목차

목차 1

저작권 고지 3

문서 최종 갱신일 3

피드백 3

2장. 데이터 모델링과 설계 4

2.1 데이터의 분류 4

2.2 복잡적응계(Complex Adaptive system; CAS)와 데이터 모델링 6

2.3 개념, 논리, 물리 모델 9

2.4 종, 관계, 속성 11

종 11

관계 17

속성 21

물리적인 구현에서의 '사원' 23

2.5 이력관리 25

속성에 대한 이력관리 25

관계에 대한 이력관리 27

종에 대한 이력관리 31

2.6 릴레이션쉽 유형 31

범위한정 32

상호작용 32

존재종속 34

2.7 계층, 유형, 카테고리 35

계층, 유형, 카테고리의 개념 35

계층구조 36

BOM(Bill-Of-Material) 40

다양한 계층구조 40

범주적 폭력 44

2.8 병렬 관계 45

사내 동호회 46

2.9 47

2.10 47

2.11 47

2.12 47

2.13 코드설계 48

저작권 고지

문서 최종 갱신일

이 문서는 2014년 01월 07일에 최종 갱신되었습니다. 최신 문서는 http://databaser.net 에 있습니다.

피드백

문서에 대한 의견이나 오류는 admin@databaser.net 로 메일을 주시거나 http://databaser.net 에 글 남겨주시면 됩니다.

2장. 데이터 모델링과 설계

데이터 모델링은 데이터베이스 구축에서 매우 중요한 과정이다. 데이터 모델에 따라 DB설계와 어플리케이션(SQL포함)의 복잡성이 결정된다. 그러므로 단순하지만 요구사항을 만족시킬 수 있도록 데이터 모델이 만들어져야 한다.

복잡하다는 것은 비용 상승을 의미한다. 잘못된 데이터 모델은 더 많은 CPU, RAM, Disk를 사용(하드웨어 관점)할 것이고, 자료구조와 알고리즘이 복잡(소프트웨어 관점)해질 것이다. 잘못된 데이터 모델은 소프트웨어를 개발하는 사람이나 하드웨어를 관리하는 시스템 엔지니어, 데이터베이스를 관리하는 DBA 모두 더 많이 생각하고, 더 많이 움직이게 할 것이며, 더 많은 시간을 필요로 하게 만들 것이다. 또한 생명주기(life-cycle)에서 가장 비용이 많은 드는 유지보수 비용에도 악영향을 끼칠 것이다. 복잡성은 유연성을 저해하는 가장 큰 요소다. 유연성의 저하로 인한 기회비용의 상실은 기업의 성장에 발목을 잡을 것이다. 그러므로 우리는 좋은 데이터 모델을 만드는데 많은 노력을 기울여야 한다.

1장에서 데이터 모델링에 필요한 이론을 살펴보았다. 2장에서는 실제 데이터 모델을 만들어 볼 것이다. 개념 모델이라고 따로 만들지는 않을 것이다. 대부분은 바로 논리 모델로 도출될 것이며, 릴레이션이 실제 어떻게 만들어지는지 검증해 볼 것이다. 데이터 모델링 연습을 하다 보면 데이터 모델링도 기법과 패턴이 있다는 것을 알게 될 것이다. 데이터 모델링도 익숙함이다. 좋은 모델이건 나쁜 모델이건 상관없다. 최대한 많이 데이터 모델을 만들어보고, 선배들에게 피드백을 받는 것이 가장 좋은 공부 방법이다.

미리 말해두겠지만, 조직이나 개인마다 용어나 단어는 정의하기 나름이다. 때에 따라서는 특수한 상황이 발생되어 시간에 따라서도 용어나 단어는 다른 뜻을 가질 수 있다. 설명을 위해서는 표준화가 필요하다. 그러므로 이 책에서는 국립국어원의 표준국어대사전의 사전적 의미를 기본으로 데이터 모델을 만들어 볼 것이다. 자, 시작해 보자.

2.1 데이터의 분류

세상에는 수많은 개체들이 존재하고, 개체들의 상호작용은 매우 복잡하여 컴퓨터 세계에 표현하는 것은 매우 어려운 일이다. 단순한 컴퓨터 세계에 현실의 복잡함을 담으려면 현실을 단순하게 관찰할 수 있어야 한다.

독자들은 그런 능력이 없다고 한탄하지 말라. 인간에게는 '분류'라는 훌륭한 도구가 있으니 말이다. 분류는 인간이 정보를 체계화하고, 이를 이용(추론)하여 결론을 도출하는 인간의 가장 기본적인 행위다. 분류는 각각의 개념, 사물, 사건 등의 개체들이 어떤 종(kind) 개념에 속하는지 파악하고, 어떤 유개념에 해당하는지 결정하는 행위다. 예를 들어, 개체 'a'가 '알파벳'라는 종에 해당된다

면 'a'는 '알파벳'의 규정성 범위에 있게 된다. '알파벳'은 '문자'로 분류될 수 있으므로 의사소통을 위한 기호체계인 것을 알 수 있다. 마찬가지로 개체 'ㄱ'이 문자로 분류되었다면 개체 'ㄱ'도 의사소통을 위한 기호체계인 것을 알 수 있으므로 개체 'a'와 개체 'ㄱ'의 차이만 알면 된다. 다음의 <그림 2.1.1>을 보자.

<그림 2.1.1>

필자는 <그림 2.1.1>에 보이는 사물을 정확히는 모르겠으나 '자동차'라는 사실만은 확실히 알고 있다. 그래서<그림 2.1.1>의 사물은 운송수단이며, 사람이 동작 시켜야 하며, 굴러가며, 대부분 철로 만들어졌으며, 화석연료를 사용한다는 것은 분명히 알 수 있다. 마찬가지로 처음 보는 물건이지만 '필기구'의 일종이라면 글을 쓰는데 사용되는 도구로써 사용된다는 것은 분명히 알 수 있다.

이처럼 데이터도 분류될 수 있다. 여러 기준에 의해 분류될 수 있겠지만, 이 책에서는 다음과 같이 분류하였다.

- 마스터 데이터
- 트랜잭션 데이터
- 참조 데이터

마스터 데이터는 행위의 주체를 설명한다. 즉, 행위를 인식하는 기준(누구, 무엇)에 대한 데이터이다. 고객, 상품, 정책 등이 이에 속한다. 마스터 데이터는 여러 시스템에 이음동의어나, 서로 다른 코드체계로 중복되어 나타나 통합의 대상이 된다. 또한 여러 카테고리 또는 분류체계가 존재한다. 여러분의 데이터베이스에서 고객이나 계정을 식별하는 컬럼들(id, account_id, cust_no와 같은)이 몇 개나 있는지 살펴보라. 엄청나게 많을 것이다. 이것은 명시적이든 묵시적이든 마스터 데이터가 여러 곳에서 쓰이고 있으며, 마스터 데이터가 잘못 되었을 경우 파급 효과가 큼을 말해주는 것이다. 전사(Enterprise)에는 당연히 여러 관점들이 있을 것이다. 마스터 데이터는 종으로 횡으로 각각의 여러 관점에 대한 요구사항을 만족시켜야 한다. 마스터 데이터는 유연하게 설계를 할 필요가 있다. 전사적인 관점에서의 설계가 불가능하다면 전사적인 관점을 취하도록 최대한 노력해야 한다.

트랜잭션 데이터는 행위의 결과나 진행을 설명한다. 즉, 트랜잭션 데이터는 마스터 데이터의 행위

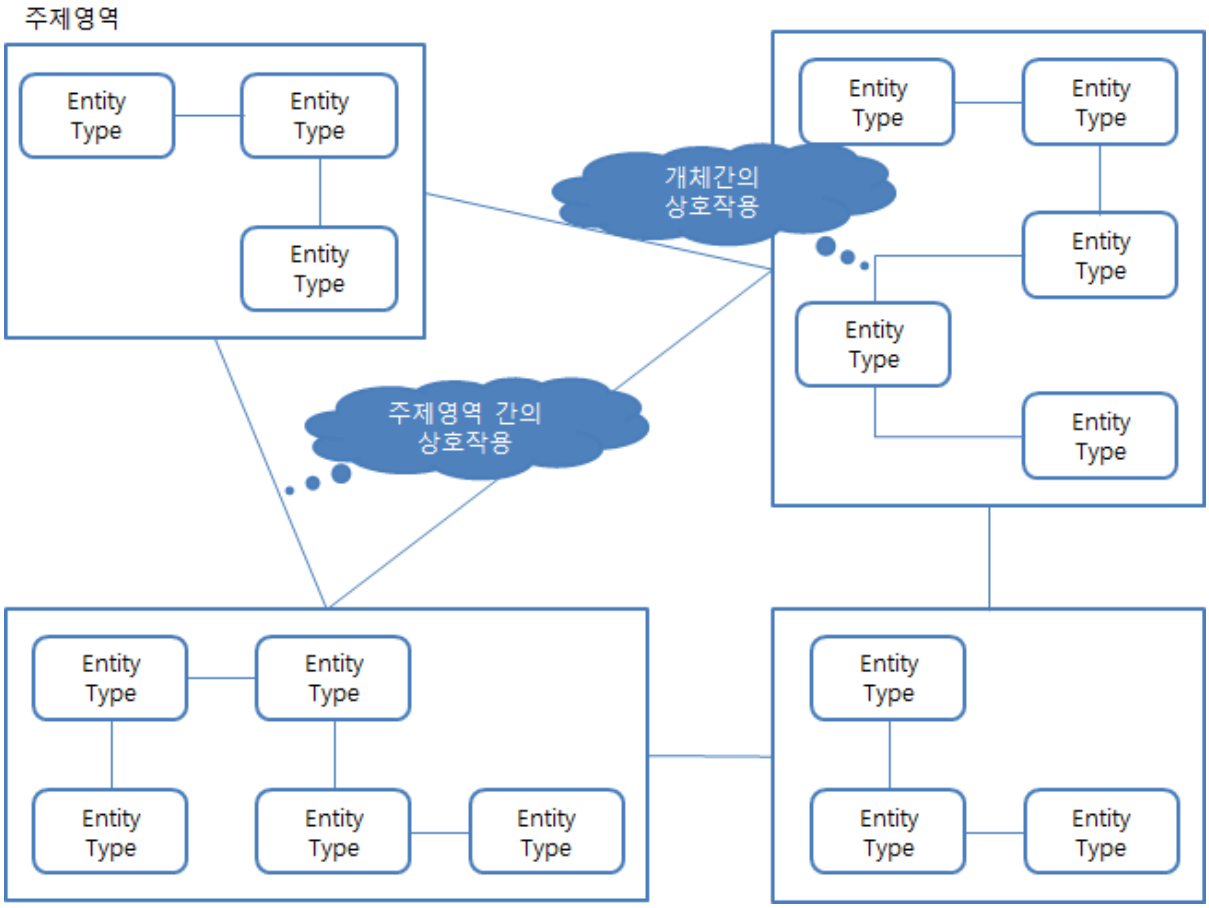
를 서술하는 데이터다. 주문, 상담, 발령 등이 이에 속한다. 트랜잭션 데이터는 마스터 데이터의 존재에 종속적이다. 왜냐하면 마스터 데이터가 어떤 행위를 해야만 트랜잭션 데이터가 만들어질 수 있기 때문이다. 예를 들어, 단순하게 '고객이 상품을 주문'하는 것을 생각해보자. 고객이 없으면 주문을 할 수 없다. 상품이 없으면 주문의 대상이 없으므로 주문을 할 수 없다. 그러므로 고객과 상품이 모두 존재해야만 '주문'이 만들어질 수 있다. 즉, 개체간의 상호작용의 상태나 결과에 대한 데이터가 트랜잭션 데이터다.

참조 데이터는 마스터 데이터도 또한 트랜잭션 데이터도 아닌 이미 존재하는 참조되는 데이터다. 주로 이미 존재하는 코드성 데이터를 지칭한다. 우편번호, 국가코드 또는 각종 표준에 관련된 데이터가 참조 데이터다.

아마도 처음에는 이러한 분류가 어려울 것이다. 왜냐하면 아직 이론을 적재하여 관찰할 수 있을 만큼의 연습이 안된 것이기 때문이다. 하지만, 걱정할 필요는 없다. 그냥 순수한 관심을 가지고 데이터 모델링 연습을 하다 보면 익숙해진다. 여러분은 단어 하나 하나에 집중하는 태도만 있으면 된다.

2.2 복잡적응계(Complex Adaptive system; CAS)와 데이터 모델링

존 홀런드는 그의 저서 '숨겨진 질서'에서 대도시에 식품(의식주 중에 식)의 물량은 공급이 끊긴다면 길어야 1~2주 버틸만큼 밖에 안 되는데, 어떻게 사람들이 굶지 않고 대도시가 황폐해지지 않는지를 여러 상호작용의 결과로 시간이 지남에 따라서 변화에 적응하는 것으로 설명한다. 이 컨셉을 데이터 모델로 옮겨 생각해 보자.

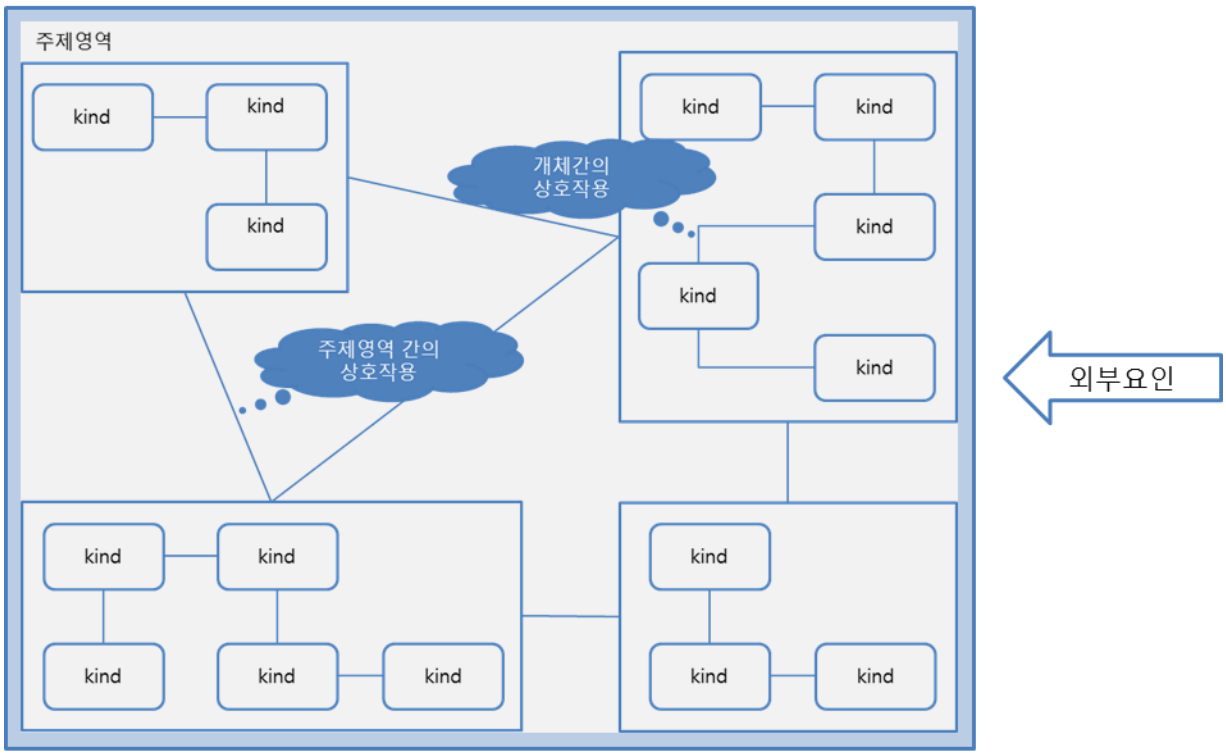


<그림 2.2.1>

프로세스는 자주 변한다고 한다. 그래서 프로세스 중심의 개발방법은 유지보수 비용이 많이 든다고 많은 책에서 설명한다. 그렇다고 데이터는 변하지 않느냐? 아니다. 변한다. 프로세스건 데이터건 어쨌든 변화하는 요인은 내부가 아닌 외부요인에 의한 것이다. 새로운 개념의 등장이나 시도 또는 법의 변화 등이 외부요인이 될 것이다. 2008년 2분기에는 '제한적 본인 인증제도'라는 것이 생겼다. 그래서 프로세스가 변했다. 각각의 상황에 따라서는 데이터 모델도 변화 당했을 것이다.

중요한 사실은 고객이 항상 변한다는 것이다. 그러므로 기업의 목적인 '고객 창출/유지'는 프로세스와 데이터의 변화를 필연적으로 가져온다. 정보 시스템은 이런 변화에 적응하기 위해서 지속적인 유지보수를 해야 한다. 변화에 적응하기 위해 A기업은 1억을 썼지만 B기업은 0.5억을 썼다면 B기업의 IT인프라가 훨씬 더 적응을 잘하게 끔 설계되었거나 또는 사람들의 기술력이 좋은 것이다. 종합적으로 다음의 그림처럼 표현할 수 있다.

일관성과 지속성이 의존하는 것들
- 상호작용
- 집단화
- 적응 or 학습



변화에 적응할 수 있는가? (변화 속의 일관성 유지)

<그림 2.2.2>

개체가 많아질수록 개체 간의 상호작용이 많아질수록 복잡해진다. 복잡한 것을 단순하게 만드는 방법은 복잡한 것을 단순해 질 때까지 쪼개보는 것이다. 이것을 '분석'이라고 한다. 어떤 복잡한 정보 시스템을 구축 할 때는 청사진을 놓고, 복잡하지 않을 정도로 단순하게 일을 쪼개어 전체 시스템을 완성해 나간다. 이를 '분할 정복'이라고 한다. 데이터 모델은 분석과 분할 정복을 이용하여 완성해 나가면 된다.

