를 리어셈블하기 위해서는, SSalesPersons, SEmployees, 및 SContacts 테이블들(QV_3) 간의 조인(join)을 수행할 필요가 있다.

[0135] 라운드트리핑 조건을 만족시키는 쿼리 및 업데이트 뷰들을 손으로 작성하는 것은 까다롭고 상당한 데이터베이스 전문 지식을 필요로 하므로, 엔티티 프레임워크에 대한 본 실시예들은 기본 제공 매핑 컴파일러(built-in mapping complier)에 의해 생성된 뷰들만을 수용한다(비록 대안 실시예들에서는 다른 컴파일러에 의해 또는 손에 의해 생성된 뷰들을 수용하는 것이 확실히 그럴듯하지만).

[0136] **매핑 컴파일러**

[0137] 엔티티 프레임워크는 EDM 스키마, 저장소 스키마, 및 매핑(매핑 아티팩트들에 대해서는 본 명세서의 메타데이터 섹션에서 논의된다)으로부터 쿼리 및 업데이트 뷰들을 생성하는 매핑 컴파일러를 포함한다. 이들 뷰들은 쿼리 및 업데이트 파이프라인들에 의해 소비된다. 이 컴파일러는 디자인 타임에 또는 제1 쿼리가 EDM 스키마에 대하여 실행되는 경우의 런타임에 호출될 수 있다. 이 컴파일러에서 이용되는 뷰 생성 알고리즘은 정확한 재작성(rewritings)을 위해 뷰를 이용한 쿼리 응답(answering-queries-using-views) 기법들에 기초한다.

[0138] 쿼리 처리

[0139] **쿼리 언어**

[0140] 일 실시예에서, 엔티티 프레임워크는 복수의 쿼리 언어와 작용하도록 디자인될 수 있다. 동일하거나 유사한 원리들이 다른 실시예들에도 확장될 수 있다고 이해하고, 여기서는 엔티티 SQL 및 LINQ 실시예들에 대하여 보다 상세히 설명한다.

[0141] *엔티티 SQL*

[0142] 엔티티 SQL은 EDM 인스턴스들을 쿼리 및 조작하기 위해 디자인된 SQL의 파생물이다. 엔티티 SQL은 표준 SQL을 다음의 점들에서 확장한다.

[0143] 1. *EDM 구성들(constructs)*(엔티티, 관계, 복잡 타입 등)*에 대한 네이티브 지원*: 생성자(constructors), 멤버 액세서(member accessors), 타입 질의(type interrogation), 관계 탐색(relationship navigation), 네스트/언네스트(nest/unnest) 등.

[0144] 2. *네임스페이스*. 엔티티 SQL은 (XQurey 및 다른 프로그래밍 언어와 유사하게) 타입들 및 함수들에 대한 그룹핑 구성으로서 네임스페이스를 이용한다.

[0145] 3. *확장 가능한 함수*. 엔티티 SQL은 어떠한 기본 함수도 지원하지 않는다. 모든 함수들(*min, max, substring* 등)은 네임스페이스에서 외부적으로 정의되고, 통상적으로 기초가 되는 저장소로부터 쿼리 내로 *가져와진다(imported)*.

[0146] 4. SQL에 비하여 서브쿼리들(sub-queries) 및 다른 구성들에 대한 더 많은 직교 처리

[0147] 엔티티 프레임워크는, 예를 들면, EntityClient 프로바이더 계층에서, 및 개체 서비스 컴포넌트에서 쿼리 언어로서 엔티티 SQL을 지원할 수 있다. 샘플 엔티티 SQL은 본 명세서의 프로그래밍 패턴 섹션에 제시되어 있다.

[0148] *언어 통합 쿼리(Language Integrated Query; LINQ)*

[0149] 언어 통합 쿼리, 즉 LINQ는 C# 및 비주얼 베이직 등의 주류 프로그래밍 언어들에 쿼리 관련 구성들을 도입하는 .NET 프로그래밍 언어들에서의 혁신이다. 쿼리 표현들은 외부 도구 또는 언어 전처리기에 의해 처리되지 않고 대신에 언어들 자체의 제1 클래스 표현이다. LINQ는 *쿼리 식들(query expressions)*이 풍부한 메타데이터, 컴파일 타임 구문 체킹, 정적인 타이핑 및 이전에는 명령형 코드만이 이용할 수 있었던 인텔리센스(IntelliSense)로부터 이익을 얻는 것을 허용한다. LINQ는 트래버설(traversal), 필터, 조인(join), 프로젝션(projection), 정렬(sorting) 및 그룹핑(grouping) 연산들이 임의의 .NET 기반 프로그래밍 언어에서 직접적이면서도 선언적인 방

식으로 표현될 수 있게 하는 범용 *표준 쿼리 연산자들(standard query operators)*의 세트를 정의한다. 비주얼 베이직 또는 C# 등의 .NET 언어들은 또한 쿼리 이해(query comprehensions) – 표준 쿼리 연산자들을 이용하는 언어 구문 확장들 – 를 지원한다. C#에서 LINQ를 이용한 쿼리의 예는 본 명세서의 프로그래밍 패턴 섹션에서 제시된다.

[0150] **정규 명령 트리(Canonical Command Trees)**

[0151] 일 실시예에서, 정규 명령 트리 – 보다 간단하게는, 명령 트리 – 는 엔티티 프레임워크 내의 모든 쿼리들의 프로그램적 (트리) 표현일 수 있다. 엔티티 SQL 또는 LINQ에 의하여 표현된 쿼리들은 먼저 구문 분석(parse)되고 명령 트리들로 변환된다; 모든 후속 처리는 명령 트리들 상에서 수행될 수 있다. 엔티티 프레임워크는 또한 쿼리들이 명령 크리 구성/편집 API들을 통하여 동적으로 구성(또는 편집)되는 것을 허용할 수 있다. 명령 트리들은 쿼리, 삽입, 업데이트, 삭제, 및 프로시저 호출을 나타낼 수 있다. 명령 트리는 하나 또는 그 이상의 식(Expressions)으로 구성된다. 식은 단순히 어떤 계산을 나타낸다 – 엔티티 프레임워크는 상수, 매개 변수, 산술 연산, 관계형 연산(프로젝션, 필터, 조인 등)을 포함하는 각종의 식을 제공할 수 있다. 마지막으로, 명령 트리들은 EntityClient 프로바이더와 기초가 되는 저장소 특정 프로바이더 간의 쿼리들에 대한 통신 수단으로서 이용될 수 있다.

[0152] **쿼리 파이프라인**

[0153] 엔티티 프레임워크의 일 실시예에서의 쿼리 실행은 데이터 저장소들에 위임될 수 있다. 엔티티 프레임워크의 쿼리 처리 인프라는 엔티티 SQL 또는 LINQ 쿼리를, 보다 단순한 쿼리들의 플랫(flat) 결과들을 보다 풍부한 EDM 구조들로 형상을 고치는(reshape) 데 이용되는 추가의 어셈블리 정보와 함께, 기초가 되는 저장소에 의해 평가될 수 있는 하나 또는 그 이상의 기본, 관계형만의 쿼리들로 분해하는 책임이 있다.

[0154] 엔티티 프레임워크는, 예를 들면, 저장소들이 SQL 서버 2000의 성능과 유사한 성능을 지원해야 한다고 가정할 수 있다. 쿼리들은 이 프로파일에 적합한 보다 단순한 플랫-관계형 쿼리들(flat-relational queries)로 분해될 수 있다. 엔티티 프레임워크의 다른 실시예들은 저장소들이 쿼리 처리의 보다 큰 부분들을 맡는 것을 허용할 수 있다.

[0155] 전형적인 쿼리는 다음과 같이 처리될 수 있다.

[0156] *구문 및 의미 분석(Syntax and Semantic Analysis).* 엔티티 SQL 쿼리는 먼저 구문 분석되고 메타데이터 서비스 컴포넌트로부터의 정보를 이용하여 의미론적으로 분석된다. LINQ 쿼리들은 구문 분석되고 적당한 언어 컴파일러의 일부로서 분석된다.

[0157] *정규 명령 트리로의 변환(Conversion to a Canonical Command Tree).* 이제 쿼리는 그것이 원래 어떻게 표현되었는지에 관계없이 명령 트리로 변환되고 유효성 검사(validate)된다.

[0158] *매핑 뷰 언폴딩(Mapping View Unfolding).* 엔티티 프레임워크 내의 쿼리들은 개념적(EDM) 스키마들을 대상으로 한다. 이들 쿼리들은 대신에 기초가 되는 데이터베이스 테이블들 및 뷰들을 참조하도록 변환되어야 한다. 이 프로세스 – 매핑 뷰 언폴딩이라고 함 – 는 데이터베이스 시스템들에서의 뷰 언폴딩 메커니즘과 유사하다. EDM 스키마와 데이터베이스 스키마 간의 매핑들은 쿼리 및 업데이트 뷰들로 컴파일된다. 그 후 쿼리 뷰는 사용자 쿼리에서 언폴드된다 – 이제 쿼리는 데이터베이스 테이블들 및 뷰들을 대상으로 한다.

[0159] *구조화 타입 제거(Structured Type Elimination).* 구조화 타입들에의 모든 참조들은 이제 쿼리로부터 제거되고, (결과 어셈블리를 가이드하는) 리어셈블리 정보(reassembly information)에 추가된다. 이것은 타입 생성자, 멤버 액세서, 타입 질의 식들을 참조한다.

[0160] *프로젝션 가지치기(Projection Pruning).* 쿼리는 분석되고, 쿼리 내의 참조되지 않은 식들은 제거된다.

[0161] *네스트 풀업(Nest Pull-up).* 쿼리 내의 (네스트된 컬렉션들을 구성하는) 임의의 네스팅 연산들은 플랫 관계형 연산자들만을 포함하는 서브트리 위의 쿼리 트리의 루트로 밀어올려진다. 통상적으로, 네스팅 연산은 좌측 외부 조인(left outer join)(또는 외부 어플라이(outer apply))으로 변환되고, 그 후 후속 쿼리로부터의 플랫 결과들은 적절한 결과들로 리어셈블된다(아래에서 결과 어셈블리 참조).

[0162] *변환(Transformations).* 쿼리를 단순화하기 위해 발견적 변환들의 세트가 적용된다. 이들은 필터 푸시다운(filter pushdowns), 어플라이-조인 변환(apply to join conversions), 경우 식 폴딩(case expression folding) 등을 포함한다. 중복된 조인들(셀프-조인(self-joins), 기본 키, 외래 키 조인)은 이 단계에서 제거된다. 여기에서 쿼리 처리 인프라는 어떤 비용 기반 최적화도 수행하지 않는다는 것에 유의한다.

[0163] *프로바이더 특정 명령으로의 변환(Translation into Provider-Specific Commands).* 이제 쿼리들(즉, 명령 트리)은 아마도 프로바이더의 네이티브 SQL 방언(dialect)의 프로바이더 특정 명령을 생성하기 위해 프로바이더들에게 넘겨진다. 이 단계를 SQLGen이라 한다.

[0164] *실행(Execution).* 프로바이더 명령들이 실행된다.

[0165] *결과 어셈블리(Result Assembly).* 그 후 프로바이더들로부터의 결과들(DataReaders)은 보다 일찍 수집된 어셈블리 정보를 이용하여 적절한 형태로 형상이 고쳐지고, 단 하나의 DataReader가 호출자에게 반환된다.

[0166] *구체화(Materialization).* 개체 서비스 컴포넌트를 통하여 발행된 쿼리들에 대하여, 그 결과들은 그 후 적절한 프로그래밍 언어 개체들로 구체화된다.

[0167] **SQLGen**

[0168] 이전 세션에서 언급한 바와 같이, 쿼리 실행은 기초가 되는 저장소에 위임될 수 있다. 그러한 실시예들에서, 쿼리는 먼저 저장소에 적합한 형태로 변환되어야 한다. 그러나, 상이한 저장소들은 SQL의 상이한 방언들을 지원하고, 엔티티 프레임워크가 그 모두를 네이티브로 지원하는 것은 비현실적이다. 대신에, 쿼리 파이프라인은 명령 트리 형태의 쿼리를 저장소 프로바이더에 넘겨줄 수 있다. 그 후 저장소 프로바이더는 명령 트리를 네이티브 명령으로 변환할 수 있다. 이것은 명령 트리를 프로바이더의 네이티브 SQL 방언으로 변환함으로써 성취될 수 있다 – 따라서 이 단계에 대한 용어 SQLGen, 그 결과의 명령은 그 후 관련 결과들을 생성하도록 실행될 수 있다.

[0169] 업데이트 처리

[0170] 이 섹션은 예시적인 ADO.NET 엔티티 프레임워크에서 업데이트 처리가 어떻게 수행될 수 있는지를 설명한다. 일 실시예에서는, 업데이터 처리에 대한 2개의 단계, 컴파일 타임 및 런타임이 있다. 본 명세서에서 제공된 양방향 뷰 섹션에서는, 매핑 명세를 뷰 식들의 컬렉션으로 컴파일하는 프로세스를 설명하였다. 이 섹션은 이들 뷰 식들이 개체 계층에서 수행된 개체 수정들(또는 EDM 계층에서의 엔티티 SQL DML 업데이트들)을 관계형 계층에서의 등가의 SQL 업데이트들로 변환하기 위해 런타임에서 어떻게 이용되는지를 설명한다.

[0171] **뷰 관리를 통한 업데이트(Updates via View Maintenance)**

[0172] 예시적인 ADO.NET 매핑 아키텍처에서 이용되는 통찰 중 하나는 양방향 뷰들을 통하여 업데이트를 전파하기 위해 구체화된 뷰 관리 알고리즘이 이용될 수 있다는 것이다. 이 프로세스는 도 7에 예시되어 있다.

[0173] 도 7의 오른편에 도시된 데이터베이스 내의 테이블들은 지속 데이터를 유지한다. 도 7의 왼편에 도시된 EntityContainer는 이 지속 데이터의 가상 상태를 나타낸다. 왜냐하면 통상적으로 EntitySet들 내의 엔티티들의 작은 비율만이 클라이언트에서 구체화되기 때문이다. 목표는 엔티티들의 상태의 업데이트 ΔEntities를 테이블들의 지속 상태의 업데이트 ΔTables로 변환하는 것이다. 이 프로세스를 점진적 뷰 관리(incremental view maintenance)라 부른다. 왜냐하면 업데이트는 엔티티의 변경된 양태들을 나타내는 업데이트 ΔEntities에 기초하여 수행되기 때문이다.

[0174] 이것은 다음의 2 단계를 이용하여 행해질 수 있다.

[0175] *1. 뷰 관리(View maintenance):*

[0176] ΔTables = ΔUpdateViews(Entities, ΔEntities)

[0177] *2. 뷰 언폴딩(View unfolding):*

[0178] ΔTables = ΔUpdateViews(QueryViews(Tables), ΔEntities)

[0179] 단계 1에서는, 업데이트 뷰들에 뷰 관리 알고리즘이 적용된다. 이것은 델타 식들 ΔUpdateViews의 세트를 생성하고, 이것은 ΔEntities 및 엔티티들의 스냅숏으로부터 ΔTables를 얻는 방법을 알려준다. 후자는 클라이언트에서 충분히 구체화되지 않으므로, 단계 2에서 델타 식들을 쿼리 뷰들과 조합하기 위해 뷰 언폴딩이 이용된다. 이들 단계들은, 함께, 초기 데이터베이스 상태 및 엔티티들에 대한 업데이트를 입력으로서 수취하고, 데이터베이스에 대한 업데이트들을 계산하는 식을 생성한다.

[0180] 이러한 접근법은 동시 개체 및 세트 기반 업데이트들(object-at-a-time and set-based updates)(즉, 데이터 조작 문들을 이용하여 표현되는 것들) 양쪽 모두에 대하여 작용하는 깔끔하고 균일한 알고리즘을 가져오고, 강건한 데이터베이스 기술을 이용한다. 실제로는, 단계 1은 종종 업데이트 변환을 위해 충분하다. 왜냐하면 많은 업데이트들은 현재의 데이터베이스 상태에 직접적으로 의존하지 않고; 그러한 상황에서 ΔTables = ΔUpdateViews(ΔEntities)이기 때문이다. 만일 ΔEntities가 캐싱된 엔티티들에 대한 동시 개체 수정들(object-at-a-time modifications)의 세트로서 주어진다면, 단계 1은 ΔUpdateViews 식들을 계산하기보다는 수정된 엔티티들에 대해 직접적으로 뷰 관리 알고리즘들을 실행함으로써 더욱 최적화될 수 있다.

[0181] **개체 상의 업데이트 변환**

[0182] 위에서 개설된 접근법을 설명하기 위해, 적어도 5년 동안 회사와 함께 한 자격이 있는 판매원에게 보너스와 진급을 제공하는 다음의 예를 생각해보자.

```
using(AdventureWorksDB aw =
      new AdventureWorksDB(…)) {
  // People hired at least 5 years ago
  Datetime d = DateTime.Today.AddYears(-5);
  var people = from p in aw.SalesPeople
               where p.HireDate < d
               select p;

  foreach(SalesPerson p in people) {
    if(HRWebService.ReadyForPromotion(p)) {
      p.Bonus += 10;
      p.Title = "Senior Sales Representative";
    }
  }
  aw.SaveChanges(); // push changes to DB
}
```

[0183]

[0184] AdventureWorksDB는 데이터베이스 연결(database connection), 메타데이터 작업 영역(metadata workspace), 및 개체 캐시 데이터 구조(object cache data structure)를 수용하고 SaveChanges 메서드를 익스포즈하는, ObjectContext라 불리는 일반 개체 서비스 클래스로부터 파생하는 도구 생성 클래스(tool-generated class)이다. 개체 서비스 섹션에서 설명한 바와 같이, 개체 캐시는 각각이 다음의 상태들: (캐시로부터) *분리(detached)*, *추가(added)*, *불변(unchanged)*, *수정(modified)*, 및 *삭제(deleted)* 중 하나에 있는 엔티티들의 목록을 유지한다. 상기 코드 프래그먼트는 SEmployees 및 SSalesPersons 테이블들에 각각 저장되어 있는 ESalesPerson 개체들의 직위(title) 및 보너스 속성들을 수정하는 업데이트를 기술한다. 개체 업데이트들을 SaveChanges 메서드에의 호출에 의해 트리거되는 대응하는 테이블 업데이트들로 변환하는 프로세스는 다음의 4개의 단계를 포함할 수 있다:

[0185] *변경 목록 생성(Change List Generation).* 엔티티 세트마다의 변경들의 목록이 개체 캐시로부터 생성된다. 업데이트들은 삭제 및 삽입된 요소들의 목록으로서 표현된다. 추가된 개체들은 삽입(inserts)이 된다. 삭제된 개체들은 삭제(deletes)가 된다.

[0186] *값 식 전파(Value Expression Propagation).* 이 단계는 변경들의 목록 및 업데이트 뷰들(메타데이터 작업 영역에 유지됨)을 취하고, 점진적 구체화된 뷰 관리 식(incremental materialized view maintenance expressions)인 ΔUpdateViews를 이용하여, 개체 변경들의 목록을 기초가 되는 피작용 테이블들(affected tables)에 대한 대수 기본 테이블 삽입 및 삭제 식들(algebraic base table insert and delete expressions)의 시퀀스로 변환한

다. 이 예에서, 관련 업데이트 뷰들은 도 6에 도시된 UV_2 및 UV_3이다. 이들 뷰는 단순 프로젝트-선택 쿼리들이므로, 뷰 관리 규칙들을 적용하는 것은 수월하다. 삽입(Δ^+) 및 삭제(Δ^-)에 대하여 동일한 다음의 ΔUpdateViews 식들이 얻어진다:

```
ΔSSalesPersons =    SELECT p.Id, p.Bonus
           FROM ΔESalesPersons AS p
ΔSEmployees =  SELECT p.Id, p.Title
           FROM ΔESalesPersons AS p
ΔSContacts =    SELECT p.Id, p.Name, p.Contact.Email,
           p.Contact.Phone FROM ΔESalesPersons AS p
```

[0187]

[0188] 위에 제시된 루프가 엔티티 E_{old} = EsalesPersons(1, 20, "", "Alice", Contact("a@sales", NULL))를 E_{new} = EsalesPersons(1, 30, "Senior ...", "Alice", Contact("a@sales", NULL))로 업데이트했다고 가정하자. 그러면, 초기 델타는 삽입에 대해서는 Δ^+ESalesOrders = {E_{new}}이고 삭제에 대해서는 Δ^-ESalesOrders = {E_{old}}이다. Δ^+SSalesPersons = {(1,30)}, Δ^-SSalesPersons = {(1,20)}이 얻어진다. 그 후 SSalesPersons 테이블 상의 계산된 삽입들 및 삭제들은 보너스(Bonus) 값을 30으로 설정하는 단 하나의 업데이트로 조합된다. SEmployees 상의 델타들도 유사하게 계산된다. SContacts에 대해서는, Δ^+SContacts = Δ^-SContacts이므로, 업데이트가 요구되지 않는다.

[0189] 피작용 기본 테이블 상에서 델타들을 계산하는 것 외에, 이 단계는 (a) 참조 무결성 제약(referential integrity constraints)를 고려하여, 테이블 업데이트들이 수행되어야 하는 정확한 순서화(ordering), (b) 데이타베이스에 대한 최종 업데이트들을 제시하기 전에 필요한 저장소 생성 키들의 검색, 및 (c) 낙관적인 동시성 제어(optimistic concurrency control)를 위한 정보의 수집에 대한 책임이 있다.

