(19)대한민국특허청(KR)
(12) 공개특허공보(A)

(51) 。Int. Cl.
G06F 13/00 (2006.01)
G06F 9/00 (2006.01)
G06F 3/00 (2006.01)
G06F 7/00 (2006.01)

(11) 공개번호 10-2006-0113353
(43) 공개일자 2006년11월02일

(21) 출원번호 10-2005-7009283
(22) 출원일자 2005년05월23일
번역문 제출일자 2005년05월23일
(86) 국제출원번호 PCT/US2004/024437
국제출원일자 2004년07월29일
(87) 국제공개번호 WO 2005/024550
국제공개일자 2005년03월17일

(30) 우선권주장 10/646,632 2003년08월21일 미국(US)
10/692,779 2003년10월24일 미국(US)
PCT/US03/26144 2003년08월21일 세계지적재산권기구(WIPO)(WO)

(71) 출원인 마이크로소프트 코포레이션
미국 워싱턴주 (우편번호 : 98052) 레드몬드 원 마이크로소프트 웨이

(72) 발명자 다트, 스코트 이.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이마이크로소프트 코포레이션 내
깁슨, 브래들리 피.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이마이크로소프트 코포레이션 내
에반스, 크리스토퍼 에이.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이마이크로소프트 코포레이션 내
헬리아, 폴 에스.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이마이크로소프트 코포레이션 내
바스칠로, 알렉산더
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이마이크로소프트 코포레이션 내
플래트, 존 씨.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이마이크로소프트 코포레이션 내
글레너, 스티브 씨.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이마이크로소프트 코포레이션 내
발로우, 나타니엘 에이치.
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이마이크로소프트 코포레이션 내
톰슨, 제이. 패트릭
미국 98052 워싱턴주 레드몬드 원 마이크로소프트 웨이마이크로소프트 코포레이션 내

(74) 대리인 주성민
백만기
이중희

심사청구 : 없음

(54) 하드웨어/소프트웨어 인터페이스 시스템에 의해 관리가능한 정보 단위들을 조직하기 위한 디지털 이미지스키마의 구현을 위한 시스템 및 방법

요약

이미지 기반 시스템에서, 이미지들(예를 들면, JPEG, TIFF, 비트맵 등)은 코어 플랫폼 객체들("이미지 아이템들" 또는, 보다 간단히, "이미지들")로서 취급되고, 이 시스템에서 이미지의 확장가능한 표현 - 즉, 이미지의 특징들 및 해당 이미지가 시스템 내의 다른 아이템들(다른 이미지들을 포함하지만 이에 한정되지 않음)에 어떻게 관련되는지 - 을 제공하는 "이미지 스키마"(Image Schema) 내에 존재한다. 이를 위하여, 이미지 스키마는 시스템 내의 이미지들에 대한 속성들(properties), 거동들(behaviors), 및 관계들(relationships)을 정의하고, 이 스키마는 또한 이미지들에 관한 규칙들, 예를 들면 데이터 특정 이미지들이 무엇을 포함해야 하는지, 데이터 특정 이미지들이 선택 사양으로 무엇을 포함할 수 있는지, 특정 이미지들이 어떻게 확장될 수 있는지 등을 정의한다.

대표도

도 7

색인어

하드웨어/소프트웨어 인터페이스 시스템, 이미지 스키마, 아이템, 속성, 관계

명세서

기술분야

<상호 참조>

본 출원은 "SYSTEMS AND METHODS FOR THE IMPLEMENTATION OF A CORE SCHEMA FOR PROVIDING A TOP-LEVEL STRUCTURE FOR ORGANIZING UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM"이라는 제목으로 2003년 8월 21일자로 출원된 미국 특허 출원 번호 10/646,632(대리인 사건 번호 MSFT-1751); 및 2003년 8월 21일자로 출원된 국제 출원 번호 PCT/US03/26144의 이점을 청구하는 2003년 10월 24일자로 출원된 미국 특허 출원 번호 10/692,779(대리인 사건 번호 MSFT-2829)에 대한 우선권을 주장하고, 이들 문헌의 개시 내용 전체가 참고로 본 명세서에 통합된다.

본 출원은 또한 다음의 공히 양도된 출원들에서 개시된 발명들에 대한 요지와 관련이 있고, 이들 출원의 내용들도 참고로 본 명세서에 통합된다: "SYSTEMS AND METHODS FOR REPRESENTING UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM BUT INDEPENDENT OF PHYSICAL REPRESENTATION"이라는 제목으로 2003년 8월 21일자로 출원된 미국 특허 출원 번호 10/647,058(대리인 사건 번호 MSFT-1748); "SYSTEMS AND METHODS FOR SEPARATING UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM FROM THEIR PHYSICAL ORGANIZATION"이라는 제목으로 2003년 8월 21일자로 출원된 미국 특허 출원 번호 10/646,941(대리인 사건 번호 MSFT-1749); "SYSTEMS AND METHODS FOR THE IMPLEMENTATION OF A BASE SCHEMA FOR ORGANIZING UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM"이라는 제목으로 2003년 8월 21일자로 출원

된 미국 특허 출원 번호 10/646,940(대리인 사건 번호 MSFT-1750); "SYSTEMS AND METHOD FOR REPRESENING RELATIONSHIPS BETWEEN UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM"이라는 제목으로 2003년 8월 21일자로 출원된 미국 특허 출원 번호 10/646,645(대리인 사건 번호 MSFT-1752); "SYSTEMS AND METHODS FOR INTERFACING APPLICATION PROGRAMS WITH AN ITEM-BASED STORAGE PLATFORM"이라는 제목으로 2003년 8월 21일자로 출원된 미국 특허 출원 번호 10/646,575(대리인 사건 번호 MSFT-2733); "STORAGE PLATFORM FOR ORGANIZING, SEARCHING, AND SHARING DATA"라는 제목으로 2003년 8월 21일자로 출원된 미국 특허 출원 번호 10/646,646(대리인 사건 번호 MSFT-2734); "SYSTEMS AND METHODS FOR DATA MODELING IN AN ITEM-BASED STORAGE PLATFORM"이라는 제목으로 2003년 8월 21일자로 출원된 미국 특허 출원 번호 10/646,580(대리인 사건 번호 MSFT-2735); "SYSTEMS AND METHODS FOR PROVIDING SYNCHRONIZATION SERVICES FOR UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM"이라는 제목으로 2003년 10월 24일자로 출원된 미국 특허 출원 번호 10/692,515(대리인 사건 번호 MSFT-2844); "SYSTEMS AND METHODS FOR PROVIDING RELATIONAL AND HIERARCHICAL SYNCHRONIZATION SERVICES FOR UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM"이라는 제목으로 2003년 10월 24일자로 출원된 미국 특허 출원 번호 10/692,508(대리인 사건 번호 MSFT-2845); "SYSTEMS AND METHODS FOR THE IMPLEMENTATION OF A SYNCHRONIZATION SCHEMAS FOR UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM"이라는 제목으로 2003년 10월 24일자로 출원된 미국 특허 출원 번호 10/693,362(대리인 사건 번호 MSFT-2846); 및 "SYSTEMS AND METHODS FOR EXTENSIONS AND INHERITANCE FOR UNITS OF INFORMATION MANAGEABLE BY A HARDWARE/SOFTWARE INTERFACE SYSTEM"이라는 제목으로 2003년 10월 24일자로 출원된 미국 특허 출원 번호 10/693,574(대리인 사건 번호 MSFT-2847).

<발명의 분야>

본 발명은 일반적으로 정보 저장 및 검색 분야에 관한 것이고, 보다 구체적으로는, 컴퓨터화된 시스템에서의 서로 다른 타입의 데이터 및 특히 이미지 데이터를 조직하고, 검색하고, 공유하기 위한 액티브 저장 플랫폼(active storage platform)에 관한 것이다.

배경기술

개개의 디스크 용량은 지난 10년간에 걸쳐서 해마다 대략 칠십 퍼센트(70%) 정도로 성장하였다. 무어의 법칙(Moore's law)은 수년에 걸쳐서 일어난 중앙 처리 장치(CPU)의 엄청난 증진(gains)을 정확히 예언하였다. 유선 및 무선 기술들은 엄청난 접속성 및 대역폭을 제공하였다. 현재의 추세가 지속한다고 추정할 때, 수년 내에 보통 수준의 랩탑 컴퓨터는 대략 1 테라바이트(TB)의 저장 용량을 소유하고 수백만의 파일들을 포함할 것이고, 500 기가바이트(GB) 드라이브들이 흔하게 될 것이다.

소비자들은 그들의 컴퓨터를 주로 통신용으로 및 개인 정보(그것이 전통적인 개인 정보 관리 프로그램(PIM : personal information manager) 스타일 데이터이든지 디지털 음악 또는 사진과 같은 미디어이든지간에)를 조직하기 위해 사용한다. 디지털 콘텐츠의 양, 및 미가공 바이트들(raw bytes)을 저장하는 능력은 엄청나게 증가하였지만, 이 데이터를 조직하고 통합하기 위해 소비자들이 이용할 수 있는 방법들은 보조를 맞추지 못하고 있다. 지식 노동자들은 정보를 관리하고 공유하는 데 막대한 양의 시간을 소비하고, 일부 연구에 따르면 지식 노동자들은 그들의 시간의 15-20%를 비생산적인 정보 관련 활동에 소비하는 것으로 추정된다. 다른 연구에 따르면 전형적인 지식 노동자는 하루에 약 2.5 시간을 정보를 검색하는 데 소비하는 것으로 추정된다.

개발자들 및 정보 기술(IT) 부문들은 상당한 양의 시간과 돈을 사람, 장소, 시간, 및 이벤트와 같은 것들을 나타내는 공통의 저장 추상화(storage abstactions)를 위한 그들 자신의 데이터 스토어(data stores)를 구축하는 데 투자한다. 이것은 중복된 작업을 초래할 뿐만 아니라, 공통의 데이터에 대한 공통의 검색 또는 공유를 위한 아무런 메커니즘도 없이 공통의 데이터의 아일랜드들(islands)을 생성하기도 한다. 오늘날 마이크로소프트 윈도즈(Microsoft Windows) 운영 시스템을 실행하는 컴퓨터 상에 얼마나 많은 주소록(address book)이 존재할 수 있는지를 좀 생각해보자. 이메일 클라이언트 및 개인 재정 프로그램과 같은 다수의 애플리케이션들이 개개의 주소록을 유지하고, 각각의 그러한 프로그램이 개별적으로 유지하는 주소록 데이터에 대한 공유가 애플리케이션들 사이에서 거의 없다. 따라서, (Microsoft Money와 같은) 재정 프로그램은 수취인들(payees)에 대한 주소들을 (Microsoft Outlook 내의 것과 같은) 이메일 콘택트 폴더(email contact folder) 내에 유지된 주소들과 공유하지 않는다. 실제로, 많은 사용자들이 다수의 디바이스를 갖고 있고 논리적으로 그들의 개인 데이터를

그들 자신들 사이에서 이동 전화(cell phones)에서 MSN 및 AOL과 같은 상업적 서비스까지를 포함하는 다양한 부가적인 소스들을 통하여 동기화시켜야 하지만, 그럼에도 공유 문서들의 제휴는 주로 이메일 메시지에 문서들을 첨부함으로써, 즉 수동으로 비효율적으로 달성된다.

이 제휴의 결여의 한 가지 이유는 컴퓨터 시스템들에서의 정보의 조직에 대한 전통적인 방법들은 파일-폴더-및-디렉토리-기반 시스템("파일 시스템")을 이용하여, 파일들을 저장하기 위해 사용되는 저장 매체의 물리적 조직의 추상화에 기초하여 복수의 파일들을 폴더들의 디렉토리 계층 구성으로 조직하는 것에 집중하였다는 점이다. 1960년대 중에 개발된 Multics 운영 시스템은 파일, 폴더, 및 디렉토리를 이용하여 운영 시스템 레벨에서 저장 가능한 데이터 단위들을 관리하는 것을 개척한 공로가 인정될 수 있다. 구체적으로, Multics는 파일들의 물리적 주소가 사용자(애플리케이션 및 최종 사용자)에게 투명하지 않았던 파일들의 계층 구성 내에서 기호 주소(symbolic addresses)를 사용하였다(그로써 파일 경로(file path)의 아이디어를 도입하였다). 이 파일 시스템은 임의의 개개의 파일의 파일 포맷과 완전히 무관하였고, 파일들 사이의 관계들은 운영 시스템 레벨에서 무관한 것으로 간주되었다(즉, 계층 구성 내의 파일의 위치 이외). Multics의 출현 이후로, 저장 가능한 데이터는 운영 시스템 레벨에서 파일, 폴더, 및 디렉토리로 조직되었다. 이들 파일들은 일반적으로 파일 시스템에 의해 유지되는 특별한 파일 내에 구현된 파일 계층 구성 자체("디렉토리")를 포함한다. 이 디렉토리는 또한 디렉토리 내의 다른 파일들 모두에 대응하는 엔트리들의 리스트 및 (여기에서 폴더라고 불리는) 계층 구성 내의 그러한 파일들의 노드 위치를 유지한다. 이러한 것이 대략 40년 동안의 기술적 현상이었다.

그러나, 파일 시스템은 컴퓨터의 물리적 저장 시스템에 상주하는 정보의 합리적인 표현을 제공하기는 하지만, 그럼에도 파일 시스템은 그 물리적 저장 시스템의 추상화이므로, 파일들의 이용은 사용자가 조작 처리하는 것(컨텍스트, 특징, 및 다른 단위들에 대한 관계를 갖는 단위들)과 운영 시스템이 제공하는 것(파일, 폴더, 및 디렉토리) 사이에 간접화 레벨(번역)(a level of indirection(interpretation))을 필요로 한다. 따라서, 사용자(애플리케이션 및/또는 최종 사용자)는 정보 단위들을 파일 시스템 구조로 강제화할 수 밖에 없으며, 그렇게 하는 것이 비효율적이거나, 불일치하거나, 또는 다르게는 바람직하지 않은 경우라 할지라도 어쩔 수 없다. 더욱이, 기존의 파일 시스템들은 개개의 파일들에 저장된 데이터의 구조에 관하여 거의 알지 못하고, 이 때문에, 그 정보의 대부분은 그것들을 기록한 애플리케이션들에게만 액세스될 수 있는(및 이해될 수 있는) 파일들 내에 감금된(locked up) 상태로 남는다. 따라서, 정보에 대한 개략적인 기술, 및 정보를 관리하기 위한 메커니즘의 이러한 결여는 개개의 사일로들(silos) 사이에 데이터 공유가 거의 없는 데이터의 사일로들의 생성으로 이어진다. 예를 들면, 많은 퍼스널 컴퓨터(PC) 사용자들은 어떤 레벨에서 그들이 상호 작용하는 사람들에 관한 정보를 포함하는 5개 이상의 별개의 스토어들 - 예를 들면, 아웃룩 콘텍트(Outlook Contacts), 온라인 계정 주소, 윈도즈 주소록(Windows Address Book), Quicken Payees, 및 인스턴트 메시징(IM) 버디 리스트 - 를 갖고 있다. 왜냐하면 파일들을 조직하는 것이 이들 PC 사용자들에게는 중요한 도전 과제를 제시하기 때문이다. 대부분의 기존의 파일 시스템들은 파일들 및 폴더들을 조직하기 위해 네스트된 폴더 메타포(nested folder metaphor)를 이용하기 때문에, 파일들의 수가 증가함에 따라서 유연성이 있고 효율적인 조직 체계(organization scheme)를 유지하기 위해 필요한 노력이 상당히 위협을 받게 된다. 그러한 경우에는, 단일 파일의 다수의 분류를 갖는 것이 매우 유익하겠지만, 기존의 파일 시스템에서 하드 또는 소프트 링크들을 이용하는 것은 성가시고 유지하기가 곤란하다.

파일 시스템들의 단점을 해결하려는 몇몇 시도들이 과거에 행해졌지만, 성공하지 못했다. 이들 이전의 시도들 중 일부는 콘텐츠 어드레싱 가능한 메모리(content addressable memory)를 사용하여 물리적 주소에 의하기보다는 콘텐츠에 의해 데이터가 액세스될 수 있게 하는 메커니즘을 제공하는 것을 포함하였다. 그러나, 이러한 노력들은 성공하지 못한 것으로 판명되었다. 왜냐하면, 콘텐츠 어드레싱 가능한 메모리는 캐시 및 메모리 관리 유닛과 같은 디바이스들에 의한 소규모 사용에 대해서는 유용한 것으로 판명되었지만, 물리적 저장 매체와 같은 디바이스들에 의한 대규모 사용은 아직은 여러가지 이유로 가능하지 않고, 따라서 그러한 해법은 전혀 존재하지 않기 때문이다. 객체 지향 데이터베이스(OODB) 시스템을 이용한 다른 시도들이 행해졌지만, 이 시도들은, 강력한 데이터베이스 특성과 양호한 비파일 표현(non-file representations)을 특징으로 하기 때문에, 파일 표현(file representations)을 다루는 데에는 효과적이지 않았고 하드웨어/소프트웨어 인터페이스 시스템 레벨에서 파일 및 폴더 기반 계층형 구조의 속도, 효율성, 및 단순성을 복제할 수 없었다. 스몰토크(SmallTalk)(및 다른 파생물들)를 이용하려고 한 것들과 같은 다른 노력들은 파일 및 비파일 표현들을 다루는 데에는 상당히 효과적인 것으로 판명되었지만 여러가지 데이터 파일들 사이에 존재하는 관계들을 효율적으로 조직하고 이용하기 위해 필요한 데이터베이스 특징은 결여하였고, 따라서 그러한 시스템들의 전체 효율성은 수용하기 어려운 것이었다. BeOS를 이용하려는 또 다른 시도들(및 다른 그러한 운영 시스템 연구)은, 얼마간의 필요한 데이터베이스 특징들을 제공하면서 파일들을 적절하게 표현할 수 있음에도 불구하고, 비파일 표현들을 다루는 데는 부적당한 것으로 판명되어, 전통적인 파일 시스템들과 동일한 주요 단점이 있다.

데이터베이스 기술은 유사한 도전 과제가 존재하는 또 다른 기술 분야이다. 예를 들면, 관계 데이터베이스(relational database) 모델은 커다란 상업적 성공을 거두었지만, 실제로 독립적 소프트웨어 벤더들(ISV: independent software vendors)은 일반적으로 (마이크로소프트 SQL 서버와 같은) 관계 데이터베이스 소프트웨어 제품들에서 이용 가능한 기능

의 작은 부분을 수행한다. 그 대신에, 그러한 제품과의 애플리케이션의 상호 작용의 대부분은 단순한 "gets" 및 "puts"의 형태로 되어 있다. 플랫폼 또는 데이터베이스 불가지론적인(platform or database agnostic) 것과 같은, 이에 대한 다수의 선뜻 명백한 이유들이 있지만, 종종 간과되는 하나의 중요한 이유는 데이터베이스가 반드시 주요 비즈니스 애플리케이션 벤더가 실제로 필요로 하는 정확한 추상화(abstractions)를 제공하지는 않는다는 사실이다. 예를 들면, 실재 세계는 "고객"(customers) 또는 "주문"(orders)과 같은 "아이템"의 개념을 (아이템들 자체로서 주문의 임베드된(embedded) "라인 아이템"(line items)과 함께) 갖고 있지만, 관계 데이터베이스는 표(tables) 및 행(rows)의 관점에서만 거론한다. 따라서, 애플리케이션은 아이템 레벨에서 일관성(consistency), 잠금(locking), 보안, 및/또는 트리거(triggers)의 특징들을 갖기를 원할 수 있지만(몇 가지를 거론하면), 일반적으로 데이터베이스는 이들 특징들을 표/행 레벨에서만 제공한다. 이것은 각각의 아이템이 데이터베이스 내의 어떤 표 내의 단일 행에 매핑되는 경우에는 유효할 수 있지만, 다수의 라인 아이템들을 갖는 주문의 경우에는 하나의 아이템이 실제로 다수의 표에 매핑되는 이유들이 있을 수 있고, 그런 경우라면, 단순한 관계 데이터베이스 시스템은 올바른 추상화들을 완전히 제공하지는 않는다. 따라서, 애플리케이션은 이들 기본적인 추상화를 제공하기 위해 데이터베이스의 상부에 논리를 구축하여야 한다. 바꾸어 말하면, 기본적인 관계 모델은 그 위에서 상위 레벨 애플리케이션들이 쉽게 개발될 수 있는 데이터 저장을 위한 충분한 플랫폼을 제공하지 않는다. 왜냐하면 기본적인 관계 모델은 애플리케이션과 저장 시스템 사이에 간접화 레벨 - 여기에서 데이터의 의미 구조는 어떤 경우에 애플리케이션에서 가시화될 수 있을 뿐이다 - 을 필요로 하기 때문이다. 어떤 데이터베이스 벤더들은 그들의 제품에 상위 레벨 기능을 구축하고 있지만(예컨대 객체 관계 능력(object relational capabilities), 새로운 조직 모델(organizational models) 등을 제공하는 것과 같은), 아직은 아무도 요구되는 일종의 포괄적인 해법(comprehensive solution)을 제공하지 못했다. 여기에서 진정으로 포괄적인 해법은 유용한 도메인 추상화(예컨대 "Persons"(사람), "Locations"(장소), "Events"(이벤트) 등)에 대해 유용한 데이터 모델 추상화(예컨대 "Items"(아이템), "Extensions"(확장), "Relationships"(관계) 등)을 제공하는 것이다).

기존의 데이터 저장 및 데이터베이스 기술에서의 전술한 결점들을 고려하여, 컴퓨터 시스템에서의 모든 타입의 데이터를 조직하고, 검색하고, 공유하는 개선된 능력을 제공하는 새로운 저장 플랫폼 - 기존의 파일 시스템 및 데이터베이스 시스템을 넘어서 데이터 플랫폼을 확장하고 넓히며, 모든 타입의 데이터에 대한 스토어가 되도록 설계되어 있는 저장 플랫폼 - 이 요구되고 있다. 본 명세서의 앞에서 참고로 통합된 관련 발명들은 이 요구를 만족시킨다.

그러나, 이미지(사진, 디지털 이미지 등)의 저장은 표준화되어 있지 않고 플랫폼들 및 애플리케이션들에 걸쳐서 일반화되어 있지 않다. 애플리케이션들은 특정 이미지 포맷(예컨대, JPEG)에 맞춤화된 애플리케이션 프로그래밍 인터페이스(API)들을 포함할 수 있지만, 그러한 애플리케이션들의 개발자들은 그 포맷을 알고 있어야 하고, 맞춤화된 API를 포함하고, 상기 포맷과 상호 운용(interoperate)하기 위해 필요한 임의의 변환들을 수행해야 한다. 이 기술 분야에서 누락되어 있는 것은 컴퓨터 시스템에서의 모든 이미지 객체들에 대한 공통의 스키마(또는 스키마들의 세트)이고, 본 발명은, 본 명세서의 앞에서 참고로 통합된 관련 발명들과 함께, 이 특정 요구를 만족시킨다.

<개요>

이하의 개요는 본 명세서의 앞에서 참고로 통합된 관련 발명들("관련 발명들")의 컨텍스트에서 기술된 발명의 다양한 양태들에 대한 개관을 제공한다. 이 개요는 본 발명의 모든 중요한 양태들에 대한 남김 없는 설명을 제공하기 위한 것도 아니고, 본 발명의 범위를 정의하기 위한 것도 아니다. 오히려, 이 개요는 다음에 오는 상세한 설명 및 도면들에 대한 서론의 역할을 하도록 의도되어 있다.

본 발명 뿐만 아니라 관련 발명들은 총괄적으로 데이터를 조직하고, 검색하고, 공유하기 위한 저장 플랫폼에 관한 것이다. 본 발명의 저장 플랫폼은 기존의 파일 시스템 및 데이터베이스 시스템을 넘어서 데이터 저장의 개념을 확장하고 넓히며, 구조화되거나, 비구조화되거나, 또는 반구조화된(semi-structured) 데이터를 포함하는 모든 타입의 데이터에 대한 스토어가 되도록 설계되어 있다.

본 발명의 저장 플랫폼은 데이터베이스 엔진 상에 구현된 데이터 스토어를 포함한다. 데이터베이스 엔진은 객체 관계 확장들(object relational extensions)을 갖는 관계 데이터베이스 엔진을 포함한다. 데이터 스토어는 데이터의 조직, 검색, 공유, 동기화, 및 보안을 지원하는 데이터 모델을 구현한다. 데이터의 구체적인 타입들은 스키마들에서 기술되고, 플랫폼은 데이터의 새로운 타입들(본질적으로 스키마들에 의해 제공된 기본적인 타입들의 서브타입들(subtypes))을 정의하도록 스키마들의 세트를 확장하는 메커니즘을 제공한다. 동기화 능력은 사용자들 또는 시스템들 사이에서의 데이터의 공유를 용이하게 한다. 기존의 파일 시스템들과의 데이터 스토어의 상호 운용성을 허용하면서도 그러한 전통적인 파일 시스템들의 제한이 없는 파일-시스템 같은 능력들(file-system-like capabilities)이 제공된다. 변경 추적 메커니즘은 데이터 스토어에 대한 변경들을 추적하는 능력을 제공한다. 저장 플랫폼은 애플리케이션들이 저장 플랫폼의 전술한 능력들 모두에 액세스하고 또한 스키마들에서 기술된 데이터에 액세스할 수 있게 하는 애플리케이션 프로그램 인터페이스들의 세트를 더 포함한다.

데이터 스토어에 의해 구현되는 데이터 모델은 아이템(items), 엘리먼트(elements), 및 관계(relationships)에 의하여 데이터 저장의 단위들을 정의한다. 아이템은 데이터 스토어에 저장 가능한 데이터의 단위이고 하나 이상의 엘리먼트 및 관계를 포함할 수 있다. 엘리먼트는 하나 이상의 필드들(본 명세서에서는 속성(property)이라고도 함)을 포함하는 타입의 인스턴스이다. 관계는 2개의 아이템들 간의 링크이다. (본 명세서에서 사용될 때, 이들 및 다른 특정한 용어들은 아주 밀접하게 사용되는 다른 용어들과 분리하기 위하여 대문자로 시작될 수 있다. 그러나, 대문자로 시작된 용어, 예를 들면 "Item"과 대문자로 시작되지 않을 때의 동일한 용어, 예를 들면 "item"을 구별하려는 아무런 의도도 없고, 그러한 구별이 가정되거나 암시되어서는 안 된다.)

컴퓨터 시스템은 복수의 아이템들 - 각각의 아이템은 하드웨어/소프트웨어 인터페이스 시스템에 의해 조작 처리될 수 있는 개별 저장 가능한 정보 단위(discrete storable unit of information)를 구성함 - 과; 상기 아이템들에 대한 조직 구조(organizational structure)를 구성하는 복수의 아이템 폴더들과; 복수의 아이템들을 조작 처리하기 위한 하드웨어/소프트웨어 인터페이스 시스템을 포함하고, 각각의 아이템은 적어도 하나의 아이템 폴더에 속하고 2 이상의 아이템 폴더에 속할 수 있다.

아이템 또는 아이템의 속성 값들의 일부는 지속성 스토어(persistent store)로부터 도출되는 것에 대립되는 것으로서 동적으로 계산될 수 있다. 바꾸어 말하면, 하드웨어/소프트웨어 인터페이스 시스템은 아이템이 저장될 것을 요구하지 않고, 아이템들의 현재 세트를 열거(enumerate)하는 능력 또는 저장 플랫폼의 그것의 식별자(이에 대해서는 애플리케이션 프로그래밍 인터페이스, 즉 API를 설명하는 섹션들에서 보다 상세히 설명됨)가 주어질 경우 아이템을 검색하는 능력과 같은 어떤 동작들이 지원된다 - 예를 들면, 아이템은 이동 전화(cell phone)의 현재 위치이거나 또는 온도 센서 상의 온도 수치일 수 있다. 하드웨어/소프트웨어 인터페이스 시스템은 복수의 아이템들을 조작 처리할 수 있고, 하드웨어/소프트웨어 인터페이스 시스템에 의해 관리되는 복수의 관계들(Relationships)에 의해 상호 연결된 아이템들을 더 포함할 수 있다.

컴퓨터 시스템을 위한 하드웨어/소프트웨어 인터페이스 시스템은 상기 하드웨어/소프트웨어 인터페이스 시스템이 이해하고 미리 정해지고 예측 가능한 방법으로 직접 처리할 수 있는 코어 아이템들(core Items)의 세트를 정의하는 코어 스키마를 더 포함한다. 복수의 아이템들을 조작 처리하기 위하여, 컴퓨터 시스템은 상기 아이템들을 복수의 관계들로 상호 연결하고 상기 관계들을 하드웨어/소프트웨어 인터페이스 시스템 레벨에서 관리한다.

저장 플랫폼의 API는 저장 플랫폼 스키마들의 세트에서 정의된 각각의 아이템, 아이템 확장, 및 관계에 대한 데이터 클래스들을 제공한다. 또한, 애플리케이션 프로그래밍 인터페이스는 상기 데이터 클래스들에 대한 거동들(behaviors)의 공통의 세트를 정의하고, 상기 데이터 클래스들과 함께, 저장 플랫폼 API에 대한 기본적인 프로그래밍 모델을 제공하는 프레임워크 클래스들의 세트를 제공한다. 저장 플랫폼 API는 하위(underlying) 데이터베이스 엔진의 쿼리 언어(query language)의 상세로부터 애플리케이션 프로그래머를 격리시키는 방식으로, 애플리케이션 프로그래머들이 데이터 스토어 내의 아이템들의 갖가지 속성들에 기초하여 쿼리들을 형성할 수 있게 하는 간략화된 쿼리 모델을 제공한다. 저장 플랫폼 API는 또한 애플리케이션 프로그램에 의해 행해진 아이템에 대한 변경들을 수집하고 그 후 그것들을 데이터 스토어가 구현되는 데이터베이스 엔진(또는 임의의 종류의 저장 엔진)에 의해 요구되는 정확한 업데이트들로 조직한다. 이에 따라 애플리케이션 프로그래머들이 메모리 내의 아이템에 대한 변경들을 행할 수 있게 되고, 데이터 스토어 업데이트들의 복잡성은 API에게 남겨진다.

본 발명의 저장 플랫폼은, 그것의 공통의 저장 토대(storage foundation) 및 조직적으로 배열된 데이터(schematized data)를 통하여, 소비자, 지식 노동자 및 기업에게 보다 효율적인 애플리케이션 개발을 가능케 한다. 그것은 그것의 데이터 모델에서의 고유한 능력들을 이용할 수 있게 할 뿐만 아니라, 기존의 파일 시스템 및 데이터베이스 액세스 방법들을 포용하고 확장하는 풍부하고 확장성이 있는 애플리케이션 프로그래밍 인터페이스를 제공한다.

(상세한 설명의 섹션 Ⅱ에서 상세히 설명되는) 상호 관련 발명들의 이러한 중요한 구조의 관점 내에서, 본 발명은 특히 (상세한 설명의 섹션 Ⅲ에서 상세히 설명되는) 컴퓨터 시스템에서의 모든 이미지 객체들(이미지 아이템들)에 대한 공통의 스키마에 관한 것이다. 본 발명의 다른 특징들 및 이점들은 본 발명에 대한 이하의 상세한 설명 및 첨부 도면들로부터 명백해질 것이다.

도면의 간단한 설명

전술한 개요는 물론, 본 발명에 대한 이하의 상세한 설명은, 첨부된 도면들과 함께 읽으면 더 잘 이해된다. 본 발명을 예시 설명하기 위하여, 본 발명의 갖가지 양태들에 대한 예시적인 실시예들이 도면들에 도시되어 있지만, 본 발명은 개시된 특정한 방법들 및 수단들에 한정되지 않는다. 도면들에서,

도 1은 본 발명의 양태들이 통합될 수 있는 컴퓨터 시스템을 나타내는 블록도.

도 2는 3개의 컴포넌트 그룹들: 하드웨어 컴포넌트, 하드웨어/소프트웨어 인터페이스 시스템 컴포넌트, 및 애플리케이션 프로그램 컴포넌트로 나누어진 컴퓨터 시스템을 예시하는 블록도.

도 2A는 파일 기반 운영 시스템(file-based operating system)에서의 디렉토리 내의 폴더들 내에 그룹화된 파일들에 대한 전통적인 트리 기반 계층형 구조를 예시하는 도면.

도 3은 저장 플랫폼을 예시하는 블록도.

도 4는 아이템, 아이템 폴더, 및 카테고리 간의 구조적 관계를 예시하는 도면.

도 5A는 아이템의 구조를 예시하는 블록도.

도 5B는 도 5A의 아이템의 복합 속성 타입들을 예시하는 블록도.

도 5C는 "Location" 아이템을 예시하는 블록도 - 그것의 복합 타입들이 더 설명(명시적으로 리스트)되어 있음 - .

도 6A는 베이스 스키마(Base Schema)에서 발견된 아이템의 서브타입(subtype)으로서 아이템을 예시하는 도면.

도 6B는 도 6A의 서브타입 아이템을 예시하는 블록도 - (그것의 순간 속성들(immediate properties) 외에) 그것의 상속 타입들(inherited types)이 명시적으로 리스트되어 있음 - .

도 7은 2개의 상위 레벨 클래스 타입인 Item 및 PropertyBase, 및 그들로부터 도출된 부가적인 베이스 스키마 타입들을 포함하는 베이스 스키마를 예시하는 블록도.

도 8A는 코어 스키마(Core Schema) 내의 아이템들을 예시하는 블록도.

도 8B는 코어 스키마 내의 속성 타입들을 예시하는 블록도.

도 9는 아이템 폴더(Item Folder), 그것의 멤버 아이템들, 및 아이템 폴더와 그것의 멤버 아이템들 간의 상호 연결 관계들(Relationships)을 예시하는 블록도.

도 10은 카테고리(Category)(이것은 또한 아이템 자체임), 그것의 멤버 아이템들, 및 카테고리와 그것의 멤버 아이템들 간의 상호 연결 관계들을 예시하는 블록도.

도 11은 저장 플랫폼의 데이터 모델의 참조 타입 계층 구성을 예시하는 도면.

도 12는 관계들이 어떻게 분류되는지를 예시하는 도면.

도 13은 통지 메커니즘을 예시하는 도면.

도 14는 2개의 트랜잭션들이 양쪽 모두 새로운 레코드를 동일한 B-트리에 삽입하는 예를 예시하는 도면.

도 15는 데이터 변경 검출 프로세스를 예시하는 도면.

도 16은 예시적인 디렉토리 트리를 예시하는 도면.

도 17은 디렉토리 기반 파일 시스템의 기존의 폴더가 저장 플랫폼 데이터 스토어 내로 이동되는 예를 도시하는 도면.

도 18은 컨테인먼트 폴더(Containment Folders)의 개념을 예시하는 도면.

도 19는 저장 플랫폼 API 스택의 기본적인 아키텍처를 예시하는 도면.

도 20은 저장 플랫폼 API의 갖가지 컴포넌트들을 개략적으로 나타내는 도면.

도 21A는 예시적인 콘택트 아이템(Contacts Item) 스키마에 대한 그림 표현.

도 21B는 도 21A의 예시적인 콘택트 아이템 스키마의 엘리먼트들에 대한 그림 표현.

도 22는 저장 플랫폼 API의 런타임 프레임워크(runtime framework)를 예시하는 도면.

도 23은 "FindAll" 동작의 실행을 예시하는 도면.

도 24는 저장 플랫폼 스키마로부터 저장 플랫폼 API 클래스들이 생성되는 프로세스를 예시하는 도면.

도 25는 파일 API가 기초고 있는 스키마를 예시하는 도면.

도 26은 데이터 보안을 위해 사용되는 액세스 마스크 포맷을 예시하는 도면이다.

도 27은 기존의 보안 영역으로부터 새로운 동일하게 보호되는 보안 영역이 만들어지는 것을 도시하는 도면(부분 a, b, 및 c).

도 28은 아이템 검색 뷰(Item search view)의 개념을 예시하는 도면.

도 29는 예시적인 아이템 계층 구성을 예시하는 도면.

도 30A는 제1 및 제2 코드 세그먼트들이 통신하는 도관(conduit)으로서의 인터페이스 Interface1을 예시하는 도면.

도 30B는 시스템의 제1 및 제2 코드 세그먼트들이 매체 M을 통하여 통신할 수 있게 하는 인터페이스 객체 I1 및 I2를 포함하는 인터페이스를 예시하는 도면.

도 31A는 인터페이스의 통신들을 변환하기 위해 인터페이스 Interface1에 의해 제공된 기능이 어떻게 다수의 인터페이스 Interface1A, Interface1B, Interface1C로 세분될 수 있는지를 예시하는 도면.

도 31B는 인터페이스 I1에 의해 제공된 기능이 어떻게 다수의 인터페이스 I1a, 인터페이스 I1b, 인터페이스 I1c로 세분될 수 있는지를 예시하는 도면.

도 32A는 무의미한 파라미터 정밀도(precision)가 무시되거나 또는 임의의 파라미터로 대체될 수 있는 시나리오를 예시하는 도면.

도 32B는 인터페이스가 파라미터들을 무시하거나 또는 인터페이스에 파라미터를 부가하도록 정의되는 대체 인터페이스에 의해 대체되는 시나리오를 예시하는 도면.

도 33A는 제1 및 제2 코드 세그먼트들이 이들 둘 다를 포함하는 모듈로 병합되는 시나리오를 예시하는 도면.

도 33B는 병합된 인터페이스를 형성하기 위해 인터페이스의 일부 또는 전부가 또 다른 인터페이스 내에 인라인(inline)으로 기입될 수 있는 시나리오를 예시하는 도면.

도 34A는 미들웨어(middleware)의 하나 이상의 피스들(pieces)을 하나 이상의 서로 다른 인터페이스들에 일치시키도록 하기 위해 상기 하나 이상의 피스들이 제1 인터페이스 상의 통신들을 어떻게 변환할 수 있는지를 예시하는 도면.

도 34B는 하나의 인터페이스로부터 통신을 수신하지만 그 기능을 제2 및 제3 인터페이스들로 송신하는 인터페이스와 함께 어떻게 코드 세그먼트가 도입될 수 있는지를 예시하는 도면.

도 35A는 JIT(just-in-time complier)가 어떻게 하나의 코드 세그먼트로부터 또 다른 코드 세그먼트로 통신들을 변환할 수 있는지를 예시하는 도면.

도 35B는 하나 이상의 인터페이스들을 동적으로 재기입하는 JIT 방법이 상기 인터페이스를 동적으로 팩터링(factor)하거나 또는 그렇지 않으면 상기 인터페이스를 변경하기 위해 적용될 수 있는 JIT 방법을 예시하는 도면.

도 36은 각각의 스키마 내의 갖가지 아이템들 간의 상호 관계를 보여주기 위해 베이스 스키마(도 7) 및 코어 스키마(도 8A)의 선택된 엘리먼트들과 함께, 이미지 스키마를 예시하는 도면.

발명의 상세한 설명

목차

I. 도입

A. 예시적인 컴퓨팅 환경

B. 전통적인 파일 기반 저장

Ⅱ. 데이터를 조직, 검색, 및 공유하기 위한 WINFS 저장 플랫폼

A. 용어 모음

B. 저장 플랫폼 개관

C. 데이터 모델

1. 아이템

2. 아이템 식별

3. 아이템 폴더 및 카테고리

4. 스키마

a) 베이스 스키마

b) 코어 스키마

5. 관계

a) 관계 선언

b) 홀딩 관계

c) 임베딩 관계

d) 참조 관계

e) 규칙 및 제약

f) 관계의 순서화

6. 확장성

a) 아이템 확장

b) NestedElement 타입 확장

D. 데이터베이스 엔진

1. UDT를 이용한 데이터 스토어 구현

2. 아이템 매핑

3. 확장 매핑

4. 네스트된 엘리먼트 매핑

5. 객체 ID

6. SQL 객체 명명

7. 칼럼 명명

8. 검색 뷰

a) 아이템

(1) 마스터 아이템 검색 뷰

(2) 타입 아이템 검색 뷰

b) 아이템 확장

(1) 마스터 확장 검색 뷰

(2) 타입 확장 검색 뷰

c) 네스트된 엘리먼트

d) 관계

(1) 마스터 관계 검색 뷰

(2) 관계 인스턴스 검색 뷰

e)

9. 업데이트

10. 변경 추적 및 툼스톤(Tomstones)

a) 변경 추적

(1) "마스터" 검색 뷰에서의 변경 추적

(2) "타입" 검색 뷰에서의 변경 추적

b) 툼스톤

(1) 아이템 툼스톤

(2) 확장 툼스톤

(3) 관계 툼스톤

(4) 툼스톤 클린업

11. 도우미 API 및 함수

a) 함수 [System.Storage].GetItem

b) 함수 [System.Storage].GetExtension

c) 함수 [System.Storage].GetRelationship

12. 메타데이터

a) 스키마 메타데이터

b) 인스턴스 메타데이터

E. 보안

F. 통지 및 변경 추적

G. 동기화

1. 저장 플랫폼 대 저장 플랫폼 동기화

a) 동기화(Sync) 제어 애플리케이션

b) 스키마 주석

c) Sync 구성

(1) 공동체 폴더 - 매핑

(2) 프로파일

(3) 스케줄

d) 충돌 핸들링

(1) 충돌 검출

(a) 지식 기반 충돌

(b) 제약 기반 충돌

(2) 충돌 처리

(a) 자동 충돌 레졸루션

(b) 충돌 로깅

(c) 충돌 검사 및 레졸루션

(d) 복제의 컨버전스(convergence) 및 충돌

레졸루션의 전파

2. 비저장 플랫폼 데이터 스토어에의 동기화

a) Sync 서비스

(1) 변경 열거

(2) 변경 적용

(3) 충돌 레졸루션

b) 어댑터 구현

3. 보안

4. 관리능력

H. 전통적인 파일 시스템 상호 운용성

I. 저장 플랫폼 API

Ⅲ. 이미지 스키마 및 하위 스키마(이미지 스키마 세트)

A. 이미지 스키마

B. 사진 스키마

C. 분석 속성 스키마

Ⅳ. 결론

I. 도입

본 발명의 주제는 법정 요건들을 충족시키도록 상세하게 설명된다. 그러나, 이 설명 자체는 이 특허의 범위를 제한하도록 의도되어 있지 않다. 도리어, 발명자들은 청구된 주제가 다른 특허 또는 장래의 기술과 관련하여, 이 명세서에서 설명된 것들과 유사한 단계들의 조합 또는 상이한 단계들을 포함하도록 다른 방법들로 구현될 수도 있다는 것을 염두에 두었다. 더욱이, "단계"(step)라는 용어는 채용된 방법들의 서로 다른 요소들을 의미하도록 본 명세서에서 사용될 수 있지만, 이 용어는 개개의 단계들의 순서가 명시적으로 기술되지 않는 한 본 명세서에서 개시된 갖가지 단계들 사이의 어떤 특정 순서를 의미하는 것으로 해석되어서는 안 된다.