종은 복잡적응계에서 '행위자(agent)'로 일치시킬 수 있다. 복잡적응계는 행위자와 행위자 간의 상호작용에 의해 만들어진다. 그러므로 복잡적응계를 알려면 우선 행위자가 뭔지 알아야 한다. 그런 후 행위자 간의 '상호작용'을 알 수 있다면 나머지는 정리만 하면 된다. 데이터 모델의 세계에서 상호작용은 '관계'로 보면 된다. 복잡적응계와는 달리 데이터 모델의 세계에서의 상호작용은 비교적 단순하므로 데이터 모델링이라는 용어만으로 겁낼 필요는 없다. 정리하자만 데이터 모델링은 다음과 같은 순서로 하면 된다.

1. 개체를 정의한다. (필요하다면 속성도 정리한다)

2. 개체간의 관계를 정의한다.
3. 보기 좋게 정리한다.

물론 각각의 정규화라든지 속성의 정의 등 세부적인 과정은 좀 더 있다. 하지만 개체, 관계의 정의를 빼면 나머지는 책을 보면서 익힐 수 있는 단순한 스킬들의 집합으로 볼 수 있다. 유전 알고리즘이나 뉴런의 메커니즘, 대도시가 적응하는 매커니즘 등을 연구하는 것에 비하면 데이터 모델링은 훨씬 덜 복잡하다.

데이터 모델에서 우선시 해야 할 데이터는 마스터 데이터다. 행위의 주체가 되는 데이터가 복잡하면 행위에 대한 설명도 복잡해진다. 그러므로 데이터 모델링에서 마스터 데이터는 최대한 추상화시키는 것이 좋다. 하지만, 관계자들이 이해할 수 없는 수준이 될 수도 있다. 이런 경우는 조직의 수준을 끌어올려 유연성 있는 데이터 모델을 완성할 수 있어야 한다.

2.3 개념, 논리, 물리 모델

개념, 논리, 물리 모델이 명확히 나누는 단순한 기준을 필자는 찾지 못했다. 그래서 이 책에서는 개념, 논리, 물리 모델의 기준에 대한 기준을 용어의 사용으로 나누고자 한다. 각 모델에서 사용되는 용어는 다음과 같다.

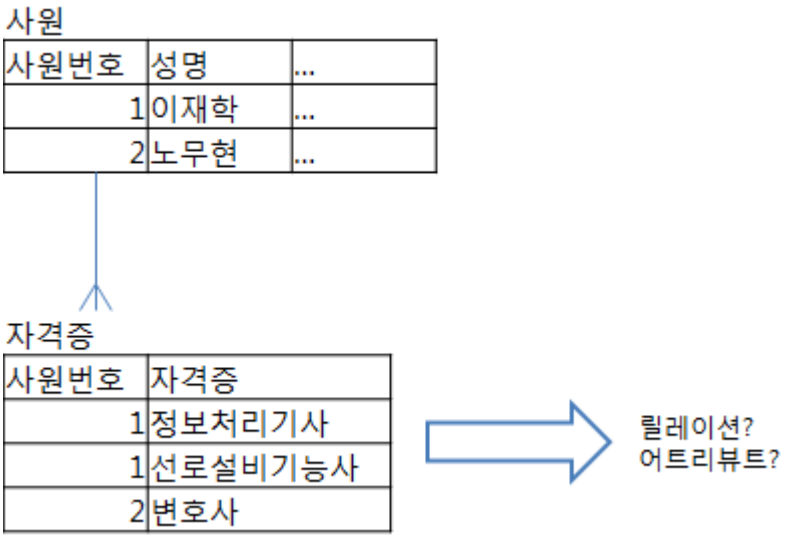
개념	논리	물리
종	릴레이션	테이블
관계	릴레이션쉽	외부키
속성	어트리뷰트	컬럼
개체	인스턴스	로우(행)

<그림 2.3.1>

여기서 각 용어들이 표에서 수평적으로 같은 위치에 있다고 해서 절대로 같거나 비슷한 것이 아니라는 것에 주의할 필요가 있다. 즉, 다음과 같이 이해해서는 절대 안 된다.

- 종 = 릴레이션 = 테이블
- 종≒ 릴레이션 ≒ 테이블

예를 들어, 다음의 <그림 2.3.2>를 보자.



<그림 2.3.2>

"자격증" 릴레이션은 다중값 어트리뷰트를 1차 정규화된 릴레이션으로 변환한 것이다. 즉, 본질은 어트리뷰트다. 이와 같이 릴레이션을 종이나 테이블로 같이 취급하면 안 된다. 개념 모델과 논리 모델의 큰 차이는 이론적인 배경의 차이다. 개념 모델은 실재론을 배경으로 '종'과 '종차'를 파악하지만, 논리 모델은 '집합'과 '명제'가 이론적 배경이 된다. '종'은 개체를 결정하지만, '집합'은 종이 결정한다. 이런 패러다임의 차이가 개념 모델과 논리 모델 사이에 갭을 만든다.

2.3절 이후로는 용어로써 개념, 논리, 물리 모델을 나눌 것이다. 다음의 영문 번역을 2.3절의 제목처럼 줄여서 사용할 것이다.

- 개념 모델 = Conceptual Data Model
- 논리 모델 = Logical Data Model
- 물리 모델 = Physical Data Model

여기서 설명되는 개념 모델은 현실세계를 이해하기 위한 모델로 형이상학적 실재론을 토대로 한 종, 관계, 속성에 대한 모델이고, 논리 모델은 1장에서 설명한 데이터를 표형태로 관리하려고 고안된 릴레이션형 모델이다. 물리 모델은 논리 모델을 컴퓨터 세계에 구현하기 위한 모델로 인덱스, 분산/분할, 테이블의 물리적 배치 등에 대한 모델인데, 특정 DBMS에 구현될 것임을 감안(여기에서는 MS-SQL Server 2012 버전을 사용할 것이다)한 것이다. 개념 모델은 DBMS에 종속되지 않는 모델이다. 물론 릴레이션형 DBMS만을 사용한다는 가정이라면 논리 모델도 특정 DBMS에 종속되지는 않는다. 이 책에서는 개념 모델에서 논리 모델인 릴레이션형 모델로의 자연스러운 전환을 기본으로 할 것이다.

개념 모델로 시작해서 논리, 물리 모델로의 자연스러운 진화를 위해서는 형이상학적 실재론을 적재하여 현실 세계를 관찰하고 이를 문서화 할 줄 알아야 한다. 여러 줄의 글보다는 하나의 그림

이 설명력이 뛰어나다고 했듯이 데이터 모델도 일종의 그림으로 표현된다. 주의해야 할 점은 현재 존재하는 데이터 모델링 툴이 개념 모델과 논리 모델을 명확히 구분해주지 않는다는 것이다. 그렇다고 Peter chen의 방식으로 모델링을 하자니 심볼의 크기가 너무 크고, 릴레이션형 모델도 표현하고 있어 역시 애매모호하다. 그러므로 릴레이션형 모델로 표현하되, 개념 모델에서 논리 모델로 한 번에 표현하고 생각할 것이다. 굳이 나누자면 개념 정의까지만 하는 것을 개념 모델로 한정하겠다.

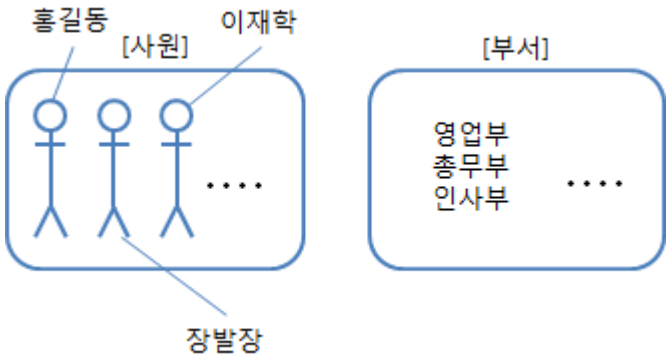
2.4 종, 관계, 속성

'사원'은 회사의 구성원을 말한다. '부서'는 기관, 기업, 조직 따위에서 일이나 사업의 체계에 따라 나뉘어 있는, 사무의 각 부문을 말한다. 사원과 부서의 데이터 모델에서 사원은 사물이며, 부서는 개념이다. 일반적으로 부서와 사원의 ERD는 다음과 같이 표현된다. 규정성에 따라 데이터 모델이 어떻게 변화하는지 살펴보자.

<그림 2.4.1>

종

1장의 형이상학적 실재론에서 종을 구별하는 기법을 살펴보았다. 3가지 기법 중 사원과 부서에 적용된 기법은 분류화(classification)다.



<그림 2.4.2>

종을 잘 도출했는지에 대한 검증은 직접 실제 개체를 논리적으로 표현해 보는 방법이 가장 단순하고 쉽다. 논리 모델인 릴레이션으로 만들어보자.

성명	성별	입사일자	최종학력
이재학	남	2007-11-12	대졸
홍길동	남	2009-02-01	대학원졸
장발장	남	2010-12-01	대졸

<그림 2.4.3>

각각의 '사원' 개체들은 성명, 성별, 입사일자, 최종학력 속성이 존재하는 것을 알 수 있다. 여기서 중요한 것은 각각의 개별 속성을 하나의 개체로 인식하는 것이 아니라 여러 속성들이 합쳐진 형태로써 개체로 인식된다는 것이다. 일단 종으로 인식되었다면 종에 속한 각각의 개체도 인식될 수 있다는 전제하에 종을 인식했을 것이다. 여기에서는 각각의 사원을 주민등록번호로 식별하므로 '주민등록번호' 속성이 추가 되었다.

성명	성별	주민등록번호	입사일자	최종학력
이재학	남	761218-*******	2007-11-12	대학원졸
홍길동	남	740114-*******	2009-02-01	대졸
장발장	남		2010-12-01	대졸

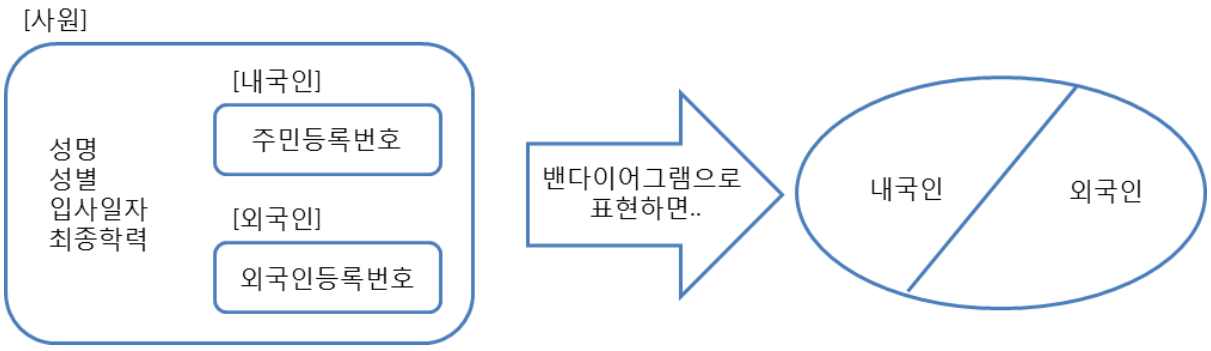
<그림 2.4.4>

'주민등록번호' 속성을 식별자라고 했으나 '장발장'은 외국인으로 주민등록번호가 없다. 그러므로 다음과 같이 '외국인등록번호'를 사용한 형태가 될 것이다.

성명	성별	외국인등록번호	주민등록번호	입사일자	최종학력
이재학	남		761218-*******	2007-11-12	대졸
홍길동	남		740114-*******	2009-02-01	대학원졸
장발장	남	770717-1500009		2010-12-01	대졸

<그림 2.4.5>

그러므로 여기서 개체와 개체 사이에 규정성의 차이가 발생함을 알 수 있다. 규정성의 차이는 종의 차이가 있음을 말한다. 그러므로 다음과 같이 일반화(generalization)될 것이다.



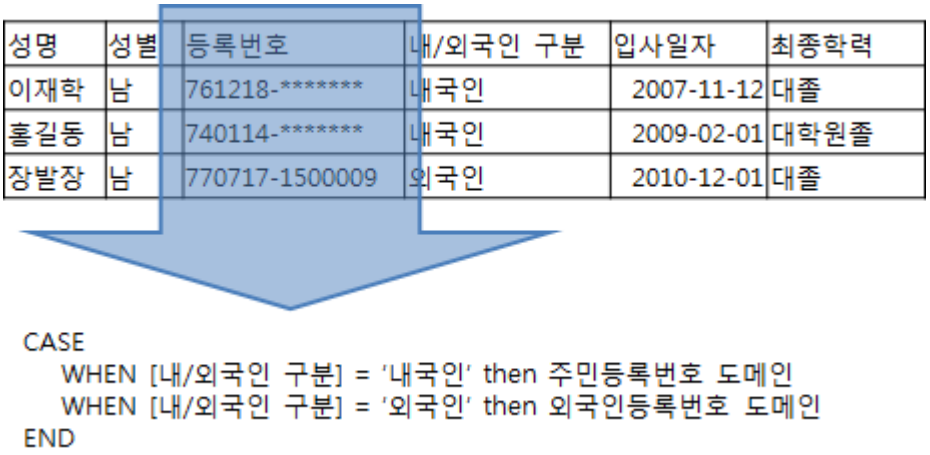
<그림 2.4.6>

표 형태로 표현하는 방법은 위의 방법 말고도 '주민등록번호'와 '외국인등록번호'를 통합하여 속성의 규정성을 확장하는 방법을 사용할 수 있다.

성명	성별	등록번호	내/외국인 구분	입사일자	최종학력
이재학	남	761218-*******	내국인	2007-11-12	대졸
홍길동	남	740114-*******	내국인	2009-02-01	대학원졸
장발장	남	770717-1500009	외국인	2010-12-01	대졸

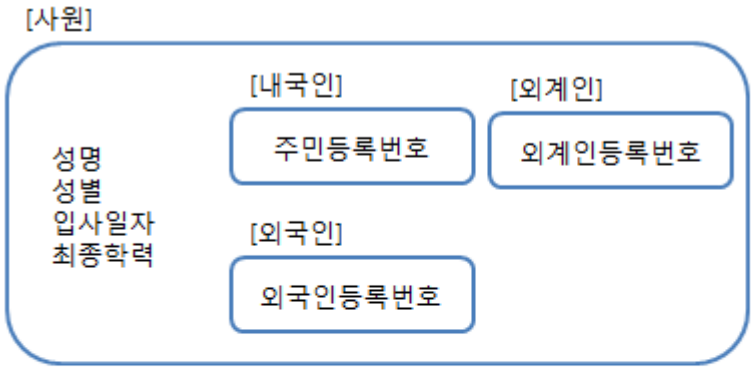
<그림 2.4.7>

'등록번호'라는 속성은 '내/외국인 구분' 속성에 따라 도메인이 달라진다. 이러한 표현 방법은 유연성이 높아지고, 속성 하나가 없어진 만큼 단순해졌다. 하지만,속성간의 도메인 종속성이 발생하는 단점이 있다.



<그림 2.4.8>

여기서 유연성이 높아졌다고 했는데, 이는 내국인, 외국인 이외에 다른 종이 생길 경우 모델상의 큰 변화가 없음을 의미한다. 데이터 모델이 현실의 변화에 큰 변화가 없다는 것은 시스템의 유지보수 비용의 절감을 의미한다. 예를 들어, '외계인'이 추가된다면 다음과 같은 데이터 모델이 될 것이다.

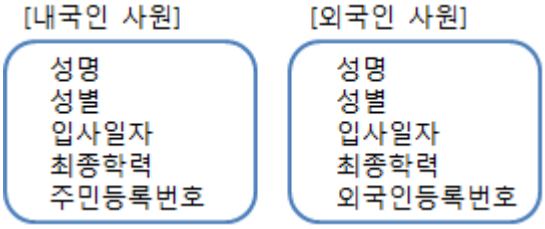


성명	성별	등록번호	내/외국인 구분	입사일자	최종학력
이재학	남	761218-*******	내국인	2007-11-12	대졸
홍길동	남	740114-*******	내국인	2009-02-01	대학원졸
장발장	남	770717-1500009	외국인	2010-12-01	대졸
깐따삐	남	780101-2020202	외계인	2012-04-01	대졸

<그림 2.4.9>

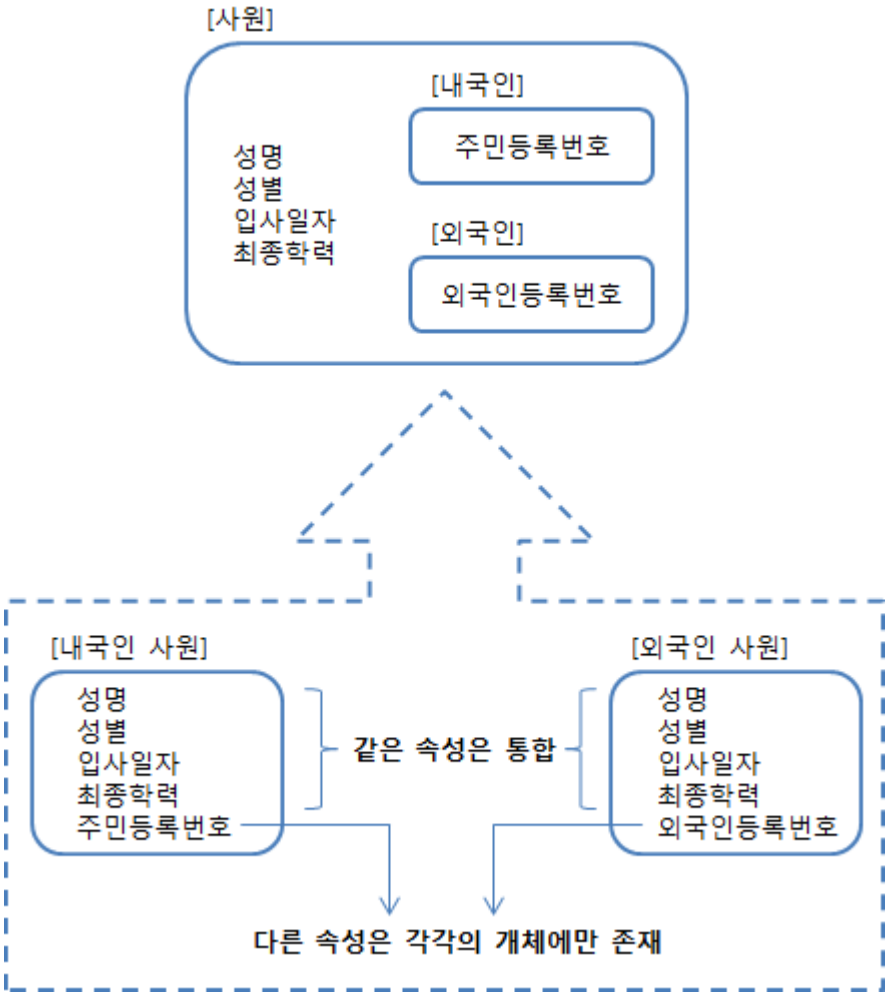
물론 외계인이 성명, 성별, 입사일자, 최종학력 같은 속성이 있다는 가정하에서다. 하지만, 먼 미래에도 외계인이 사원이 되는 것은 현실세계에 존재할 가능성이 거의 없으므로 데이터 모델 자체는 유연성이 있으나 필요 없는 유연성이므로 이 모델에서의 유연성은 장점도 단점도 아니므로 유연성이 장점이라고 해서는 안 된다. 데이터 모델은 반드시 현실을 반영해야 하며, 일치성이 있어야 한다.