[0190] *SQL DML 또는 저장된 프로시저 호출 생성.* 이 단계는 삽입 및 삭제된 델타들 및 동시성 처리에 관련된 추가의 주석들의 목록을 SQL DML 또는 저장된 프로시저 호출들의 시퀀스로 변환한다. 이 예에서, 피작용 판매원에 대하여 생성된 업데이트 문들은 다음과 같다:

```
BEGIN TRANSACTION
UPDATE [dbo].[SSalesPersons] SET [Bonus]=30
WHERE [SalesPersonID]=1
UPDATE [dbo].[SEmployees]
SET [Title]= N'Senior Sales Representative'
WHERE [EmployeeID]=1
END TRANSACTION
```

[0191]

[0192] *캐시 동기화.* 일단 업데이트가 수행되면, 캐시의 상태는 데이터베이스의 새로운 상태와 동기화된다. 따라서, 필요하다면, 새로운 수정된 관계형 상태를 그의 대응하는 엔티티 및 개체 상태로 변환하기 위해 미니 쿼리 처리(mini-query-processing) 단계가 수행된다.

[0193] 메타데이터

[0194] 메타데이터 서브시스템은 데이터베이스 카탈로그와 유사하고, 엔티티 프레임워크의 디자인 타입 및 런타임 요구들을 만족시키도록 디자인된다.

[0195] **메타데이터 아티팩트**

[0196] 메타데이터 아티팩트는 예를 들면 다음을 포함할 수 있다:

[0197] 개념적 스키마(CSDL 파일): 개념적 스키마는 CSDL 파일(Conceptual Schema Definition Language)에서 정의될 수 있고 애플리케이션의 데이터의 개념적 뷰를 기술하는 엔티티 세트들 및 EDM 타입들(엔티티 타입, 관계)을 포함한다.

[0198] 저장소 스키마(SSDL 파일): 저장소 스키마 정보(테이블, 열, 키 등)는 CSDL 어휘 용어들을 이용하여 표현될 수 있다. 예를 들면, EntitySets는 테이블들을 나타내고, 속성들을 열들을 나타낸다. 이들은 SSDL(Store Schema Definition Language) 파일에서 정의될 수 있다.

[0199] C-S 매핑 명세(MSL 파일): 개념적 스키마와 저장소 스키마 간의 매핑은 통상적으로 MSL 파일(Mapping Specification Language) 내의 매핑 명세에서 캡처된다. 이 명세는 매핑 컴파일러에 의해 쿼리 및 업데이트 뷰들을 생성하는 데 이용된다.

[0200] 프로바이더 명시(Provider Manifest): 프로바이더 명시는 각 프로바이더에 의해 지원되는 기능에 대한 설명을 제공할 수 있고, 다음의 예시적인 정보를 포함할 수 있다:

[0201] 1. 프로바이더에 의해 지원되는 원시 타입들(varchar, int 등), 및 그것들에 대응하는 EDM 타입들(string, int32 등).

[0202] 2. 프로바이더에 대한 기본 제공 기능들(및 그들의 서명들).

[0203] 이 정보는 쿼리 분석의 일부로서 엔티티 SQL 파서(parser)에 의해 이용될 수 있다. 이들 아티팩트 외에, 메타데이터 서브시스템은 또한 생성된 개체 클래스들, 및 이들과 대응하는 개념적 엔티티 타입들 간의 매핑들을 계속 추적할 수 있다.

[0204] **메타데이터 서비스 아키텍처**

[0205] 엔티티 프레임워크에 의해 소비되는 메타데이터는 상이한 포맷으로 상이한 소스들로부터 올 수 있다. 메타데이터 서브시스템은 메타데이터 런타임이 상이한 메타데이터 지속 포맷들/소스들의 상세들과 관계없이 작용하는 것을 허용하는 통합된 저레벨 메타데이터 인터페이스들의 세트 상에 구축될 수 있다.

[0206] 예시적인 메타데이터 서비스들은 다음을 포함할 수 있다:

[0207] 상이한 메타데이터 타입들의 열거.

[0208] 키에 의한 메타데이터 검색.

[0209] 메타데이터 브라우징/탐색.

[0210] (예를 들어, 쿼리 처리를 위한) 과도 메타데이터(transient metadata)의 생성.

[0211] 세션 독립적 메타데이터 캐싱 및 재사용.

[0212] 메타데이터 서브시스템은 다음의 컴포넌트들을 포함한다. 메타데이터 캐시는 상이한 소스들로부터 검색된 메타데이터를 캐싱하고, 메타데이터를 검색 및 조작하기 위한 공통의 API를 소비자들에게 제공한다. 메타데이터는 상이한 형태로 표현되고, 상이한 위치들에 저장될 수 있으므로, 메타데이터 서브시스템은 유리하게는 로더 인터페이스(loader interface)를 지원할 수 있다. 메타데이터 로더들은 로더 인터페이스를 구현하고, 적절한 소스(CSDL/SSDL 파일 등)로부터 메타데이터를 로딩하는 책임이 있다. 메타데이터 작업 영역은 애플리케이션에 대한 완전한 메타데이터의 세트를 제공하기 위해 몇 개의 메타데이터를 집계한다. 메타데이터 작업 영역은 통상적으로 개념적 모델, 저장소 스키마, 개체 클래스, 및 이들 구성 간의 매핑들에 관한 정보를 포함한다.

[0213] 도구들

[0214] 일 실시예에서, 엔티티 프레임워크는 또한 개발 생산성을 증대시키기 위해 디자인 타임 도구들의 컬렉션을 포함할 수 있다. 예시적인 도구들은 다음과 같다:

[0215] *모델 디자이너(Model designer)*: 애플리케이션의 개발의 초기 단계들 중 하나는 개념적 모델의 정의이다. 엔티티 프레임워크는 애플리케이션 디자이너들 및 분석자들이 엔티티들 및 관계들에 의하여 그들의 애플리케이션의 주요 개념들을 기술하는 것을 허용한다. 모델 디자이너는 이 개념적 모델링 작업이 대화형으로 수행되는 것을 허용하는 도구이다. 디자인의 아티팩트들은 데이터베이스에서 그의 상태를 지속할 수 있는 메타데이터 컴포넌트에서 직접 캡처된다. 모델 디자이너는 또한 (CSDL에 의하여 특정되는) 모델 기술들(model descriptions)을 생성 및 소비할 수 있고, 관계형 메타데이터로부터 EDM 모델들을 합성할 수 있다.

[0216] *매핑 디자이너(Mapping designer)*: 일단 EDM 모델이 디자인되면, 개발자는 개념적 모델이 어떻게 관계형 데이터에 매핑되는지를 특정할 수 있다. 이 작업은 도 8에 도시된 사용자 인터페이스를 제시할 수 있는 매핑 디자이너에 의해 용이하게 된다. 매핑 디자이너는 개발자들이 도 8의 사용자 인터페이스의 왼편에 제시된 엔티티 스키마 내의 엔티티들 및 관계들이 사용자 인터페이스의 오른편에 제시된 데이터베이스 스키마에 반영된 바와 같이 데이터베이스 내의 테이블들 및 열들에 어떻게 매핑되는지를 기술하는 것을 돕는다. 도 8의 중간 부분에 제시된 그래프 내의 연결들은 엔티티 SQL 쿼리들의 동등들(equalities)로서 선언적으로 특정된 매핑 식들을 시각화한다. 이들 식들은 쿼리 및 업데이트 뷰들을 생성하는 양방향 매핑 컴파일 컴포넌트에의 입력이 된다.

[0217] *코드 생성(Code generation)*: EDM 개념적 모델은 ADO.NET 코드 패턴들(명령, 연결, 데이터 리더)에 기초한 잘 알려진 상호작용 모델을 제공하므로 다수의 애플리케이션들에 대하여 충분하다. 그러나, 다수의 애플리케이션들은 강 타입의 개체들로서의 데이터와 상호작용하기를 선호한다. 엔티티 프레임워크는 입력으로서 EDM 모델을 취하고 엔티티 타입들에 대하여 강 타입의 CLR 클래스들을 생성하는 코드 생성 도구들의 세트를 포함한다. 코드 생성 도구들은 또한 모델(예컨대, ObjectQuery<SalesPerson>)에 의해 정의된 모든 엔티티 및 관계 세트들에 대하여 강 타입의 컬렉션들을 익스포즈하는 강 타입의 개체 컨텍스트(예컨대, AdventureWorksDB)를 생성할 수 있다.

[0218] **추가의 양태들 및 실시예들**

[0219] 매핑 서비스

[0220] 일 실시예에서, 도 1의 114와 같은 매핑 컴포넌트는 매핑의 모든 양태들을 관리하고 엔티티 클라이언트 프로바이더(111)에 의해 내부적으로 사용된다. *매핑(mapping)*은 2개의 잠재적으로 상이한 타입 공간들 내의 구성들 간의 변환을 논리적으로 특정한다. 예를 들면, 엔티티 – 이 용어가 위에서 사용될 때 개념적 공간에서의 – 는 도 8에 그래프식으로 도시된 바와 같이 저장 공간 내의 데이터베이스 테이블들에 의하여 특정될 수 있다.

[0221] *규정된(prescribed)* 매핑들은 시스템이 자동으로 구성들에 대한 적절한 매핑들을 결정하는 것들이다. *비규정된(Non-prescribed)* 매핑들은 애플리케이션 디자이너들이 매핑의 다양한 면(facet)들을 제어하는 것을 허용한다. 매핑은 수개의 면들을 가질 수 있다. 매핑의 종점들(엔티티, 테이블 등), 매핑되는 속성들의 세트, 업데이트 거동, 지연 로딩 등의 런타임 효과, 업데이트들에 대한 충돌 해결 거동(conflict-resolution behavior) 등은 그러한 면들의 일부분의 목록일 뿐이다.

[0222] 일 실시예에서, 매핑 컴포넌트(114)는 매핑 뷰들을 생성할 수 있다. 저장소 공간과 스키마 공간 간의 매핑을 생각해보자. 엔티티는 하나 또는 그 이상의 테이블들로부터의 행들로 구성된다. *쿼리 뷰들(Query Views)*은 스키마 공간 내의 엔티티를 저장소 공간 내의 테이블들에 의하여 쿼리로서 표현한다. 엔티티들을 쿼리 뷰들을 평가함으로써 구체화될 수 있다.

[0223] 엔티티들에 세트에 대한 변경들이 대응하는 저장소 테이블들에 도로 반영될 필요가 있을 경우, 그 변경들은 쿼리 뷰들을 통하여 역 방식(reverse fashion)으로 전파될 수 있다. 이것은 데이터베이스 내의 뷰-업데이트 문제에도 유사하다 – 업데이트 전파 프로세스는 논리적으로 쿼리 뷰(들)의 역(inverse)(들)에 대하여 업데이트를 수행한다. 이 때문에, *업데이트 뷰(update views)*의 개념을 도입한다 – 이 뷰들은 저장소 테이블들을 엔티티들에 의하여 기술하고, 쿼리 뷰(들)의 역으로서 간주될 수 있다.

[0224] 그러나, 많은 경우에, 실제로 흥미가 있는 것은 점진적 변경(incremental changes)이다. *업데이트 델타 뷰들(Update Delta Views)*은 테이블들에 대한 변경들을 대응하는 엔티티 컬렉션들에 대한 변경들에 의하여 기술하는 뷰들(쿼리들)이다. 그러므로, 엔티티 컬렉션들(또는 애플리케이션 개체들)에 대한 업데이트 처리는 업데이트 델타 뷰들을 평가함으로써 테이블들에 대한 적절한 변경들을 계산하고, 그 후 이들 변경들을 테이블들에 적용하는 것을 포함한다.

[0225] 유사한 방식으로, *쿼리 델타 뷰들(Query Delta Views)*은 엔티티 컬렉션들에 대한 변경들을 기초가 되는 테이블들에 대한 변경들에 의하여 기술한다. 무효화(invalidations), 및 보다 일반적으로, 통지들(notifications)은 쿼리 델타 뷰들의 사용을 필요로 할 수 있는 시나리오들이다.

[0226] 데이터베이스 내의 뷰들과 마찬가지로, 쿼리들로서 표현된 매핑 뷰들은 그 후 사용자 쿼리들에 의해 구성될 수 있어, 결국 매핑들의 처리가 보다 일반화된다. 마찬가지로, 쿼리들로서 표현된 매핑 델타 뷰들은 업데이트들을

처리하는 보다 일반적이고 우아한 접근법을 허용한다.

[0227] 일 실시예에서, 매핑 뷰들의 능력은 억제될 수 있다. 매핑 뷰에서 이용되는 쿼리 구성들은 엔티티 프레임워크에 의해 지원되는 모든 쿼리 구성들의 서브세트에 불과할 수 있다. 이것은 보다 단순하고 보다 효율적인 매핑 식들을 허용한다 – 특히 델타 식의 경우에.

[0228] 델타 뷰들은 업데이트(및 쿼리) 뷰들로부터 업데이트(및 쿼리) 델타 뷰들을 생성하는 대수 변경 계산 방식을 이용하여 매핑 컴포넌트(114)에서 계산될 수 있다. 대수 변경 계산 방식의 또 다른 양태들에 대해서는 후술한다.

[0229] 업데이트 델타 뷰들은 엔티티 프레임워크가 계산 애플리케이션들에 의해 행해진 엔티티 변경들을 데이터베이스 내의 저장소 레벨 업데이트들로 자동 변환함으로써 업데이트들을 지원하는 것을 허용한다. 그러나, 많은 경우에, 매핑은 성능 및/또는 데이터 무결성을 위해 추가의 정보로 보강될 수 있다.

[0230] 일부 경우에, 엔티티들에 대한 업데이트들을 그의 기초가 되는 저장소 테이블의 일부 또는 전부에 직접 매핑하는 것은 바람직하지 않을 수 있다. 그러한 경우, 신뢰 경계(trust boundary)를 유지할 뿐만 아니라 데이터 유효성 검사를 가능하게 하기 위해 업데이트들은 저장된 프로시저들을 통과해야만 한다. 매핑은 저장된 프로시저들의 명세가 엔티티들에 대한 업데이트들 및 쿼리들을 처리하는 것을 허용한다.

[0231] 매핑은 또한 개체 서비스들(131)에서의 낙관적인 동시성 제어에 대한 지원을 제공할 수 있다. 구체적으로, 엔티티의 속성들은 타임스탬프 또는 버전 필드 등의 *동시성 제어(concurrency-control)* 필드들로서 마킹될 수 있고, 이들 개체들에 대한 변경들은 저장소에 있는 동시성 제어 필드들의 값들이 엔티티에서와 동일한 경우에만 성공할 것이다. 양쪽의 낙관적인 동시성 제어 필드들은 애플리케이션 개체 계층에서만 관련되고, 저장소 특정 계층(120)에서는 관련되지 않는다는 것에 유의한다.

[0232] 일 실시예에서, 애플리케이션 디자이너들은 매핑의 다양한 양태들을 기술하기 위해 매핑 명세 언어(Mapping Specification Language; MSL)를 이용할 수 있다. 전형적인 매핑 명세는 다음의 섹션들 중 하나 이상을 포함한다.

[0233] 1. *델타(Dalta)* 영역은 클래스, 테이블 및/또는 EDM 타입들에 대한 기술(description)들을 포함할 수 있다. 이들 기술들은 현존하는 클래스/테이블/타입을 기술할 수 있고, 또는 그러한 인스턴스들을 생성하기 위해 이용될 수도 있다. 서버 생성 값들, 제약들, 기본 키 등이 이 섹션의 일부로서 특정된다.

[0234] 2. *매핑(Mapping)* 섹션은 타입 공간들 간의 실제 매핑들을 기술한다. 예를 들면, EDM 엔티티의 각 속성은 테이블(또는 테이블들의 세트)로부터의 하나 또는 그 이상의 열들에 의하여 특정된다.

[0235] 3. *런타임(Runtime)* 영역은 실행을 제어하는 다양한 노브들(knobs), 예를 들어, 낙관적인 동시성 제어 매개 변수들 및 페칭 전략(fetching strategy)를 특정할 수 있다.

[0236] **매핑 컴파일러**

[0237] 일 실시예에서, 도메인 모델링 도구 매핑 컴포넌트(172)는 매핑 명세를 쿼리 뷰, 업데이트 뷰, 및 대응하는 델타 뷰들로 컴파일하는 매핑 컴파일러를 포함할 수 있다. 도 9는 쿼리 및 업데이트 뷰들을 생성하기 위해 MSL을 컴파일하는 것을 도시한다.

[0238] 컴파일 파이프라인은 다음의 단계들을 수행한다:

[0239] 1. API(900)로부터 호출되는 뷰 생성기(View Generator)(902)는 개체 ↔ 엔티티 매핑 정보(901)(MSL에 의하여 특정됨)를 변환하고 O ↔ E(Object to Entity) 공간에서 쿼리 뷰, 업데이트 뷰, 및 대응하는 (쿼리 및 업데이트) 델타 식들(904)을 생성한다. 이 정보는 메타데이터 저장소(908)에 배치될 수 있다.

[0240] 2. 뷰 생성기(View Generator)(906)는 엔티티 ↔ 저장소 매핑 정보(903)(MSL에 의하여 특정됨)를 변환하고 E ↔S(Entity to Store) 공간에서 쿼리 뷰, 업데이트 뷰, 및 대응하는 (쿼리 및 업데이트) 델타 식들(907)을 생성한다. 이 정보는 메타데이터 저장소(908)에 배치될 수 있다.

[0241] 3. 종속성 분석(Dependency Analysis)(909) 컴포넌트는 뷰 생성기(906)에 의해 생성된 뷰들을 조사하고 참조 무결성(referential integrity) 및 다른 그러한 제약들을 위반하지 않는 업데이트들에 대한 일관성 있는 종속성 순서(dependency order)(910)를 결정한다. 이 정보는 메타데이터 저장소(908)에 배치될 수 있다.

[0242] 4. 그 후 뷰들, 델타 식들, 및 종속성 순서(908)는 메타데이터 서비스 컴포넌트(도 1의 112)에 넘겨진다.

[0243] **업데이트 처리**

[0244] 이 섹션은 업데이트 처리 파이프라인을 설명한다. 일 실시예에서, 엔티티 프레임워크는 2종류의 업데이트를 지원할 수 있다. 단일 개체 변경*(single object changes)*은 개체 그래프를 탐색하면서 개개의 개체들에 행해지는 변경들이다. 단일 개체 변경을 위하여, 시스템은 현 트랜잭션에서 생성, 업데이트, 및 삭제된 개체들을 계속 추적한다. 이것은 개체 계층(들)에서만 사용 가능하다. *쿼리 기반 변경(query-based changes)*은, 예를 들어, 테이블들을 업데이트하기 위해 관계형 데이터베이스에서 행해지는 것과 같이, 개체 쿼리에 기초하여 업데이트/삭제 문을 발행함으로써 행해지는 변경들이다. 도 1의 131과 같은 개체 프로바이더들은 쿼리 기반 변경이 아니라 단일 개체 변경을 지원하도록 구성될 수 있다. 한편, 엔티티 클라이언트 프로바이더(111)는 단일 개체 변경이 아니라 쿼리 기반 변경을 지원할 수 있다.

[0245] 도 10은 예시적인 일 실시예에서의 업데이트 처리에 대한 예시를 보여준다. 도 10에서 애플리케이션 계층(1000)에 있는 애플리케이션의 사용자(1001)는 그러한 애플리케이션에 의해 조작된 데이터에 대한 변경들을 저장(1002)할 수 있다. 개체 프로바이더 계층(1010)에서는, 변경 목록이 컴파일된다(1011). 그 변경 목록에 대하여 변경 그룹핑(1012)이 수행된다. 제약 처리(constraint handling)(1013)는 메타데이터 저장소(1017)에 저장되는 제약 정보 및 종속성 모델(1022)을 생성할 수 있다. 확장된 연산들이 실행된다(1014). 동시성 제어 식이 생성되고(1015), 동시성 모델(1023)이 메타데이터 저장소(1017)에 저장된다. 개체-엔티티 변환기(object to entity converter)(1016)는 개체-엔티티 델타 식들(1024)을 메타데이터 저장소(1017)에 저장할 수 있다.

[0246] 엔티티 식 트리(entity expression tree)(1018)가 EDM 프로바이더 계층(1030)에 넘겨진다. 선택적 업데이트 분할기(selective update splitter)(1031)가 특정 업데이트들을 선택하고 그것들을 필요에 따라 분할할 수 있다. EDM 저장소 변환기(1032)는 엔티티-저장소 델타 식들(1033)을 메타데이터 저장소(1036)에 저장할 수 있다. 쿼리 뷰 언폴딩 컴포넌트(1034)는 쿼리 매핑 뷰들(1035)을 메타데이터 저장소(1036)에 저장할 수 있다. 엔티티-저장소 보상(entity to store compensation)(1037)이 수행되고, 저장소 식 트리(1038)가 저장소-프로바이더 계층(1040)에 전달된다.

[0247] 저장소 프로바이더 계층(1040)에서는, 단순화기(simplifier) 컴포넌트(1041)가 먼저 동작할 수 있고, 그에 이어서 SQL 생성 컴포넌트(1042)는 데이터베이스(1044)에서 실행될 SQL 업데이트들(1043)을 생성한다. 임의의 업데이트 결과들은 서버 생성 값들을 처리하기 위한 EDM 프로바이더 계층(1030) 내의 컴포넌트(1039)에 전달될 수 있다. 컴포넌트(1039)는 결과들을 개체 프로바이더 계층 내의 유사한 컴포넌트(1021)에 전달할 수 있다. 마지막으로, 임의의 결과들 또는 업데이트 확인(1003)이 애플리케이션 계층(1000)에 반환된다.

[0248] 전술한 바와 같이, 업데이트 델타 뷰들은 매핑 컴파일의 일부로서 생성된다. 이들 뷰들은 업데이트 처리에서 저장소에 있는 테이블들에 대한 변경들을 식별하기 위해 이용된다.

