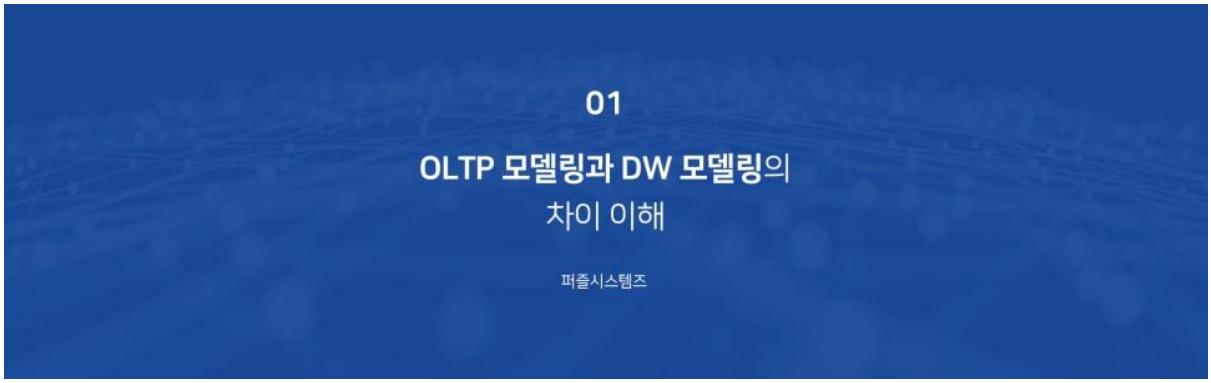



정보계용 DW 데이터 모델을 접하는 초창기에 한번 쯤은 겪게 되는 것이 과연 기간계용 OLTP 데이터 모델과의 차이가 무엇이고 왜 이렇게 설계해야 하나 등일 것이다. 인터넷 검색을 조금만 해도 두 모델의 차이점을 언급하는 자료는 쉽게 찾을 수 있을 것이지만 조금 다른 방식으로 얘기를 풀어볼까 한다.

기간계용 OLTP 데이터 모델과 정보계용 DW 데이터 모델 모두 기본적으로 ERD를 이용하여 표현한다. ERD(Entity Relationship Diagram)는 이름에서 알 수 있듯이 ㈜엔터티(Entity)와 엔터티 사이의 관계(Relationship)를 도식화 한 것으로 OLTP 모델과 DW 모델은 언뜻 보기에 특별한 차이가 없어 보인다. 특히, 복잡한 DW 모델을 보면 더욱 그러하다. 많은 사람들이 이미 OLTP 모델에 비해 DW 모델이 상대적으로 더 비정규화되어 있고 스타 스키마나 스노우플레이크 스키마 기법을 적용한다는 것 쯤은 알고 있는데 근본적인 차이에 대해서는 여전히 고개를 갸웃한다. OLTP 모델도 필요에 의해서 비정규화 과정을 거치고 집계 테이블을 반영하다 보면 그게 그거 같다. 이는 두 모델 사이의 근본적인 목적 보다 단순히 설계를 위해 사용하는 도구 기능과 기법에 초점을 맞춰서 이해하려 해서 그런 것으로 보인다. 이제 두 가지 모델의 목적이 무엇인가를 살펴봄으로써 그러한 의구심을 줄여 보고자 한다.

우선 OLTP 데이터 모델은 비즈니스 수행 과정에서의 트랜잭션 처리를 원활하게 처리하고 관리하는 것을 목표로 한다. 또한 빈번한 데이터 입력 및 수정 과정에서 데이터의 정합성을 보장하도록 설계 한다. 이러한 특성상 엔터티 사이의 관계가 무척이나 엄격하며 우선적으로 그 조건을 충족해야 한다. 비유하자면, 할아버지가 있어야 아버지가 있을 수 있고, 아버지가 있어야 아들이 있을 수 있다(존재관계). 또한, 사람이 있어야 사람의 행동이 있을 수 있다(행위 관계).

이는 엔터티 사이에는 전제 조건이나 선행 조건이 존재하고 데이터가 발생시 그 조건을 만족해야 함을 의미한다. 예를 들어, 사원은 (반드시 소속 부서가 있어야 한다면) 부서 없는 사원은 존재할 수 없다. (주문의 대상이 반드시 있어야 한다면) 상품 없는 주문은 발생할 수 없다. 또한 (주문의 주체가 반드시 있어야 한다면) 고객 없는 주문은 발생할 수 없다. 그리고, 그러한 제약 조건은 구체적인 엔터티 사례(예를 들어, 고객의 경우 김기수, 홍가람 등 개별 고객)가 발생시에 충족되어야 한다. 따라서, 기본적으로 제약 조건이 없는 데이터(주로 최상위의 마스터성 코드 테이블)들부터 순차적으로 데이터를 등록하게 된다.

또한, 데이터 중복 관리를 최소화함으로써 빈번한 데이터 입력 및 수정 과정에서도 데이터의 정

합성을 확보하려 노력한다. 설계 측면에서는 정규화 과정이 그에 해당한다.

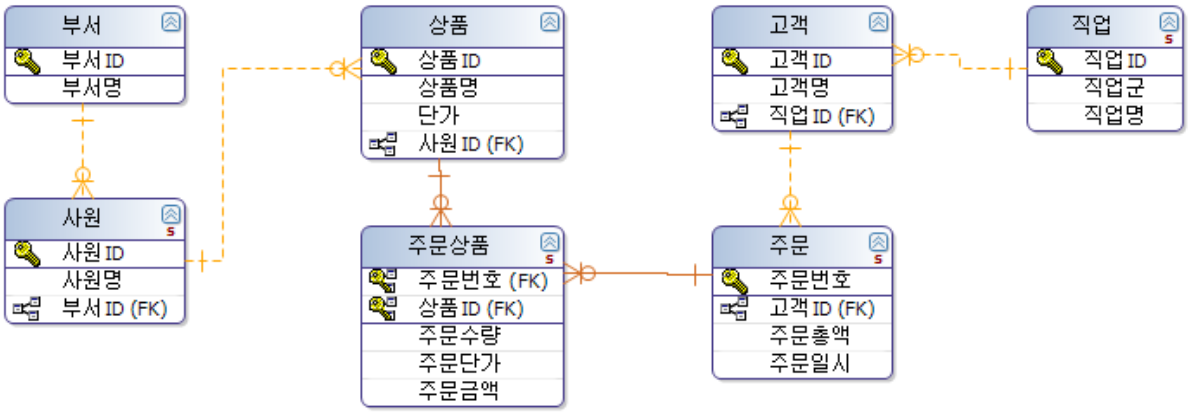



그림 1. 기간계 OLTP 데이터 모델

반면에 DW 데이터 모델은 모든 데이터를 누락 없이 체계적으로 적재하고, 분석 대상이 되는 데이터를 접근하는 경로를 설계하는 것이다. 이 때 경로는 고정된 것이 아니며 어떠한 경로로 접근하더라도 동일한 방식으로 접근할 수 있어야 한다. 당연히, 동일한 결과를 얻어야 하며 비슷한 성능을 내야 한다. (스타스키마 또는 스노우플레이크 스키마 기법의 적용 배경). 또한 접근 경로는 분석의 요구가 다양한 만큼 다양한 형태로 추가될 수 있다. 따라서, 표현은 동일한 ERD 형태로 하지만 목적 자체가 다른 것이다. 그림 2의 DW 모델 경우를 보자.

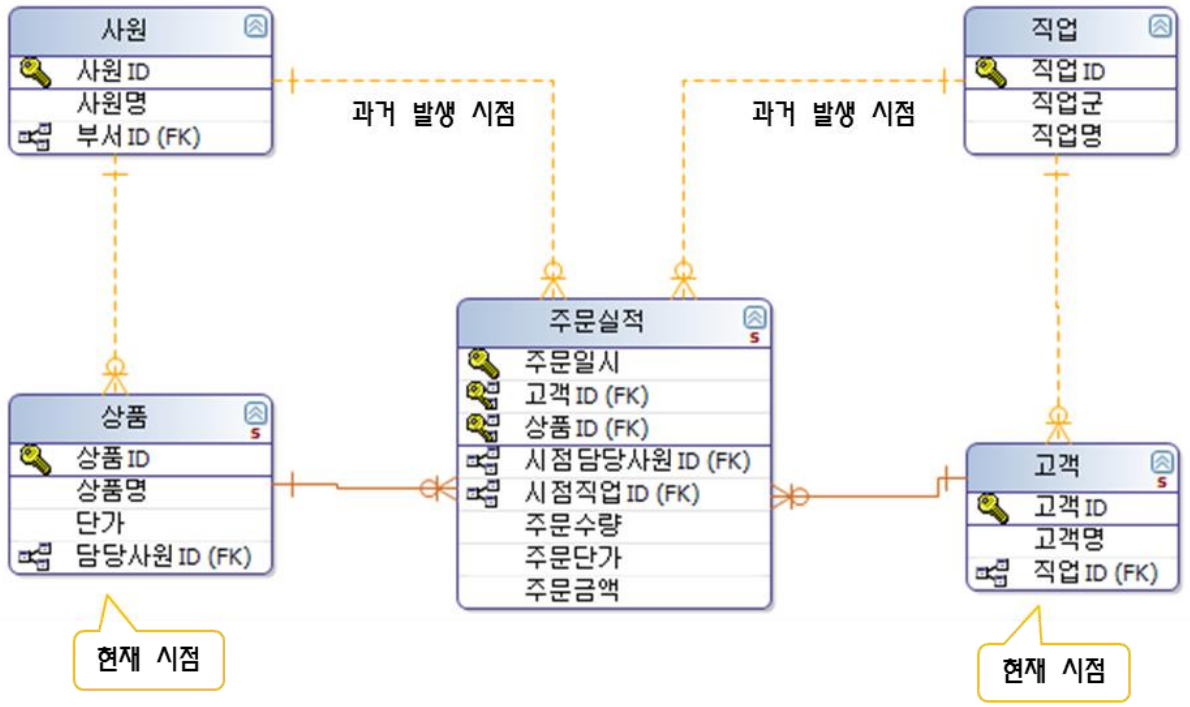



그림 2. 정보계 DW 데이터 모델

언뜻 보기에 기간계 OLTP 데이터 모델과 큰 차이가 없어 보이지만, 분석하고자 하는 데이터(주문실적)를 중심으로 접근 경로(사원, 상품, 직업, 고객)를 설계하고 있다. 소위 스타스키마의 전형적인 형태로서, 특정 트랜잭션 처리나 데이터 업데이트를 위한 최적화 형태가 아니다.

이 때, 사용자가 어떠한 경로로 주문 실적을 접근하던 동일한 결과와 비슷한 성능을 보장받아야 한다. 성능 보장은 과거에는 많은 부분이 개발자나 DBA의 몫이었는데 최근 시장의 많은 DBMS 제품들이 기능 개선을 통해 이를 효과적으로 지원하고 있다.

예제 모델에서 하나 더 눈에 띄는 것은 주문실적에 시점담당사원, 시점직업 등 기간계의 주문 관련 테이블에 없던 것들이 추가된 것이다. 이는 데이터를 현재 시점 뿐만 아니라 과거 시점 기준으로도 분석하겠다는 요구를 반영한 것으로 상품담당사원이력, 고객직업이력 등과 같은 이력 데이터를 이용하여 주문실적 적재시 함께 만들어낸다. 당연히, OLTP 기간계와 달리 과거 시점에만 존재했던 사원이나 직업도 관련 마스터 테이블에 모두 추가해야 한다.