실제로 구축되어 있는 데이터 모델은 다음과 같을 수도 있다. 이렇게 구축되어 있어도 비웃거나 해서는 안 된다. 그 당시에는 외국인은 사원이 될 수 없는 업무 규칙이 있었을 지도 모른다. DBA가 퇴사했었을지도 모른다. 그러므로 좋은 모델을 도출하고자 하는 순수한 마음으로 현재의 모델을 살펴보는 것이 좋다.



<그림 2.4.10>

모델러는종을 보고 규정성의 차이가 많은지 적은지 알 수 있어야 한다. 만약 규정성의 차이가 크지 않은데, 종이 따로 존재한다면 종을 통합하여 데이터 모델의 종을 일반화하여 유형으로 묶어 종의 수를 줄여 개체간의 상호작용 수를 줄여 전체 시스템의 복잡도를 떨어뜨리는 것이 좋다.



<그림 2.4.11>

이와 같은 방법도 물론 일반화다. 물론 Bottom-Up과 Top-Down의 차이일 뿐이다. Bottom-Up과 Top-Down 중 어떤 방법이 좋고 나쁜지는 상관없다. 단지 현실세계를 잘 표현할 수 있기만 하면 된다.

우리는 앞서 데이터 모델을 표 형태로 표현해보았다. 아마 눈치 챈 독자들도 있을 것이다. 필자는 개념 모델을 자연스럽게 논리 모델(릴레이션형 모델)로 전환하게 하려고 했던 것이다. 위에서 사원의 일반화된 모델을 다음과 같이 2가지 형태의 릴레이션으로 전환한 것이다.

A타입

성명	성별	등록번호	내/외국인 구분	입사일자	최종학력
이재학	남	761218-*******	내국인	2007-11-12	대졸
홍길동	남	740114-*******	내국인	2009-02-01	대학원졸
장발장	남	770717-1500009	외국인	2010-12-01	대졸

B타입

성명	성별	외국인등록번호	주민등록번호	입사일자	최종학력
이재학	남		761218-*******	2007-11-12	대졸
홍길동	남		740114-*******	2009-02-01	대학원졸
장발장	남	770717-1500009		2010-12-01	대졸

<그림 2.4.12>

음영으로 표시한 부분이 후보키가 될 것인데, 여기서 '사원'은 마스터 데이터에 해당되므로 기본키가 짧은 것이 좋다. 왜냐하면 사원 개체들은 많은 개체들과 관계를 가질 것이기 때문이다. 릴레이션형 모델의 관계는 외부키로 표현[1]된다. 외부키는 관계를 가지는 릴레이션의 기본키로부터 만들어진다. 그러므로 기본키가 2개의 어트리뷰트로 만들어졌다면 외부키도 2개의 어트리뷰트가 된다. 그러므로 조금 단순화할 필요가 있다. 물론 어트리뷰트 1개가 기본키가 되더라도 어트리뷰트의 도메인이 복잡하거나 데이터 크기가 커질 수 있다. 그래서 일반적으로 마스터 데이터에 해당되는 릴레이션의 기본키는 대체키를 사용하게 된다. 여기에서는 사원번호가 추가 된다.

A, B타입 어떤 모델을 선택해도 괜찮다. 하지만 필자라면 A타입대신에 B타입을 선택할 것이다. 앞서 설명했듯이 유연성은 고려대상이 아니다. A타입은 속성간의 종속성이 있는데, 속성간의 종속성은 스키마만으로 알 수 없거나, 코드 테이블을 참조해야만 한다. 이는 시스템의 복잡성을 가중시킨다.

이와는 다르게 B타입은 실제 데이터를 보지 않고 스키마만으로도 쉽게 업무규칙이 드러난다. 물론 외국인과 내국인을 명확하게 나누어주는 컬럼이 없어 암묵적으로 외국인등록번호와 주민등록번호를 통해 알아내야 한다. 하지만 유도 어트리뷰트로 내/외국인을 구분할 수 있으므로 역시 B타입이 선택할 것이다. 사원번호를 추가하면 다음과 같은 모델이 된다.

사원번호	성명	성별	외국인등록번호	주민등록번호	입사일자	최종학력
1	이재학	남		761218-*******	2007-11-12	대졸
2	홍길동	남		740114-*******	2009-02-01	대학원졸
3	장발장	남	770717-1500009		2010-12-01	대졸

<그림 2.4.13>

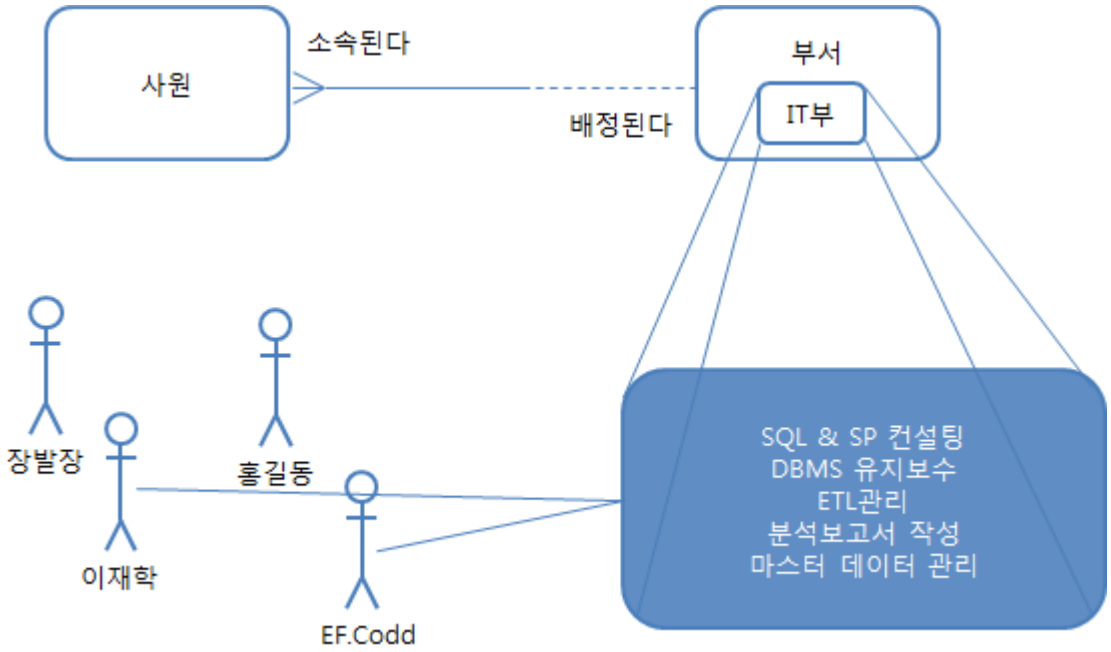
[1]그렇다고 모든 외부키가 개념 모델의 관계를 표현한 것은 아니다.

실무에서는 이런 패턴이 자주 등장하는데, 대부분 B타입보다는 A타입이 선호된다. 왜냐하면 데이터 모델의 유연성과 데이터베이스 비전문간들의 null에 대한 이해 부족 때문이다. 물론 하드웨어가 크게 발전한 것도 한 몫 한다. 성능 보다는 유연성에 조금 더 집중하여 외부의 변화에 쉽게 적응할 수 있는 시스템을 만드는 것이 시스템 전체의 비용을 줄여줄 것이다. 위의 모델은 '사원'에 한정하였기에 B타입을 선택할 수 있었다. 행위의 주체가 되는 사람이나 조직에 대한 상당히 일반화된 데이터 모델도 추후에 살펴볼 것이다.

관계

이제 관계에 대해 살펴보자. 형이상학적 실재론에서의 관계는 대칭적 관계와 비대칭적 관계가 있었다. 비대칭적 관계는 데이터의 생성 순서에 따른 종속 관계와 개체를 한정 짓는데 사용하는 관계가 있다. 관계에 대해서는 2장의 마지막에 정리할 것이다. 여기에서는 '사원'과 '부서'가 어떤 관계인지 파악해 보도록 하겠다.

회계부서에서 데이터베이스 백업 작업을 할 수 있는 역량을 가졌다고 해서그들 스스로가 데이터베이스 백업을 하지는 않는다. 왜냐하면 백업은 IT부서의 업무이기 때문이다. 부서는 부서 나름대로의 업무 영역이 있으며, 해당 부서에 배정된 사원은 부서의 업무 범위에서 일을 하게 된다. 즉, 부서는 사원이 어떤 일들을 해야 하는지에 대한 범위를 한정하는 관계를 가진다.



<그림 2.4.14>

부서와 사원은 1:다의 관계다. 말로 풀어보면 다음과 같다.

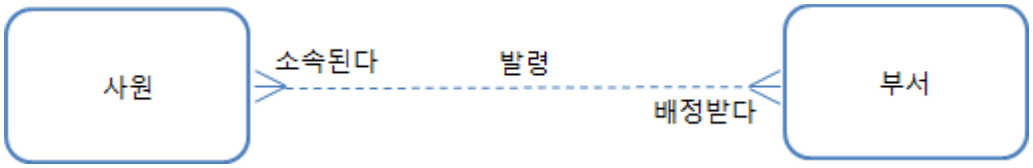
- 각각의 사원은 반드시 하나의 부서에만 소속된다.
- 각각의 부서는 때때로 여러 사원이 배정된다.

실제로 데이터 모델링을 하다 보면 관계의 명칭이 애매모호한 경우가 많다.위 예제의 '소속된다', '배정된다'와 같은 관계 동사도 찾아내는 것이 쉽지만은 않다. 또한 방향성과 잘 맞는지도 모르겠다. 이유가 뭘까? 관계가 대칭적이기 때문이다. 부모와 자식과 같은 경우는 단 방향이기 때문에 관계 명칭이 비교적 쉽게 도출된다. 하지만, 사원과 부서간의 관계는 대칭 관계이므로 양방향성을 가진다. 그러므로 단방향의 관계 명칭을 도출하는 것이 어렵다. 다른 서적이나 강좌에서는 단방향, 양방향 모두의 관계를 정의하라고 하지만, 이 책에서는 대칭 관계와 비대칭 관계를 구분할 것이다. 대칭 관계는 대부분 적당한 관계 명칭 하나로도 표현력이 충분하다. 만약 관계를 표시(선)하는 것만으로도 의사전달이 충분하다고 판단되면 관계 명칭을 과감히 생략해도 된다. 모두가 알고 있는 사실을 굳이 어색한 명칭으로 끼워 맞추는 것은 낭비다.

참고: 1:다 관계는 부모-자식 or 마스터-디테일?

1:다 관계에서 1측의 릴레이션을 부모, 다측의 릴레이션을 자식으로 부르는 경우도 있고, 1측을 마스터, 다측을 디테일로 부르는 경우도 있다. 문맥상으로 본다면 무엇을 의미하는지 대략 알 수 있지만, 1:다 라고 해서 무조건 부모-자식 관계나 마스터-디테일 관계가 아니다. 부모-자식 관계는 존재 종속(existence-dependent) 관계다. 즉, 한 개체의 존재가 다른 개체의 존재에 직접적인 영향을 끼치는 경우를 말한다. 마스터-디테일에서 디테일은 마스터의 설명력을 더해주는 역할을 하는 것일 뿐 서로 다른 개체를 표현하는 것은 아니다. 사원과 부서의 관계는 부모-자식 관계도 아니고, 마스터-디테일 관계도 아닌 그냥 일반적인 관계다.

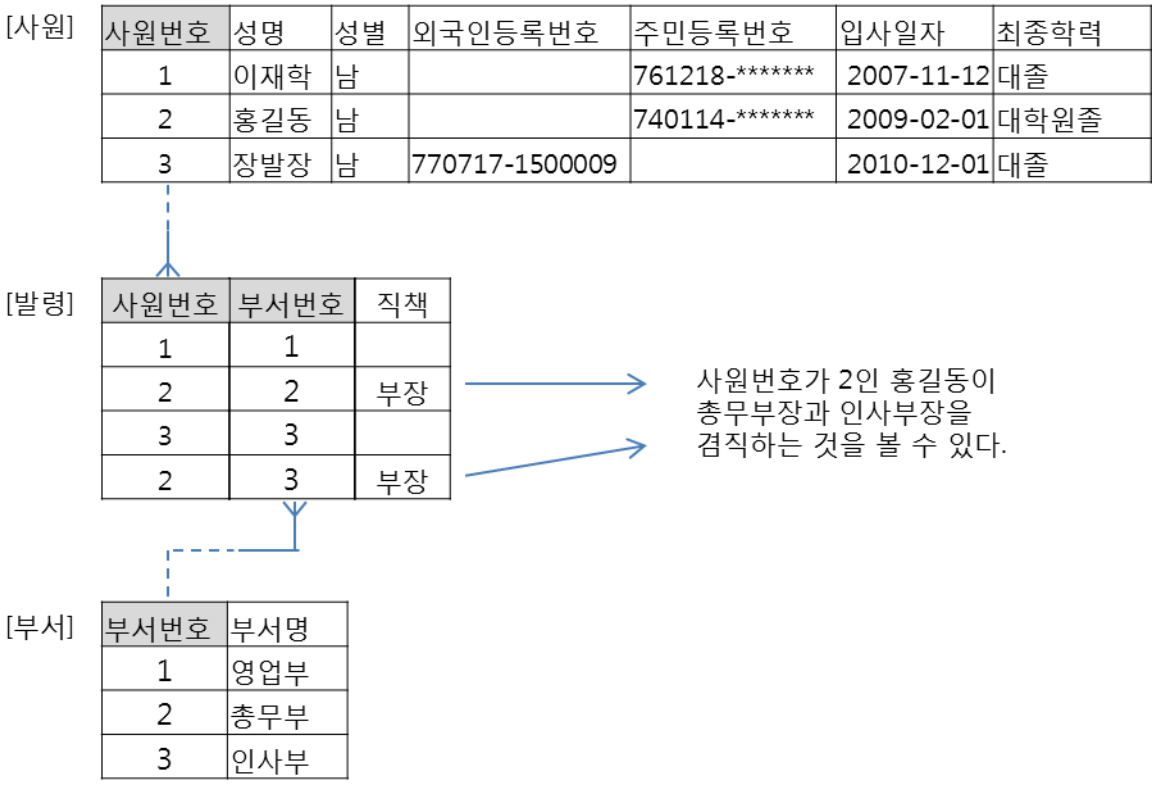
한 명의 사원이 2개 이상의 부서에 소속될 수 있다는 요구사항이 있다고 가정해 보자. 예를 들면, 부서장 겸직의 경우다. 그러면 1:다의 관계는 다:다의 관계가 된다.



<그림 2.4.15>

릴레이션으로 만들어보면 다음과 같이 3개의 릴레이션으로 구성될 것이다. 사원과 부서의 중간에

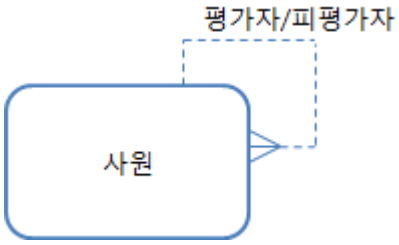
있는 릴레이션이 관계를 표현한'발령' 릴레이션이다.



<그림 2.4.16>

이러한 관계 요구사항은 간과하기 쉽다. 그래서 종종 실무자들이 최종 구현된 결과물을 사용하다가 뭔가 잘못되었음을 알게 되고 그제서야 개발자들에게 추가적인 요구사항이 전달된다. 대규모 프로젝트의 경우 이렇게 늦은 시기에 자주 설계가 변경된다면 변경이 어려우므로 소위 말하는 '땜빵'을 하기 시작한다. 여기 저기 덕지덕지 컬럼과 테이블이 추가되는 등의 일이 벌어진다. 프로젝트가 어떻게든 마무리 되어도 이렇게 '땜빵'한 시스템은 변화에 취약해 질 수밖에 없다. 한 부분을 변경하면 종속성으로 인한 부수효과(side-effect)로 인해 시스템의 장애가 자주 발생하고는 한다. 결국은 시스템의 생명주기(life-cycle)가 짧아지게 된다. 그러므로 설계 이전 과정에서 최대한 요구사항을 밝혀내는 것이 중요하다.

이제까지는 종이 다른 개체들 간의 관계에 대해서만 알아보았다. 종이 같은 개체들 간의 관계는 일반적으로 순환하는 관계로 표현된다. 다음은 사원간에 평가자와 피평가자의 관계를 나타내고 있다.