A. 예시적인 컴퓨팅 환경

본 발명의 다수의 실시예들은 컴퓨터 상에서 실행될 수 있다. 도 1 및 이하의 논의는 본 발명의 구현될 수 있는 적당한 컴퓨팅 환경에 대한 간단한 일반적인 설명을 제공하도록 의도되어 있다. 필수적인 것은 아니지만, 본 발명의 갖가지 양태들은 클라이언트 워크스테이션 또는 서버와 같은 컴퓨터에 의해 실행되는 프로그램 모듈과 같은 컴퓨터 실행 가능한 명령들의 일반적인 문맥에서 설명될 수 있다. 일반적으로, 프로그램 모듈들은 특정 작업을 수행하거나 또는 특정 추상(abstract) 데

이터 타입들을 구현하는 루틴, 프로그램, 객체, 컴포넌트, 데이터 구조 등을 포함한다. 또한, 본 발명은 핸드 헬드 디바이스, 멀티 프로세서 시스템, 마이크로프로세서 기반 또는 프로그램 가능한 소비자 전자 기기, 네트워크 PC, 미니컴퓨터, 메인프레임 컴퓨터 등을 포함하는 다른 컴퓨터 시스템 구성을 이용하여 실시될 수도 있다. 본 발명은 또한 통신 네트워크를 통하여 연결되어 있는 원격 처리 디바이스들에 의해 작업이 수행되는 분산 컴퓨팅 환경에서 실시될 수도 있다. 분산 컴퓨팅 환경에서는, 프로그램 모듈들은 로컬 메모리 기억 장치 및 원격 메모리 기억 장치 양쪽 모두에 위치할 수 있다.

도 1에 도시된 바와 같이, 예시적인 범용 컴퓨팅 시스템은, 처리 장치(21), 시스템 메모리(22), 및 이 시스템 메모리를 포함하는 각종 시스템 컴포넌트들을 상기 처리 장치(21)에 연결하는 시스템 버스(23)를 포함하는 종래의 퍼스널 컴퓨터(20) 또는 그와 유사한 것을 포함한다. 시스템 버스(23)는 각종 버스 아키텍처 중 어느 하나를 이용한 메모리 버스 또는 메모리 컨트롤러, 주변 버스, 및 로컬 버스를 포함하는 몇몇 타입의 버스 구조들 중 어느 하나일 수 있다. 시스템 메모리는 판독 전용 메모리(ROM)(24) 및 랜덤 액세스 메모리(RAM)(25)를 포함한다. 시동(start-up) 중과 같이 퍼스널 컴퓨터(20) 내의 엘리먼트들 간의 정보 전송을 돕는 기본 루틴들을 포함하는 기본 입출력 시스템(BIOS)(26)은 ROM(24)에 저장된다. 퍼스널 컴퓨터(20)는 또한 도시되지 않은 하드 디스크로부터 판독하고 거기에 기록하기 위한 하드 디스크 드라이브(27), 탈착 가능한 자기 디스크(29)로부터 판독하거나 거기에 기록하기 위한 자기 디스크 드라이브(28), 및 CD ROM 또는 다른 광학 매체와 같은 탈착 가능한 광 디스크(31)로부터 판독하거나 거기에 기록하기 위한 광 디스크 드라이브(30)를 포함할 수 있다. 하드 디스크 드라이브(27), 자기 디스크 드라이브(28), 및 광 디스크 드라이브(30)는 각각 하드 디스크 드라이브 인터페이스(32), 자기 디스크 드라이브 인터페이스(33), 및 광 디스크 드라이브 인터페이스(34)에 의해 시스템 버스(23)에 접속된다. 이 드라이브들과 그들의 관련 컴퓨터 판독 가능 매체는 컴퓨터 판독 가능 명령어, 데이터 구조, 프로그램 모듈 및 퍼스널 컴퓨터(20)용의 다른 데이터의 불휘발성 저장을 제공한다. 여기에서 설명된 예시적 환경은 하드 디스크, 탈착 가능한 자기 디스크(29), 및 탈착 가능한 광 디스크(31)를 채용하지만, 숙련된 당업자라면 자기 카세트, 플래시 메모리 카드, 디지털 비디오 디스크, 베르누이 카트리지, 랜덤 액세스 메모리(RAMs), 판독 전용 메모리(ROMs) 등과 같이, 컴퓨터에 의해 액세스 가능한 데이터를 저장할 수 있는 다른 타입의 컴퓨터 판독 매체도 또한 이 예시적인 운영 환경에서 사용될 수 있다는 것을 알 것이다. 마찬가지로, 이 예시적인 환경은 또한 열 센서 및 보안 또는 화재 경보 시스템과 같은 여러 타입의 모니터링 디바이스, 및 다른 정보 소스를 포함할 수도 있다.

하드 디스크, 자기 디스크(29), 광 디스크(31), ROM(24) 또는 RAM(25) 상에는, 운영 시스템(35), 하나 이상의 애플리케이션 프로그램(36), 다른 프로그램 모듈들(37) 및 프로그램 데이터(38)를 포함하는, 다수의 프로그램 모듈들이 저장될 수 있다. 사용자는 키보드(40) 및 포인팅 디바이스(42)와 같은 입력 디바이스들을 통하여 퍼스널 컴퓨터(20)에 커맨드 및 정보를 입력할 수 있다. (도시되지 않은) 다른 입력 디바이스들은 마이크, 조이스틱, 게임 패드, 위성 디스크(satellite disk), 스캐너 또는 그와 유사한 것을 포함할 수 있다. 이들 및 다른 입력 디바이스들은 흔히 시스템 버스에 연결되어 있는 직렬 포트 인터페이스(46)를 통하여 처리 장치(21)에 접속되지만, 병렬 포트, 게임 포트 또는 유니버설 시리얼 버스(USB)와 같은 다른 인터페이스들에 의해 접속될 수도 있다. 모니터(47) 또는 다른 타입의 디스플레이 디바이스도 비디오 어댑터(48)와 같은 인터페이스를 통하여 시스템 버스(23)에 접속된다. 모니터(47) 이외에, 퍼스널 컴퓨터는 일반적으로 스피커 및 프린터와 같은 (도시되지 않은) 다른 주변 출력 디바이스들을 포함한다. 도 1의 예시적인 시스템은 또한 호스트 어댑터(55), 소형 컴퓨터 시스템 인터페이스(SCSI) 버스(56), 및 SCSI 버스(56)에 접속된 외부 기억 장치(62)를 포함한다.

퍼스널 컴퓨터(20)는 원격 컴퓨터(49)와 같은 하나 이상의 원격 컴퓨터들에의 논리적 접속을 이용한 네트워크 환경에서 동작할 수도 있다. 원격 컴퓨터(49)는 또 다른 퍼스널 컴퓨터, 서버, 라우터, 네트워크 PC, 피어 디바이스(peer device) 또는 다른 공통의 네트워크 노드일 수 있고, 도 1에는 메모리 기억 디바이스(50)만이 예시되어 있지만, 일반적으로 퍼스널 컴퓨터(20)와 관련하여 상술한 엘리먼트들 중의 다수 또는 전부를 포함한다. 도 1에 도시된 논리적 접속은 근거리 통신망(LAN)(51) 및 광역 통신망(WAN)(52)를 포함한다. 그러한 네트워킹 환경들은 사무실, 기업 광역(enterprise wide) 컴퓨터 네트워크, 인트라넷 및 인터넷에서 흔한 것이다.

LAN 네트워크 환경에서 사용될 때, 퍼스널 컴퓨터(20)는 네트워크 인터페이스 또는 어댑터(53)를 통하여 LAN(51)에 접속된다. WAN 네트워크 환경에서 사용될 때, 퍼스널 컴퓨터(20)는 일반적으로 모뎀(54)이나 또는 인터넷과 같은 광역 통신망(52)을 통하여 통신을 설정하기 위한 다른 수단을 포함한다. 내장형이거나 또는 외장형일 수 있는 모뎀(54)은 직렬 포트 인터페이스(46)를 통하여 시스템 버스(23)에 접속된다. 네트워킹크 환경에서, 퍼스널 컴퓨터(20)와 관련하여 도시된 프로그램 모듈들, 또는 그것들의 일부는, 원격 메모리 기억 장치에 저장될 수 있다. 도시된 네트워크 접속들은 예시적인 것이고 컴퓨터들 간에 통신 링크를 설정하는 다른 수단이 사용될 수도 있다는 것을 알 것이다.

도 2의 블록도에 예시된 바와 같이, 컴퓨터 시스템(200)은 대략 3개의 컴포넌트 그룹: 하드웨어 컴포넌트(202), 하드웨어/소프트웨어 인터페이스 시스템 컴포넌트(204), 및 애플리케이션 프로그램 컴포넌트(206)(본 명세서의 특정한 상황에서 "사용자 컴포넌트" 또는 "소프트웨어 컴포넌트"라고도 함)로 나누어질 수 있다.

컴퓨터 시스템(200)의 갖가지 실시예들에서, 그리고 다시 도 1을 참조하면, 하드웨어 컴포넌트(202)는 중앙 처리 장치(CPU)(21), 메모리(ROM(24) 및 ROM(25) 양쪽 모두), 기본 입출력 시스템(BIOS)(26), 및 특히 키보드(40), 마우스(42), 모니터(47), 및/또는 (도시되지 않은) 프린터와 같은 각종 입출력(I/O) 디바이스들을 포함할 수 있다. 하드웨어 컴포넌트(202)는 컴퓨터 시스템(200)에 대한 기본적인 물리적 인프라구조(infrastructure)를 포함한다.

애플리케이션 프로그램 컴포넌트(206)는 컴파일러, 데이터베이스 시스템, 워드 프로세서, 사무용 프로그램, 비디오 게임 등을 포함하면서도 이들에 한정되지 않는 다양한 소프트웨어 프로그램들을 포함한다. 애플리케이션 프로그램은 다양한 사용자들(머신, 다른 컴퓨터 시스템, 및/또는 최종 사용자)을 위해 문제를 해결하고, 해법을 제공하고, 데이터를 처리하기 위해 컴퓨터 자원들(computer resources)이 이용되도록 하는 수단을 제공한다.

하드웨어/소프트웨어 인터페이스 시스템 컴포넌트(204)는 대개의 경우에, 그것 자체가 셸(shell) 및 커널(kernel)을 포함하는 운영 시스템을 포함한다(그리고, 일부 실시예에서는, 단지 그것만으로 이루어질 수 있다). "운영 시스템"(OS: operarating system)은 애플리케이션 프로그램들과 컴퓨터 하드웨어 간의 매개자의 역할을 하는 특별한 프로그램이다. 하드웨어/소프트웨어 인터페이스 시스템 컴포넌트(204)는 또한 컴퓨터 시스템 내의 운영 시스템 대신에 또는 그것에 더하여, 가상 머신 관리자(VMM: virtual machine manager), 공통 언어 런타임(CLR: Common Language Runtime) 또는 그것의 기능적 등가물, 자바 가상 머신(JVM : Java Virtual Machine) 또는 그것의 기능적 등가물, 또는 다른 그러한 소프트웨어 컴포넌트들을 포함할 수 있다. 하드웨어/소프트웨어 인터페이스 시스템의 용도는 사용자가 애플리케이션 프로그램들을 실행할 수 있는 환경을 제공하는 것이다. 임의의 하드웨어/소프트웨어 인터페이스 시스템의 목적은 컴퓨터 하드웨어를 효율적으로 이용할 뿐만 아니라, 컴퓨터 시스템을 사용하기 편리하게 하는 것이다.

하드웨어/소프트웨어 인터페이스 시스템은 일반적으로 시동 시에 컴퓨터 시스템에 로드되고 그 후에 컴퓨터 시스템 내의 모든 애플리케이션 프로그램들을 관리한다. 애플리케이션 프로그램들은 애플리케이션 프로그램 인터페이스(API)를 통하여 서비스를 요구함으로써 하드웨어/소프트웨어 인터페이스 시스템과 상호 작용한다. 어떤 애플리케이션 프로그램들은 최종 사용자들이 커맨드 언어 또는 그래픽 사용자 인터페이스(GUI)와 같은 사용자 인터페이스를 통하여 하드웨어/소프트웨어 인터페이스 시스템과 상호 작용할 수 있게 한다.

하드웨어/소프트웨어 인터페이스 시스템은 전통적으로 애플리케이션들에 대한 각종 서비스를 수행한다. 다수의 프로그램들이 동시에 실행될 수 있는 멀티태스킹 하드웨어/소프트웨어 인터페이스 시스템에서는, 하드웨어/소프트웨어 인터페이스 시스템은 어느 애플리케이션들이 어떤 순서로 실행되어야 하는지 및 전환을 위해 다른 애플리케이션으로 스위칭하기 전에 각각의 애플리케이션에 대해 얼마나 많은 시간이 할당되어야 하는지를 결정한다. 하드웨어/소프트웨어 인터페이스 시스템은 또한 다수의 애플리케이션들 사이에서의 내부 메모리의 공유를 관리하고, 하드 디스크, 프린터, 및 다이얼업 포트(dial-up ports)와 같은 부착된 하드웨어 디바이스들로의 입력 및 그들로부터의 출력을 처리한다. 하드웨어/소프트웨어 인터페이스 시스템은 또한 동작 상태 및 발생했을지도 모르는 에러들에 관한 메시지를 각각의 애플리케이션에게(및, 어떤 경우에는, 최종 사용자에게) 송신한다. 하드웨어/소프트웨어 인터페이스 시스템은 또한 일괄 작업(예를 들면, 인쇄)의 관리를 오프로드(offload)함으로써 개시 애플리케이션(initiating application)이 이 작업으로부터 해방되어 다른 처리 및/또는 동작을 재개할 수 있도록 할 수 있다. 병렬 처리를 제공할 수 있는 컴퓨터들에서는, 하드웨어/소프트웨어 인터페이스 시스템은 또한 프로그램이 2 이상의 프로세서 상에서 동시에 실행되도록 그 프로그램의 분할을 관리한다.

하드웨어/소프트웨어 인터페이스 시스템 셸(여기에서는 간단히 "셸"이라고 함)은 하드웨어/소프트웨어 인터페이스 시스템에 대한 대화형 최종 사용자 인터페이스이다. (셸은 또한 "커맨드 번역기"(command interpreter)라고도 하고, 운영 시스템에서는, "운영 시스템 셸"(operating system shell)이라고도 한다). 셸은 애플리케이션 프로그램들 및/또는 최종 사용자들에 의해 직접 액세스 가능한 하드웨어/소프트웨어 인터페이스 시스템의 외부층이다. 셸과는 대조적으로, 커널은 하드웨어 컴포넌트들과 직접 상호 작용하는 하드웨어/소프트웨어 인터페이스 시스템의 가장 안쪽 층이다.

본 발명의 다수의 실시예들은 컴퓨터화된 시스템들에 특히 적합한 것으로 생각되지만, 이 명세서 내의 어떤 것도 본 발명을 그러한 실시예들에 한정하도록 의도되어 있지 않다. 그에 반하여, 여기에서 사용된 "컴퓨터 시스템"이라는 용어는 정보를 저장하고 처리할 수 있고 저장된 정보를 이용하여, 해당 디바이스가 본질상 전자적이든, 기계적이든, 논리적이든, 또는 가상적이든 상관없이, 디바이스 자체의 거동 또는 실행을 제어할 수 있는 임의의 및 모든 디바이스들을 두루 포함하도록 의도되어 있다.

B. 전통적인 파일 기반 저장

오늘날 대부분의 컴퓨터 시스템에서, "파일"(files)은 애플리케이션 프로그램, 데이터 세트 등은 물론 하드웨어/소프트웨어 인터페이스 시스템을 포함할 수 있는 저장 가능한 정보 단위이다. 모든 최신의 하드웨어/소프트웨어 인터페이스 시스템(Windows, Unix, Linux, Mac OS, 가상 머신 시스템 등)에서, 파일은 하드웨어/소프트웨어 인터페이스 시스템에 의해 조작 처리될 수 있는 기본적인 개별 (저장 가능하고 검색 가능한) 정보 단위(예를 들면, 데이터, 프로그램 등)이다. 파일 그룹들은 일반적으로 "폴더"(folders)로 조직된다. 마이크로소프트 윈도즈, 매킨토시 OS, 및 다른 하드웨어/소프트웨어 인터페이스 시스템에서, 폴더는 하나의 정보 단위로서 검색되고, 이동되고, 다르게는 조작 처리될 수 있는 파일들의 컬렉션이다. 이 폴더들은 다시 (아래에서 더 상세히 논의되는) "디렉토리"(directory)라고 하는 트리 기반 계층 구성 배열로 조직된다. DOS, z/OS 및 대부분의 Unix 기반 운영 시스템과 같은, 어떤 다른 하드웨어/소프트웨어 인터페이스 시스템들에서는, "디렉토리" 및/또는 "폴더"라는 용어들이 상호 교환 가능하고, 초기 애플 컴퓨터 시스템들(예를 들면, Apple Ⅱe)은 디렉토리 대신에 "카탈로그"(catalog)라는 용어를 사용하였지만, 여기에서 사용될 때 모든 이들 용어들은 같은 의미이고 상호 교환 가능한 것으로 생각되고 계층형 정보 저장 구조들 및 그들의 폴더 및 파일 컴포넌트들에 대한 모든 다른 동등한 용어들 및 참조들을 더 포함하도록 의도되어 있다.

전통적으로, 디렉토리(폴더들의 디렉토리(directory of folders)라고도 알려져 있음)는, 파일들이 폴더들로 그룹화되고 폴더는 다시 디렉토리 트리를 포함하는 상대적 노드 위치(relative nodal locations)에 따라서 배열되어 있는 트리 기반 계층형 구조이다. 예를 들면, 도 2A에 예시된 바와 같이, DOS 기반 파일 시스템 베이스 폴더("루트 디렉토리")(212)는 복수의 폴더(214)를 포함할 수 있고, 이 폴더들 각각은 또한 (그 특정 폴더의 "하위 폴더들"로서) 부가적인 폴더들(216)을 포함할 수 있고, 이것들 각각은 또한 부가적인 폴더들(218)을 무한히 포함할 수 있다. 이들 폴더들 각각은 하나 이상의 파일들(220)을 가질 수 있지만, 하드웨어/소프트웨어 인터페이스 시스템 레벨에서, 폴더 내의 개개의 파일들은 트리 계층 구성에서의 그들의 위치 이외에 아무 것도 공통으로 갖지 않는다. 파일들을 폴더 계층 구성들로 조직하는 이러한 방법은 이들 파일들을 저장하기 위해 사용되는 전형적인 저장 매체(예를 들면, 하드 디스크, 플로피 디스크, CD-ROM 등)의 물리적 조직을 간접적으로 반영한다는 것은 놀라운 일이 아니다.

전술한 것들 외에, 각각의 폴더는 그것의 하위 폴더 및 그것의 파일들에 대한 컨테이너이다. 즉, 각각의 폴더는 그것의 하위 폴더들 및 파일들을 소유한다. 예를 들면, 폴더가 하드웨어/소프트웨어 인터페이스 시스템에 의해 삭제되는 경우, 그 폴더의 하위 폴더들 및 파일들도 또한 삭제된다(각각의 하위 폴더의 경우에는, 그 자신의 하위 폴더들 및 파일들을 반복적으로 더 포함한다). 마찬가지로, 각각의 파일은 일반적으로 오직 하나의 폴더에 의해서만 소유되며, 파일은 복사될 수 있고 그 복사본은 상이한 폴더에 위치할 수 있지만, 파일의 복사본 자체는 원본과 직접적인 관계가 없는 별개의 분리된 단위이다(예를 들면, 원본 파일에 대한 변경은 하드웨어/소프트웨어 인터페이스 시스템 레벨에서 복사본 파일에 반영되지 않는다). 따라서, 이와 관련하여, 파일들 및 폴더들은 본질상 특징적으로 "물리적"이다. 왜냐하면, 폴더들은 물리적 컨테이너처럼 취급되고, 파일들은 이들 컨테이너 내의 별개의 분리된 물리적 엘리먼트들로서 취급되기 때문이다.

Ⅱ. 데이터를 조직, 검색, 및 공유하기 위한 WINFS 저장 플랫폼

본 발명은, 본 명세서의 앞에서 논의된 바와 같이 참고로 반영된 관련 발명들과 조합하여, 데이터를 조직하고, 검색하고, 공유하기 위한 저장 플랫폼에 관한 것이다. 본 발명의 저장 플랫폼은 위에서 논의된 기존의 파일 시스템 및 데이터베이스 시스템을 넘어서 데이터 플랫폼을 확장하고 넓히며, 아이템이라고 불리는 새로운 형태의 데이터를 포함하는 모든 타입의 데이터에 대한 스토어가 되도록 설계되어 있다.

A. 용어 모음

본 명세서 및 청구항들에서 사용될 때, 이하의 용어들은 이하의 의미를 갖는다:

- "아이템"은 단순한 파일과 달리, 하드웨어/소프트웨어 인터페이스 시스템 셸에 의해 최종 사용자에게 노출되는 모든 객체들에 걸쳐서 공통으로 지원되는 속성들의 기본 세트를 갖는 객체인, 하드웨어/소프트웨어 인터페이스 시스템에 액세스할 수 있는 저장 가능한 정보의 단위이다. 아이템은 또한 새로운 속성들 및 관계들이 도입될 수 있게 하는 특징들을 포함하는 모든 아이템 타입들에 걸쳐서 공통으로 지원되는 (그리고 나중에 상세히 논의되는) 속성들 및 관계들을 갖는다.

- "운영 시스템"(OS)은 애플리케이션 프로그램들과 컴퓨터 하드웨어 간의 매개자의 역할을 하는 특별한 프로그램이다. 운영 시스템은 대개의 경우에 셸 및 커널을 포함한다.

● "하드웨어/소프트웨어 인터페이스 시스템"은, 컴퓨터 시스템의 기초를 이루는 하드웨어 컴포넌트들과 컴퓨터 시스템 상에서 실행되는 애플리케이션들 간의 인터페이스의 역할을 하는 소프트웨어, 또는 하드웨어와 소프트웨어의 조합이다. 하드웨어/소프트웨어 인터페이스 시스템은 전형적으로 운영 시스템을 포함한다(그리고, 일부 실시예들에서는, 단지 그것만으로 이루어질 수 있다). 하드웨어/소프트웨어 인터페이스 시스템은 또한 컴퓨터 시스템 내의 운영 시스템 대신에 또는 그것에 더하여, 가상 머신 관리자(VMM), 공통 언어 런타임(CLR) 또는 그것의 기능적 등가물, 자바 가상 머신(JVM) 또는 그것의 기능적 등가물, 또는 다른 그러한 소프트웨어 컴포넌트들을 포함할 수 있다. 하드웨어/소프트웨어 인터페이스 시스템의 용도는 사용자가 애플리케이션 프로그램들을 실행할 수 있는 환경을 제공하는 것이다. 임의의 하드웨어/소프트웨어 인터페이스 시스템의 목적은 컴퓨터 하드웨어를 효율적으로 이용할 뿐만 아니라, 컴퓨터 시스템을 사용하기 편리하게 하는 것이다.

B. 저장 플랫폼 개관

도 3을 참조하면, 저장 플랫폼(300)은 데이터베이스 엔진(314) 상에 구현된 데이터 스토어(302)를 포함한다. 일 실시예에서, 데이터베이스 엔진은 객체 관계 확장들(object relational extensions)을 갖는 관계 데이터베이스 엔진을 포함한다. 일 실시예에서, 관계 데이터베이스 엔진(314)은 마이크로소프트 SQL 서버 관계 데이터베이스 엔진을 포함한다. 데이터 스토어(302)는 데이터의 조직, 검색, 공유, 동기화, 및 보안을 지원하는 데이터 모델(304)을 구현한다. 데이터의 특정 타입들은 스키마(340)와 같은 스키마들에서 기술되고, 저장 플랫폼(300)은 아래에서 더 상세히 설명되는 바와 같이 그 스키마들을 확장할 뿐만 아니라 그 스키마들을 전개(deploy)하기 위한 도구들(346)을 제공한다.

데이터 스토어(302) 내에서 구현되는 변경 추적 메커니즘(306)은 데이터 스토어에 대한 변경들을 추적하는 능력을 제공한다. 데이터 스토어(302)는 또한 보안 능력들(308) 및 프로모션/디모션(promotion/demotion) 능력(310)을 제공하고, 이 양자에 대해서는 아래에서 더 상세히 논의된다. 데이터 스토어(302)는 또한 데이터 스토어(302)의 능력들을 저장 플랫폼을 이용하는 애플리케이션 프로그램들(예를 들면, 애플리케이션 프로그램들(350a, 350b, 및 350c)) 및 다른 저장 플랫폼 컴포넌트들에게 노출시키기 위한 애플리케이션 프로그래밍 인터페이스들(312)의 세트를 제공한다. 본 발명의 저장 플랫폼은 또한 애플리케이션 프로그램들(350a, 350b, 및 350c)과 같은 애플리케이션 프로그램들이 저장 플랫폼의 전술한 능력들 모두에 액세스하고 스키마들에서 기술된 데이터에 액세스할 수 있게 하는 애플리케이션 프로그램 인터페이스들(API)(322)을 더 포함한다. 저장 플랫폼 API(322)는 OLE DB API(324) 및 마이크로소프트 윈도즈 Win32 API(326)와 같은 다른 API들과 조합하여 애플리케이션 프로그램들에 의해 사용될 수 있다.

본 발명의 저장 플랫폼(300)은 사용자들 또는 시스템들 사이에서의 데이터의 공유를 용이하게 하는 동기화 서비스(330)를 포함하는 갖가지 서비스들(328)을 애플리케이션 프로그램들에게 제공할 수 있다. 예를 들면, 동기화 서비스(330)는 다른 포맷들을 갖는 데이터 스토어들(342)에의 액세스뿐만 아니라, 데이터 스토어(302)와 동일한 포맷을 갖는 다른 데이터 스토어들(340)과의 상호 운용성을 가능케 할 수 있다. 저장 플랫폼(300)은 또한 윈도즈 NTFS 파일 시스템(318)과 같은 기존의 파일 시스템과의 데이터 스토어(302)의 상호 운용성을 허용하는 파일 시스템 능력을 제공한다. 적어도 일부 실시예들에서, 저장 플랫폼(320)은 또한 데이터에 작용될 수 있고 다른 시스템들과의 상호 작용을 가능케 하는 부가적인 능력들을 애플리케이션 프로그램들에게 제공할 수 있다. 이들 능력들은 정보 에이전트(Info Agent) 서비스(334) 및 통지 서비스(332)와 같은 부가적인 서비스(328)의 형태로는 물론, 다른 유틸리티(336)의 형태로도 구현될 수 있다.

적어도 일부 실시예들에서, 저장 플랫폼은 컴퓨터 시스템의 하드웨어/소프트웨어 인터페이스 시스템 내에 구현되거나, 또는 그것의 필수적인 부분을 형성한다. 예를 들면, 그리고 제한 없이, 본 발명의 저장 플랫폼은 운영 시스템, 가상 머신 관리자(VMM), 공통 언어 런타임(CLR) 또는 그것의 기능적 등가물, 자바 가상 머신(JVM) 또는 그것의 기능적 등가물 내에 구현되거나, 또는 그것의 필수적인 부분을 형성할 수 있다. 본 발명의 저장 플랫폼은, 그것의 공통의 저장 토대(storage foundation) 및 조직적으로 배열된 데이터(schematized data)를 통하여, 소비자, 지식 노동자 및 기업에게 보다 효율적인 애플리케이션 개발을 가능케 한다. 그것은 그것의 데이터 모델에서의 고유한 능력들을 이용할 수 있게 할 뿐만 아니라, 기존의 파일 시스템 및 데이터베이스 액세스 방법들을 포용하고 확장하는 풍부하고 확장성이 있는 애플리케이션 프로그래밍 표면적을 제공한다.

이하의 설명에서, 및 여러 도면들에서, 본 발명의 저장 플랫폼(300)은 "WinFS"로 불릴 수 있다. 그러나, 저장 플랫폼을 언급하기 위해 이러한 이름을 사용하는 것은 오로지 설명의 편의를 위한 것일 뿐 결코 제한하려는 것이 아니다.

C. 데이터 모델

본 발명의 저장 플랫폼(300)의 데이터 스토어(302)는 스토어 내에 상주하는 데이터의 조직, 검색, 공유, 동기화, 및 보안을 지원하는 데이터 모델을 구현한다. 본 발명의 데이터 모델에서, "아이템"은 저장 정보의 기본적인 단위이다. 데이터 모델은 아이템 및 아이템 확장을 선언하고 아이템들 간의 관계를 설정하고 아이템 폴더 및 카테고리 내의 아이템들을 조직하기 위한 메커니즘을 제공하는데, 이에 대해서는 아래에서 더 상세히 설명한다.

데이터 모델은 2개의 기본 메커니즘, 타입(Types) 및 관계(Relationships)에 의존한다. 타입은 타입의 인스턴스의 형태를 제어하는 포맷을 제공하는 구조이다. 포맷은 속성들의 순서화된 세트로서 표현된다. 속성(Property)은 주어진 타입의 값 또는 값들의 세트에 대한 이름이다. 예를 들면 USPostalAddress 타입은 Street(가), City(시), Zip(우편번호), State(주) 속성들을 갖고, 여기에서 Street, City 및 State는 String(문자열) 타입이고, Zip은 Int32 타입이다. Street는 다중 값(즉, 값들의 세트)일 수 있고 따라서 주소가 Street 속성에 대해 2 이상의 값을 갖도록 허용한다. 시스템은 다른 타입들의 구성에 이용될 수 있는 특정 기본 타입들(primitive types)을 정의한다 - 이들은 String, Binary, Boolean, Int16, Int32, Int64, Single, Double, Byte, DateTime, Decimal 및 GUID를 포함한다. 타입의 속성들은 기본 타입들 중 임의의 것 또는 (아래에서 지적된 얼마간의 제한을 갖고) 구성된 타입들 중 임의의 것을 이용하여 정의될 수 있다. 예를 들면 Coordinate 및 Address 속성들을 가진 Location 타입이 정의될 수 있고, 여기에서 Address 속성은 상술한 바와 같이 USPostalAddress 타입이다. 속성들은 또한 필수 사양이거나 선택 사양일 수 있다.

관계(Relationships)는 선언될 수 있고 2개의 타입의 인스턴스들의 세트들 간의 매핑을 나타낸다. 예를 들면 어느 사람들이 어느 장소들에 사는지를 정의하는 Person(사람) 타입과 Location(장소) 타입 간에 선언된 LivesAt이라고 불리는 관계가 있을 수 있다. 관계는 이름과, 2개의 종점(endpoints), 즉 소스 종점 및 타깃 종점을 갖는다. 관계는 또한 속성들의 순서화된 세트를 가질 수 있다. 소스 및 타깃 종점들 모두는 이름 및 타입을 가질 수 있다. 예를 들면 LiveAt 관계는 Person 타입의 Occupant(거주자)라고 불리는 소스와 Location 타입의 Dwelling(거주지)라고 불리는 타깃을 갖고 또한 거주자가 거주지에 살았던 기간을 나타내는 StartDate(시작 날짜) 및 EndDate(종료 날짜) 속성들을 갖는다. 사람은 시간에 따라 다수의 거주지에 살 수 있고 거주지는 다수의 거주자를 가질 수 있으므로 StartDate 및 EndDate 정보를 가장 많이 넣을 것 같은 장소는 관계 자체 상일 것이다.

관계는 종점 타입으로서 주어진 타입에 의해 구속되는 인스턴스들 간의 매핑을 정의한다. 예를 들면, LivesAt 관계는 자동차가 거주자인 관계가 될 수 없다. 왜냐하면 자동차는 사람이 아니기 때문이다.

데이터 모델은 타입들 간의 서브타입-슈퍼타입 관계의 정의를 허용한다. BaseType 관계로도 알려져 있는 서브타입-슈퍼타입 관계는 타입 A가 타입 B에 대한 BaseType이면 그것은 B의 모든 인스턴스가 또한 A의 인스턴스인 경우이어야 한다라는 식으로 정의된다. 이것을 표현하는 다른 방법은 B에 따르는 모든 인스턴스는 A에 따라야 한다는 것이다. 만일, 예를 들어 A가 String(문자열) 타입의 Name(이름) 속성을 갖는 반면에 B는 Int16 타입의 Age(나이) 속성을 갖는다면, B의 임의의 인스턴스는 이름과 나이 양쪽 모두를 가져야 한다. 타입 계층 구성은 루트에 단 하나의 슈퍼타입을 갖는 트리로서 파악될 수 있다. 루트로부터의 브랜치들(branches)은 제1 레벨 서브타입들을 제공하고, 이 레벨에서의 브랜치들은 제2 레벨 서브타입들을 제공하는 등등으로 그들 자신이 어떠한 서브타입도 갖고 있지 않은 가장 먼 리프(leaf-most) 서브타입들까지 진행된다. 트리는 균일한 깊이가 되도록 구속되지 않지만 어떠한 사이클도 포함할 수 없다. 주어진 타입은 0 또는 다수의 서브타입들과 0 또는 하나의 슈퍼타입을 가질 수 있다. 주어진 인스턴스는 기껏해야 하나의 타입과 그 타입의 슈퍼타입들에 따를 수 있다. 바꾸어 말하면, 트리 내의 임의의 레벨에서의 주어진 인스턴스에 있어서 그 인스턴스는 그 레벨에서 기껏해야 하나의 서브타입에 따를 수 있다. 타입의 인스턴스들이 또한 그 타입의 서브타입의 인스턴스가 되어야 한다면 그 타입은 Abstract라고 한다.

1. 아이템

아이템은 간단한 파일과 달리, 저장 플랫폼에 의해 최종 사용자 또는 애플리케이션 프로그램에게 노출되는 모든 객체들에 걸쳐서 공통으로 지원되는 속성들의 기본 세트를 갖는 객체인, 저장 가능한 정보의 단위이다. 아이템은 또한 아래에서 논의되는 바와 같이, 새로운 속성들 및 관계들이 도입될 수 있게 하는 특징들을 포함하는 모든 아이템 타입들에 걸쳐서 공통으로 지원되는 속성들 및 관계들을 갖는다.

아이템들은 복사, 삭제, 이동, 열기, 인쇄, 백업, 복원, 복제 등과 같은 통상의 동작들을 위한 객체들이다. 아이템들은 저장되고 검색될 수 있는 단위들이고 저장 플랫폼에 의해 조작 처리되는 모든 형태의 저장 가능한 정보는 아이템, 아이템의 속성, 또는 아이템들 간의 관계로서 존재하고, 이들 각각에 대해서는 아래에서 더 상세히 논의된다.

아이템들은 콘택트(Contacts), 사람, 서비스, 장소, (모든 각종의) 문서 등과 같이 데이터의 실재 세계의 쉽게 이해할 수 있는 단위들을 나타내기 위한 것이다. 도 5A는 아이템의 구조를 예시하는 블록도이다. 아이템의 부적당한 이름은 "Location"이다. 아이템의 적당한 이름은 "Core.Location"이고 이것은 이 아이템 구조가 코어 스키마(Core Schema) 내의 특정 타입의 아이템으로서 정의되는 것을 나타낸다. (코어 스키마에 대해서는 나중에 더 상세히 논의된다.)

Location 아이템은 EAddresses, MetropolitanRegion, Neighborhood, 및 PostalAddress를 포함하는 복수의 속성들을 갖는다. 각각에 대한 속성의 특정 타입은 속성 이름의 바로 다음에 표시되고 콜론(":")에 의해 속성 이름과 분리된다. 타입 이름의 우측에는, 해당 속성 타입에 대해 허용된 값들의 수가 각괄호들("[]") 사이에 표시되고 콜론(":") 우측의 별표("*")는 불특정된 및/또는 무제한의 수("다수")를 나타낸다. 콜론 우측의 "1"는 기껏해야 하나의 값이 있을 수 있음을 나타낸다. 콜론 좌측의 "0"은 속성이 선택 사양임을(전혀 값이 없을 수 있음을) 나타낸다. 콜론 좌측의 "1"은 적어도 하나의 값이 있어야 함을(속성이 요구됨을) 나타낸다. Neighborhood 및 MetropolitanRegion은 둘 다 미리 정의된 데이터 타입이거나 "단순 타입"(simple type)(여기에서는 대문자를 사용하지 않고서 표시됨)인 "nvarchar" 타입(또는 등가물)이다. 그러나, EAddresses 및 PostalAddresses는 각각 EAddress 및 PostAddress 타입의 정의된 타입이거나 또는 "복합 타입"(complex types")(여기에서는 대문자를 사용하여 표시됨)의 속성들이다. 복합 타입은 하나 이상의 단순 데이터 타입들로부터 및/또는 다른 복합 타입들로부터 도출되는 타입이다. 아이템의 속성들에 대한 복합 타입들은 또한 "네스트된 엘리먼트들"(nested elements)을 구성한다. 왜냐하면 복합 타입의 상세들은 그것의 속성들을 정의하기 위해 순간 아이템(immediate Item) 내에 네스트되고, 이들 복합 타입들에 관한 정보는 이들 속성들을 갖는 아이템과 함께(나중에 논의되는 바와 같이, 아이템의 경계 내에) 유지되기 때문이다. 타이핑(typing)의 이 개념들은 잘 알려져 있고 숙련된 당업자들에 의해 쉽게 이해된다.

도 5B는 복합 속성 타입 PostalAddress 및 EAddress를 예시하는 블록도이다. PostalAddress 속성 타입은 속성 타입 Postal Address의 아이템이 0 또는 하나의 City 값, 0 또는 하나의 CountryCode 값, 0 또는 하나의 MailStop 값, 및 임의의 수(0 내지 다수)의 PostalAddressType들 등등을 가질 것으로 예상될 수 있음을 정의한다. 이런 식으로, 아이템 내의 특정 속성에 대한 데이터의 형상이 이로써 정의된다. EAddress 속성 타입은 도시된 바와 같이 유사하게 정의된다. 비록 이 애플리케이션에서는 선택적으로 사용되지만, Location 아이템 내의 복합 타입들을 표현하는 다른 방법은 거기에 리스트된 각각의 복합 타입의 개개의 속성들과 함께 아이템을 그리는 것이다. 도 5C는 Location 아이템을 예시하는 것으로 그것의 복합 타입들이 더 설명되어 있는 블록도이다. 그러나, 이 도 5C에서의 Location 아이템에 대한 이 대안적인 표현은 도 5A에 예시된 것과 동일한 아이템에 대한 것임을 이해해야 한다. 본 발명의 저장 플랫폼은 또한 서브타이핑(subtyping)을 허용하고, 그에 따라서 하나의 속성 타입이 다른 것의 서브타입이 될 수 있다(그 하나의 속성 타입은 다른, 부모 속성 타입의 속성들을 상속한다).

속성들 및 그들의 속성 타입들과 유사하지만 다르게, 아이템들은 서브타입의 주제일 수도 있는 그들 자신의 아이템 타입들을 고유하게 표현한다. 바꾸어 말하면, 본 발명의 몇몇 실시예들에서의 저장 플랫폼은 하나의 아이템이 다른 아이템의 서브타입이 되도록 허용한다(그에 따라서 그 하나의 아이템이 다른, 부모 아이템의 속성들을 상속한다). 또한, 본 발명의 다양한 실시예들에서, 모든 아이템은 베이스 스키마에서 발견되는 제1 및 기본적인 아이템 타입인 "Item" 아이템 타입의 서브타입이다. (베이스 스키마에 대해서는 또한 나중에 상세히 논의될 것이다.) 도 6A는 아이템, 즉 이 인스턴스 내의 Location 아이템을, 베이스 스키마 내에서 발견되는 Item 아이템 타입의 서브타입인 것으로 예시한다. 이 도면에서, 화살표는 (모든 다른 아이템들처럼) Location 아이템이 Item 아이템 타입의 서브타입임을 나타낸다. Item 아이템 타입은, 모든 다른 아이템들이 그로부터 도출되는 기본적인 아이템으로서, ItemId 및 각종 타임스탬프들과 같은 다수의 중요한 속성들을 갖고, 그에 의하여 운영 시스템 내의 모든 아이템들의 표준 속성들을 정의한다. 본 도면에서, Item 아이템 타입의 이들 속성들은 Location에 의해 상속되고 그에 의하여 Location의 속성들이 된다.

Item 아이템 타입으로부터 상속된 Location 아이템 내의 속성들을 표현하는 다른 방법은 거기에 리스트된 부모 아이템으로부터의 각각의 속성 타입의 개개의 속성들과 함께 Location을 그리는 것이다. 도 6B는 Location 아이템을 예시하는 것으로 그것의 순간 속성들(immediate properties) 외에 그것의 상속 타입들이 설명되어 있는 블록도이다. 이 아이템은 도 5A에 예시된 것과 동일한 아이템임을 유의하고 이해해야 한다. 다만, 본 도면에서는 Location이 그것의 모든 속성들, 이 도면 및 도 5A에 도시된 즉시 속성들, 및 이 도면에는 도시되어 있지만 도 5A에는 도시되지 않은 상속 속성들(도 5A에서는 이들 속성들이 Location 아이템이 Item 아이템 타입의 서브타입임을 화살표로 도시함으로써 참조된다)과 함께 예시되어 있다.

아이템들은 독립 실행형 객체들(stand-alone objects)이고, 따라서, 아이템을 삭제하면, 그 아이템의 모든 순간 및 상속 속성들이 또한 삭제된다. 마찬가지로, 아이템을 검색할 때, 수신되는 것은 아이템 및 그것의 모든 순간 및 상속 속성들(그것의 복합 속성 타입들에 관한 정보를 포함)이다. 본 발명의 어떤 실시예들은 특정 아이템을 검색할 때 속성들의 서브세트

를 요구할 수 있게 하지만, 다수의 그러한 실시예들에 대한 디폴트는 검색될 때 그것의 모든 순간 및 상속 속성들과 함께 아이템을 제공하는 것이다. 또한, 아이템들의 속성들은 해당 아이템의 타입의 기존 속성들에 새로운 속성들을 부가함으로써 확장될 수도 있다. 이들 "확장"(extensions)은 그 후에 아이템의 진실한 속성들이고 해당 아이템 타입의 서브타입들은 자동적으로 확장 속성들을 포함할 것이다.

아이템의 "경계"(boundary)는 그것의 속성들(복합 속성 타입, 확장 등을 포함)에 의해 표현된다. 아이템의 경계는 또한 복사, 삭제, 이동, 생성 등과 같은 아이템에 대해 행해지는 동작의 한계를 표현한다. 예를 들면, 본 발명의 몇몇 실시예에서, 아이템이 복사될 때, 해당 아이템의 경계 내의 모든 것이 또한 복사된다. 각각의 아이템에 대하여, 경계는 다음의 것들을 포함한다:

- 아이템의 아이템 타입 및, 그 아이템이 다른 아이템의 서브타입이면(모든 아이템들이 베이스 스키마 내의 단 하나의 아이템 및 아이템 타입으로부터 도출되는 본 발명의 몇몇 실시예의 경우와 같이), 임의의 적용 가능한 서브타입 정보(즉, 부모 아이템 타입에 관한 정보). 만일 복사되고 있는 원본 아이템이 다른 아이템의 서브타입이면, 그 복사본도 그 동일한 아이템의 서브타입일 수 있다.

- 아이템의 복합 타입 속성들 및 확장들(만일 있다면). 만일 원본 아이템이 복합 타입들의 속성들(원시(native) 또는 확장)을 갖고 있다면, 그 복사본도 동일한 복합 타입들을 가질 수 있다.