[0249] 저장소에 있는 관련 테이블들의 세트에 대하여, 엔티티 프레임워크는 유리하게는 특정 순서로 업데이트들을 적용할 수 있다. 예를 들면, 외래 키 제약들의 존재는 변경들이 특정 시퀀스로 적용될 것을 요구할 수 있다. 종속성 분석 단계(매핑 컴파일의 일부) 식별자들은 컴파일 타임에 계산될 수 있는 임의의 종속성 순서화(dependency ordering) 요건들을 식별한다.

[0250] 일부 경우에, 정적인 종속성 분석 기법은, 예를 들어, 순환 참조 무결 성(cyclic referential integrity) 제약들(또는 자기 참조 무결성(self-referential integrity) 제약들)에 대하여 충분하지 않을 수 있다. 엔티티 프레임워크는 낙관적인 접근법을 채택하고, 그러한 업데이트들이 통과하는 것을 허용한다. 런타임에서, 주기가 검출되면, 예외가 제기된다.

[0251] 도 10에 도시된 바와 같이, 애플리케이션 계층(1000)에서의 인스턴스 기반 업데이트를 위한 업데이트 처리 파이프라인은 다음의 단계들을 갖는다:

[0252] 변경 그룹핑(1012): 변경 추적기(change tracker)로부터의 상이한 개체 컬렉션들에 따라 변경들을 그룹핑한다. 예를 들어, 컬렉션 Person으로부터의 모든 변경들은 그 컬렉션에 대한 삽입, 삭제, 및 업데이트 세트로 그룹핑된다.

[0253] 제약 처리(1013): 이 단계는 값 계층에서 어떤 비즈니스 로직도 실행되지 않는다는 사실을 보상하는 임의의 동작들을 수행한다 – 본질적으로, 그것은 개체 계층이 변경 세트를 확장하는 것을 허용한다. (EDM 제약들을 고

려하는) 계단식-삭제 보상 및 종속성 순서화(cascade-delete compensation and dependency ordering)가 여기에서 수행된다.

[0254] 확장된 연산 실행(1014): 여분의(예컨대, 삭제) 연산들이 실행되어 대응하는 비즈니스 로직이 실행될 수 있게 된다.

[0255] 동시성 제어 식 생성기(1015): 수정된 개체들이 진부(stale)한지를 검출하기 위해, 매핑 메타데이터에서 특정된 타임스탬프 열 또는 열들의 세트를 조사하는 식을 생성할 수 있다.

[0256] 개체-EDM 변환(1016): 삽입, 삭제, 및 업데이트 개체 세트들에 의하여 특정된 변경 목록들은 이제, 도 9를 참조하여 설명된 매핑 컴파일 후에 저장되는, 메타데이터 저장소(1017)에 저장된 매핑 델타 식들을 이용하여 변환된다. 이 단계 후에, 변경들은 EDM 엔티티들에 의해서만 표현된 식 트리들(1018)로서 사용 가능하다.

[0257] 단계 1018로부터의 식 트리는 다음으로 EDM-프로바이더-계층(1030) 내의 EDM 프로바이더에 전달된다. EDM 프로바이더에서, 식 트리는 처리되고 변경들은 저장소에 제시된다. 이 식 트리(1018)는 또한 다른 식으로 생성될 수도 있다는 것에 유의한다 – 애플리케이션이 EDM 프로바이더에 대하여 직접 프로그램하는 경우, 그것은 그것에 대하여 DML 문을 실행할 수 있다. 그러한 DML 문은 먼저 EDM 프로바이더에 의해 EDM 식 트리(1018)로 변환된다. DML 문으로부터 또는 애플리케이션 계층(1000)으로부터 얻어진 식 트리는 다음과 같이 처리된다:

[0258] 선택적 업데이트 분할기(1031): 이 단계에서는, 업데이트들 중 일부가 삽입 및 삭제들로 분할된다. 일반적으로, 업데이트들을 그대로 하위 계층들에 전파한다. 그러나, 특정 경우에는, 그 경우에 대하여 델타 식 규칙들이 개발되지 않았기 때문에 또는 정확한 변환이 실제로 기본 테이블에 대한 삽입 및/또는 삭제들로 귀결되기 때문에, 그러한 업데이트를 수행하는 것이 불가능할 수 있다.

[0259] EDM-저장소 변환(1032): EDM-레벨 식 트리(1018)는 적절한 매핑으로부터의 델타 식들을 이용하여 저장소 공간으로 변환된다.

[0260] 쿼리 매핑 뷰 언폴딩(1034): 식 트리(1018)는 일부 EDM-레벨 개념들을 포함할 수 있다. 그것들을 제거하기 위하여, 쿼리 매핑 뷰들(1035)를 이용하여 식 트리를 언폴딩하여 저장소-레벨 개념들만에 의하여 트리(1038)를 획득한다. 트리(1038)는 옵션으로 E-S 보상 컴포넌트(1037)에 의해 처리된다.

[0261] 이제 저장소 공간 용어들로 되어 있는 식 트리(1038)는 이제 저장소 프로바이더 층(1040) 내의 저장소 프로바이더에 제공되고, 그것은 다음의 단계들을 수행한다:

[0262] 단순화(1041): 식 트리는 논리 식 변환 규칙을 이용하여 단순화된다.

[0263] SQL 생성(1042): 식 트리가 주어지면, 저장소 프로바디어는 식 트리(1038)로부터 실제 SQL(1043)을 생성한다.

[0264] SQL 실행(1044): 데이터베이스 상에서 실제 변경들이 수행된다.

[0265] 서버 생성 값들: 서버에 의해 생성된 값들이 EDP 계층(1030)에 반환된다. 프로바이더(1044)는 서버 생성 값들을 그것들을 매핑을 이용하여 EDM 개념들로 변환하는 계층(1030) 내의 컴포넌트(1039)에 전달한다. 애플리케이션 계층(1000)은 이들 변경들(1003)을 픽업하여 그것들을 그 계층에서 이용되는 다양한 애플리케이션들 및 개체들에서 설치될 개체 레벨 개념들에 전파한다.

[0266] 많은 경우에, 저장소 테이블들은 예를 들어 데이터베이스 관리자(Database Administrator; DBA) 정책들 때문에 직접적으로 업데이트 가능하지 않을 수 있다. 테이블들에 대한 업데이트들은 저장된 프로시저들을 통해서만 가능할 수 있어 특정 유효성 검사 체크가 수행될 수 있다. 그러한 상황에서, 매핑 컴포넌트는 "원시(raw)" 삽입, 삭제, 및 업데이트 SQL 문들을 실행하기보다는 개체 변경들을 이들 저장된 프로시저들에의 호출들로 변환해야만 한다. 다른 경우에, "저장된" 프로시저들은 EDP(1010)에서 및 애플리케이션 계층(1000)에서 특정될 수 있다 – 그러한 경우에, 매핑 컴포넌트는 수정된 개체들을 EDM 공간으로 변환한 다음, 적절한 프로시저를 호출해야만 한다.

[0267] 이들 시나리오를 가능하게 하기 위해, MSL은 저장된 프로시저들이 매핑의 일부로서 특정되는 것을 허용하고; 추가로, MSL은 또한 다양한 데이터베이스 열들이 저장된 프로시저들의 매개 변수들에 어떻게 매핑되는지를 특정하는 메커니즘을 지원한다.

[0268] EDP 계층(1010)은 낙관적인 동시성 제어를 지원한다. CDP가 변경들의 세트를 저장소에 송신할 때, 변경된 행들은 이미 다른 트랜잭션에 의해 수정되었을 수도 있다. CDP는 사용자들이 그러한 충돌들을 삭제하고, 그 후 그

러한 충돌들을 해결할 수 있는 방법을 지원해야만 한다.

[0269] MSL은 충돌 검출을 위한 단순한 메커니즘들 – 타임스탬프, 버전 번호, 변경된 열 열들 – 을 지원한다. 충돌들이 검출되는 경우, 예외(exception)가 제기되고, 충돌하는 개체들(또는 EDM 엔티티들)은 애플리케이션에 의해 충돌 해결을 위해 사용가능하다.

[0270] 예시적인 매핑 요건

[0271] 매핑 인프라는 유리하게는 다양한 연산들을 애플리케이션 공간으로부터 관계형 공간으로 변환하는, 예를 들면, 개발자에 의해 작성된 개체 쿼리들이 관계형(저장소) 공간으로 변환되는 능력을 제공할 수 있다. 이들 변환들은 데이터의 과도한 복사 없이 효율적이어야 한다. 매퍼는 다음의 예시적인 동작들에 대한 변환들을 제공할 수 있다:

[0272] 1. 쿼리 : 개체 쿼리들은 백엔드 관계형 도메인으로 변환될 필요가 있고 데이베이스로부터 얻어진 튜플(tuple)들은 애플리케이션 개체들로 변환될 필요가 있다. 이들 쿼리들은 세트 기반 쿼리들(예컨대, CSQL 또는 C# 시퀀스들) 또는 탐색 기반(예컨대, 참조들의 단순 추종)일 수 있다는 것에 유의한다.

[0273] 2. 업데이트: 애플리케이션에 의해 그의 개체들에 행해진 변경들은 도로 데이터베이스로 전파될 필요가 있다. 다시, 개체들에 행해진 변경들은 세트 기반 또는 개개의 개체들에 대한 것일 수 있다. 고려할 또 다른 디멘전은 수정되는 개체들이 메모리에 완전히 로딩되는지 부분적으로 로딩되는지이다(예를 들어, 개체를 놓아주는(hang off) 컬렉션이 메모리에 존재하지 않을 수도 있다). 부분적으로 로딩된 개체들에 대한 업데이트에 대해서는, 이들 개체들이 메모리에 완전히 로딩될 필요가 없는 디자인들이 바람직할 수 있다.

[0274] 3. 무효화 또는 통지 : 중간 계층 또는 클라이언트 계층에서 실행하는 애플리케이션들은 백엔드에서 어떤 개체들이 변경될 때 통지받기를 원할 수 있다. 따라서, OR-매핑 컴포넌트는 개체 레벨에서의 등록들을 관계형 공간으로 변환해야 한다; 유사하게, 수정된 튜플들에 관하여 클라이언트에 의해 메시지들이 수신되는 경우, OR-매퍼는 이들 통지들을 개체 변경들로 변환해야만 한다. WinFS는 그의 와처(Watcher) 메커니즘을 통하여 그러한 "통지"를 지원하지만, 그 경우, 매핑은 미리 규정되는 반면, 엔티티 프레임워크는 비규정된 매핑에 대하여 와처들을 지원해야만 한다는 것에 유의한다.

[0275] 4. 중간 계층 또는 클라이언트 계층에서 실행하는 엔티티 프레임워크 프로세스로부터의 진부한 개체들을 무효화하기 위해 통지와 유사한 메커니즘이 또한 필요하다 – 만일 엔티티 프레임워크가 충돌하는 판독/기록을 처리하기 위한 낙관적인 동시성 제어에 대한 지원을 제공한다면, 애플리케이션들은 엔티티 프레임워크에 캐싱된 데이터가 합리적으로 신선하다는(따라서 개체들의 판독/기록으로 인해 트랜잭션들이 중단되지 않는다는) 것을 보증할 수 있고; 그렇지 않다면, 그것들은 오래된 데이터에 대한 판정을 행하거나 및/또는 그들의 트랜잭션이 나중에 중단되게 할 수 있다. 따라서, 통지와 같이, OR-매퍼는 데이터베이스 서버들로부터의 "무효화" 메시지를 개체 무효화로 변환해야만 할 수 있다.

[0276] 5. 백업/복원/동기: 엔티티들의 백업 및 미러링은 일부 실시예들에서 통합될 수 있는 2가지 특징들이다. 이들 특징들에 대한 요건들은 단순히 OR-매퍼의 관점으로부터 엔티티들에 대한 특수화된 쿼리로 변환될 수 있고; 그렇지 않다면, 그러한 연산들에 대한 특별한 지원이 제공될 수 있다. 마찬가지로, 동기(sync)는 개체 변경, 충돌 등을 저장소로 변환하고 반대로 저장소 변경, 충돌 등을 개체로 변환하기 위한 OR-매핑 엔진으로부터의 지원을 필요로 할 수 있다.

[0277] 6. 동시성 제어에의 참여: OR 매퍼는 유리하게는 낙관적인 동시성 제어가 애플리케이션에 의해 이용될 수 있는 상이한 방법들, 예를 들어, 타임스탬프 값, 어떤 특정 필드들의 세트를 이용하는 것 등을 지원할 수 있다. OR 매퍼는 타임스탬프 속성 등의 동시성 제어 정보를 개체 공간으로/으로부터 및 관계형 공간으로부터/으로 변환해야만 한다. OR-매퍼는 심지어 (예를 들어, Hibernate와 같은) 비관적인 동시성 제어에 대한 지원을 제공할 수도 있다.

[0278] 7. 런타임 오류 보고: 본 명세서에서 설명된 예시적인 실시예에서, 런타임 오류들은 통상적으로 저장소 레벨에서 일어날 것이다. 이들 오류들은 애플리케이션 레벨로 변환될 수 있다. OR 매퍼는 이들 오류 변환을 용이하게 하기 위해 이용될 수 있다.

[0279] 매핑 시나리오

[0280] 엔티티 프레임워크가 지원할 수 있는 예시적인 개발자 시나리오들을 논의하기 전에, OR-매퍼의 다양한 논리 부분들을 설명한다. 일 실시예에서는, 도 11에 도시된 바와 같이 OR-매핑에 5개의 부분들이 있다:

[0281] 1. 개체/클래스/XML(별칭은 애플리케이션 공간)(1101): 개발자는 선택한 언어로 클래스 및 개체를 특정한다 - 궁극적으로, 이들 클래스는 CLR 어셈블들로 컴파일되고 리플렉션(reflection) API들을 통하여 액세스 가능하다. 이들 클래스는 지속(persistent) 및 비지속(non-persistent) 멤버들도 포함한다; 또한, 언어 특정 상세가 이 부분에 포함될 수 있다.

[0282] 2. 엔티티 데이터 모델 스키마(별칭은 개념적 공간)(1102): EDM 공간은 개발자에 의해 데이터를 모델링하기 위해 이용된다. 위에서 논한 바와 같이, 데이터 모델의 명세는 EDM 타입, 연결(associations)에 의한 엔티티들 간의 관계, 상속 등에 의하여 행해진다.

[0283] 3. 데이터베이스 스키마(별칭은 저장소 공간)(1103): 이 공간에서, 개발자는 테이블들이 어떻게 배치되는지를 특정하고, 외래 키 및 기본 키 제약 등의 제약들도 여기에서 특정된다. 이 공간에서의 명세는 벤더 특정 특징들, 예컨대, 네스팅된 테이블(nested tables), UDT 등을 이용할 수 있다.

[0284] 4. 개체-EDM 매핑(1104): 이 매핑은 다양한 개체들 및 EDM 엔트리들이 서로 어떻게 관련되는지를 특정한다. 예를 들어, 어레이는 일 대 다 연결(one-to-many association)로 매핑될 수 있다. 이 매핑은 명백(trivial)/항등(identity)인 것이 필수적인 것은 아니라는 점에 유의한다. 예를 들어, 복수의 클래스가 주어진 EDM 타입에 매핑될 수도 있고 또는 그 반대일 수도 있다. 이들 매핑에서 중복/역정규화(redundancy/denormalization)가 없을 수 있다는 것에 유의한다(물론, 역정규화의 경우에는, 개체들/엔티티들을 일관성 있게 유지하는 문제들에 이를 수 있다).

[0285] 5. EDM-저장소 매핑(1105): 이 매핑은 EDM 엔티티들 및 타입들이 데이터베이스의 상이한 테이블들에 어떻게 관련되는지를 특정한다. 예를 들어, 상이한 상속 매핑 전략들이 여기에서 이용될 수 있다.

[0286] 개발자는 공간들(1101, 1102, 또는 1103) 중 하나 이상의 공간 및 그들 간의 하나 또는 그 이상의 매핑들 간의 대응하는 매핑들을 특정할 수 있다. 만일 임의의 데이터 공간이 없어지면, 개발자는 그 공간을 생성하는 방법에 대한 힌트를 제공하거나 EDP가 대응하는 규정된 매핑들을 이용하여 그 공간들을 자동으로 생성할 것을 기대할 수 있다. 예를 들면, 개발자가 현존하는 클래스들, 테이블들 및 그들 간의 매핑을 특정하면, EDP는 내부 EDM 스키마 및 대응하는 개체-EDM 및 EDM-저장소 매핑들을 생성한다. 물론, 대부분의 일반적인 경우에, 개발자는 완전한 제어권을 갖고 2개의 매핑과 함께 이들 공간 내의 데이터 모델들을 특정할 수 있다. 아래의 테이블은 EDP에서 지원되는 상이한 시나리오들을 보여준다. 이것은 개발자가 개체들, EDM 엔티티들, 테이블들을 특정하거나 특정하지 않을 수 있는 경우와 총망라한 목록이다.

[0287]

시나리오	개체 특정?	CDM 특정?	테이블 특정?	매핑 특정
(A)	Y			
(B)		Y		
(C)			Y	
(D)	Y	Y		OE
(E)	Y		Y	OS
(F)		Y	Y	ES
(G)	Y	Y	Y	OE, ES

[0288] EDP가 지원하기를 원하는 상기 시나리오들에 따라서, 특정되지 않은 데이터 공간들 및 매핑들을 생성하는 도구들을 제공해야 할 것이다(미리 정해진 방식으로 또는 힌트들이 제공된다면 그 힌트들에 기초하여). 내부 OR 매핑 엔진은 매핑의 5가지 부분들 전부(개체, EDM 명세, 테이블, OE 매핑, ES 매핑)가 이용 가능하다고 가정한다. 따라서, 매핑 디자인은 대부분의 일반적인 케이스, 즉, 상기 테이블에서의 (G)를 지원해야 한다.

[0289] 매핑 명세 언어(MAPPING SPECIFICATION LANGUAGE)

[0290] 개발자의 관점으로부터 OR 매퍼의 "눈에 보이는(visible)" 부분들 중 하나는 매핑 명세 언어 즉 MSL이다 - 개

발자는 얼마나 다양한 매핑의 부분들이 이 언어를 이용하여 서로 결합하는지를 특정한다. 런타임 컨트롤들(예컨대, 지연 페칭, 낙관적인 동시성 제어 문제들)도 MSL을 이용하여 특정된다.

[0291] 우리는 매핑을 3가지 상이한 개념들로 나눈다 – 각 개념은 매핑 프로세스에 대한 상이한 관심사들을 처리한다. 이들 명세가 단일 파일에 저장되는지, 복수의 파일에 저장되는지, 또는 외부 리포지토리를 통하여 특정되는지(예컨대, 데이터 명세에 대하여)를 기술하지는 않는다는 점에 유의한다.

[0292] 1. 데이터 명세: 이 영역에서, 개발자는 클래스 기술(descriptions), 테이블 기술, 및 EDM 기술을 특정할 수 있다. 이들 기술은 생성을 위한 명세로서 제공될 수도 있고 또는 이미 존재하는 테이블들/개체들에 대한 명세들일 수도 있다.

[0293] 개체 및 테이블 명세들은 우리의 포맷으로 기술될 수도 있고 또는 그것들은 어떤 가져오기 도구(import tool)를 이용하여 외부 메타데이터 리포지토리로부터 가져와질 수도 있다.

[0294] 서버 생성 값들, 제약들, 기본 키 등의 명세는 이 섹션에서 행해진다는 것에 유의한다(예를 들어, EDM 명세에서, 제약들은 타입 명세의 일부로서 특정된다).

[0295] 2. 매핑 명세: 개발자는 다양한 개체들, EDM 타입들, 및 테이블들에 대한 매핑들을 특정한다. 개발자들이 개체-EDM, EDM-저장소, 및 개체-저장소 매핑들을 특정하는 것이 허용된다. 이 섹션은 데이터 명세와의 최소의 중복을 가지려고 노력한다.

[0296] 모든 3가지 매핑 케이스(OS, ES 및 OE)에서, 최상위 레벨에서 "직접적으로" 또는 다른 클래스 내에서 "간접적으로" 각 클래스에 대한 매핑들을 특정한다. 각 매핑에서, 필드/속성은 다른 필드, 필드들의 스칼라 함수, 컴포넌트, 또는 세트로 매핑된다. 업데이트를 허용하기 위해, 이들 매핑들은 양방향일 필요가 있다. 즉, 개체로부터 저장소 공간으로 가고 되돌아가는 것은 어떤 정보도 잃어서는 안 되고; 또한 개체들이 판독만 되도록 비양방향(non-bidirectional) 매핑도 허용될 수 있다.

[0297] 개체-EDM 매핑: 일 실시예에서는, 각 개체마다에 대한 매핑을 EDM 타입들에 의하여 특정한다.

[0298] EDM-저장소 매핑: 일 실시예에서는, 각 엔티티마다에 대한 매핑을 테이블들에 의하여 특정한다.

[0299] 개체-저장소 매핑: 일 실시예에서는, 각 개체마다에 대한 매핑을 테이블들에 의하여 특정한다.

[0300] 3. 런타임 명세: 일 실시예에서는, 개발자들이 실행을 제어하는 다양한 노브(knob)들, 예를 들어, 낙관적인 동시성 제어 매개 변수들, 및 페칭 전략을 특정하는 것이 허용된다.