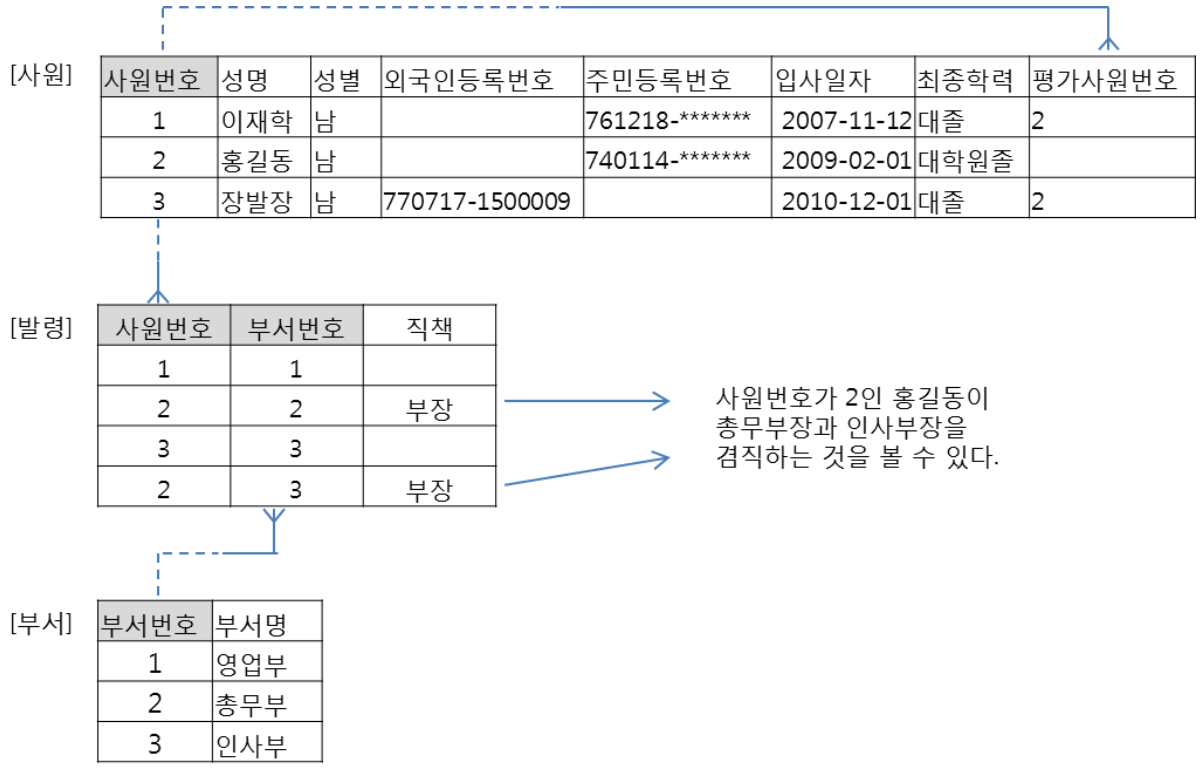
<그림 2.4.17>

이를 릴레이션으로 만들어보면 다음과 같다.

사원번호	성명	성별	외국인등록번호	주민등록번호	입사일자	최종학력	평가사원번호
1	이재학	남		761218-*******	2007-11-12	대졸	2
2	홍길동	남		740114-*******	2009-02-01	대학원졸	
3	장발장	남	770717-1500009		2010-12-01	대졸	2

<그림 2.4.18>

사원번호가 1인 이재학 사원은 사원번호가 2인 홍길동에게 평가를 받는 피평가자다. 물론 사원번호 3인 장발장도 마찬가지로 피평가자다. 마찬가지로 부서도 상/하위 개념이 있으므로 순환관계로 표현될 수 있지만, 여기에서는 맛보기 수준으로만 살펴보고 순환관계에 대한 자세한 이야기는 다른 절에서 다루도록 하겠다. 궁금한 독자는 목차를 참고하여 바로 내용을 확인해도 좋다. 순환관계가 적용된 릴레이션은 다음과 같다.



<그림 2.4.19>

앞서 데이터를 마스터, 트랜잭션, 참조 데이터로 분류했었다. 여기서 [사원], [부서]는 마스터 데이터이며, [발령]은 트랜잭션 데이터다. 참조 데이터는 없다.

속성

주민등록번호나 외국인등록번호는 국내에 거주하는 각각의 사람들을 식별하기 위해 국가기관에서 만들어낸 일종의 설계 속성이다. 사원과 부서에서 실제로 다루어야 할 속성들의 수는 전적으로 요구사항에 달려 있다. 요구사항은 예측되는 미래의 요구사항도 포함된다. 부서의 경우 계속된 예제에서는 부서명만 관리하면 되므로 설명을 생략하고 사원의 속성을 가지고 이야기 해보자. 사원의 경우 성명, 성별, 입사일자, 최종학력이 고려대상이 된다.

성명은 사원의 '성 + 명'으로 복합속성이 될 수도 있고, 또한 한글, 영문, 한문과 같은 속성값에 대한 유형이 있을 수 있다. 영어권의 DB설계의 예제들을 보면 First_Name, Middle_Name, Last_Name과 같은 컬럼들을 쉽게 볼 수 있다. 그렇다고 무턱대고 이름과 성을 분리하기에는 설득력이 모자라다. 우리는 1장의 이론에서 릴레이션의 특성 중 어트리뷰트의 원자값에 대해 알아보았다. 이를 토대로 복합 어트리뷰트로 취급할 것인지 아니면 단일 어트리뷰트로 취급할 것인지 결정하면 된다. 원자값이라는 것은 사용 패턴을 보고 판단하면 된다. 예를 들어, Last_Name과 같은 경우 트랜잭션 처리에서 하나로 취급된다면 의미 있는 최소단위로 원자값이다. 외국의 경우

결혼을 하면 남자의 성으로 바뀌므로 따로 취급하는 경향이 많다. 왜냐하면 일반적으로 남/녀의 비율이 각각 50%이므로 UPDATE가 빈번하기 때문이다. 하지만, 우리나라는 성과 이름이 거의 바뀌지 않으므로 '성'과 '명'은 복합 어트리뷰트가 아닌 단일 어트리뷰트로 취급하는 것이 일반적이다. First_Name, Middle_Name도 변경 횟수가 이상치로 봐도 될 만큼 빈도가 매우 낮기 때문에 단일 어트리뷰트로 취급하는 것이 좋다. 하지만, Middle_Named을 생략해서 조회하는 경우가 있다면 First_Name과 Middle_Name은 분리되는 것이 옳은 판단일 것이다.

성별은 단일 어트리뷰트로 상식적인 수준으로 생각할 수 있다. 주의해야 할 것은 성별과 같은 어트리뷰트 값이 코드화 될 수 있는 적은 종류라면 전사적인 관점에서 다른 코드체계를 따르지 않도록 전사 표준을 먼저 살펴보는 것이 좋다. 예를 들면, A시스템에서는 'M' or 'F'로 표시하고 B시스템에서는 '남' or 여'로 표기한다면 추후 전사적인 관점을 취할 때에 또 다른 비용이 들어가게 된다. 그러므로 코드화 될 후보가 있다면 반드시 표준을 따르거나 표준화하도록 노력해야 한다.

입사일자에 따라 안식년이라든지 포상휴가가 주어진다면 특별히 관리될 필요가 있다. 또한 재입사의 경우는 입사일자를 어떻게 할 것인가에 대한 고려도 필요하다. 이러한 비즈니스 룰들은 각 조직마다 다르므로 사규나 복지와 관련된 문서나 이해관계자를 통해 알아내야 한다.

최종학력도 대졸자가 입사하여 재직 중에 대학원을 졸업하는 경우가 있다면 특별히 관리되어야 한다. 만약 최종학력에 따라 연봉이 다르게 책정되거나 하는 경우가 있다면 이 또한 특별히 관리될 필요가 있다. 발생할 수 있는 데이터의 개수가 확정적이므로 최종학력도 코드화 대상이다.

논리 모델의 경우는 어트리뷰트의 도메인 정의해야 한다. 도메인을 정의하는 것은 어렵지 않으므로 '발령' 릴레이션의 '직책'에 대해서만 도메인을 정의해보자.

이름(한글)	직책
이름(영문)	Position
약어	Pos
의미	직무상의 책임
데이터 타입	char
길이	6
형식	
숫자정확도	
허용범위 (입력 가능한 값의 목록)	과장, 부장, 본부장, 사장
유일성	Non-Unique
널값허용	Null
기본값	
룰	

<표 2.4.1>

물리적인 구현에서의 '사원'

<그림 2.4.12>에서 필자는 B타입을 선택했었다. 하지만, B타입의 선택은 여전히 탐탁지 않다. A타입으로 선택했다면 '등록번호'는 후보키로도 손색이 없다. 하지만, B타입은 '외국인등록번호'와 '주민등록번호'가 후보키가 되기에는 null을 허용해야 하므로 후보키의 자격이 없다.

A타입

성명	성별	등록번호	내/외국인 구분	입사일자	최종학력
이재학	남	761218-*******	내국인	2007-11-12	대졸
홍길동	남	740114-*******	내국인	2009-02-01	대학원졸
장발장	남	770717-1500009	외국인	2010-12-01	대졸

B타입

성명	성별	외국인등록번호	주민등록번호	입사일자	최종학력
이재학	남		761218-*******	2007-11-12	대졸
홍길동	남		740114-*******	2009-02-01	대학원졸
장발장	남	770717-1500009		2010-12-01	대졸

<그림 2.4.12>

또한 '외국인등록번호'와 '주민등록번호'로 자주 쓰이는 검색조건이라면 인덱스를 생성해야 할 것이다. A타입의 경우 '등록번호'에 만 인덱스를 생성하면 되지만, B타입의 경우는 '외국인등록번호'와 '주민등록번호'에 대해 인덱스를 생성해야만 한다. B타입의 경우 SQL도 복잡해지므로 이런 경우는 B타입이 좋은 선택이 아니다. 예를 들면, 다음과 같은 패턴의 SQL이 만들어진다.

```
--A타입
select 성명, 성별, 입사일자
from 사원
where 등록번호 = '770717-1500009'

--B타입
select 성명, 성별, 입사일자
from 사원
where isnull(주민등록번호, 외국인등록번호) = '770717-1500009'
```

B타입의 경우처럼 SQL이 작성될 우려가 있다는 것은 모델러에게는 반가운 소식은 아니다. 만약 SQL을 검수하는 사람이 없다면 우울한 일이 될 수 있다. 하지만, SQL 숙련자라면 다음과 같이 SQL을 작성할 것이다. (Where 절에 상수 값이나 매개변수가 왼쪽에 위치한 예제가 찾기 힘듦)

```
--B타입
```

```
select 성명, 성별, 입사일자
from 사원
where '770717-1500009' in (주민등록번호, 외국인등록번호)
```

이렇게 SQL을 작성하면 컬럼을 변형시키지 않아 인덱스를 이용한 검색을 할 수 있다. 만약 이러한 패턴의 SQL이 매우 빈번하게 실행되어 성능에 악영향을 끼친다면 A타입을 선택해야 한다. 하지만, '사원'의 경우는 많아야 만 건 단위이므로 문제가 되지는 않는다. 두 타입의 모델 중에서 SQL이 더 단순하다는 것은 모델이 더 단순함을 의미한다.

다른 예로 외국인, 내국인의 수를 구하는 요구사항이 있다고 가정해보자. 다음과 같이 SQL 작성될 것이다.

```
--A타입
select
      count(case when [내/외국인 구분] = '내국인' then 1 end) 내국인수
,     count(case when [내/외국인 구분] = '외국인' then 1 end) 외국인수
from 사원


--B타입
select
      count(주민등록번호) 내국인수
,     count(외국인등록번호) 외국인수
from 사원
```

이 경우는 SQL이 A타입이 B타입보다 복잡하다. 이 SQL만 본다면 A타입이 더 복잡하다. 결국은 A타입을 선택할 것인지 B타입을 선택할 것인지는 요구사항에 대한 조사와 분석이 얼마나 철저히 이루어지느냐에 따라서 결정될 것이다. 정보(요구사항 조사/분석의 결과)의 질에 따라서 의사결정을 하게 되므로 역시 요구사항 조사/분석은 매우 중요하다.

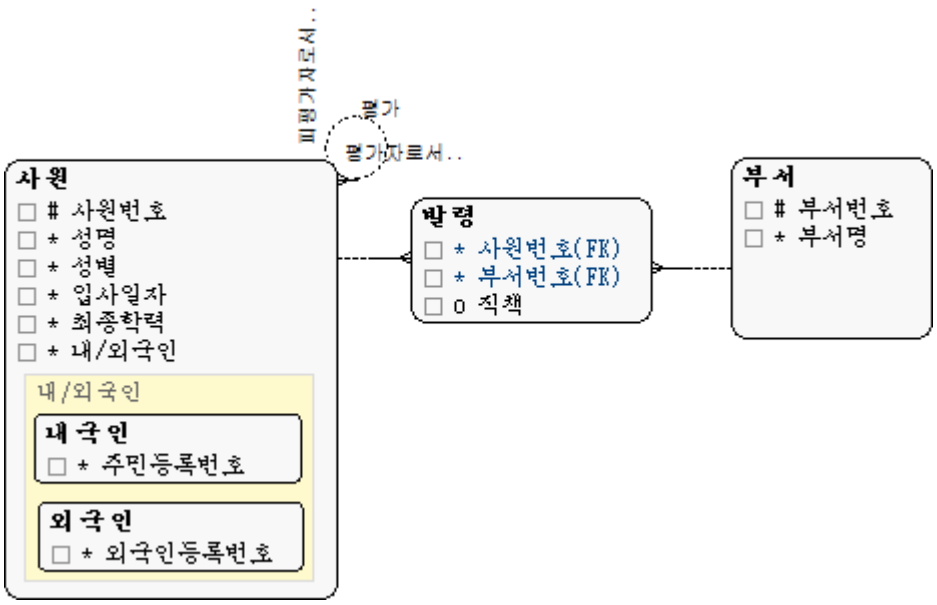
다시 말하지만, B타입보다는 A타입이 선호된다. 유연성도 있을뿐더러 개발자들에게 익숙한 패턴이라면 A타입을 선택하는 것이 옳다. 그러므로 정보가 충분하지 못하다면[2] A타입을 선택하자. 또한 A타입은 데이터만으로도 충분한 설명력을 지닐 수 있다. 하지만, B타입은 문서를 봐야만 알 수 있는 경우도 있다. 필자는 편의상 이력관리까지 B타입으로 설명할 것이다.

[2]아마도 대부분은 충분하지 못할 것이다.

2.5 이력관리

이력관리는 종, 관계, 속성이 시간에 따른 변화를 관리하고자 하는 것이다. 데이터 모델링에서 이력관리부분을 가장 마지막에 수행하는 것이 정석이다. 마지막에 해야 하는 이유는 이력관리의 대상이 종, 관계, 속성의 변경에 대한 것이기 때문이다. 데이터 모델링은 업무를 정의하는 과정의 도구인데, 정의되지 않은 것을 대상으로 정의한다는 자체가 논리적으로 맞지 않다. (많은 책에서 이유를 밝혀주지 않아 필자 나름대로의 생각하고 경험한 이유)

앞서 2.4절에서 사원과 부서에 관련된 데이터 모델을 만들어 보았다. 2.4절의 모델은 UML의 객체 다이어그램(object-diagram)과 비슷한 인스턴스 다이어그램(instance-diagram)로 표현했었다. 나름대로 이해를 빠르게 하기 위해 인스턴스 다이어그램으로 표현하였는데, 이제부터는 논리 모델로 표현할 것이다. 인스턴스 다이어그램은 필요에 따라 표현될 것이다. 2.4절의 사원-부서 인스턴스 다이어그램을 논리 모델로 표현하면 다음과 같다.



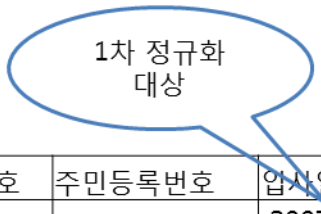
<그림 2.5.1>

이 모델은 이력관리를 포함하지 않았었다. 물론 종속성이 없기 때문에 정규화도 필요 없다. 이제 이력관리를 포함하여 모델을 좀 더 발전시켜 보자.

속성에 대한 이력관리

부서의 속성은 이력관리가 필요 없다고 가정하고, 사원의 속성과 관계 대해서만 집중하자. 먼저

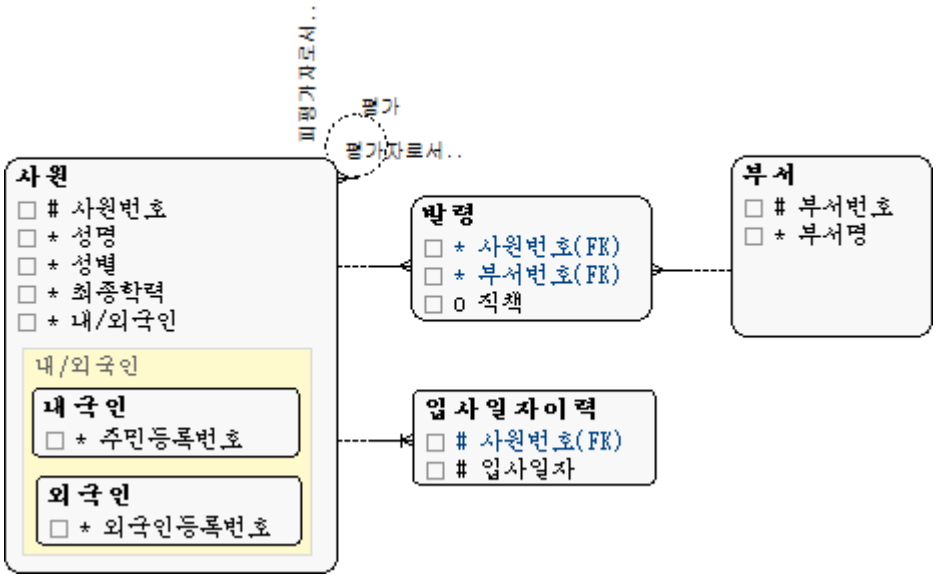
사원번호다. '사원번호'는 이미 '주민등록번호'의 대체키로 정의했으므로 '주민등록번호'와 함께 이력관리의 대상에 제외된다. '성명', '성별'의 경우도 시간이 지나도 변화가 없다는 가정하에 제외하자. '입사일자'의 경우 재입사에 대한 관리를 한다고 가정해보자. 만약 '이재학'사원이 재입사를 했다면 '입사일자'는 다중값 속성이 된다. 그러므로 '입사일자' 속성은 다음과 같이 1차 정규화의 다생이 된다.