물론, 여전히 OLTP 모델에서와 같이 엔터티 간 관계에서의 전제 조건이나 선행 조건을 만족하는 것이 중요하다. 그러면 그 차이는 무엇인가? 미리 요약하면 DW 모델에서는 데이터 발생 시점이 아니라 데이터 적재 시점 (또는 조회 시점) 기준으로 일정한 개입을 통해 그러한 관계를 충족 시킨다는 것이다. 그럼 그 의미를 조금 더 구체적으로 살펴 보자.

기본적으로 DW 데이터를 적재시 시스템 오픈 시점 기준으로 데이터를 초기 적재하고 이후 일정한 시간 간격으로 증분 적재를 한다. 이 때, 통상적으로 OLTP 모델의 경우처럼 사원, 상품, 고객 등과 같은 마스터성 데이터 적재를 먼저하고 주문실적 같은 실적 데이터를 적재한다. 가장 눈에 띄는 차이점이라면, 사원, 상품과 같은 기준성 테이블에 Unknown 등과 같은 기준 데이터를 인위적으로 미리 추가한다. 그렇게 하는 이유는 무엇인가?

예를 들어, 과거 10년치의 주문실적을 분석하고자 하는데 상품 마스터 관리 소홀로 최근 상품만 관리되고 있고 상품별 담당사원 이력도 관리되지 않고 있다고 하자. 그렇다면 주문실적에는 상품ID가 있지만 상품 테이블에는 해당 상품ID가 없는 경우가 발생할 수 있다. 이 때의 선택은?

우선, 주문실적의 상품ID를 Unknown에 해당하는 ID로 업데이트하거나 분석용 상품ID_2를 추가해서 이중으로 관리할 수 있을 것이다. 또는 관리되지 않는 주문실적의 상품ID를 데이터 적재 시점

에 상품 마스터에 먼저 추가하고 상품의 나머지 속성값들은 NULL로 채우거나 대체값으로 채울 수 있을 것이다. 그런데, 이 경우 해당 상품의 담당 사원을 주문 실적에 함께 관리하지 않았다면 역시 알 방법이 없다. 따라서, 상품의 담당사원도 Unknown으로 처리해야 하고, 사원에도 엔터티 간 관계를 만족할 수 있도록 Unknown이란 사원 데이터를 추가해야 한다.

이상으로 OLTP 모델과 DW 모델 사이의 몇 가지 차이점을 살펴봤다. 차이는 이 외에도 더 많겠지만 개인의 몫으로 남겨두고, 여기에서는 모델을 설계하거나 해석할 때 보다 확신을 가지기를 기대하며, 이를 위해 각자의 기준을 세우는데 필요한 목적과 배경에 대해 최소한으로 살펴보았다.

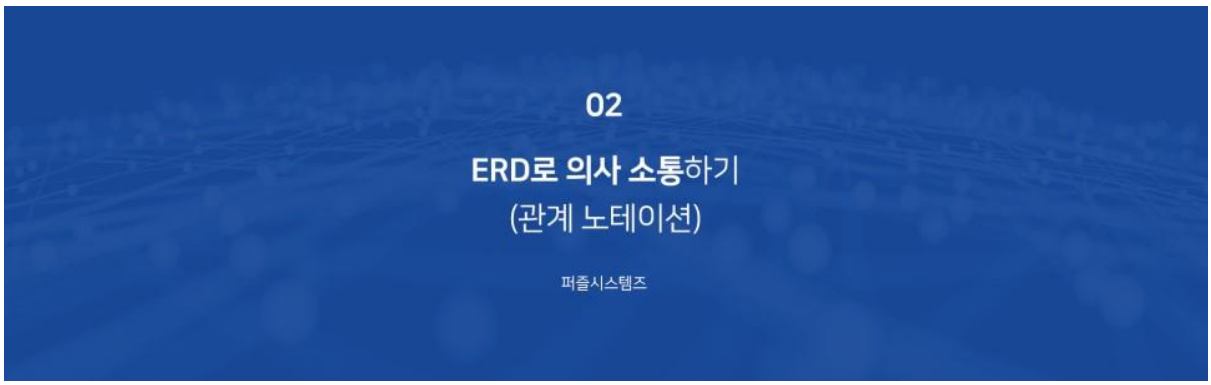
---

주)

엔터티 타입(Entity Type)은 공통 속성을 가지는 엔터티(Entity)들의 집합이다. 예를 들어, 사원은 사원ID, 사원명, 직급 등과 같은 공통 속성들을 가지는 엔터티 타입이다.

엔터티(Entity) 또는 엔터티 인스턴스 (Entity Instance)는 엔터티 타입의 구체적인 하나의 사례이다. 예를 들어, 사번 009, 권오주, 컨설팅사업부 등의 속성값을 가지는 구체적인 하나의 사원 개체가 엔터티이다.

엄격하게는 이렇게 개념이 구분되지만 많은 경우, 둘을 혼용하여 구분 없이 엔터티란 용어를 많이 사용한다. 그러므로, 여기에서도 특별히 구분하지 않을 것이므로 문맥에 따라 이해하길 바란다.




데이터 모델링은 비즈니스와 관련된 데이터 요구 사항들을 분석하고 정의하는 프로세스로서, 대개 문제에 대한 해결책을 그래픽으로 표현한다.

여기에서 데이터 모델링을 위한 기법은 여러 가지가 있을 수 있는데 가장 대중적으로 일반화된 것이 관계형 모델이고, 이 때 모델을 표현하는 방법으로 ERD(Entity Relationship Diagram)를 사용한다. (조금 더 기술적으로 표현하면 ERD는 관계형 모델을 사용하여 데이터베이스 구조를 표현하는 다이어그램이다.)

여기에서 중요한 것은 ERD는 의사소통의 수단으로서 모두가 동일하게 이해하는 약속된 언어여야 한다. 데이터 모델을 설계하는 과정에서 또는 이미 마무리된 설계를 살펴볼 때, 현업이던, 모델러던, 개발자던 모두가 같은 의미로 이해하고 해석해야 한다. 문제는 그림 1에서 보는 바와 같이 사용하는 툴에 따라, 같은 툴이라도 표기법에 따라 같은 의미를 다르게 표현할 수 있다는 것이다. 따라서, 데이터 모델링을 함에 있어 사용하는 툴의 표기법을 먼저 숙지해서 정확한 의사소통이 가능하도록 하는 것이 매우 중요하다.

출처 : Data Modeling Essentials Third Edition

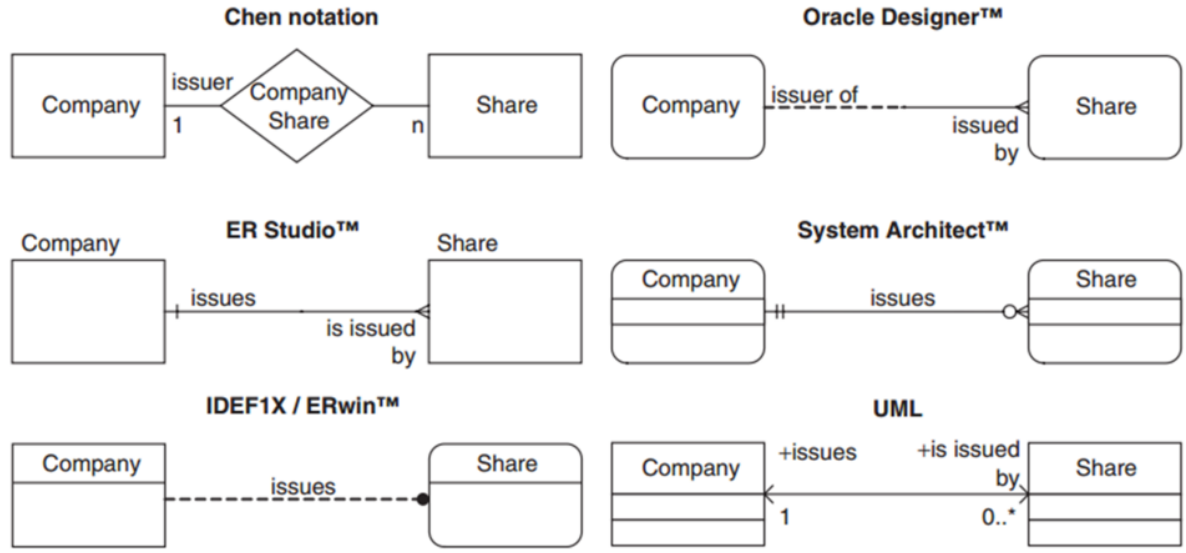



그림 1. 다양한 관계 표현법

ERD에서 엔터티 사이의 관계를 나타내는 가장 대중적인 표기법으로는 소위 까마귀발 (Crow's Foot)로 불리는 표기법이 있다. 그림 2에서와 같이 관계 표시의 끝점 위치에 대시(1 의미), 원 (0 의미), 까마귀발(N 의미) 등을 조합하여 카디널리티를 표현한다. 또한, 주1)엔터티 간의 주2)식별 관계는 실선으로 비식별 관계는 점선으로 표현한다.

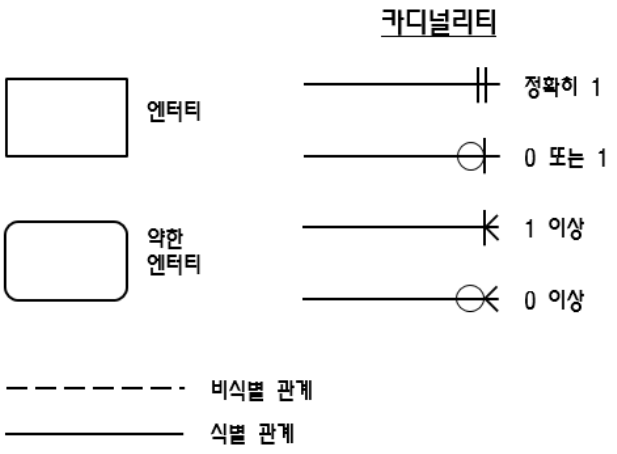



그림 2. Crow Foot 표기법

이 표기법은 1980년대 Richard Barker에 의해 처음 사용되기 시작했으며 오라클 툴셋에 적용되고 있다(Barker Notation). 또한, James Martin과 Clive Finkelstein에 의해 개발된 IE(Information Engineering) 표기법에서도 사용되고 있다. (IE 방식도 태생이 둘이다 보니 두 계열 간에도 약간의 차이가 있다.) 그러나, 까마귀 발이라고 같은 까마귀 발이 아니다. 실제로 Barker의 표기법은 그림 2와 차이가 있다. 우선 카디널리티 표현의 경우 점선으로 0 또는 1을 표현하고, 실선으로 정확한 1을 표현한다. 대시는 식별 관계를 표현한다. 따라서, 같은 까마귀발 표현법이라고 하더라도 Barker 방식인지, IE 방식인지에 따라 해석 방식이 다름을 인지해야 한다.

다음으로 많이 사용되는 것 중에 IDEF1X(Integration Definition for Information Modeling) 표기법이 있다. 이 표기법은 1980년대 미 공군에서 사용하면서 개발되었다. 그림 3에서와 같이 관계 끝에 작은 원을 이용하여 카디널리티를 표현한다. 원이 없으면 정확히 1, 빈 원은 0 또는 1, 속이 꽉 찬 원은 0 또는 N을 의미한다. 1 또는 N의 경우는 P를 같이 표현한다. 또한, 엔터티 간의 식별 관계는 실선으로 비식별 관계는 점선으로 표현한다.