- "소유 관계"(ownership relationships) 상의 아이템의 레코드들, 즉 부모 아이템("소유하는 아이템"(Owning Item))에 의해 어떤 다른 아이템들("타깃 아이템")이 소유되는지에 대한 아이템 자신의 리스트. 이것은 특히 아래에서 더 상세히 논의되는 아이템 폴더, 및 모든 아이템들이 적어도 하나의 아이템 폴더에 속해야 한다고 하는 아래에서 언급되는 규칙과 관련이 있다. 또한, 아래에서 더 상세히 논의되는, 임베드된 아이템들(embedded items)과 관련하여, 임베드된 아이템은 복사, 삭제 등과 같은 동작을 위하여 임베드되어 있는 아이템의 일부로 간주된다.

2. 아이템 식별

아이템들은 ItemID을 갖는 글로벌 아이템 공간 내에서 고유하게 식별된다. Base.Item 타입은 그 아이템에 대한 ID를 저장하는 GUID 타입의 필드 ItemID를 정의한다. 아이템은 데이터 스토어(302)에서 정확히 하나의 ID를 가져야만 한다.

아이템 참조(item reference)는 아이템을 위치 지정하고 식별하기 위한 정보를 포함하는 데이터 구조이다. 데이터 모델에서, 모든 아이템 참조 타입들이 그로부터 도출되는 ItemReference라고 명명된 추상 타입이 정의된다. ItemReference 타입은 Resolve라고 명명된 가상의 메서드를 정의한다. 이 메서드는 참조가 주어졌을 때 아이템을 검색하는 함수를 구현하는, ItemReference의 구상 서브타입들(concrete subtypes)에 의해 무효화된다(overridden). Resolve 메서드는 저장 플랫폼 API 322의 파트로서 호출된다.

ItemIDReference는 ItemReference의 서브타입이다. 그것은 Locator 및 ItemID 필드를 정의한다. Locator 필드는 아이템 도메인을 명명한다(즉, 식별한다). 그것은 Locator의 값을 아이템 도메인으로 리졸브(resolve)할 수 있는 로케이터 레졸루션 메서드에 의해 처리된다. ItemID 필드는 Item ID 타입이다.

ItemPathReference는 Locator 및 Path 필드를 정의하는 ItemReference의 특수화(specialization)이다. Locator 필드는 아이템 도메인을 식별한다. 그것은 Locator의 값을 아이템 도메인으로 리졸브할 수 있는 로케이터 레졸루션 메서드에 의해 처리된다. Path 필드는 Locator에 의해 제공된 아이템 도메인에 루트(root)를 둔 저장 플랫폼 네임스페이스 내의 (상대) 경로를 포함한다.

이런 타입의 참조는 세트 연산(set operation)에서 사용될 수 없다. 이 참조는 일반적으로 경로 레졸루션 처리를 통하여 리졸브되어야 한다. 저장 플랫폼 API(322)의 리졸브 메서드는 이 기능을 제공한다.

위에서 논의된 참조 형태들은 도 11에 예시된 참조 타입 계층 구성을 통하여 표현된다. 이들 타입으로부터 상속하는 부가적인 참조 타입들은 스키마들에서 정의될 수 있다. 그것들은 타깃 필드의 타입으로서 관계 선언에서 사용될 수 있다.

3. 아이템 폴더 및 카테고리

아래에서 더 상세히 논의되겠지만, 아이템들의 그룹들은 아이템 폴더(파일 폴더와 혼동되지 않아야 할 것이다)라고 불리는 특별한 아이템들로 조직된다. 그러나, 대부분의 파일 시스템에서와 달리, 아이템은 2 이상의 아이템 폴더에 속할 수 있고, 그에 따라 하나의 아이템 폴더에서 아이템이 액세스되고 변경될 경우, 이 변경된 아이템은 다른 아이템 폴더로부터 직접 액세스될 수 있다. 본질적으로, 하나의 아이템에 대한 액세스가 서로 다른 아이템 폴더들로부터 발생될 수는 있지만, 실제로 액세스되는 것은 사실상 바로 그 동일한 아이템이다. 그러나, 아이템 폴더는 반드시 그것의 멤버 아이템들 전부를 소유할 필요는 없고, 다만 다른 폴더들과 함께 아이템들을 공동 소유할 수 있고, 그에 따라 아이템 폴더의 삭제로 인해 반드시 아이템이 삭제될 필요는 없다. 그럼에도 불구하고, 본 발명의 몇몇 실시예들에서는, 아이템이 적어도 하나의 아이템 폴더에 속해야 하고 따라서 특정 아이템에 대한 유일한 아이템 폴더가 삭제되면, 일부 실시예들에서, 그 아이템이 자동으로 삭제되거나, 또는 다른 실시예들에서, 그 아이템이 자동으로 디폴트 아이템 폴더(예를 들면, 갖가지 파일-및-폴더 기반 시스템들에서 사용되는 유사하게 명명된 폴더들과 개념적으로 유사한 "TrashCan"(휴지통) 아이템 폴더)의 멤버가 된다.

또한 아래에서 더 상세히 논의되겠지만, 아이템들은 (a) 아이템 타입(또는 타입들), (b) 특정한 순간 또는 상속 속성(또는 속성들), 또는 (c) 아이템 속성에 대응하는 특정 값(또는 값들)과 같은 공통의 기술 특성(common described characteristics)에 기초한 카테고리들에 속할 수도 있다. 예를 들면, 개인의 콘택트 정보(personal contacts information)에 대한 특정 속성들을 포함하는 아이템은 자동으로 콘택트 카테고리에 속할 것이고, 콘택트 정보 속성들을 갖는 어떠한 아이템이든지 마찬가지로 자동으로 이 카테고리에 속할 것이다. 마찬가지로, "New York City"의 값을 갖는 location 속성을 갖는 어떠한 아이템이든지 자동으로 NewYorkCity 카테고리에 속할 것이다.

카테고리들은 아이템 폴더들과는 개념적으로 다르다. 즉, 아이템 폴더들은 상호 관련이 없는(즉, 공통의 기술 특성이 없는) 아이템들을 포함할 수 있는 반면, 카테고리 내의 각각의 아이템은 해당 카테고리에 대해 기술되는 공통의 타입, 속성, 또는 값("공통성")을 갖고, 카테고리 내의 다른 아이템들에 대한 또는 다른 아이템들 사이에서의 그것의 관계에 대한 토대를 형성하는 것이 이러한 공통성이다. 또한, 특정 폴더 내의 아이템의 멤버십은 해당 아이템의 임의의 특정한 특징에 기초하여 강제적이지 않은 반면, 어떤 실시예들에서 하나의 카테고리에 카테고리별로(categorically) 관련된 공통성을 갖는 모든 아이템들은 하드웨어/소프트웨어 인터페이스 시스템 레벨에서 자동으로 해당 카테고리의 멤버가 될 것이다. 개념적으로, 카테고리들은 또한 그 멤버십이 (예컨대 데이터베이스의 컨텍스트에서의) 특정 쿼리의 결과에 기초하는 가상 아이템 폴더들로서 간주될 수 있고, (카테고리의 공통성들에 의해 정의된) 이 쿼리의 조건들을 만족시키는 아이템들은 따라서 카테고리의 멤버십을 포함할 것이다.

도 4는 아이템들, 아이템 폴더들, 및 카테고리들 간의 구조적 관계를 예시한다. 복수의 아이템들(402, 404, 406, 408, 410, 412, 414, 416, 418, 420)은 각종 아이템 폴더들(422, 424, 426, 428, 430)의 멤버들이다. 어떤 아이템들은 2 이상의 아이템 폴더에 속할 수 있다. 예를 들어, 아이템(402)은 아이템 폴더들(422, 424)에 속한다. 어떤 아이템들, 예를 들어, 아이템들(402, 404, 406, 408, 410, 412)은 또한 하나 이상의 카테고리들(432, 434, 436)의 멤버들이고, 다른 아이템들, 예를 들어, 아이템들(414, 416, 418, 420)은 아무런 카테고리에도 속하지 않을 수 있다(비록 이것은 임의의 속성의 소유가 자동으로 카테고리 내의 멤버십을 의미하는 어떤 실시예들에서는 거의 있을 것 같지 않고, 따라서 그러한 실시예에서는 어떠한 카테고리의 멤버도 되지 않기 위하여 아이템이 전혀 특징이 없어야 하겠지만). 폴더들의 계층형 구조와는 대조적으로, 카테고리들 및 아이템 폴더들은 도시된 바와 같이 지향성 그래프와 더 많이 유사한 구조들을 갖는다. 어떤 경우든, 아이템들, 아이템 폴더들, 및 카테고리들은 모두 아이템들이다(비록 서로 다른 아이템 타입들이기는 하지만).

파일들, 폴더들, 및 디렉토리들과는 대조적으로, 본 발명의 아이템들, 아이템 폴더들, 및 카테고리들은 본질상 특징적으로 "물리적"이 아니다. 왜냐하면 그것들은 물리적 컨테이너들의 개념적 등가물을 갖고 있지 않고, 따라서 아이템들은 2 이상의 그러한 로케이션에 존재할 수 있기 때문이다. 카테고리들로 조직될 뿐만 아니라 2 이상의 아이템 폴더 로케이션에 아이템들이 존재할 수 있는 능력은 이 기술 분야에서 현재 이용 가능한 정도를 넘어서, 하드웨어/소프트웨어 인터페이스 레벨에서 향상되고 풍부해진 정도의 데이터 조작 처리 및 저장 구조 능력들을 제공한다

4. 스키마

a) 베이스 스키마(Base Schema)

아이템들의 생성 및 이용에 대한 보편적인 토대를 제공하기 위하여, 본 발명의 저장 플랫폼의 여러 실시예들은 아이템들 및 속성들을 생성하고 조직하기 위한 개념적 프레임워크를 확립하는 베이스 스키마를 포함한다. 이 베이스 스키마는 아이템들 및 속성들의 어떤 특별한 타입들, 및 이 특별한 기본적 타입들의 특징들을 정의하고, 그로부터 서브타입들이 더 도출될 수 있다. 이 베이스 스키마의 이용으로 프로그래머가 아이템들(및 그들 각각의 타입들)과 속성들(및 그들 각각의 타입

들)을 개념적으로 구별할 수 있게 된다. 또한, 모든 아이템들(및 그들의 대응하는 아이템 타입들)이 베이스 스키마 내의 이 기본적 아이템(및 그것의 대응하는 아이템 타입)으로부터 도출되기 때문에, 베이스 스키마는 모든 아이템들이 소유할 수 있는 속성들의 기본적 세트를 제시한다.

도 7에 예시된 바와 같이, 그리고 본 발명의 몇몇 실시예들과 관련하여, 베이스 스키마는 3개의 상위 레벨 타입들: Item, Extension, 및 PropertyBase를 정의한다. 도시된 바와 같이, 아이템 타입은 이 기본적 "Item" 아이템 타입의 속성들에 의해 정의된다. 이와 대조적으로, 상위 레벨 속성 타입 "PropertyBase"는 미리 정의된 속성들을 갖지 않고 단지 그로부터 다른 모든 속성 타입들이 도출되고 그를 통하여 모든 도출된 속성 타입들이 상호 관련되는(단일 특성 타입으로부터 공통으로 도출되는) 앵커(anchor)에 불과하다. Extension 타입 속성들은 그 확장이 어느 아이템을 확장하는지를 정의할 뿐만 아니라 아이템이 다수의 확장들을 가질 수 있도록 하나의 확장과 다른 확장을 구별하는 식별 정보를 정의한다.

ItemFolder는, 아이템으로부터 상속되는 속성들 외에, 그것의 멤버들(만일 있다면)에의 링크를 확립하기 위한 관계(Relationship)를 특징짓는 Item 아이템 타입의 서브타입인 반면, IdentityKey 및 Property는 모두 PropertyBase의 서브타입들이다. 그리고, CategoryRef는 IdentityKey의 서브타입이다.

b) 코어 스키마(Core Schema)

본 발명의 저장 플랫폼의 여러 실시예들은 또한 상위 레벨 아이템 타입 구조에 대한 개념적 프레임워크를 제공하는 코어 스키마를 포함한다. 도 8A는 코어 스키마 내의 아이템들을 예시하는 블록도이고, 도 8B는 코어 스키마 내의 속성 타입들을 예시하는 블록도이다. 파일-및-폴더 기반 시스템들에서 서로 다른 확장자들(*.com, *.exe, *.bat, *.sys 등) 및 다른 그러한 기준(criteria)으로 파일들 사이에 행해진 구별은 코어 시키마의 기능과 유사하다. 아이템 기반 하드웨어/소프트웨어 인터페이스 시스템에서, 코어 스키마는 모든 아이템들을 아이템 기반 하드웨어/소프트웨어 인터페이스 시스템이 이해하고 미리 정해지거나 또는 예측 가능한 방법으로 직접 처리할 수 있는 하나 이상의 코어 스키마 아이템 타입들로 직접적으로(아이템 타입에 의해) 또는 간접적으로(아이템 서브타입에 의해) 특징화하는 코어 아이템 타입들의 세트를 정의한다. 미리 정의된 아이템 타입들은 아이템 기반 하드웨어/소프트웨어 시스템 내의 가장 공통적인 아이템들을 반영하고 따라서 코어 스키마를 포함하는 이들 미리 정의된 아이템 타입들을 이해하는 아이템 기반 하드웨어/소프트웨어 인터페이스 시스템에 의해 어느 정도의 효율성이 얻어진다.

어떤 실시예들에서는, 코어 스키마는 확장 가능하지 않다. 즉, 코어 스키마의 일부인 특정한 미리 정의되고 도출된 아이템 타입들을 제외하고 베이스 스키마 내의 아이템 타입으로부터 어떠한 부가적인 아이템 타입들도 서브타이핑될 수 없다. 코어 스키마에 대한 확장들을 방지함으로써(즉, 코어 스키마에 새로운 아이템들을 부가하는 것을 방지함으로써), 저장 플랫폼은 코어 스키마 타입들을 사용을 명령한다. 왜냐하면 모든 후속의 아이템 타입이 반드시 코어 스키마 아이템 타입의 서브타입은 아니기 때문이다. 이러한 구조는 부가적인 아이템 타입들을 정의하는 데 있어서 합당한 정도의 유연성을 가능케 하는 한편으로, 또한 코어 아이템 타입들의 미리 정의된 세트를 갖는 이점들을 보존한다.

본 발명의 여러 실시예들에서, 그리고 도 8A를 참조하여, 코어 스키마에 의해 지원되는 특정한 아이템 타입들은 다음의 것들 중 하나 이상을 포함할 수 있다:

- Categories(카테고리): 이 아이템 타입(및 그로부터 도출된 서브타입들)의 아이템들은 아이템 기반 하드웨어/소프트웨어 인터페이스 시스템 내의 유효한 카테고리들을 나타낸다.

- Commodities(상품): 가치 있는 식별 가능한 것들인 아이템들.

- Devices(디바이스) : 정보 처리 능력들을 지원하는 논리적 구조를 갖는 아이템들.

- Documents(문서): 아이템 기반 하드웨어/소프트웨어 인터페이스 시스템에 의해 번역(interprete)되지 않지만 대신에 이 문서 타입에 대응하는 애플리케이션 프로그램에 의해 번역되는 아이템들.

- Events(이벤트): 환경 내의 어떤 사건들(occurrences)을 기록하는 아이템들.

- Locations(장소): 물리적 위치들(예를 들면, 지리적 위치들)을 나타내는 아이템들.

- Messages(메시지): (아래에서 정의된) 2 이상의 주체(principal)들 사이의 통신 아이템들.

- Principals(주체): ItemId(예를 들면, 사람, 조직, 그룹, 가족, 공기관(authority), 서비스 등의 식별 정보)와는 별도로 적어도 하나의 한정적으로(definitively) 입증할 수 있는 ID를 갖는 아이템들.

- Statements(스테이트먼트): 정책(policies), 가입(subscriptions), 자격 증명(credentials) 등을 포함하지만 이들에 한정되지 않는 환경에 관한 특별한 정보를 갖는 아이템들.

마찬가지로, 그리고 도 8B를 참조하여, 코어 스키마에 의해 지원되는 특정 속성 타입들은 다음의 것들 중 하나 이상을 포함할 수 있다:

- (베이스 스키마 내의 기본적 PropertyBase 타입으로부터 도출된) Certificates.

- (베이스 스키마 내의 IdentityKey 타입으로부터 도출된) Principal Identity Keys.

- (베이스 스키마 내의 Property 타입으로부터 도출된) Postal Address.

- (베이스 스키마 내의 Property 타입으로부터 도출된) Rich Text.

- (베이스 스키마 내의 Property 타입으로부터 도출된) EAdress.

- (베이스 스키마 내의 Relationship 타입으로부터 도출된) IdentitySecurityPackage.

- (베이스 스키마 내의 Relationship 타입으로부터 도출된) RoleOccupancy.

- (베이스 스키마 내의 Relationship 타입으로부터 도출된) BasicPresence.

이들 아이템들 및 속성들은 도 8A 및 도 8B에서 제시된 그들 각각의 속성들에 의해 더 설명된다.

5. 관계(Relationships)

관계들은 하나의 아이템이 소스로 지정되고 다른 아이템이 타깃으로 지정되는 바이너리 관계들이다. 소스 아이템 및 타깃 아이템은 관계에 의해 관련된다. 소스 아이템은 일반적으로 관계의 수명(life-time)을 제어한다. 즉, 소스 아이템이 삭제되면, 아이템들 간의 관계도 삭제된다.

관계들은 컨테인먼트(Containment) 및 참조(Reference) 관계들로 분류된다. 컨테인먼트 관계들은 타깃 아이템들의 수명을 제어하는 반면, 참조 관계들은 어떠한 수명 관리 의미론(semantics)도 제공하지 않는다. 도 12는 관계들이 분류되는 방식을 예시한다.

컨테인먼트 관계 타입들은 홀딩(Holding) 및 임베딩(Embedding) 관계들로 분류된다. 하나의 아이템에 대한 모든 홀딩 관계들이 제거되면, 그 아이템은 삭제된다. 홀딩 관계는 참조 카운팅 메커니즘을 통하여 타깃의 수명을 제어한다. 임베딩 관계들은 복합 아이템들의 모델링을 가능케 하고 배타적인(exclusive) 홀딩 관계들로 간주될 수 있다. 아이템은 하나 이상의 홀딩 관계의 타깃이 될 수 있지만, 아이템은 정확히 하나의 임베딩 관계의 타깃이 될 수 있다. 임베딩 관계의 타깃인 아이템은 임의의 다른 홀딩 및 임베딩 관계들의 타깃이 될 수 없다.

참조 관계들은 타깃 아이템의 수명을 제어하지 않는다. 그것들은 댕글링(dangling)으로 될 수 있다. 즉, 타깃 아이템이 존재하지 않을 수 있다. 참조 관계들은 (즉, 원격 데이터 스토어들을 포함하는) 글로벌 아이템 네임스페이스 내의 어디에서도 아이템들에 대한 참조들을 모델링하기 위해 사용될 수 있다.

아이템을 페칭(fetching)함으로써 그것의 관계들이 자동으로 페칭되지 않는다. 애플리케이션들은 아이템의 관계들을 명시적으로 요구해야 한다. 게다가, 관계들을 수정함으로써 타깃 아이템의 소스가 수정되지 않고, 마찬가지로, 관계를 부가함으로써 소스/타깃 아이템이 영향을 받지는 않는다.

a) 관계 선언(Relationship Declaration)

명시적 관계 타입들은 다음의 엘리먼트들로 정의된다:

- Name 특성(attribute)에서 관계 이름이 특정된다.

- 관계 타입과, 홀딩, 임베딩, 참조 중 하나. 이것은 Type 특성에서 특정된다.

- 소스 및 타깃 종점들(endpoints). 각각의 종점은 참조된 아이템의 이름 및 타입을 특정한다.

- 소스 종점 필드는 일반적으로 ItemID 타입이고(선언되지 않음) 그것은 관계 인스턴스로서 동일한 데이터 스토어 내의 아이템을 참조해야 한다.

- 홀딩 및 임베딩 관계들의 경우, 타깃 종점 필드는 ItemIDReference 타입이어야 하고 그것은 관계 인스턴스로서 동일한 스토어 내의 아이템을 참조하여야 한다. 참조 관계들의 경우, 타깃 종점은 임의의 ItemReference 타입일 수 있고 다른 저장 플랫폼 데이터 스토어들 내의 아이템들을 참조할 수 있다.

- 선택 사양으로 스칼라 또는 PropertyBase 타입 중 하나 이상의 필드들이 선언될 수 있다.

- 관계 인스턴스들은 글로벌 관계표(global relationships table)에 저장된다.

- 모든 관계 인스턴스는 조합(소스 ItemID, 관계 ID)에 의해 고유하게 식별된다. 관계 ID는 그들의 타입에 상관없이 주어진 아이템에서 발생(source)된 모든 관계들에 대해 주어진 소스 ItemID 내에서 고유하다.

소스 아이템은 관계의 소유자이다. 소유자로서 지정된 아이템이 관계의 수명을 제어하는 한편, 관계 자체는 그것과 관련이 있는 아이템들로부터 분리된다. 저장 플랫폼 API(322)는 아이템과 관련된 관계들을 노출시키기 위한 메커니즘들을 제공한다.

여기에 관계 선언의 일례가 있다:

```
<Relationship Name="Employment" BaseType="Reference" >
        <Source Name="Employee" ItemType="Contact.Person"/>
        <Target Name="Employer" ItemType="Contact.Organization"
          ReferenceType="ItemIDReference" />
        <Property Name="StartDate" Type="the storage
        platformTypes.DateTime" />
        <Property Name="EndDate" Type="the storage
        platformTypes.DateTime" />
        <Property Name="Office" Type="the storage
        platformTypes.DateTime" />
</Relationship>
```

이것은 참조 관계의 일례이다. 소스 참조에 의해 참조되는 person 아이템이 존재하지 않으면 이 관계는 생성될 수 없다. 또한, person 아이템이 삭제되면, 사람(person)과 조직(organization) 간의 관계 인스턴스들은 삭제된다. 그러나, Organization 아이템이 삭제되면, 관계는 삭제되지 않고 그것은 댕글링 상태로 된다.

b) 홀딩 관계(Holding Relationships)

홀딩 관계들은 타깃 아이템들의 참조 카운트 기반 수명 관리(reference count based life-time management)를 모델링하기 위해 사용된다.

아이템은 아이템들에 대한 0 이상의 관계들에 대한 소스 종점일 수 있다. 임베드된 아이템(embedded Item)이 아닌 아이템은 하나 이상의 홀딩 관계들에서의 타깃이 될 수 있다.

타깃 종점 참조 타입은 ItemIDReference이어야 하고 그것은 관계 인스턴스로서 동일한 스토어 내의 아이템을 참조하여야 한다.

홀딩 관계들은 타깃 종점의 수명 관리를 시행(enforce)한다. 홀딩 관계 인스턴스 및 그것이 타깃으로 하는 아이템의 생성은 원자 동작(atomic operation)이다. 동일한 아이템을 타깃으로 하는 부가적인 홀딩 관계 인스턴스들이 생성될 수 있다. 주어진 아이템을 타깃 종점으로 하는 마지막 홀딩 관계 인스턴스가 삭제되면 타깃 아이템도 삭제된다.

관계 선언에서 특정된 종점 아이템들의 타입들은 일반적으로 관계의 인스턴스가 생성될 때 시행될 것이다. 종점 아이템들의 타입들은 관계가 확립된 후에는 변경될 수 없다.

홀딩 관계들은 아이템 네임스페이스를 형성하는 데 중요한 역할을 한다. 그것들은 소스 아이템에 상대적인 타깃 아이템의 이름을 정의하는 "Name" 속성을 포함한다. 이 상대 이름은 주어진 아이템으로부터 발생된 모든 홀딩 관계들에 대해 고유하다. 루트 아이템에서 시작하여 주어진 아이템까지 이 상대 이름들의 순서화된 리스트는 그 아이템에 대한 전체 이름(full name)을 형성한다.

홀딩 관계들은 지향성 비순환 그래프(directed acyclic graph: DAG)를 형성한다. 홀딩 관계가 생성되면 시스템은 사이클이 생성되지 않도록 보장하고, 따라서 아이템 네임스페이스가 DAG를 형성하도록 보장한다.

홀딩 관계는 타깃 아이템의 수명을 제어하지만, 그것은 타깃 종점 아이템의 동작의 일관성을 제어하지는 않는다. 타깃 아이템은 홀딩 관계를 통하여 그것을 소유하는 아이템으로부터 동작상으로 독립적이다. 홀딩 관계의 소스인 아이템에 대한 복사, 이동, 백업 및 다른 동작들은 동일한 관계의 타깃인 아이템에 영향을 미치지 않는다. 예를 들면, 즉 폴더 아이템을 백업함으로써 폴더 내의 모든 아이템들(FolderMember 관계의 타깃들)이 자동으로 백업되지는 않는다.

다음은 홀딩 관계의 일례이다:

```
<Relationship Name="FolderMembers" BaseType="Holding" >
        <Source Name="Folder" ItemType="Base.Folder"/>
        <Target Name="Item" ItemType="Base.Item"
          ReferenceType="ItemIDReference" />
</Relationship>
```

FolderMembers 관계는 아이템들의 일반적 컬렉션(generic collection)으로서 폴더의 개념을 가능케 한다.

c) 임베딩 관계(Embedding Relationships)

임베딩 관계들은 타깃 아이템의 수명에 대한 배타적 제어의 개념을 모델링한다. 그것들은 복합 아이템들의 개념을 가능케 한다.

임베딩 관계 인스턴스 및 그것이 타깃으로 하는 아이템의 생성은 원자 동작이다. 아이템은 0 또는 그 이상의 임베딩 관계의 소스일 수 있다. 그러나, 아이템은 하나의 및 단 하나의 임베딩 관계의 타깃일 수 있다. 임베딩 관계의 타깃인 아이템은 홀딩 관계의 타깃일 수 없다.

타깃 종점 참조 타입은 ItemIDReference이어야 하고 그것은 관계 인스턴스로서 동일한 데이터 스토어 내의 아이템을 참조해야 한다.

관계 선언에서 특정된 종점 아이템들의 타입들은 일반적으로 관계의 인스턴스가 생성될 때 시행될 것이다. 종점 아이템들의 타입들은 관계가 확립된 후에는 변경될 수 없다.

임베딩 관계들은 타깃 종점의 동작적인 일관성을 제어할 수 있다. 예를 들면 아이템의 직렬화(serializing)의 동작은 해당 아이템뿐만 아니라 그들의 타깃들 모두로부터 발생하는 모든 임베딩 관계들의 직렬화(serialization)를 포함할 수 있고, 아이템을 복사하는 것은 또한 모든 그것의 임베드된 아이템들을 복사한다.

다음은 선언의 일례이다:

```
<Relationship Name="ArchiveMembers" BaseType="Embedding" >
        <Source Name="Archive" ItemType="Zip.Archive" />
        <Target Name="Member" ItemType="Base.Item "
          ReferenceType="ItemIDReference" />
        <Property Name="ZipSize" Type="the storage
        platformTypes.bigint" />
        <Property Name="SizeReduction" Type="the storage
        platformTypes.float" />
</Relationship>
```

d) 참조 관계(Reference Relationships)

참조 관계는 그것이 참조하는 아이템의 수명을 제어하지 않는다. 심지어, 참조 관계들은 타깃의 존재를 보증하지도 않고, 관계 선언에서 특정된 대로 타깃의 타입을 보증하지도 않는다. 이것은 참조 관계들이 댕글링으로 될 수 있다는 것을 의미한다. 또한, 참조 관계는 다른 데이터 스토어들 내의 아이템들을 참조할 수 있다. 참조 관계들은 웹 페이지들에서의 링크들과 유사한 개념으로 간주될 수 있다.

다음은 참조 관계 선언의 일례이다:

```
<Relationship Name="DocumentAuthor" BaseType="Reference" >
        <Sourc  ItemType="Document"
        ItemType="Base.Document"/>
        <Target ItemType="Author" ItemType="Base.Author"
          ReferenceType="ItemIDReference" />
        <Property Type="Role" Type="Core.CategoryRef" />
        <Property Type="DisplayName" Type="the storage
        platformTypes.nvarchar(256)" />
</Relationship>
```

타깃 종점에는 임의의 참조 타입이 허용된다. 참조 관계에 관여하는 아이템들은 임의의 아이템 타입일 수 있다.

참조 관계들은 아이템들 간의 대부분의 비수명 관리 관계들(non-lifetime management relationships)을 모델링하기 위해 사용된다. 타깃의 존재는 시행되지 않으므로, 참조 관계는 느슨하게 연결된 관계들(loosely-coupled relationships)을 모델링하기에 편리하다. 참조 관계는 다른 컴퓨터 상의 스토어들을 포함한 다른 데이터 스토어들 내의 아이템들을 타깃팅하기 위해 사용될 수 있다.

e) 규칙 및 제약(Rules and Constraints)

다음의 부가적인 규칙들 및 제약들이 관계들에 적용된다:

- 아이템은 (정확히 하나의 임베딩 관계) 또는 (하나 이상의 홀딩 관계들)의 타깃이어야 한다. 하나의 예외는 루트 아이템이다. 아이템은 0 이상의 참조 관계들의 타깃일 수 있다.

- 임베딩 관계의 타깃인 아이템은 홀딩 관계들의 소스일 수 없다. 그것은 참조 관계들의 소스일 수 있다.

- 아이템은 그것이 파일로부터 프로모트(promote)되면 홀딩 관계의 소스일 수 없다. 그것은 임베딩 관계 및 참조 관계들의 소스일 수 있다.

- 파일로부터 프로모트되는 아이템은 임베딩 관계의 타깃일 수 없다.

f) 관계들의 순서화(Ordering of Relationships)

적어도 하나의 실시예에서, 본 발명의 저장 플랫폼은 관계들의 순서화를 지원한다. 이 순서화는 베이스 관계 정의에서 "Order"라고 명명된 속성을 통하여 성취된다. Order 필드에는 어떠한 고유성 제약(uniqueness constraint)도 없다. 동일한 "order"(순서) 속성 값을 갖는 관계들의 순서는 보증되지 않지만, 그것들은 보다 낮은 "order" 값을 갖는 관계들 뒤에 그리고 보다 높은 "order" 필드 값을 갖는 관계들 앞에 순서화될 수 있다는 것은 보증된다.

애플리케이션들은 조합(SourceItemID, RelationshipID, Order)에 기초하여 순서화함으로써 디폴트 순서의 관계들을 얻을 수 있다. 주어진 아이템으로부터 발생된 모든 관계 인스턴스들은 컬렉션 내의 관계들의 타입에 상관없이 단일 컬렉션으로서 순서화된다. 그러나 이것은 주어진 타입의 모든 관계들(예를 들면, FolderMembers)이 주어진 아이템에 대한 관계 컬렉션의 순서화된 서브세트임을 보증한다.

관계들을 조작 처리하기 위한 데이터 스토어 API(312)는 관계들의 순서화를 지원하는 동작들의 세트를 구현한다. 다음의 용어들이 그 동작들의 설명을 돕기 위해 도입된다:

- *RelFirst*는 순서 값 *OrdFirst*를 갖는 순서화된 컬렉션 내의 첫 번째 관계이다.
- *RelLast*는 순서 값 *OrdLast*를 갖는 순서화된 컬렉션 내의 마지막 관계이다.
- *RelX*는 순서 값 *OrdX*를 갖는 컬렉션 내의 주어진 관계이다.
- RelPrev는 OrdX보다 작은 순서 값 OrdPrev를 갖는 RelX에 가장 가까운 컬렉션 내의 관계이다.
- *RelNext*는 *OrdX*보다 큰 순서 값 *OrdNext*를 갖는 *RelX*에 가장 가까운 컬렉션 내의 관계이다.

동작들은 다음을 포함하지만 이들에 제한되지는 않는다:

- *InsertBeforeFirst*(*SourceItemID*, *Relationship*)은 컬렉션 내의 첫 번째 관계로서 관계를 삽입한다. 새로운 관계의 "Order" 속성의 값은 *OrdFirst*보다 작을 수 있다.
- *InsertAfterLast*(*SourceItemID*, *Relationship*)는 컬렉션 내의 마지막 관계로서 관계를 삽입한다. 새로운 관계의 "Order" 속성의 값은 OrdLast보다 클 수 있다.
- *InsertAt*(*SourceItemID*, *ord*, *Relationship*)는 "Order" 속성에 대한 특정된 값을 갖는 관계를 삽입한다.
- *InsertBefore*(*SourceItemID*, *ord*, *Relationship*)는 주어진 순서 값을 갖는 관계 앞에 관계를 삽입한다. 새로운 관계에는 OrdPrev와 ord를 포함하지 않고 이들 사이에 있는 "Order" 값이 할당된다.
- *InsertAfter*(*SourceItemID*, *ord*, *Relationship*)는 주어진 순서 값을 갖는 관계 뒤에 관계를 삽입한다. 새로운 관계에는 ord와 OrdNext를 포함하지 않고 이들 사이에 있는 "Order" 값이 할당된다.
- *MoveBefore*(*SourceItemID*, *ord*, *RelationshipID*)는 주어진 관계 ID를 갖는 관계를 특정된 "Order" 값을 갖는 관계 앞에 이동시킨다. 이 관계에는 OrdPrev와 ord를 포함하지 않고 이들 사이에 있는 새로운 "Order" 값이 할당된다.
- *MoveAfter*(*SourceItemID*, *ord*, *RelationshipID*)는 주어진 관계 ID를 갖는 관계를 특정된 "Order" 값을 갖는 관계 뒤에 이동시킨다. 이 관계에는 ord와 OrdNext를 포함하지 않고 이들 사이에 있는 새로운 순서 값이 할당된다.

앞에서 언급한 바와 같이, 모든 아이템은 아이템 폴더의 멤버이어야 한다. 관계(Relationships)의 관점에서, 모든 아이템은 아이템 폴더와 관계를 가져야 한다. 본 발명의 몇몇 실시예에서, 어떤 관계들은 아이템들 간에 존재하는 관계들에 의해 표현된다.

본 발명의 다양한 실시예들에서 구현될 때, 관계(Relationship)는 하나의 아이템(소스)에 의해 또 하나의 아이템(타깃)으로 "확장"되는 지향성 바이너리 관계(directed binary 관계)를 제공한다. 관계는 소스 아이템(그것을 확장한 아이템)에 의해 소유되고, 따라서 소스가 제거되면 관계는 제거된다(예를 들면, 관계는 소스 아이템이 삭제될 때 삭제된다). 더욱이, 어떤 경우에, 관계는 (공동 소유의) 타깃 아이템의 소유권을 공유하고, 그러한 소유권은 (관계 속성 타입에 대하여 도 7에 도시된 바와 같이) 관계의 IsOwned 속성(또는 그것의 등가물)에 반영될 것이다. 이들 실시예에서, 새로운 IsOwned 관계의 생성은 타깃 아이템 상의 참조 카운트를 자동으로 증가시키고, 그러한 관계의 삭제는 타깃 아이템 상의 참조 카운트를 감소시킬 것이다. 이들 특정 실시예들에서, 아이템들은 0보다 큰 참조 카운트를 갖는다면 계속해서 존재하고, 그 카운트가 0

에 도달하면 자동으로 삭제된다. 다시, 아이템 폴더는 다른 아이템들(이들 다른 아이템들은 아이템 폴더의 멤버십을 포함함)에 대한 관계들의 세트를 갖는(또는 가질 수 있는) 아이템이다. 여기에서 설명된 기능을 성취하기 위해 본 발명에 의해 Relationship들의 다른 실제의 구현들이 가능하고 예상된다.

실제의 구현들에 상관없이, 관계는 하나의 객체로부터 다른 객체로의 선택 가능한 연결이다. 하나의 아이템이 2 이상의 아이템 폴더에는 물론, 하나 이상의 카테고리에 속할 수 있는 능력은, 이들 아이템들, 폴더들, 및 카테고리들이 공용(public)이든 전용(private)이든, 아이템 기반 구조에서 존재(또는 그것의 결여)에 주어진 의미들에 의해 결정된다. 이들 논리적 관계들은, 물리적 구현에 상관없이, 여기에서 설명된 기능을 성취하기 위해 구체적으로 이용되는, 관계들의 세트에 할당된 의미들이다. 논리적 관계들은 아이템과 그것의 폴더(들) 또는 카테고리들(혹은 반대로) 사이에 확립된다. 왜냐하면, 본질적으로, 아이템 폴더들 및 카테고리들은 각각 특별한 타입의 아이템이기 때문이다. 따라서, 아이템 폴더들 및 카테고리들은 임의의 다른 아이템과 동일하게 작용될 수 있고 - 제한 없이 복사되고, 이메일 메시지에 부가되고, 문서 내에 임베드되는 등등 - 아이템 폴더들 및 카테고리들은 다른 아이템들에 대해서와 동일한 메커니즘을 이용하여 직렬화(serialize) 및 비직렬화(de-serialize)될(가져오기(import) 및 내보내기(export)될) 수 있다. (예를 들면, XML에서 모든 아이템들은 직렬화 포맷을 가질 수 있고, 이 포맷은 아이템 폴더들, 카테고리들, 및 아이템들에 동등하게 적용된다.)

아이템과 그 아이템 폴더(들) 간의 관계를 나타내는 전술한 관계들은 아이템으로부터 아이템 폴더로, 또는 아이템 폴더로부터 아이템으로, 또는 양쪽 모두로 논리적으로 확장할 수 있다. 아이템으로부터 아이템 폴더로 논리적으로 확장하는 관계는 그 아이템 폴더가 해당 아이템에 공용이고 그것의 멤버십 정보를 해당 아이템과 공유한다는 것을 나타내고, 반대로, 아이템으로부터 아이템 폴더로의 논리적 관계의 결여는 아이템 폴더가 해당 아이템에 전용이고 그것의 멤버십을 해당 아이템과 공유하지 않는다는 것을 나타낸다. 마찬가지로, 아이템 폴더로부터 아이템으로 논리적으로 확장하는 관계는 그 아이템이 해당 아이템 폴더에 공용이고 공유 가능하다는 것을 나타내는 반면, 아이템 폴더로부터 아이템으로의 논리적 관계의 결여는 그 아이템이 전용이고 공유 불가능하다는 것을 나타낸다. 따라서, 아이템 폴더가 다른 시스템에 내보내기될 때, 그것은 새로운 컨텍스트에서 공유되는 "공용" 아이템들이고, 아이템이 다른 공유 가능한 아이템들에 대한 그것의 아이템 폴더들을 검색할 때, 그것은 거기에 속하는 공유 가능한 아이템들에 관한 정보를 그 아이템에 제공하는 "공용" 아이템 폴더들이다.

도 9는 아이템 폴더(이것은 다시 아이템 자체이기도 하다), 그것의 멤버 아이템들, 및 아이템 폴더와 그것의 멤버 아이템들 간의 상호 연결 관계들을 예시하는 블록도이다. 아이템 폴더(900)는 멤버들로서 복수의 아이템들(902, 904, 906)을 갖는다. 아이템 폴더(900)는 자신으로부터 아이템(902)으로의 관계(912)를 갖고 이것은 그 아이템(902)이 공용이고 아이템 폴더(900), 그것의 멤버들(904, 906), 및 임의의 다른 아이템 폴더들, 카테고리들, 또는 아이템 폴더(900)에 액세스할지도 모르는 아이템들(도시되지 않음)에게 공유 가능하다는 것을 나타낸다. 그러나, 아이템(902)으로부터 아이템 폴더(900)로의 관계는 없고 이것은 아이템 폴더(900)가 아이템(902)에 전용이고 그것의 멤버십 정보를 아이템(902)과 공유하지 않는다는 것을 나타낸다. 한편, 아이템(904)은 자신으로부터 아이템 폴더(900)로의 관계(924)를 갖고 이것은 아이템 폴더(900)가 공용이고 그것의 멤버십 정보를 아이템(904)와 공유한다는 것을 나타낸다. 그러나, 아이템 폴더(900)로부터 아이템(904)으로의 관계는 없고 이것은 아이템(904)이 전용이고 아이템 폴더(900), 그것의 다른 멤버들(902, 906), 및 임의의 다른 아이템 폴더들, 카테고리들, 또는 아이템 폴더(900)에 액세스할지도 모르는 아이템들(도시되지 않음)에게 공유 불가능하다는 것을 나타낸다. 아이템들(902, 904)에 대한 그것의 관계들(또는 그것의 결여)과 대조적으로, 아이템 폴더(900)는 자신으로부터 아이템(906)으로의 관계(916)를 갖고 아이템(906)은 역으로 아이템 폴더(900)로의 관계(926)를 갖고, 이것들은 함께 아이템(906)이 공용이고 아이템 폴더(900), 그것의 멤버들(902, 904), 및 임의의 다른 아이템 폴더들, 카테고리들, 또는 아이템 폴더(900)에 액세스할지도 모르는 아이템들(도시되지 않음)에게 공유 가능하고, 또한 아이템 폴더(900)가 공용이고 그것의 멤버십 정보를 아이템(906)과 공유한다는 것을 나타낸다.

앞에서 논의한 바와 같이, 아이템 폴더 내의 아이템들은 아이템 폴더들이 "기술"(describe)되지 않기 때문에 공통성을 공유할 필요가 없다. 한편, 카테고리들은 그것의 멤버 아이템들 모두에게 공통인 공통성에 의해 기술된다. 따라서 카테고리의 멤버십은 기술된 공통성을 갖는 아이템들에 고유하게 제한되고, 어떤 실시예들에서는, 카테고리의 기술을 만족시키는 모든 아이템들이 자동으로 그 카테고리의 멤버들로 된다. 따라서, 아이템 폴더들은 사소한 타입 구조들이 그들의 멤버십에 의해 표현되도록 허용하는 반면, 카테고리들은 정의된 공통성에 기초하여 멤버십을 허용한다.

물론 카테고리 기술들은 본질상 논리적이고, 따라서 카테고리는 타입들, 속성들, 및/또는 값들의 임의의 논리적 표현에 의해 기술될 수 있다. 예를 들면, 카테고리에 대한 논리적 표현은 2개의 속성들 중 하나 또는 양쪽 모두를 갖는 아이템들을 포함하는 그것의 멤버십일 수 있다. 만일 카테고리에 대한 이들 기술된 속성들이 "A" 및 "B"이면, 카테고리 멤버십은 속성 A는 갖고 B는 갖지 않는 아이템들, 속성 B는 갖고 A는 갖지 않는 아이템들, 속성 A 및 B 양쪽 모두를 갖는 아이템들을 포함

할 수 있다. 속성들에 대한 이 논리적 표현은 논리적 연산자 "OR"에 의해 기술되고 카테고리에 의해 기술된 멤버들의 세트는 속성 A OR B를 갖는 아이템들이다. 숙련된 당업자라면 알겠지만, (제한 없이 "AND", "XOR", 및 "NOT"를 단독으로 또는 조합하여 포함하는) 유사한 논리적 피연산자들이 또한 카테고리를 기술하기 위해 사용될 수 있다.

(기술되지 않은) 아이템 폴더들 및 (기술된) 카테고리들 간의 구별에도 불구하고, 본 발명의 여러 실시예들에서 아이템 폴더들 및 아이템들에 대하여 위에서 개시한 것과 본질적으로 동일하게 카테고리들은 아이템들에 관계하고 아이템들은 카테고리에 관계한다.