사원번호	성명	성별	외국인등록번호	주민등록번호	입사일자	최종학력
1	이재학	남		761218-*******	2007-11-12 2011-10-01	대졸
2	홍길동	남		740114-*******	2009-02-01	대학원졸

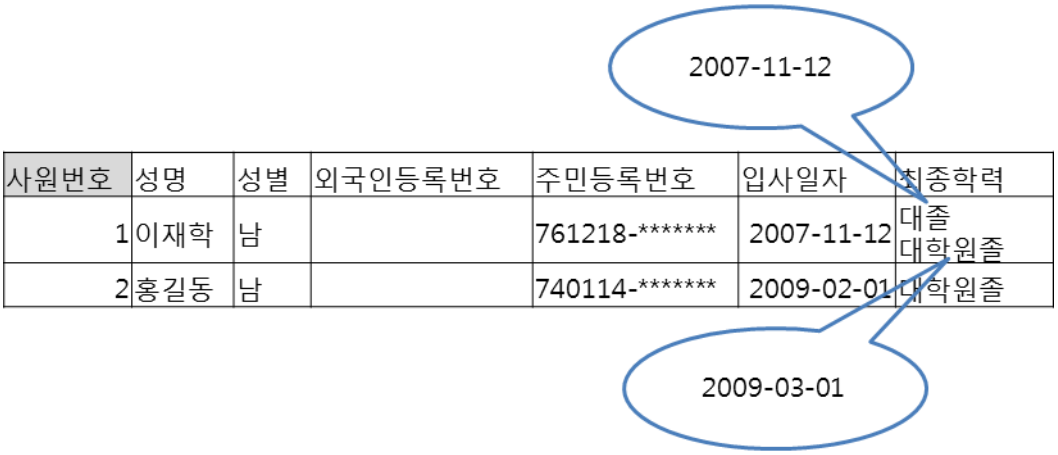
<그림 2.5.2>

그러므로 '입사일자'에 대한 이력관리를 한다고 가정하면 데이터 모델은 다음과 같이 '입사일자이력' 릴레이션을 추가된다.



<그림 2.5.3>

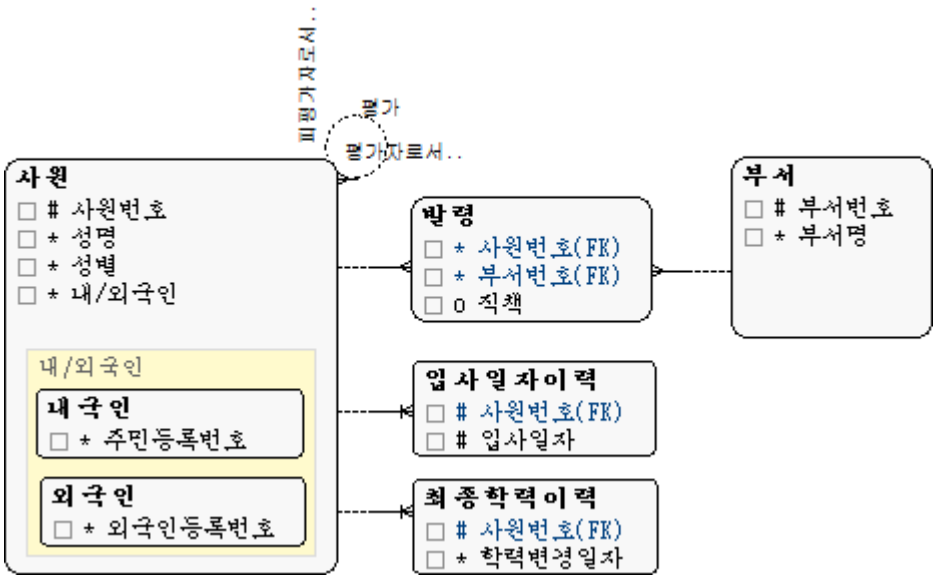
'최종학력'도 이력관리를 한다고 가정해보자. '입사일자'와 마찬가지로 하나의 사원에 2개 이상의 어트리뷰트값을 가지므로 1차 정규화 대상이 된다. 이와 더불어 값 자체가 원자값이 아니게 된다.



사원번호	성명	성별	외국인등록번호	주민등록번호	입사일자	최종학력
1	이재학	남		761218-*******	2007-11-12	대졸 대학원졸
2	홍길동	남		740114-*******	2009-02-01	대학원졸

<그림 2.5.4>

'최종학력'에 대한 이력관리를 한다면 '최종학력이력' 릴레이션을 추가하여 다음과 같이 모델이 변경된다.

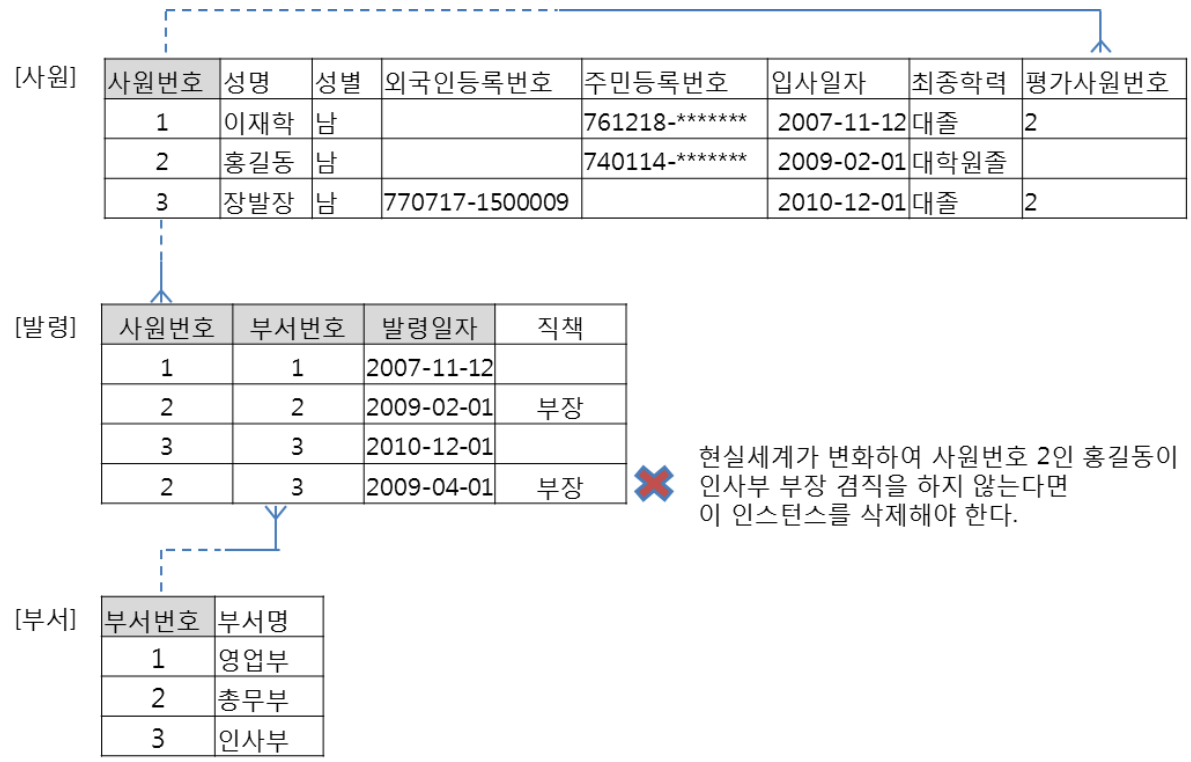


<그림 2.5.5>

관계에 대한 이력관리

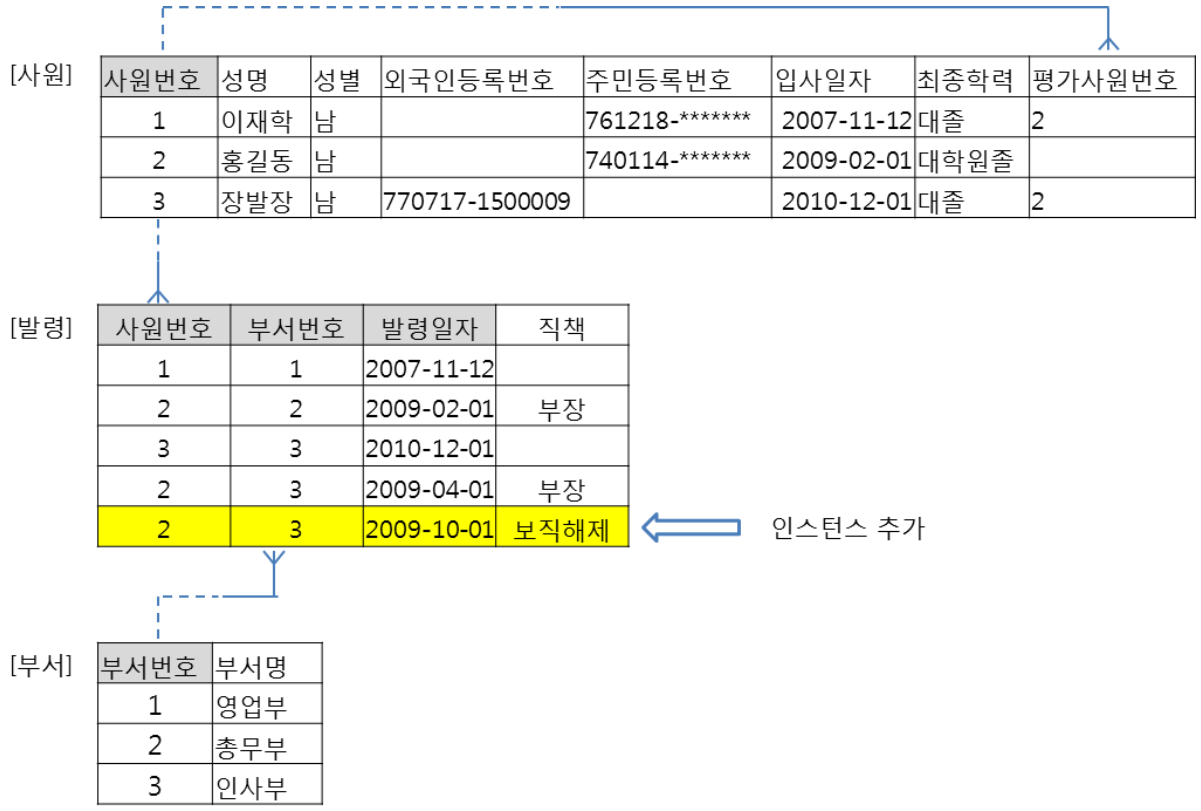
이제 관계에 대한 이력관리를 해보자. 앞서 만들어 본 데이터 모델에서 '입사일자이력'과 '최종학력이력'은 개념 모델의 '관계'가 아닌 릴레이션의 릴레이션쉽이다. 그러므로 관계에 대한 이력관리 대상이 아니다. 여기서 살펴볼 관계는 사원-사원 관계와 사원-부서의 관계다. 먼저 사원-부서 관계부터 살펴보자.

사원-부서 관계는 '발령' 릴레이션으로 모델링 되었다. 앞서 '현재'만을 가정했으므로 '발령일자'와 같은 속성은 없었다. 편의상 '발령일자'를 추가했다는 가정하에 '홍길동' 사원이 '2009-10-01'부터 인사부 부장 겸직하지 않는다는 가정을 해보자.



<그림 2.5.6>

현재만을 고려한다면 '홍길동' 사원이 인사부 부장 직책을 맡고 있다는 인스턴스를 삭제해야 한다. 그렇다고 삭제를 하지 않으면 인사부 부장에 대한 직책 수행자가 누구인지 불명확해 진다. 그러므로 만약 2009년 04월 01일부터 2009년 10월 01일까지 '홍길동' 사원이 인사부 부장이었다는 이력을 관리해야 한다. 릴레이션은 다음과 같이 변경해야 한다.



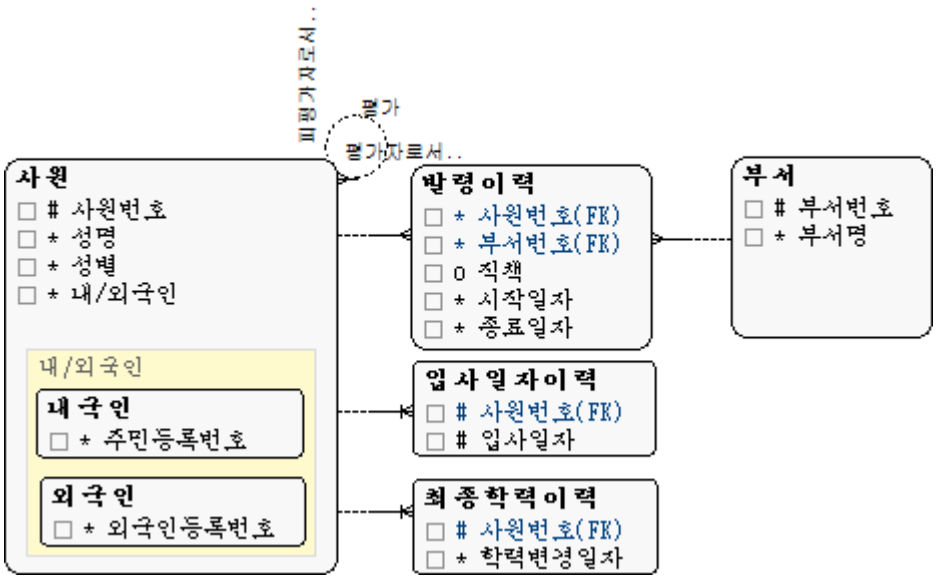
<그림 2.5.7>

앞서 정의한 직책에 대한 도메인 정의에서 입력 허용 범위는 {과장, 부장, 본부장, 사장} 이었다. 위의 인스턴스 다이어그램에서는 인스턴스가 추가되었지만 '직책'의 도메인에 문제가 생겼음을 알 수 있다. 그래서 직책의 입력 허용 범위에 {과장, 부장, 본부장, 사장, 보직해제} 와 같이 '보직해제'를 추가한다면 '직책' 자체의 의미가 확장되어 불명확해진다. 그러므로 다음과 같이 시작~종료 형태로 이력을 관리해야 한다.

사원번호	부서번호	시작일자	종료일자	직책
1	1	2007-11-12	9999-12-31	
2	2	2009-02-01	9999-12-31	부장
3	3	2010-12-01	9999-12-31	
2	3	2009-04-01	2009-09-30	부장

<그림 2.5.8>

이를 반영한 논리 모델은 다음과 같다.



<그림 2.5.9>

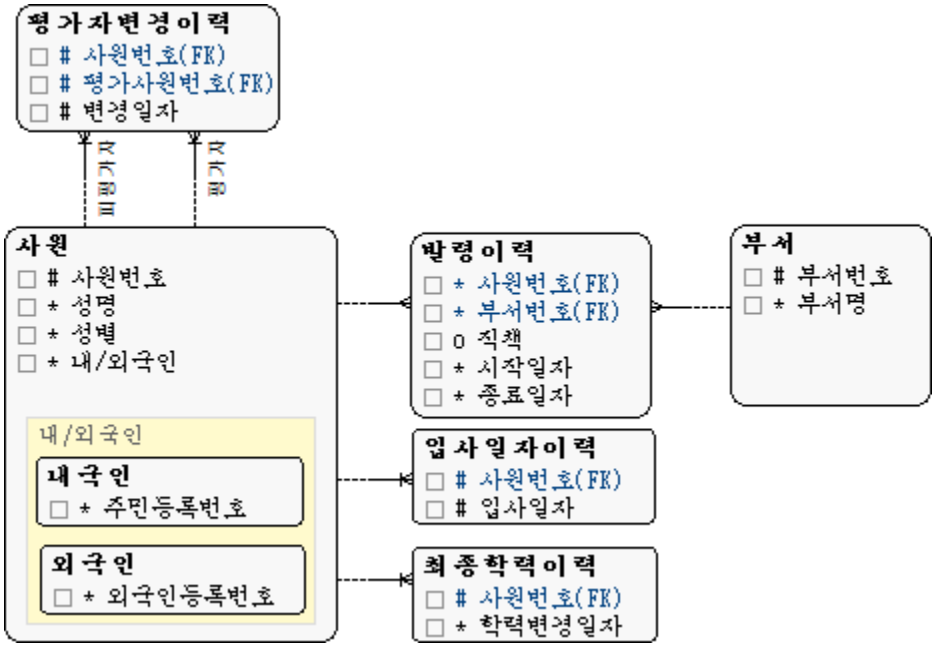
마지막으로 사원-사원 관계에 대한 이력을 관리해보자. 사원간의 관계는 '평가자-피평가자'관계다. 만약 이 관계가 시간에 따라 변화하고, 이를 관리하고자 한다면 다음과 같이 변경되어야 한다.

사원번호	평가사원번호	변경일자
1	2	2009-02-01
3	2	2010-12-01
3	1	2011-01-01

사원번호	성명	성별	외국인등록번호	주민등록번호	입사일자	최종학력
1	이재학	남		761218-*******	2007-11-12	대졸
2	홍길동	남		740114-*******	2009-02-01	대학원졸
3	장발장	남	770717-1500009		2010-12-01	대졸

<그림 2.5.10>

이 관계는 시작~종료 형태로 관리하지 않아도 문제가 없다. 변경된 데이터 모델은 다음과 같다.



<그림 2.5.11>

이력관리를 시작~종료 형태의 범위로 관리 하는지 아니면 변화가 발생한 시점만을 관리하는지는 현실의 업무와 속성의 도메인을 보면 판단할 수 있다. 일반적으로 시작~종료로 관리하는 것은 SQL의 성능문제 때문인데, 이는 미신이다. 이에 대한 논의는 3장에서 할 것이다.

종에 대한 이력관리

종에 속성이 삭제/추가되거나 관계가 삭제/추가되는 경우 또는 의미가 확장되는 경우 종에 대한 이력관리를 한다. 종의 이력관리는 누가, 언제, 왜, 무엇을, 어떻게 변경했는지 기록, 관리, 추적, 감리하는 일이다. 이는 데이터 모델의 형상 관리라고도 볼 수 있는데, 종에 대한 이력관리는 데이터 품질 관리의 데이터 구조 관리 영역이다. 또한 메타 데이터 관리와도 연관이 있다. 이 영역은 책에서 이야기하고자 하는 범위를 벗어난다. 그러므로 다른 서적을 참고하여 학습하기 바란다.