[0301] OPerson 개체가 주소들의 세트를 포함하는 경우에 대한 매핑 파일들의 일례가 여기에 있다. 이 개체는 EDM 엔티티 타입에 매핑되고 그 세트는 인라인 세트 타입에 매핑된다. 데이터는 2개의 테이블 – 사람들에 대한 하나의 세트와 주소들에 대한 다른 하나의 세트에 저장된다. 전술한 바와 같이, 개발자가 모든 개체들, EDM 타입들 및 테이블들을 특정하는 것이 필수적인 것은 아니다 – 우리는 단지 상기 테이블로부터의 케이스 (G)를 보여주고 있다. 명세들은 임의의 특정 구문을 기술해서는 안 된다; 그것들은 본 명세서에서 개시된 개념들을 중심으로 해서 시스템의 디자인을 예시하고 가능하게 하도록 의도되어 있다.

[0302] *개체 명세*

[0303]
```
ObjectSpec OPerson {
    string name;
    Set<Address> addrs;
}
```

```
ObjectSpec OAddress {
    string state;
}
```

[0304] *EDM 명세*

[0305] 하나의 엔티티 타입 CPerson 및 인라인 타입 CAddress를, 각 CPerson이 CAddress 항목들의 컬렉션을 갖도록 특정한다.

[0306]
```
EDMSpec Entity CPerson {
    string name;
    int pid;
    Set<CAddress> addrs;
    Key (pid)
}
```

```
EDMSpec InlineType CAddress {
    string state;
    int aid;
}
```

[0307] *저장소 명세*

[0308] 2개의 테이블 타입 SPerson 및 SAddress를 그들의 키(tpid 및 taid)와 함께 특정한다.

```
TableSpec SPerson {
    int pid,
    nvarchar(10) name,
    Key {pid}
}
```

```
TableSpec SAddress {
    int aid,
    string state,
    Key: {aid}
}
```

[0309]

[0310] *개체-CDM 매핑*

[0311] OPerson에 대한 다음의 매핑은 개체 타입 OPerson이 엔티티 OPerson에 매핑되는 것을 말한다. 그 다음의 목록은 OPerson의 각 필드가 어떻게 매핑되는지를 특정한다 – 이름은 이름에 매핑되고 주소 컬렉션은 주소 컬렉션에 매핑된다.

```
Object-CDM OPerson {
    EntitySpec = CPerson
    name↔name
    addrs↔addrs
    }
}
```

```
Object-CDM OAddress {
    InlineTypeSpec = CAddress
    state ↔ state
}
```

[0312]

[0313] *EDM-저장소 매핑*

[0314] EDM 엔티티 타입 CPerson은 그의 키 및 이름 cname 특성들(attributes)과 함께 테이블 타입 SPerson에 매핑된다. 인라인 타입(InlineType) CAddress는 단순한 방식으로 SAddress에 매핑된다. 테이블 SAddress는 외래 키를 SPerson에 저장할 수 있다는 것에 유의한다; 이 제약은 매핑에서가 아니라, 테이블의 데이터 모델 명세에서 특정되었을 수 있다.

```
EDM-Store CPerson {
    TableSpec = SPerson
    name ↔ name
    pid ↔ pid
}
```

```
EDM-Store CAddress {
    TableSpec = SAddress
    aid ↔ aid
    state ↔ state
}
```

```
EDM-Store CPerson_Address {
    TableSpec = SAddress
    aid ↔ aid
    pid ↔ pid
}
```

[0315]

[0316] *런타임 명세*

[0317] 개발자는 OPerson에 대한 낙관적인 동시성 제어가 pid 및 name 필드들 상에서 행해지는 것을 특정하기를 원할 수 있다. OAddress에 대하여, 그는 state 필드에 대한 동시성 제어를 특정할 수 있다.

```
RuntimeSpec OPerson {
    Concurrency fields: {pid, name}
}
```

```
RuntimeSpec OAddress
    Concurrency fields: {state}
}
```

[0318]

[0319] 매핑 디자인 개관

[0320] Hibernate 및 ObjectSpaces 등의 대부분의 OR-매핑 기술들은 중요한 단점이 있다 – 그것들은 비교적 특별 방식(ad-hoc manner)으로 업데이트를 처리한다. 개체 변경들이 서버에 도로 푸시될 필요가 있는 경우, 이들 시스템에 의해 이용되는 메커니즘들은 케이스별로 업데이트를 처리하고 그에 따라 시스템의 확장성이 제한된다. 더 많은 매핑 케이스들이 지원될수록, 업데이트 파이프라인은 더욱 복잡해지고 업데이트들에 대한 매핑들을 구성하는 것이 어렵다. 시스템이 진화할수록 시스템의 이 부분은 그것이 정확한 것을 보장하면서 변경하기가 귀찮게 된다.

[0321] 그러한 문제를 피하기 위하여, 2가지 타입의 "매핑 뷰들" – 쿼리들을 변환하는 데 도움이 되는 하나와 업데이트들을 변환하는 데 도움이 되는 다른 하나 – 을 이용하여 매핑 프로세스를 수행하는 새로운 접근법을 이용한다. 도 12에 도시된 바와 같이, MSL 명세(1201)가 EDP에 의해 처리될 때, 그것은 코어 매핑 엔진(core mapping engine)의 실행을 위해 내부적으로 2개의 뷰(1202 및 1203)를 생성한다. 나중에 알겠지만, 이들 뷰에 의하여 매핑을 모델링함으로써, 관계형 데이터베이스들에서 구체화된-뷰(materialized-view) 기술에 대한 기존

의 지식을 이용할 수 있다 – 특히, 정확하고, 우아하고, 확장성 있는 방식으로 업데이트들을 모델링하기 위한 점진적 뷰-관리 기법들을 이용한다. 이제 이들 2가지 타입의 매핑 뷰들에 대하여 설명한다.

[0322] 테이블 데이터를 개체들에 매핑하기 위해 쿼리 매핑 뷰(Query Mapping Views) 즉 QMView들의 개념을 이용하고 개체 변경들을 테이블 업데이트들에 매핑하기 위해 업데이트 매핑 뷰들(Update Mapping Views) 즉 UMView들의 개념을 이용한다. 이들 뷰는 그것들이 구성되는 (주요) 이유 때문에 명명된다. 쿼리 뷰는 개체 쿼리들을 관계형 쿼리들로 변환하고 입력되는 관계형 튜플들을 개체들로 변환한다. 따라서, EDM-저장소 매핑에 대하여, 각 QView는 EDM 타입이 다양한 테이블들로부터 어떻게 구성되는지를 보여준다. 예를 들면, Person 엔티티가 2개의 테이블 T_P 및 T_A의 조인(join)으로부터 구성된다면, Person을 이들 2개의 테이블 간의 조인에 의하여 특정한다. 쿼리가 Person 컬렉션에 대하여 요청되는 경우, Person에 대한 QMView는 Person을 T_P 및 T_A에 의한 식으로 대체하고; 그 후 이 식은 적절한 SQL을 생성한다. 그 후 쿼리는 데이터베이스에서 실행되고; 서버로부터 회신이 수신되면, QMView는 반환된 튜플들로부터 개체들을 구체화한다.

[0323] 개체 업데이트들을 처리하기 위하여, 변경들을 QMView들을 통하여 푸시하고 관계형 데이터베이스에 대하여 개발된 "뷰 업데이트(view update)" 기술을 이용하는 것을 생각할 수 있다. 그러나, 업데이트 가능한 뷰들은 그들에 대한 다수의 제한을 갖고 있다. 예를 들어, SQL 서버는 복수의 기본 테이블들이 뷰 업데이트를 통하여 수정되는 것을 허용하지 않는다. 따라서, EDP에서 허용되는 매핑의 타입들을 제한하는 대신, 본 발명의 실시예들은 보다 적은 수의 제한을 갖는 구체화된-뷰 기술의 다른 양태 – 뷰 관리(view maintenance)를 이용한다.

[0324] 시스템 내의 각 테이블을 EDM 타입들에 의하여 표현하기 위해 업데이트 매핑 뷰들 즉 UMView들을 특정한다. 즉, 어떤 의미에서, UMView들은 QMView들의 역이다. EDM-저장소 경계 상의 테이블 타입에 대한 UMView는 해당 테이블 타입의 열들을 구성하기 위해 상이한 EDM 타입들이 이용되는 방법을 제시한다. 따라서, Person 개체 타입이 테이블 타입들 T_P 및 T_A에 매핑한다고 특정하였다면, Person 타입에 대한 QMView를 T_P 및 T_A에 의하여 생성할 뿐만 아니라, 또한 (T_A의 경우와 유사하게) Person 개체 타입이 주어진 경우 T_P의 행이 어떻게 구성될 수 있는지를 특정하는 UMView를 생성한다. 만일 트랜잭션이 어떤 Person 개체들을 생성, 삭제, 또는 업데이트하면, 개체들로부터의 그러한 변경들을 T_P 및 T_A에 대한 SQL 삽입, 업데이트 및 삭제 문들로 변환하기 위해 업데이트 뷰들을 이용할 수 있다 – UMView들은 (CDM 타입들을 통하여) 개체들로부터 관계형 튜플들이 어떻게 얻어지는지를 알려주므로 이들 업데이트를 수행하는 데 도움이 된다. 도 13 및 14는 QMView들 및 UMView들이 쿼리 및 업데이트 변환에서 어떻게 이용되는지를 고레벨에서 보여준다.

[0325] 개체들에 대한 뷰로서 테이블들을 모델링하기 위한 이 접근법이 주어지면, 개체들에 대한 업데이트들을 도로 테이블들에 전파하는 프로세스는 개체들이 "기본 관계들(base relations)"이고 테이블들이 "뷰들(views)"인 경우의 뷰-관리 문제와 유사하다. 뷰-관리 문제를 다루는 방대한 양의 데이터베이스 문헌이 있고 우리의 목적을 위해 그것을 이용할 수 있다. 예를 들면, 기본 관계들에 대한 점진적 변경들이 뷰들에 대한 점진적 변경들로 어떻게 변환도리 수 있는지를 보여주는 상당한 수의 연구가 있다. 우리는 뷰들에 대한 점진적 업데이트들을 수행하기 위해 필요한 식들을 결정하기 위해 대수 접근법(algebraic approach)을 이용할 수 있다 – 우리는 이들 식을 델타 식이라고 칭한다. 점진적 뷰 관리를 위하여, 절차상의 것에 대립하는 것으로서, 대수 접근법을 이용하는 것은 최적화 및 업데이트 단순화에 더 많이 순종하므로 적절하다.

[0326] 일반적으로, EDP의 코어 엔진에서 매핑 뷰를 이용하는 이점들을 다음을 포함한다:

[0327] 1. 뷰들은 개체들과 관계들 간의 맵들을 표현하기 위한 상당한 양의 능력 및 유연성을 제공한다. 우리는 OR-매핑 엔진의 코어 부분에서 제한된 뷰-식(view-expression) 언어로 착수할 수 있다. 시간과 자원이 허용할 때, 뷰들의 능력은 시스템을 우아하게 진화시키는 데 이용될 수 있다.

[0328] 2. 뷰들은 쿼리들, 업데이트들 및 뷰들 자체를 가지고 상당히 우아하게 조합하는 것으로 알려져 있다. 특히 업데이트들에 관련한 조합성(composability)은 보다 일찍이 시도된 OR-매핑 접근법들 중 일부에서 해결하기 어려운 문제였다. 뷰 기반 기술을 채택함으로써, 그러한 관심사를 피할 수 있다.

[0329] 뷰의 개념을 이용함으로써 데이터베이스 문헌 내의 상당한 수의 연구를 이용할 수 있다.

[0330] 업데이트에 대한 구조적 계층화(ARCHITECTURAL LAYERING FOR UPDATES)

[0331] 본 발명의 양태들의 구현에서 고려할 중요한 문제는, 쿼리 및 업데이트 매핑 뷰들이 표현되는 매핑 뷰 언어(Mapping View Language)(즉 MVL)의 능력이 어떤 것이냐이다. 그것은 대체로 EDM과 저장소 간의 매핑들과 함께

개체들과 EDM 간의 모든 비규정적인(non-prescriptive) 매핑들을 캡처할 만큼 강력하다. 그러나, 모든 비관계형(non-relational) CLR 및 EDM 개념들을 내재적으로 지원하는 MVL의 경우, 모든 그러한 구성들에 대한 델타 식들 또는 점진적 뷰 업데이트 규칙들을 디자인할 필요가 있다. 특히, 예시적인 일 실시예는 다음의 비관계형 대수 연산자들/개념들에 대한 업데이트 규칙들을 요구할 수 있다:

[0332] 복잡 타입(complex types) – 개체들, 튜플 생성자들(tuple constructors), 플래트닝(flattening), 복소 상수(complex constants) 등의 부분들에의 액세스.

[0333] 컬렉션(collections) – 네스팅(nesting) 및 언네스팅(unnesting), 세트 구성/플래트닝(set construction/flattening), 크로스 어플라이(cross apply) 등.

[0334] 어레이/목록(arrays/lists) – 요소들의 순서화(ordering)는 관계형 구성이 아니고; 명백히, 순서화된 목록들에 대한 대수는 상당히 복잡하다.

[0335] 모델링될 필요가 있는 CLR/C#에서의 다른 EDM 구성들 및 개체 구성들

[0336] 이들 구성들에 대한 점진적 업데이트를 위한 델타 식들을 개발하는 것이 가능하다. MVL에서 내재적으로 구성들의 큰 세트를 지원하는 것에 있어서의 주요 문제점은 그것이 코어 엔진을 상당히 복잡하게 할 수 있다는 점이다. 일 실시예에서, 보다 바람직한 접근법은 "코어 매핑 엔진(core mapping engine)"이 단순한 MVL을 처리하도록 시스템을 계층화하고 그 후 이 코어의 위에 비관계형 구성들을 계층화하는 것일 수 있다. 이제 그러한 디자인에 대하여 설명한다.

[0337] OR-매핑을 위한 우리의 접근법은 "계층화(layering)"에 의해 상기 문제들을 처리한다 – 컴파일 타임에, 먼저 개체, EDM, 및 데이터베이스 공간들(WinFS는 네스팅, UDP 등을 지원한다) 내의 각 비관계형 구성을 대응하는 관계형 구성으로 미리 규정된 방식으로 변환하고 그 후 관계형 구성들 간의 요청된 비규정된 변환들을 수행한다. 우리는 이러한 접근법을 계층화된 뷰 매핑 접근법(layered view mapping approach)라고 칭한다. 예를 들면, 클래스 CPerson이 주소들의 컬렉션을 포함한다면, 먼저 이 컬렉션을 일 대 다 연결로서 관계형 구성으로 변환하고 그 후 관계형 공간 내의 테이블들로의 요청된 비규정된 변환을 수행한다.

[0338] **MVL 분해(MVL Breakdown)**

[0339] MVL은 2개의 계층 – 관계형 용어의 실제 비규정적 매핑을 다루는 것과 비관계형 구성들의 관계형 용어로의 규정적 변환 – 으로 분해된다. 전자의 언어는 (관계형(Relational)-MVL에 대한) R-MVL로 불리고 대응하는 매핑들은 R-MVL 매핑이라고 불린다; 마찬가지로, 후자의 (보다 강력한) 언어는 (비관계형(Non-relational)-MVL에 대한) N-MVL로 불리고 그 매핑들은 N-MVL 매핑이라고 불린다.

[0340] 일 실시예에서, 매핑은 모든 비관계형 구성들이 쿼리 및 업데이트 파이프라인들의 끝으로 푸시되도록 디자인을 구성함으로써 제공된다. 예를 들면, 개체 구체화는 개체, 어레이, 포인트 등을 구성하는 것을 포함할 수 있다 – 그러한 "연산자들(operators)"은 쿼리 파이프라인의 최상부로 푸시된다. 유사하게, 개체들에 대해 업데이트들이 일어날 경우, 파이프라인의 맨 처음에 있는 비관계형 개체들(예를 들어, 네스팅된(nested) 컬렉션들, 어레이들)에 대한 변경들을 변환하고 그 후 이들 변경들을 업데이트 파이프라인을 통하여 전파한다. WinFS와 같은 시스템에서는, UDT들로의 업데이트 파이프라인의 맨 끝에서 변환할 필요가 있다.

[0341] 비규정된 매핑들을 R-MVL로 제한함으로써, 우리는 이제 점진적 뷰 관리 규칙들을 필요로 하는 관계형 구성들의 작은 세트를 갖는다 – 그러한 규칙들은 이미 관계형 데이터베이스에 대하여 개발되었다. 우리는 R-MVL에서 허용되는 단순화된 구성들/스키마들을 관계형으로 표현된 스키마(Relatlonally-Expressed Schema) 즉 RES라고 한다. 따라서, 일부 비관계형 구성이 개체 도메인에서 지원될 필요가 있는 경우, 우리는 대응하는 RES 구성 및 개체와 RES 구성 간에 규정된 변환을 찾아낸다(come up with). 예를 들어, 우리는 개체 컬렉션을 RES 공간에서의 일 대 다 연결로 변환할 수 있다. 더욱이, 비관계형 구성들 N에 대한 업데이트들을 전파하기 위해, 우리는 N으로부터의 삽입, 삭제, 및 업데이트를 N의 대응하는 RES 구성으로 변환하는 델타 식을 찾아낸다. 이들 델타 식은 미리 규정되고 디자인 타임에 우리에 의해 생성된다는 것에 유의한다. 예를 들어, 우리는 컬렉션에 대한 변경들을 일 대 다 연결 상에 어떻게 푸시하는지를 알고 있다. 실제 비규정된 매핑들에 대한 델타 식들은 관계형 데이터베이스에 대한 점진적 뷰 관리를 이용하여 자동으로 생성된다. 이 계층화된 방법은 과도한 비관계형 구성들에 대한 일반화된 점진적 뷰 관리를 찾아내는 요건을 제거할 뿐만 아니라 내부 업데이트 파이프라인을 단순화한다.

[0342] 우리의 계층화된 매핑 접근법은 통지 파이프라인에 대해서도 유사한 이익을 갖는다는 것에 유의한다 – 서버로부터 튜플들에 대한 변경들이 수신되면 우리는 그것들을 개체들에 대한 점진적 변경들로 변환할 필요가 있다. 이것은 이들 변경을 전파하기 위해 쿼리 매핑 뷰들을 이용할 필요가 있다는, 즉 QMView들에 대한 델타 식들을 생성한다는 것을 제외하고 업데이트 파이프라인과 동일한 요건이다.

[0343] 업데이트 및 통지 파이프라인을 단순화하는 것은 별문제로 하고, MVL을 계층화하는 것은 중요한 이점을 갖는다 – 그것은 "상위 언어(upper languages)"(개체, EDM, 데이터베이스)가 코어 매핑 엔진에 크게 영향을 주지 않고 진화하는 것을 허용한다. 예를 들면, EDM에 새로운 개념이 추가되면, 우리가 해야할 것은 그것을 해당 구성에 대한 대응하는 RES로 변환하는 규정된 방법을 찾아내는 것이다. 유사하게, SQL 서버에 비관계형 개념이 있다면(예컨대, UDT, 네스팅), 우리는 이들 구성들을 MVL로 미리 정해진 방식으로 변환할 수 있고 MVL 및 코어 엔진에 최소한의 영향을 줄 수 있다. RES-저장소와 저장소 테이블들 간의 변환은 반드시 항등 변환(identity translation)은 아니라는 점에 유의한다. 예를 들면, UDT, 네스팅 등을 지원하는 (WinFS 백엔드 등의) 백엔드 시스템에서, 변환은 규정된 개체 관계와 유사하다.

[0344] 도 15는 매핑 뷰들의 컴파일 타임 및 런타임 처리를 예시한다. 1501, 1502, 및 1503에 의해 예시된 바와 같이 MSL에서의 데이터 모델 및 매핑 명세가 주어지면, 우리는 먼저 비관계형 구성들(1511, 1512, 1513)에 대한 대응하는 RES들(1521, 1522, 및 1523), 및 이들 구성들과 RES들 간의 규정된 변환들, 즉, N-MVL 매핑들을 생성한다. 그 후 우리는 개발자에 의해 요청된 비규정된 매핑들에 대한, 쿼리 및 업데이트 매핑 뷰들, R-MVL에서의 개체-EDM 및 R-MVL에서의 EDM-저장소를 생성한다 – 이들 매핑 뷰들은 R-MVL을 이용하여 RES들에 작용한다는 것에 유의한다. 이 시점에서, 우리는 쿼리 및 업데이트 매핑 뷰들에 대한 델타 식들(뷰 관리 식들)을 생성한다 – 그러한 규칙들은 관계형 구성들에 대하여 개발되었다. QMView들에 대한 델타 식들은 통지를 위하여 필요하다는 것에 유의한다. N-MVL 매핑들의 경우, 델타 식들은 디자인 타임에 우리에 의해 결정된다. 왜냐하면 이들 매핑은, 예를 들어, 우리가 Address 컬렉션을 일 대 다 연결로 매핑할 때 규정되고, 또한 우리는 대응하는 뷰 관리 식들을 디자인하기 때문이다.

[0345] 상기 뷰들 및 변환들(N-MVL 및 R-MVL)이 주어지면, 우리는 그것들을 조합하여 저장소(1533) 내의 테이블들에 의하여 개체들(1531)을 표현할 수 있는 쿼리 매핑 뷰들, 및 개체들(1531)에 의하여 테이블들(1533)을 표현할 수 있는 업데이트 매핑 뷰들을 얻을 수 있다. 도시되어 있는 바와 같이, 우리는 1532에서의 EDM 엔티티들이 런타임 동안 매핑으로부터 완전히 제거되지 않도록 매핑 뷰들을 유지하기로 결정할 수 있다 – 이들 뷰들을 유지하기 위한 가능한 이유는 EDM 제약들을 이용하는 특정 종류의 쿼리 최적화를 가능하게 하는 것이다. 물론, 이것은 우리가 실제로 런타임에 EDM 엔티티들을 저장한다는 것을 의미하지는 않는다.