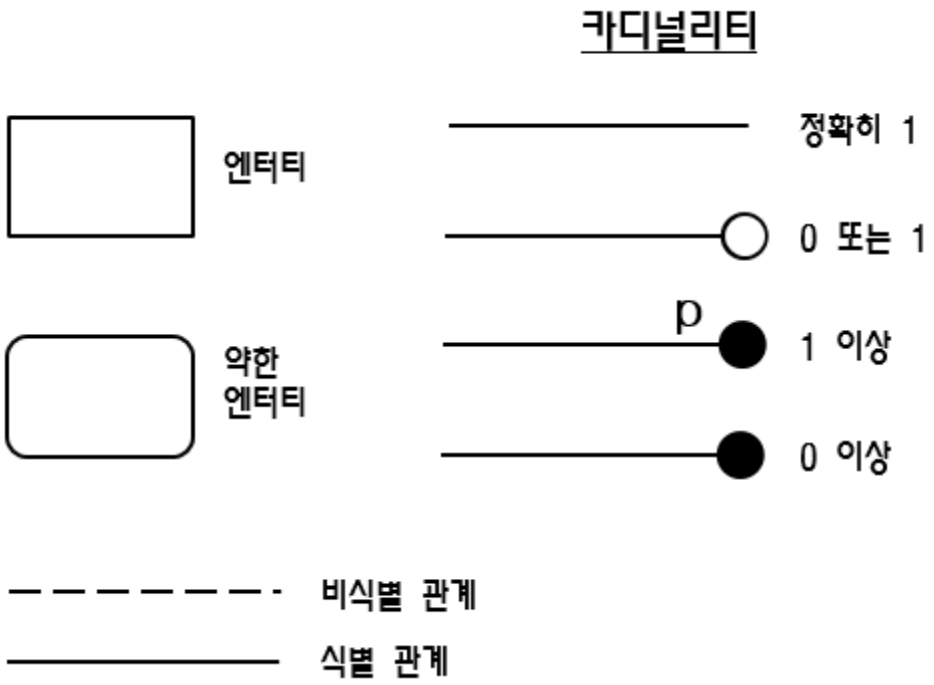



그림 3. IDEF1X 표기법

그런데, 변형으로 모두 속이 꽉 찬 원을 사용하면서 Z(0 또는 1), P(1 또는 N), N(정확히 N개) 등을 함께 표시하여 나타내기도 한다. 이때, 아무 표시가 없으면 0, 1 또는 N을 의미한다.

지금 쯤이면 사용하는 툴의 표기법을 왜 먼저 숙지해야 한다고 하는지 더 피부에 와 닿을 듯하다. 공동 작업시 표기법에 대한 숙지 없이 정확한 의사소통이 불가능하기 때문이다. 그리고, 지금까지 설명한 것도 절대적인 것이 아니다. 큰 맥락에서 그렇다는 것이고 툴에 따라 세부적으로 여전히 차이가 날 수 있으므로 반드시 사용하는 툴의 도움말을 통해 표기법을 따로 숙지해야 한다.

그럼 위에서 언급한 표기법을 염두에 두고 실제 툴을 사용하여 다음의 시나리오를 표현해 봄으로써 그 차이를 느껴 보도록 하겠다. (예제에 나타나는 수퍼타입-서브타입 관계는 나중에 별도의 아티클에서 살펴보도록 하겠다.)

시나리오

하나의 고객은 여러 건의 주문을 할 수 있다.

하나의 주문은 반드시 하나의 고객에 의해 이뤄진다.

하나의 주문은 반드시 하나 이상의 주문상품을 포함한다.

하나의 주문에 같은 상품이라도 다른 가격으로 주문 될 수 있다.

하나의 주문상품은 반드시 하나의 주문에 포함된다.

하나의 상품은 여러 개의 주문상품에 포함될 수 있다.

하나의 주문상품은 반드시 하나의 상품을 포함한다.

고객은 법인고객과 개인고객이 있으며 하나의 고객은 어느 한쪽에만 속한다.

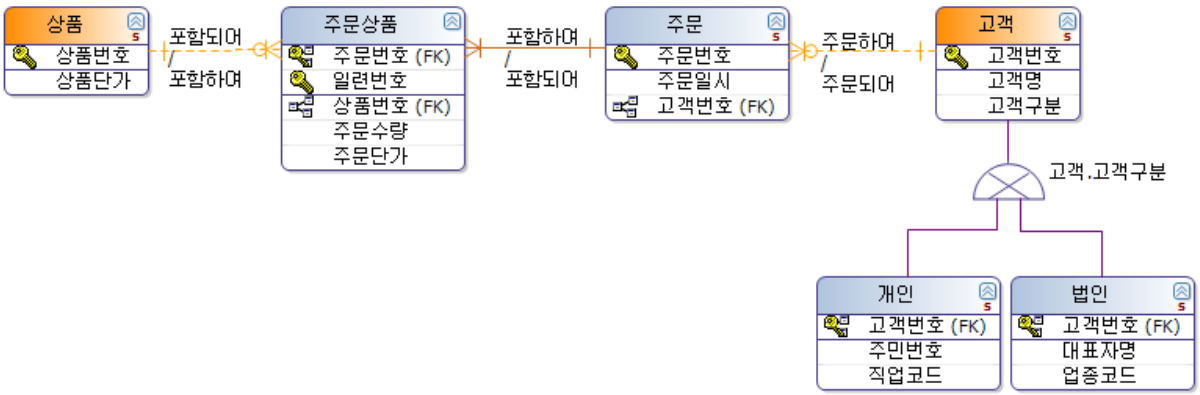



그림 4. MicroDesigner의 IE 표기법으로 설계 (까마귀발)

그림 4의 IE 표기법을 보자. 참고로, 엔터티 중에 오렌지 색으로 표현한 것(상품, 고객)은 Strong Entity를 의미한다.

고객이 상위 엔터티이고 주문은 하위 엔터티이다. 관계를 점선으로 표시했으므로 비식별 관계이다. 즉, 고객 번호는 주문의 식별자에 포함되지 않는다. 까마귀 발을 이용한 카디널리티 표현은 고객은 주문을 안 할 수도 있고 한번 할 수도 있고 여러 번 할 수도 있음을 의미한다 (0, 1 or N). 또한, 하나의 주문은 정확히 하나의 고객에 의해 이뤄짐을 나타낸다 (1). 표현상의 차이점은 MicroDesigner에서는 대시 2개를 하나로 표현하고 있다는 것이다.

다음으로 주문과 주문상품을 보면 주문이 상위 엔터티이고 주문상품이 하위 엔터티이다. 관계를

실선으로 표시했으므로 식별 관계이다. 즉, 주문의 식별자인 주문번호가 주문상품의 식별자 일부로 포함된다. 까마귀 발을 이용한 카디널리티 표현은 하나의 주문은 하나 또는 그 이상의 주문상품을 포함해야 하며 (1 or N), 하나의 주문 상품은 정확히 하나의 주문에 포함되어야 함(1) 을 의미한다.

다음으로 상품과 주문상품의 관계는 고객과 주문 사이의 관계에 준해서 해석하면 된다.

이어서, DA#의 Barker 표기법과 MicroDesigner의 IDEF1X 표기법을 이용한 예제를 추가했다. 해석은 각자의 몫으로 남기겠다.

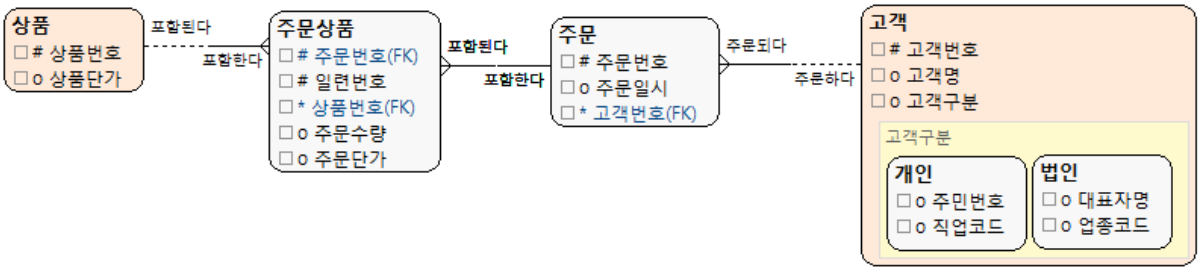



그림 5. DA#의 Barker 표기법으로 설계 (까마귀발)

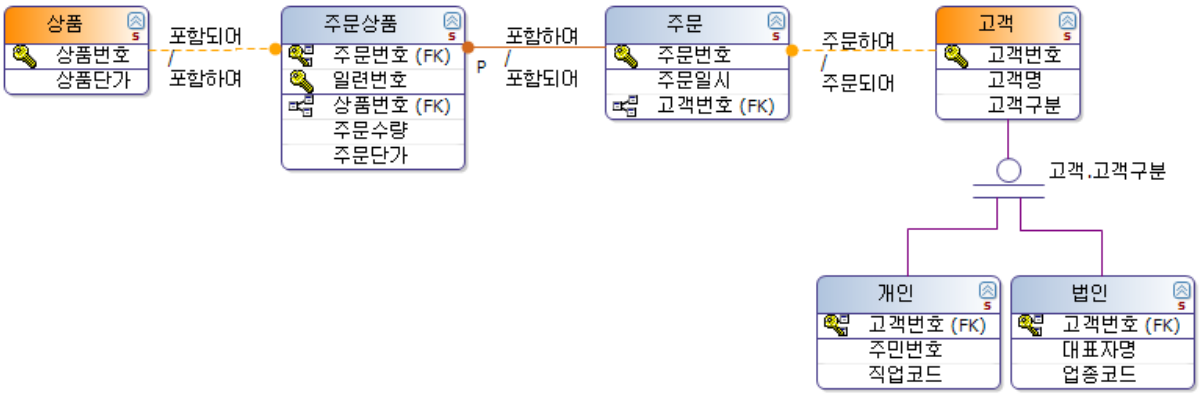



그림 6. MicroDesigner의 IDEF1X 표기법으로 설계

---

주1) 그림 2, 그림 3의 표기법에서 엔터티의 의미는 독립적으로 존재 가능한 엔터티를 의미하며 직사각형으로 표현한다 (Strong Entity). 예를 들어, 예제에서 고객 엔터티가 있다. 반대로 약한 엔터티(Weak Entity)는 다른 엔터티의 존재 없이는 존재할 수 없는 엔터티를 의미한다. 예를 들어, 예제에서 주문 엔터티가 있다.

주2) 식별 관계는 하위 엔터티의 존재 여부가 상위 엔터티에 의존함을 의미한다. 따라서, 상위 엔터티의 식별자가 하위 엔터티를 구분하는 식별자(1차키)에 일부로 포함된다. 반대로 비식별 관계는 하위 엔터티의 존재여부가 상위 엔터티에 의존하지 않음을 의미한다. 따라서, 하위 엔터티의 식별자에 상위 엔터티의 식별자가 포함되지 않는다.


이전 아티클에서 ERD 는 의사소통의 수단으로서 모두가 동일하게 이해하는 약속된 언어여야 한다고 수 차례에 걸쳐 강조했다. 이를 위해, 사용하는 툴의 표기법을 먼저 숙지해서 정확한 의사소통이 가능하도록 하는 것이 매우 중요하다 했다. 그러한 연속선상에서 표기법별로 ㈜수퍼-서브 타입의 표현 형태들과 의미들을 살펴보도록 하겠다.