2.6 릴레이션쉽 유형

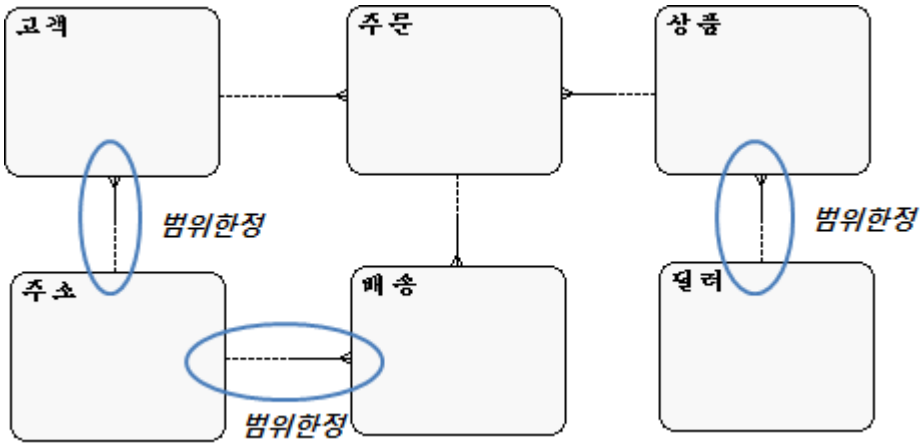
릴레이션쉽은 다음과 같이 3가지 유형으로 나눌 수 있다.

- 범위한정: 개체에 대한 개념의 범위 한정(역학과 책임, 코드, 단 방향 참조)
- 상호작용: 개체간의 상호작용 결과(양방향 참조)
- 존재종속: 개체의 탄생의 순서(부모자식, 순서쌍)

좀 더 찾아보면 세부적인 유형이 나올 수 있을 것이지만, 반드시 이 3가지 경우에 속할 것이다. 각각을 더 자세히 살펴보자.

범위한정

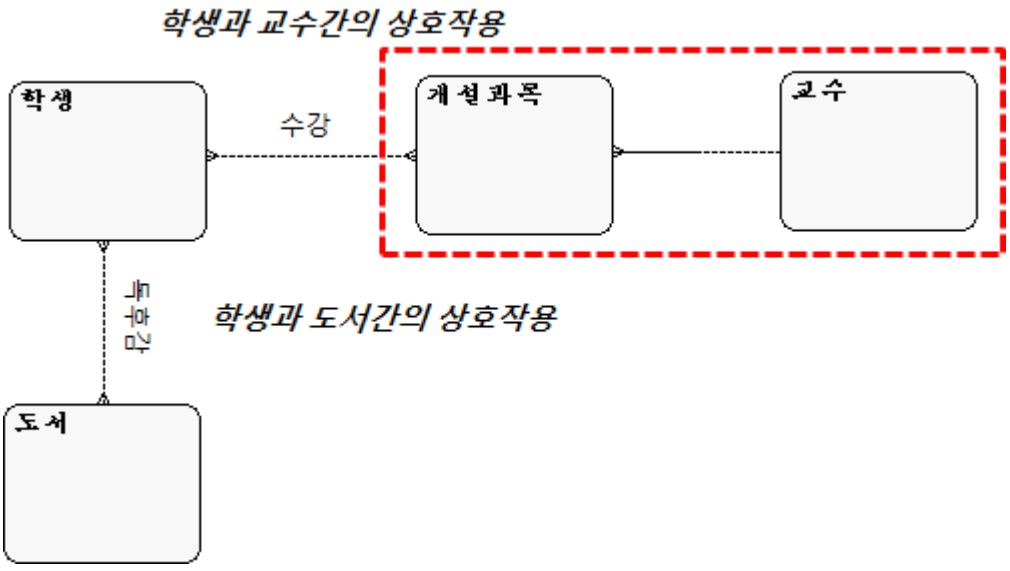
이 유형은 행위의 주체에 해당하는 데이터에 대한 역할이나 개념의 범위 등을 한정 짓는 릴레이션쉽을 말한다. 앞서 살펴본 사원-부서의 릴레이션쉽이 이 유형에 속한다. 사원-부서의 릴레이션쉽이 다:다 관계로 풀렸다면 상호작용 유형로 착각할 수 있으나, 부서는 사원의 R&R을 한정하고 있다. 즉, 부서의 업무 범위가 사원이 할 수 있는 행위를 제한하는 것이다. 오픈 마켓 모델로 예를 들면 고객-주소, 주소-배송, 딜러-상품 릴레이션쉽이 이 유형에 해당한다.



<그림 2.6.1>

상호작용

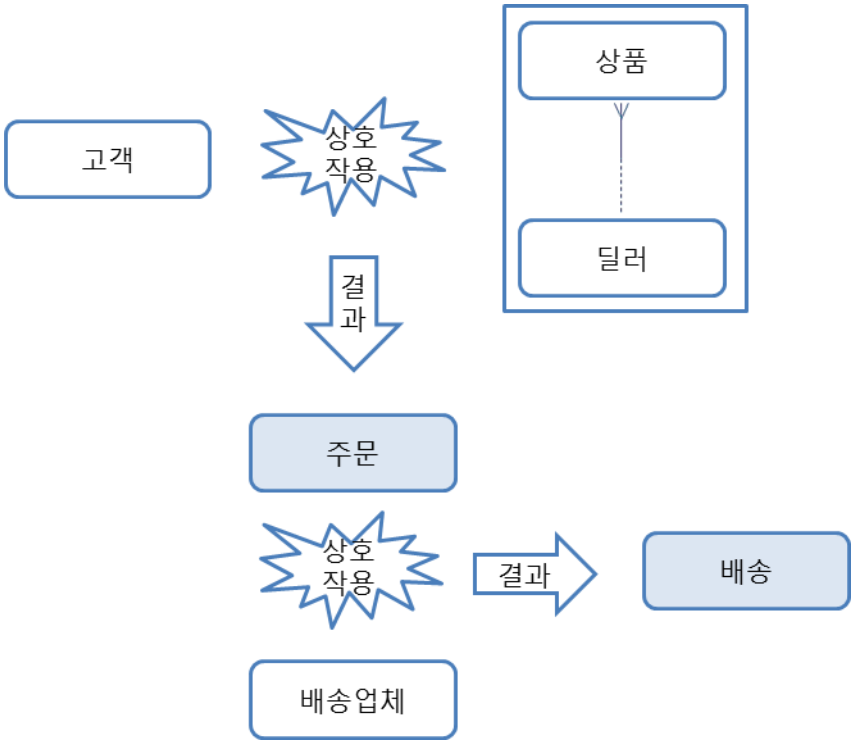
이 유형은 행위의 주체가 되는 개체들이 상호작용하는 릴레이션쉽을 말한다. 다음의 그림을 보자.



<그림 2.6.2>

학생은 교수로부터 강의를 받고, 교수는 학생에게 강의할 것이다. 이런 교수와 학생의 상호작용으로 학생은 학점을 취득할 것이며, 교수는 강의라는 역할을 수행할 수 있게 된다. 즉, 학생-교수간의 상호작용으로 기존에 없던 것이 새롭게 만들어진다. 새롭게 만들어진 것이 '수강'이며, 이는 1장에서 설명했던 '시너지'다. 학생-도서의 관계도 상호작용 관계다. 학생이 도서를 읽어야만 독후감이 만들어 질 수 있다.

상호작용 릴레이션쉽은 많은 경우 카디널리티가 다:다 인데, 1:1 또는 1:다 릴레이션쉽 모두 상호작용 유형 될 수 있으므로 세심한 관찰이 필요하다. 예를 들어, 여러 명의 학생이 지도를 받고 각각의 교수가 여러 학생을 지도한다면 교수:학생 = 1:다 릴레이션쉽이 된다. 또 하나 주의해야 할 사항은 상호작용이 1개의 릴레이션쉽으로만 끝나는 것이 아니라는 것이다. 오픈 마켓을 보면 고객은 자신이 필요로 하는 상품을 구매하고, 딜러는 상품을 판매하는 상호작용을 한다. 상호작용의 결과는 '주문'이다. 배송업체는 고객과 딜러의 상호작용의 결과인 '주문과' 또 다른 상호작용으로 '배송'을 하게 된다.

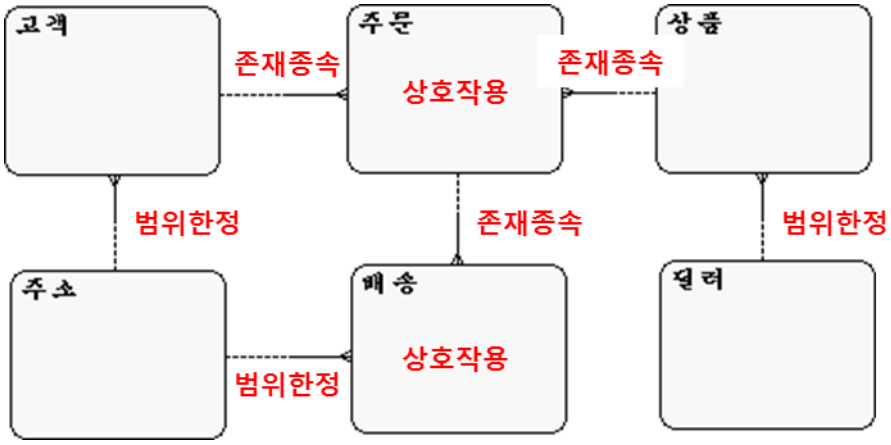


<그림 2.6.3>

존재종속

아빠와 엄마는 부부 관계다. 아빠의 정자와 엄마의 난자가 결합하여 자식이 태어난다. 여기서 아빠와 엄마의 관계는 위에서 설명한 상호작용 유형이다. 아빠-자식 또는 엄마-자식의 관계가 존재종속유형이다.

위의 그림에서 주문-배송 간의 관계에 주목하자. 배송은 반드시 주문이 먼저 있어야만 만들어 질 수 있다. 즉, 배송은 주문에 존재 종속이다. 오픈마켓 모델에서 릴레이션쉽의 유형을 정리해보면 다음과 같다.



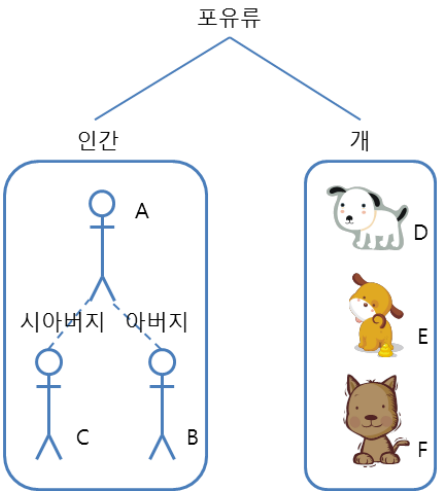
<그림 2.6.4>

2.7 계층, 유형, 카테고리

계층, 유형, 카테고리는 분류를 위해 필요하다. 1장의 형이상학적 실재론에서 우리는 '유사함'에 대해 살펴보았듯이 철학적인 관점에서 이들을 살펴보자.

계층, 유형, 카테고리의 개념

다음의 그림으로 설명될 수 있다.



<그림 2.7.1>

계층은 개체들 간의 조직화가 이루어진 모습(인간 A, B, C)을 연상할 수도 있고, <그림 2.7.1>에서 보이는 전체적인 위계가 있는 구조를 연상할 수도 있다. 계층은 이 두 가지를 다 말하는데, 주의할 점이 있다. <그림 2.7.1>에서 인간 A는 C, B와 관계를 가지며, 비대칭 관계로 위계가 있다. 하지만 인간 A, B, C가 '친구'라는 관계라면 대칭 관계로 위계가 없으므로 계층이 아니게 된다. '인간'과 '개'는 개체에 대한 종이며, '포유류'는 종에 대한 유형이다. 카테고리는 최상위 유형을 말한다. 여기서는 '포유류'가 카테고리다. 이를 릴레이션 구조로 변경해 보면 다음과 같다.

개체

개체번호	개체명	카테고리키
1	A	2
2	B	2
3	C	2
4	D	3
5	E	3
6	F	3

카테고리

카테고리키	계층	구분	상위계층
1	포유류	유형	
2	인간	종	1
3	개	종	1

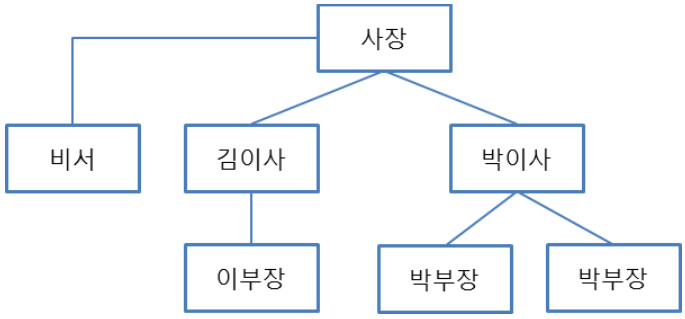
계층

개체번호	상위개체번호
2	1
3	1

<그림 2.7.2>

계층구조

개체는 개체들 간에 계층을 이룰 수 있다. 흔히 볼 수 있는 형태는 아래의 그림과 같은 사원들간의 계층구조다.



<그림 2.7.3>

물론 부서나 팀과 같은 회사의 조직구조도 이와 마찬가지로 계층이다. 이런 형태는 균형을 이룬 형태가 아니며, 변할 수 있는 여지가 많기 때문에 유연한 구조로 설계되어야 한다. 예전(물론 지금도)에는 다음과 같이 '릴레이션의 어트리뷰트는 단일값이어야 한다'는 릴레이션의 특성을 어긴 형태로 많이 설계되었다.

사원번호	계층코드	사원명	직급
1	A	이재학	사장
2	A2A1	김개똥	이사
3	A2	박말숙	이사
4	A2A11	박정희	부장
5	A2A12	박길동	부장
6	A2A11	이순신	부장
7	A3	최충헌	비서

<그림 2.7.4>

계층코드 'A11'은 단순히 '이순신' 사원의 식별자가 아니다. 'A11'에는 자신의 상위 관리자가 'A1', 'A'라는 의미가 내포되어 있다. 즉, 2가 이상의 의미를 가진다. 이와 같은 형태는 다음과 같은 SQL로 조회해 볼 수 있다.

```
--drop table 사원
createtable사원
(
사원번호intprimarykey
,계층코드varchar(20)
,사원명varchar(20)
,직급varchar(20)
);
go

insertinto사원(사원번호,계층코드,사원명,직급)
select 1,'A','이재학','사장'unionall
select 2,'A1','김개똥','이사'unionall
select 3,'A2','박말숙','이사'unionall
select 4,'A21','박정희','부장'unionall
select 5,'A22','박길동','부장'unionall
select 6,'A11','이순신','부장'unionall
select 7,'A3','최충헌','비서'
```

```
go

select
  replicate('        ',len(계층코드)-1)+사원명+'('+직급+')'계층구조
from사원
orderby계층코드

/*
계층구조
------------------------------
이재학(사장)
김개똥(이사)
이순신(부장)
박말숙(이사)
박정희(부장)
박길동(부장)
최충헌(비서)
*/
```

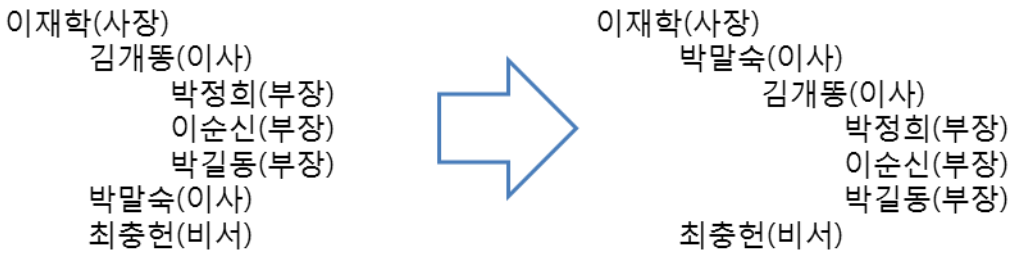
만약 박정희, 박길동 두 부장을 김개똥 이사 밑으로 인사발령을 한다면 다음과 같은 update 구문이 필요할 것이다.

```
update사원
set계층코드=replace(계층코드,'A2','A1')
where계층코드like'A2%'
and계층코드<>'A2'

select
replicate('        ',len(계층코드)-1)+사원명+'('+직급+')'계층구조
from사원
orderby계층코드

/*
계층구조
------------------------------
이재학(사장)
김개똥(이사)
```

```
박정희(부장)
이순신(부장)
박길동(부장)
박말숙(이사)
최충헌(비서)
*/
```

이 패턴은단순하지만 매우 견고하여 변화에 취약하다는 단점을 가지게 되는데, 이는 앞서 언급한 어트리뷰트는 단일값을 가져야 한다는 기본 사항을 지키지 않아 나타나는 문제다. 어트리뷰트가 단일값이 아니어서 최종적으로 우리가 겪게되는 현상은 '갱신 이상(anomaly)' 이다.

갱신 이상(anomaly)의 발생은 부분적이다. 최하위 계층에서는 갱신 이상이 발생되지 않는다. 하지만, 계층 구조 상에서 Leaf Node가 아닌 모든 행은 갱신시 이상현상을 가져오게 된다. 그림으로 도식해보면 다음과 같다.



<그림 2.7.5>

위의 사원 테이블에서'김개똥 이사'가 '박말숙 이사' 하위에 있게 된다면 '김개똥 이사'뿐만 아니라 계층구조상의 '박정희', '이순신', '박길동' 부장들의 사원번호도 모두 갱신되어야 한다.