[0346] 도 16은 상이한 컴포넌트들이 상술한 뷰 컴파일 프로세스를 어떻게 성취하는지를 보여준다. 애플리케이션들은 API(1600)를 호출한다. 뷰 생성기들(1601, 1603)은 다음 3가지 기능: 비관계형 구성들을 RES 구성들로 변환하는 기능, 쿼리/업데이트 뷰들을 생성하는 기능, 및 업데이트들 및 통지들을 전파하기 위한 델타 식들을 생성하는 기능의 책임이 있다. 그것들은 이들 기능들을 수행할 때 메타데이터(1602)를 이용할 수 있다. OE 뷰 구성자(view composer)(1605)는 개체 및 EDM 정보를 취하여 그것을 우리가 EDM 타입들에 의하여 개체들의 대수 식들을 갖도록 구성하고; 유사하게, ES 뷰 구성자(1606)는 테이블들에 의하여 EDM 타입들의 대수 식을 생성한다. 우리는 이들 뷰들을 OS 뷰 구성자(1607)에서 더 구성하고 메타데이터 저장소(1608)에서 단 하나의 뷰들의 세트를 얻는다. 위에서 논한 바와 같이, 우리는 가능한 쿼리 최적화 기회들에 대하여 2개의 뷰들의 세트를 유지할 수 있다. 마지막으로, 종속성 분석 컴포넌트(1604)가 ES 뷰 생성기 출력에 작용하여 메타데이터 저장소(1608)에 종속성 순서를 제공할 수도 있다.

[0347] **맵 컴파일 요약(Map Complilation Summary)**

[0348] 요약하면, 클래스, EDM 타입, 또는 테이블의 각 명세 M에 대하여, 우리는 대응하는 RES들 및 M과 대응하는 RES 간의 규정된 변환들을 생성한다. 따라서 우리는 도 15에 예시된 바와 같이 다음을 생성한다:

[0349] 1. M에 대응하는 RES – RES-COM(M), RES-개체(M) 또는 RES-저장소(M)으로 표시됨.

[0350] 2. RES 관계들에 의하여 각 명세 M을 표현하는 규정된 변환.

[0351] 3. M에 의하여 그러한 RES 관계를 표현하는 규정된 변환.

[0352] 4. 쿼리 매핑 뷰들: 2개의 그러한 뷰가 있다 – EDM 타입들에 의하여 개체들을 표현하는 OE QMView들 및 저장

소(테이블들)에 의하여 EDM 타입들을 표현하는 ES QMView들.

[0353] 5. 업데이트 매핑 뷰들: 2개의 그러한 뷰가 있다 – 개체들에 의하여 EDM 타입들을 표현하는 OE UMView들 및 EDM 타입들에 의하여 저장소 테이블들을 표현하는 ES UMView들.

[0354] 6. 업데이트들의 점진적 관리를 위하여, 우리는 또한 UMIView들에 대한 델타 식들을 생성한다.

[0355] 이들 뷰들을 구성한 후에, 우리는 4개의 맵으로 끝난다. 이들 맵은 메타데이터 저장소(1608)에 저장되고 총괄하여 컴파일된 매핑 뷰들(Compiled Mapping Views)이라 불린다:

[0356] 쿼리 맵들(Query Maps): CDM/테이블들에 의하여 개체들/CDM을 표현한다.

[0357] 업데이트 맵들(Update Maps): CDM/개체들에 의하여 테이블들/CDM을 표현한다.

[0358] 업데이트 델타 식들(Update Delta Expressions): CDM/개체들에 대한 델타들에 의하여 테이블들/CDM에 대한 델타들을 표현한다.

[0359] 통지 델타 식들(Notification Delta Expressions): CDM/테이블들에 대한 델타들에 의하여 개체들/CDM에 대한 델타들을 표현한다.

[0360] 종속성 순서(Dependency Order): 상이한 관계들에 대하여 다양한 삽입, 삭제, 업데이트 동작들이 수행되어야 하는 순서 – 이 순서는 데이터베이스 제약들이 업데이트 처리 동안에 위반되지 않는 것을 보증한다.

[0361] **컬렉션 예**

[0362] 이제 우리는 고려하고 있는 Person 예에 대한 규정된 변환들 및 비규정된 변환을 간략히 보여준다. 우리는 쿼리 및 업데이트 매핑 뷰들 양자 모두를 제시한다 – 대응하는 뷰 관리 표현들에 대해서는 아래에서 더 논의한다.

[0363] **RES들**

[0364] 우리는 OPerson을 단순히 이름과 pid를 반영하는 RES 구성 R_OPerson으로 변환한다; 유사하게, 우리는 OAddress를 R_OAddress로 변환한다. 주소들의 컬렉션을 변환하기 위해, 우리는 일 대 다 연결 R_OPerson_Address를 이용한다. 유사하게, EDM 구성들에 대해서도 마찬가지이다. 테이블들(R_SPerson, R_SAddress)에 대한 RES들은 SPerson 및 SAddress로의 항등 매핑들(identity mappings)이다. 이들 RES는 다음과 같다:

[0365]

R_OPerson(pid, name) R_OAddress(aid, state) R_OPerson_Address(pid, aid)	R_CPerson(pid, name) R_CAddress(aid, state) R_CPerson_Address(pid, aid)	R_SPerson(pid, name) R_SAddress(pid, aid, state)

[0366] **쿼리 매핑 뷰들**

[0367] 우리는 (개체-EDM 및 EDM-저장소 매핑들에 걸쳐서 구성되는) 개체-저장소 매핑을 보여준다.

[0368] *RES 공간에서의 비규정된 뷰들*

[0369] 개체와 EDM 저장소 간의 매핑들은 본질적으로 항등(identity)이다. 모든 3개의 뷰들 R_CPerson, R_CAddress 및 R_CPerson_Address는 단순히 R_SPerson 및 R_SAddress 상의 프로젝션들이다.

```
CREATE VIEW
R_OPerson (pid, name)
AS
SELECT pid, name
FROM R_CPerson
```

```
CREATE VIEW
R_OPerson_Address (pid, aid)
AS
SELECT pid, aid
FROM R_CPerson_Address
```

```
CREATE VIEW
R_OAddress (aid, state) AS
SELECT aid, state
FROM R_CAddress
```

```
CREATE VIEW
R_CPerson (pid, name)
AS
SELECT pid, name
FROM R_SPerson
```

```
CREATE VIEW
R_CPerson_Address (pid, aid)
AS
SELECT pid, aid
FROM R_SAddress
```

```
CREATE VIEW
R_CAddress (aid, state) AS
SELECT aid, state
FROM R_SAddress
```

[0370]

[0371] *규정된 변환(RES-개체들에 의하여 표현된 개체들)*

[0372] OPerson은 R_OPerson, R_OAddress 및 R_OPerson_Address를 이용하여 R_OPerson_Address와 R_OAddress와의 조인을 행하고 그 결과를 네스팅함으로써 표현된다.

```
CREATE PRESCRIBED VIEW OPerson (pid, name, addrs) AS
SELECT pid, name, NEST(SELECT Address(a.aid, a.state)
                       FROM R_OAddress a, R_OPerson_Address pa
                       WHERE pa.pid = p.pid AND a.aid = pa.aid)
R_OPerson p
```

[0373]

[0374] *CPerson의 구성된 뷰*

[0375] 단순화 후의 구성된 표현은 다음과 같을 수 있다(이 예에 대하여 테이블들과 그들의 RES 구성들 간에 항등 변환을 갖는다는 것을 상기한다):

```
CREATE VIEW OPerson (pid, name, addrs) AS
SELECT pid, name, NEST(SELECT Address(a.aid, a.state)
                       FROM SAddress a
                       WHERE a.pid = p.pid)
SPerson p
```

[0376]

[0377] 최종 뷰는 "직접" 매핑 접근법을 이용하여 획득할 것으로 기대했을 수 있는 것을 나타낸다. RES 접근법의 하나의 이익은 우리가 업데이트 파이프라인에 대한 델타 식들을 볼 때, 및 또한 우리가 쿼리 매핑 뷰들에 대한 델타 식들을 필요로 하는 경우 통지 파이프라인에서 나타난다.

[0378] **업데이트 매핑 뷰들**

[0379] *RES 공간에서의 비규정된 뷰들*

[0380] R_SPerson에 대한 UMView는 단순히 R_CPerson 상의 프로젝션인 반면 R_SAddress는 R_CAddress를 일 대 다 연결 테이블 – R_CPerson_Address를 조인함으로써 구성된다. CDM과 개체 공간 간의 매핑은 항등이다.

```
CREATE VIEW
R_CPerson (pid, name)
AS
SELECT pid, name
FROM R_OPerson
```

```
CREATE VIEW
R_CPerson_Address (pid, aid)
AS
SELECT pid, aid
FROM R_OPerson_Address
```

```
CREATE VIEW
R_CAddress (aid, state) AS
SELECT aid, state
FROM R_OAddress
```

```
CREATE VIEW R_SPerson (pid,
name) AS
SELECT pid, name,
FROM R_CPerson
```

```
CREATE VIEW R_SAddress(aid, pid, state) AS
SELECT aid, pid, state
FROM R_CPerson_Address, R_CAddress
WHERE R_CPerson_Address.aid =
R_CAddress.aid
```

[0381]

[0382] *규정된 변환(개체들에 의하여 표현된 RES-개체들)*

[0383] 우리는 업데이트들이 개체 공간으로부터 RES 공간으로 푸시될 수 있도록 개체들을 RES들로 변환할 필요가 있다. R_OPerson에 대한 규정된 변환은 단순한 프로젝션인 반면 R_OAddress 및 R_OPerson_Address에 대한 변환들은 사람과 그의 주소들 간에 조인을 수행함으로써 성취된다. 이것은 "포인터 조인(pointer join)" 또는 "탐색 조인(navigation join)"이다.

[0384]

CREATE PRESCRIBED VIEW R_OPerson (name, pid) AS SELECT name, pid FROM OPerson	CREATE PRESCRIBED VIEW R_OAddress(state, aid) AS SELECT a.state, a.aid FROM OPerson p, p.addrs a	CREATE PRESCRIBED VIEW R_OPerson_Address(pid, aid) AS SELECT pid, aid FROM OPerson p, p.addrs a

[0385] *구성된 업데이트 매핑 뷰들*

[0386] 우리는 상기 뷰들(및 어떤 단순화화 함께)을 구성하여 다음의 구성된 업데이트 매핑 뷰들을 얻는다:

[0387]

CREATE VIEW SPerson (pid, name) AS SELECT pid, name, FROM OPerson	CREATE VIEW SAddress(aid, pid, state) AS SELECT a.aid, p.pid, a.state FROM OPerson p, p.addrs a

[0388] 따라서, 테이블 SPerson은 OPerson 상의 단순한 프로젝션으로서 표현될 수 있는 반면 SAddress는 OPerson을 그의 주소들과 조인함으로써 얻어진다.

[0389] **뷰 유효성 검사(View Validation)**

[0390] 생성된 뷰들이 만족시킬 필요가 있는 중요한 속성은 정보의 어떤 손실도 방지하기 위해 그것들이 "라운드트립(roundtrip)"해야 한다는 것이고, 우리는 엔티티/개체가 저장되고 그 후 검색될 때 정보의 손실이 없는 것을 보증해야 한다. 바꾸어 말하면, 우리는 모든 엔티티들/개체들 D에 대하여 보증하기를 원한다:

[0391] D = QMView(UMView(D))

[0392] 우리의 뷰 생성 알고리즘은 이 속성을 보증한다. 만일 이 속성이 참이면, 우리는 또한 "쿼리 및 업데이트 뷰들이 라운드트립한다" 또는 양방향이라고 말한다. 우리는 이제 사람-주소 예에 대하여 이 속성을 증명한다. 간결함을 위하여, 우리는 RES 공간에서의 라운드트리핑에 초점을 맞춘다.

[0393] *R_OPerson에 대한 유효성 검사*

[0394] OPerson을 쿼리 뷰 내의 SPerson으로 대체하면, 다음을 얻는다:

[0395]
```
R_OPerson(pid, name, age) =
SELECT pid, name, age FROM (SELECT pid, name, age FROM R_SPerson)
```

[0396] 단순화하면 다음을 얻는다.

[0397]
```
R_OPerson(pid, name, age) = SELECT pid, name, age FROM R_SPerson
```

[0398] 이것은 SELECT*FROM Person에 상당한다.

[0399] *OPerson_Address에 대한 유효성 검사*

[0400] R_OPerson_Address의 경우에는, 약간 더 복잡하다. 다음과 같다:

[0401]
```
R_OPerson_Address (pid, aid) = SELECT pid, aid FROM R_SAddress
```

[0402] R_SAddress를 대체하면, 다음을 얻는다:

[0403]
```
R_OPerson_Address (pid, aid) =
SELECT pid, aid
FROM (SELECT aid, pid, state
      FROM R_OPerson_Address pa, R_OAddress a
      WHERE pa.aid = a.aid)
```

[0404] 이것은 다음과 같이 단순화된다:

[0405]
```
R_OPerson_Address (pid, aid) =
SELECT pid, aid FROM R_OPerson_Address pa, R_OAddress a WHERE pa.aid =
a.aid
```

[0406] 상기한 것이 실제로 SELECT*FROM R_OPerson_Address인 것을 보여주기 위해, 우리는 외래 키 종속성 R_OPerson_Address.aid → R_OAddress.aid를 가질 필요가 있다. 만일 이 종속성이 유지되지 않으면,우리는 라운드트립할 수 없다. 그러나 세트 값의 속성(set-valued property) addrs의 범위가 R_OAddress이므로 그것은 유지된다. 이 외래 키 제약은 2가지 방법으로 기술될 수 있다:

[0407] 

[0408] 상기 식에서 이 제약을 대체하면 다음과 같다:

[0409] R_OPerson_Address (pid, aid) = SELECT pid, aid FROM R_OPerson_Address

[0410] *Address에 대한 유효성 검사*

[0411] R_OAddress는 다음과 같이 주어진다:

[0412] R_OAddress (aid, state) = SELECT aid, state FROM R_SAddress

[0413] R_SAddress를 대체하면 다음과 같다.

[0414]
```
R_OAddress(aid, state) =
SELECT aid, state
FROM (SELECT aid, pid, state
      FROM R_OPerson_Address pa, R_OAddress a
      WHERE pa.aid = a.aid)
```

[0415] 이것은 다음과 같이 바꿔 기술될 수 있다:

[0416]
```
R_OAddress(aid, state) = SELECT aid, state FROM R_OPerson_Address pa;
R_OAddress a,
                    WHERE pa.aid = a.aid
```

[0417] 여기서, 외래 키 종속성 R_OAddress.aid → R_OPerson_Address.aid가 유지된다면 R_OPerson_Address와의 조인은 중복이다. 이 종속성은 R_OAddress가 R_OPerson에 실체론적으로 종속하는(즉, addrs가 구성(composition)인) 경우에만 유지된다. 만일 그것이 참이 아니라면, 우리의 뷰들은 라운드트립하지 않을 것이다. 따라서, 우리는 다음의 제약을 갖는다:

[0418] πaid, state(R_OAddress ⋈aid R_OPerson_Address) = R_OAddress

[0419] 따라서, 우리는 다음의 식을 얻는다:

[0420] R_OAddress(aid, state) = SELECT aid, state FROM R_OAddress

[0421] 쿼리 변환

[0422] **쿼리 변환기(Query Translator)**

[0423] EDP 쿼리 변환기(EQT)는 매핑 메타-데이터를 이용함으로써 개체/EDM 공간으로부터의 쿼리들을 프로바이더 공간으로 변환하는 책임이 있다. 사용자 쿼리들은 각종 구문들, 예를 들어, eSQL, C# 시퀀스, VB SQL 등으로 표현될 수 있다. EQT 아키텍처는 도 17에 도시되어 있다. 이제 EQT의 상이한 컴포넌트들에 대하여 설명한다.

[0424] 파서(parser)(1711)는 eSQL, LINQ(Language Integrated Query), C# 시퀀스, 및 VB SQL을 포함하는 몇몇 형식들 중 하나로 표현된 사용자 쿼리를 파싱(parsing)함으로써 구문 분석을 수행한다. 이때 임의의 구문 오류가 검출되어 플래그된다.

[0425] LINQ의 경우, 구문 분석(및 의미 분석)은 언어(C#, VB 등) 자체의 구문 분석 단계와 통합된다. eSQL의 경우,

구문 분석 단계는 쿼리 프로세서의 일부이다. 통상적으로 언어마다 하나의 구문 분석기가 있다.

[0426] 구문 분석 단계의 결과는 파스 트리(parse tree)이다. 그 후 이 트리는 의미 문석 단계(1712)에 공급된다.

[0427] 매개 변수 바인더 및 의미 분석기 컴포넌트(1712)는 사용자 쿼리들 내의 매개 변수들을 관리한다. 이 모듈은 쿼리 내의 매개 변수들의 데이터 타입 및 값들을 추적한다.

[0428] 의미 분석 단계는 구문 분석 단계(1711)에 의해 생성된 파스 트리를 의미적으로 유효성 검사한다. 쿼리 내의 임의의 파라미터들은 이때 이미 바인딩되어 있어야 한다. 즉, 그들의 데이터 타입들이 알려져야 한다. 여기에서 임의의 의미 오류들이 검출되고 플래그된다; 만일 성공적이면, 이 단계의 결과는 의미 트리(semantic tree)이다.

[0429] LINQ의 경우, 전술한 바와 같이, 의미 분석 단계는 언어 자체의 의미 분석 단계들과 통합된다는 것에 유의한다. 언어마다 하나의 구문 트리(syntax tree)가 있으므로 통상적으로 언어마다 하나의 의미 분석기가 있다.

[0430] 의미 분석 단계는 논리적으로 다음을 포함한다:

[0431] 1. 이름 분석(Name Resolution): 쿼리 내의 모든 이름들이 이때 분석된다. 이것은 범위(extents), 타입, 타입의 속성, 타입의 메서드 등에의 참조들을 포함한다. 부가적인 효과로서, 그러한 식들의 데이터타입들도 추론된다. 이 하위 단계(sub-phase)는 메타데이터 컴포넌트와 상호작용한다.

[0432] 2. 타입 체크 및 추론(Type Checking and Inferencing): 쿼리 내의 식들이 타입 체크되고, 결과 타입들이 추론되다.

[0433] 3. 유효성 검사: 여기서는 다른 종류의 유효성 검사가 일어난다. 예를 들면, SQL 프로세서에서는, 만일 쿼리 블록이 그룹-바이 절(group-by clause)을 포함하면, 이 단계는 선택 목록이 그룹-바이 키(group-by keys) 또는 집계 함수(aggregate functions)만을 참조할 수 있는 제한을 실시하기 위해 이용될 수 있다.

[0434] 의미 분석 단계의 결과는 의미 트리(semantic tree)이다. 이때, 쿼리는 유효하다고 간주된다 – 나중에 쿼리 컴파일 동안 언제든지 더 이상의 의미 오류가 일어나지 않을 것이다.

[0435] 대수화 단계(algebraization phase)(1713)는 의미 분석 단계(1712)의 결과를 취하여, 그것을 대수 변환에 더 적합한 형식으로 변환한다. 이 단계의 결과는 논리 확장된 관계형 연산자 트리(logical extended relational operator tree)이고, 별칭은 대수 트리(algebra tree)이다.

[0436] 대수 트리는 코어 관계형 대수 연산자들 – select, project, join, union - 에 기초하고, 이것을 nest/unnest 및 pivot/unpivot과 같은 추가의 연산들로 확장한다.

[0437] 쿼리 변환기의 뷰 언폴딩 단계(view unfolding phase)(1714)는 사용자 쿼리에서 참조되는 임의의 개체들을, 아마도 재귀적으로, QMView 식들로 대체한다. 뷰 변환 프로세스의 맨 마지막에, 우리는 저장소 용어의 의하여 쿼리를 기술하는 트리를 얻는다.

[0438] 개체 계층의 경우에, 뷰 언폴딩은 저장소 공간에 이르는 내내 행하여졌을 수 있고(메타데이터 리포지토리 내에 최적화된 OS 매핑이 저장된 경우) 또는 쿼리 트리는 EDM 계층으로 변환되었을 수 있다. 후자의 경우, 우리는 이 트리를 취하고 그것을 EDM 개념들이 이제 저장소 개념들로 변환되는 요건과 함께 뷰 언폴딩 컴포넌트에 재공급한다.

[0439] 변환/단순화 컴포넌트(1715)는 프로바이더(1730) 특정일 수 있고, 또는 대안 실시예에서는 다양한 프로바이더들에 의해 이용될 수 있는 EDP 일반 컴포넌트(EDP-generic component)일 수 있다. 쿼리 트리에 대하여 변환을 수행하는 몇 가지 이유가 있다.

[0440] 1. 저장소에 푸시하는 연산자: EQT는 복잡 연산자들(complex operators)(예컨대, join, filter, aggregate)를 저장소에 푸시한다. 그렇지 않으면, 그러한 연산자들은 EDP에서 구현되어야 할 것이다. EDP의 값 구체화 계층은 네스팅 등의 "비관계형 보상(non-relational compensating)" 연산들만을 수행한다. 만일 우리가 연산자 X를 쿼리 트리 내의 값 구체화 노드들 아래로 푸시할 수 없고 값 구체화 계층이 연산 X를 수행할 수 없다면, 우리는 그 쿼리를 불법(illegal)이라고 선언한다. 예를 들면, 쿼리가 프로바이더에 푸시될 수 없는 집계 연산(aggregation operation)이라면, 우리는 그 쿼리를 불법이라고 선언할 것이다. 왜냐하면 값 구체화 계층은 어떤 집계도 수행하지 않기 때문이다.