도 10은 카테고리(이것은 다시 아이템 자체이기도 하다), 그것의 멤버 아이템들, 및 카테고리와 그것의 멤버 아이템들 간의 상호 연결 관계들을 예시하는 블록도이다. 카테고리(1000)는 멤버들로서 복수의 아이템들(1002, 1004, 1006)을 갖고, 이것들 모두는 카테고리(1000)에 의해 기술된 공통의 속성들, 값들, 또는 타입들(1008)(공통성 기술(1008'))의 어떤 조합을 공유한다. 카테고리(1000)는 자신으로부터 아이템(1002)으로의 관계(1012)를 갖고 이것은 그 아이템(1002)이 공용이고 카테고리(1000), 그것의 멤버들(1004, 1006), 및 임의의 다른 카테고리들, 아이템 폴더들, 또는 카테고리(1000)에 액세스할지도 모르는 아이템들(도시되지 않음)에게 공유 가능하다는 것을 나타낸다. 그러나, 아이템(1002)으로부터 카테고리(1000)로의 관계는 없고 이것은 카테고리(1000)가 아이템(1002)에 전용이고 그것의 멤버십 정보를 아이템(1002)과 공유하지 않는다는 것을 나타낸다. 한편, 아이템(1004)은 자신으로부터 카테고리(1000)로의 관계(1024)를 갖고 이것은 카테고리(1000)가 공용이고 그것의 멤버십 정보를 아이템(1004)와 공유한다는 것을 나타낸다. 그러나, 카테고리(1000)로부터 아이템(1004)으로 확장된 관계는 없고 이것은 아이템(1004)이 전용이고 카테고리(1000), 그것의 다른 멤버들(1002, 1006), 및 임의의 다른 카테고리들, 아이템 폴더들, 또는 카테고리(1000)에 액세스할지도 모르는 아이템들(도시되지 않음)에게 공유 불가능하다는 것을 나타낸다. 아이템들(1002, 1004)과의 그것의 관계들(또는 그것의 결여)과 대조적으로, 카테고리(1000)는 자신으로부터 아이템(1006)으로의 관계(1016)를 갖고 아이템(1006)은 역으로 카테고리(1000)로의 관계(1026)를 갖고, 이것들은 함께 아이템(1006)이 공용이고 카테고리(1000), 그것의 멤버들(1002, 1004), 및 임의의 다른 카테고리들, 아이템 폴더들, 또는 카테고리(1000)에 액세스할지도 모르는 아이템들(도시되지 않음)에게 공유 가능하고, 또한 카테고리(1000)가 공용이고 그것의 멤버십 정보를 아이템(1006)과 공유한다는 것을 나타낸다.

마지막으로, 카테고리들 및 아이템 폴더들은 그들 자신이 아이템들이고, 아이템들은 서로 관계할 수 있기 때문에, 카테고리들이 아이템 폴더들에 관계할 수도 있고 아이템 폴더들이 카테고리들에 관계할 수도 있고, 어떤 다른 실시예들에서 카테고리들, 아이템 폴더들, 및 아이템들은 각각 다른 카테고리들, 아이템 폴더들, 및 아이템들에 관계할 수 있다. 그러나, 여러 실시예들에서, 아이템 폴더 구조들 및/또는 카테고리 구조들은 하드웨어/소프트웨어 인터페이스 시스템 레벨에서 사이클들을 포함하는 것이 금지된다. 아이템 폴더 및 카테고리 구조들이 지향성 그래프들과 유사한 경우에, 사이클들을 금지하는 실시예들은 지향성 비순환 그래프들(DAGs: directed acyclic graphics)과 유사하고, 이것은 그래프 이론의 기술 분야에서의 수학적 정의에 의해 아무런 경로도 시작하지 않고 동일한 정점(vertex)에서 종료하지 않는 지향성 그래프들이다.

6. 확장성(Extensibility)

저장 플랫폼은 상술한 바와 같이 스키마들의 초기 세트(340)를 구비하도록 의도되어 있다. 그러나, 또한, 적어도 일부 실시예들에서, 저장 플랫폼은 독립적 소프트웨어 벤더(ISV들)를 포함하는 고객들이 새로운 스키마들(344)(즉, 새로운 아이템 및 네스트된 엘리먼트 타입들)을 생성하도록 허용한다. 이 섹션은 스키마들의 초기 세트(340)에서 정의된 아이템 타입들 및 네스트된 엘리먼트 타입들(또는 간단히 "엘리먼트" 타입들)을 확장함으로써 그러한 스키마들을 생성하기 위한 메커니즘을 다룬다.

바람직하게는, 아이템 및 네스트된 엘리먼트 타입들의 초기 세트의 확장은 다음과 같이 제약을 받는다:

- ISV는 새로운 아이템 타입들, 즉 서브타입 Base.Item을 도입하도록 허용된다.
- ISV는 새로운 네스트된 엘리먼트 타입들, 즉 서브타입 Base.NestedElement를 도입하도록 허용된다.
- ISV는 새로운 확장들, 즉 서브타입 Base.NestedElement를 도입하도록 허용된다.

그러나,

- ISV는 저장 플랫폼 스키마들의 초기 세트(340)에 의해 정의된 어떠한 타입들(아이템, 네스트된 엘리먼트, 또는 확장 타입)도 서브타이핑할 수 없다.

저장 플랫폼 스키마들의 초기 세트에 의해 정의된 아이템 타입 또는 네스트된 아이템 타입은 ISV 애플리케이션의 요구에 정확히 부합하지 않을 수도 있으므로, ISV들이 타입을 맞춤화(customize)하도록 허용할 필요가 있다. 이것은 확장(Extensions)의 개념으로 허용된다. 확장들은 공고히 타이핑된 인스턴스들이지만 (a) 그것들은 독립적으로 존재할 수 없고 (b) 그것들은 아이템 또는 네스트된 엘리먼트에 첨부되어야 한다.

스키마 확장성에 대한 요구를 다루는 것 외에, 확장들은 또한 "멀티타이핑"(multi-typing) 문제를 다루도록 의도되어 있다. 일부 실시예들에서, 저장 플랫폼은 다수의 상속 또는 중첩 서브타입들을 지원하지 않을 수 있으므로, 애플리케이션들은 중첩 타입 인스턴스들을 모델링하는 방법으로서 확장들을 이용할 수 있다(예를 들면, Document는 안전한 문서일 뿐만 아니라 법적 문서이다).

a) 아이템 확장(Item extensions)

아이템 확장성을 제공하기 위하여, 데이터 모델은 Base.Extension이라고 명명된 추상 타입을 더 정의한다. 이것은 확장 타입들의 계층 구성에 대한 루트 타입이다. 애플리케이션들은 특정 확장 타입들을 생성하기 위해 Base.Extension들을 서브타이핑할 수 있다.

Base.Extension 타입은 베이스 스키마에서 다음과 같이 정의된다.

```
<Type Name="Base.Extension" IsAbstract="True">
        <Propety Name="ItemID"
                  Type="the storage platformTypes.uniqueidentified"
                  Nullable="false"
                  MultiValued="false"/>
        <Property Name="ExtensionID"
                  Type="the storage platformTypes.uniqueidentified"
                  Nullable="false"
                  MultiValued="false"/>
</Type>
```

ItemID 필드는 확장이 관련되어 있는 아이템의 ItemID를 포함한다. 이 ItemID를 갖는 아이템이 존재하여야 한다. 주어진 ItemID를 갖는 아이템이 존재하지 않으면 확장은 생성될 수 없다. 아이템이 삭제되면 동일한 ItemID를 갖는 모든 확장들이 삭제된다. 투플(tuple)(ItemID,ExtensionID)은 확장 인스턴스를 고유하게 정의한다.

확장 타입의 구조는 아이템 타입의 구조와 유사하다:

- 확장 타입들은 필드들을 갖는다.
- 필드들은 프리미티브(primitive) 또는 네스트된 엘리먼트 타입들일 수 있다.
- 확장 타입들은 서브타이핑될 수 있다.

다음의 제한들이 확장 타입들에 적용된다:

- 확장들은 관계들의 소스 및 타깃일 수 없다.
- 확장 타입 인스턴스들은 아이템과 독립하여 존재할 수 없다.
- 확장 타입들은 저장 플랫폼 타입 정의에서 필드 타입들로서 사용될 수 없다.

주어진 아이템 타입과 관련될 수 있는 확장 타입들에는 아무런 제약도 없다. 어떠한 확장 타입이든지 임의의 아이템 타입을 확장하도록 허용된다. 아이템에 다수의 확장 인스턴스들이 첨부되는 경우, 그것들은 구조 및 작용 모두에서 서로 독립적이다.

확장 인스턴스들은 아이템과 별도로 저장되고 액세스된다. 모든 확장 타입 인스턴스들은 글로벌 확장 뷰(global extension view)로부터 액세스 가능하다. 그것들이 관련되어 있는 아이템의 타입이 무엇이든 상관없이 주어진 확장 타입의 모든 인스턴스들을 반환(return)할 효율적인 쿼리가 구성될 수 있다. 저장 플랫폼 API들은 아이템들 상의 확장들을 저장, 검색 및 수정할 수 있는 프로그래밍 모델을 제공한다.

확장 타입들은 저장 플랫폼 단일 상속 모델을 이용하여 서브타이핑된 타입일 수 있다. 확장 타입으로부터 도출함으로써 새로운 확장 타입이 생성된다. 확장의 구조 또는 작용은 아이템 타입 계층 구성의 구조 또는 거동들을 무효화(override)하거나 대체할 수 없다. 아이템 타입들과 마찬가지로, 확장 타입 인스턴스들은 확장 타입과 관련된 뷰를 통하여 직접 액세스될 수 있다. 확장의 ItemID는 그것들이 어느 아이템에 속하는지를 나타내고 글로벌 아이템 뷰로부터 대응하는 아이템 객체를 검색하기 위해 사용될 수 있다. 확장들은 동작의 일관성을 위하여 아이템의 일부로 간주된다. 복사/이동, 백업/복원 및 저장 플랫폼이 정의하는 다른 공통의 동작들은 아이템의 일부로서 확장들에 작용할 수 있다.

다음 예를 생각해보자. Contact 타입이 윈도즈 타입 세트에서 정의된다.

```
<Type Name="Contact" BaseType="Base.Item" >
        <Property Name="Name"
                  Type="String"
                  Nullable="false"
                  MultiValued="false"/>
        <Property Name="Address"
                  Type="Address"
                  Nullable="true"
                  MultiValued="false"/>

</Type>
```

CRM 애플리케이션 개발자는 저장 플랫폼에 저장된 콘택트들에 CRM 애플리케이션 확장을 첨부하기를 원할 것이다. 이 애플리케이션 개발자는 애플리케이션이 조작 처리할 수 있는 부가적인 데이터 구조를 포함할 CRM 확장을 정의할 것이다.

```
<Type Name="CRMExtension" BaseType="Base.Extension" >
        <Property Name="CustomerID"
                  Type="String"
                  Nullable="false"
                  MultiValued="false"/>
        ...
</Type>
```

또한 HR 애플리케이션 개발자가 콘택트와 함께 부가적인 데이터를 첨부하기를 원할 수 있다. 이 데이터는 CRM 애플리케이션 데이터와 독립적이다. 다시 애플리케이션 개발자는 확장을 생성할 수 있다.

```
<Type Name="HRExtension" EBaseType="Base.Extension" >
        <Property Name="EmployeeID"
                  Type="String"
                  Nullable="false"
                  MultiValued="false"/>
        ...
</Type>
```

CRMExtension 및 HRExtension은 Contact 아이템들에 첨부될 수 있는 2개의 독립된 확장들이다. 그것들은 서로 독립적으로 생성되고 액세스될 수 있다.

상기 예에서, CRMExtension 타입의 필드들 및 메서드들은 Contact 계층 구성의 필드들 또는 메서드들을 무효화할 수 없다. CRMExtension 타입의 인스턴스들은 Contact 이외의 아이템 타입들에 첨부될 수 있다는 것에 유의해야 할 것이다.

Contact 아이템이 검색될 때, 그것의 아이템 확장들은 자동으로 검색되지 않는다. Contact 아이템이 주어진 경우, 그것의 관련 아이템 확장들은 동일한 ItemID를 갖는 확장들에 대하여 글로벌 확장 뷰를 쿼리(query)함으로써 액세스될 수 있다.

시스템 내의 모든 CRMExtension 확장들은 그들이 어느 아이템에 속하는지에 상관없이 CRMExtension 타입 뷰를 통하여 액세스될 수 있다. 아이템의 모든 아이템 확장들은 동일한 아이템 id를 공유한다. 상기 예에서, Contact 아이템 인스턴스들 및 첨부된 CRMExtension 및 HRExtension 인스턴스들은 동일한 ItemID를 공유한다.

다음의 표는 아이템(Item), 확장(Extension) 및 NestedElement 타입들 간의 유사점 및 차이점들을 요약한다:

아이템 대 아이템 확장 대 NestedElement

	아이템	아이템 확장	NestedElement
아이템 ID	그 자신의 아이템 id를 가짐	아이템의 아이템 id를 공유함	그 자신의 아이템 id를 갖지 않음. 네스트된 엘리먼트는 아이템의 일부(part)임.
저장	아이템 계층 구성이 그 자신의 표에 저장됨	아이템 확장 계층 구조가 그 자신의 표에 저장됨	아이템과 함께 저장됨
쿼리/검색	아이템 표들을 쿼리할 수 있음	아이템 확장 표들을 쿼리할 수 있음	일반적으로 포함하고 있는 아이템 컨텍스트(containing item context) 내에서만 쿼리될 수 있음
쿼리/검색 범위	아이템 타입의 모든 인스턴스들에 걸쳐서 검색할 수 있음	아이템 확장 타입의 모든 인스턴스들에 걸쳐서 검색할 수 있음	일반적으로 단일의 (포함하고 있는) 아이템의 네스트된 엘리먼트 타입 내에서만 검색할 수 있음
관계 의미 (Relationship semantics)	아이템들에의 관계들을 가질 수 있음	아이템 확장들에의 관계가 없음	네스트된 엘리먼트들에의 관계가 없음
아이템들에의 관련	홀딩, 임베드된(embedded) 및 소프트한 관계들을 통하여 다른 아이템들에 관련될 수 있음	일반적으로 확장들을 통해서만 관련될 수 있음. 확장 의미는 임베드된 아이템 의미와 유사함	필드들을 통하여 아이템에 관련됨. 네스트된 아이템들은 아이템의 일부임

b) NestedElement 타입 확장

네스트된 엘리먼트 타입들은 아이템 타입들과 동일한 메커니즘으로 확장되지 않는다. 네스트된 엘리먼트들의 확장들은 네스트된 엘리먼트 타입들의 필드들과 동일한 메커니즘으로 저장되고 액세스된다.

데이터 모델은 Element라고 명명된 네스트된 엘리먼트 타입들에 대한 루트를 정의한다:

```
<Type Name="Element"
    IsAbstract="True">
        <Property Name="ElementID"
                Type="the storage platformTypes.uniqueidentifier"
                Nullable="false"
                MultiValued="false"/>
</Type>
```

NestedElement 타입은 이 타입으로부터 상속한다. NestedElement 엘리먼트 타입은 부가적으로 엘리먼트들의 멀티세트인 필드를 정의한다.

```
<Type Name="NestedElement" BaseType="Base.Element"
        IsAbstract="True">
        <Property Name="Extensions"
                Type="Base.Element"
                Nullable="false"
                MultiValued="true"/>
</Type>
```

NestedElement 확장들은 다음과 같이 아이템 확장들과 다르다:

- 네스트된 엘리먼트 확장들은 확장 타입이 아니다. 그것들은 Base.Extension 타입에서 기원(root)되는 확장 타입 계층 구성에 속하지 않는다.

- 네스트된 엘리먼트 확장들은 아이템의 다른 필드들과 함께 저장되고 전역적으로(globally) 액세스 가능하지 않다 - 주어진 확장 타입의 모든 인스턴스들을 검색하는 쿼리가 구성될 수 없다.

- 이들 확장들은 (아이템의) 다른 네스트된 엘리먼트들이 저장되는 것과 동일하게 저장된다. 다른 네스트된 세트들처럼, NestedElement 확장들은 UDT에 저장된다. 그것들은 네스트된 엘리먼트 타입의 Extensions 필드를 통하여 액세스 가능하다.

● 다중 값 속성들에 액세스하기 위해 사용되는 컬렉션 인터페이스들은 또한 타입 확장들의 세트에 액세스하고 그것을 되풀이 반복(iterating over)하기 위해 사용된다.

다음 표는 아이템 확장들 및 NestedElement 확장들을 요약하고 비교한다.

아이템 확장 대 NestedElement 확장

	아이템 확장	NestedElement 확장
저장	아이템 확장 계층 구성이 그 자신의 표들에 저장됨	네스트된 엘리먼트들과 같이 저장됨
쿼리/검색	아이템 확장 표들을 쿼리할 수 있음	일반적으로 포함하고 있는 아이템 컨텍스트 내에서만 쿼리될 수 있음
쿼리/검색 범위	아이템 확장 타입의 모든 인스턴스들에 걸쳐서 검색할 수 있음	단일의 (포함하고 있는) 아이템의 네스트된 엘리먼트 타입 인스턴스들 내에서만 검색할 수 있음
프로그램성	특별한 확장 API들 및 확장 표에 대한 쿼리를 필요로 함	NestedElement 확장들은 네스트된 엘리먼트의 임의의 다른 다중 값 필드와 유사함; 통상의 네스트된 엘리먼트 타입 API들이 사용됨
거동(Behavior)	거동을 관련시킬 수 있음	아무런 거동도 허용되지 않음(?)
관계 의미	아이템 확장들에의 관계가 없음	NestedElement 확장들에의 관계가 없음
아이템 ID	아이템의 아이템 id를 공유함	그 자신의 아이템 id를 갖지 않음. NestedElement 확장은 아이템의 일부임

D. 데이터베이스 엔진

위에서 언급한 바와 같이, 데이터 스토어는 데이터베이스 엔진 상에서 구현된다. 본 실시예에서, 데이터베이스 엔진은 객체 관계 확장들(object relational extensions)로, 마이크로소프트 SQL 서버 엔진과 같은, SQL 쿼리 언어를 구현하는 관계 데이터베이스 엔진(relational database engine)을 포함한다. 이 섹션은 본 실시예에 따라서, 데이터 스토어가 구현하는 데이터 모델의 관계 스토어에의 매핑을 설명하고 저장 플랫폼 클라이언트들에 의해 소비되는 논리적 API에 대한 정보를 제공한다. 그러나, 상이한 데이터베이스 엔진이 이용될 때 상이한 매핑이 이용될 수 있음은 물론이다. 실제로, 관계 데이터베이스 엔진 상에서 저장 플랫폼 개념 데이터베이스 모델을 구현하는 것 외에, 그것은 다른 타입의 데이터베이스들, 예를 들면, 객체 지향(object-oriented) 및 XML 데이터베이스들 상에서도 구현될 수 있다.

객체 지향(OO) 데이터베이스 시스템은 프로그래밍 언어 객체들(예를 들면, C++, Java)에 대한 영속성 및 트랜잭션들을 제공한다. "아이템"의 저장 플랫폼 개념은 객체 지향 시스템에서의 "객체"(Object)에도 잘 매핑된다(비록 임베드된 컬렉션들이 객체들에 부가되어야 하겠지만). 상속 및 네스트된 엘리먼트 타입들처럼, 다른 저장 플랫폼 타입 개념들도 객체 지향 타입 시스템들을 매핑한다. 객체 지향 시스템들은 전형적으로 이미 객체 아이덴티티(identity)를 지원하고, 따라서 아이템 아이덴티티는 객체 아이덴티티에 매핑될 수 있다. 아이템 거동들(동작들)은 객체 메서드들에 잘 매핑된다. 그러나, 객체 지향 시스템들은 전형적으로 조직 능력(organizational capabilities)이 없고 검색에서 취약하다. 또한, 객체 지향 시스템들은 비구조화된 및 반구조화된 데이터에 대한 지원을 제공하지 않는다. 여기에서 설명된 완전한 저장 플랫폼 데이터 모델을 지원하기 위하여, 관계, 폴더, 및 확장과 같은 개념들이 객체 데이터 모델에 부가될 필요가 있을 것이다. 게다가, 프로모션, 동기화, 통지, 및 보안과 같은 메커니즘들이 구현될 필요가 있을 것이다.

객체 지향 시스템들과 유사하게, XML 데이터베이스들은, XSD(XML Schema Definition)에 기초하여, 단일 상속 기반 타입 시스템(single-inheritance based type system)을 지원한다. 본 발명의 아이템 타입 시스템은 XSD 타입 모델에 매핑될 수 있을 것이다. XSD들은 또한 거동들에 대한 지원을 제공하지 않는다. 아이템들에 대한 XSD들은 아이템 거동들(item behaviors)로 증가(augment)되어야 할 것이다. XML 데이터베이스들은 단일 XSD 문서들을 다루고 조직화 및 넓은 검색 능력들이 없다. 객체 지향 데이터베이스들에서와 같이, 여기에서 설명된 데이터 모델을 지원하기 위하여, 관계, 및 폴더와 같은 다른 개념들이 그러한 XLM 데이터베이스들에 통합될 필요가 있을 것이고, 또한 동기화, 통지 및 보안과 같은 메커니즘들이 구현될 필요가 있을 것이다.

다음의 하위 섹션들과 관련하여, 개시된 일반적인 정보의 이해를 용이하게 하기 위하여 약간의 예시들이 제공된다. 도 13은 통지 메커니즘을 예시하는 도면이다. 도 14는 2개의 트랜잭션들이 양쪽 모두 새로운 레코드를 동일한 B-트리에 삽입하는 예를 예시하는 도면이다. 도 15는 데이터 변경 검출 프로세스를 예시한다. 도 16은 예시적인 디렉토리 트리를 예시한다. 도 17은 디렉토리 기반 파일 시스템의 기존의 폴더가 저장 플랫폼 데이터 스토어 내로 이동되는 예를 도시한다.

1. UDT를 이용한 데이터 스토어 구현

본 실시예에서는, 일 실시예에서 마이크로소프트 SQL 서버 엔진을 포함하는 관계 데이터베이스 엔진(314)이 내장형(built-in) 스칼라 타입들(built-in scalar types)을 지원한다. 내장형 스칼라 타입들은 "원시적"(native)이고 "단순"(simple)하다. 그것들은 사용자가 그들 자신의 타입들을 정의할 수 없다는 점에서 원시적이고 그것들이 복합 구조를 캡슐화(encapsulate)할 수 없다는 점에서 단순하다. 사용자 정의 타입들(이하에서는, UDT들)은 사용자들이 복합의 구조화된 타입들을 정의하여 원시 스칼라 타입 시스템을 확장할 수 있게 함으로써 상기 원시 스칼라 타입 시스템을 넘어서 타입 확장성에 대한 메커니즘을 제공한다. 일단 사용자에 의해 정의되면, UDT는 내장형 스칼라 타입이 사용될 수 있는 타입 시스템 내의 어디에서든지 사용될 수 있다.

본 발명의 일 양태에 따르면, 저장 플랫폼 스키마들은 데이터베이스 엔진 스토어 내의 UDT 클래스들에 매핑된다. 데이터 스토어 아이템들은 Base.Item 타입으로부터 도출하는 UDT 클래스들에 매핑된다. 아이템들처럼, 확장들도 UDT 클래스들에 매핑되고 상속을 이용한다. 루트 확장 타입은 Base.Extension이고, 그로부터 모든 확장 타입들이 도출된다.

UDT는 CLR 클래스이다 - 그것은 상태(즉, 데이터 필드들) 및 거동(즉, 루틴들)을 갖는다. UDT들은 관리되는 언어들 - C#, VB.NET 등 - 중 임의의 것을 이용하여 정의된다. UDT 메서드들 및 연산자들은 그 타입의 인스턴스에 대하여 T-SQL에서 호출될 수 있다. UDT는 로우(row) 내의 칼럼(column)의 타입, T-SQL 내의 루틴의 파라미터의 타입, 또는 T-SQL 내의 변수의 타입일 수 있다.

저장 플랫폼 스키마들의 UTD 클래스들에의 매핑은 하이 레벨에서 상당히 직접적이다(fairly straightforward). 일반적으로, 저장 플랫폼 스키마는 CLR 네임스페이스에 매핑된다. 저장 플랫폼 타입은 CLR 클래스에 매핑된다. CLR 클래스 상속은 저장 플랫폼 타입 상속을 반영하고, 저장 플랫폼 속성은 CLR 클래스 속성에 매핑된다.

2. 아이템 매핑

아이템들이 전역적으로 검색 가능한 것이 바람직하고, 상속 및 타입 대체성(substitutability)에 대한 본 실시예의 관계 데이터에서의 지원이 있을 경우, 데이터베이스 스토어 내의 아이템 저장에 대한 하나의 가능한 구현은 모든 아이템들을 Base.Item 타입의 칼럼을 갖는 단일 표에 저장하는 것일 것이다. 타입 대체성을 이용하여, 모든 타입의 아이템들이 저장될 수 있고, 검색들은 Yukon의 "is of (Type)" 연산자를 이용하여 아이템 타입 및 서브타입에 의해 필터링될 수 있다.

그러나, 그러한 방법과 관련된 오버헤드(overhead)에 대한 우려 때문에, 본 실시예에서는, 아이템들이 상위 레벨 타입에 의해 나누어지고, 따라서 각각의 타입 "패밀리"의 아이템들은 별개의 표에 저장된다. 이 분할 방식 하에서는, Base.Item으로부터 직접 상속하는 각각의 아이템 타입에 대해 표가 생성된다. 이것들 아래에 상속하는 타입들은 위에서 설명한 바와 같이 타입 대체성을 이용하여 적당한 타입 패밀리 표에 저장된다. Base.Item으로부터의 상속의 제1 레벨만이 특별하게 취급된다.

모든 아이템들에 대해 전역적으로 검색 가능한 속성들의 복사본을 저장하기 위하여 "음영"(shadow) 표가 사용된다. 이 표는, 그를 통해 모든 데이터 변경들이 행해지는, 저장 플랫폼 API의 Update() 메서드에 의해 유지될 수 있다. 타입 패밀리 표들과 다르게, 이 글로벌 아이템 표는, 전체 UDT 아이템 객체가 아니라, 아이템의 상위 레벨 스칼라 속성들만을 포함한다. 이 글로벌 아이템 표는 ItemID 및 TypeID를 노출시킴으로써 타입 패밀리 표에 저장된 아이템 객체에의 탐색을 가능케 한다. ItemID는 일반적으로 데이터 스토어 내의 아이템을 고유하게 식별할 것이다. TypeID는 여기에서 설명되지 않은 메타데이터를 이용하여 타입 이름 및 해당 아이템을 포함하는 뷰에 매핑될 수 있다. 그것의 ItemID에 의해 아이템을 찾는 것은 공통의 동작일 수 있으므로, 글로벌 아이템 표의 컨텍스트에서 및 다른 경우에서, 아이템의 ItemID가 주어지면 아이템 객체를 검색하기 위한 GetItem() 함수가 제공된다.

편리한 액세스를 위하여 그리고 가능한 한 구현 상세를 감추기 위하여, 아이템들의 모든 쿼리들은 위에서 설명한 아이템 표들 상에 구축된 뷰들에 대한 것일 수 있다. 구체적으로, 뷰들은 적당한 타입의 패밀리 표에 대하여 각각의 아이템 타입마

다 생성될 수 있다. 이들 타입 뷰들은 서브타입들을 포함하여, 관련 타입의 모든 아이템들을 선택할 수 있다. 편의를 위하여, UDT 객체 외에, 뷰들은 상속 필드들을 포함하여, 해당 타입의 상위 레벨 필드들 모두에 대한 칼럼들을 노출시킬 수 있다.

3. 확장 매핑

확장들은 아이템들과 매우 유사하고 몇 가지의 동일한 요건들을 갖는다. 또 하나의 루트 타입 지원 상속으로서, 확장들은 저장 시에 다수의 동일한 고려(considerations) 및 트레이드오프(trade-offs)의 대상이 된다. 이 때문에, 단일 표 방법보다는, 유사 타입 패밀리 매핑이 확장들에 적용된다. 물론, 다른 실시예들에서, 단일 표 방법이 사용될 수 있다. 본 실시예에서, 확장은 ItemID에 의해 정확히 하나의 아이템과 관련되고, 아이템의 컨텍스트에서 고유한 ExtensionID를 포함한다. 아이템들에서와 같이, ItemID 및 ExtensionID 쌍으로 이루어진 그것의 아이덴티티가 주어지면 확장을 검색하기 위한 함수가 제공될 수 있다. 아이템 타입 뷰들과 마찬가지로, 각각의 확장 타입마다 뷰가 생성된다.

4. 네스트된 엘리먼트 매핑

네스트된 엘리먼트들(Nested Elements)은 깊게 네스트된 구조들을 형성하기 위해 아이템들, 확장들, 관계들, 또는 다른 네스트된 엘리먼트들에 임베드될 수 있는 타입들이다. 아이템들 및 확장들처럼, 네스트된 엘리먼트들은 UDT들로서 구현되지만, 그것들은 아이템들 및 확장들 내에 저장된다. 따라서, 네스트된 엘리먼트들은 그들의 아이템 및 확장 컨테이너들의 저장 매핑 이외에 저장 매핑을 갖지 않는다. 바꾸어 말하면, 시스템 내에 NestedElement 타입들의 인스턴스들을 직접 저장하는 표가 없고, 네스트된 엘리먼트들에 특정하게 전용되는 뷰가 없다.

5. 객체 아이덴티티(Object Identity)

데이터 모델 내의 각각의 엔트리, 즉 각각의 아이템, 확장 및 관계는 고유의 키 값(key value)을 갖는다. 아이템은 그것의 ItemId에 의해 고유하게 식별된다. 확장은 (ItemId, ExtensionId)의 혼합 키에 의해 고유하게 식별된다. 관계는 혼합 키 (ItemId, RelationshipId)에 의해 식별된다. ItemId, ExtensionId 및 RelationshipId는 GUID 값들이다.

6. SQL 객체 명명

데이터 스토어 내에 생성된 모든 객체들은 저장 플랫폼 스키마 이름으로부터 도출된 SQL 스키마 이름으로 저장될 수 있다. 예를 들면, 저장 플랫폼 베이스 스키마(흔히 "베이스"(Base)라 불림)는 "[System.Storage].Item"과 같은 "[System.Storage]" SQL 스키마 내의 타입들을 생성할 수 있다. 생성된 이름들은 명명 충돌(naming conflicts)을 제거하기 위하여 한정자(qualifier)가 앞에 붙는다. 적당할 경우, 감탄 문자(!)가 이름의 각각의 논리적 파트에 대한 분리자로서 사용된다. 아래의 표는 데이터 스토어 내의 객체들에 대해 사용되는 명명 규칙을 약술(outline)한다. 각각의 스키마 엘리먼트(아이템, 확장, 관계 및 뷰)가 데이터 스토어 내의 인스턴스들에 액세스하기 위해 사용되는 수식된 명명 규칙(decorated naming convention)과 함께 리스트되어 있다.

객체	이름 수식	설명	예
마스터 아이템 검색 뷰	**Master!Item**	현재의 아이템 도메인 내의 아이템들의 요약을 제공함.	[System.Storage]. [**Master!Item**]
타입 아이템 (Typed Item) 검색 뷰	ItemType	아이템 및 임의의 부모 타입(들)로부터의 모든 속성 데이터를 제공함.	[AcmeCorp.Doc]. [OfficeDoc]
마스터 확장 검색 뷰	**Master!Extension**	현재의 아이템 도메인 내의 모든 확장들의 요약을 제공함.	[System.Storage]. **[Master!Extension]**
타입 확장 검색 뷰	**Extension**!extensionType	확장에 대한 모든 속성 데이터를 제공함.	[AcmeCorp.Doc]. [**Extension**!StickyNote]
마스터 관계 뷰	**Master!Relationship**	현재의 아이템 도메인 내의 모든 관계들의 요약을 제공함.	[System.Storage]. **[Master!Relationship]**
관계 뷰	**Relationship**!relationshipName	주어진 관계와 관련된 모든 데이터를 제공함.	[AcmeCorp.Doc]. [**Relationship**!AuthorsFromDocument]
뷰	**View**!viewName	스카마 뷰 정의에 기초하여 칼럼/타입을 제공	[AcmeCorp.Doc]. [**View**!DocumentTitles]

7. 칼럼 명명

임의의 객체 모델을 스토어 내에 매핑할 경우, 애플리케이션 객체와 함께 저장된 부가적인 정보 때문에 명명 충돌의 가능성이 발생한다. 명명 충돌을 피하기 위하여, 모든 비타입 특정 칼럼들(non-type specific columns)(타입 선언 내의 명명된 속성에 직접 매핑하지 않는 칼럼들)은 밑줄(_) 문자가 앞에 붙는다. 본 실시예에서, 밑줄(_) 문자들은 임의의 식별자 속성의 시작 문자로서 허용되지 않는다. 또한, CLR과 데이터 스토어 간의 명명을 단일화하기 위하여, 저장 플랫폼 타입들 또는 스키마 엘리먼트(관계 등)의 모든 속성들은 대문자로 된 첫 번째 문자를 가져야 한다.

8. 검색 뷰(Search Views)

저장된 콘텐츠를 검색하기 위하여 저장 플랫폼에 의해 뷰들이 제공된다. 각각의 아이템 및 확장 타입에 대하여 SQL 뷰가 제공된다. 또한, (데이터 모델에 의해 정의된 바와 같이) 관계들 및 뷰들을 지원하기 위해 뷰들이 제공된다. 저장 플랫폼 내의 모든 SQL 뷰들 및 하위 표들(underlying tables)은 읽기 전용이다. 아래에서 더 상세히 설명되겠지만, 데이터는 저장 플랫폼 API의 Update() 메서드를 이용하여 저장되거나 변경될 수 있다.

저장 플랫폼 스키마에서 명시적으로 정의된(스키마 설계자에 의해 정의되고, 저장 플랫폼에 의해 자동으로 생성되지 않은) 각각의 뷰는 명명된 SQL 뷰 [<schema-name>].[View!<view-name>]에 의해 액세스 가능하다. 예를 들면, 스키마 "AcmePublisher.Books"에서 "BookSales"라고 명명된 뷰는 이름 "[AcmePublisher.Books].[View!BookSales]"를 이용하여 액세스 가능할 것이다. 뷰의 출력 포맷은 뷰별로(on a per-view basis) 맞춤화(custom)되므로(뷰를 정의하는 당사자(party)에 의해 제공된 임의의 쿼리에 의해 정의됨), 칼럼들은 스키마 뷰 정의에 기초하여 직접 매핑된다.

저장 플랫폼 데이터 스토어 내의 모든 SQL 검색 뷰들은 칼럼들에 대한 다음의 순서화 규칙을 이용한다:

- ItemId, ElementId, RelationshipId,...와 같은 뷰 결과의 논리적 "키" 칼럼(들).
- TypeId와 같은 결과의 타입에 관한 메타데이터 정보.
- CreateVersion, UpdateVersion, ...와 같은 변경 추적 칼럼들.
- 타입 특정 칼럼(들)(선언된 타입의 속성들)
- 타입 특정 뷰들(패밀리 뷰들)은 또한 객체를 반환하는 객체 칼럼을 포함한다.

각각의 타입 패밀리의 멤버들은 일련의 아이템 뷰들을 이용하여 검색 가능하고, 데이터 스토어 내의 아이템 타입마다 하나의 뷰가 있다. 도 28은 아이템 검색 뷰의 개념을 예시하는 도면이다.

a) 아이템

각각의 아이템 검색 뷰는 특정 타입 또는 그것의 서브타입들의 아이템의 각각의 인스턴스에 대한 행(row)을 포함한다. 예를 들면, Document에 대한 뷰는 Document, LegalDocument 및 ReviewDocument의 인스턴스들을 반환할 수 있다. 이 예가 주어지면, 아이템 뷰들은 도 29에 도시된 바와 같이 개념화될 수 있다.

(1) 마스터 아이템 검색 뷰(Master Item Search View)

저장 플랫폼 데이터 스토어의 각각의 인스턴스는 마스터 아이템 뷰(Master Item View)라고 불리는 특별한 아이템 뷰를 정의한다. 이 뷰는 데이터 스토어 내의 각각의 아이템에 관한 요약 정보를 제공한다. 이 뷰는 아이템 타입 속성마다 하나의 칼럼을 제공하고, 아이템의 타입을 기술하는 칼럼 및 변경 추적 및 동기화 정보를 제공하기 위해 사용되는 몇몇 칼럼들을 제공한다. 마스터 아이템 뷰는 이름 "[System.Storage].[Master!Item]"을 이용하여 데이터 스토어 내에서 식별된다.

칼럼	타입	설명
ItemId	ItemId	아이템의 저장 플랫폼 아이덴티티
_TypeId	TypeId	아이템의 TypeId - 아이템의 정확한 타입을 식별하고 메타데이터 카탈로그를 이용하여 그 타입에 관한 정보를 검색하기 위해 이용될 수 있음.

_RootItemId	ItemId	이 아이템의 수명을 제어하는 제1 비임베드 선조(non-embeded ancestor)의 ItemId.
<글로벌 변경 추적>	...	글로벌 변경 추적 정보
<아이템 속성들>	이용불가능(n/a)	아이템 타입 속성마다 하나의 칼럼

(2) 타입 아이템 검색 뷰(Typed Item Search Views)

각각의 아이템 타입은 또한 검색 뷰를 갖는다. 루트 아이템 뷰와 유사하지만, 이 뷰는 또한 "_Item" 칼럼을 통하여 아이템 객체에의 액세스를 제공한다. 각각의 타입 아이템 검색 뷰는 이름 [*schemaName*].[*itemTypeName*]을 이용하여 데이터 스토어 내에서 식별된다. 예를 들면, [AcmeCorp.Doc].[OfficeDoc].

칼럼	타입	설명
ItemId	ItemId	아이템의 저장 플랫폼 아이덴티티
<타입 변경 추적>	...	타입 변경 추적 정보
<부모 속성>	<속성 특정>	부모 속성마다 하나의 칼럼
<아이템 속성>	<속성 특정>	이 타입의 배타적 속성마다 하나의 칼럼
_Item	아이템의 CLR 타입	CLR 객체 - 선언된 아이템의 타입

b) 아이템 확장

WinFS 스토어 내의 모든 아이템 확장들은 또한 검색 뷰들을 이용하여 액세스 가능하다.

(1) 마스터 확장 검색 뷰

데이터 스토어의 각각의 인스턴스는 마스터 확장 뷰(*Master Extension View*)라고 불리는 특별한 확장 뷰를 정의한다. 이 뷰는 데이터 스토어 내의 각각의 확장에 관한 요약 정보를 제공한다. 이 뷰는 확장 속성마다 하나의 칼럼을 제공하고, 확장의 타입을 설명하는 칼럼 및 변경 추적 및 동기화 정보를 제공하기 위해 사용되는 몇몇 칼럼들을 제공한다. 마스터 확장 뷰는 이름 "[System.Storage].[Master!Extension]"을 이용하여 데이터 스토어 내에서 식별된다.

칼럼	타입	설명
ItemId	ItemId	이 확장이 관련되어 있는 아이템의 저장 플랫폼 아이덴티티
ExtensionId	ExtensionId(GUID)	이 확장 인스턴스의 Id
_TypeId	TypeId	확장의 TypeId - 확장의 정확한 타입을 식별하고 메타데이터 카탈로그를 이용하여 그 확장에 관한 정보를 검색하기 위해 이용될 수 있음.
<글로벌 변경 추적>	...	글로벌 변경 추적 정보
<확장 속성들>	<속성 특정>	확장 타입 속성마다 하나의 칼럼

(2) 타입 확장 검색 뷰(Typed Extension Search Views)

각각의 확장 타입은 또한 검색 뷰를 갖는다. 마스터 확장 뷰와 유사하지만, 이 뷰는 또한 _Extension 칼럼을 통하여 아이템 객체에의 액세스를 제공한다. 각각의 타입 확장 검색 뷰는 이름 [schemaName].[Extension!extensionTypeName]을 이용하여 데이터 스토어 내에서 식별된다. 예를 들면, [AcmeCorp.Doc].[Extension!OfficeDocExt].

칼럼	타입	설명
ItemId	ItemId	이 확장과 관련되어 있는 아이템의 저장 플랫폼 아이덴티티
ExtensionId	ExtensionId(GUID)	이 확장 인스턴스의 Id
<타입 변경 추적>	...	타입 변경 추적 정보
<부모 속성>	<속성 특정>	부모 속성마다 하나의 칼럼
<확장 속성>	<속성 특정>	이 타입의 배타적 속성마다 하나의 칼럼
_Extension	확장 인스턴스의 CLR 타입	CLR 객체 - 선언된 확장의 타입

c) 네스트된 엘리먼트(Nested Elements)

모든 네스트된 엘리먼트들은 아이템, 확장 또는 관계 인스턴스들 내에 저장된다. 그러므로, 그것들은 적당한 아이템, 확장, 또는 관계 검색 뷰를 쿼리함으로써 액세스된다.

d) 관계(Relationships)

위에서 논의한 바와 같이, 관계들은 저장 플랫폼 데이터 스토어 내의 아이템들 간의 연결의 기본적인 단위를 형성한다.

(1) 마스터 관계 검색 뷰

각각의 데이터 스토어는 마스터 관계 뷰(Master Relationship View)를 제공한다. 이 뷰는 데이터 스토어 내의 모든 관계 인스턴스들에 관한 정보를 제공한다. 마스터 관계 뷰는 이름 "[System.Storage].[Master!Relationship]"을 이용하여 데이터 스토어 내에서 식별된다.

칼럼	타입	설명
ItemId	ItemId	소스 종점의 아이덴티티(ItemId)
RelationshipId	RelationshipId(GUID)	관계 인스턴스의 id
_RelTypeId	관계 TypeId	관계의 RelTypeId - 메타데이터 카탈로그를 이용하여 관계 인스턴스의 타입을 식별함.
<글로벌 변경 추적>	...	글로벌 변경 추적 정보
TargetItemReference	ItemReference	타깃 종점의 아이덴티티
_Relationship	관계	이 인스턴스에 대한 관계 객체의 인스턴스

(2) 관계 인스턴스 검색 뷰

각각의 선언된 관계는 또한 특정 관계의 모든 인스턴스들을 반환하는 검색 뷰를 갖는다. 마스터 관계 뷰와 유사하지만, 이 뷰는 또한 관계 데이터의 각각의 속성에 대한 명명된 칼럼들을 제공한다. 각각의 관계 인스턴스 검색 뷰는 이름 [schemaName].[**Relationship**!relationshipName]을 이용하여 데이터 스토어 내에서 식별된다. 예를 들면, [AcmeCorp.Doc].[Relationship!DocumentAuthor].

칼럼	타입	설명
ItemId	ItemId	소스 종점의 아이덴티티(ItemId)
RelationshipId	RelationshipId(GUID)	관계 인스턴스의 Id
<타입 변경 추적>	...	타입 변경 추적 정보
TargetItemReference	ItemReference	타깃 종점의 아이덴티티
<소스 이름>	ItemId	소스 종점 아이덴티티(ItemId에 대한 별명)의 명명된 속성
<타깃 이름>	ItemReference 또는 도출된 클래스	타깃 종점 Id(TargetItemReference에 대한 별명 및 캐스트)의 명명된 속성
<관계 속성>	<속성 특정>	관계 정의의 속성마다 하나의 칼럼
_Relationship	관계 인스턴스의 CLR 타입	CLR 객체 - 선언된 관계의 타입

e)

9. 업데이트

저장 플랫폼 데이터 스토어 내의 모든 뷰들은 읽기 전용이다. 데이터 모델 엘리먼트(아이템, 확장 또는 관계)의 새로운 인스턴스를 생성하거나, 또는 기존의 인스턴스를 업데이트하기 위하여, 저장 플랫폼 API의 ProcessOperation 또는 ProcessUpdategram 메서드들이 이용되어야 한다. ProcessOperation 메서드는 수행될 액션을 상술(detail)하는 "연산"(operation)을 소비하는 데이터 스토어에 의해 정의된 단일의 저장된 프로시저(procedure)이다. ProcessUpdategram 메서드는 수행될 액션들의 세트를 집합적으로 상술하는 "updategram"이라고 알려진, 연산들의 순서화된 세트를 취하는 저장된 프로시저이다.