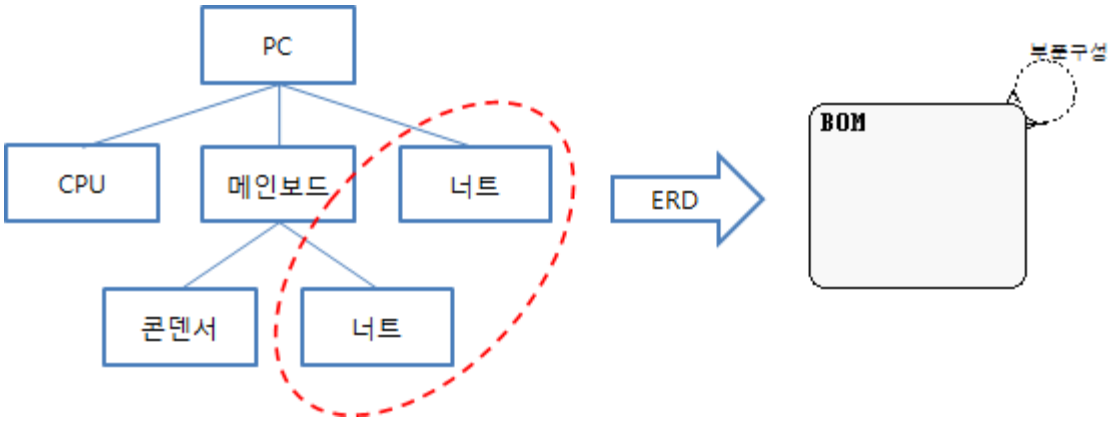
```
update사원
set계층코드='A2'+계층코드
where계층코드like'A1%'

/*
계층코드계층구조
------------------- ------------------------------------------
A                   이재학(사장)
A2                        박말숙(이사)
A2A1                                김개똥(이사)
A2A11                                     박정희(부장)
A2A11                                     이순신(부장)
```

```
A2A12                                   박길동(부장)
A3                          최충헌(비서)
*/
```

갱신 이상이 발생함에도 이런 패턴이 자주 쓰이는 이유는 예전에 DBMS가 재귀 쿼리를 지원하지 않았을 때의 방법만을 알고 있다거나 단순하기 때문이다. 현재의 대부분의 DBMS들은 재귀쿼리를 지원하고, 관리되어야 하는 데이터가 많아짐에 따라 이런 단순한 구조로는 요구사항을 만족시키기 어려워졌다. 예를 들어, 위의 패턴은 BOM(Bill-Of-Material) 솔루션으로는 적합하지 않다. 그래서 요즘에는 순환 릴레이션쉽 패턴으로 계층 구조를 표현한다.

BOM(Bill-Of-Material)

BOM은 A라는 부품이 어떤 부품들로 구성되는가에 대한 데이터다. 생산의 관점에서 정의를 내렸지만, '부품'이 '기능' 또는 '서비스'와 같은 단어로 대체된다면 설계 관점의 BOM이 된다. 아래의 <그림 2.7.5>를 보자.



<그림 2.7.6>

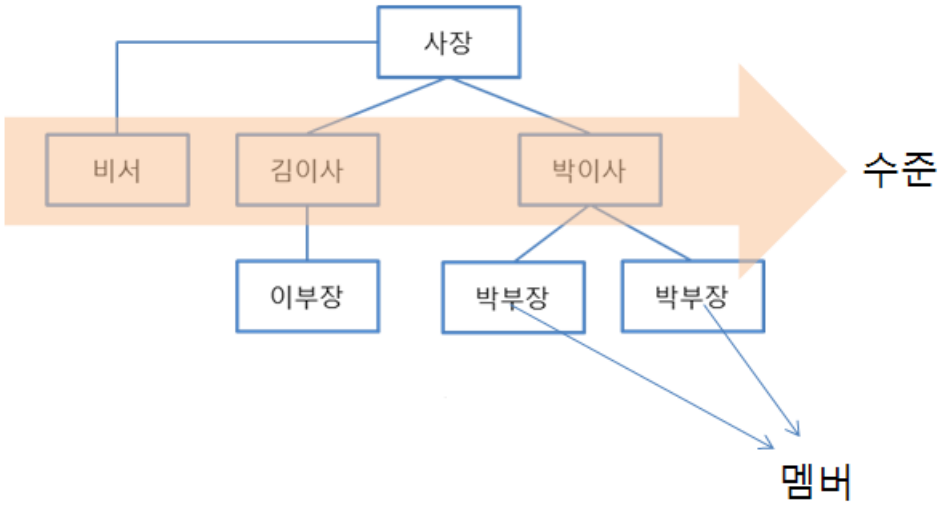
PC는 CPU, 메인보드, 너트로 구성된다. 메인보드는 콘덴서, 너트로 구성된다. 점선으로 표시한 것처럼 '너트'는 여러 부품을 구성하고 있다.

실제로 다양한 BOM이 존재한다. 어떤 제품이 모여 제품 군을 이룬다던지, SOA(service oriented architecture)에서 여러 서비스가 조합되어 또 다른 서비스를 구성하는 것도 일종의 BOM이라고 볼 수 있다. 기본적으로는 <그림 2.7.5>의 ERD와 같은 형태로 다:다 순환관계로 표현되며, 추가적은 정보들이 군더더기로 붙는다.

다양한 계층구조

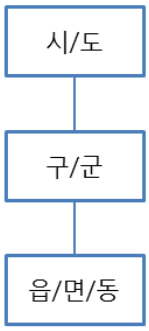
계층구조를 이해하려면 몇 가지 용어를 알아야 한다. 수준은 위계의 범위를 말한다. <그림 2.7.7>

에서 비서, 김이사, 박이사는 이사 수준으로 같은 위상을 가진다. 멤버는 각각의 수준에 있는 계층구조를 구성하는 개체를 말한다.

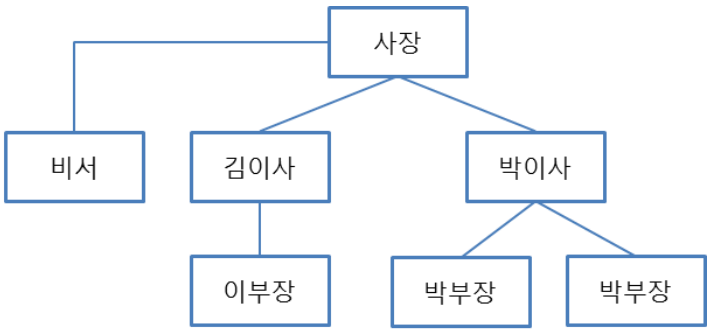


<그림 2.7.7>

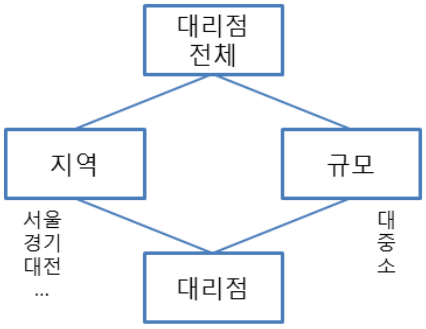
계층구조는 <그림 2.7.8>과 같이 4가지 정도로 분류된다.



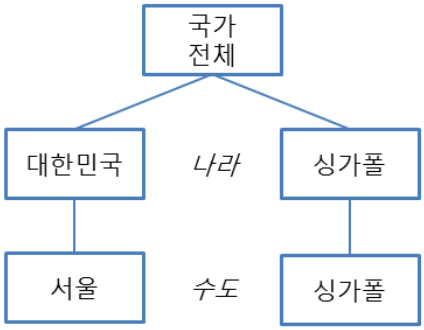
<일반 계층구조>



<불균형 계층구조>



<다중 계층구조>

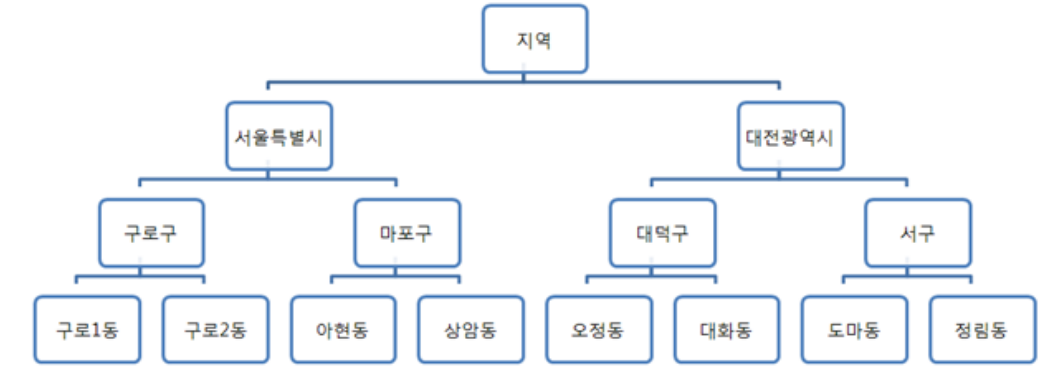


<비정형 계층구조>

<그림 2.7.8>

일반(=균형) 계층구조는 모든 분기가 동일한 수준이다. 동일한 수준의 멤버들의 논리적인 부모 멤버들 역시 동일한 수준의 멤버다. 불균형 계층구조는 계층구조의 분기가 다른 수준이다. <그림 2.7.7>에서 보는 바와 같이 비서의 경우는 자식 멤버가 없다. 다중 계층 구조는 자식 멤버가 2개 이상의 부모 멤버를 가지는 경우를 말한다. [대리점전체-지역-대리점] 계층구조와 [대리점전체-규모-대리점] 계층구조가 혼합된 형태다. 비정형 계층구조는 멤버들의 수준이 일정하지 않은 경우를 말한다.

여기서 OLAP의 계층구조를 간단히 살펴보았지만, 실제로 계층구조를 모델링 하는 방법은 순환형태가 대부분이다. 수준이 고정적이라면 수평적인 형태를 가질 수도 있다. <그림 2.7.8>은 계층구조를 모델링 하는 2가지 일반적인 방법이다.



방법1

지역키	시/도	구/군	읍면동
1	서울특별시	구로구	구로동
2	대전광역시	유성구	궁동
...	...	...	...

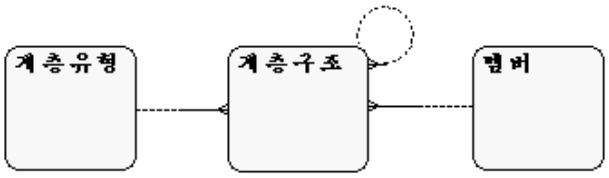
방법2

지역키	지역명	부모지역키
1	지역	
2	서울특별시	1
3	구로구	2
4	구로동	3
5	대전광역시	1
6	유성구	5
7	궁동	6
...	...	...

<그림 2.7.9>

일반 계층구조는 방법1, 방법2 모두 가능하다. 하지만, 수준의 수가 많고 변화가 자주 일어난다면 좀 더 유연한 구조인 방법2를 선택하는 것이 좋다. 이는 불균형 계층구조에도 해당되는 말이다. 불균형 계층구조는 일반적으로 방법2가 많이 선택된다.

다중 계층구조라면 계층구조를 표현하는 데이터와 어떤 계층구조인지를 나타내는 데이터가 필요하다. 그러므로 다음과 같이 모델링 된다.



계층유형

계층유형키	계층유형명
1	지역
2	규모

계층

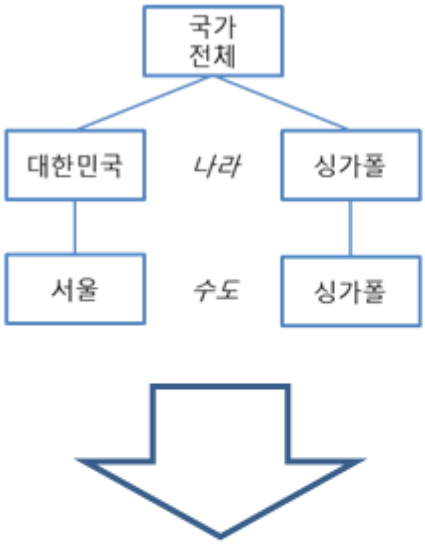
계층유형키	멤버키	상위멤버키
1	1	
1	2	1
1	3	1
1	4	2
1	5	2
1	6	3
2	1	
2	7	1
2	8	1
2	9	1
2	4	8
2	5	9
2	6	7

멤버

멤버키	멤버명
1	대리점전체
2	대전
3	대구
4	서부지점
5	동부지점
6	대구지점
7	대
8	중
9	소

<그림 2.7.10>

비정형 계층구조는 수준을 반드시 명시해야만 한다. 그래야만 데이터의 구조가 명확해진다. 다음의 <그림 2.7.11>은 비정형 계층구조를 나타낸다.



멤버

멤버키	멤버명
1	국가전체
2	대한민국
3	서울
4	싱가폴

계층

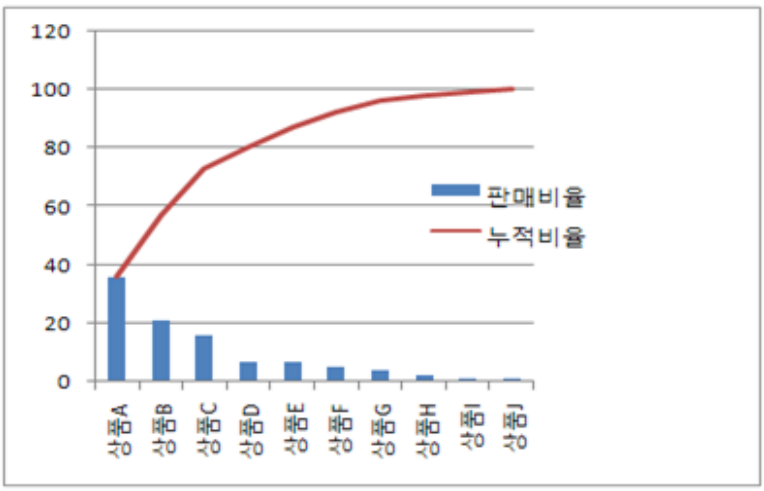
계층키	수준	부모계층키	멤버키
1	전체		1
2	나라	1	2
3	나라	1	4
4	수도	2	3
5	수도	3	4

<그림 2.7.11>

범주적 폭력

범주적 폭력이란, 어떤 존재와는 다른 이질적인 존재들을 어떤 하나의 범주로 묶어 일반화하는 것을 말한다. 다음의 <그림 2.7.12>을 보자.

상품명	판매비율	누적비율
상품A	36	36
상품B	21	57
상품C	16	73
상품D	7	80
상품E	7	87
상품F	5	92
상품G	4	96
상품H	2	98
상품I	1	99
상품J	1	100

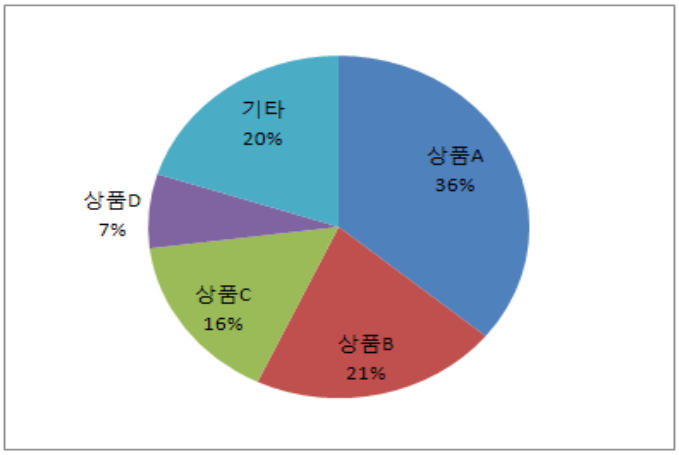


<그림 2.7.12>

다음과 같은 가정을 해보자.

상품A ~ 상품D까지의 누적비율이 80%다. 그렇다고 상품A ~ 상품D가 아닌 다른 상품들의 매출을 성장/유지 시키는 데에 대한 노력이 상품A ~ 상품D에 비해 적은 것이 아니다. 매출 규모를 유지 시키기 위하여 상품A ~ 상품D 이외의 상품들도 중요하다는 것을 경영진도 알고 있다.

자, 이제부터 상품E ~ 상품J를 '기타'로 묶어 다음과 같은 파이 차트를 그려 경영진에게 보고한다고 해보자.



<그림 2.7.13>

위 차트에서의 비율이 1년 동안 거의 변화하지 않았고, 이 형태로 계속 경영진에게 보고했다면 '기타'에 해당되는 상품들은 어떻게 변화했는지를 알 수 없게 된다. 만약 상품H가 잘 팔려서 전체 판매의 10%를 차지했다면 상품D는 '기타'로 분류되고, 상품H가 '기타'에서 빠져 나왔을 것이다.

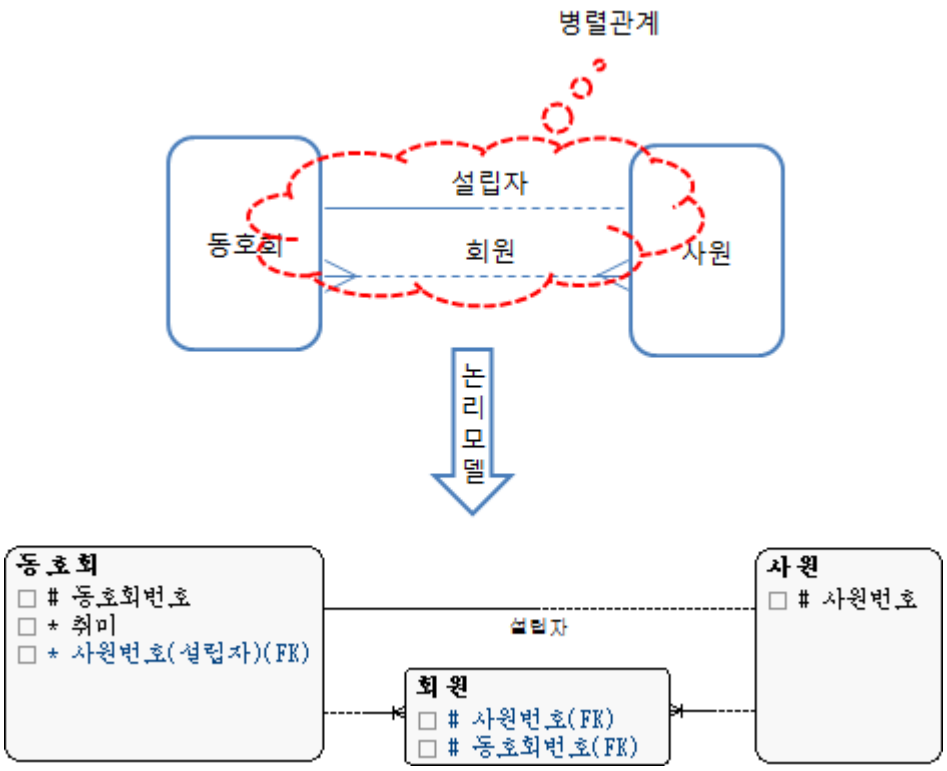
물론, 상품 판매에 대한 분석을 했을 테고, '기타'로 분류된 상품들의 변화를 파악했을 것이다. 문제는 이것이다. 데이터베이스에서 '범주적 폭력'을 행사했을 때에는 반드시 분석이 뒤따라야만 한다는 것이다. 폭력은 반드시 대가를 치러야 함을 기억해야 한다.