[0441] 향상된 성능: 쿼리의 감소된 복잡성은 중요하고 우리는 거대한 쿼리들을 백엔드 저장소에 송신하는 것을 피하고

싶다. 예를 들면, WinFS에서의 현재의 쿼리들 중 일부는 매우 복잡하고 실행하는 데 많은 양의 시간이 걸린다(대응하는 손으로 쓴 쿼리들이 그 10배 이상(more than an order of magnitude) 더 빠르다).

[0442] 향상된 디버그 능력: 보다 단순한 쿼리들은 또한 개발자가 시스템을 디버그하고 무엇이 진행중인지를 이해하는 것을 더욱 용이하게 할 것이다.

[0443] 변환/단순화 모듈(1715)은 쿼리를 나타내는 대수 트리의 일부 또는 전부를 동등한 서브트리들로 변환할 수 있다. 이들 발견적 방법 기반(heuristic-based) 변환들은 논리적이라는, 즉, 비용 모델을 이용하여 행해지지 않는다는 것에 유의한다. 논리 변환들의 종류는 다음의 예시적인 프로바이더 특정 서비스들을 포함할 수 있다:

[0444] 서브-쿼리 플래트닝(sub-query flattening)(뷰 및 네스트된 서브 쿼리들)

[0445] 조인 제거(join elimination)

[0446] 술어 제거 및 통합(predicate elimination and consolidation)

[0447] 술어 푸시다운(predicate pushdown)

[0448] 공통 하위 식 제거(common sub-expression elimination)

[0449] 프로젝션 가지치기(projection pruning)

[0450] 외부 조인(outer join) → 내부 조인(inner join) 변환

[0451] 좌측 상관 제거(eliminating left-correlation)

[0452] 이 SQL 생성 모듈(1731)은 프로바이더 컴포넌트(1730)의 일부이다. 왜냐하면 생성된 SQL은 프로바이더에 특정하기 때문이다. 단순화 후에, 대수 트리는 프로바이더에게 넘겨지고 프로바이더는 적절한 SQL을 생성하기 전에 프로바이더 특정 변환 또는 단순화를 더 수행할 수 있다.

[0453] 서버에서 쿼리가 실행된 후에, 그 결과들은 EDM 클라이언트에 스트리밍된다. 프로바이더(1730)는 EDM 엔티티들로서 결과들을 얻기 위해 애플리케이션에 의해 이용될 수 있는 DataReader들을 익스포즈한다. 값 구체화 서비스(1741)는 이들 리더들을 취하고 그것들을 (새로운 DataReader들로서) 관련 EDM 엔티티들로 변환할 수 있다. 이들 엔티티들은 애플리케이션에 의해 소비될 수 있고 새로운 DataReader들은 상위 개체 구체화 서비스에 전달될 수 있다.

[0454] EQT(1700)는 쿼리 트리 내의 연산자로서 구체화(materializatin)를 나타낸다. 이것은 통상의 쿼리 변환 파이프라인이 EDM 공간에서 개체들을 생성하는 것을 허용하고, 그 후 그것들은 실제 구체화를 수행하기 위해 특별한 대역외 연산들(out-of-band operations)을 요구하는 대신, 사용자들에게 바로 공급된다. 이것은 또한 사용자 쿼리들에 대해 수행될 부분 개체 페치(partial object fetch), 이거 로딩(eager loading) 등과 같은 다양한 최적화를 허용한다.

[0455] **쿼리 예**

[0456] 우리가 전개해오고 있는 Person-Address 예를 생각해보자. 사용자는 다음의 쿼리 – 워싱턴주(WA) 내의 모든 사람들을 찾기 – 를 실행하기를 원한다. 우리는 이 쿼리를 의사-CSQL로 다음과 같이 작성할 수 있다:

[0457]
```
SELECT x.name FROM OPerson x, x.addrs y WHERE y.state = "WA"
```

[0458] 만일 우리가 이 시점에서 Person에 대한 쿼리 뷰를 이용하여 뷰-언폴딩을 행한다면, 다음이 얻어진다:

[0459]
```
SELECT x.name
FROM (SELECT pid, name,
      NEST(SELECT OAddress(a.aid, a.state) FROM SAddress a where a.pid =
p.pid)
      FROM SPerson p) as x, x.addrs y
WHERE y.state = "WA"
```

[0460] 이 쿼리는 백엔드 서버에 송신되기 전에 다음과 같이 단순화될 수 있다:

[0461]
```
SELECT p.name
FROM SPerson p, SAddress a
WHERE p.pid = a.pid
```

[0462] **메타데이터**

[0463] EQT는 쿼리의 컴파일 및 실행 동안에 다양한 메타데이터를 요구한다. 이 메타데이터는 다음을 포함한다.

[0464] 애플리케이션-공간(application-space) 메타데이터: 사용자 쿼리들을 유효성 검사하기 위해 의미 분석 동안에 요구되는 범위/컬렉션(Extents/Collections), 타입, 타입 속성, 타입 메서드에 관한 정보.

[0465] 스키마-공간(schema-space) 메타데이터: 뷰 컴파일 동안에 요구되는 엔티티 컬렉션(Entity Collections), CDM 타입 및 속성들에 관한 정보. 변환을 위한 엔티티들에 대한 제약들과 엔티티들 간의 관계에 관한 정보.

[0466] 저장소-공간(storage-space) 메타데이터: 전술한 바와 같음.

[0467] 애플리케이션 → 스키마 매핑들: 뷰 확장(View Expansion)을 위해 요구되는 뷰 정의를 나타내는 논리 연산자 트리.

[0468] 스키마 → 저장소 매핑들: 전술한 바와 같음.

[0469] **오류 보고 파이프라인(Error Reporting Pipeline)**

[0470] 쿼리 처리의 다양한 단계들에서의 오류들은 사용자가 이해할 수 있는 용어로 보고되어야 한다. 다양한 컴파일 및 실행 타임 오류들이 쿼리 처리 동안에 일어날 수 있다. 구문 및 의미 분석 동안의 오류들은 주로 애플리케이션 공간에 있고, 매우 작은 특별 처리를 요구한다. 변환 동안의 오류들은 주로 리소스 오류(resource errors)(메모리 초과(out-of-memory) 등)이고, 어떤 특별 처리를 필요로 한다. 코드 생성 및 후속의 쿼리 실행 동안의 오류들은 적절히 처리될 필요가 있을 수 있다. 오류 보고에 있어서의 주요 도전적 과제는 추상화의 하위 레벨들에서 일어나는 런타임 오류들을 애플리케이션 레벨에서 의미 있는 오류들로 매핑하는 것이다. 이것은 우리가 저장소, 개념적, 및 애플리케이션 매핑들을 통하여 하위 레벨 오류들을 처리할 필요가 있다는 것을 의미한다.

[0471] **쿼리 예**

[0472] 우리의 샘플 OO 쿼리는 워싱턴주에 주소를 갖는 모든 사람들의 이름을 페치한다:

[0473]
```
SELECT p.name
FROM OPerson p, p.addrs as a
WHERE a.state = 'WA'
```

[0474] *단계 1: 관계형 용어로의 변환*

[0475] 이 쿼리는 R_OPerson, R_OPerson_Address, 및 R_OAddress에 의하여 표현된 다음의 순전히 관계형 쿼리로 변환될 수 있다. 본질적으로, 우리는 다양한 탐색 속성들(점 "." 식들)을 필요한 경우 조인 식들(join expressions)로 확장하고 있다.

[0476]
```
SELECT p.name
FROM R_OPerson p, R_OPerson_Address pa, R_OAddress a
WHERE p.pid = pa.pid AND pa.aid = a.aid AND a.state = 'WA'
```

[0477] 쿼리는 여전히 개체 도메인에 있고 개체 범위들의 용어로 되어 있는 것에 유의한다.

[0478] *단계 2: 뷰 언폴딩: 저장소 공간으로의 변환*

[0479] 이제 우리는 쿼리를 SQL로 변환하기 위해 뷰 언폴딩을 행한다:

```
SELECT p.name
FROM (SELECT pid, name, age FROM SPerson) p,
     (SELECT pid, aid FROM SAddress) pa,
     (SELECT aid, state FROM SAddress) a
WHERE p.pid = pa.pid AND pa.aid = a.aid AND a.state = 'WA'
```

[0480]

[0481] *단계 3: 쿼리 단순화*

[0482] 우리는 이제 일련의 논리적 변환들을 적용하여 이 쿼리를 단순화할 수 있다.

```
SELECT p.name
FROM SPerson p, SAddress pa, SAddress a
WHERE p.pid = pa.pid AND pa.aid = a.aid AND a.state = 'WA'
```

[0483]

[0484] 이제, 우리는 SAddresss(aid)의 기본 키 상의 중복 셀프-조인(self-join)을 제거하여 다음을 얻을 수 있다:

```
SELECT p.name
FROM SPerson p, SAddress a
WHERE p.pid = a.pid AND a.state = 'WA'
```

[0485]

[0486] 상기한 것의 모두는 상당히 수월하다. 이제 우리는 SQL 서버에 송신될 수 있는 쿼리를 갖게 된다.

[0487] 업데이트를 위한 컴파일 타임 처리

[0488] EDP는 애플리케이션들이 새로운 개체들을 생성하고, 그것들을 업데이트하고 그것들을 삭제하고 그 후 이들 변경들을 지속성 있게 저장하는 것을 허용한다. OR-매핑 컴포넌트는 이들 변경들이 백엔드 저장소 변경들로 정확하게 변환되는 것을 보증할 필요가 있다. 전술한 바와 같이, 우리는 개체들에 의하여 테이블을 선언하는 업데이트 매핑 뷰들을 이용한다. 그러한 뷰들을 이용함으로써, 우리는 기본 관계들에 대한 변경들이 뷰들에 전파될 필요가 있는 경우 업데이트 전파 문제를 구체화된 뷰 관리 문제로 본질적으로 바꾸었고; UMView들의 경우에, "기본 관계들(base relations)"은 개체들이고 "뷰들(views)"은 테이블들이다. 이런 식으로 문제를 모델링함으로써, 우리는 관계형 데이터베이스 분야에서 개발되어 온 뷰 관리 기술에 대한 지식을 이용할 수 있다.

[0489] **업데이트 매핑 뷰 생성**

[0490] 쿼리 경우에서와 같이, 업데이트들에 대한 다수의 매핑 작업이 컴파일 타임에 수행된다. 클래스들, EDM 타입들, 및 테이블들에 대한 관계형으로 표현된 스키마들(Relationally Expressed Schemas)과 함께, 우리는 이들 타입들과 대응하는 RES 구성들 간의 규정된 변환들을 생성한다. 우리는 또한 EDM 타입들과 클래스들의 RES 구성들 사이 및 저장소 테이블들과 EDM 타입들의 fEES 구성들 사이의 업데이트 매핑 뷰들을 생성한다.

[0491] 우리가 전개해오고 있는 Person-Address 예의 도움을 받아 이들 UMView들을 이해해보자. 구성된 개체들에 대한 RES 구성들(R_OPerson, R_OAddress, R_OPerson_Address)을 상기한다.

[0492] *업데이트 매핑 뷰(개체들의 RES에 의하여 표현된 테이블들의 RES)*

[0493] R_OPerson에 대한 UMView는 단순히 R_SPerson 상의 프로젝션인 반면 R_SAddress는 R_OAddress를 일 대 다 연결 테이블 – R_OPerson_Address와 조인함으로써 구성된다.

```
CREATE VIEW R_SPerson (pid, name) AS
SELECT pid, name
FROM R_OPerson
```

```
CREATE VIEW R_SAddress(aid, pid, state) AS
SELECT aid, pid, state
FROM R_OPerson_Address pa, R_OAddress a
WHERE pa.aid = a.aid
```

[0494]

[0495] *규정된 변환(개체들에 의하여 표현된 RES)*

[0496] 우리는 업데이트들이 개체 공간으로부터 RES 공간으로 푸시될 수 있도록 개체들을 RES들로 변환할 필요가 있다. 우리는 개체의 가상 메모리 주소를 pid 및 aid 키들로 변환하기 위해 "o2r" 함수를 이용한다 – 이 구현에서 우리는 단순히 개체의 그림자 상태(shadow state)로부터 키들을 얻을 수 있다. R_OPerson에 대한 규정된 변환은

단순한 프로젝션인 반면 R_OAddress 및 R_OPerson_Address에 대한 변환들은 사람과 그의 주소들 간의 조인을 수행함으로써 얻어진다.

[0497]

CREATE PRESCRIBED VIEW R_OPerson (name, pid) AS SELECT name, pid FROM OPerson	CREATE PRESCRIBED VIEW R_OAddress(state, aid) AS SELECT a.state, a.aid FROM OPerson p, p.addrs a
CREATE PRESCRIBED VIEW R_OPerson_Address(pid, aid) AS SELECT p.pid, a.aid. FROM OPerson p, p.addrs a	

[0498] *구성된 업데이트 매핑 뷰들*

[0499] 우리는 상기 뷰들(및 어떤 단순화와 함께)을 구성하여 다음의 구성된 업데이트 매핑 뷰들을 얻는다:

[0500]

CREATE VIEW SPerson (pid, name) AS SELECT pid, name FROM OPerson	CREATE VIEW SAddress(aid, pid, state) AS SELECT a.aid, p.pid, a.state FROM OPerson p, p.addrs a

[0501] 따라서, 테이블 SPerson은 OPerson 상의 단순한 프로젝션으로서 표현될 수 있는 반면 SAddress는 OPerson을 그의 주소들과 조인함으로써 얻어진다.

[0502] **델타 식 생성**

[0503] 애플리케이션이 그의 개체 변경들이 백엔드에 저장될 것을 요구할 경우, 실시예들은 이들 변경들을 백엔드 저장소로 변환할 수 있다. 즉, 개체들(기본 관계들)의 델타 식들에 의하여 테이블들(뷰들)의 델타 식들을 생성할 수 있다. 이것은 뷰-생성/컴파일 프로세스의 RES 구성들로의 분해가 실제로 도움이 되는 영역이다. 비규정된 매핑들에 대한 델타 식들은 비교적 용이하게 생성될 수 있다. 왜냐하면 이들 매핑들은 관계형 공간에 있고(RES들은 순전히 관계형이다) 이 목표를 성취하기 위해 관계형 데이터베이스에서 다수의 연구가 행해졌기 때문이다. 예를 들면, 데이터베이스 문헌에서, 선택(selections), 프로젝션(projections), 내부(inner) 또는 외부(outer) 또는 세미(semi)-조인(join), 유니언(unions), 교차(intersections), 및 차이(differences) 등의 관계형 연산자들에 의하여 표현되는 뷰들에 대하여 델타 식 규칙들이 개발되어왔다. 비관계형 구성들의 경우, 우리가 해야 할 모든 것은 비관계형 구성들을 RES 공간으로/으로부터 변환하는 규정된 델타 식들을 디자인하는 것이다.

[0504] 우리의 Person 예를 이용하여 델타 식들을 이해해보자. RES 구성(예컨대, R_SAddress)가 2개의 개체 컬렉션들(R_OAddress 및 R_OPerson_Address)의 조인으로서 표현되는 경우를 생각해보자. 그러한 뷰에 대한 델타 식은 다음의 규칙들을 이용하여 얻어질 수 있다(조인 뷰는 V = R JOIN S라고 가정한다):

[0505]
$$i(V) = [i(R) \text{ JOIN } S^{new}] \text{ UNION } [i(S) \text{ JOIN } R^{new}]$$
$$d(V) = [d(R) \text{ JOIN } S] \text{ UNION } [d(S) \text{ JOIN } R]$$

[0506] 이 식에서, i(X) 및 d(X)는 관계 또는 뷰 X에 대한 삽입된 및 삭제된 튜플들을 나타내고 R^{new}는 모든 그의 업데이트들이 적용된 후의 기본 관계들 R의 새로운 값을 나타낸다.

[0507] 따라서, 런타임에서 업데이트들을 용이하게 하기 위해, 예시적인 일 실시예는 먼저 컴파일 타임에서 다음의 델타 식들을 생성한다:

[0508] 1. 업데이트된 개체 컬렉션들(1801)의 그룹들에 대한 델타 변경 식들에 의하여 표현된 RES 관계들(1811)에 대한 규정된 델타 변경 식들(1803), 예를 들어, i(OPerson)에 의하여 표현된 i(R_OPerson).

[0509] 2. RES 관계들(1812)에 대한 델타 변경 식들에 의하여 표현된 테이블들(1802)에 대한 규정된 델타 변경 식들(1804), 예를 들어, i(R_SPerson)에 의하여 표현된 i(SPerson).

[0510] 3. 개체들의 RES 관계들의 델타 식들에 의하여 표현된 테이블들의 RES 관계들에 대한 델타 식들(1813), 예를 들어, i(R_OPerson)에 의하여 표현된 i(R_SPerson).

[0511] 우리는 (1), (2), 및 (3)을 구성하여 개체들(1821)(예컨대, OPerson)에 대한 델타 식들에 의하여 표현된 테이블

들(1822)(예컨대, SPerson)에 대한 델타 식(1820)을 얻을 수 있다. 이러한 구성(composition)은 도 18에 예시되어 있다. 따라서, 쿼리들의 경우에서와 같이, 컴파일 타임에서, 우리는 이제 개체들로부터 테이블들로의 직접 변환을 갖는다. 업데이트들의 경우에, 우리는 실제로 델타 식들을 얻기 위해 RES 분해(breakdown)를 이용하였다(QMView들의 경우, 이러한 이점은 통지를 위해 적용될 수 있다).

[0512] 업데이트들에 대한 델타 식들이 필요하지 않다는 점에 유의한다 – 나중에 알겠지만, 모델 업데이트들은 그것들을 삽입 및 삭제 세트에 넣는 것에 의해 모델링될 수 있고; 변경들이 실제로 데이터베이스에 적용되기 전에 후처리(post-processing) 단계가 나중에 그것들을 도로 업데이트들로 다시 변환한다. 이러한 접근법에 대한 한 가지 이유는 점진적 뷰 관리에 대한 현존하는 연구는 통상적으로 업데이트들을 위한 델타 식들을 갖고 있지 않다는 점이다. 그러한 식들이 개발되는 대안적인, 보다 복잡한 구현들도 가능하다.

[0513] 뷰 구성이 실행된 후에, 테이블들에 대한 델타 식들은 순전히 개체 컬렉션들 및 개체들의 삽입 및 삭제 세트들에 의하여 표현된다. 예를 들어, i(SPerson)은 OPerson, i(OPerson), 및 d(OPerson)에 의하여 표현된다. 이들 델타 식들의 일부는 계산될 개체 컬렉션들을 필요로 할 수 있다. 예를 들어, i(OPerson)은 그의 계산을 위해 EPerson을 필요로 한다. 그러나, 전체 컬렉션은 EDP 클라이언트에서 캐싱되지 않을 수 있다(또는 우리는 컬렉션의 가장 일관성 있고 최신의 값에 대하여 연산을 실행하기를 원할 수 있다). 이러한 문제를 처리하기 위하여, 우리는 개체 컬렉션들을 대응하는 쿼리 매핑 뷰들을 이용하여 언폴딩한다. 예를 들어, 우리는 OPerson에 대한 QMView를 이용하고 그것을 SPerson 및 필요하다면 다른 관계들에 의하여 표현한다. 따라서, 일 실시예에서, 컴파일 프로세스의 맨 마지막에, SPerson에 대한 모든 델타 식들은 i(OPerson), d(OPerson), 및 관계 SPerson 자체에 의하여 표현된다 – 런타임에서, OPerson의 삽입 및 삭제 세트들이 주어지면, 우리는 이제 서버에서 실행될 수 있는 관련 SQL 문들을 생성할 수 있다.

[0514] 요약하면, 개체들과 테이블들의 RES 구성들 간의 QMView들 및 UMView들 및 이들 구성들과 테이블들/개체들 간의 규정된 변환들이 주어지면, 다음의 예시적인 단계들이 수행될 수 있다:

[0515] 1. 상기 단계 1, 2, 및 3에서 언급된 델타 식들을 생성한다.

[0516] 2. 개체들(OPerson)의 델타 식들 및 개체 컬렉션들 자체에 의하여 표현된 테이블들(SPerson)에 대한 델타 식들을 갖도록 이들 식들을 구성한다.

[0517] 3. 개체 컬렉션들을 그들의 QMView들을 이용하여 언폴딩하여 개체들 및 테이블들 자체의 델타 식들에 의하여 표현된 테이블들(SPerson)에 대한 델타 식들을 얻는다. 즉, 개체 컬렉션들이 제거된다. 실시예들이 이러한 언폴딩을 피하거나 전체 컬렉션이 클라이언트에서 캐싱되는 것을 아는 것을 허용하는 특별한 경우들이 존재할 수 있다.

[0518] 4. 런타임 작업을 감소시키도록 식을 단순화/최적화한다.

[0519] 여기에서 명시적으로 제시된 특정 구현들 외에도, 숙련된 당업자들이라면 여기에 개시된 명세서를 고려하여 다른 양태들 및 구현들을 명백히 알 수 있을 것이다. 본 명세서 및 예시된 구현들은 다음의 청구항들의 진정한 범위 및 정신에 관하여, 단지 예로서 간주되는 것으로 의도된다.

도면의 간단한 설명

[0011] 본 발명에 따른 점진적 뷰 관리를 갖는 매핑 아키텍처를 위한 시스템들 및 방법들에 대하여 첨부 도면들을 참조하여 더 설명한다.

[0012] 도 1은 여기서 고려되는 예시적인 엔티티 프레임워크(Entity Framework)의 아키텍처를 도시한다.

[0013] 도 2는 예시적인 관계형 스키마(relational schema)를 도시한다.