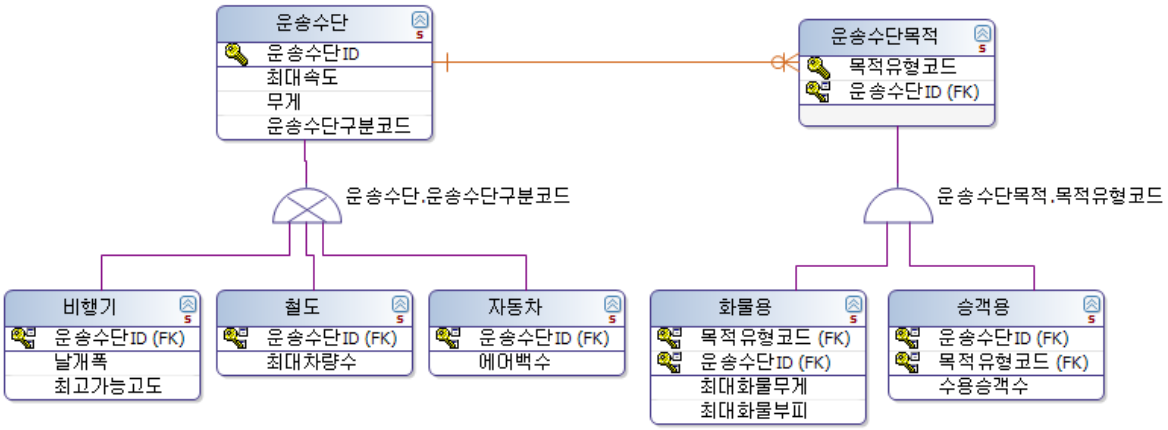



그림 1. MicroDesigner 의 IE 표기법으로 설계

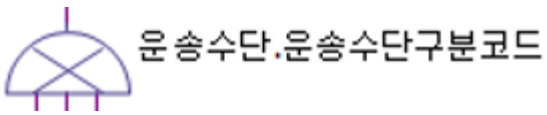



그림 1-1. 서브타입의 Exclusive 관계 표현

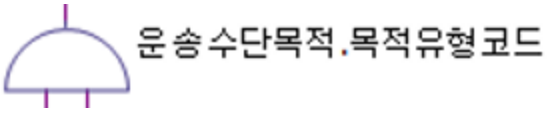



그림 1-2. 서브타입의 Inclusive 관계 표현

그림 1 은 MicroDesigner 에서 IE 표기법으로 설계한 수퍼-서브 타입 모델이다. 좌측부터 해석을 하면 공통의 속성들을 가지는 운송수단 엔터티 타입이 수퍼 타입이고, 각각의 고유한 속성들을

추가적으로 가지는 비행기, 철도, 자동차 엔터티 타입이 서브 타입이다. 그 의미를 해석해 보면 모든 운송수단은 비행기, 철도, 자동차 중에 오직 하나에 속함을 뜻한다. 예를 들어, 동시에 비행기이며 자동차인 것들은 존재하지 않는다. (물론 이것이 진리는 아니다. 가까운 미래에 도로에서 주행 중이던 자동차가 하늘을 나는 것을 보게 될 것이고 그것을 반영하려면 모델은 수정되어야 한다.) 그림 1-1 은 그러한 배타적 관계를 나타낸다. 이 때, 서브 타입을 구분하는 구분자인 운송수단구분코드는 비행기, 철도, 자동차에 해당하는 3 개의 코드 값을 가질 수 있다.

오른쪽도 같은 방식으로 해석하면 운송수단목적 엔터티 타입이 수퍼 타입이고 화물용, 승객용 엔터티 타입이 서브 타입이다. 그러나 의미는 앞에서와 다르다. 즉, 하나의 운송수단은 화물용과 승객용 중 하나에만 속하는 것이 아니라 두 개 모두에 속하는 경우도 있음을 뜻한다. 그림 1-2 는 그러한 포함 관계를 나타낸다. 이 때, 서브 타입을 구분하는 구분자인 목적유형코드는 화물용, 승객용, 화물용&승객용에 해당하는 3 개의 코드 값을 가질 수 있다.

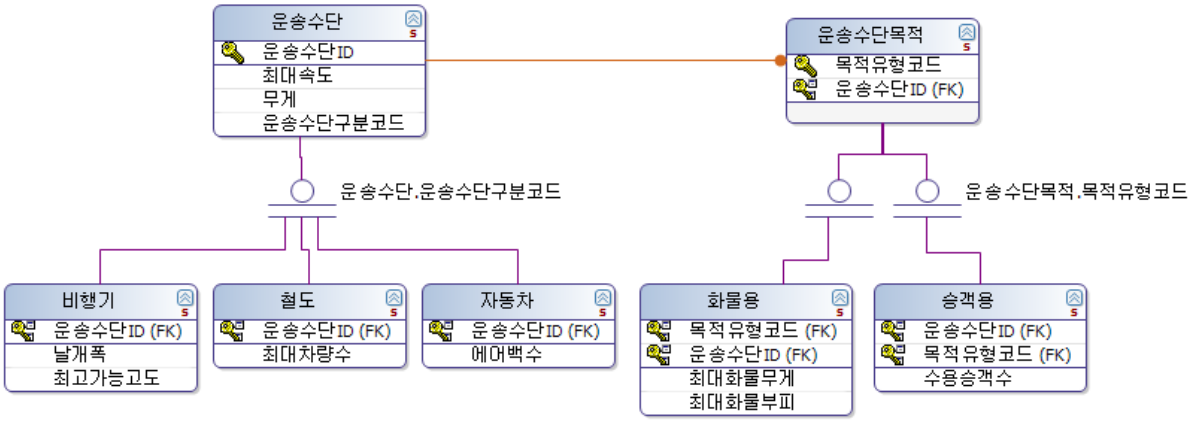



그림 2. MicroDesigner 의 IDEF1X 표기법으로 설계 (Complete)

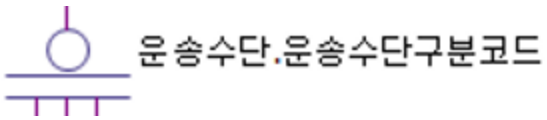



그림 2-1. 서브타입의 Complete/Exclusive 표현

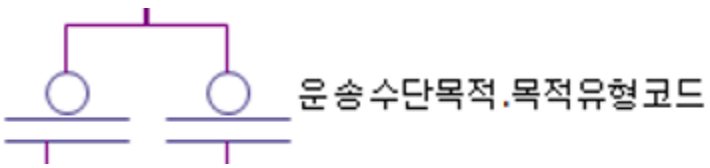



그림 1-2. 서브타입의 Complete/Inclusive 표현

그림 2 는 그림 1 의 IE 표기법을 IDEF1X 표기법으로 전환한 것이다. IE 표기법과 달리 Inclusive 관계를 두 개의 심볼로 표시한다. 아쉽게도 많은 툴에서, Inclusive 관계를 IE 와 IDEF1X 양방향으로 자동 전환을 지원하지 못한다. 위 예제 그림은 두 개의 서브-타입을 따로 그려서

Inclusive 를 표현한 것이다. 물론, 동일한 서브 타입 구분자를 표시함으로써 Inclusive 인 것을 알 수는 있지만, 시각적으로 쉽게 눈에 들어오지 않는다. 팁으로 관계선 색깔을 다른 색으로 설정하거나, 서브 타입의 색을 다른 색으로 처리하는 것도 한가지 방법이다.

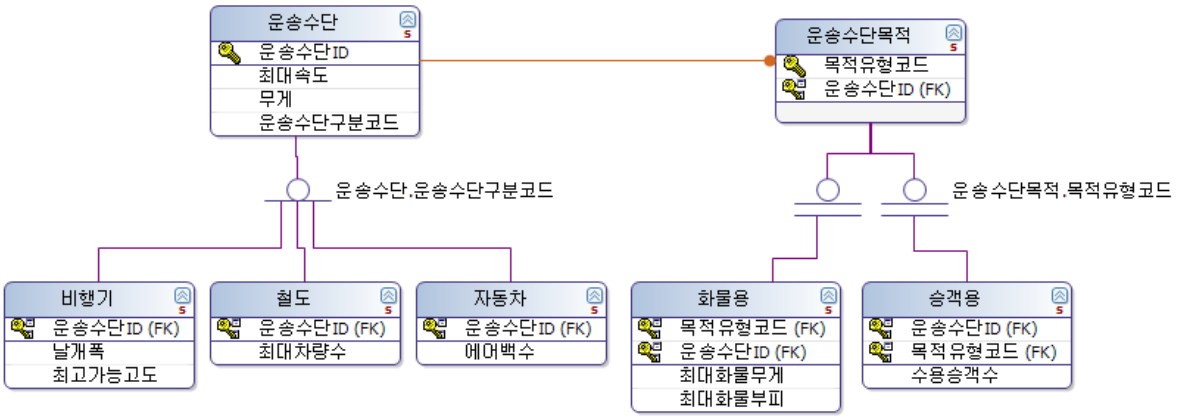



그림 3. MicroDesigner 의 IDEF1X 표기법으로 설계 (Incomplete)

그런데, IDEF1X 표기법에서는 또 다른 개념을 지원한다. Complete, Incomplete 개념이다. Complete 는 서브 타입이 모두 확정되었음을 의미한다. Incomplete 는 모델링 과정에서 아직 모든 서브 타입을 확정했다고 확신할 수 없을 때 활용할 수 있다. 예를 들어, 운송수단 중에 서브 타입으로 선박이 더 있을 수 있는데 지금까지의 요건들을 살펴볼 때 확신할 수 없다고 하면, Incomplete 로 해 놓고 나중에 다시 살펴보는 것이다. 그림 3 에서 좌측의 수퍼-서브 타입 심볼은 그러한 의미를 담고 있다

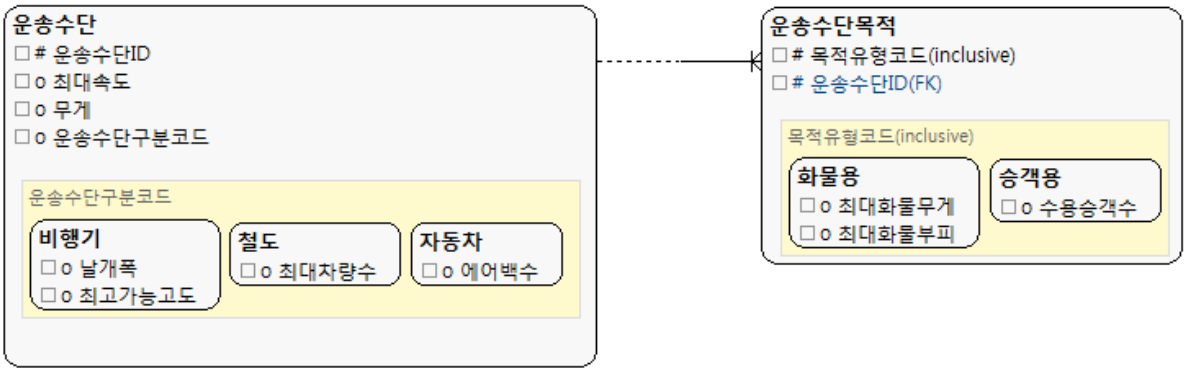



그림 4. DA#의 Barker 표기법으로 설계

그림 4 는 그림 1 의 IE 표기법 모델을 Barker 표기법으로 전환한 것이다. IE 와 IDEF1X 와 달리 서브타입을 수퍼 타입의 내부에 표현하고 있다. 예제에서 사용한 툴에서는 기본적으로 수퍼-서브 타입 관계를 Exclusive 로 처리하고 Inclusive 인 경우는 구분자에 (Inclusive)를 함께 표시하고 있다.

이상에서 살펴본 바와 같이 표기법에 따라 수퍼-서브 타입을 표현하는 방법과 정도가 다르다. 또한 툴에 따라서도 조금씩 다를 수 있다. 따라서, 다시 한번 강조하건 데 프로젝트에 참여하는 모든 멤버들이 사용하는 툴에서의 표기법을 숙지함으로써 공통의 언어와 공통의 용어로 의사소통하는 것이 가능하도록 해야 한다.