연산 포맷(operation format)은 확장 가능하고 스키마 엘리먼트들에 대하여 갖가지 연산들을 제공한다. 몇 가지의 공통 연산들은 다음을 포함한다:

1. 아이템 연산:

a. CreateItem(임베딩 또는 홀딩 관계의 컨텍스트에서 새로운 아이템을 생성함)

b. UpdateItem(기존의 아이템을 업데이트함)

2. 관계 연산:

a. CreateRelationship(참조 또는 홀딩 관계의 인스턴스를 생성함)

b. UpdateRelationship(관계 인스턴스를 업데이트함)

c. DeleteRelationship(관계 인스턴스를 제거함)

3. 확장 연산:

a. CreateExtension(기존의 아이템에 확장을 부가함)

b. UpdateExtension(기존의 확장을 업데이트함)

c. DeleteExtension(확장을 삭제함)

10. 변경 추적 및 툼스톤(Change Tracking & Tombstones)

아래에서 더 상세히 논의되는 바와 같이, 데이터 스토어에 의해 변경 추적 및 툼스톤 서비스들이 제공된다. 이 섹션은 데이터 스토어에서 노출된 변경 추적 정보에 대한 개요를 제공한다.

a) 변경 추적

데이터 스토어에 의해 제공된 각각의 검색 뷰는 변경 추적 정보를 제공하기 위해 사용되는 칼럼들을 저장하고, 이 칼럼들은 모든 아이템, 확장 및 검색 뷰에 걸쳐서 공통이다. 스키마 설계자들에 의해 명시적으로 정의된, 저장 플랫폼 스키마 뷰들은 변경 추적 정보를 자동으로 제공하지 않는다 - 그러한 정보는 뷰 자체가 구축되어 있는 검색 뷰들을 통하여 간접적으로 제공된다.

데이터 스토어 내의 각각의 엘리먼트에 대하여, 변경 추적 정보는 두 곳 - "마스터" 엘리먼트 뷰 및 "타입"(typed) 엘리먼트 뷰로부터 얻을 수 있다. 예를 들면, AcmeCorp.Document.Document 아이템 타입에 관한 변경 추적 정보는 마스터 아이템 뷰 "[System.Storage].[Master!Item]" 및 타입 아이템 검색 뷰 [AcmeCorp.Document].[Document]로부터 얻을 수 있다.

(1) "마스터" 검색 뷰에서의 변경 추적

마스터 검색 뷰들 내의 변경 추적 정보는 엘리먼트의 생성 및 업데이트 버전들에 관한 정보, 어느 동기 파트너(sync partner)가 그 엘리먼트를 생성하였는지, 어느 동기 파트너가 마지막으로 그 엘리먼트를 업데이트하였는지 및 생성 및 변경을 위한 각각의 파트너로부터의 버전 번호들에 관한 정보를 제공한다. (아래에서 설명되는) 동기 관계들 내의 파트너들은 파트너 키(partner key)에 의해 식별된다. [System.Storage.Store].ChangeTrackingInfo 타입의 _ChangeTrackingInfo라고 명명된 단일 UDT 객체는 모든 이런 정보를 포함한다. 이 타입은 System.Storage 스키마에서 정의된다. _ChangeTrackingInfo는 아이템, 확장 및 관계에 대한 모든 글로벌 검색 뷰들에서 얻을 수 있다. ChangeTrackingInfo의 타입 정의는 다음과 같다:

```
<Type Name="ChangeTrackingInfo" BaseType="Base.NestedElement">
  <FieldProperty Name="CreationLocalTS"       Type="SqlTypes.SqlInt64"
  Nullable="False" />
  <FieldProperty Name="CreatingPartnerKey"
  Type="SqlTypes.SqlInt32"        Nullable="False" />
  <FieldProperty Name="CreatingPartnerTS"
  Type="SqlTypes.SqlInt64"        Nullable="False" />
  <FieldProperty Name="LastUpdateLocalTS"
  Type="SqlTypes.SqlInt64"        Nullable="False" />
  <FieldProperty Name="LastUpdatingPartnerKey"
  Type="SqlTypes.SqlInt32"        Nullable="False" />
  <FieldProperty Name="LastUpdatingPartnerTS" Type="SqlTypes.SqlInt64"
  Nullable="False" />
</Type>
```

이들 속성은 다음의 정보를 포함한다:

칼럼	설명
_CreationLocalTS	로컬 머신에 의한 생성 타임 스탬프
_CreatingPartnerKey	이 엔티티를 생성한 파트너의 PartnerKey. 엔티티가 로컬로(locally) 생성되었다면, 이것은 로컬 머신의 PartnerKey이다.
_CreatingPartnerTS	_CreatingPartnerKey에 대응하는 파트너에서 이 엔티티가 생성된 시간의 타임스탬프.
_LastUpdateLocalTS	로컬 머신에서의 업데이트 시간에 대응하는 로컬 타임스탬프.
_LastUpdatingPartnerKey	이 엔티티를 마지막으로 생성한 파트너의 PartnerKey. 엔티티에 대한 마지막 업데이트가 로컬로 행해졌다면, 이것은 로컬 머신의 PartnerKey이다.
_LastUpdatingPartnerTS	_LastUpdatingPartnerKey에 대응하는 파트너에서 이 엔티티가 생성된 시간의 타임스탬프.

(2) "타입" 검색 뷰에서의 변경 추적

글로벌 검색 뷰로서 동일한 정보를 제공하는 것 외에, 각각의 타입 검색 뷰는 동기 토폴로지(sync topology)에서의 각각의 엘리먼트의 동기 상태를 기록하는 부가적인 정보를 제공한다.

칼럼	타입	설명
<글로벌 변경 추적>	...	글로벌 변경 추적으로부터의 정보
_ChangeUnitVersions	멀티세트<ChangeUnitVersion>	파트너 엘리먼트 내의 변경 유닛들의 버전 번호들의 기술
_ElementSyncMetadata	ElementSyncMetadata	동기화 런타임에만 관심 있는 이 아이템에 관한 부가적인 버전 독립형 메타데이터.
_VersionSyncMetadata	VersionSyncMetadata	동기화 런타임에만 관심 있는 이 버전에 관한 부가적인 버전 특정 메타데이터.

b) 툼스톤

데이터 스토어는 아이템, 확장 및 관계에 대한 툼스톤 정보를 제공한다. 툼스톤 뷰들은 한 곳에서의 라이브(live) 및 툼스톤(tombstoned) 엔티티들(아이템, 확장 및 관계들)에 관한 정보를 제공한다. 아이템 및 확장 툼스톤 뷰들은 대응하는 객체에의 액세스를 제공하지 않는 반면, 관계 툼스톤 뷰는 관계 객체에의 액세스를 제공한다(이 관계 객체는 툼스톤 관계의 경우에 NULL이다).

(1) 아이템 툼스톤

아이템 툼스톤들은 뷰 [System.Storage].[Tombstone!Item]을 통하여 시스템으로부터 검색된다.

칼럼	타입	설명
ItemId	ItemId	아이템의 아이덴티티
_TypeId	TypeId	아이템의 타입
<아이템 속성>	...	모든 아이템에 대해 정의된 속성들
_RootItemId	ItemId	이 아이템을 포함하는 첫 번째 논임베딩(non-embedding) 아이템의 ItemId.

_ChangeTrackingInfo	ChangeTrackingInfo 타입의 CLR 인스턴스	이 아이템에 대한 변경 추적 정보
_IsDeleted	BIT	이것은 라이브 아이템들에 대해서는 0이고, 툼스톤 아이템들에 대해서는 1인 플래그임.
_DeletionWallclock	UTCDATETIME	아이템을 삭제한 파트너에 따른 UTC 벽시계 날짜 시간. 이것은 아이템이 라이브이면 NULL이다.

(2) 확장 툼스톤

확장 툼스톤들은 뷰 [System.Storage].[Tombstone!Extension]을 사용하여 시스템으로부터 검색된다. 확장 변경 추적 정보는 ExtensionId 속성의 부가와 함께 아이템들에 대해 제공된 것과 유사하다.

칼럼	타입	설명
ItemId	ItemId	확장을 소유하는 아이템의 Id
ExtensionId	ExtensionId	확장의 확장 Id
_TypeID	TypeId	확장의 타입
_ChangeTrackingInfo	ChangeTrackingInfo 타입의 CLR 인스턴스	이 확장에 대한 변경 추적 정보
_IsDeleted	BIT	이것은 라이브 아이템들에 대해서는 0이고, 툼스톤 확장들에 대해서는 1인 플래그임.
_DeletionWallclock	UTCDATETIME	확장을 삭제한 파트너에 따른 UTC 벽시계 날짜 시간. 이것은 확장이 라이브이면 NULL이다.

(3) 관계 툼스톤

관계 툼스톤들은 뷰 [System.Storage].[Tombstone!Relationship]을 통하여 시스템으로부터 검색된다. 관계 툼스톤 정보는 확장들에 대해 제공된 것과 유사하다. 그러나, 관계 인스턴스의 타깃 ItemRef에 대한 부가적인 정보가 제공된다. 게다가, 관계 객체도 선택된다.

칼럼	타입	설명
ItemId	ItemId	관계를 소유한 아이템의 아이덴티티(관계 소스 종점의 아이덴티티)
RelationshipId	RelationshipId	관계의 RelationshipId
_TypeID	TypeId	관계의 타입
_ChangeTrackingInfo	ChangeTrackingInfo 타입의 CLR 인스턴스	이 관계에 대한 변경 추적 정보
_IsDeleted	BIT	이것은 라이브 아이템들에 대해서는 0이고, 툼스톤 확장들에 대해서는 1인 플래그임.
_DeletionWallclock	UTCDATETIME	관계를 삭제한 파트너에 따른 UTC 벽시계 날짜 시간. 이것은 관계가 라이브이면 NULL이다.
_Relationship	관계의 CLR 인스턴스	이것은 라이브 관계에 대한 관계 객체이다. 이것은 툼스톤 관계들에 대해서는 NULL이다.
TargetItemReference	ItemReference	타깃 종점의 아이덴티티

(4) 툼스톤 클린업(Tombstone Cleanup)

툼스톤 정보의 무한 성장을 막기 위하여, 데이터 스토어는 툼스톤 클린업 태스크를 제공한다. 이 태스크는 툼스톤 정보가 폐기될 수 있는 때를 결정한다. 이 태스크는 로컬 생성/업데이트 버전에 대한 경계(bound)를 계산한 다음 모든 이전의 툼스톤 버전들을 폐기함으로써 툼스톤 정보를 절단한다(truncate).

11. 도우미 API 및 함수(Helper APIs and Functions)

베이스 매핑은 또한 다수의 도우미 함수들을 제공한다. 이들 함수는 데이터 모델에 대한 공통의 연산들을 돕기 위하여 제공된다.

a) 함수 [System.Storage].GetItem

ItemId가 주어질 경우 아이템 객체를 반환한다

//

Item GetItem(ItemId ItemId)

b) 함수 [System.Storage].GetExtension

//ItemId 및 ExtensionId가 주어질 경우 확장 객체를 반환한다

//

Extension GetExtension(ItemId ItemId, ExtensionId ExtensionId)

c) 함수 [System.Storage].GetRelationship

//ItemId 및 RelationshipId가 주어질 경우 관계 객체를 반환한다

//

Relationship GetRelationship(ItemId ItemId, RelationshipId RelationshipId)

12. 메타데이터(Metadata)

스토어 내에서 표현되는 2 가지 타입의 메타데이터가 있다: 인스턴스 메타데이터(아이템의 타입 등), 및 타입 메타데이터.

a) 스키마 메타데이터

스키마 메타데이터는 메타 스카마로부터의 아이템 타입들의 인스턴스들로서 데이터 스토어 내에 저장된다.

b) 인스턴스 메타데이터

인스턴스 메타데이터는 아이템의 타입을 쿼리하기 위해 애플리케이션에 의해 사용되고 아이템과 관련된 확장들을 찾는다. 아이템에 대한 ItemId가 주어질 경우, 애플리케이션은 아이템의 타입을 반환하는 글로벌 아이템 뷰를 쿼리하고 이 값을 이용하여 아이템의 선언된 타입에 대한 정보를 반환하는 Meta.Type 뷰를 쿼리할 수 있다. 예를 들면,

```
// Return metadata Item object for given Item instance
//
SELECT m._Item AS metadataInfoObj
FROM [System.Storage].[Item] i INNER JOIN [Meta].[Type] m ON i._TypeId = m.ItemId
WHERE i.ItemId = @ItemId
```

E. 보안

일반적으로, 모든 보안 가능한 객체들(securable objects)은 도 26에 도시된 액세스 마스크 포맷을 이용하여 그들의 액세스 권리들을 배열한다. 이 포맷에서, 하위 16 비트는 객체 특정 액세스 권리에 대한 것이고, 다음 7 비트는 대부분의 객체 타입들에 적용되는 표준 액세스 권리에 대한 것이고, 4개의 상위 비트는 각각의 객체 타입이 표준 및 객체 특정 권리들의 세트에 매핑할 수 있는 일반적 액세스 권리들(generic access rights)을 특정하기 위해 사용된다. ACCESS_SYSTEM_SECURITY 비트는 객체의 SACL에 액세스할 권리에 대응한다.

도 26의 액세스 마스크 구조에서, 아이템 특정 권리들은 객체 특정 권리(Object Specific Rights) 섹션에 배치된다(하위 16비트). 본 실시예에서는, 저장 플랫폼이 보안 관리를 위해 2 세트의 API들 - Win32 및 저장 플랫폼 API를 노출시키기 때문에, 저장 플랫폼 객체 특정 권리들의 설계에 동기를 주기 위해 파일 시스템 객체 특정 권리들이 고려되어야 한다.

본 발명의 저장 플랫폼에 대한 보안 모델은 본 명세서의 앞에서 참고로 통합된 관련 출원들에 상세히 설명되어 있다. 이와 관련하여, 도 27(부분 a, b, 및 c)은 보안 모델의 일 실시예에 따라서, 기존의 보안 영역으로부터 새로운 동일하게 보호되는 보안 영역이 만들어지는 것을 도시한다.

F. 통지 및 변경 추적

본 발명의 다른 양태에 따르면, 저장 플랫폼은 애플리케이션들이 데이터 변경을 추적하도록 허용하는 통지 능력(notifications capability)을 제공한다. 이 특징은 주로 휘발성 상태를 유지하거나 데이터 변경 이벤트에 대한 비즈니스 로직을 실행하는 애플리케이션들을 위하여 의도되어 있다. 애플리케이션들은 아이템, 아이템 확장 및 아이템 관계에 대한 통지들을 등록한다. 통지들은 데이터 변경들이 행해진 후에 비동기적으로 전달된다. 에플리케이션들은 아이템, 확장 및 관계 타입뿐만 아니라 동작 타입에 의해서 통지들을 필터링할 수 있다.

일 실시예에 따르면, 저장 플랫폼 API(322)는 통지를 위해 2종류의 인터페이스들을 제공한다. 첫째로, 애플리케이션들은 아이템, 아이템 확장 및 아이템 관계들에 대한 변경들에 의해 트리거된 단순 데이터 변경 이벤트들을 등록한다. 둘째로, 애플리케이션들은 아이템, 아이템 확장 및 아이템들 간의 관계들의 세트들을 모니터하기 위해 "워처"(watcher) 객체들을 생성한다. 워처 객체의 상태는 저장될 수 있고 시스템 장애 후에 또는 시스템이 연장된 시간 기간 동안 오프라인으로 된 후에 재생성될 수 있다. 단일 통지가 다수의 업데이트들을 반영할 수도 있다.

이 기능에 관한 부가적인 상세는 본 명세서의 앞에서 참고로 통합된 관련 출원들에서 찾을 수 있다.

G. 동기화

본 발명의 다른 양태에 따르면, 저장 플랫폼은 (i) 저장 플랫폼의 다수의 인스턴스들(각각이 그 자신의 데이터 스토어(302)를 가짐)이 유연성 있는 규칙 세트에 따라서 그들의 콘텐츠의 파트들을 동기화하도록 허용하고, (ii) 제3자들이 본 발명의 저장 플랫폼의 데이터 스토어를 동기화하는 인프라구조에 사유 프로토콜(proprietary protocols)을 구현하는 다른 데이터 소스들을 제공하는 동기화 서비스(330)를 제공한다.

관여하는 복제본들(participating replicas)의 그룹 사이에서 저장 플랫폼 대 저장 플랫폼 동기화가 발생한다. 예를 들어, 도 3을 참조하면, 저장 플랫폼(300)의 데이터 스토어(302)와 아마도 다른 컴퓨터 시스템 상에서 실행되는 저장 플랫폼의 다른 인스턴스의 제어 하의 다른 원격 데이터 스토어(338) 사이의 동기화를 제공하는 것이 바람직할 수 있다. 이 그룹의 전체 멤버십은 임의의 주어진 시간에 임의의 주어진 복제본에게 반드시 알려질 필요는 없다.

서로 다른 복제본들이 독립적으로(즉, 동시에) 변경들을 행할 수 있다. 동기화의 프로세스는 모든 복제본들이 다른 복제본들에 의해 행해진 변경들을 알아차리게 하는 것으로 정의된다. 이 동기화 능력은 본래부터 멀티마스터(multi-master)이다.

본 발명의 동기화 능력은 복제본들이:

- 다른 복제본이 어떤 변경들을 알아차리고 있는지를 결정하고;
- 이 복제본이 알아채지 못하는 변경들에 관한 정보를 요구하고;
- 다른 복제본이 알아채지 못하는 변경들에 관한 정보를 전달하고;
- 2개의 변경들이 서로 충돌하는 때를 결정하고;
- 변경들을 로컬로 적용하고;
- 컨버전스(convergence)를 보증하기 위해 다른 복제본들에 충돌 레졸루션들(conflict resolutions)을 전달하고;
- 충돌 레졸루션들을 위한 특정 정책들에 기초하여 충돌들을 리졸브(resolve)하도록 허용한다.

1. 저장 플랫폼 대 저장 플랫폼 동기화(storage platform-to-storage platform sync)

본 발명의 저장 플랫폼의 동기화 서비스(330)의 주 응용은 저장 플랫폼의 다수의 인스턴스들(각각이 그 자신의 데이터 스토어를 가짐)을 동기화하는 것이다. 동기화 서비스는 (데이터베이스(314)의 하위 표들(underlying tables)보다는) 저장 플랫폼 스키마의 레벨에서 동작한다. 따라서, 예를 들면, 아래에서 논의되는 바와 같이 동기화 세트들을 정의하기 위해 "범위"(Scopes)가 사용된다.

동기화 서비스는 "최종 변경"(net changes)의 원리에 입각하여 동작한다. (트랜잭션 복제(transactional replication)와 같은) 개개의 동작들을 기록하고 보내기보다는, 동기화 서비스는 이들 동작들의 최종 결과를 보내고, 따라서 다수의 동작들의 결과들을 단일의 결과적인 변경으로 통합한다.

동기화 서비스는 일반적으로 트랙잭션 경계들을 고려하지 않는다. 바꾸어 말하여, 단일 트랜잭션에서 저장 플랫폼 데이터 스토어에 대해 2개의 변경이 행해지면, 이들 변경들이 모든 다른 복제들에서 자동으로 적용된다고 하는 보장이 없다 -- 하나가 다른 하나 없이 나타날 수도 있다. 이 원리의 예외는 동일 트랜잭션에서 동일 아이템에 대해 2개의 변경이 행해지면, 이들 변경들은 다른 복제들에 자동으로 보내지고 적용되도록 보증된다. 따라서, 아이템들은 동기화 서비스의 일관성 단위들(consistency units)이다.

a) 동기화(Sync) 제어 애플리케이션

임의의 애플리케이션이 동기화 서비스에 접속하여 동기화 동작을 개시할 수 있다. 그러한 애플리케이션은 동기화를 수행하기 위해 요구되는 모든 파라미터들을 제공한다(아래 sync 프로파일 참조). 그러한 애플리케이션들은 여기에서 Sync 제어 애플리케이션들(SCAs: Sync Controlling Applications)이라고 불린다.

2개의 저장 플랫폼 인스턴스들을 동기화할 때, SCA에 의해 한쪽에서 sync가 개시된다. 그 SCA는 원격 파트너와 동기화하도록 로컬 동기화 서비스에게 알린다. 다른 쪽에서, 동기화 서비스는 발신 기계(originating machine)로부터 동기화 서비스에 의해 보내진 메시지들에 의해 깨어난다(awoken). 그것은 목적지 기계(destination machine) 상에 존재하는 지속 구성 정보(persistent configuration information)(아래 매핑 참조)에 기초하여 응답한다. 동기화 서비스는 스케줄대로 또는 이벤트들에 응답하여 실행될 수 있다. 이들 경우에, 스케줄을 구현하는 동기화 서비스는 SCA가 된다.

동기화를 가능케 하기 위하여, 2개의 조치가 취해져야 한다. 첫째로, 스키마 설계자는 (아래에서 설명되는 바와 같이 변경 단위들(Change Units)을 지정하는) 적당한 sync 의미론으로 저장 플랫폼 스키마에 주석을 달아야 한다. 둘째로, 동기화는 (아래에서 설명되는 바와 같이) 동기화에 관여할 저장 플랫폼의 인스턴스를 갖는 모든 머신들 상에서 적당히 구성되어야 한다.

b) 스키마 주석(Schema annotation)

동기화 서비스의 기본적인 개념은 변경 단위(Change Unit)의 개념이다. 변경 단위는 저장 플랫폼에 의해 개별적으로 추적되는 최소의 스키마 단편(smallest piece of schema)이다. 모든 변경 단위에 대하여, 동기화 서비스는 마지막 sync 이래로 그것이 변경되었는지 또는 변경되지 않았는지 여부를 결정할 수 있을 것이다.

스카마 내의 변경 단위들을 지정하는 것은 몇 가지 목적에 도움이 된다. 첫째로, 그것은 유선 상에서 동기화 서비스가 얼마나 채티(chatty)한지를 결정한다. 변경이 변경 단위 내에서 행해질 경우, 전체 변경 단위가 다른 복제본들에게 보내진다. 왜냐하면 동기화 서비스는 변경 단위의 어느 파트가 변경되었는지를 알지 못하기 때문이다. 둘째로, 그것은 충돌 검출의 입도(granularity)를 결정한다. 2개의 동시 발생의 변경들(이들 용어는 후속 섹션들에서 상세히 정의됨)이 동일한 변경 단위에 대해 행해질 경우, 동기화 서비스는 충돌을 제기(raise)하고, 다른 한편으로, 2개의 동시 발생의 변경들이 서로 다른 변경 단위들에 대해 행해지면, 아무런 충돌도 제기되지 않고 그 변경들은 자동으로 병합된다. 셋째로, 그것은 시스템에 의해 유지된 메타데이터의 양에 상당히 영향을 미친다. 동기화 서비스 메타데이터의 많은 것은 변경 단위마다 유지되고, 따라서 변경 단위들을 작게 하면 sync의 부하(overhead)가 증가된다.

변경 단위들을 정의하는 것은 올바른 트레이드오프들을 찾는 것을 필요로 한다. 그 때문에, 동기화 서비스는 스키마 설계자들이 프로세스에 관여하도록 허용한다.

일 실시예에서, 동기화 서비스는 엘리먼트보다 큰 변경 단위들을 지원하지 않는다. 그러나, 그것은 스키마 설계자들이 엘리먼트보다 작은 변경 단위들을 특정할 능력을 지원한다 --- 즉, 엘리먼트의 다수의 속성들(attributes)을 별개의 변경 단위로 그룹화하는 것을 지원한다. 그 실시예에서, 이것은 다음의 구문을 이용하여 달성된다:

```
<Type Name="Appointment" MajorVersion="1" MinorVersion="0" ExtendsType="Base.Item"
        ExtendsVersion="1">

   <Field Name="MeetingStatus"  Type="the storage platformTypes.uniqueidentifier Nullable="False"/>
   <Field Name="OrganizerName" Type="the storage platformTypes.nvarchar(512)" Nullable="False"/>
   <Field Name="OrganizerEmail"  Type="the storage platformTypes.nvarchar(512)"
      TypeMajorVersion="1"                    MultiValued="True"/>
    ...
   <ChangeUnit Name="CU_Status">
       <Field Name="MeetingStatus"/>
   </ChangeUnit>

   <ChangeUnit Name="CU_Organizer"/>
       <Field Name="OrganizerName" />
       <Field Name="OrganizerEmail" />
  </ChangeUnit>
  ...

</Type>
```

c) Sync 구성

그들의 데이터의 특정 파트들을 sync로 유지하기를 원하는 저장 플랫폼 파트너들의 그룹은 sync 공동체(sync community)라고 불린다. 이 공동체의 멤버들은 sync에 머무르기를 원하지만, 그것들은 반드시 데이터를 정확히 동일하게 나타낼 필요는 없다. 바꾸어 말하면, sync 파트너들은 그들이 동기화하고 있는 데이터를 변환할 수 있다.

피어 투 피어 시나리오(peer-to-peer scenario)에서는, 피어들이 그들의 파트너들 모두에 대한 변환 매핑들을 유지하는 것은 비실용적이다. 대신에, 동기화 서비스는 "공동체 폴더들"(Community Folders)을 정의하는 방법을 취한다. 공동체 폴더는 모든 공동체 멤버들이 동기화하고 있는 가상적 "공유 폴더"(shared folder)를 나타내는 추상화이다.

이 개념은 예에 의해 가장 잘 설명된다. 만일 조(Joe)가 그의 몇 개의 컴퓨터들의 내문서(My Documents) 폴더들을 sync로 유지하기를 원하면, 조(Joe)는 이를테면 JoesDocuments라고 불리는 공동체 폴더를 정의한다. 그 후, 모든 컴퓨터 상에서, Joe는 가상적 JoesDocuments 폴더와 로컬 My Documents 폴더 사이의 매핑을 구성한다. 이 시점부터, Joe의 컴퓨터들이 서로 동기화할 때, 그것들은 그들의 로컬 아이템들보다는, JoesDocuments 내의 문서들에 관하여 이야기(talk)한다. 이런 식으로, 모든 Joe의 컴퓨터들은 상대가 누구인지 모르고도 서로 이해한다 -- 공동체 폴더는 sync 공동체의 공통어(lingua franca)가 된다.

동기화 서비스를 구성하는 것은 3 단계로 이루어진다: (1) 로컬 폴더들과 공동체 폴더들 간의 매핑들을 정의하는 단계; (2) 무엇이 동기화되는지(예를 들면, 누구와 동기화하는지 및 어떤 서브세트들이 송신되어야 하는지 및 어느 것이 수신하였는지)를 결정하는 sync 프로파일들을 정의하는 단계; (3) 상이한 sync 프로파일들이 실행되어야 하는 스케줄들을 정의하거나, 또는 그것들을 수동으로 실행하는 단계.

(1) 공동체 폴더 - 매핑

공동체 폴더 매핑들은 개개의 머신들 상에 XML 구성 파일들로서 저장된다. 각각의 매핑은 다음의 스키마를 갖는다:

/mappings/communityFolder

이 엘리먼트는 이 매핑이 대상으로 하는 공동체 폴더를 명명한다. 이 이름은 폴더들의 구문 규칙 다음에 온다.

/mappings/localFolder

이 엘리먼트는 매핑이 변환하는 로컬 폴더를 명명한다. 이 이름은 폴더들의 구문 규칙 다음에 온다. 이 폴더는 매핑이 유효하기 위해 항상 존재해야 한다. 이 폴더 내의 아이템들은 이 매핑마다의 동기화를 위해 고려된다.

/mappings/transformations

이 엘리먼트는 아이템들을 공동체 폴더로부터 로컬 폴더로 변환하고 다시 역으로 변환하는 방법을 정의한다. 만일 존재하지 않거나 비어 있다면, 아무런 변환도 수행되지 않는다. 특히, 이것은 아무런 ID도 매핑되지 않는다는 것을 의미한다. 이 구성은 주로 폴더의 캐시를 생성하는 데 유용하다.

/mappings/transformations/mapIDs

이 엘리먼트는 공동체 ID들을 재사용하기보다는 새로이 생성된 로컬 ID들이 공동체 폴더로부터 매핑된 모든 아이템들에 할당되도록 요구한다. Sync Runtime은 아이템들을 앞뒤로 변환하기 위해 ID 매핑들을 유지할 것이다.

/mappings/transformations/localRoot

이 엘리먼트는 공동체 폴더 내의 모든 루트 아이템들이 특정 루트의 자식으로 되도록 요구한다.

/mappings/runAs

이 엘리먼트는 누구의 권한 하에서 이 매핑에 대한 요구들이 처리되는지를 제어한다. 만일 없다면, 송신자(sender)가 추측된다.

/mappings/runAs/sender

이 엘리먼트의 존재는 이 매핑에 대한 메시지들의 송신자가 가장(impersonate)되어야 한다는 것을 나타내고, 그의 자격증명(credentials) 하에서 처리될 것을 요구한다.

(2) 프로파일

Sync 프로파일은 동기화를 시작(kick off)하기 위해 요구되는 파라미터들의 전체 세트이다. 그것은 sync를 개시하기 위해 SCA에 의해 Sync Runtime에 공급된다. 저장 플랫폼 대 저장 플랫폼 동기화를 위한 Sync 프로파일들은 다음의 정보를 포함한다:

- 변경을 위한 소스 및 목적지의 역할을 하는 로컬 폴더;
- 동기화하기 위한 원격 폴더 이름 - 이 폴더는 위에서 정의된 매핑에 의하여 원격 파트너로부터 공개(publish)되어야 한다;
- 방향(Direction) - 동기화 서비스는 송신 전용(send-only), 수신 전용(receive-only), 및 송수신 sync를 지원한다;
- 로컬 필터 -- 원격 파트너에 송신할 로컬 정보를 선택한다. 로컬 폴더에 대한 저장 플랫폼 쿼리로서 표현된다;
- 원격 필터 - 원격 파트너로부터 검색할 원격 정보를 선택한다. 공통체 폴더에 대한 저장 플랫폼 쿼리로서 표현된다;
- 변환 -- 아이템들을 로컬 포맷으로 및 로컬 포맷으로부터 변환하는 방법을 정의한다;
- 로컬 보안 --- 원격 종점으로부터 검색된 변경들이 원격 종점(가장됨)의 허가(permissions) 하에서 적용되어야 할지 또는 로컬로 sync를 개시한 사용자의 허가 하에서 적용되어야 할지를 특정한다;
- 충돌 레졸루션 정책 -- 충돌들이 리젝트(reject)되어야 할지, 로깅되어야 할지, 또는 자동으로 리졸브(resolve)되어야 할지를 특정한다 - 후자의 경우에, 그것은 어느 충돌 리졸버(conflict resolver)를 사용할지는 물론 그것에 대한 구성 파라미터들을 특정한다.

동기화 서비스는 Sync 프로파일들의 간단한 작성을 허용하는 런타임 CLR 클래스(runtime CLR class)를 제공한다. 프로파일들은 또한 용이한 저장을 위하여 XML 파일들로 및 XML 파일들로부터 직렬화될 수 있다(흔히 스케줄과 함께). 그러나, 저장 플랫폼 내에 모든 프로파일들의 저장되는 표준 위치가 없다. SCA들은 그것을 고집하지 않고 즉석에서(on the

spot) 마음대로 프로파일을 작성할 수 있다. sync를 개시하기 위해 로컬 매핑을 가질 필요는 없음을 주의해야 한다. 모든 sync 정보는 프로파일에서 특정될 수 있다. 그러나, 매핑은 원격 사이드에 의해 개시된 sync 요구에 응답하기 위하여 필요하다.

(3) 스케줄

일 실시예에서, 동기화 서비스는 그 자신의 스케줄링 인프라구조를 제공하지 않는다. 대신에, 그것은 이 태스크를 수행하기 위해 다른 컴포넌트에 의지한다 - 마이크로소프트 윈도즈 운영 시스템과 함께 이용가능한 윈도즈 스케줄러(Windows Scheduler). 동기화 서비스는 SCA로서 기능하고 XML 파일 내에 저장된 sync 프로파일에 기초하여 동기화를 트리거하는 명령 행 유틸리티(command-line utility)를 포함한다. 이 유틸리티는 스케줄대로, 또는 사용자 로그온 또는 로그오프와 같은 이벤트들에 응답하여 동기화를 실행하도록 윈도즈 스케줄러를 구성하는 것을 매우 용이하게 한다.

d) 충돌 핸들링(Conflict Handling)

동기화 서비스 내의 충돌 처리는 3 단계로 나누어진다: (1) 변경 적용 시간에 일어나는 충돌 검출 - 이 단계는 변경이 안전하게 적용될 수 있는지를 결정한다; (2) 자동 충돌 레졸루션 및 로깅 - (충돌이 검출된 직후에 일어나는) 이 단계 중에 충돌이 리졸브될 수 있는지를 알기 위해 자동 충돌 리졸버가 참고(consult)된다 - 만일 충돌이 리졸브될 수 없다면, 충돌은 선택적으로(optionally) 로깅될 수 있다; (3) 충돌 검사 및 레졸루션 - 이 단계는 몇몇 충돌들이 로깅되면 발생되고, sync 세션의 컨텍스트 밖에서 발생된다 - 이 때, 로깅된 충돌들은 리졸브되고 로그로부터 제거될 수 있다.

(1) 충돌 검출

본 실시예에서, 동기화 서비스는 2가지 유형의 충돌들을 검출한다: 지식 기반 충돌 및 제약 기반 충돌.

(a) 지식 기반 충돌(Knowledge-based conflicts)

지식 기반 충돌은 2개의 복제본들이 동일한 변경 단위에 대해 독립적인 변경들을 행할 때 발생한다. 2개의 변경이 서로에 대한 지식이 없이 행해지는 경우에 -- 바꾸어 말하면, 첫 번째 것의 버전이 두 번째 것의 지식에 의해 커버되지 않고 또한 그 반대인 경우에 두 변경들은 독립적이라고 불린다. 동기화 서비스는 위에서 설명한 바와 같이 복제본의 지식에 기초하여 그러한 모든 충돌들을 자동으로 검출한다.

때로는 충돌들을 변경 단위의 버전 히스토리에서의 분기들(forks)로서 간주하는 것이 유익하다. 만일 변경 단위의 일생에서 아무런 충돌도 일어나지 않으면, 그것의 버전 히스토리는 단순 체인(simple chain)이다 -- 각각의 변경은 이전의 것 후에 일어난다. 지식 기반 충돌의 경우에, 2개의 변경들은 동시에 일어나서, 체인이 나누어지고 버전 트리가 되게 한다.

(b) 제약 기반 충돌(Constraint-based conflicts)

독립적인 변경들이 함께 적용될 때 무결성 제약(integrity constraint)을 위반하는 경우들이 있다. 이를테면, 동일 디렉토리 내에 동일 이름을 갖는 파일을 생성하는 2개의 복제본들이 그러한 충돌을 일으킬 수 있다.

제약 기반 충돌은 (지식 기반 충돌과 마찬가지로) 2개의 독립적인 변경들을 수반하지만, 그것들은 동일한 변경 단위에 영향을 미치지 않는다. 도리어, 그것들은 상이한 변경 단위들에 영향을 미치지만 그들 사이에 존재하는 제약을 갖는다.

동기화 서비스는 변경 적용 시간에 제약 위반을 검출하고 자동으로 제약 기반 충돌들을 제기한다. 제약 기반 충돌들을 리졸브하는 데에는 통상적으로 제약을 위반하지 않도록 변경들을 수정하는 맞춤 코드(custom code)가 요구된다; 동기화 서비스는 그렇게 하기 위한 범용 메커니즘을 제공하지 않는다.

(2) 충돌 처리(Conflict Processing)

충돌이 검출되면, 동기화 서비스는 (Sync 프로파일 내의 sync 개시자(initiator)에 의해 선택된) 3개의 조치들 중 하나를 취할 수 있다: (1) 변경을 리젝트하고, 그것을 송신자에게로 반환한다; (2) 충돌을 충돌 로그 내로 로깅한다; 또는 (3) 충돌을 자동으로 리졸브한다.

만일 변경이 리젝트되면, 동기화 서비스는 마치 그 변경이 복제본에 도착하지 않은 것처럼 동작한다. 발신자에게 부정 응답(negative acknowledgement)이 송신된다. 이 레졸루션 정책은 주로 로깅 충돌들이 있음직하지 않은 (파일 서버들과 같은) 헤드 없는 복제본들(head-less replicas) 상에서 유용하다. 대신에, 그러한 복제본들은 충돌들을 리젝트함으로써 다른 것들에게 그 충돌들을 처리하도록 강요한다.

Sync 개시자들은 그들의 Sync 프로파일들에서 충돌 레졸루션을 구성한다. 동기화 서비스는 다음의 방법으로 단일 프로파일에서 다수의 충돌 리졸버들을 결합하는 것을 지원한다 - 첫째로, 시도될 충돌 리졸버들의 리스트를 그들 중 하나가 성공할 때까지 잇따라 특정함으로써, 그리고 둘째로, 충돌 리졸버들을 충돌 유형들과 관련시킴으로써, 예를 들면, 업데이트-업데이트 지식 기반 충돌들을 하나의 리졸버에게 보내고, 다른 모든 충돌들은 로그에게 보냄으로써.

(a) 자동 충돌 레졸루션

동기화 서비스는 다수의 디폴트 충돌 리졸버들을 제공한다.

이 리스트는 다음의 것들을 포함한다:

- 로컬-윈(local-wins): 로컬로 저장된 데이터와 충돌 중이면 착신되는 변경들(incoming changes)을 무시한다;

- 원격-윈(remote-wins): 착신되는 변경들과 충돌 중이면 로컬 데이터를 무시한다;

- 라스트-라이터-윈(last-writer-wins): 변경의 타임스탬프에 기초하여 변경 단위마다 로컬-윈 또는 원격-윈 중 어느 하나를 고른다(동기화 서비스는 일반적으로 클록 값들에 의지하지 않으므로, 이 충돌 리졸버는 그 규칙의 유일한 예외임에 유의한다);

- 결정적(Deterministic): 모든 복제본들에 대해 동일하지만 다른 경우에는 의미 없도록 보증되는 방식으로 승자를 고른다 - 동기화 서비스들의 일 실시예는 파트너 ID들의 사전식 비교(lexicographic comparisons)를 이용하여 이 특징을 구현한다.

게다가, ISV들은 그들 자신의 리졸버들을 구현하고 설치할 수 있다. 맞춤 충돌 리졸버들(custom conflict resolvers)은 구성 파라미터들을 수용(accept)할 수 있고, 그러한 파라미터들은 Sync 프로파일의 충돌 레졸루션 섹션의 SCA에 의해 특정되어야 한다.

충돌 리졸버가 충돌을 핸들링할 때, 그것은 (충돌하는 변경 대신에) 수행될 필요가 있는 동작들의 리스트를 런타임에 도로 반환한다. 그 후 동기화 서비스는 충돌 핸들러가 고려한 것을 포함하는 원격 지식을 적절히 조정하는 이러한 동작들을 적용한다.

레졸루션을 적용하는 동안 다른 충돌이 검출되는 것이 가능하다. 그러한 경우에, 새로운 충돌은 원래의 처리가 다시 시작하기 전에 리졸브되어야 한다.

충돌들을 아이템의 버전 히스토리에서의 분기들로 간주할 때, 충돌 레졸루션들은 2개의 분기들을 결합하여 단일 점을 형성하는 조인들(joins)으 간주될 수 있다. 따라서, 충돌 레졸루션들은 버전 히스토리들을 DAG들로 변화시킨다.

(b) 충돌 로깅(Conflict Logging)

매우 특별한 종류의 충돌 리졸버가 충돌 로거(Conflict Logger)이다. 동기화 서비스는 충돌들을 ConflictRecord 타입의 아이템들로서 로깅한다. 이들 레코드들은 충돌하고 있는 아이템들에 다시 관련된다(그 아이템들 자신이 삭제되지 않는 한). 각각의 충돌 레코드는, 충돌을 일으킨 착신되는 변경(incoming change); 충돌의 유형: 업데이트-업데이트, 업데이트-삭제, 삭제-업데이트, 삽입-삽입, 또는 제약; 및 착신되는 변경의 버전 및 그것을 송신한 복제본의 지식을 포함한다. 로깅된 충돌들은 아래에서 설명되는 바와 같이 검사 및 레졸루션을 위해 이용 가능하다.

(c) 충돌 검사 및 레졸루션

동기화 서비스는 충돌 로그를 검사하고 그 안의 충돌들의 레졸루션들을 제안하기 위해 API를 애플리케이션들에 제공한다. API는 애플리케이션이 모든 충돌들, 또는 주어진 아이템에 관련된 충돌들을 열거(enumerate)할 수 있게 한다. 그것은 또한 그런 애플리케이션들이 다음 3가지 방법 중 하나로 로깅된 충돌들을 리졸브할 수 있게 한다: (1) 원격 윈 -- 로깅된 변경을 수용하고 충돌하는 로컬 변경을 덮어쓰기(overwriting)함; (2) 로컬 윈 -- 로깅된 변경의 충돌하는 파트들을 무시함; 및 (3) 새로운 변경을 제안 -- 애플리케이션이, 그것의 의견으로, 충돌을 리졸브하는 병합(merge)을 제안하는 경우. 일단 충돌들이 애플리케이션에 의해 리졸브되면, 동기화 서비스는 충돌들을 로그로부터 제거한다.

(d) 복제본들의 컨버전스 및 충돌 레졸루션들의 전파(Convergence of replicas and Propagation of Conflict Resolutions)

복잡한 동기화 시나리오들에서, 동일한 충돌이 다수의 복제본들에서 검출될 수 있다. 이런 경우가 발생하면, 몇 가지 것들이 일어날 수 있다: (1) 충돌은 하나의 복제본 상에서 리졸브될 수 있고, 레졸루션이 다른 것에 보내진다; (2) 충돌은 양쪽 복제본들에서 자동으로 리졸브된다; 또는 (3) 충돌은 양쪽 복제본들에서 수동으로(충돌 검사 API를 통하여) 리졸브된다.

컨버전스를 보증하기 위하여, 동기화 서비스는 충돌 레졸루션들을 다른 복제본들에 전송한다. 충돌을 리졸브하는 변경이 복제본에 도착하면, 동기화 서비스는 이 업데이트에 의해 리졸브되는 로그 내의 임의의 충돌 레코드들을 자동으로 찾고 그것들을 제거한다. 이런 의미에서, 하나의 복제본에서의 충돌 레졸루션은 다른 모든 복제본들에서 구속력이 있다(binding).

만일 동일한 충돌에 대해 서로 다른 복제본들에 의해 서로 다른 승자들이 선택된다면, 동기화 서비스는 구속력 있는 충돌 레졸루션의 원리를 적용하여 2개의 레졸루션 중 하나를 골라 다른 것에 대해 자동으로 승리하게 한다. 승자는 언제나 동일한 결과를 생성하도록 보증되는 결정적 방식으로 선택된다(일 실시예는 복제본 ID 사전적 비교를 이용한다).