[0014] 도 3은 예시적인 엔티티 데이터 모델(Entity Data Model; EDM) 스키마를 도시한다.

[0015] 도 4는 엔트티 스키마(좌측)와 데이터베이스 스키마(우측) 간의 매핑을 도시한다.

[0016] 도 5는 엔티티 스키마 및 관계형 스키마 상의 쿼리들에 의하여 표현되는 매핑을 도시한다.

[0017] 도 6은 도 5의 매핑을 위한 매핑 컴파일러에 의해 생성된 양방향 뷰들 – 쿼리 및 업데이트 뷰들 – 을 도시한다.

[0018] 도 7은 구체화된 뷰 관리 알고리즘들을 이용하여 업데이트들을 양방향 뷰들을 통하여 전파하는 프로세스를 도시

한다.

[0019] 도 8은 매핑 디자이너 사용자 인터페이스를 도시한다.

[0020] 도 9는 쿼리 및 업데이트 뷰들을 생성하기 위해 매핑 명세 언어(Mapping Specification Language; MSL)에서 특정된 매핑을 컴파일하는 것을 도시한다.

[0021] 도 10은 업데이트 처리를 도시한다.

[0022] 도 11은 개체 관계형(OR) 매퍼(mapper)의 예시적인 논리 부분들을 도시한다.

[0023] 도 12는 MSL 명세에서 특정된 매핑을 처리할 때 엔티티 데이터 플랫폼(Entity Data Platform; EDP)에 의해 쿼리 및 업데이트 뷰를 생성하는 것을 도시한다.

[0024] 도 13은 쿼리 변환에서 QMView를 이용하는 것을 도시한다.

[0025] 도 14는 업데이트 변환에서 UMView를 이용하는 것을 도시한다.

[0026] 도 15는 매핑 뷰들의 컴파일-타임 및 런타임 처리를 도시한다.

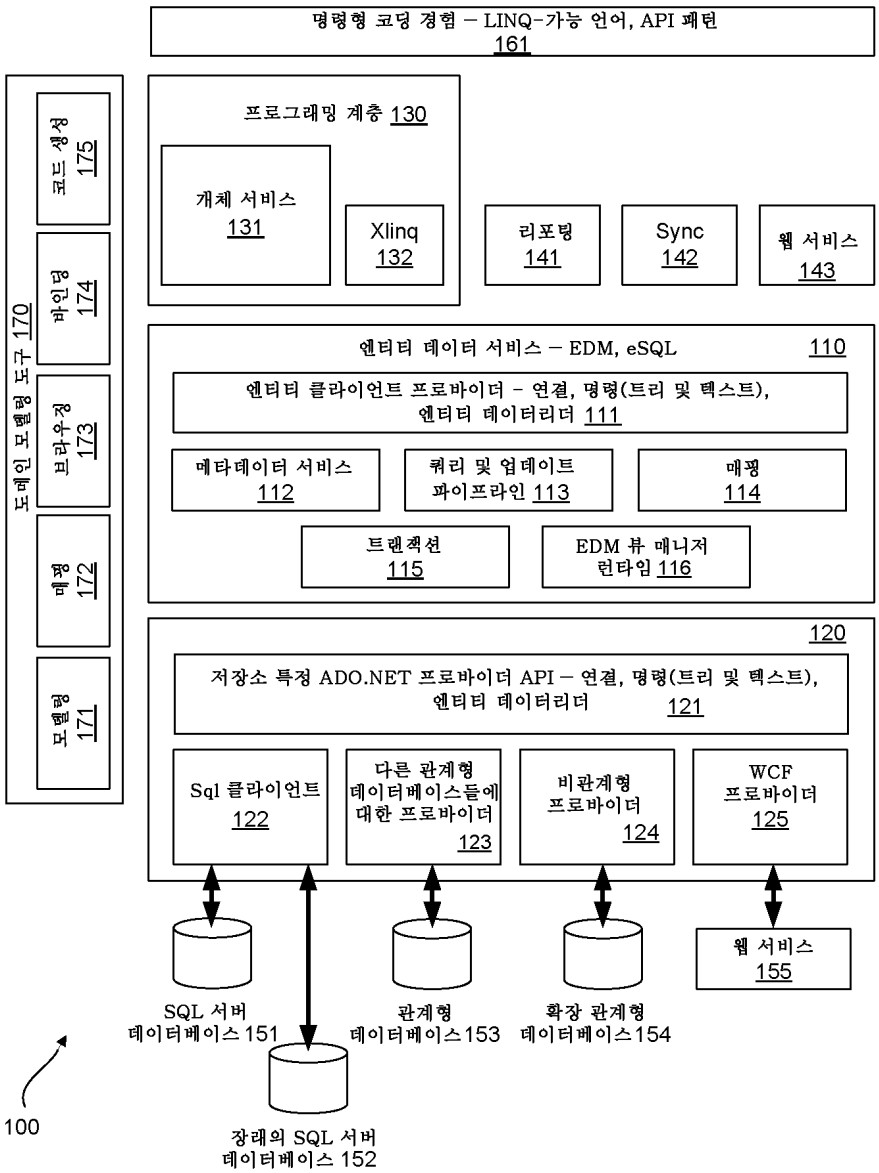
[0027] 도 16은 뷰 컴파일(view compliation) 프로세스에서의 각종 컴포넌트들의 상호작용을 도시한다.

[0028] 도 17은 EDP 쿼리 변환기(EDP Query Translator; EQT) 아키텍처를 도시한다. EQT는 쿼리들을 개체/EDM 공간으로부터 데이터베이스 공간으로 변환하기 위해 메타데이터의 매핑을 이용한다.

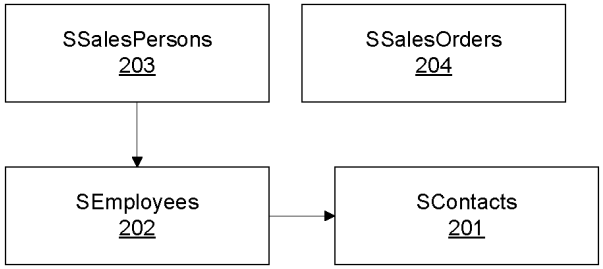
[0029] 도 18은 개체들에 대한 델타 표현들에 의하여 테이블들에 대한 델타 표현을 얻기 위해 각종의 델타 표현들을 구성(compose)하는 것을 도시한다.

도면

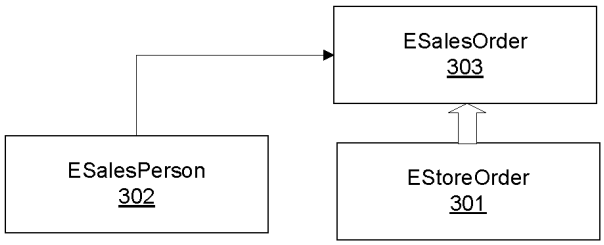
도면1


명령형 코딩 경험 – LINQ-가능 언어, API 패턴
161
도메인 모델링 도구 170
코드 생성 175
바인딩 174
브라우징 173
매핑 172
모델링 171
프로그래밍 계층 130
개체 서비스 131
Xlinq 132
리포팅 141
Sync 142
웹 서비스 143
엔티티 데이터 서비스 – EDM, eSQL 110
엔티티 클라이언트 프로바이더 – 연결, 명령(트리 및 텍스트), 엔티티 데이터리더 111
메타데이터 서비스 112
쿼리 및 업데이트 파이프라인 113
매핑 114
트랜잭션 115
EDM 뷰 매니저 런타임 116
120
저장소 특정 ADO.NET 프로바이더 API – 연결, 명령(트리 및 텍스트), 엔티티 데이터리더 121
Sql 클라이언트 122
다른 관계형 데이터베이스들에 대한 프로바이더 123
비관계형 프로바이더 124
WCF 프로바이더 125
웹 서비스 155
SQL 서버 데이터베이스 151
관계형 데이터베이스 153
확장 관계형 데이터베이스 154
장래의 SQL 서버 데이터베이스 152
100

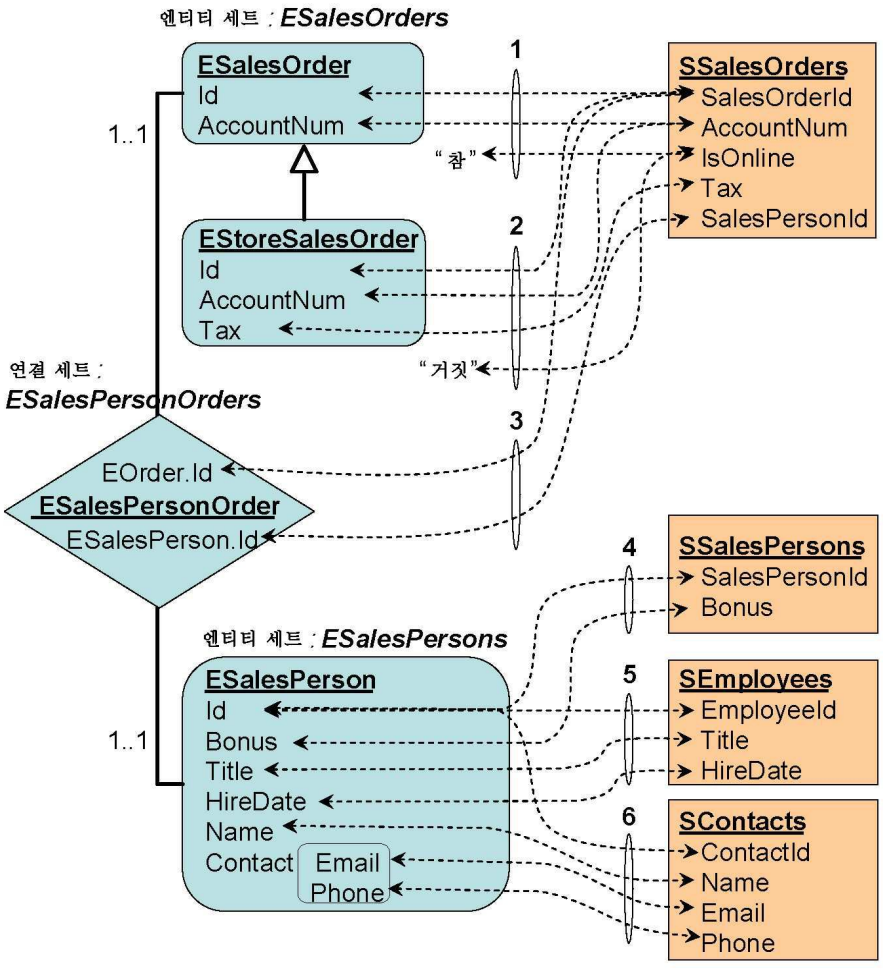
도면2


SSalesPersons 203
SSalesOrders 204
SEmployees 202
SContacts 201

도면3


ESalesOrder
303
ESalesPerson
302
EStoreOrder
301

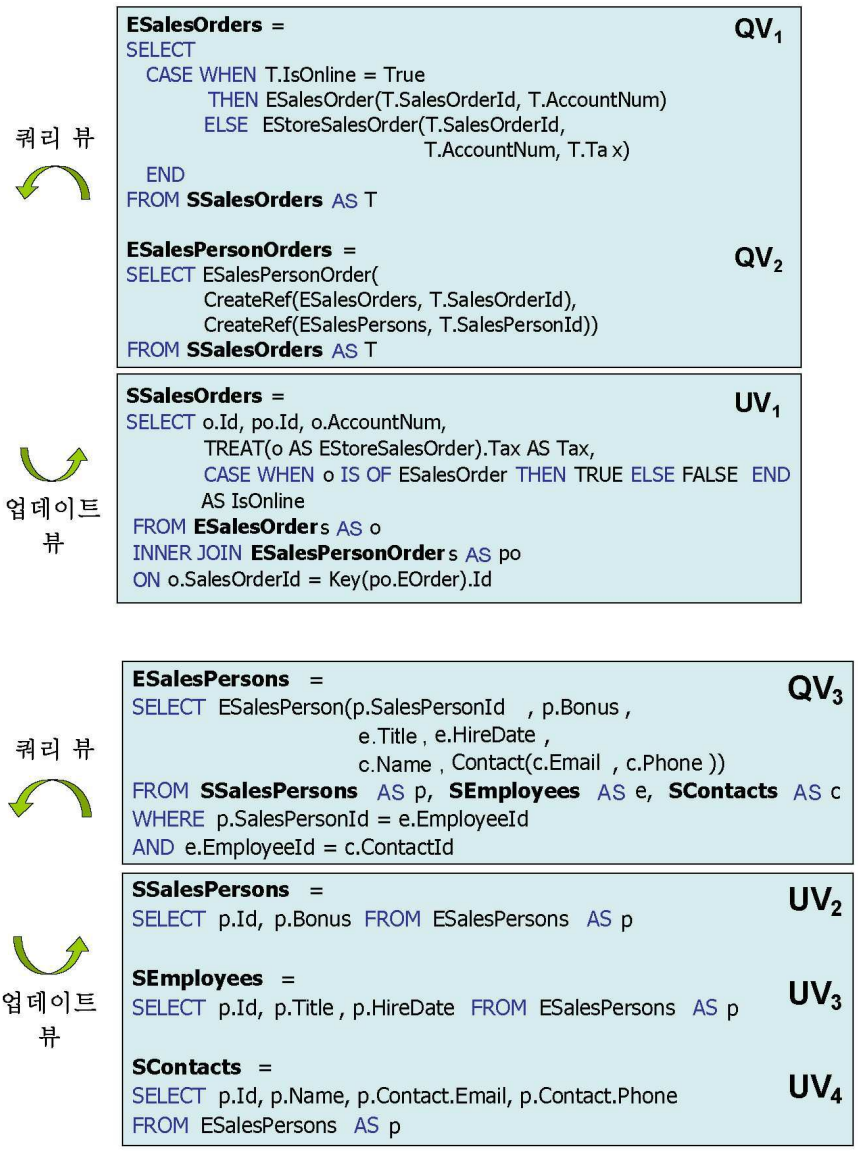
도면4


엔티티 세트 : ESalesOrders
ESalesOrder
Id
AccountNum
1..1
1
"참"
SSalesOrders
SalesOrderId
AccountNum
IsOnline
Tax
SalesPersonId
EStoreSalesOrder
Id
AccountNum
Tax
2
"거짓"
연결 세트 :
ESalesPersonOrders
3
EOrder.Id
ESalesPersonOrder
ESalesPerson.Id
4
SSalesPersons
SalesPersonId
Bonus
엔티티 세트 : ESalesPersons
ESalesPerson
Id
Bonus
Title
HireDate
Name
Contact
Email
Phone
1..1
5
SEmployees
EmployeeId
Title
HireDate
6
SContacts
ContactId
Name
Email
Phone

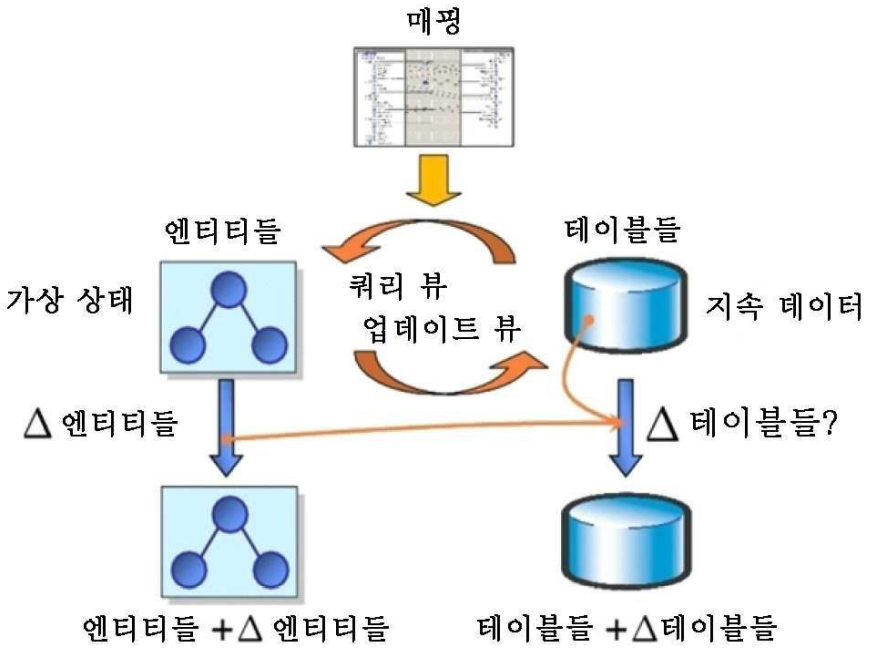
도면5

SELECT o.Id, o.AccountNum FROM **ESalesOrders** o WHERE o IS OF ESalesOrder	=	SELECT SalesOrderId, AccountNum FROM **SSalesOrders** WHERE IsOnline = "true"	1
SELECT o.Id, o.AccountNum, o.Tax FROM **ESalesOrders** o WHERE o IS OF EStoreSalesOrder	=	SELECT SalesOrderId, AccountNum, Tax FROM **SSalesOrders** WHERE IsOnline = "false"	2
SELECT o.EOrder.Id, o.ESalesPerson.Id FROM **ESalesPersonOrders o**	=	SELECT SalesOrderId, SalesPersonId FROM **SSalesOrders**	3
SELECT p.Id, p.Bonus FROM **ESalesPersons** p	=	SELECT SalesPersonId, Bonus FROM **SSalesPersons**	4
SELECT p.Id, p.Title, p.HireDate FROM **ESalesPersons** p	=	SELECT EmployeeId, Title, HireDate FROM **SEmployees**	5
SELECT p.Id, p.Name, p.Contact.Email, p.Contact.Phone p	=	SELECT ContactId, Name, Email, Phone FROM **SContacts**	6

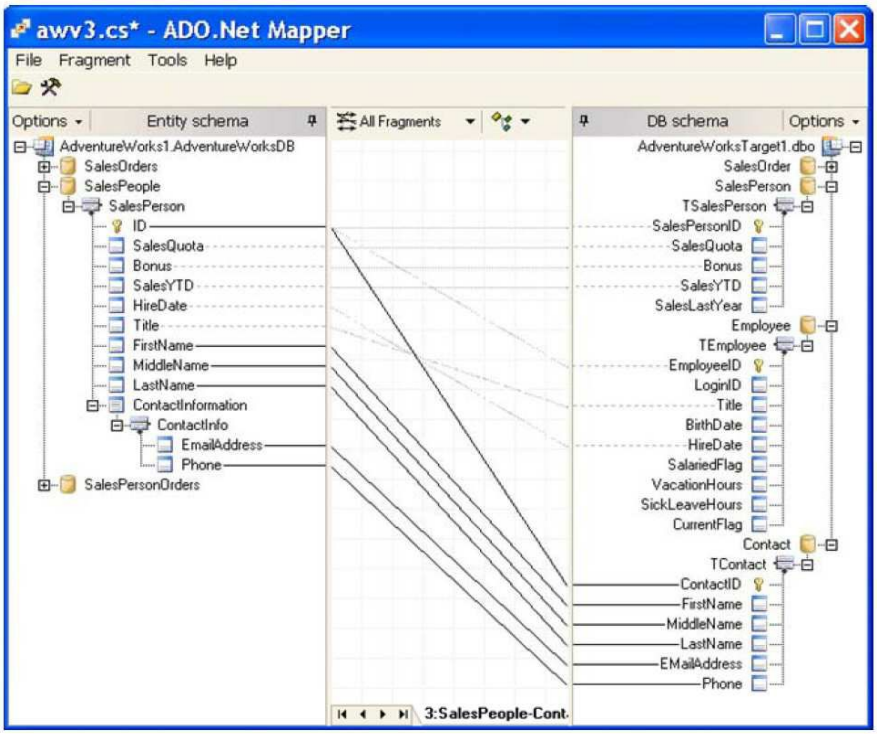
도면6


ESalesOrders = QV1
SELECT
CASE WHEN T.IsOnline = True
THEN ESalesOrder(T.SalesOrderId, T.AccountNum)
ELSE EStoreSalesOrder(T.SalesOrderId,
T.AccountNum, T.Ta x)
END
FROM SSalesOrders AS T
ESalesPersonOrders = QV2
SELECT ESalesPersonOrder(
CreateRef(ESalesOrders, T.SalesOrderId),
CreateRef(ESalesPersons, T.SalesPersonId))
FROM SSalesOrders AS T
쿼리 뷰
SSalesOrders = UV1
SELECT o.Id, po.Id, o.AccountNum,
TREAT(o AS EStoreSalesOrder).Tax AS Tax,
CASE WHEN o IS OF ESalesOrder THEN TRUE ELSE FALSE END
AS IsOnline
FROM ESalesOrders AS o
INNER JOIN ESalesPersonOrders AS po
ON o.SalesOrderId = Key(po.EOrder).Id
업데이트 뷰
ESalesPersons = QV3
SELECT ESalesPerson(p.SalesPersonId , p.Bonus ,
e.Title , e.HireDate ,
c.Name , Contact(c.Email , c.Phone))
FROM SSalesPersons AS p, SEmployees AS e, SContacts AS c
WHERE p.SalesPersonId = e.EmployeeId
AND e.EmployeeId = c.ContactId
쿼리 뷰
SSalesPersons = UV2
SELECT p.Id, p.Bonus FROM ESalesPersons AS p
SEmployees = UV3
SELECT p.Id, p.Title , p.HireDate FROM ESalesPersons AS p
SContacts = UV4
SELECT p.Id, p.Name, p.Contact.Email, p.Contact.Phone
FROM ESalesPersons AS p
업데이트 뷰