---

주) 데이터 모델링에서 수퍼-서브 타입을 활용하는 주된 목적은 일반화되어 통합된 것을 구체화하며 분류하여 비즈니스 의미를 보다 명확하게 표현하기 위함이다. 예를 들어, 단순히 고객이라고 하는 것 보다는 개인고객, 법인고객이라고 하는 것이 보다 그 의미가 명확하다. 이 때, 개인고객과 법인고객이 가지는 속성 중에 공통적인 것들을 모아서 수퍼 타입으로 표현하고, 개인고객과 법인고객은 각자 고유의 속성들 만을 가지는 서브 타입으로 표현한다. (고유한 속성이 없다면 서브 타입으로 표현해 봐야 의미가 없다. 그러나, 의사소통용으로는 여전히 쓸모가 있다.) 이 때의 장점으로 개별 서브 타입에만 의미를 가지는 속성들을 쉽게 파악할 수 있다. 또한, 수퍼 타입과 개별 서브 타입들은 독립적인 엔터티 타입으로서 다른 엔터티 타입들과 독립적으로 관계를 표현할 수 있다.


DW(Data Warehouse) 설계에 왕도가 있을까? 미리 결론적으로 말하면 왕도는 절대로 없다고 본다. 상황에 맞게 더 나은 길을 찾아가는 지혜가 필요할 뿐...

한 때 DW 업계의 양대 거두가 들고나온 접근 방식을 가지고 갑론을박하던 시절이 있었다. 양대 거두란 Bill Inmon 과 Ralph Kimball 이다. Bill Inmon 은 Hub and Spoke 방식을 들고 나왔고, Ralph Kimball 은 Dimensional Bus Architecture 방식을 들고 나왔다. 그러나, 세상에 완벽한 것은 없다. 결국 서로 자기가 옳다고 치열하게 싸우고 있었는데 (심한 경우, 상대방 방식을 쓰면 5 년 안에 짐 쌀 것이란 말도 직접 들었었다. 누구라고 밝힐 수는 없고...) 어느 날 보면 은근슬쩍 서로의 장점을 취하는 쪽으로 조금씩 타협을 하고 있음을 보게 된다.

실제로 실전 모델링에서는 어느 방식을 사용하던 상황에 맞게 응용하고 필요한 경우 다른 쪽의 방식까지도 확대 적용하는 능력이 더 중요하다. 따라서, 각 방식의 특징과 차이점을 정확하게 이해하는 것이 중요하다. 이제 두 가지 방식에 대해 간단하게 살펴보도록 하겠다.

그림 1 은 Bill Inmon 의 Hub and Spoke 방식이다. Hub and Spoke 는 유일한 데이터공급자 역할을 하는 하나의 통합 DW 를 중심으로 주변에 연결된 데이터마트들의 전체적인 모습을 의미한다. 개인적 취향에 따라 EDW(Enterprise Data Warehouse) 방식, CIF(Corporate Information Factory) 방식, 3NF DW 방식 등으로 부르기도 한다.

그림에서 Staging 영역은 데이터 처리를 위한 저장소와 통상 ETL(Extraction, Transformation and Loading)이라 부르는 데이터 처리 로직의 집합을 의미한다. 차원은 적재 및 분석의 대상이 되는 데이터를 정의하고 설명하는 정보들로서 고객, 상품, 사원 등과 같은 기준성 정보라고 이해하면 된다. 또한 데이터를 소비하는 최종 사용자에게는 데이터를 접근하는 경로 및 데이터를 요약 및 집계하는 기준이 된다. Presentation 영역은 최종 사용자, 리포팅 도구, 분석 도구, 각종 어플리케이션 등에서 직접 쿼리가 가능하도록 데이터를 가다듬어 저장하는 곳을 의미한다. 이 방식에서 DW 는 Staging 과 Presentation 역할을 모두 포함하게 된다. 즉, 데이터 흐름의 연속선상에 있어서 필요한 데이터 처리(예를 들어, 데이터마트로의 데이터 적재)를 하는 대상임과 동시에 최종 사용자가 데이터 쿼리를 요청하는 직접적인 대상이 되기도 하는 것이다.

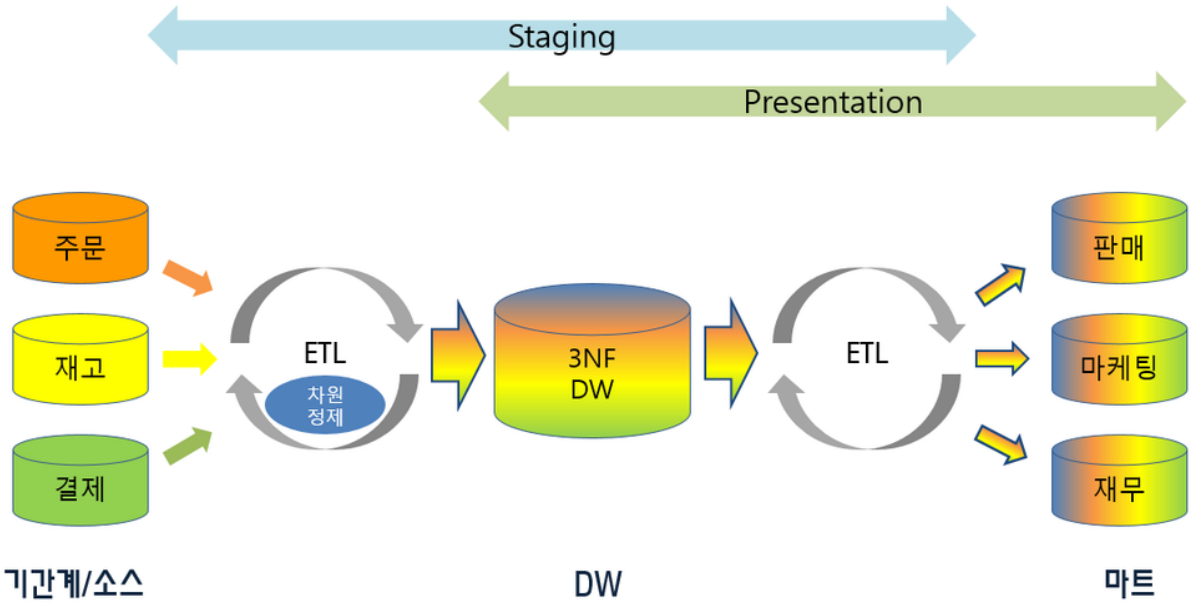



그림 1. Hub and Spoke 방식

이 방식의 핵심은 정규화된 통합 DW 를 설계하는 것이다. 하나의 통합 DW 에는 관리하고자 하는 가장 상세 수준의 원자 데이터(Atomic Data)를 적재함으로써 임의 수준의 데이터 요구에 대응할

수 있도록 한다. DW 는 보다 사용자 요구에 맞게 설계된 데이터마트로의 유일한 데이터 공급자 역할을 함으로써 전사에 걸친 데이터 정합성을 보장한다 (One version of truth). 예를 들어, 주문금액은 판매, 마케팅, 재무 데이터마트 모두에서 동일한 값을 가지도록 보장하는 것이다. (단, ETL 배치 실행 시간 차이나 시간 지체로 인한 버전 차이는 발생할 수 있다.) 이 때, 데이터마트는 보통 사용자의 다양한 데이터 접근 경로를 원활히 지원하며 보다 요약되고 집계된 데이터들을 제공할 수 있도록 설계된다. 즉, 어떤 식으로든 자연스럽게 스타 스키마와 스노우플레이크 스키마 형태를 띠게 된다.

그림 1 에서 하나 주의 깊게 볼 것은 색깔 처리 부분이다. 색깔별로 데이터의 출생과 흐름이 구분됨을 의미한다. 데이터마트를 보면 대개 판매, 마케팅, 재무 등과 같이 관심 부문별로 주제영역을 설계한다. 따라서, 업무 중심의 기간계 등과 같은 여러 곳으로부터 모아진 데이터들이 자연스럽게 섞이고 여러 마트들에 걸쳐 중복되어 나타나는 형태가 된다. 차원은 별도로 공통의 공간을 만들어 여러 마트에서 공유할 수도 있고 마트별로 중복하여 가져갈 수도 있다. 여하튼 중복이 존재한다는 것은 그만큼 버전을 맞춰주기 위한 노력이 필요함을 의미한다.

그림 2 는 Ralph Kimball 의 ㈜DW 버스 방식이다. 그림에서 보듯이 데이터마트가 따로 있는 것이 아니라 DW 가 곧 데이터마트이고 데이터마트가 곧 DW 가 된다. 따라서, 데이터의 중복이 없다. 또한, DW 는 순수한 Presentation 영역으로서 최종 사용자가 직접적으로 요청하는 쿼리의 대상 자체가 된다. 그러므로, 임의 유형의 다양한 쿼리 경로를 제공하기 위해 자연스럽게 스타 스키마 또는 스노우플레이크 스키마 형태를 띠게 된다.

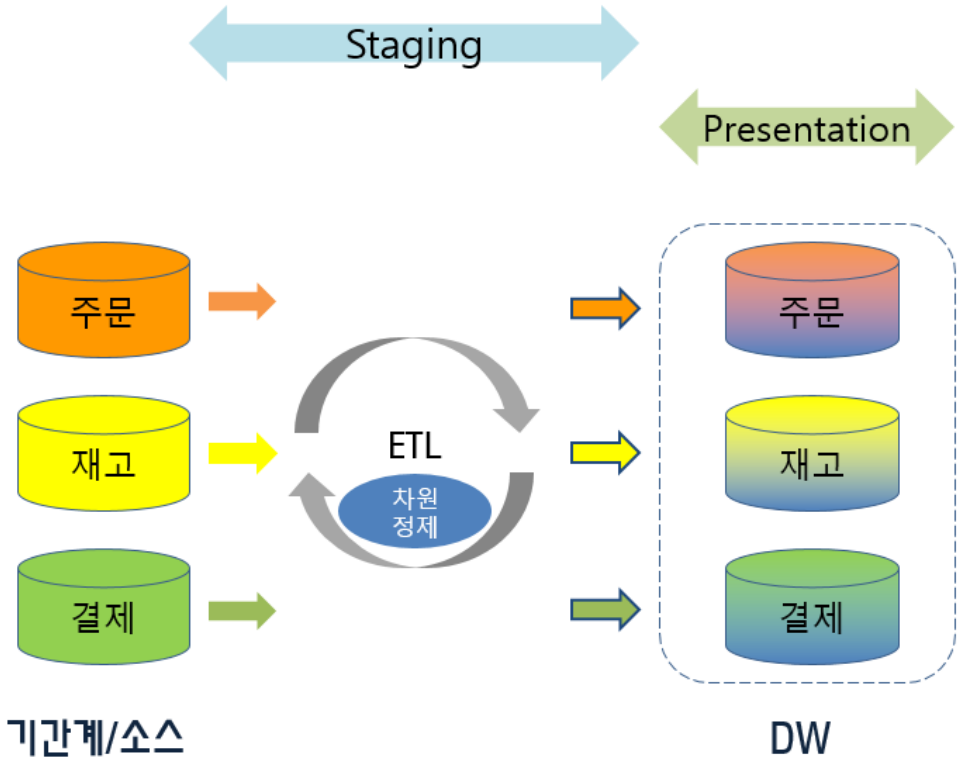



그림 2. DW 버스 방식

이 방식의 핵심은 견고하고 확장성 있게 DW 버스를 설계하는 것이다. 소위 말하는 Conformed Dimension 을 잘 설계하는 것이다. 전사에 걸쳐 표준화된 공유 차원을 설계하는 것으로 이해하면 된다. 예를 들어, 고객이라는 전사에 걸쳐 표준화된 차원은 어느 마트 영역에서도 함께 공유하여 사용할 수 있는 공통 차원인 것이다. 또한 Conformed Fact 를 설계한다. 전사에 걸쳐 표준화된 하나의 유일한 팩트를 설계하는 것이다. 예를 들어, 주문금액은 주문이라고 하는 논리적인 마트 영역에만 유일하게 존재하며 필요하면 다른 영역에서 언제든지 바로 참조하는 것이다. 이 때 참조하는 연결 고리로서 Conformed Dimension 을 사용한다.