만일 동일한 충돌에 대해 서로 다른 복제본들에 의해 서로 다른 "새로운 변경"이 제안된다면, 동기화 서비스는 이 새로운 충돌을 특별한 충돌로 취급하고 충돌 로거(Conflict Logger)를 이용하여 그것이 다른 복제본들에 전파되지 못하도록 한다. 그러한 상황은 수동 충돌 레졸루션에 의해 일반적으로 일어난다.

2. 비저장 플랫폼 데이터 스토어들에의 동기화(Synchronizing to non-storage platform data stores)

본 발명의 저장 플랫폼의 다른 양태에 따르면, 저장 플랫폼은 저장 플랫폼이 Miscrosoft Exchange, AD, Hotmail 등과 같은 레거시(legacy) 시스템들에 동기화하도록 허용하는 Sync 어댑터들을 구현하기 위한 아키텍처를 ISV들에 제공한다. Sync 어댑터들은 아래에서 설명하는 바와 같이 동기화 서비스에 의해 제공되는 다수의 Sync 서비스로부터 이익을 얻는다.

그 이름에도 불구하고, Sync 어댑터들은 어떤 저장 플랫폼 아키텍처로의 플러그인(plug-ins)으로서 구현될 필요가 없다. 원한다면, "sync 어댑터"는 단순히 동기화 서비스 런타임 인터페이스들을 이용하여 변경 열거 및 적용과 같은 서비스들을 획득하는 임의의 애플리케이션일 수 있다.

다른 것들이 주어진 백엔드(backend)에의 동기화를 구성하고 실행하는 것을 보다 간단하게 하기 위하여, Sync 어댑터 라이터들은 상술한 바와 같이 Sync 프로파일이 주어질 경우 sync를 실행하는 표준 Sync 어댑터 인터페이스를 노출시키도록 고무된다. 프로파일은 어댑터에 구성 정보를 제공하고, 그 어댑터들 중 일부는 런타임 서비스들(예를 들면, 동기화할 폴더)을 제어하기 위해 Sync 런타임(Sync Runtime)에 전달된다.

a) Sync 서비스

동기화 서비스는 어댑터 라이터들에게 다수의 sync 서비스를 제공한다. 이 섹션에 나머지에서는, 그 위에서 저장 플랫폼이 동기화를 행하고 있는 머신을 "클라이언트"라고 부르고 어댑터가 이야기하고 있는 대상인 비저장 플랫폼 백엔드는 "서버"라고 부르는 것이 편리하다.

(1) 변경 열거(Change Enumeration)

동기화 서비스에 의해 유지되는 변경 추적 데이터에 기초하여, 변경 열거는 sync 어댑터들이 이 파트너와의 마지막 동기화가 시도된 이래로 데이터 스토어 폴더에 일어난 변경들을 쉽게 열거할 수 있게 한다.

변경들은 "앵커"(anchor) - 마지막 동기화에 관한 정보를 나타내는 불투명 구조 - 의 개념에 기초하여 열거된다. 앵커는 이전의 섹션들에서 설명한 바와 같이 저장 플랫폼 지식의 형태를 취한다. 변경 열거 서비스를 이용하는 Sync 어댑터들은 2개의 넓은 카테고리에 속한다: "저장된 앵커"(stored anchors)를 이용하는 것들 대 "공급된 앵커"(supplied anchors)를 이용하는 것들.

그 구별은 마지막 sync에 관한 정보가 어디에 - 클라이언트에 또는 서버에 - 저장되어 있는지에 기초한다. 종종 어댑터들이 이 정보를 클라이언트에 저장하는 것이 보다 용이하다 - 백 엔드는 종종 이 정보를 편리하게 저장할 수 없다. 다른 한편으로, 다수의 클라이언트들이 동일한 백엔드(backend)에 동기화하면, 이 정보를 클라이언트에 저장하는 것은 비효율적이고 어떤 경우 부정확하다 - 그것은 한 클라이언트가 이미 서버에 푸시 업(push-up)한 변경들을 다른 하나의 클라이언트가 알아채지 못하게 한다. 만일 어댑터가 서버 저장된 앵커를 사용하기를 원한다면, 그 어댑터는 변경 열거 시간에 저장 플랫폼에 다시 그것을 공급할 필요가 있다.

저장 플랫폼이 앵커를 유지하기 위해서는(로컬 저장을 위해서든 원격 저장을 위해서든), 저장 플랫폼은 서버에서 성공적으로 적용된 변경들을 알게 되어야 할 필요가 있다. 이들 및 오직 이들 변경들만이 앵커에 포함될 수 있다. 변경 열거 동안, Sync 어댑터들은 승인 인터페이스(Acknowledgement interface)를 이용하여 어떤 변경들이 성공적으로 적용되었는지를 보고할 수 있다. 동기화의 마지막에서, 공급된 앵커들을 이용하는 어댑터들은 (성공적으로 적용된 변경들 모두를 통합하는) 새로운 앵커를 판독하여 그것을 그들의 백엔드에 송신해야 한다.

흔히, 어댑터들은 그들이 저장 플랫폼 데이터 스토어에 삽입하는 아이템들과 함께 어댑터 특정 데이터를 저장할 필요가 있다. 그러한 데이터의 통상의 예들은 원격 ID들 및 원격 버전들(타임스탬프)이다. 동기화 서비스는 이 데이터를 저장하기 위한 메커니즘을 제공하고, 변경 열거는 반환되는 변경들과 함께 이 부가적인 데이터를 수신하기 위한 메커니즘을 제공한다. 이것은 대부분의 경우에 어댑터들이 데이터베이스를 다시 쿼리할 필요성을 제거한다.

(2) 변경 적용(Change Application)

변경 적용은 Sync 어댑터들이 그들의 백엔드로부터 로컬 저장 플랫폼에 수신된 변경들을 적용할 수 있게 한다. 어댑터들은 그 변경들을 저장 플랫폼 스키마로 변환할 것으로 기대된다. 도 24는 저장 플랫폼 스키마로부터 저장 플랫폼 API 클래스들의 생성되는 프로세스를 예시한다.

변경 적용의 주요 기능은 충돌들을 자동으로 검출하는 것이다. 저장 플랫폼 대 저장 플랫폼 sync의 경우에서와 같이, 충돌은 2개의 중첩하는 변경들이 서로에 대한 지식 없이 행해지는 것으로 정의된다. 어댑터들이 변경 적용을 사용할 때, 그것들은 어떤 충돌 검출이 수행되는지에 관하여 앵커를 특정해야 한다. 변경 적용은 어댑터의 지식에 의해 커버되지 않는 중첩하는 로컬 변경이 검출되면 충돌을 제기한다. 변경 열거와 유사하게, 어댑터들은 저장된 앵커 또는 공급된 앵커를 이용한다. 변경 적용은 어댑터 특정 메타데이터의 효율적인 저장을 지원한다. 그러한 데이터는 어댑터에 의해, 적용되는 변경들에 첨부될 수 있고, 동기화 서비스에 의해 저장될 수도 있다. 그 데이터는 다음 변경 열거에서 반환될 수도 있다.

(3) 충돌 레졸루션

위에서 설명된 충돌 레졸루션 메커니즘들(로깅 및 자종 레졸루션 옵션들)은 sync 어댑터들에게도 이용가능하다. Sync 어댑터들은 변경들을 적용할 때 충돌 레졸루션에 대한 정책을 특정할 수 있다. 만일 특정되면, 충돌들은 특정된 충돌 핸들러에 전달되어 리졸브될 수 있다(만일 가능하다면). 충돌들은 또한 로깅될 수 있다. 어댑터들은 로컬 변경을 백엔드에 적용하려고 시도할 때 충돌을 검출할 수도 있다. 그러한 경우에도, 어댑터는 정책에 따라서 리졸브되도록 충돌을 Sync 런타임에 전달할 수 있다. 게다가, Sync 어댑터들은 동기화 서비스에 의해 검출된 임의의 충돌들이 처리를 위해 다시 그들에게 보내지도록 요구할 수 있다. 이것은 백엔드가 충돌들을 저장하고 리졸브할 수 있는 경우에 특히 편리하다.

b) 어댑터 구현

어떤 "어댑터들"은 단순히 런타임 인터페이스들을 이용하는 애플리케이션들이지만, 어댑터들은 표준 어댑터 인터페이스들을 구현하도록 고무된다. 이들 인터페이스는 Sync 제어 애플리케이션들로 하여금, 어댑터가 주어진 Sync 프로파일에 따라서 동기화를 수행하도록 요구하고; 진행중인 동기화를 취소하고; 진행중인 sync에 대한 진행 보고(progress reporting)(퍼센트 완료)를 수신할 수 있게 한다.

3. 보안

동기화 서비스는 저장 플랫폼에 의해 구현되는 보안 모델을 가능한 한 적게 도입하려고 노력한다. 동기화에 대한 새로운 권리를 정의하기보다는, 기존의 권리들이 사용된다. 구체적으로,

- 데이터 저장 아이템을 판독할 수 있는 누구든지 그 아이템에 대한 변경들을 열거할 수 있고;
- 데이터 저장 아이템에 기입할 수 있는 누구든지 그 아이템에 대한 변경들을 적용할 수 있고;
- 데이터 저장 아이템을 확장할 수 있는 누구든지 sync 메타데이터를 그 아이템과 관련시킬 수 있다.

동기화 서비스는 안전한 저작자 정보(authorship information)를 유지하지 않는다. 사용자 U에 의해 복제본 A에서 변경이 행해지고 복제본 B에 전달(forward)되는 경우, 그 변경이 원래 A에서(또는 U에 의해) 행해졌다는 사실이 소실된다. 만일 B가 이 변경을 복제본 C에 전달하면, 이것은 A의 권한이 아니라 B의 권한 하에서 행해지는 것이다. 이것은 다음의 제한을 초래한다: 만일 복제본이 그 자신의 변경들을 아이템에 행하도록 위임되지 않으면, 그것은 다른 것들에 의해 행해진 변경들을 전달할 수 없다.

동기화 서비스가 개시될 때, 그것은 Sync 제어 애플리케이션에 의해 행해진다. 동기화 서비스는 SCA의 아이덴티티를 가장하여 그 아이덴티티 하에서 모든 동작들(로컬로 그리고 원격으로) 수행한다. 예시하기 위하여, 사용자 U가 로컬 동기화 서비스로 하여금 사용자 U가 판독 액세스할 수 없는 아이템들에 대한 원격 저장 플랫폼으로부터 변경들을 검색하게 할 수 없다는 것을 주시하자.

4. 관리능력(Manageability)

복제본들의 분산 공동체를 모니터링하는 것은 복잡한 문제이다. 동기화 서비스는 복제본들의 상태에 관한 정보를 수집하고 분배하기 위해 "스위프"(sweep) 알고리즘을 사용한다. 스위프 알고리즘의 속성들은 모든 구성된 복제본들에 관한 정보가 결국 수집되고 실패(비응답) 복제본들이 검출되는 점을 보장한다.

이 전공동체적(community-wide) 모니터링 정보는 모든 복제본에서 이용 가능하게 된다. 이 모니터링 정보를 검사하고 관리적 결정을 행하기 위해 임의로 선택된 복제본에서 모니터링 도구들이 실행될 수 있다. 영향받은 복제본들(affected replicas)에서 직접 임의의 구성 변경들이 행해져야 한다.

H. 전통적 파일 시스템 상호 운용성

위에서 언급한 바와 같이, 본 발명의 저장 플랫폼은, 적어도 일부 실시예들에서, 컴퓨터 시스템의 하드웨어/소프트웨어 인터페이스 시스템의 필수적인 부분으로서 구현되도록 의도되어 있다. 예를 들면, 본 발명의 저장 플랫폼은 마이크로소프트 윈도즈 계열 운영 시스템들과 같은 운영 시스템의 필수적인 부분으로서 구현될 수 있다. 그 입장에서, 저장 플랫폼 API는 운영 시스템 API들의 일부가 되고 그를 통하여 애플리케이션 프로그램들이 운영 시스템과 상호 작용한다. 따라서, 저장 플랫폼은 애플리케이션 프로그램들이 운영 시스템 상에 정보를 저장하는 수단이 되고, 그러므로 저장 플랫폼의 아이템 기반 데이터 모델은 그러한 운영 시스템의 전통적 파일 시스템을 대체한다. 예를 들면, 마이크로소프트 윈도즈 계열 운영 시스템들에서 구현될 때, 저장 플랫폼은 그 운영 시스템에서 구현된 NFTS 파일 시스템을 대체할 수 있다. 현재, 애플리케이션 프로그램들은 윈도즈 계열 운영 시스템에 의해 노출된 Win32 API들을 통하여 NTFS 파일 시스템의 서비스들에 액세스한다.

그러나, 본 발명의 저장 플랫폼으로 NTFS 파일 시스템을 완전히 대체하려면 기존의 Win32 기반 애플리케이션 프로그램들의 재코딩(recoding)이 필요할 것이고 그러한 재코딩은 바람직하지 않을 것임을 인지하면, 본 발명의 저장 플랫폼이 NTFS와 같은 기존의 파일 시스템들과의 상호 운용성을 제공하는 것이 유익할 것이다. 그러므로, 본 발명의 일 실시예에서, 저장 플랫폼은 Win32 프로그래밍 모델에 의지하는 애플리케이션 프로그램들이 저장 플랫폼의 데이터 스토어뿐만 아니라 전통적인 NTFS 파일 시스템의 콘텐츠에 액세스할 수 있게 한다. 이 때문에, 저장 플랫폼은 용이한 상호 운용성을 돕기 위하여 Win32 명명 규칙의 슈퍼세트인 명명 규칙을 사용한다. 또한, 저장 플랫폼은 Win32 API를 통하여 저장 플랫폼에 저장된 파일들 및 디렉토리들에 액세스하는 것을 지원한다.

이 기능의 부가적인 상세는 본 명세서의 앞에서 참고로 통합된 관련 출원들에서 얻을 수 있다.

I. 저장 플랫폼 API

저장 플랫폼은 애플리케이션 프로그램들이 위에서 논의된 저장 플랫폼의 특징들 및 능력들에 액세스할 수 있게 하고 또한 데이터 스토어에 저장된 아이템들에 액세스할 수 있게 하는 API를 포함한다. 이 섹션은 본 발명의 저장 플랫폼의 저장 플랫폼 API의 일 실시예를 설명한다. 이 기능에 관한 상세는 본 명세서의 앞에서 참고로 통합된 관련 출원들에서 얻을 수 있고, 편의를 위하여 이 정보의 일부가 아래에 요약되어 있다.

도 18을 참조하면, 컨테인먼트 폴더(Containment Folder)는 다른 아이템들에 대한 홀딩 관계를 포함하는 아이템이고 파일 시스템 폴더의 공통 개념의 등가물이다. 각각의 아이템은 적어도 하나의 컨테인먼트 폴더 내에 "포함"(contained)된다.

도 19는 본 발명에 따라, 저장 플랫폼 API의 기본 아키텍처를 예시한다. 저장 플랫폼 API는 로컬 데이터 스토어(302)와 이야기하기 위해 SQLClient(1900)를 이용하고 또한 원격 데이터 스토어(예를 들면 데이터 스토어(340))와 이야기하기 위해 SQLClient(1900)를 이용할 수도 있다. 로컬 스토어(302)는 또한 DQP(Distributed Query Processor)를 이용하거나 또는 위에서 설명한 저장 플랫폼 동기화 서비스("Sync")를 통하여 원격 데이터 스토어(340)와 이야기할 수도 있다. 저장 플랫폼 API(322)는 또한, 위에서도 설명한 바와 같이, 데이터 스토어 통지를 위한 브리지 API로 기능하여, 애플리케이션의 가입들(subscriptions)을 통지 엔진(332)에 전달하고 통지들을 애플리케이션(예를 들면, 애플리케이션들(350a, 350b, 또는 350c))에 라우팅한다. 일 실시예에서, 저장 플랫폼 API(322)는 또한 Microsoft Exchange 및 AD 내의 데이터에 액세스할 수 있도록 제한된 "제공자"(provider) 아키텍처를 정의할 수도 있다.

도 20은 저장 플랫폼 API의 여러 컴포넌트들을 개략적으로 나타낸다. 저장 플랫폼 API는 다음의 컴포넌트들로 이루어진다: 저장 플랫폼 엘리먼트 및 아이템 타입들을 나타내는 데이터 클래스들(2002); (2) 객체 지속성(object persistence)을 관리하고 지원 클래스들(2006)을 제공하는 런타임 프레임워크(2004); 및 (3) 저장 플랫폼 스키마들로부터 CLR 클래스들을 생성하기 위해 이용되는 도구들(2008).

주어진 스키마로부터 생기는 클래스들의 계층 구성은 해당 스키마 내의 타입들의 계층 구성을 직접 반영한다. 일례로서, 도 21A 및 도 21B에 도시된 바와 같이 Contacts 스키마에서 정의된 아이템 타입들을 고려해보자.

도 22는 동작 중인 런타임 프레임워크를 예시한다. 런타임 프레임워크는 다음과 같이 동작한다:

1. 애플리케이션(350a, 350b, 또는 350c)이 저장 플랫폼 내의 아이템에 바인딩한다.

2. 프레임워크(2004)는 바인딩된 아이템에 대응하는 ItemContext 객체(2202)를 생성하고 그것을 애플리케이션에 반환한다.

3. 애플리케이션은 이 ItemContext 상에 Find를 제시하여 아이템들의 컬렉션(collection)을 수취한다; 반환된 컬렉션은 개념적으로 (관계들에 기인하는) 객체 그래프(2204)이다.

4. 애플리케이션은 데이터를 변경, 삭제, 및 삽입한다.

5. 애플리케이션은 Update() 메서드를 호출함으로써 변경들을 저장(save)한다.

도 23은 "FindAll" 동작의 실행을 예시한다.

도 24는 저장 플랫폼 스키마로부터 저장 플랫폼 API 클래스들이 생성되는 프로세스를 예시한다.

도 25는 파일 API가 기초하고 있는 스키마를 예시한다. 저장 플랫폼 API는 파일 객체들을 취급하기 위한 네임스페이스를 포함한다. 이 네임스페이스는 System.Storage.Files라고 불린다. System.Storage.Files 내의 클래스들의 데이터 멤버들은 저장 플랫폼 스토어에 저장된 정보를 직접 반영한다; 이 정보는 파일 시스템 객체들로부터 "프로모트(promote)"되거나 또는 Win32 API를 사용하여 본래부터 생성될 수 있다. System.Storage.Files 네임스페이스는 2개의 클래스를 갖는다: FileItem 및 DirectoryItem. 이들 클래스의 멤버들 및 그것들의 메서드들은 도 25의 스키마 도면을 봄으로써 쉽게 추측된다. FileItem 및 DirectoryItem은 저장 플랫폼 API로부터 읽기 전용이다. 그것들을 수정하기 위해서는, Win32 API 또는 System.IO 내의 클래스들을 사용해야 한다.

API들과 관련하여, 프로그래밍 인터페이스(또는 더 간단히 말해서, 인터페이스)는 하나 이상의 코드 세그먼트(들)가 하나 이상의 다른 코드 세그먼트(들)에 의해 제공된 기능과 통신하거나 그 기능에 액세스할 수 있게 하기 위한 임의의 메커니즘, 프로세스, 프로토콜로 간주될 수 있다. 다르게는, 프로그래밍 인터페이스는 시스템의 다른 컴포넌트(들)의 하나 이상의 메커니즘(들), 메서드(들), 함수 호출(들), 모듈(들) 등에 통신 결합가능한 시스템의 컴포넌트의 하나 이상의 메커니즘(들), 메서드(들), 함수 호출(들), 모듈(들), 객체(들) 등으로 간주될 수 있다. 바로 앞의 문장에서 "코드 세그먼트"(segment of code)라는 용어는, 적용된 용어에 상관없이 또는 코드 세그먼트들이 따로따로 컴파일되든 아니든, 또는 코드 세그먼트들이 소스, 중간 혹은 객체 코드로서 제공되든, 코드 세그먼트들이 런타임 시스템 또는 프로세스에서 이용되든 아니든, 또는 그것들이 동일한 머신 상에 있든 서로 다른 머신들 상에 있든 또는 다수의 머신들에 걸쳐서 분산되어 있든, 또는 코드 세그먼트들에 의해 표현되는 기능이 전적으로 소프트웨어로 구현되든, 전적으로 하드웨어로 구현되든, 또는 하드웨어와 소프트웨어의 조합으로 구현되든 상관없이, 코드의 하나 이상의 명령 또는 라인들을 포함하도록 의도되어 있고, 예를 들면, 코드 모듈들, 객체들, 서브루틴들, 함수들 등을 포함한다.

관념적으로, 프로그래밍 인터페이스는 도 30A 또는 도 30A에 도시된 바와 같이, 포괄적으로 생각될 수 있다. 도 30A는 시스템의 제1 및 제2 코드 세그먼트들이 통신하는 도관(conduit)으로서 인터페이스 Interface1을 예시한다. 도 30B는 시스템의 제1 및 제2 세그먼트들이 매체 M을 통하여 통신할 수 있게 하는 인터페이스 객체들 I1 및 I2(제1 및 제2 코드 세그먼트들의 파트일 수도 아닐 수도 있음)을 포함하는 것으로 인터페이스를 예시한다. 도 30B의 보기에서는, 인터페이스 객체들 I1 및 I2를 동일 시스템의 별개의 인터페이스들로 생각할 수도 있고 객체들 I1 및 I2에 매체 M을 더한 것이 인터페이스를 구성하는 것을 생각할 수도 있다. 도 30A 및 30B는 양방향 흐름 및 그 흐름의 양쪽에서의 인터페이스를 보여주지만, 어떤 구현들은 한 방향으로의 정보 흐름만을 갖거나(또는 아래에서 설명하는 바와 같이 아무런 정보 흐름도 없거나) 한 쪽에 하나의 인터페이스 객체만을 가질 수 있다. 제한이 아니라 예로서, 애플리케이션 프로그래밍 인터페이스(API), 엔트리 포인트(entry point), 메서드, 함수, 서브루틴, 원격 프로시저 호출(remote procedure call), 및 컴포넌트 객체 모델(COM: component object model) 인터페이스와 같은 용어들이 프로그래밍 인터페이스의 정의 내에서 두루 포함된다.

그러한 프로그래밍 인터페이스의 양태들은 제1 코드 세그먼트가 제2 코드 세그먼트에 정보(여기서 "정보"는 넓은 의미로 사용되고 데이터, 커맨드, 요구(requests) 등을 포함한다)를 전송하는 메서드; 제2 코드 세그먼트가 상기 정보를 수신하게 하는 메서드; 및 상기 정보의 구조, 시퀀스, 구문, 조직, 스키마, 타이밍 및 콘텐츠를 포함할 수 있다. 이와 관련하여, 하위 전송 매체(underlying transport medium) 자체는, 그 매체가 유선이든 무선이든, 또는 양자의 조합이든, 상기 정보가 상기 인터페이스에 의해 정의된 방식으로 전송되기만 한다면, 인터페이스의 동작에 중요하지 않을 수 있다. 어떤 경우에, 정보는 종래의 의미에서 한 방향 또는 양방향으로 전달되지 않을 수도 있다. 왜냐하면 하나의 코드 세그먼트가 제2의 코드 세그먼트에 의해 수행되는 기능에 단순히 액세스하는 경우와 같이, 정보 전송은 다른 메커니즘(예를 들면, 코드 세그먼트들 간의 정보 흐름과 별도로 버퍼, 파일 등에 배치된 정보)을 통하거나 또는 존재하지 않을 수 있기 때문이다. 이들 양태들의 어느 것이든 또는 전부가 주어진 상황에서, 예를 들면 코드 세그먼트들이 느슨하게 연결된 또는 긴밀하게 연결된 구성에서 시스템의 일부인지에 따라서 중요할 수 있고, 따라서 이 리스트는 예시적이고 비제한적인 것으로 간주되어야 할 것이다.

프로그래밍 인터페이스의 이 개념은 숙련된 당업자에게 공지되어 있고 본 발명의 전술한 상세한 설명으로부터 분명하다. 그러나, 프로그래밍 인터페이스를 구현하는 다른 방법들이 있고, 명백히 배제되지 않는 한, 이것들도 본 명세서의 말미에 제시된 청구항들에 의해 포함되도록 의도된다. 그런 다른 방법들은 도 30A 및 30B의 극히 단순화한 보기(views)보다 난재해거나 복잡해보일 수 있지만, 그럼에도 불구하고 그것들은 유사한 기능을 수행하여 동일한 전체 결과를 성취한다. 이제부터 프로그래밍 인터페이스의 몇몇 예시적인 대체 구현예들을 간단히 설명해보겠다.

팩터링(factoring): 하나의 코드 세그먼트로부터 다른 것으로의 통신은 그 통신을 다수의 이산 통신들(discrete communications)로 분해함으로써 간접적으로 성취될 수 있다. 이것은 도 31A 및 도 31B에 개략적으로 도시되어 있다. 도시된 바와 같이, 몇몇 인터페이스들은 기능의 분할 가능한 세트들의 관점으로 설명될 수 있다. 따라서, 도 30A 및 도 30B의 인터페이스 기능은, 수학적으로 24, 즉 2×2×3×2를 제공할 수 있는 것처럼, 팩터링되어 동일한 결과를 성취할 수 있다. 따라서, 도 31A에 예시된 바와 같이, 인터페이스 Interface1에 의해 제공된 기능은 그 인터페이스의 통신들을 다수의 인터페이스 Interface1A, Interface1B, Interface1C 등으로 변환하도록 세분될 수 있으나, 동일한 결과를 실현한다. 도 31B에 예시된 바와 같이, 인터페이스 I1에 의해 제공된 기능은 동일한 결과를 성취하면서 다수의 인터페이스 I1a, I1b, I1c 등으로 세분될 수 있다. 마찬가지로, 제1 코드 세그먼트로부터의 정보를 수신하는 제2 코드 세그먼트의 인터페이스 I2는 다수의 인터페이스 I2a, I2b, I2c 등으로 팩터링될 수 있다. 팩터링할 때, 제1 코드 세그먼트와 함께 포함되는 인터페이스의 수는 제2 코드 세그먼트와 함께 포함되는 인터페이스의 수와 일치할 필요는 없다. 도 31A 및 도 31B의 어느 경우든, 인터페이스 Interface1 및 I1의 기능적 의미는 각각 도 30A 및 도 30B에서와 여전히 동일하다. 인터페이스들의 팩터링은 또한 그 팩터링을 알아내기 곤란하도록 결합(associtave), 교환(commutative), 및 다른 수학적 특성들을 따를 수 있다. 이를

테면, 동작들의 순서화는 중요하지 않을 수 있고, 따라서, 인터페이스에 의해 수행되는 기능은 그 인터페이스에 도달하기 훨씬 전에 코드 또는 인터페이스의 다른 피스(piece)에 의해 수행될 수 있고, 또는 시스템의 별도의 컴포넌트에 의해 수행될 수 있다. 또한, 프로그래밍 기술 분야의 통상의 기술을 가진 자라면 동일한 결과를 성취하는 다른 함수 호출들을 행하는 갖가지 방법들이 있다는 것을 알 수 있다.

재정의(Redefinition): 일부 경우에, 프로그래밍 인터페이스의 어떤 특징들(aspects)(예를 들면, 파라미터들)을 무시하거나, 부가하거나 또는 재정의하면서도 의도된 결과를 성취하는 것이 가능할 수 있다. 이것은 도 32A 및 도 32B에 예시되어 있다. 예를 들면, 도 30A의 인터페이스 Interface1은 3개의 파라미터인 입력(input), 정밀도(precision) 및 출력(output)을 포함하고, 제1 코드 세그먼트로부터 제2 코드 세그먼트로 발행되는 호출인, 함수 호출 Square(입력, 정밀도, 출력)를 포함한다고 가정하자. 만일 중앙의 파라미터인 정밀도가, 도 32A에 도시된 바와 같이, 주어진 시나리오에서 중요하지 않다면, 그것은 무시되거나 또는 의미 없는(이 상황에서) 파라미터로 대체되는 것도 좋을 수 있다. 또한 중요하지 않은 부가적인 파라미터를 부가할 수도 있다. 어느 경우든, 입력이 제2 코드 세그먼트에 의해 제곱(square)된 후에 출력이 반환되는 한, square의 기능은 성취될 수 있다. 정밀도는 컴퓨터 시스템의 어떤 다운스크림 또는 다른 부분에게 의미 있는 파라미터일지도 모르지만, 제곱(square)을 계산하는 한정된 목적을 위해 정밀도가 필요하지 않다고 일단 인지되면, 그것은 대체되거나 무시될 수 있다. 예를 들면, 유효 정밀도 값을 전달하는 대신에, 결과에 악영향을 미치지 않으면 생일 날짜와 같은 무의미한 값이 전달될 수 있다. 마찬가지로, 도 32B에 도시된 바와 같이, 인터페이스 I1은 그 인터페이스에 파라미터들을 부가하거나 또는 무시하도록 재정의된, 인터페이스 I1'에 의해 대체된다. 인터페이스 I2는 마찬가지로 불필요한 파라미터들, 즉 다른 곳에서 처리될 수 있는 파라미터들을 무시하도록 재정의된, 인터페이스 I2'에 의해 대체될 수 있다. 여기에서 요점은 일부 경우에 프로그래밍 인터페이스가 어떤 목적을 위해 필요하지 않은, 파라미터들과 같은, 특징들을 포함할 수 있고, 따라서 그것들은 무시되거나 재정의되거나, 또는 다른 목적을 위해 다른 곳에서 처리될 수 있다는 점이다.

인라인 코딩(Inline Coding): 2개의 별개의 코드 모듈들 간의 "인터페이스"가 형태를 변경하도록 그 코드 모듈들의 기능의 일부 또는 전부를 병합하는 것도 실행할 수 있을 것이다. 예를 들면, 도 30A 및 도 30B의 기능은 각각 도 33A 및 도 33B의 기능으로 변환될 수 있다. 도 33A에서, 도 30A의 이전의 제1 및 제2 코드 세그먼트들은 그들 모두를 포함하는 하나의 모듈로 병합된다. 이 경우에, 코드 세그먼트들은 여전히 서로 통신할 수 있지만 인터페이스는 단일 모듈에 보다 적합한 형태로 적용될 수 있다. 따라서, 예를 들면, 형식적인 호출(Call) 및 반환(Return) 문장들은 더 이상 필요하지 않을 수 있지만, 인터페이스 Interface1에 따른 마찬가지의 처리 또는 응답(들)이 여전히 유효할 수 있다. 마찬가지로, 도 33B에 도시된 바와 같이, 도 30B로부터의 인터페이스 I2의 일부(또는 전부)가 인터페이스 I1에 인라인으로 기입되어 인터페이스 I1"을 형성할 수 있다. 예시된 바와 같이, 인터페이스 I2는 I2a 및 I2b로 분할되고, 인터페이스 부분 I2a는 인터페이스 I1과 함께 인라인으로 코딩되어 인터페이스 I1"을 형성하였다. 구체적인 예를 위해, 도 30B로부터의 인터페이스 I1이 함수 호출 square(입력, 출력)을 수행하는 것을 생각해보자. 여기서, 상기 함수 호출 square(입력, 출력)은 인터페이스 I2에 의해 수신되고, 이 인터페이스 I2는 제2 코드 세그먼트에 의해 입력과 함께 전달된 값을 처리(그것을 제곱(square))한 후에, 제곱된 결과를 출력과 함께 도로 전달한다. 그런 경우에, 제2 코드 세그먼트에 의해 수행되는 처리(입력을 제곱)는 인터페이스에 대한 호출 없이 제1 코드 세그먼트에 의해 수행될 수 있다.

분리(Divorce): 하나의 코드 세그먼트로부터 다른 것으로의 통신은 그 통신을 다수의 이산 통신으로 분해함으로써 성취될 수 있다. 이것은 도 34A 및 도 34B에 개략적으로 도시되어 있다. 도 34A에 도시된 바와 같이, 미들웨어의 하나 이상의 피스(들)(이것들은 원본 인터페이스로부터의 기능 및/또는 인터페이스 기능들을 분리(divorce)하므로, 분리 인터페이스(들)(Diverce Interface(s))이라 함)이 제공되어 제1 인터페이스 Interface1 상에서 통신들을 변환하여 그것들을 상이한 인터페이스, 이 경우 인터페이스들 Interface2A, Interface2B 및 Interface2C에 적합하게 한다. 이것은 예를 들면 Interface1 프로토콜에 따라서 운영 시스템과 통신하도록 설계된 애플리케이션들의 설치된 베이스가 있지만, 그 운영 시스템이 상이한 인터페이스, 이 경우에는 인터페이스들 Interface2A, Interface2B 및 Interface2C를 사용하도록 변경되는 경우에 행해질 수 있다. 요점은 제2 코드 세그먼트에 의해 사용되는 원본 인터페이스가 더 이상 제1 코드 세그먼트에 의해 사용되는 인터페이스와 호환되지 않도록 변경되고, 따라서 이전 인터페이스와 새로운 인터페이스를 호환되게 하기 위하여 매개물(intermediary)이 사용된다는 점이다. 마찬가지로, 도 34B에 도시된 바와 같이, 인터페이스 I1로부터 통신들을 수신하는 분리 인터페이스 DI1를 갖고, 예를 들면 DI2와 함께 동작하도록 재설계된 인터페이스들 I2a 및 I2b에 인터페이스 기능을 송신하는 분리 인터페이스 DI2를 갖는 제3 코드 세그먼트가 도입될 수 있지만, 동일한 기능적 결과를 제공한다. 마찬가지로, DI1 및 DI2는 함께 동작하여 도 30B의 인터페이스 I1 및 I2의 기능을 새로운 운영 시스템으로 변환하지만, 동일하거나 유사한 기능적 결과를 제공한다.

재기입(Rewriting): 또 다른 가능한 변형은 인터페이스 기능을 어떤 다른 것으로 대체하면서도 동일한 전체 결과를 성취하도록 코드를 동적으로 재기입하는 것이다. 예를 들면, 중간 언어(예를 들면, 마이크로소프트 IL, Java ByteCode 등)로 제시된 코드 세그먼트가 (예컨대 .Net 프레임워크, Java 런타임 환경, 또는 다른 유사한 런타임 타입 환경들에 의해 제공된 것과 같은) 실행 환경에서 JIT(Just-in-Time) 컴파일러 또는 인터프리터로 제공되는 시스템이 존재할 수 있다. JIT 컴파일

러는 제1 코드 세그먼트로부터의 통신들을 제2 코드 세그먼트로 동적으로 변환하도록, 즉 그것들을 제2 코드 세그먼트(원본 또는 상이한 제2 코드 세그먼트)에 의해 요구될 수 있는 상이한 인터페이스에 적합하게 하도록 기입될 수 있다. 이것은 도 35A 및 도 35B에 도시되어 있다. 도 35A에서 알 수 있는 바와 같이, 이 방법은 위에서 설명된 분리(Divorce) 시나리오와 유사하다. 이것은 예를 들면, 애플리케이션들의 설치된 베이스가 Interface 1 프로토콜에 따라서 운영 시스템과 통신하도록 설계되어 있지만, 운영 시스템이 상이한 인터페이스를 사용하도록 변경되는 경우에 행해질 수 있다. JIT 컴파일러는 설치된 베이스 애플리케이션들로부터의 진행중인 통신들을 운영 시스템의 새로운 인터페이스에 적합하게 하기 위해 사용될 수 있다. 도 35B에 도시된 바와 같이, 인터페이스(들)을 동적으로 재기입하는 이 방법은 또한 인터페이스(들)를 동적으로 팩터링하거나 또는 다르게 변경하기 위해 적용될 수 있다.

또한 대안 실시예들을 통하여 인터페이스로서 동일하거나 유사한 결과를 성취하기 위한 상술한 시나리오들은 다양한 방법으로, 직렬로 및/또는 병렬로, 또는 다른 중재 코드에 의해 조합될 수도 있다는 것에 주목해야 할 것이다. 따라서, 위에서 제시된 대안 실시예들은 상호 배타적이지 않고 도 30A 및 도 30B에서 제시된 일반적 시나리오들과 동일하거나 동등한 시나리오들을 생성하도록 혼합되고, 매칭되고, 결합될 수 있다. 또한, 대부분의 프로그래밍 구성들에서와 같이, 여기에서 설명되어 있지 않지만, 그럼에도 불구하고 본 발명의 사상 및 범위에 의해 표현되는 인터페이스의 동일하거나 유사한 기능을 성취하는 다른 유사한 방법들이 있다는 것에 주목한다. 즉, 그것은 적어도 부분적으로 인터페이스의 값의 아래에 있는 인터페이스에 의해 표현되는 기능, 및 인터페이스의 값의 아래에 있는 인터페이스에 의해 가능하게 되는 유리한 결과들이라는 것에 주목한다.

Ⅲ. 이미지 스키마 및 하위 스키마(이미지 스키마 세트)

여기에 개시된 본 발명의 갖가지 실시예들에서, 이미지들(예를 들면, JPEG, TIFF, 비트맵 등)은 코어 플랫폼 객체들("이미지 아이템들" 또는, 보다 간단히, "이미지들")로서 취급되고, 본 발명은 이 시스템에서 이미지의 확장 가능한 표현 - 즉, 이미지의 특징들 및 해당 이미지가 시스템 내의 다른 아이템들(다른 이미지들을 포함하지만 이에 한정되지 않음)에 어떻게 관련되는지 - 을 제공하는 "이미지 스키마"(Image Schema)를 포함한다 이를 위하여, 이미지 스키마는 시스템 내의 이미지들에 대한 속성들(properties), 거동들(behaviors), 및 관계들(relationships)을 정의하고, 이 스키마는 또한 이미지들에 관한 규칙들, 예를 들면, 데이터 특정 이미지들이 무엇을 포함해야 하는지, 데이터 특정 이미지들이 선택 사양으로 무엇을 포함할 수 있는지, 특정 이미지들이 어떻게 확장될 수 있는지, 등등을 정의한다. 이미지 스키마는 (GIF, TIFF, JPEG, 및 다른 공지된 이미지 객체 타입들을 포함하는) 이미지들을 제시하기 위한 포맷들의 속성들은 물론 이미지의 의미 콘텐츠(semantic contents)를 나타내는 속성들을 포함하는, 여러 다른 종류의 이미지들을 나타내기 위해 필요한 타입 정보를 포함한다. 본 발명의 갖가지 실시예들의 이미지 스키마는 그를 근거로 모든 이미지 관련 기능이 구축되는 토대이다.

이미지 스키마 외에, "Photo"(사진), "Analysis Properties"(분석 속성), 및 "Locaton"(장소) 아이템들에 대한 관련 종속 스키마들(subordinate schemas)이 또한 제공되어 여기에서 논의되고(전체적으로, "이미지 스키마 세트"), 본 발명의 어떤 실시예들은 하나 이상의 이들 종속 스키마를 포함한다. 사진 스키마는 사진 객체("사진 아이템" 또는, 간단히 "Photo")의 확장 가능한 표현이고, 여기서 사진 아이템 타입은 이미지 아이템 타입의 서브타입이다. 분석 속성 스키마(AP 스키마)는 사진들에 대한 자동 얼굴 인식, 이미지 유사도(image similarity) 등등과 같은 진보된 비교 기능들을 가능케 할 사진 아이템에 대한 분석 속성들(AP들)의 확장 가능한 표현이다. 장소 스키마는 사진 아이템에 대한 물리적(지리적) 위치의 확장 가능한 표현이다.

도 36A 및 도 36B는 본 발명의 갖가지 실시예들에 대한 이미지 스키마(아이템들 및 속성들)을 (도 7의) 베이스 스키마 및 (도 8A 및 도 8B의) 코어 스키마로부터의 선택된 아이템들과 함께 예시하여 각각의 스키마 간의 관계를 보여준다. (편의상, 예시에서 개개의 속성들은 생략되었다.)

A. 이미지 스키마

앞에서 논의한 바와 같이, 이미지 스키마는 여러 다른 종류의 이미지 아이템들을 나타내기 위해 필요한 아이템들(Items), 속성들(Properties), 및 관계들(Relationships)을 포함한다. 이것은 특정 이미지 타입들(GIF, TIFF, JPEG 등)을 제시하기 위한 원시 파일 포맷들의 속성들은 물론 이미지의 의미 콘텐츠를 나타내는 속성들을 포함한다. 이를 위하여, 임의의 이미지 아이템에 있어서, 이미지 타입(Image Type)은 모든 이미지들에 의해 공유되는 베이스 아이템 타입이고, 도 36에서 예시된 바와 같이 Core.Document 아이템 타입(즉, "Core" 스키마 내의 "Document" 타입)의 직접 확장이다.
(Core.Document 타입은 도 8A에 예시된 코어 스키마의 일부이다. 이미지 타입은 이미지를 일반적으로 기술하고 또한 포맷에 상관없이 모든 이미지들에 적용 가능한 필드들을 포함한다. 이미지 스키마에 대한 주요 아이템 타입은 이미지 타입이고, 다음은 이미지 타입에 대한 몇몇 필드들이다.

속성 이름	설명
Rating	이 그림의 레이팅, 이것은 소비자 레이팅, 부모 레이팅(parental ratings) 등을 나타낼 수 있다.
SizeX	화소로 나타낸 이미지의 폭
SizeY	화소로 나타낸 이미지의 높이
BitDepth	화소당 비트 깊이(1, 8, 16, 24 등)
ColorSpace	이 그림에 대한 색공간의 이름. 예를 들면, "sRGB".
ColorSpaceVersion	이 그림에 대한 색공간의 버전. 예를 들면, "1.2".
TimesPrinted	이 이미지가 인쇄된 횟수. 이 필드는 애플리케이션들에 의해 유지된다.
RegionOfInterest	이미지 내의 관심 영역들의 컬렉션. 예를 들면 얼굴 또는 중요한 랜드마크를 갖는 사각형들.
History	이 이미지의 히스토리. 내가 어떤 필터를 적용하였는지? 내가 누구에게 그것을 메일하였는지? 내가 언제 그것을 인쇄하였는지?
DigitalNegativeID	이 필드는 동일 이미지의 모든 복사본에 대해 동일할 것이다. 이미지의 복사본이 편집되면 사용자는 그것을 동일하게 유지하기를 원하는지, 아니면 이 버전을 새로운 디지털 네거티브(Digital Negative)로 생각하는지를 결정한다.
OtherVersions	동일 그림의 다른 버전들에의 링크들의 컬렉션. 다운레벨 버전들, 소규모 버전들 등. 이것은 임베딩 관계이어야 한다. 만일 이미지가 변경되면 모든 그것의 자식 이미지들도 애플리케이션에 의해 변경된 것으로 간주되어야 할 것이다. 이미지가 삭제되면, 모든 그것의 자식 이미지들은 WinFS에 의해 삭제될 것이다.
MetadataLifecycle	메타데이터 라이프사이클을 제어하는 변수들의 세트.