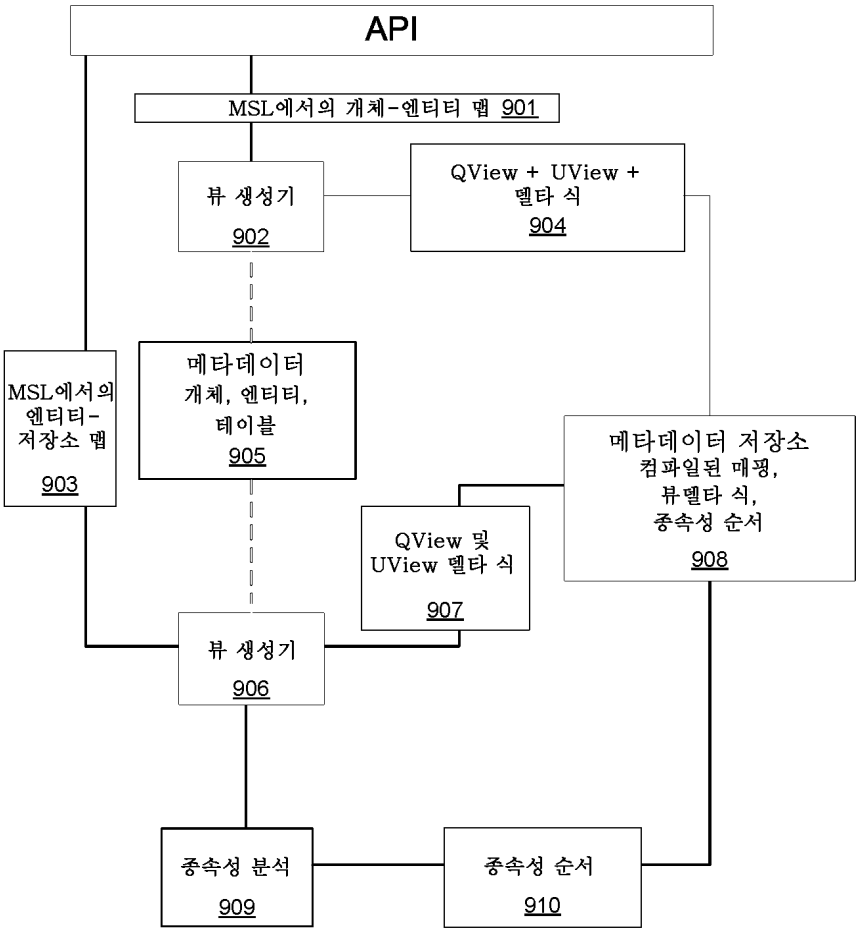
도면7


매핑
엔티티들
테이블들
가상 상태
쿼리 뷰
업데이트 뷰
지속 데이터
Δ 엔티티들
Δ 테이블들?
엔티티들 +Δ 엔티티들
테이블들 +Δ테이블들

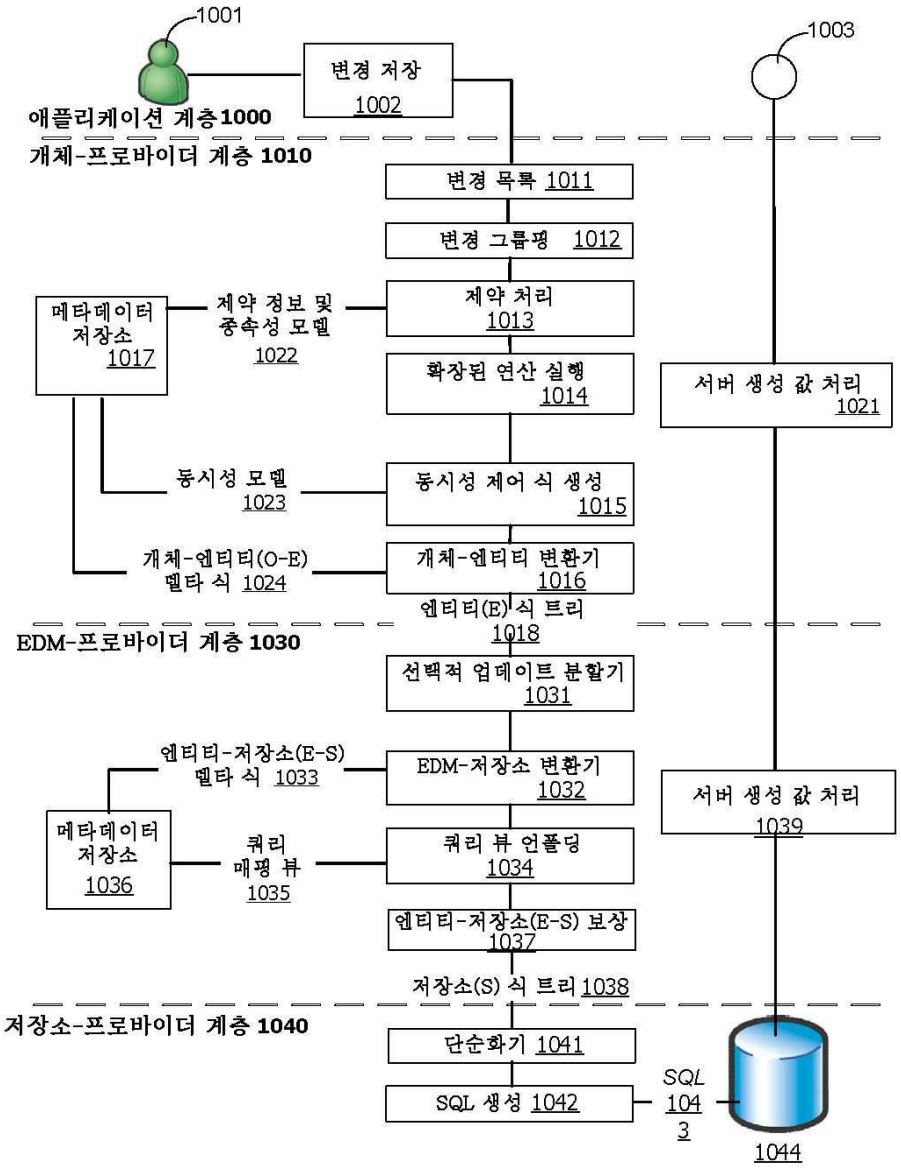
도면8


awv3.cs* - ADO.Net Mapper
File Fragment Tools Help
Options
Entity schema
All Fragments
DB schema
Options
AdventureWorks1.AdventureWorksDB
SalesOrders
SalesPeople
SalesPerson
ID
SalesQuota
Bonus
SalesYTD
HireDate
Title
FirstName
MiddleName
LastName
ContactInformation
ContactInfo
EmailAddress
Phone
SalesPersonOrders
AdventureWorksTarget1.dbo
SalesOrder
SalesPerson
TSalesPerson
SalesPersonID
SalesQuota
Bonus
SalesYTD
SalesLastYear
Employee
TEmployee
EmployeeID
LoginID
Title
BirthDate
HireDate
SalariedFlag
VacationHours
SickLeaveHours
CurrentFlag
Contact
TContact
ContactID
FirstName
MiddleName
LastName
EMailAddress
Phone
3:SalesPeople-Cont

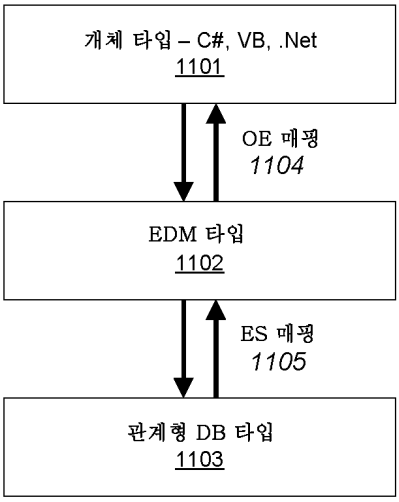
도면9


API
MSL에서의 개체-엔티티 맵 901
뷰 생성기
902
QView + UView +
델타 식
904
MSL에서의
엔티티-
저장소 맵
903
메타데이터
개체, 엔티티,
테이블
905
메타데이터 저장소
컴파일된 매핑,
뷰델타 식,
종속성 순서
908
QView 및
UView 델타 식
907
뷰 생성기
906
종속성 분석
909
종속성 순서
910

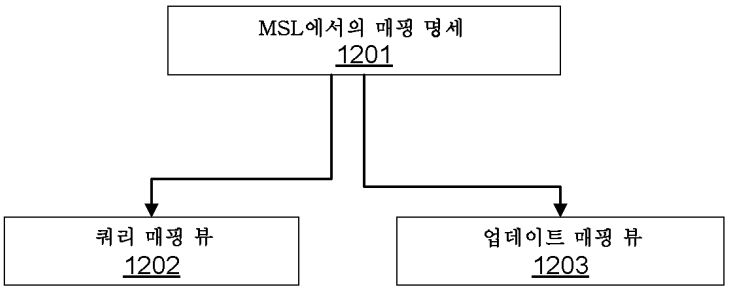
도면10


1001
변경 저장
1002
1003
애플리케이션 계층1000
개체-프로바이더 계층 1010
변경 목록 1011
변경 그룹핑 1012
메타데이터 저장소 1017
제약 정보 및 종속성 모델 1022
제약 처리 1013
확장된 연산 실행 1014
서버 생성 값 처리 1021
동시성 모델 1023
동시성 제어 식 생성 1015
개체-엔티티(O-E) 델타 식 1024
개체-엔티티 변환기 1016
엔티티(E) 식 트리 1018
EDM-프로바이더 계층 1030
선택적 업데이트 분할기 1031
엔티티-저장소(E-S) 델타 식 1033
EDM-저장소 변환기 1032
서버 생성 값 처리 1039
메타데이터 저장소 1036
쿼리 매핑 뷰 1035
쿼리 뷰 언폴딩 1034
엔티티-저장소(E-S) 보상 1037
저장소(S) 식 트리 1038
저장소-프로바이더 계층 1040
단순화기 1041
SQL 생성 1042
SQL 1043
1044

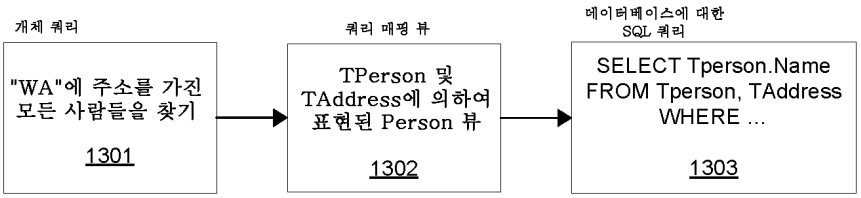
도면11



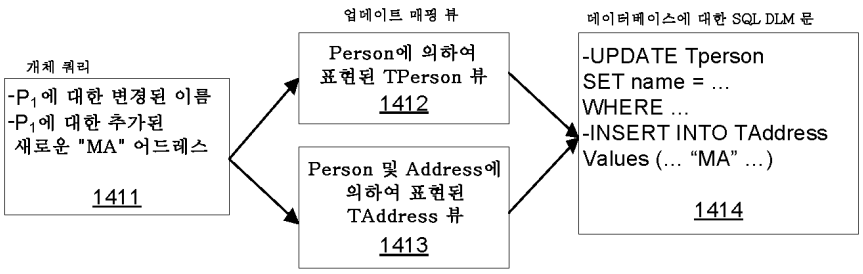
도면12



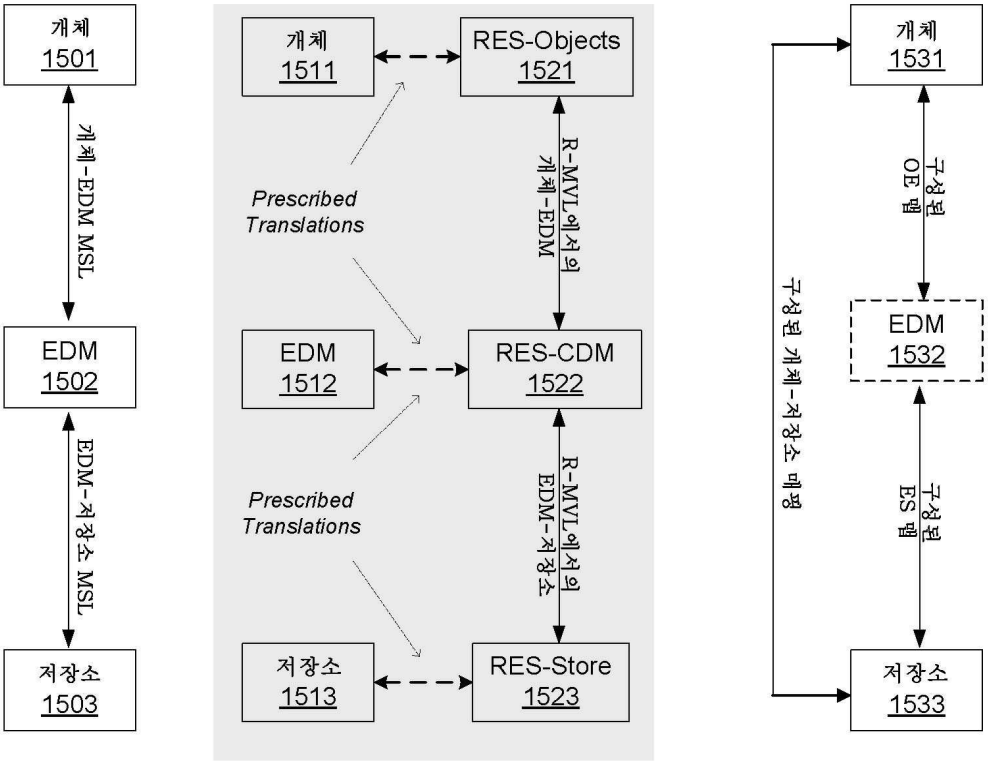
도면13



도면14

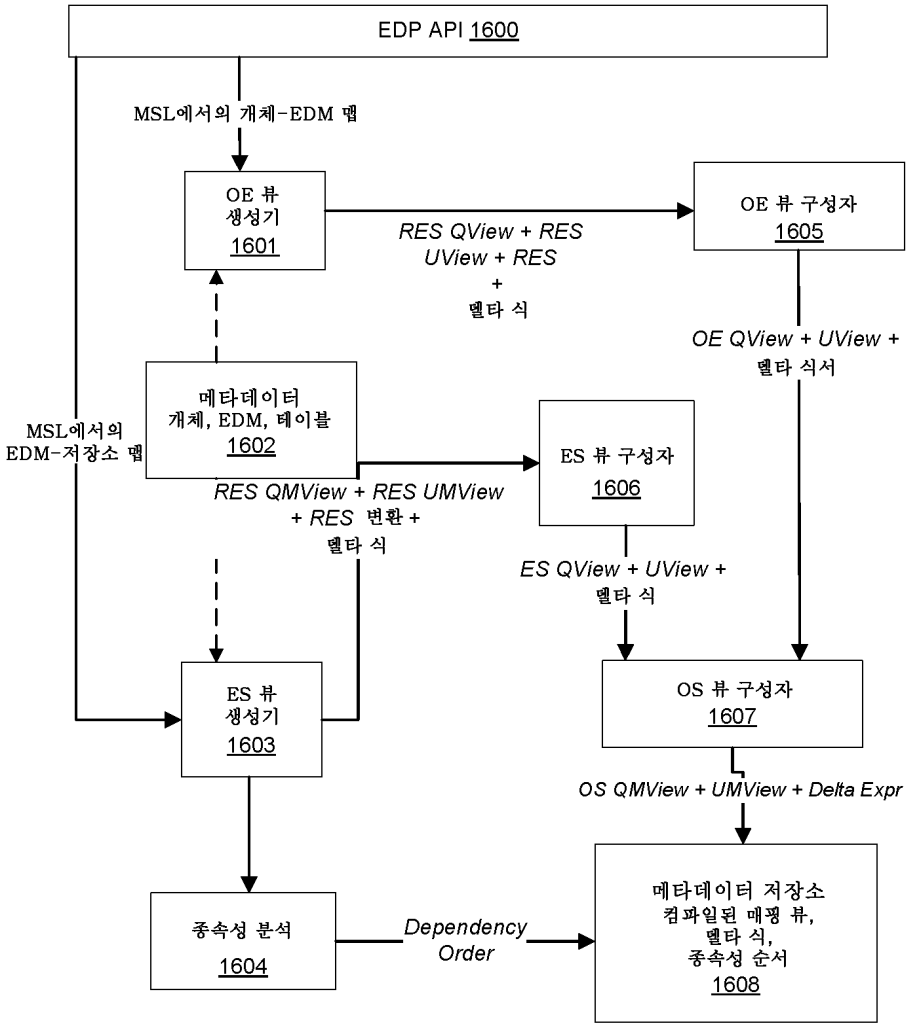


도면15

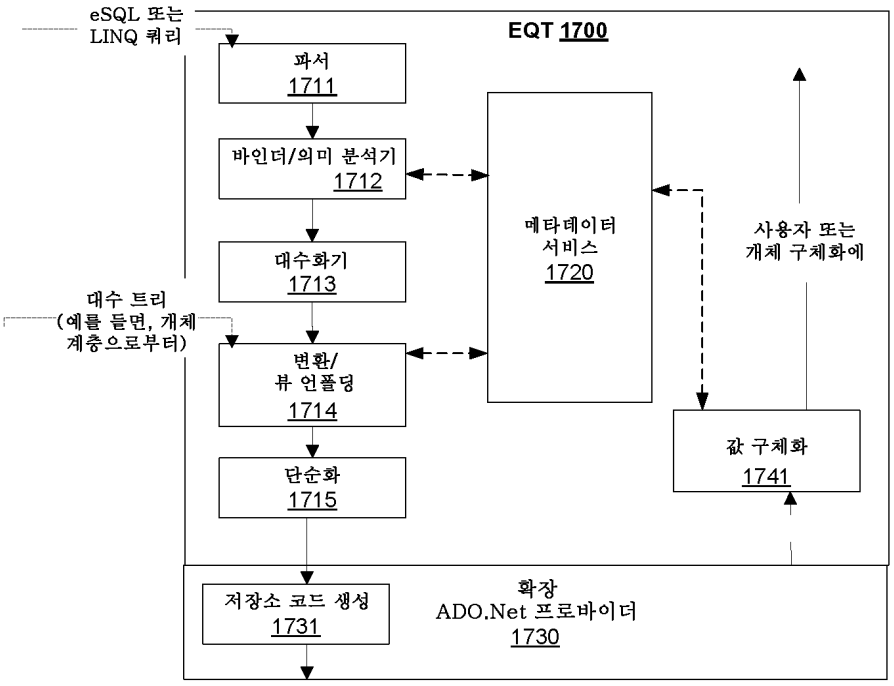


매핑 명세 RES 공간으로의 분해 구성된 매핑

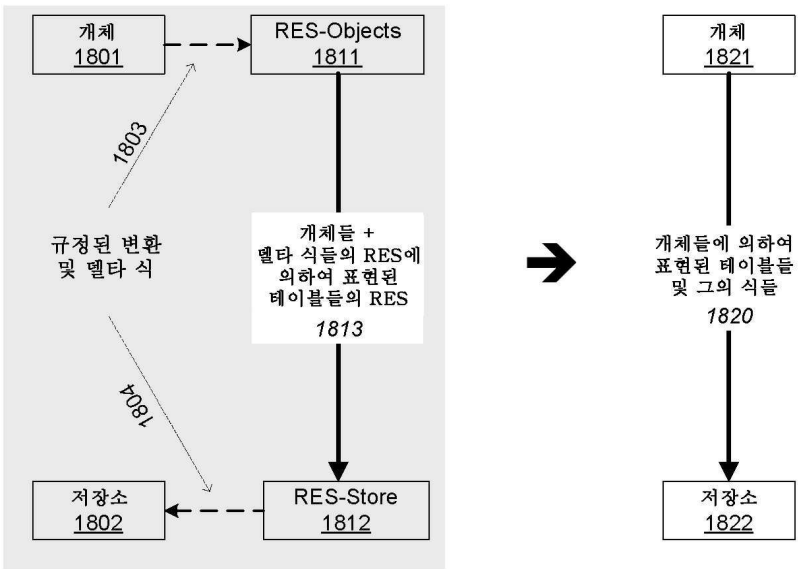
도면16


EDP API 1600
MSL에서의 개체-EDM 맵
OE 뷰 생성기 1601
RES QView + RES UView + RES + 델타 식
OE 뷰 구성자 1605
OE QView + UView + 델타 식서
메타데이터 개체, EDM, 테이블 1602
MSL에서의 EDM-저장소 맵
RES QMView + RES UMView + RES 변환 + 델타 식
ES 뷰 구성자 1606
ES QView + UView + 델타 식
ES 뷰 생성기 1603
OS 뷰 구성자 1607
OS QMView + UMView + Delta Expr
종속성 분석 1604
Dependency Order
메타데이터 저장소 컴파일된 매핑 뷰, 델타 식, 종속성 순서 1608

도면17


eSQL 또는
LINQ 쿼리
EQT 1700
파서
1711
바인더/의미 분석기
1712
대수화기
1713
대수 트리
(예를 들면, 개체
계층으로부터)
변환/
뷰 언폴딩
1714
단순화
1715
메타데이터
서비스
1720
사용자 또는
개체 구체화에
값 구체화
1741
저장소 코드 생성
1731
확장
ADO.Net 프로바이더
1730

도면18


개체
1801
RES-Objects
1811
1803
규정된 변환
및 델타 식
개체들 +
델타 식들의 RES에
의하여 표현된
테이블들의 RES
1813
1804
저장소
1802
RES-Store
1812
개체
1821
개체들에 의하여
표현된 테이블들
및 그의 식들
1820
저장소
1822