그림 2 를 주의 깊게 들여다 보면 마트 부분이 Hub and Spoke 방식과 다름을 알 수 있다. 즉, 관심 부문이 아니라, 비즈니스 프로세스와 매핑하여 주제 영역들을 설계하고 있다. 따라서, 자연스럽게 마트 영역간 데이터 중복 문제가 배제된다. 그림에서 색깔 부분을 주의해서 보기 바란다. 주문 마트 영역은 주문 업무 시스템에서 온 데이터만을 포함함을 의미한다. 재고와 결제도 마찬가지이다. 그리고, 모든 마트 영역은 DW 버스를 통해 견고하게 서로 연결되고 묶이는 것이다.

이상 가장 대표적인 특징 위주로 Hub and Spoke 방식과 DW Bus 방식에 대해 간단하게 살펴보았다. 보다 상세한 내용과 차이점들은 각자의 몫으로 남겨둔다.

앞에서 미리 얘기했듯이 어느 방식이 더 낫다고 확언할 수 없다. 각자 특징이 있고 장단점이 있다. 따라서, 유능한 모델러라면 어느 한가지 방식만으로 설계하는 것을 고집할 것이 아니라 큰 틀을 흔들지 않으면서도 다른 방식의 장점을 취할 수 있는지도 함께 고민하는 지혜를 발휘해야 할 것이다. 원칙은 지키되 비즈니스 요건에 가장 적합한 하이브리드 방식을 찾아내야 하는 것이다.

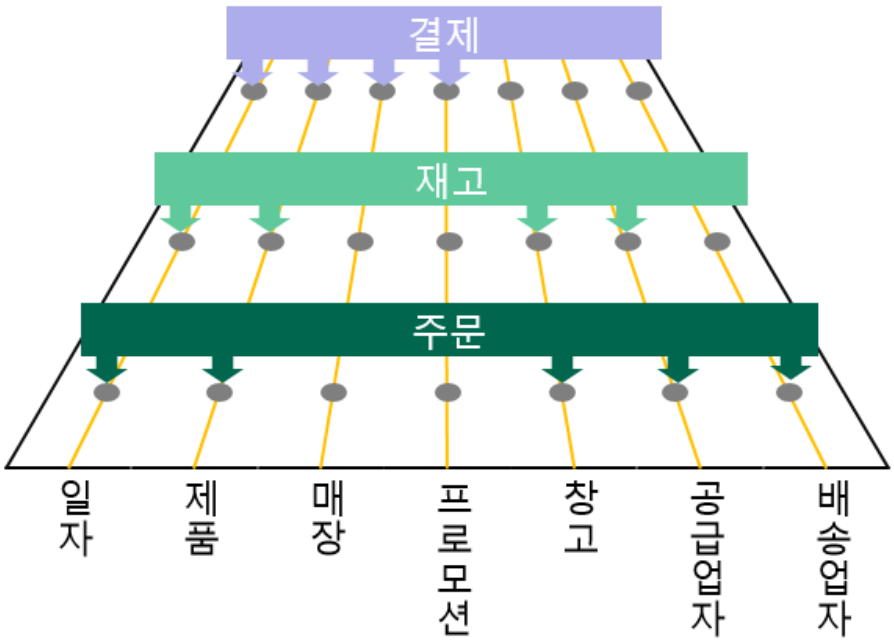



그림 3. DW 버스 아키텍처

주) DW 버스란 그림 3 에서 보듯이 차원들을 잘 정비해서 모아 놓은 것으로서, DW 의 뼈대 역할을 하도록 한 것이다. 주문, 재고, 결제와 같은 마트 영역을 단계적으로 추가할 때 기존의 모델을 흔들지 않고 확장성 있게 계속 추가할 수 있다. 이 때, 차원은 각각의 마트 영역들을 연결하고 묶어주는 역할을 하게 된다. 또한, 향후까지 고려하여 관리하고자 하는 가장 상세 수준까지 설계함으로써 모델을 확장할 때 견고함을 유지하도록 한다. 이는 Hub and Spoke 방식의 DW 와 마찬가지로 관리하고자 하는 가장 상세 수준의 원자 데이터(Atomic Data)를 적재함으로써 임의 수준의 데이터 요구에 대응할 수 있음을 의미한다.


최종 사용자 또는 각종 툴에서 직접 접근하여 활용할 데이터를 설계할 때, 팩트와 차원을 구분해서 설계한다. 팩트는 주문금액, 주문수량 등과 같은 분석 대상 자체가 되는 데이터를 의미한다. 차원은 고객, 상품, 사원 등과 같이 그러한 팩트를 설명하고 적재하는 기준이 되는 데이터를 의미한다. 차원은 팩트를 접근하는 경로의 역할을 하며 여러 팩트를 묶어서 분석할 수 있는 가교 역할도 한다. 또한 차원들은 여러 속성들을 가짐으로써 팩트에 대해 보다 다양하고 세밀한 분석이 가능하다. 예를 들어, 고객 차원의 경우 성별, 결혼여부, 학력, 신용등급 등 다양한 속성을 가진다. 때로는 일부 속성은 필요에 의해 독립적인 차원으로 설계하기도 한다.

그러나 항상 달콤한 열매만 있는 것은 아니다. 팩트와 차원은 밀접하게 연결되어 있는데, 시간이 지남에 따라 차원의 속성값이 바뀌면 여러 가지 문제들이 발생한다. 예를 들어, 고객의 신용등급이 A 등급에서 B 등급으로 바뀌면 해당 고객의 모든 주문 실적(금액, 수량 등)을 바라볼 때 어느 기준으로 볼지 이슈가 된다. 우선 현재 등급인 B 등급으로 모두 소급해서 보는 것이 있을 것이고, 과거 실적은 A 등급으로 현재 이후 실적은 B 등급으로 보는 것도 있을 것이다. 물론, 현업한테 물어보면 일단 돌아오는 답변은 "둘 다 가능하면 좋지요"일 것이다. (모름지기 욕심은 끝이 없는 법이니까...) 현명한 모델러라면 고객의 요구조건을 어떻게든 해결할 수 있도록 설계하는 것도 중요하지만, 사안별로 어떠한 것을 얻을 수 있고 어떠한 것을 잃을 수 있음을 충분히 이해시키고 토의함으로써 적절한 수준에서 타협점을 찾는 것이 중요하다. Everything is possible 이 결코 자랑이 아니란 것이다. 이제 요구조건도 해결하고 필요하면 적절하게 타협도 하기 위해 알아야 할 차원 속성의 변화 관리에 대해서 살펴보도록 하겠다. 우선 속성의 값이 느리게 변하는 경우 해당 차원을 SCD(Slowly Changing Dimension)라 하고 빠르게 변하는 경우는 RCD(Rapidly Changing Dimension)라 한다. 느리냐 빠르냐의 절대적인 기준은 없고 상대적인

것이며 설계할 때 상황에 따른 선택의 문제이다. 우선 SCD 를 관리하는 방법으로 8 가지 유형 정도를 생각할 수 있다. 이제 각각에 대하여 살펴보도록 하겠다.

첫째 유형은 속성값이 절대로 바뀌지 않는 것이다 (Type 0). 예를 들어, 고객의 최초신용등급은 시간이 지나도 바뀌지 않는다. 따라서, Type 0 유형은 단지 SCD 의 유형들을 분류하는데 필요한 하나의 참고 유형으로서 밖에 의미를 가지지 않는다.

둘째 유형은 기존의 속성값을 새로운 값으로 덮어쓰는 것이다 (Type 1). 따라서, 과거의 속성 변경 이력을 추적할 수 없고 항상 가장 최근의 속성값 기준으로 팩트를 집계 및 조회할 수 있다. 이 때, 차원 테이블의 주)대리키 값은 변화가 없으므로 해당 차원 테이블을 참조하는 팩트 테이블은 따로 업데이트가 필요 없다. 그림 1 의 예제를 보면 상품의 담당부서가 교육팀에서 전략팀으로 바뀌었다. 따라서, 상품과 관련된 과거의 모든 실적은 전략팀 기준으로 밖에 볼 수 없다. 더 이상 교육팀 기준으로는 볼 수 없다. 참고로, 키로 사용하는 상품대리키 값은 변화가 없다.

**변경 전**

| 상품대리키 | 상품ID | 상품명 | 담당부서 | 최초등록일 | 최종수정일 |
|---|---|---|---|---|---|
| 1 | ABC001 | 영재교육 | 교육팀 | 2014-01-01 | 2014-01-01 |

**변경 후**

| 상품대리키 | 상품ID | 상품명 | 담당부서 | 최초등록일 | 최종수정일 |
|---|---|---|---|---|---|
| 1 | ABC001 | 영재교육 | 전략팀 | 2014-01-01 | 2014-05-01 |

그림 1. SCD Type 1 예제

셋째 유형은 기존의 속성값은 유지하고 새로운 행을 추가하는 것이다 (Type 2). 이 방식은 과거의 속성 이력을 모두 추적할 수 있고, 각 시점의 속성값 기준으로 팩트를 집계 및 조회할 수 있다. 이 때, 과거 속성값이 속한 행의 대리키 값은 변화가 없으므로 해당 차원 테이블을 참조하는 팩트 테이블은 따로 업데이트가 필요 없다. 그러나, 변경이 빈번하게 발생하면 차원 테이블의 행이 너무 많아진다는 단점이 있다.

**변경 전**

| 상품대리키 | 상품ID | 상품명 | 담당부서 | 시작일자 | 종료일자 | 현재여부 |
|---|---|---|---|---|---|---|
| 1 | ABC001 | 영재교육 | 교육팀 | 2014-01-01 | 9999-12-31 | Y |

**변경 후**

| 상품대리키 | 상품ID | 상품명 | 담당부서 | 시작일자 | 종료일자 | 현재여부 |
|---|---|---|---|---|---|---|
| 1 | ABC001 | 영재교육 | 교육팀 | 2014-01-01 | 2014-04-30 | |
| 2 | ABC001 | 영재교육 | 전략팀 | 2014-05-01 | 9999-12-31 | Y |

그림 2-1. SCD Type 2 예제

때로는 Type 1 과 Type 2 를 중복 적용하기도 한다. 예를 들어, 영재교육을 단편으로 출시했다가 시리즈로 내기로 계획을 변경했다면, 기존의 상품명 영재교육을 일괄적으로 영재교육 I 으로 변경하도록 설계할 수 있다.

**변경 전**

| 상품대리키 | 상품ID | 상품명 | 담당부서 | 시작일자 | 종료일자 | 현재여부 |
|---|---|---|---|---|---|---|
| 1 | ABC001 | 영재교육 | 교육팀 | 2014-01-01 | 9999-12-31 | Y |

**변경 후**

| 상품대리키 | 상품ID | 상품명 | 담당부서 | 시작일자 | 종료일자 | 현재여부 |
|---|---|---|---|---|---|---|
| 1 | ABC001 | 영재교육 I | 교육팀 | 2014-01-01 | 2014-04-30 | |
| 2 | ABC001 | 영재교육 I | 전략팀 | 2014-05-01 | 9999-12-31 | Y |

그림 2-2. SCD Type 1 + 2 예제

넷째 유형은 속성 변경값들을 저장할 공간으로 여러 개의 속성들을 추가하여 설계하는 것이다 (Type 3). 장점은 과거 속성값과 현재 속성값 기준으로 조합하여 분석할 수 있다는 것이다. 특히, 단순 조합이 아닌 속성값간 연관관계가 의미 있을 때 보다 더 의미 있는 분석을 할 수 있다. 예를 들어, 대부 회사의 경우 고객의 신용등급이 중요한데 최초등급, 이전등급, 현재등급 등을 조합하여 등급변경에 따른 연체율의 상관관계를 볼 수 있다. 이 때 대리키 값은 변동이 없으므로 팩트 테이블은 따로 업데이트가 필요 없다. 단점으로는 속성 변경 이력을 무한정 반영할 수 없다는 것이다. 당연히 과거의 모든 이력을 기록하는 것도 어렵다. 따라서, 가능한 변경 빈도가 작고 예측 가능한 경우 또는 특정 의미 있는 변경 이력만 반영하는 경우에 적용하는 것이 좋다.