이미지 스키마는 또한 다음과 같이 Base.PropertyBase로부터 확장하는 부가적인 속성들(네스트된 엘리먼트들)을 포함한다:

- "Region": Region 속성은 이미지 내의 영역을 나타내고, 다음의 필드들을 포함한다:

속성 이름	설명
Left	화소로 나타낸 영역의 좌표
Top	화소로 나타낸 영역의 좌표
Right	화소로 나타낸 영역의 좌표
Bottom	화소로 나타낸 영역의 좌표

- "RegionOfInterest": 이 속성은 이미지 내의 특별히 흥미있는 정보를 나타내고, 다음의 필드들을 포함한다:

속성 이름	설명
Person	흥미있는 이 영역 내에서 표현되는 사람들의 리스트. 만일 이 영역이 사람이 아닌 어떤 것(랜드마크 등)을 포함하면 이 필드는 NULL일 것이다.
Region	이미지 내의 관심 영역들의 컬렉션. 예를 들면 얼굴 또는 중요한 랜드마크를 갖는 사각형. 만일 NULL이면 그 사각형은 미지의 것이다(그들이 어디 있는지 알지 못하고 그림 내에서 사람들을 캡처하기 위해 사용됨).
Confidence	메타데이터 제공자에 의해 반환되는, 이 특정 매치의 신뢰(confidence)

게다가, 이미지 스키마는 또한 다음과 같이 도 36A에 예시된 바와 같이 이미지 아이템과 다른 아이템들 간의 일련의 관계들을 포함할 수 있다:

- "PersonReference": 이미지는 특정 사진(Photo) 내의 사람(들)을 나타내기 위해 콘택트 또는 기본(Principal) 아이템(도 8A의 Core.Principal)에의 하나 이상의 PersonReference 링크들을 가질 수 있고, 다음의 필드를 포함한다:

속성 이름	설명

DisplayName	참조된 사람(Person)의 디스플레이 이름. 이 필드는 링크 장체가 댕글링인 경우에 특히 유용하지만, 링크가 댕글링이 아닌 경우에도, 이 필드는 이 특정 이미지 내에서 링크된 콘택트 내의 사람이 얼마나 정확히 참조되었는지를 나타내기 위해 사용될 수 있다.

● "EventReference": 이미지는 또한 특정 사진에 대한 이벤트를 나타내기 위해 Event 아이템(도 8A의 Core.Event)에의 EventReference 링크들을 가질 수 있고, 다음의 필드를 포함한다:

특성 이름	설명
DisplayName	참조된 이벤트(Event)의 디스플레이 이름. 이 필드는 링크 장체가 댕글링인 경우에 특히 유용하지만, 링크가 댕글링이 아닌 경우에도, 이 필드는 이 특정 이미지 내에서 이벤트가 얼마나 정확히 참조되었는지를 나타내기 위해 사용될 수 있다.

● "LocationReference": 이미지는 또한 특정 사진의 장소를 나타내기 위해 Location 아이템(도 8A의 Core.Location)에의 LocationReference 링크들을 가질 수 있고, 다음의 필드들을 포함한다:

속성 이름	설명
Latitude	이 사진이 찍힌 장소의 위도. 이것은 흔히 디지털 카메라들로부터 얻을 수 있다.
Longitude	이 사진이 찍힌 장소의 경도. 이것은 흔히 디지털 카메라들로부터 얻을 수 있다.
Altitude	이 사진이 찍힌 장소의 고도. 이것은 흔히 디지털 카메라들로부터 얻을 수 있다.
Heading	이 사진이 찍힌 장소의 방향(카메라가 향한 방향). 이것은 흔히 디지털 카메라들로부터 얻을 수 있다.
LocationName	이 사진이 찍힌 장소의 이름. 이것은 자유로운 형태이 문자열일 수 있지만, 그것을 장소를 나타내는 의미 있는 부분 문자열(substring)로 분해하는 것을 권고한다. 예를 들면, "Seattle", "Washington", "USA"는 이 다중값 필드의 3개의 분리된 인스턴스들로 설정될 수 있다.

B. 사진(Photo) 스키마

이미지 스키마의 종속 스키마인 사진 스키마는 사실상 어떤 종류의 사진들인 이미지들에 적용된다. 임의의 사진 아이템들에 있어서, 사진 타입은 이미지 포맷에 상관없이 사진을 기술하는 속성들의 세트를 나타내고, 사진 타입은 도 36에 예시된 바와 같이 Image.Image 타입(즉, "Image" 스키마 내의 "Image" 타입)을 확장한다. 다음은 Photo 타입에 대한 몇몇 필드들이다:

속성 이름	설명
DateTaken	사진이 찍힌 날짜. 이 필드는 EXIF 정보로부터 얻을 수도 있고 또는 수동으로 설정될 수도 있다.
DateAcquired	디지털 카메라 또는 스캐너와 같은 디바이스로부터 사진이 획득된 날짜.
AcquisitionSessionID	획득 세션(acquisition session)의 고유 ID. 예를 들어 동일한 필름 롤(roll of film)로부터 다수의 사진들을 스캔했다면 이 ID는 동일한 롤로부터의 모든 사진들에 대해 동일할 것이다.
Orientation	이미지의 방위를 특정하는 필드. 그 의미는 EXIF 태그(넘버 1-8)와 동일하다. 더 많은 정보를 원하면 http://jpegclub.org/exif_orientation.html을 참조한다. NULL은 미지의 것(unknown)을 의미한다. 인물/풍경 방위는 이 태그 및 SizeX/SizeY로부터 결정적으로 계산될 수 있다.
Location	이 사진이 찍힌 장소
Event	이 이미지와 관련된 이벤트. 예를 들면, "생일", "크리스마스".
CameraMake	사진을 찍기 위해 사용된 카메라의 제조사의 이름. 예: "캐논".
CameraModel	사진을 찍기 위해 사용된 카메라의 모델명. 예: "S50".
ExposureTime	초로(보다 전형적으로는, 초의 분수(franctions of seconds)로) 나타낸 셔터 속도(노출 시간)
Aperture	노출하기 위해 사용된 조리개 설정
IsoSpeed	ISO 감도 레이팅에 상당하는 필름 속도
Flash	플래시가 켜졌나? 0=플래시 없음. 1-플래시 켜짐. NULL은 플래시가 켜졌는지 여부를 모른다는 의미.

RedEyeUsed	적목(RedEye) 모드가 사용되었나? 0=아니오. 1=예. NULL은 모른다는 의미. 이것은 진보된 쿼리들(advanced queries)에 대해서 유용할 수 있다. 예를 들면 적목이 사용되었다면, 이것은 인물 사진일 가능성이 높다.
ExposureMode	노출 모드. 다음의 5개 값들이 허용된다: 자동, 셔터 우선, 조리개 우선, ISO 우선, 수동
SubjectDistance	카메라에 의해 측정된 피사체까지의 거리. 이 거리는 미터로 나타낸다.

C. 분석 속성(Anaylsis Properties) 스키마

디지털 사진들의 경우는, 분석 애플리케이션에 의해 사진들 상에서 속성들의 세트가 계산될 수 있다. 그러나, 이들 속성들은 시간 및 프로세서 자원들의 관점에서 계산하고 재계산하는 데 비용이 많이 든다. 더욱이, 이들 필드들은 애플리케이션 특정적(application specific)이고, 다른 애플리케이션들은 이들 필드의 내부 포맷을 이해하지 못할 수 있다.

사진 아이템의 경우는, 이들 애플리케이션에 의해 사용하기에 앞서 분석 속성들의 표준 세트가 대신 계산되고 사진 아이템 타입에 대한 확장의 형태로 사진 아이템에 부가될 수 있다. 분석 속성 스키마(AP 스키마)는 그 자체가 도 7에 예시된 바와 같이 베이스 스키마의 Base.Extension 확장 타입의 확장인 AP 확장에 대한 AP 타입을 제공함으로써 바로 그것을 행한다. AP 타입 확장은 다음의 필드들을 포함한다:

속성 이름	**설명**
ColorHistogram	유사도 검출을 위해 이용되는 컬러 히스토그램
GrayHistogram	유사도 검출을 위해 이용되는 그레이 히스토그램
SimilarityIndex	유사도 검출을 위해 이용되는 이미지 텍스처 데이터

Ⅳ. 결론

위에서 예시 설명한 바와 같이, 본 발명은 데이터를 조직하고, 검색하고, 공유하기 위한 저장 플랫폼에 관한 것이다. 본 발명의 저장 플랫폼은 기존의 파일 시스템 및 데이터베이스 시스템을 넘어서 데이터 저장의 개념을 확장하고 넓히며, 관계(표 형태의) 데이터, XML, 및 아이템이라고 불리는 새로운 형태의 데이터와 같은 구조화된, 또는 비구조화된, 또는 반구조화된 데이터를 포함하는 데이터의 모든 타입들에 대한 스토어가 되도록 설계되어 있다. 그것의 공통의 저장 토대 및 조직적으로 배열된 데이터를 통하여, 본 발명의 저장 플랫폼은 소비자, 지식 노동자 및 기업에게 보다 효율적인 애플리케이션 개발을 가능케 한다. 그것은 그것의 데이터 모델 내에 고유한 능력들을 이용할 수 있게 할 뿐만 아니라, 기존의 파일 시스템 및 데이터베이스 액세스 방법들도 포용하고 확장하는 풍부하고 확장성이 있는 애플리케이션 프로그래밍 인터페이스를 제공한다. 본 발명의 넓은 발명적 개념을 벗어나지 않고 위에서 설명한 실시예들에 대해 변경이 행해질 수 있음은 물론이다. 따라서, 본 발명은 개시된 특정 실시예들에 한정되지 않고, 첨부된 청구의 범위에 의해 정의된 발명의 요지 및 범위 내에 있는 모든 변형들을 망라하도록 의도되어 있다.

위로부터 명백히 알 수 있는 바와 같이, 본 발명의 갖가지 시스템, 방법, 및 양태들의 전부 또는 일부는 프로그램 코드(즉, 명령어)의 형태로 구현될 수 있다. 이 프로그램 코드는 플로피 디스크, CD-ROM, CD-RW, DVD-ROM, DVD-RAM, 자기 테이프, 플래시 메모리, 하드 디스크 드라이브, 또는 임의의 다른 기계 판독가능한 저장 매체를 포함하지만 이들에 한정되지 않는 자기적, 전기적 및 광학적 저장 매체와 같은 컴퓨터 판독가능한 매체 상에 저장될 수 있고, 프로그램 코드는 컴퓨터 또는 서버와 같은 기계에 로딩되어 그 기계에 의해 실행되고, 그 기계는 본 발명을 실시하기 위한 장치가 된다. 본 발명은 또한 예컨대, 전기 배선 또는 케이블링을 통하여, 광섬유를 통하여, 인터넷 또는 인트라넷을 포함하는 네트워크를 통하여, 또는 임의의 다른 전송 형태와 같은, 어떤 전송 매체를 통하여 전송되는 프로그램 코드의 형태로 구현될 수도 있는데, 프로그램 코드는 컴퓨터와 같은 기계에 로딩되어 그 기계에 의해 실행되고, 그 기계는 본 발명을 실시하기 위한 장치가 된다. 범용 프로세서 상에서 실행될 때, 프로그램 코드는 그 프로세서와 조합하여 특정 논리 회로들과 유사하게 동작하는 독특한 장치를 제공한다.

(57) 청구의 범위

청구항 1.

컴퓨터 시스템용 하드웨어/소프트웨어 인터페이스 시스템을 위한 컴퓨터 판독가능한 명령어들을 갖는 컴퓨터 판독가능한 매체로서,

상기 하드웨어/소프트웨어 인터페이스 시스템은, 이미지와 관련되고 상기 하드웨어/소프트웨어 인터페이스 시스템에 의해 이해 가능한 속성들을 갖는 복수의 개별 정보 단위들("아이템들")을 조작 처리(manipulate)하는 컴퓨터 판독가능한 매체.

청구항 2.

제1항에 있어서,

상기 하드웨어/소프트웨어 인터페이스 시스템은 적어도 하나의 이미지 아이템 및 적어도 하나의 이미지 속성을 정의하는 이미지 스키마(image schema)를 포함하는 컴퓨터 판독가능한 매체.

청구항 3.

제2항에 있어서,

상기 이미지 스키마 내의 적어도 하나의 아이템은, 기본 아이템 타입을 구성하고, 상기 하드웨어/소프트웨어 인터페이스 시스템에서 조작 처리되는 다른 모든 이미지 아이템들이 도출되는 기본 아이템(foundational Item)인 컴퓨터 판독가능한 매체.

청구항 4.

제3항에 있어서,

상기 기본 이미지 아이템 타입은, 레이팅(rating); 사이즈 - 사이즈는 화소들(pixels)로 나타낸 이미지의 폭 또는 높이를 구성함 -; 이미지 비트 깊이(image bit depth); 색공간(color space); 색공간의 버전; 상기 이미지가 인쇄된 횟수; 상기 이미지 내의 관심 영역; 상기 이미지에 관한 히스토리; 디지털 네거티브 ID(digital negative identification); 동일 이미지의 다른 버전들에의 링크들의 컬렉션; 및 이미지에 대한 메타데이터 라이프사이클을 포함하는 속성 그룹 중에서 적어도 하나의 속성을 포함하는 컴퓨터 판독가능한 매체.

청구항 5.

제2항에 있어서,

상기 이미지 스키마 내의 적어도 하나의 속성은 상기 이미지의 영역에 대한 속성이고, 상기 영역 속성은 상기 영역의 좌측(left), 상측(top), 우측(right) 및 하측(bottom) 좌표들에 대한 필드들을 포함하는 컴퓨터 판독가능한 매체.

청구항 6.

제5항에 있어서,

상기 이미지 스키마 내의 적어도 하나의 속성은 상기 이미지의 관심 영역에 대한 속성이고, 상기 관심 영역 속성은 영역(region)에 대한 필드를 포함하는 컴퓨터 판독가능한 매체.

청구항 7.

제6항에 있어서,

상기 이미지의 상기 관심 영역 속성은 주체(principal)(예를 들면, 사람(person))에 대한 필드를 더 포함하는 컴퓨터 판독가능한 매체.

청구항 8.

제6항에 있어서,

상기 이미지의 상기 관심 영역 속성은 신뢰(confidence)에 대한 필드를 더 포함하는 컴퓨터 판독가능한 매체.

청구항 9.

제2항에 있어서,

상기 이미지 아이템은 주체 아이템(principal item)(예를 들면, 사람)과의 관계(예를 들면, 링크)를 갖는 컴퓨터 판독가능한 매체.

청구항 10.

제2항에 있어서,

상기 이미지 아이템은 이벤트 아이템과의 관계(예를 들면, 링크)를 갖는 컴퓨터 판독가능한 매체.

청구항 11.

제2항에 있어서,

상기 이미지 아이템은 장소(location) 아이템과의 관계(예를 들면, 링크)를 갖는 컴퓨터 판독가능한 매체.

청구항 12.

제3항에 있어서,

상기 하드웨어/소프트웨어 인터페이스 시스템은 적어도 하나의 사진(photo) 아이템 및 적어도 하나의 사진 속성을 정의하는 사진 스키마를 포함하고, 상기 사진 아이템 타입은 이미지 아이템 타입의 확장인 컴퓨터 판독가능한 매체.

청구항 13.

제12항에 있어서,

상기 사진 스키마 내의 적어도 하나의 아이템은, 기본 아이템 타입을 구성하고, 상기 하드웨어/소프트웨어 인터페이스 시스템에서 조작 처리되는 다른 모든 사진 아이템들이 도출되는 기본 아이템(foundational Item)인 컴퓨터 판독가능한 매체.

청구항 14.

제13항에 있어서,

상기 기본 사진 아이템 타입은, 상기 사진이 찍힌 날짜; 상기 사진이 획득된 날짜; 상기 사진에 대한 획득 세션의 고유 ID; 방위; 장소; 이벤트; 카메라의 제조사; 카메라의 모델; 노출 시간; 조리개(aperture); ISO 속도; 상기 사진에 대해 플래시가 사용되었는 지의 여부에 대한 표시; 상기 사진에 대해 적목 모드(red-eye mode)가 사용되었는 지의 여부에 대한 표시; 노출 모드; 및 상기 사진의 피사체의 거리를 포함하는 속성 그룹 중에서 적어도 하나의 속성을 포함하는 컴퓨터 판독가능한 매체.

청구항 15.

제3항에 있어서,

상기 하드웨어/소프트웨어 인터페이스 시스템은 사진 아이템의 적어도 하나의 분석 속성(AP) 및 적어도 하나의 AP 속성을 정의하는 분석 속성 스키마를 포함하고, 상기 사진 아이템 타입은 이미지 아이템 타입의 확장인 컴퓨터 판독가능한 매체.

청구항 16.

제15항에 있어서,

상기 AP는 컬러 히스토그램(color histogram); 계조 히스토그램(gray histogram); 및 유사도 인덱스(similarity index)를 포함하는 속성 그룹 중에서 적어도 하나의 속성을 포함하는 컴퓨터 판독가능한 매체.

청구항 17.

이미지와 관련되고 하드웨어/소프트웨어 인터페이스 시스템에 의해 이해 가능한 속성들을 갖는 복수의 개별 정보 단위들("아이템들")을 조작 처리하는 시스템으로서,

상기 시스템은 적어도 하나의 이미지 아이템 및 적어도 하나의 이미지 속성을 정의하는 이미지 스키마를 포함하고, 상기 이미지 스키마 내의 상기 이미지 아이템은 기본 아이템 타입을 구성하고, 상기 하드웨어/소프트웨어 인터페이스 시스템에서 조작 처리되는 다른 모든 이미지 아이템들이 도출되는 기본 아이템인 시스템.

청구항 18.

제17항에 있어서,

상기 기본 이미지 아이템 타입은, 레이팅; 사이즈 - 사이즈는 화소들로 나타낸 이미지의 폭 또는 높이를 구성함 -; 이미지 비트 깊이; 색공간; 색공간의 버전; 상기 이미지가 인쇄된 횟수; 상기 이미지 내의 관심 영역; 상기 이미지에 관한 히스토리; 디지털 네거티브 ID; 동일 이미지의 다른 버전들에의 링크들의 컬렉션; 및 이미지에 대한 메타데이터 라이프사이클을 포함하는 속성 그룹 중에서 적어도 하나의 속성을 포함하는 시스템.

청구항 19.

제17항에 있어서,

상기 이미지 스키마 내의 적어도 하나의 속성은 상기 이미지의 영역에 대한 속성이고, 상기 영역 속성은 상기 영역의 좌측, 상측, 우측 및 하측 좌표들에 대한 필드들을 포함하는 시스템.

청구항 20.

제17항에 있어서,

상기 이미지 스키마 내의 적어도 하나의 속성은 상기 이미지의 관심 영역에 대한 속성이고, 상기 관심 영역 속성은 영역(region)에 대한 필드를 포함하는 시스템.

청구항 21.

제17항에 있어서,

상기 이미지의 상기 관심 영역 속성은 주체(principal)(예를 들면, 사람(person))에 대한 필드를 더 포함하는 시스템.

청구항 22.

제17항에 있어서,

상기 이미지의 상기 관심 영역 속성은 신뢰(confidence)에 대한 필드를 더 포함하는 시스템.

청구항 23.

제17항에 있어서,

상기 이미지 아이템은 주체 아이템(principal item)(예를 들면, 사람)과의 관계(예를 들면, 링크)를 갖는 시스템.

청구항 24.

제17항에 있어서,

상기 이미지 아이템은 이벤트 아이템과의 관계(예를 들면, 링크)를 갖는 시스템.

청구항 25.

제17항에 있어서,

상기 이미지 아이템은 장소(location) 아이템과의 관계(예를 들면, 링크)를 갖는 시스템.

청구항 26.

제17항에 있어서,

상기 하드웨어/소프트웨어 인터페이스 시스템은 적어도 하나의 사진(photo) 아이템 및 적어도 하나의 사진 속성을 정의하는 사진 스키마를 포함하고, 상기 사진 아이템 타입은 이미지 아이템 타입의 확장인 시스템.

청구항 27.

제17항에 있어서,

상기 사진 스키마 내의 적어도 하나의 아이템은, 기본 아이템 타입을 구성하고, 상기 하드웨어/소프트웨어 인터페이스 시스템에서 조작 처리되는 다른 모든 사진 아이템들이 도출되는 기본 아이템인 시스템.

청구항 28.

제27항에 있어서,

상기 기본 사진 아이템 타입은, 상기 사진이 찍힌 날짜; 상기 사진이 획득된 날짜; 상기 사진에 대한 획득 세션의 고유 ID; 방위; 장소; 이벤트; 카메라의 제조사; 카메라의 모델; 노출 시간; 조리개(aperture); ISO 속도; 상기 사진에 대해 플래시가 사용되었는지 여부에 대한 표시; 상기 사진에 대해 적목 모드(red-eye mode)가 사용되었는지 여부에 대한 표시; 노출 모드; 및 상기 사진의 피사체의 거리를 포함하는 속성 그룹 중에서 적어도 하나의 속성을 포함하는 시스템.

청구항 29.

제27항에 있어서,

상기 하드웨어/소프트웨어 인터페이스 시스템은 사진 아이템의 적어도 하나의 분석 속성(AP) 및 적어도 하나의 AP 속성을 정의하는 분석 속성 스키마를 포함하고, 상기 사진 아이템 타입은 이미지 아이템 타입의 확장인 시스템.

청구항 30.

제29항에 있어서,

상기 AP는 컬러 히스토그램(color histogram); 계조 히스토그램(gray histogram); 및 유사도 인덱스(similarity index)를 포함하는 속성 그룹 중에서 적어도 하나의 속성을 포함하는 시스템.

청구항 31.

하드웨어/소프트웨어 인터페이스 시스템에 의해 이해가능한 속성들을 갖는 이미지에 관한 복수의 개별 정보 단위들("이미지 아이템들")을 조작 처리하는 하드웨어/소프트웨어 인터페이스 시스템을 위한 방법으로서,

적어도 하나의 이미지 아이템 및 적어도 하나의 이미지 속성을 정의하는 이미지 스키마를 설정하는 단계와;

상기 이미지 스키마 내의 아이템을, 기본 아이템 타입을 구성하고 상기 하드웨어/소프트웨어 인터페이스 시스템에서 조작 처리되는 다른 모든 이미지 아이템들이 도출되는 기본 아이템으로 설정하는 단계

를 포함하는 방법.

청구항 32.

제31항에 있어서,

상기 기본 이미지 아이템 타입은, 레이팅; 사이즈 - 사이즈는 화소들로 나타낸 이미지의 폭 또는 높이를 구성함 -; 이미지 비트 깊이; 색공간; 색공간의 버전; 상기 이미지가 인쇄된 횟수; 상기 이미지 내의 관심 영역; 상기 이미지에 관한 히스토리; 디지털 네거티브 ID; 동일 이미지의 다른 버전들에의 링크들의 컬렉션; 및 이미지에 대한 메타데이터 라이프사이클을 포함하는 속성 그룹 중에서 적어도 하나의 속성을 포함하는 방법.

청구항 33.

제31항에 있어서,

상기 이미지 스키마 내의 적어도 하나의 속성은 상기 이미지의 영역에 대한 속성이고, 상기 영역 속성은 상기 영역의 좌측, 상측, 우측 및 하측 좌표들에 대한 필드들을 포함하는 방법.

청구항 34.

제33항에 있어서,

상기 이미지 스키마 내의 적어도 하나의 속성은 상기 이미지의 관심 영역에 대한 속성이고, 상기 관심 영역 속성은 영역(region)에 대한 필드, 주체(principal)(예를 들면, 사람(person))에 대한 필드, 또는 신뢰(confidence)에 대한 필드를 포함하는 방법.

청구항 35.

제31항에 있어서,

상기 이미지 아이템은 주체 아이템(principal item)(예를 들면, 사람), 이벤트 아이템, 또는 장소(location) 아이템과의 관계(예를 들면, 링크)를 갖는 방법.

청구항 36.

제31항에 있어서,

사진 디지털 이미지 아이템(Photo)이 찍힌 지리적 위치를 나타내기 위해, 상기 Photo와 상기 지리적 위치에 대응하는 장소 아이템(Location) 사이의 관계(Location Relationship)를 설정함으로써, 상기 Photo가 상기 Location에 기초하여 쿼리될 수 있도록 하는 단계를 더 포함하는 방법.

청구항 37.

제36항에 있어서,

사진 디지털 이미지 아이템(Photo)이 찍힌 지리적 위치를 나타내기 위해, 상기 지리적 위치에 대응하도록 상기 Location Relationship에 대한 장소 속성(LocProp)을 설정하는 단계를 더 포함하는 방법.

청구항 38.

제31항에 있어서,

사진 디지털 이미지 아이템(Photo) 내의 적어도 하나의 사람을 나타내기 위해, 상기 Photo와 상기 사람에 관한 아이템(Contact) 사이의 관계를 설정함으로써, 상기 Photo가 상기 Contact에 기초하여 쿼리될 수 있도록 하는 단계를 더 포함하는 방법.

청구항 39.

제31항에 있어서,

사진 디지털 이미지 아이템(Photo)에서 표현된 적어도 하나의 이벤트를 나타내기 위해, 상기 Photo와 상기 이벤트에 관한 아이템(Event) 사이의 관계를 설정함으로써, 상기 Photo가 상기 Event에 기초하여 쿼리될 수 있도록 하는 단계를 더 포함하는 방법.

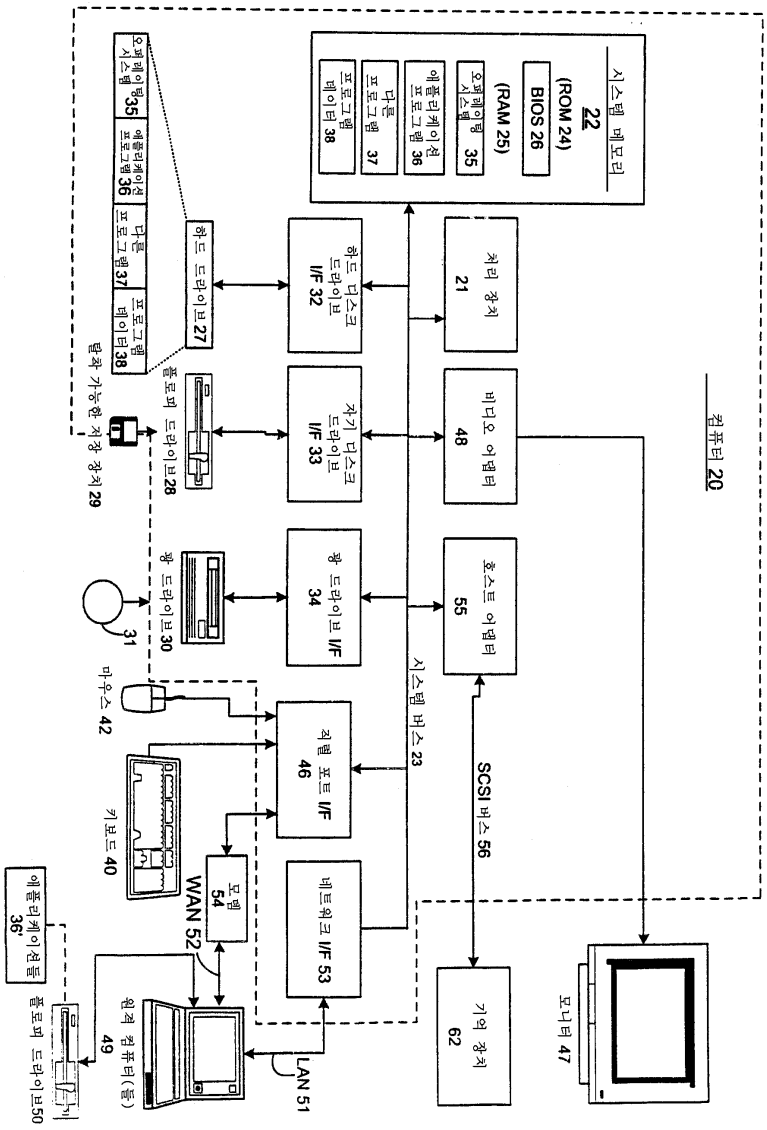
청구항 40.

제31항에 있어서,

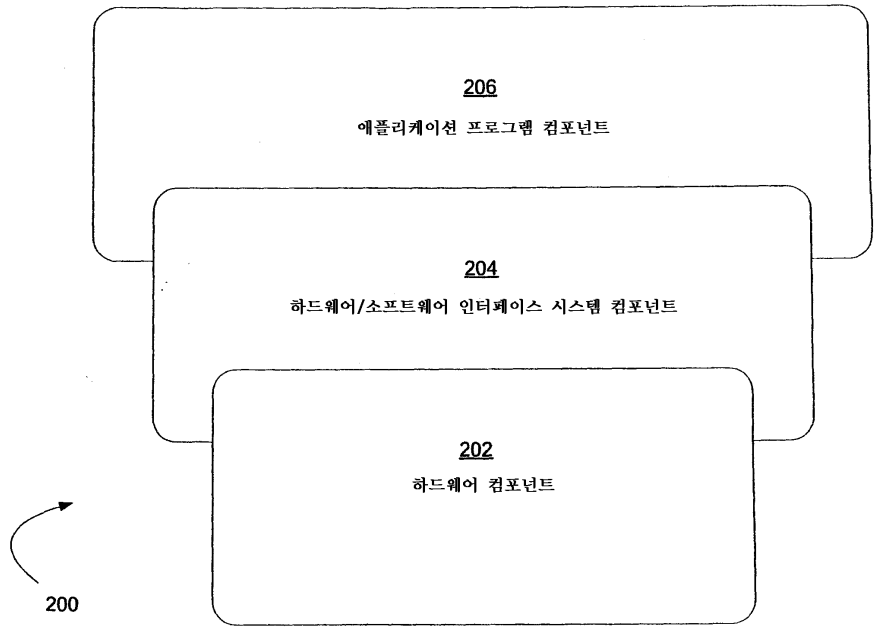
제1 디지털 이미지 아이템(Image)으로부터, (a) 상기 제1 이미지가 도출되는 부모 이미지 또는 (b) 상기 제1 이미지로부터 도출된 자식 이미지 중 어느 하나인 제2 이미지로의 관계를 설정하는 단계를 더 포함하는 방법.

도면

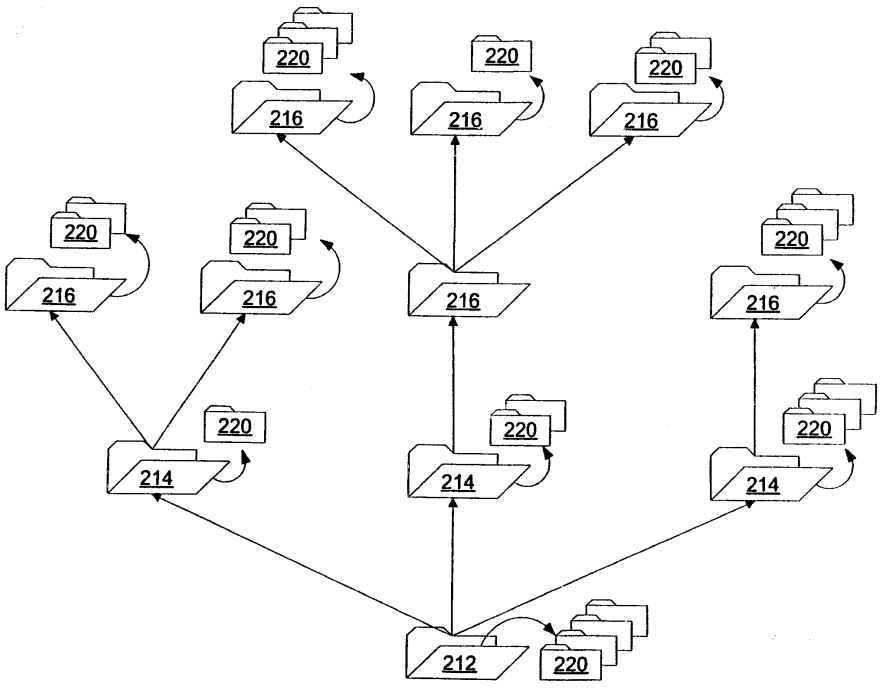
도면1


컴퓨터 20
시스템 메모리 22
(ROM 24)
BIOS 26
(RAM 25)
오퍼레이팅 시스템 35
어플리케이션 프로그램 36
다른 프로그램 37
프로그램 데이터 38
처리 장치 21
하드 디스크 드라이브 I/F 32
자기 디스크 드라이브 I/F 33
광 드라이브 I/F 34
비디오 어댑터 48
호스트 어댑터 55
시스템 버스 23
SCSI 버스 56
직렬 포트 I/F 46
네트워크 I/F 53
모뎀 54
WAN 52
LAN 51
하드 드라이브 27
플로피 드라이브 28
탈착 가능한 저장 장치 29
광 드라이브 30
31
마우스 42
키보드 40
모니터 47
기억 장치 62
원격 컴퓨터(들) 49
어플리케이션들 36'
플로피 드라이브 50

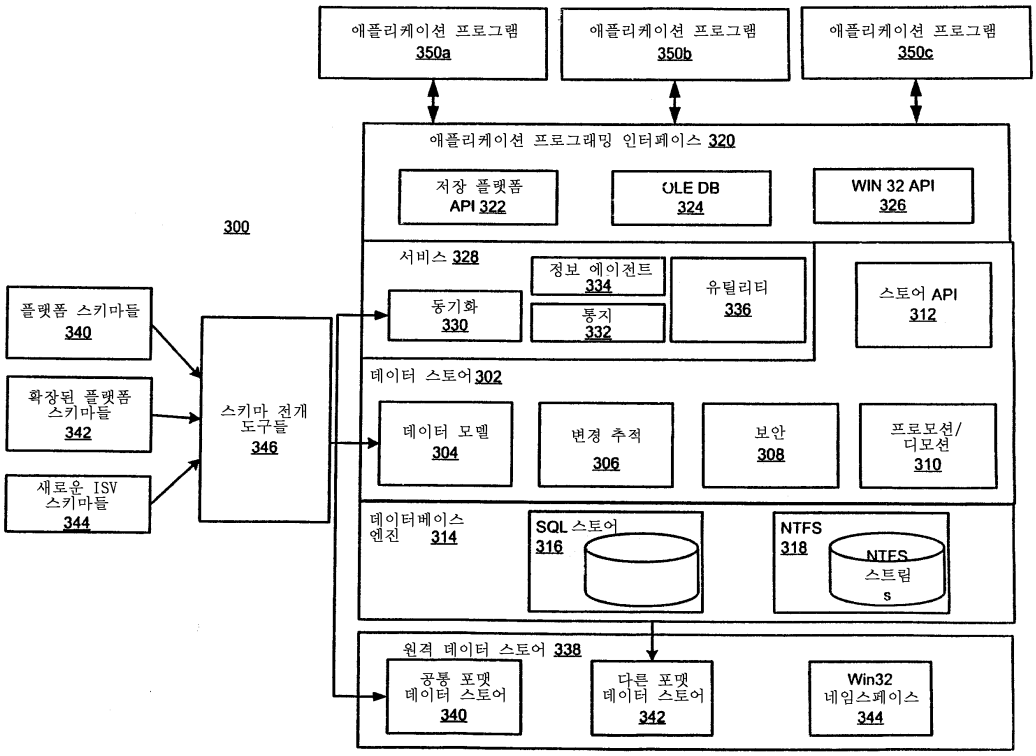
도면2


206
에플리케이션 프로그램 컴포넌트
204
하드웨어/소프트웨어 인터페이스 시스템 컴포넌트
202
하드웨어 컴포넌트
200

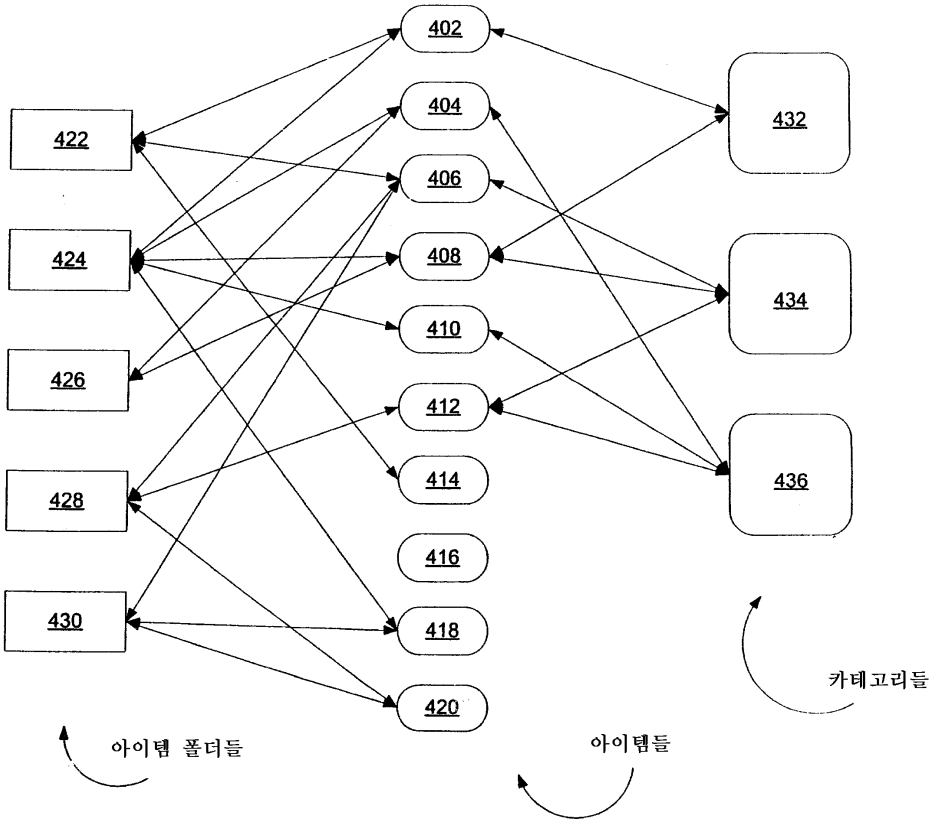
도면2A


220
216
214
212

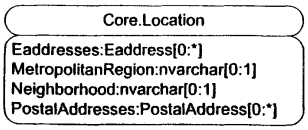
도면3


애플리케이션 프로그램
350a
애플리케이션 프로그램
350b
애플리케이션 프로그램
350c
애플리케이션 프로그래밍 인터페이스 320
저장 플랫폼 API 322
OLE DB 324
WIN 32 API 326
300
서비스 328
정보 에이전트 334
동기화 330
통지 332
유틸리티 336
스토어 API 312
플랫폼 스키마들 340
확장된 플랫폼 스키마들 342
새로운 ISV 스키마들 344
스키마 전개 도구들 346
데이터 스토어 302
데이터 모델 304
변경 추적 306
보안 308
프로모션/디모션 310
데이터베이스 엔진 314
SQL 스토어 316
NTFS 318
NTFS 스트림s
원격 데이터 스토어 338
공통 포맷 데이터 스토어 340
다른 포맷 데이터 스토어 342
Win32 네임스페이스 344

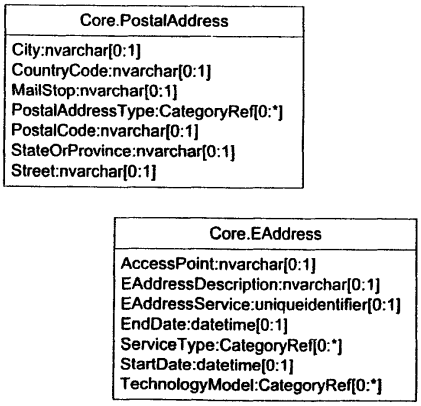
도면4


402
404
406
408
410
412
414
416
418
420
422
424
426
428
430
432
434
436
아이템 폴더들
아이템들
카테고리들

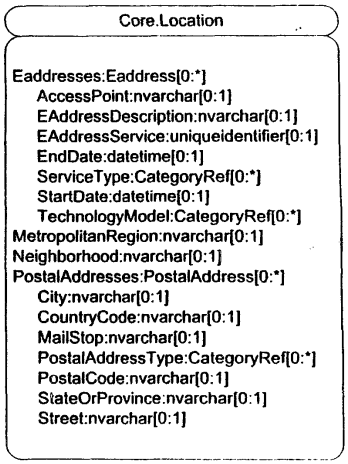
도면5A



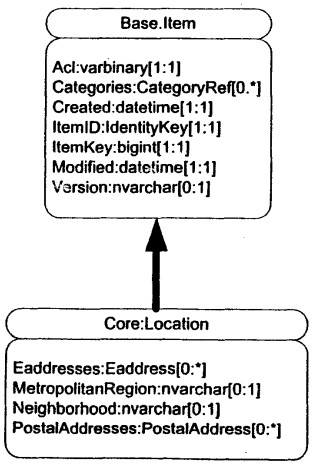
도면5B



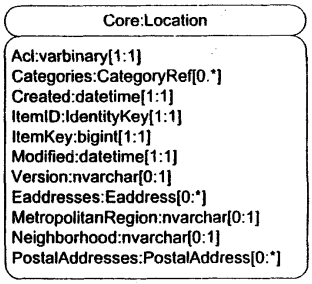
도면5C



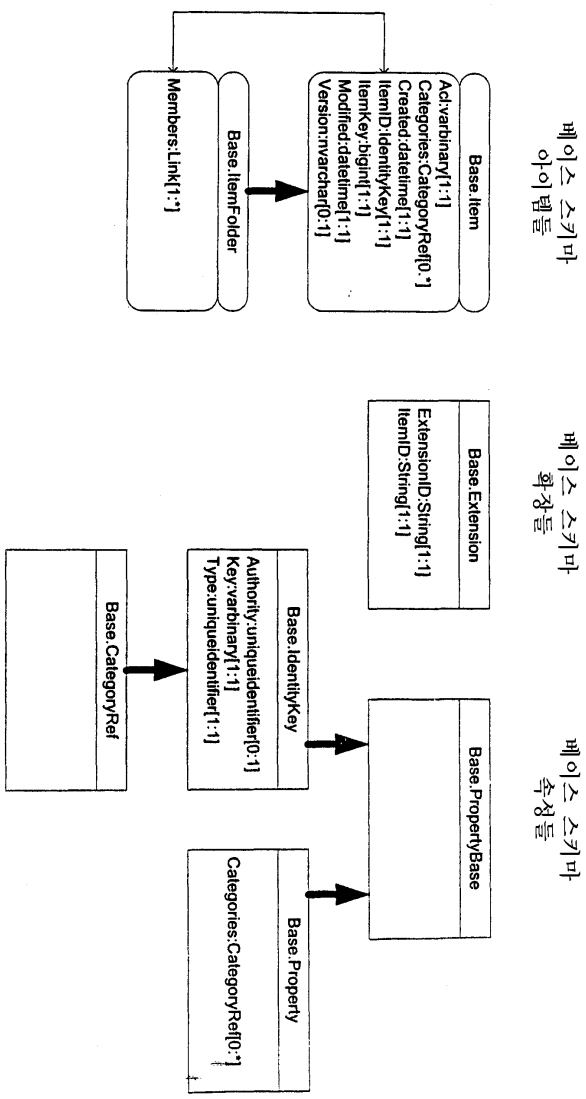
도면6A



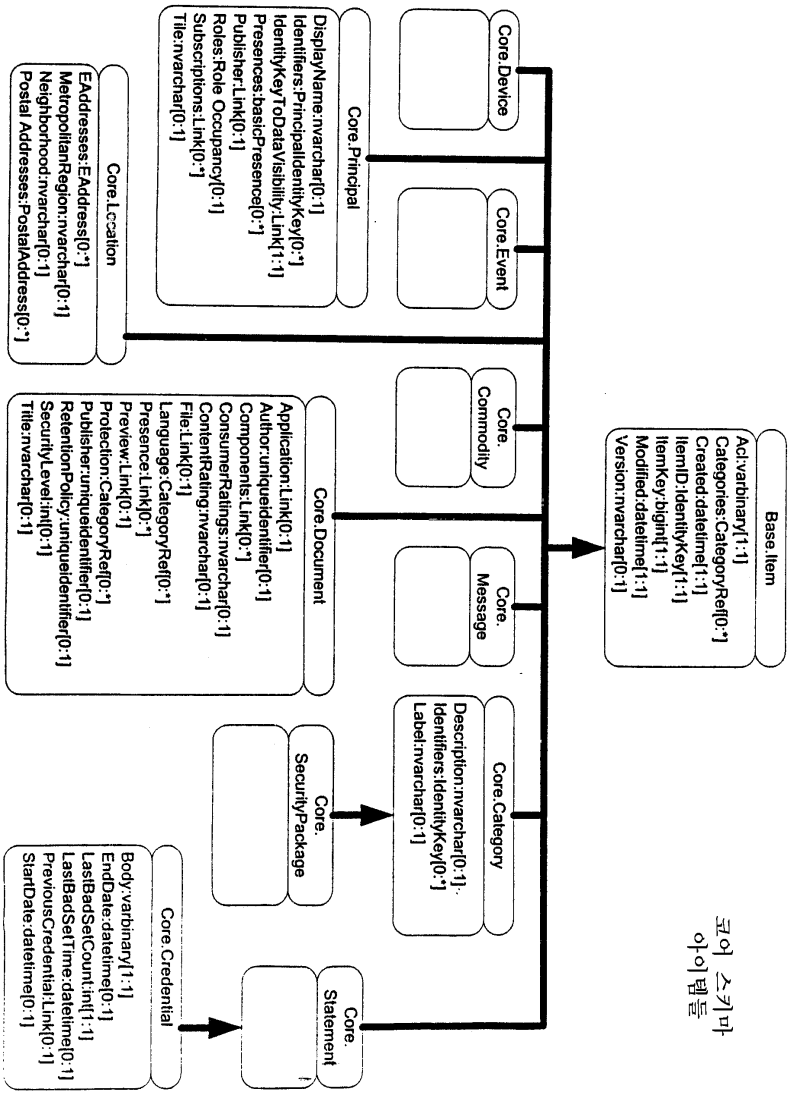
도면6B


Core:Location
Acl:varbinary[1:1]
Categories:CategoryRef[0.*]
Created:datetime[1:1]
ItemID:IdentityKey[1:1]
ItemKey:bigint[1:1]
Modified:datetime[1:1]
Version:nvarchar[0:1]
Eaddresses:Eaddress[0:*]
MetropolitanRegion:nvarchar[0:1]
Neighborhood:nvarchar[0:1]
PostalAddresses:PostalAddress[0:*]