2.8 병렬 관계

개체와 개체간에는 다수의 관계가 존재할 수 있다. 부모와 자식 관계일 수도 있고, 고용자와 피고용자의 관계일 수도 있다. 이러한 여러 관계를 '병렬 관계'라고 하자. 2.8절에서는 병렬 관계에 대해서 논리 모델을 통해 알아보도록 하겠다.

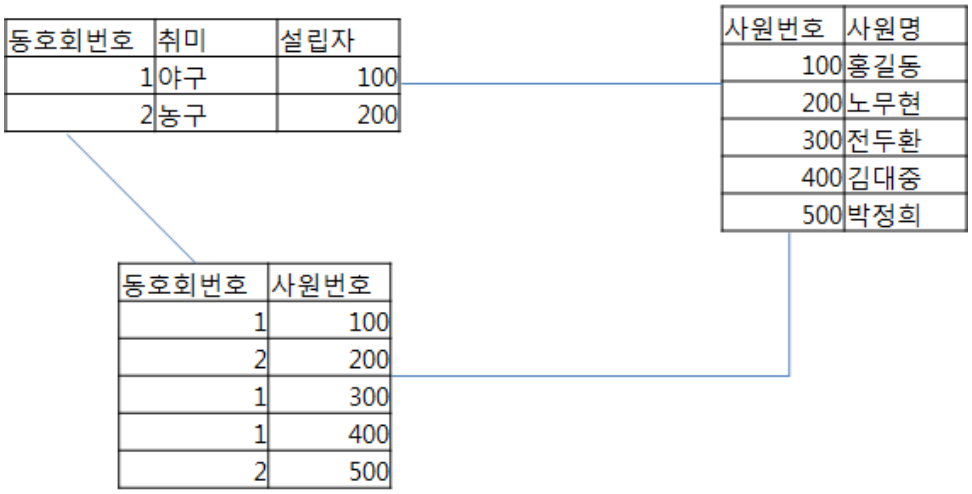
사내 동호회

사내 동호회는 같은 취미를 가진 사원(사람)들의 모임이다. 사내 동호회는 사원이 없으면 존재 할 이유가 없는 논리적인 개체다. 동호회와 사원간에는 2개의 관계 존재한다. 하나는 "설립자"라는 관계이고, 또 다른 하나는 "동호회원"이다. 이를 논리 모델로 표현한 것이 <그림 2.8.1>이다.



<그림 2.8.1>

동호회 릴레이션과 사원 릴레이션 사이에 하나의 릴레이션쉽과 동호회와 사원간의 릴레이션쉽을 관리하게 될 릴레이션인 회원 릴레이션으로 만들어진다. 이와 같이 2개의 릴레이션에서 2개 이상의 관계가 식별될 때 이를 '병렬 관계'라고 말한다. 실제 인스턴스 다이어그램을 보자.



<그림 2.8.2>

사원 릴레이션의 투플은 퇴사시 삭제된다. 이런 경우에 어떻게 할 것인가?

2.9

2.10

2.11

2.12

2.13 코드설계

코드설계는 굉장히 어려운 문제다. 코드설계를 어떻게 하느냐에 따라 시스템의 복잡성이 달라진다. 특히 정보계 시스템에서 코드에 따른 변화 문제는 하나의 Chapter를 모두 할당할 만큼 어려운 주제다. 이 절에서는 다음의 내용을 다룰 것이다.

코드의 존재이유

코드는 다음의 2가지 이유로 사용된다.

- 데이터 압축
- 데이터 식별

다음의 코드 테이블을 보자.

코드	코드명
1	키보드
2	마우스
3	모니터

1은 '키보드'를 의미한다. 1은 1바이트이며, '키보드'는 6바이트다. 그러므로 6바이트 문자열 대신 1바이트의 숫자로 대신한다면 무려 6배의 압축효과가 있다. 숫자이므로 문자열보다는 연산도 간단하다. 만약 '키보드'가 'keyboard'라고 변경되었을 경우 '키보드'를 update하거나 다음과 같이 새로운 코드를 발급하여 변경에 대한 이력도 관리 할 수 있다.

코드	코드명	변경전코드
1	키보드	
2	마우스	
3	모니터	
4	keyboard	1

이런 경우 '키보드'와 'keyboard'는 다른 것으로 인식한다는 의미도 된다. 만약 이 둘을 같은 것으로 인식하고자 한다면 코드 1과 4가 같음을 모델에 표현해야 한다. 다음과 같은 방법을 생각해 보자.

코드	코드명	변경전코드	통합코드
1	키보드		1
2	마우스		2
3	모니터		3
4	keyboard	1	1

통합코드	통합코드명
1	키보드
2	마우스
3	모니터

Packed Data Type(oo코드)

Packed Data Type이란, 일반적인 프로그래밍의 관점에서 본다면 '묶인 데이터' 쯤으로 보면 되고, 데이터베이스의 도메인에서 이야기 하자면 '복합 어트리뷰트'쯤이 된다. 복합 어트리뷰트는어트리뷰트의 값이 원자값이 아닌 여러값을 가질 때 불려지는 용어다. 예를 들면, 주소, 주민번호, 학번과 같은 어트리뷰트들이다. 이들은 다음과 같이 쪼개질 수 있다.

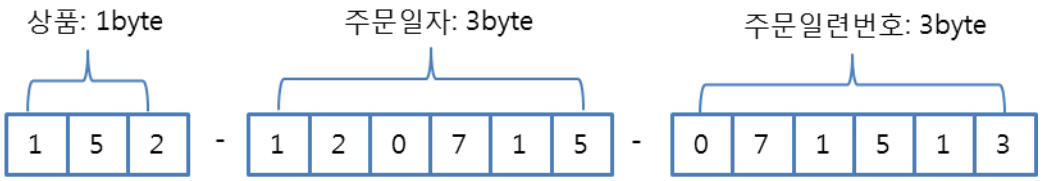
- 주소: 시/도, 구/군, 읍/면/동
- 주민등록번호: 생년월일, 성별, 행정기관코드, 일련번호, 가중치코드
- 학번: 입학년도, 학과번호, 일련번호

원자값은 해당 조직 범위의 전체적인 관점에서 볼 때에 원자값이어야 한다. 주의해야 할 점은 이렇게 쪼갠 값이 원자값이 아닐 수 있다는 것이다. 원자값은 값이 더 이상 쪼개어 질 수 없는 것이 아니라 '의미의 최소 단위'를 말한다. '주소'의 경우 해당 조직에서 단지'우편물을 보내기 위한 체계' 정도로필요하다면 '시/도', '구/군'등의 쪼개어진 값은 원자값이 아니다. 왜냐하면 우편물을 보내기 위해서는 '시/도', '구/군', '읍/면/동', '번지 상세주소'가 모두 포함된 것이 최소단위이기 때문이다. 만약 마케팅팀에서 '고객에 대한 지역별로 세분화하여 특징을 파악한 뒤 지역마다 차별화된 마케팅 전략을 세운다'는 요구가 있다면 '지역별'이 어느 단위까지인지가 원자값을 정의하는 기준이 될 것이다.

데이터베이스는 최소한의 중복을 지향한다. 하지만 아직까지도 중복된 속성을 Primary Key로 사용하는 곳이 많이 있다. 예를 들면 '상품번호 + 주문일 + 일련번호'와 같은 주문번호가 그것이다. 문자열로 표현하면 '152-120715-071513'과 같은 형태일 것이다. 문자열로 15Btye를 사용해야 한

다. 그래서 다음과 같은 조건에서 주문번호 코드체계를 만들었다고 가정해 보자.

- 상품코드: 200 Row
- 주문일자: YYMMDD
- 재고수량: 999999
- 주문건수: 50 만 건



날짜의 경우 bit단위로 연산을 하면 더 줄일 수 있다.DD(5bit), MM(4bit), YY(7bit) = 15bit 이므로 16bit를 사용한다고 하면 2byte면 된다.

주문번호를6Byte로 만들면 주문 테이블에 어떤 상품이 언제 또는 어떤 기간에 몇 개나 팔렸는지 주문번호 1개로 모두 알아낼 수 있다. 그러므로 주문테이블의 상품번호, 주문일자 컬럼은 삭제해도 되지 않을까? 뭔가 복잡해졌지만, 상품번호, 주문일자 컬럼을 삭제하였으니 테이블이 단순해졌으므로 코드 체계의 복잡성을 상쇄시켰다고 생각할 수도 있다.

이것이 여러분이 복잡하게 여러 의미를 내포하고 있는 '주문코드'나 '부서코드'의 역사적 배경이다. 어떤 경우는 주문코드만 봐도 어떤 매장에서 어떤 상품이 주문되었는지 알 수 있으므로 매우 의미있다고 할 수 있을 것이다. 하지만, 시간이 지날수록 원인도 모른채 주문 테이블에 쌓여가는 Row수 만큼 사용자들의 불만도 쌓일 것이다. 왜냐하면 이는 하드웨어가 매우 빈약하던 시절의 파일 처리 방식이기 때문이다. 예전의 방식으로 RDBMS를 쓴 다면 성능을 보장받을 수 없다.

이러한 Packed Data Type을 데이터베이스에서 사용할 때의 가장 큰 문제는 '원자값' 문제다. 위 주문번호의 예로 2012년 07월 15일의 상품코드별 주문건수를 알고자 한다면 다음과 같은 SQL문을 사용해야 할 것이다.

```
select
```

```
right(replace(str(convert(tinyint,substring(주문번호, 1, 1))),' ','0'),3)상품코드

,count(*)주문건수

from (

select

convert(binary(1), 152)+

convert(binary(1), 12)+

convert(binary(1), 07)+

convert(binary(1), 15)+

convert(binary(3), 071513)주문번호

)주문테이블

where 1=1

and right(replace(str(convert(tinyint,substring(주문번호, 2, 1))),' ','0'),2)+

right(replace(str(convert(tinyint,substring(주문번호, 3, 1))),' ','0'),2)+

right(replace(str(convert(tinyint,substring(주문번호, 4, 1))),' ','0'),2)='120715'

groupby

right(replace(str(convert(tinyint,substring(주문번호, 1, 1))),' ','0'),3)
```

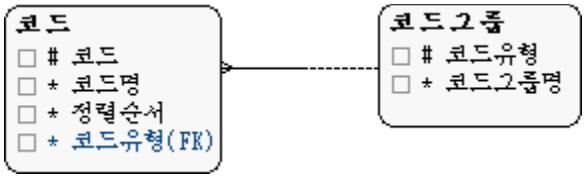
DBMS는 이런 유형의 SQL문이 날아올 때마다 5천 만 건의 주문테이블을 풀스캔 할 것이다. 개발자는 특정일의 특정 주문 1건 조회하는데 몇분이 걸리냐면서 DBMS를 탓하고, 사용자는 울분을 토하고 있을 것이다. 데이터베이스의 특성을 이해하지 못하고, 일반적인 어플리케이션의 개발 방법을 데이터베이스에 그대로 반영한 탓일 것이다. 하지만 때로는 Packed Data Type이 데이터베이스에서도 아주 유용하게 쓰일 경우가 있다. 예를 들면, IP와 같이 DBMS에서 지원되지 않는 데이터 형식일 것이다. 대부분 문자열로 'xxx.xxx.xxx.xxx'형식으로 표현하는데, IP 1개만 조회할 경우는 괜찮으나 범위를 검색하는 경우는 아주 진상을 떨게 된다. 또 다른 예로 DW에서 골치 아픈 주제인 Surrogate Key를 이런 형태로 사용하면 유용하다.

Packed Data Type이 데이터베이스에서는 부하만 유발하는 쓸모 없는 것은 아니다. 앞서 이야기

한 것과 같은 특정 상황에서는 Packed Data Type 형태로 사용하면 쏠쏠한 재미를 볼 수 있다. 아이러니 한 것은 인간들이 단순한 일련번호보다 복잡한 코드를 더 잘 외우는 상황이다. 그러므로 아예 Packed Data Type을 사용하지 않을 수 없고 충분한 검토를 거친 후에 사용을 결정해야 한다.

코드통합

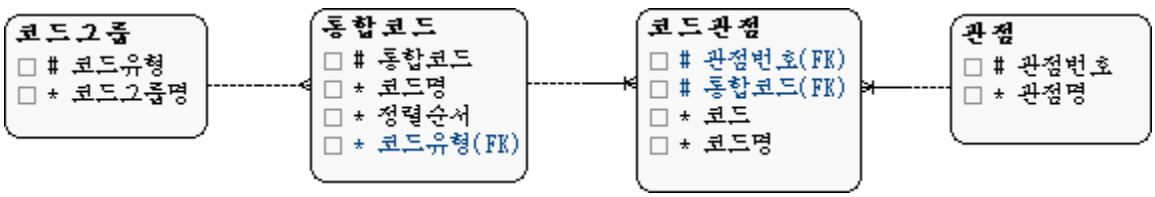
코드통합은 Context에 따라서 의미가 달라질 수 있다. 일반적인 DI(Data Integration)에서의 코드통합은 'Single-view'를 의미한다. 하지만, MDM(Master Data Management)에서의 코드통합은 조금 다른 의미를 가진다. MDM에서는 모든 관점을 존중하는 것을 기본으로 한다. 그러므로 Single-view도 물론 포함하고, 여러 사용자(부서, 개인, 시스템 등)의 관점을 포괄하는 것을 기본으로 한다. 일반적으로 통합된 코드는 기본적으로 다음과 같은 모양을 가진다.

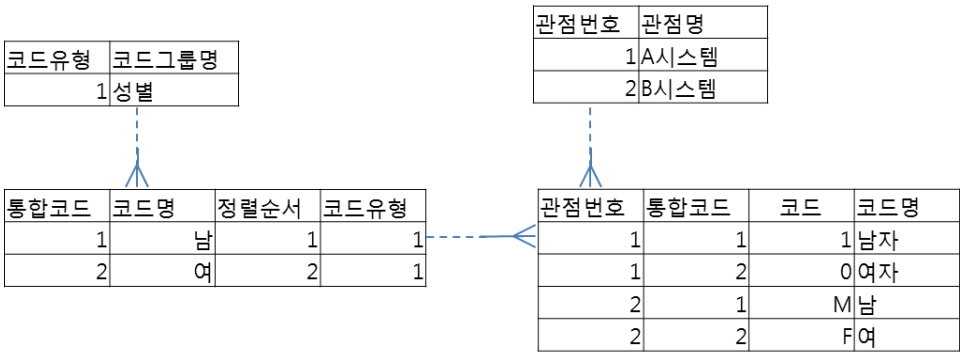


코드	코드명	정렬순서	코드그룹
1	남	1	1
2	여	2	1
3	생존	1	2
4	휴면	2	2
5	이탈	3	2

코드그룹	코드그룹명
1	성별
2	고객상태

만약 '성별' 코드가 A시스템과 B시스템이 서로 상이한 경우라면 DI의 경우 이를 하나의 관점으로 통합할 것이다. MDM 관점이라면 통합된 관점을 포함하여 모든 관점을 존중할 것이다. MDM의 관점이 DI 관점을 포함하고 있으므로 다음과 같이 모델을 변경할 수 있을 것이다.





코드유형	코드그룹명
1	성별

관점번호	관점명
1	A시스템
2	B시스템

통합코드	코드명	정렬순서	코드유형
1	남	1	1
2	여	2	1

관점번호	통합코드	코드	코드명
1	1	1	남자
1	2	0	여자
2	1	M	남
2	2	F	여

코드 통합의 장점은 코드와 관련된 테이블 몇 개만 알면 시스템 전체의 코드를 알 수 있다는 것이다. 그러므로 개발이 쉽다. 또한 중앙집중식이므로 코드의 관리가 쉽고, Single-View를 제공하므로 데이터의 일관성을 확보할 수 있다. (Single-View는 관리하기 나름이다)

코드 통합의 단점은 통합된 코드를 사용하는 경우 코드가 한 곳에만 존재하기 때문에 코드의 유실이 있는 경우 해당 코드를 사용하는 어플리케이션들이 모두 동작하지 않을 수 있다는 것과 예제에서와 같이 bit형으로 표현할 수 있는 성별을 숫자형이나 또는 문자열로 표현되기 때문에 도메인일 넓어질 수 있다는 단점이 있다. 도메인이 넓어진다는 의미는 2가지를 말한다. 데이터 타입이 커짐에 따라 서버의 자원 사용량이 많아져 전체적인 성능저하를 가져오는 것과 데이터 품질 저하의 가능을 내포하는 것이다. 또 다른 성능관점에서 코드가 통합되기 때문에 데이터가 많아져 참조무결성이나 조회 시에 더 많은 서버 자원을 사용해야 하는 단점이 있다.

가장 이상적인 데이터 모델은 전사적인 관점을 유지하여 최대한 중복이 없는 정규화된 모델이다. 필자의 견해로는 이렇게 통합된 코드 테이블이 존재하는 이유는 제대로 된 설계도가 없었기 때문이라 생각된다. 설계도만 있다면 어떤 코드가 무엇인지에 대한 내용을 애써서 찾을 필요가 없다. 이런 설계패턴은 릴레이션형 데이터베이스를 제대로 이해하지 못해 생겨난 것으로 지양할 필요가 있다.

이력관리, 카테고리

다중 카테고리

맵핑 테이블

코드의 재사용에 따른 문제

코드의 재사용