**변경 전**

| 고객대리키 | 고객ID | 고객명 | 최초등급 | 등급 | 직전등급 | 변경일 |
|---|---|---|---|---|---|---|
| 1 | M001 | 고길동 | A | A | A | 2014-01-01 |

**변경 후**

| 고객대리키 | 고객ID | 고객명 | 최초등급 | 등급 | 직전등급 | 변경일 |
|---|---|---|---|---|---|---|
| 1 | M001 | 고길동 | A | B | A | 2014-03-01 |

그림 3. SCD Type 3 예제

다섯째 유형은 변경이 발생하는 속성들을 모아서 별도의 차원(mini-dimension) 으로 독립시키는 것이다 (Type 4). 차원 테이블이 매우 크고 변경이 빈번하게 발생하면 Type 2 의 방법도 크기, 성능, 관리 측면에서 문제가 생길 수 있다. 이런 경우, 자주 분석하거나 빈번하게 변경되는 속성들을 모아서 별도의 차원으로 분리시키는 것을 고려할 수 있다. 그림 4 는 고객 차원에서 일부 속성들을 모아서 별도의 미니-차원으로 분리시킨 예제이다. 이때 미니-차원과 고객 차원 테이블은 독립적으로 팩트 테이블과 연결되므로 고객별 인구통계정보의 변경 내역(예를 들어, 연령대 변경)을 팩트 테이블에서 함께 관리하는 부수적 효과도 기대할 수 있다.

| 인구통계ID | 연령대 | 신용등급 | 연소득 |
|---|---|---|---|
| 1 | 21~25 | C | ~1999 |
| 2 | 21~25 | C | 2000~3999 |
| 3 | 21~25 | C | 4000~ |
| 4 | 21~25 | B | ~1999 |
| 5 | 21~25 | B | 2000~3999 |
| 6 | 21~25 | B | 4000~ |
| ... | ... | ... | ... |
| 10 | 26~30 | C | ~1999 |
| 11 | 26~30 | C | 2000~3999 |
| 12 | 26~30 | C | 4000~ |
| ... | ... | ... | ... |

그림 4. SCD Type 4 예제

이 방식의 단점은 속성값들의 가능한 조합수가 너무 많으면 분리한 효과가 없다는 것이다. (물론, 실제로 존재 가능한 조합만 데이터를 생성하는 방법도 고려할 수 있다.) 따라서, 연소득처럼 속성값이 연속인 것은 미리 적절하게 구간대를 정의하여 가능한 조합수를 줄이는 것이 좋다. 만약에 상세한 연소득 값을 따로 관리해야 한다면 연결된 팩트 테이블에서 함께 관리하는 것을 고려할 수 있다. 또한, 연소득 구간처럼 사전에 설정한 구간대를 변경하고자 하면 팩트 테이블의 기존 정보를 업데이트해야 하는 일이 발생할 수 있어 사전에 구간 설정시 신중히 고려해야 한다. 팁으로 원래의 일차 차원 테이블에서 현재 상태에 해당하는 미니-차원의 키를 외래키로 참조할 수도 있다. 예를 들어, 고객 차원 테이블에 현재의 인구통계 ID 를 추가하는 것이다. 그림 5 는 미니-차원 추가를 활용한 설계 예제이다.

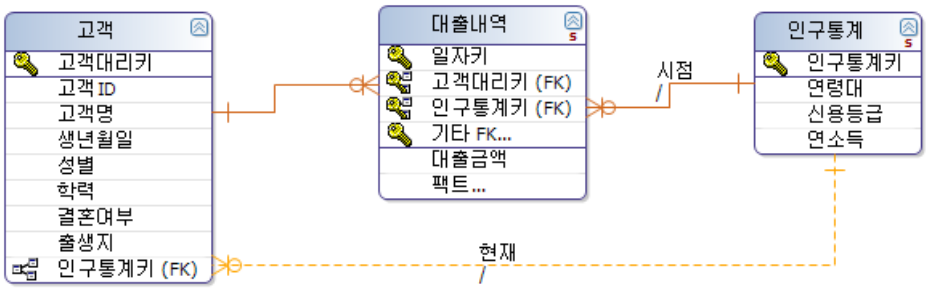



그림 5. 미니-차원 설계 예제

이상으로 기본적인 SCD 관리 유형들에 대하여 살펴 보았다. 다음 기사에서는 그러한 기본 유형들을 혼합한 하이브리드 형태들을 살펴 보겠다.

---

주) 대리키(Surrogate Key)는 비즈니스 업무에서 사용하는 자연키(Natural Key)에 대응해서 임의로 채번하는 키로서 보통 의미 없는 정수형 일련번호를 사용한다. 주로 하나의 팩트 테이블을

중심으로 주변에 관계를 가지는 여러 개의 차원 테이블이 나타나는 스타스키마 또는 스노우플레이크 스키마 형태 설계에서 많이 활용된다. 그림 6 에서 고객 ID 에 대응하는 고객대리키, 상품 ID 에 대응하는 상품대리키가 대리키에 해당한다. 장점으로는 최악의 경우 자연키가 바뀌더라도 연결된 팩트 테이블 (예를 들어, 주문실적)은 대리키를 참조하고 있으므로 기존 데이터가 영향을 받지 않는다는 것이다. 또한, 동일한 자연키에 대하여 여러 개의 대리키를 가져갈 수 있으므로 SCD Type2 와 같은 속성의 이력관리 전략에 효과적이다. 덤으로 대개의 자연키는 문자열인 경우가 많은데 크기가 작은 정수형을 사용해서 저장 공간 절약 및 성능 향상도 기대할 수 있다.


지난 SCD 설계 접근 (기본 유형) 기사에서는 시간이 지남에 따라 속성값들이 변경되는 경우의 이력 관리 전략의 기본 유형들에 대하여 살펴보았다. 이번 기사에서는 이어서 하이브리드 유형들을 살펴보도록 하겠다. (사실 일부는 이미 이전 기사 내용에 포함되어 있다.)

여섯째 유형은 Type 4 와 Type 1 을 결합하는 방식이다 (Type 5). 그림 7 은 이전 기사에서 소개했던 Type 4 의 미니-차원 설계 예제이다. 인구통계가 미니-차원이고 고객은 인구통계의 일차 차원이다. 여기에서 고객의 인구통계키는 현재의 고객 상태 (연령대, 신용등급, 연소득)를 나타낸다. 즉, Type 1 에 해당한다. 따라서, 전체적으로 Type 4 + Type 1 에 해당하는 방식이다.

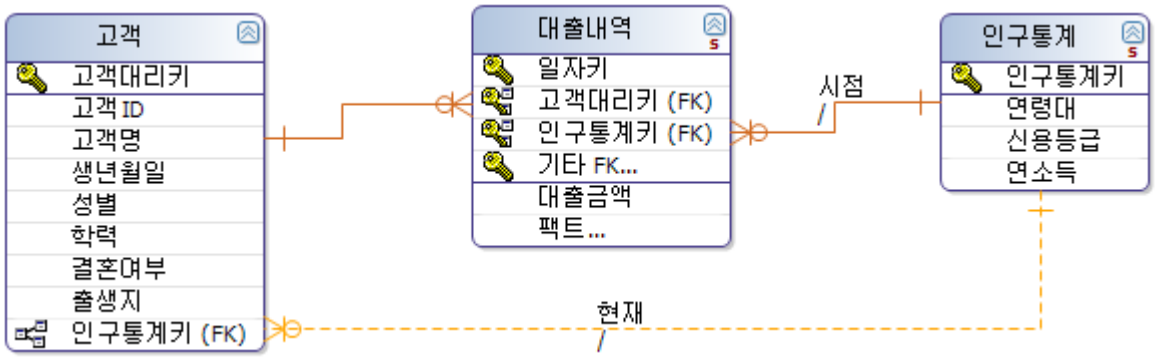



그림 7. 미니-차원 설계 예제

예제를 보면 고객의 현재 상태(인구통계)를 팩트 테이블 (대출내역) 없이 카운트하거나, 현재 상태 기준으로 팩트를 집계할 때 효과적이다. 그런데, Presentation 영역으로서의 DW 또는 마트는 최종 사용자를 최대한 배려해서 설계해야 한다. 예를 들어, 그림 8-1 과 같이 대출내역의 팩트들을 접근하고 분석할 때 시점의 고객 상태와 현재의 고객 상태를 명확하게 구분해서 접근할 수 있도록 함으로써 혼동이 발생하지 않도록 해야 한다. 예제에서 현재인구통계는 인구통계를 이용하여 뷰를 만드는 것을 고려할 수 있다. 이 때, 현재인구통계를 Type 1 Outrigger 라고 한다. 한걸음 더 나아가, 그림 8-2 와 같이 최종사용자에게 더 간단한 형태로 제공되도록 고려할 수 있다. 이때 역시 뷰를 활용하면 효과적이다.

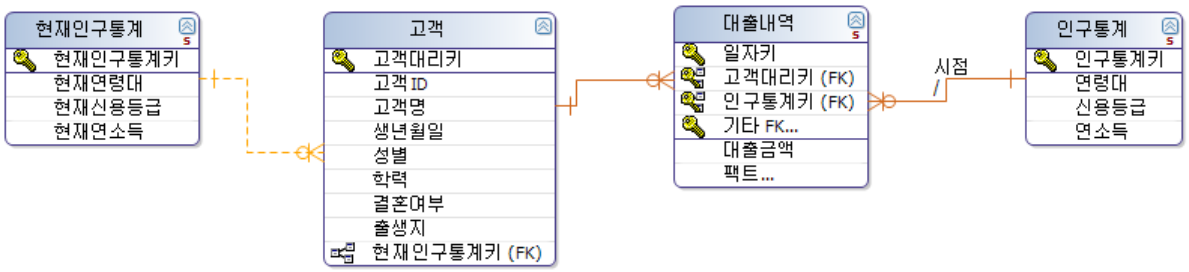



그림 8-1. SCD Type 5 설계 예제 (1)

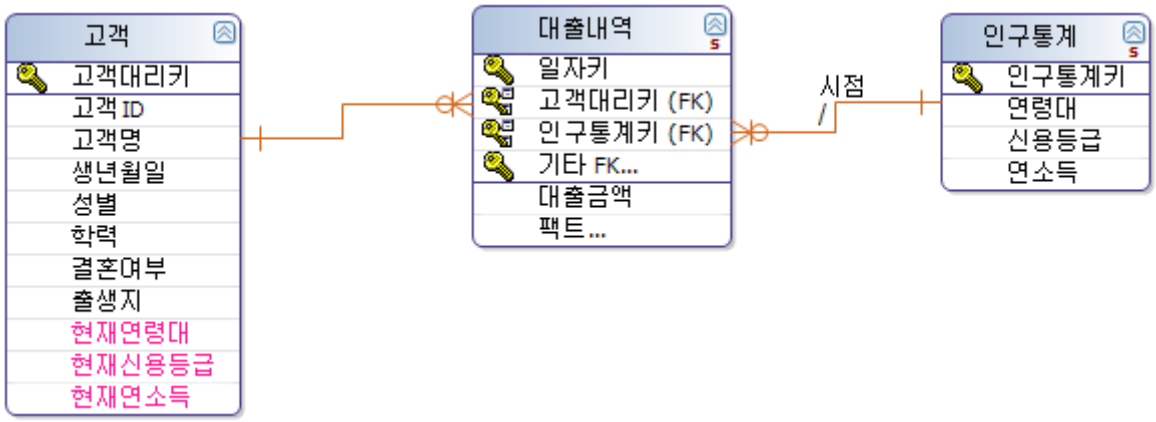



그림 8-2. SCD Type 5 설계 예제 (2)

일곱째 유형은 Type 1 의 속성을 Type 2 차원에 추가하는 것이다 (Type 6). 이미 이전 기사에도 관련 내용이 포함되어 있다. 여기서는 살짝 예제 시나리오를 바꿔서 더 살펴보겠다. 그림 9 를 보면 시점부서, 시작일자, 종료일자, 현재여부 등은 Type 2 방식을 위한 속성들임을 쉽게 알 수 있다. 그리고, 현재부서는 Type 1 방식을 위한 속성이라는 것도 쉽게 알 수 있다. 여기에 더해 부서명을 관리하기 위해 하나의 부서명을 더 추가해서 2 개의 부서명 (현재부서, 시점부서)을 속성을 관리하는 것은 Type 3 에 해당한다. 결과적으로 Type 6 방식은 Type 1 + Type 2 + Type 3 에 해당한다.