도면7

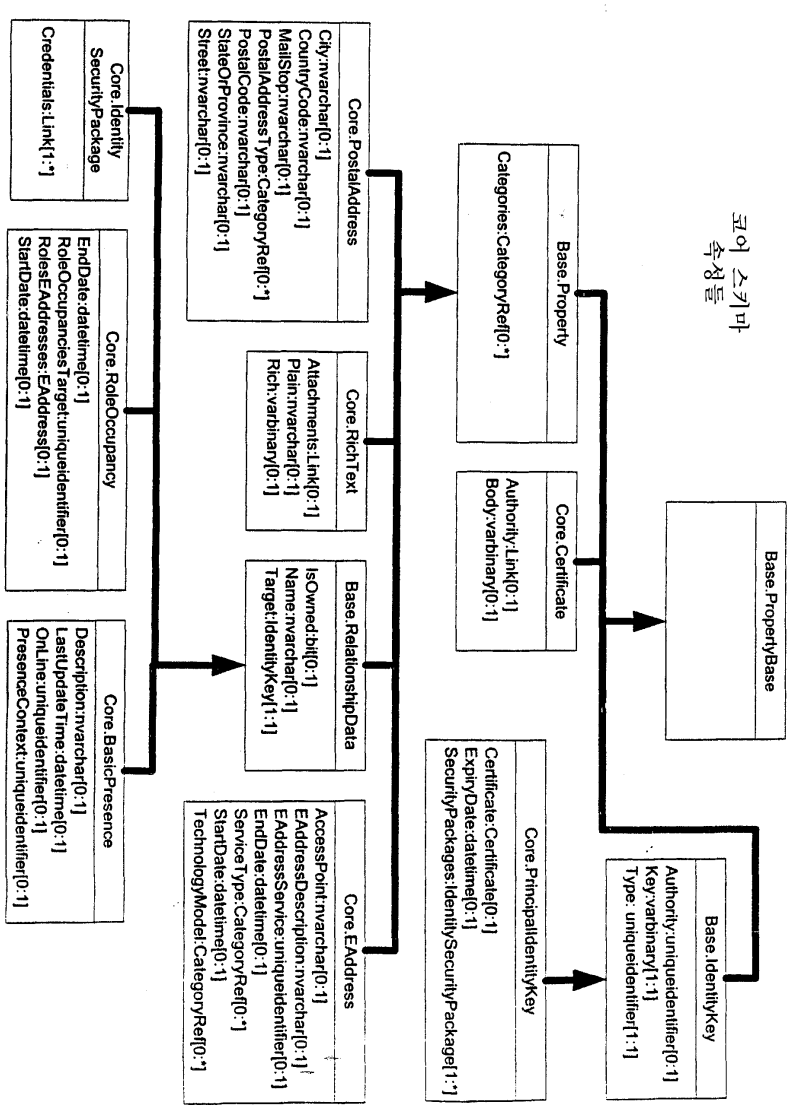

베이스 스키마 아이템들
Base.Item
Acl:varbinary[1:1]
Categories:CategoryRef[0.*]
Created:datetime[1:1]
ItemID:IdentityKey[1:1]
ItemKey:bigint[1:1]
Modified:datetime[1:1]
Version:nvarchar[0:1]
Base.ItemFolder
Members:Link[1:*]
베이스 스키마 확장들
Base.Extension
ExtensionID:String[1:1]
ItemID:String[1:1]
베이스 스키마 속성들
Base.PropertyBase
Base.IdentityKey
Authority:uniqueIdentifier[0:1]
Key:varbinary[1:1]
Type:uniqueIdentifier[1:1]
Base.Property
Categories:CategoryRef[0:*]
Base.CategoryRef

도면8A

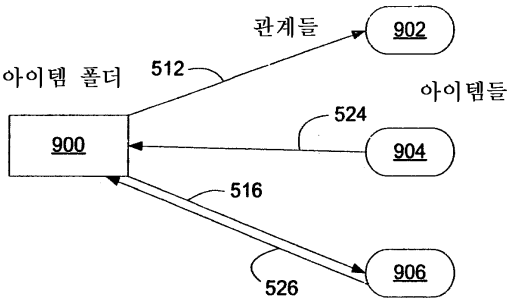


코어 스키마
아이템들

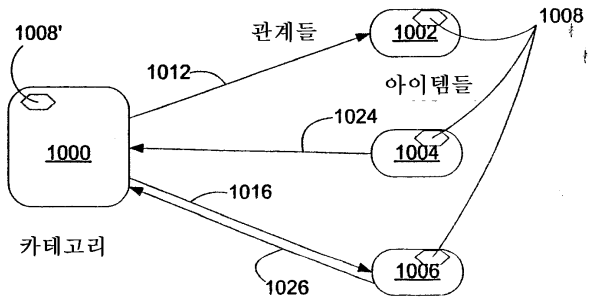
도면8B


코어 스키마 속성들
Base.PropertyBase
Base.Property
Categories:CategoryRef[0:*]
Core.Certificate
Authority:Link[0:1]
Body:varbinary[0:1]
Core.PrincipalIdentityKey
Certificate:Certificate[0:1]
ExpiryDate:datetime[0:1]
SecurityPackages:IdentitySecurityPackage[1:*]
Base.IdentityKey
Authority:uniqueidentifier[0:1]
Key:varbinary[1:1]
Type: uniqueidentifier[1:1]
Core.PostalAddress
City:nvarchar[0:1]
CountryCode:nvarchar[0:1]
MailStop:nvarchar[0:1]
PostalAddressType:CategoryRef[0:*]
PostalCode:nvarchar[0:1]
StateOrProvince:nvarchar[0:1]
Street:nvarchar[0:1]
Core.RichText
Attachments:Link[0:1]
Plain:nvarchar[0:1]
Rich:varbinary[0:1]
Base.RelationshipData
IsOwned:bit[0:1]
Name:nvarchar[0:1]
Target:IdentityKey[1:1]
Core.EAddress
AccessPoint:nvarchar[0:1]
EAddressDescription:nvarchar[0:1]
EAddressService:uniqueidentifier[0:1]
EndDate:datetime[0:1]
ServiceType:CategoryRef[0:*]
StartDate:datetime[0:1]
TechnologyModel:CategoryRef[0:*]
Core.Identity
SecurityPackage
Credentials:Link[1:*]
Core.RoleOccupancy
EndDate:datetime[0:1]
RoleOccupanciesTarget:uniqueidentifier[0:1]
RolesEAddresses:EAddress[0:1]
StartDate:datetime[0:1]
Core.BasicPresence
Description:nvarchar[0:1]
LastUpdateTime:datetime[0:1]
OnLine:uniqueidentifier[0:1]
PresenceContext:uniqueidentifier[0:1]

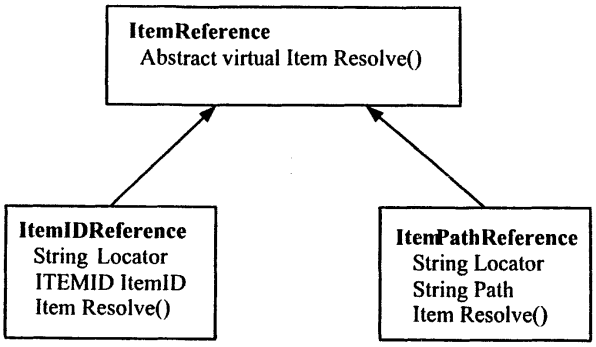
도면9


관계들
902
아이템 폴더
512
아이템들
524
900
904
516
906
526

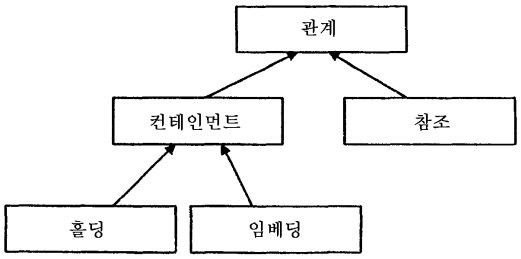
도면10



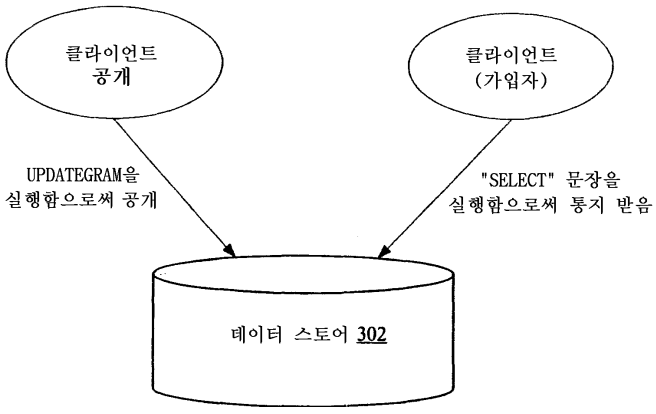
도면11



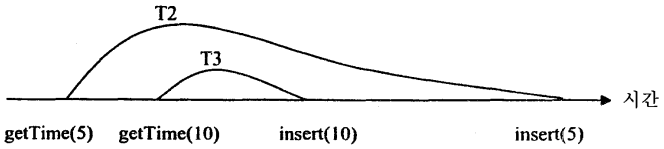
도면12



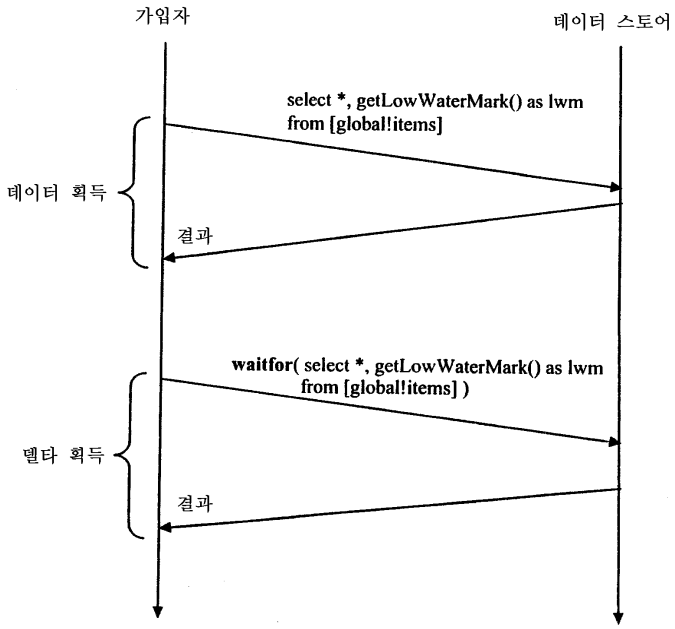
도면13



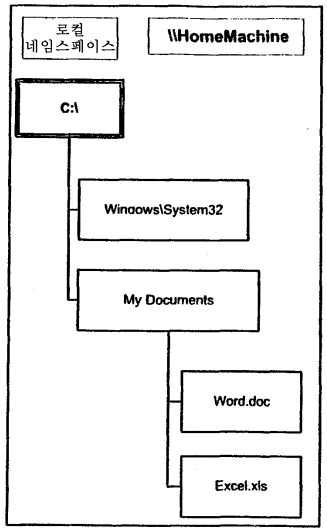
도면14



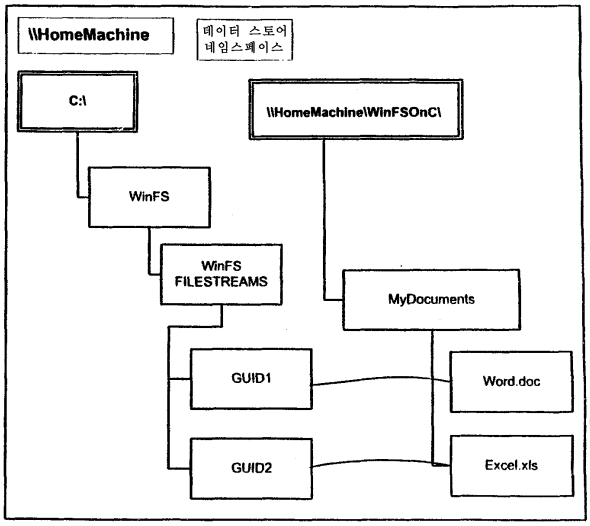
도면15



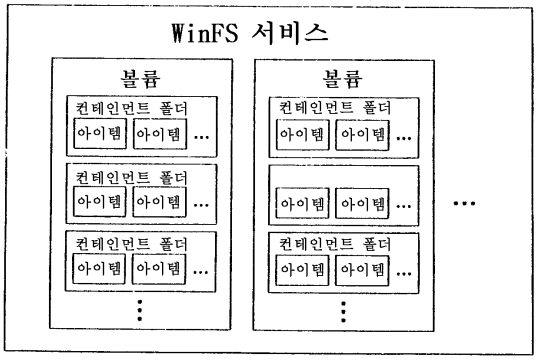
도면16



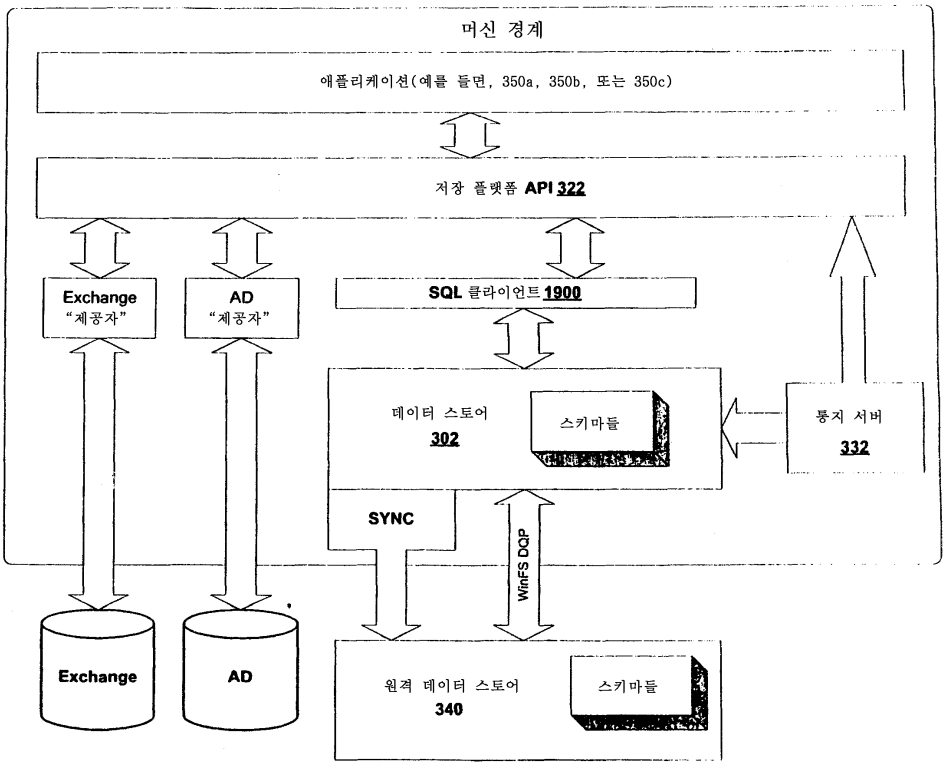
도면17


\\HomeMachine
데이터 스토어
네임스페이스
C:\
\\HomeMachine\WinFSOnC\
WinFS
WinFS
FILESTREAMS
MyDocuments
GUID1
Word.doc
GUID2
Excel.xls

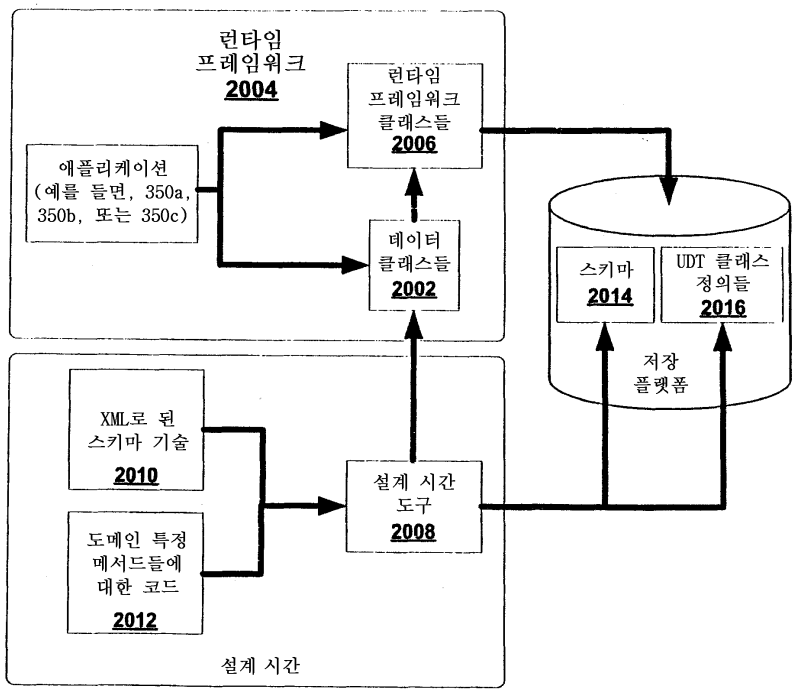
도면18


WinFS 서비스
볼륨
볼륨
컨테인먼트 폴더
아이템
아이템
...

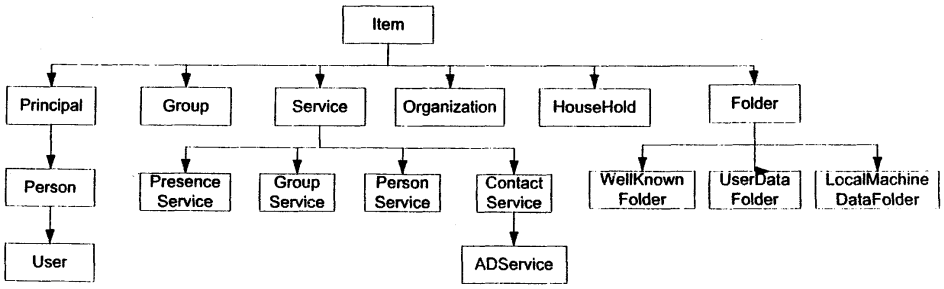
도면19


머신 경계
애플리케이션(예를 들면, 350a, 350b, 또는 350c)
저장 플랫폼 API 322
Exchange
"제공자"
AD
"제공자"
SQL 클라이언트 1900
데이터 스토어
302
스키마들
통지 서버
332
SYNC
WinFS DQP
Exchange
AD
원격 데이터 스토어
340
스키마들

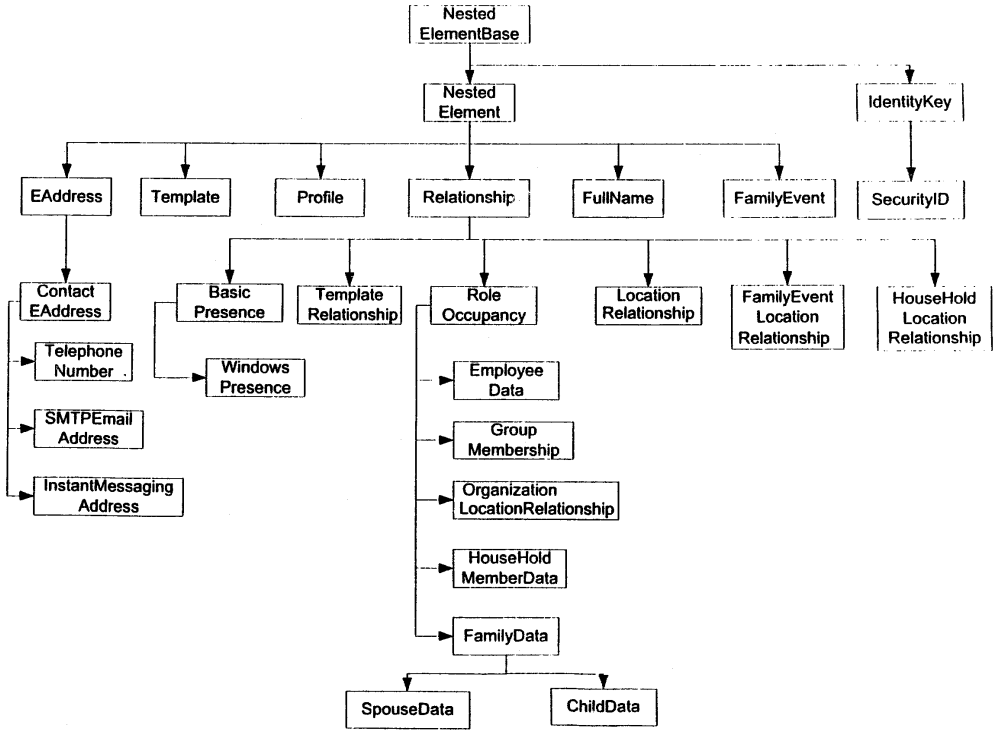
도면20


런타임
프레임워크
2004
애플리케이션
(예를 들면, 350a,
350b, 또는 350c)
런타임
프레임워크
클래스들
2006
데이터
클래스들
2002
스키마
2014
UDT 클래스
정의들
2016
저장
플랫폼
XML로 된
스키마 기술
2010
도메인 특정
메서드들에
대한 코드
2012
설계 시간
도구
2008
설계 시간

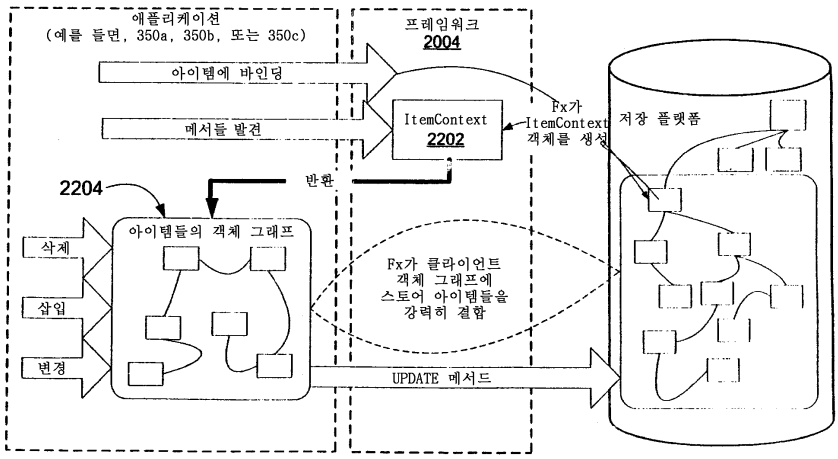
도면21A



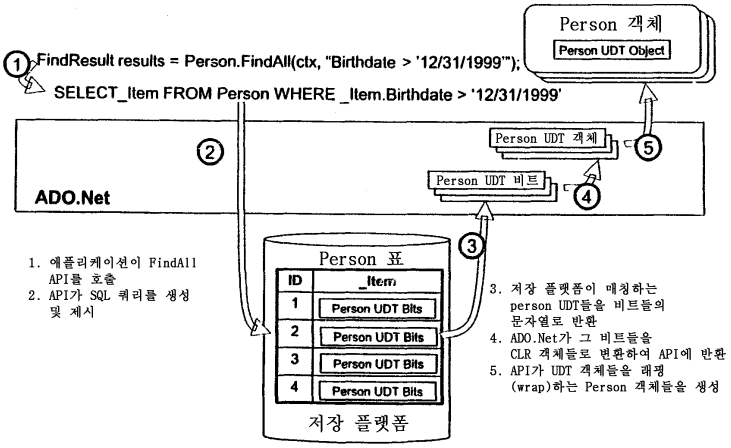
도면21B



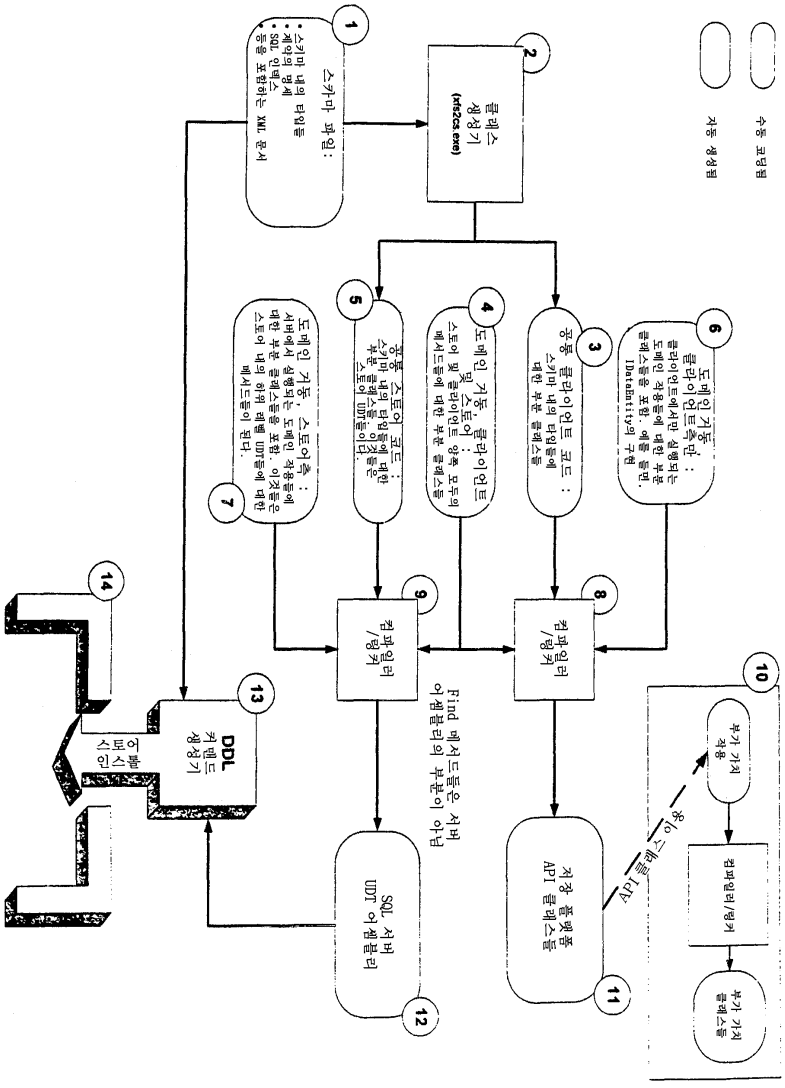
도면22



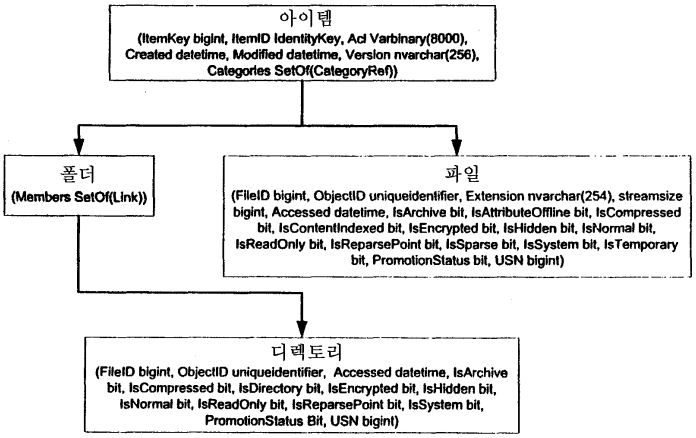
도면23


Person 객체
Person UDT Object
1 FindResult results = Person.FindAll(ctx, "Birthdate > '12/31/1999'");
SELECT_Item FROM Person WHERE _Item.Birthdate > '12/31/1999'
ADO.Net
Person UDT 객체
Person UDT 비트
Person 표
ID
_Item
Person UDT Bits
저장 플랫폼
1. 애플리케이션이 FindAll API를 호출
2. API가 SQL 쿼리를 생성 및 제시
3. 저장 플랫폼이 매칭하는 person UDT들을 비트들의 문자열로 반환
4. ADO.Net가 그 비트들을 CLR 객체들로 변환하여 API에 반환
5. API가 UDT 객체들을 래핑 (wrap)하는 Person 객체들을 생성

도면24


스키마 파일 :
클래스 생성기
(ddz2cs.exe)
자동 생성됨
수동 프로필
공통 클라이언트 코드 :
도메인 기능, 스토어 : 클라이언트 양쪽 모두의
공통 스토어 코드 :
도메인 기능 :
컴파일러/링커
컴파일러/링커
DDL 커맨드 생성기
스토어 인스톨
Find 메서드들은 서버 어셈블리의 부분이 아님
SQL 서버 UDT 어셈블리
저장 플랫폼 API 클래스들
API 클래스 이용
부가 기능 적용
컴파일러/링커
부가 기능 클래스들

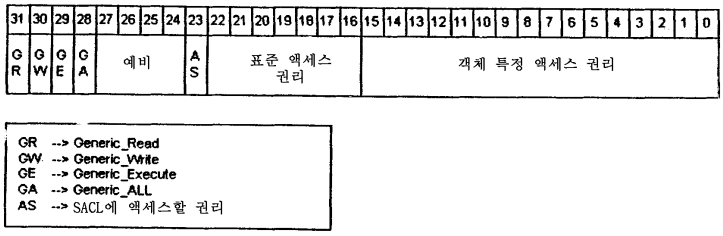
도면25



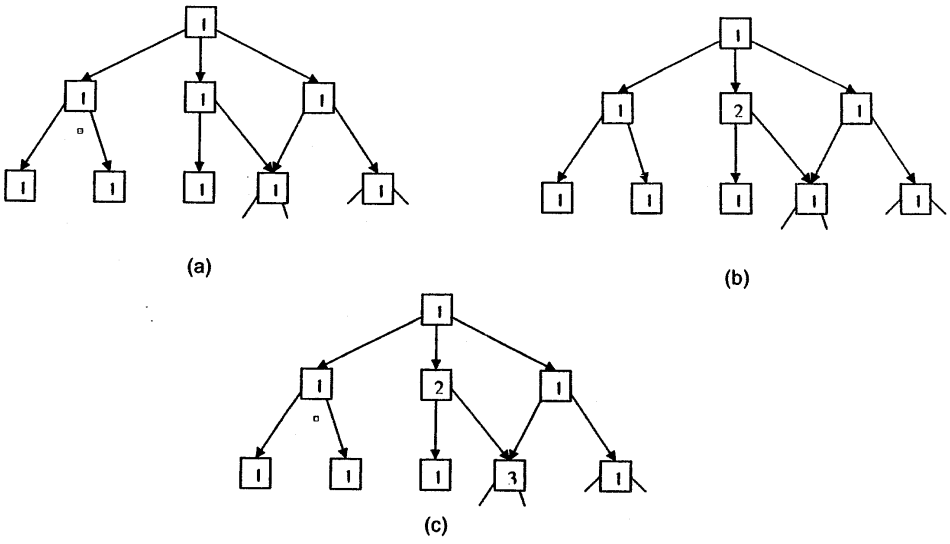
주: 그레이 박스들은 베이스 스키마에서 정의된 타입들이지만, 여기서는 완벽을 위해서 도시됨.

파일 스키마에서 정의된 아이템들

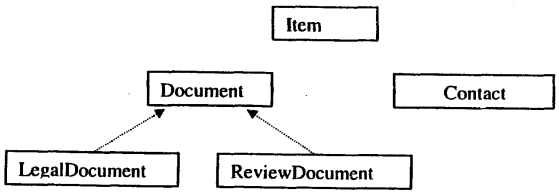
도면26



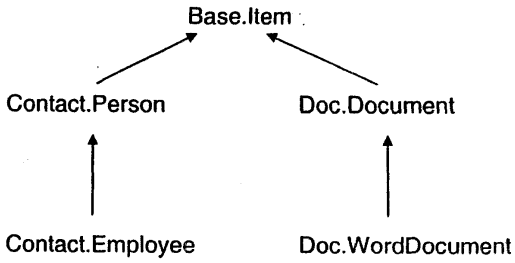
도면27



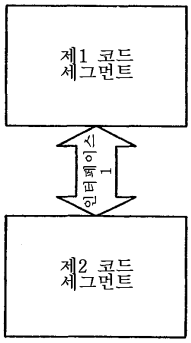
도면28



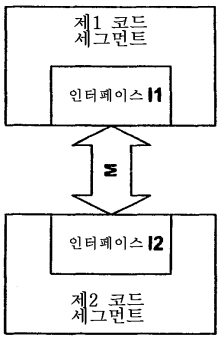
도면29



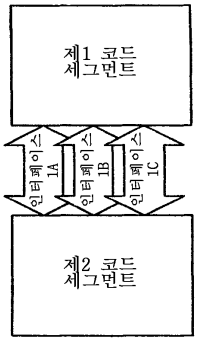
도면30A



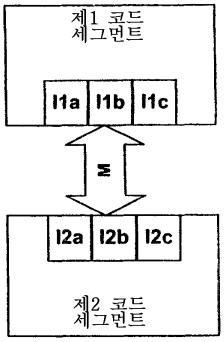
도면30B



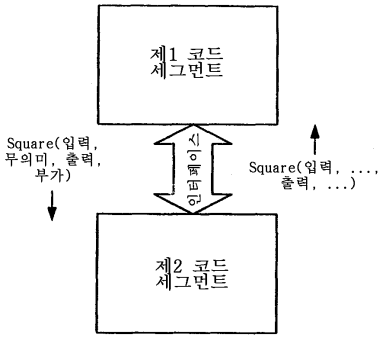
도면31A



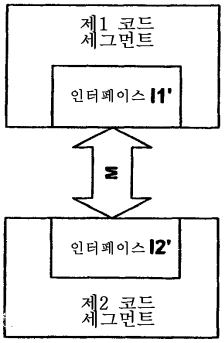
도면31B



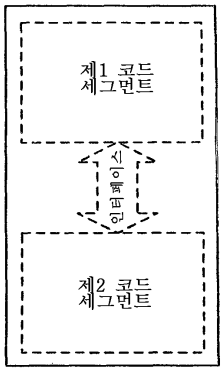
도면32A



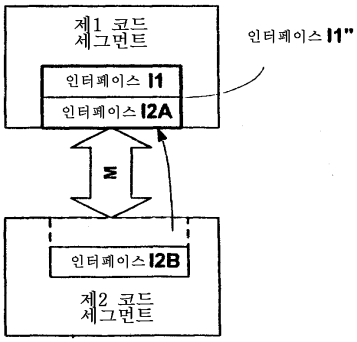
도면32B



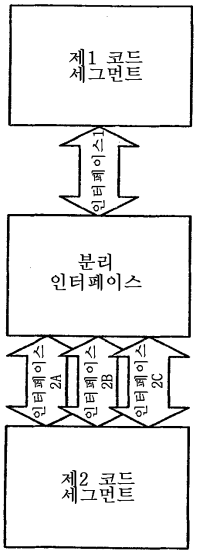
도면33A



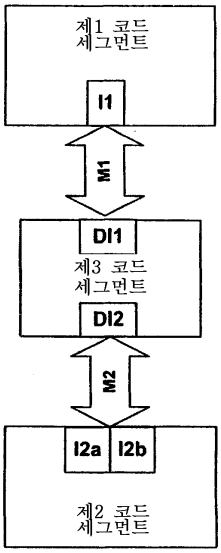
도면33B



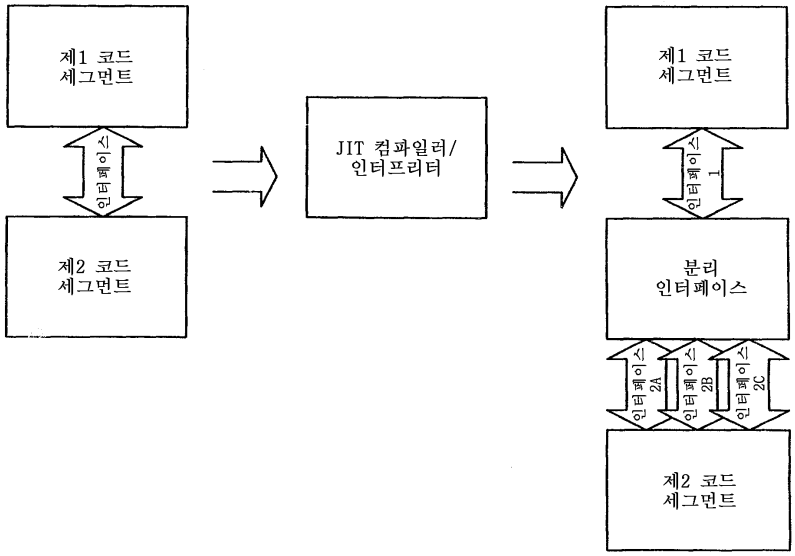
도면34A



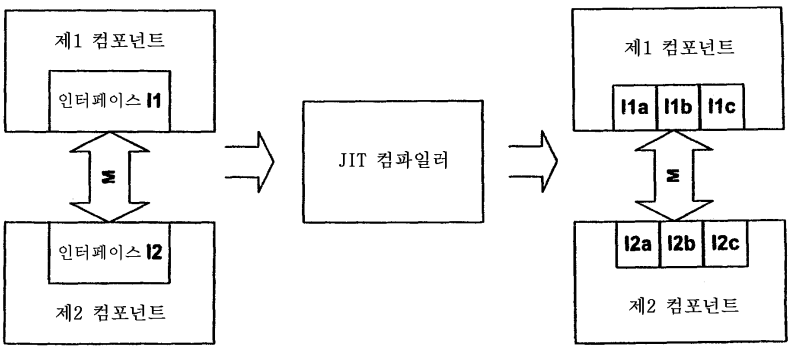
도면34B



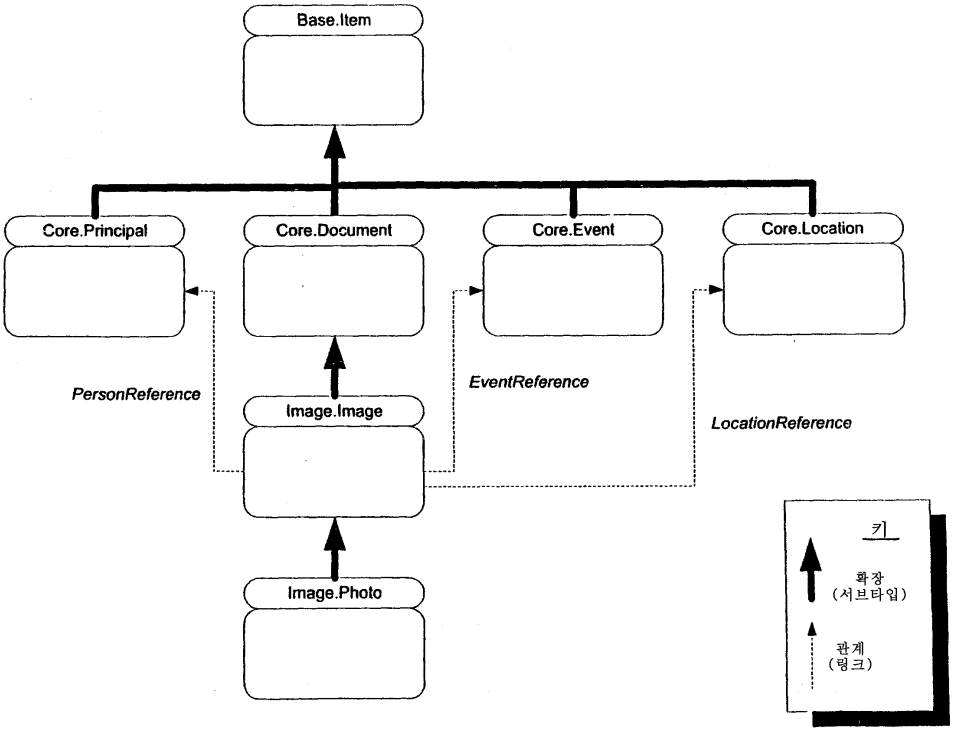
도면35A



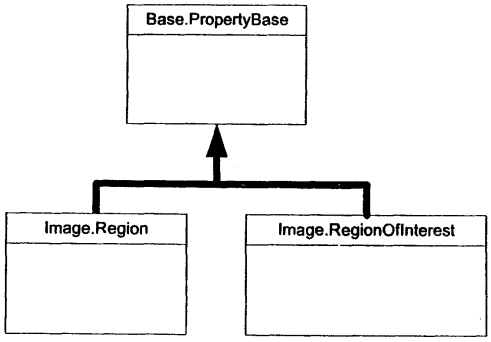
도면35B



도면36A


Base.Item
Core.Principal
Core.Document
Core.Event
Core.Location
PersonReference
EventReference
LocationReference
Image.Image
Image.Photo
키
확장
(서브타입)
관계
(링크)

도면36B


Base.PropertyBase
Image.Region
Image.RegionOfInterest