변경 전

| 상품대리키 | 상품ID | 상품명 | 현재부서 | 시점부서 | 시작일자 | 종료일자 | 현재여부 |
|---|---|---|---|---|---|---|---|
| 1 | ABC001 | 영재교육 | 교육팀 | 교육팀 | 2014-01-01 | 9999-12-31 | Y |

1차 변경 후 : 교육팀 → 전략팀

| 상품대리키 | 상품ID | 상품명 | 현재부서 | 시점부서 | 시작일자 | 종료일자 | 현재여부 |
|---|---|---|---|---|---|---|---|
| 1 | ABC001 | 영재교육 | 전략팀 | 교육팀 | 2014-01-01 | 2014-04-30 | N |
| 2 | ABC001 | 영재교육 | 전략팀 | 전략팀 | 2014-05-01 | 9999-12-31 | Y |

2차 변경 후 : 전략팀 → 기획팀

| 상품대리키 | 상품ID | 상품명 | 현재부서 | 시점부서 | 시작일자 | 종료일자 | 현재여부 |
|---|---|---|---|---|---|---|---|
| 1 | ABC001 | 영재교육 | 기획팀 | 교육팀 | 2014-01-01 | 2014-04-30 | N |
| 2 | ABC001 | 영재교육 | 기획팀 | 전략팀 | 2014-05-01 | 2014-09-30 | N |
| 3 | ABC001 | 영재교육 | 기획팀 | 기획팀 | 2014-10-01 | 9999-12-31 | Y |

그림 9. SCD Type 6 예제

예제는 영재교육이라는 상품을 담당하는 부서가 교육팀 à 전략팀 à 기획팀으로 바뀌는 이력 관리를 나타내고 있다. 연결된 팩트 테이블에는 데이터가 각각의 이력 구간에 해당하는 상품대리키로 적재되기 때문에 변경 발생에 따른 기존 데이터의 업데이트가 필요 없다. 최종 사용자는 언제든지 과거 모든 담당 부서별 실적을 볼 수도 있고, 현재 담당 부서로 모두 귀속시킨 전체 실적도 볼 수 있다.

여덟째 유형은 Type 2 방식에 더해서 팩트 테이블에서 차원 테이블의 자연키도 참조하는 방식이다 (Type 7). 이 때 자연키는 영구불변인 키를 사용한다 (수퍼자연키). 만약에 미래에 자연키 자체가 변경가능성이 있으면 매핑가능한 다른 키를 사용한다. (예를 들어, 해당 상품의 첫번째 대리키를 수퍼자연키로 활용할 수 있다.) 그림 10-1 을 보면 주문실적 테이블에서 상품 테이블을 상품대리키와 상품 ID 로 이중 참조하고 있다.

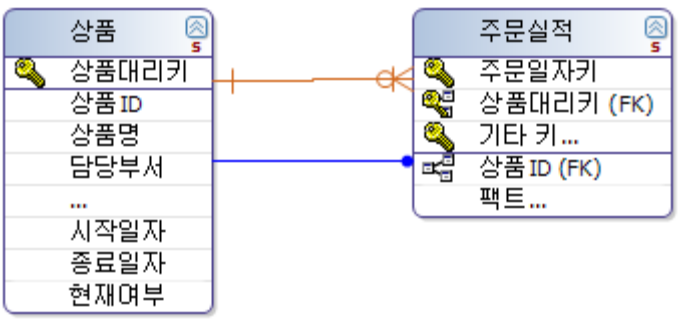



그림 10-1. SCD Type 7 예제 (1)

이전 기사에서 다뤘듯이 Type 2 방식은 과거의 속성 이력을 모두 관리 가능하며 시점별로 유효한 속성값 기준으로 팩트에 대한 집계가 가능하다. 예를 들어, 예제에서 상품과 주문실적 테이블간 상품대리키를 이용해서 조인한 후 상품대리키 기준으로 집계하면 된다. 여기에 더해서, 상품테이블의 현재여부가 'Y' 라는 조건과 함께 자연키(상품 ID)를 이용한 조인을 하면, 과거 모든 실적을 현재 기준으로 귀속 시킨 결과를 얻는다. 즉, 결과적으로 Type 6 와 같은 효과를 볼 수 있다. 차이점이라면 Type 6 에서는 변경을 관리할 속성이 많아지면 적용하기 어려운 문제를 해결했다는 것이다. 즉, Type 7 은 Type 6 에서 추가 속성이 많아질 때의 문제를 팩트 테이블에 참조 키를 하나 더 가져가는 방식을 취해서 해결한 것으로 이해하면 된다.

그런데, 단순히 문제 해결에 그치지 않고 최종 사용자를 위한 배려를 여전히 잊지 말아야 한다. 최종 사용자가 모델러가 설계한 것을 제대로 이해하고 알아서 잘 활용할 것이라 기대하면 안 된다. 따라서, 그림 10-1 을 10-2 와 같이 보다 직관적으로 명확하게 이해할 수 있도록 풀어서 설계하는 것을 고려할 수 있다. 이 때, 현재상품은 상품에서 현재여부가 'Y'인 것만 추출하는 뷰를 활용할 수 있다.

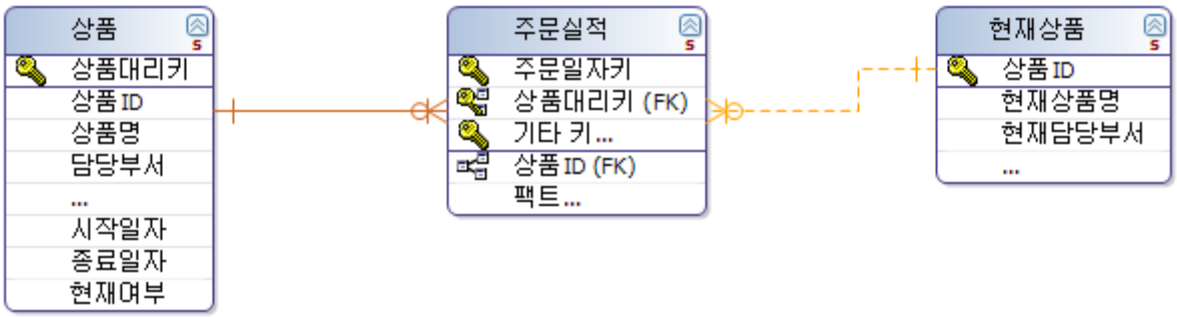



그림 10-2. SCD Type 7 예제 (2)

아마도 가장 난감한 요구 중에 하나가 조회 시점에 임의의 과거 시점 기준으로 모든 실적을 귀속시켜 파악하고자 하는 경우일 것이다. 예를 들어, 현재가 2014 년 말인데 2013 년 말의 상품 기준으로 실적으로 보고 싶은 것이다. Type 7 방식은 이러한 요구에 대한 훌륭한 해법이 될 수 있다. 우선, 상품 테이블을 대상으로 시작일자와 종료일자를 이용하여 2013 년 말의 상품 내역을 뽑아낸다. 그리고, 그 결과를 상품 ID 를 이용하여 주문실적과 조인 처리한다. 적어도 SQL 을 이용해서 원하는 것을 제공할 수 있는 것이다. 그러나 여전히 원하는 결과를 쉽게 얻을 순 없다. 예를 들어, 2014 년 신상품의 주문 실적은 2013 년말 상품들에 귀속되지 않으므로 일반적인 미매핑과 구분해서 별도의 예외 처리를 해야 할지 말아야 할지 고민해야 한다. 아마도 가장 큰 애로사항은 DB 의 물리적인 스키마 내지는 중간의 데이터모델과 강하게 맞물려 있어 커스터마이징이 어려운 일반 범용 클라이언트 툴들을 이용하는 경우일 것이다. 따라서, 최종 사용자의 역량을 고려해서 기능들을 제한적으로 제공하는 것도 필요하다. 잘못하면 개발의 편자가 될 수 있기 때문이다.

이상으로 2 회에 걸쳐 SCD 의 관리 방식에 대해서 8 가지 유형을 살펴 보았다. Type 0 는 형식상의 구분이고 Type 1~4 가 실질적인 기본 유형이다. Type 5~7 은 기본 유형들을 조합해서

응용하는 하이브리드 유형이다. 응용하기에 따라서 하이브리드 유형은 더 나올 수도 있을 것이지만 대부분의 비즈니스 요구들은 기사에서 다룬 유형들 내에서 해결할 수 있을 것으로 본다.

참고로 또 다른 속성 변경 관리 대상으로 RCD(Rapidly Changing Dimension)가 있는데 해결 방식은 간단하다. 너무 빈번하게 속성 변경이 발생하면 Type 2 를 방식으로 해도 테이블이 너무 커지므로 효과적이지 않게 된다. 이 경우, 단순히 해당 속성을 별도의 독립된 차원으로 분리하는 것을 고려할 수 있다. 하나의 속성으로 구성된 미니-차원(Type 4)으로 이해해도 된다. 이 때, 실적 이벤트가 발생시 팩트 테이블에 해당 시점의 속성값을 함께 기록함으로써 이력 관리 효과를 본다. 다만, 실적이 없는 경우는 이력 단절이 생길 수 있으므로 관리용으로 속성 이력 테이블을 별도 고려할 수 있다.


이전 *DW 설계 접근 방식의 이해* 기사에서 Presentation 영역의 의미를 설명했다. 기억을 새로이 하는 차원에서 다시 한번 적어보겠다.

*Presentation 영역은 최종 사용자, 리포팅 도구, 분석 도구, 각종 어플리케이션 등에서 직접 쿼리가 가능하도록 데이터를 가다듬어 저장하는 곳을 의미한다.*

즉, Presentation 영역은 중간 데이터 처리가 아니라 데이터 소비자들이 원활히 조회 및 분석할 수 있도록 데이터 처리가 끝난 결과물의 저장소인 것이다. 여기서 중요한 것은 데이터 처리가 아닌 데이터 소비자를 위한 설계를 고려해야 한다는 것이다. 예를 들어, 최종 사용자가 SQL 을 이용하여 직접 데이터에 접근 시 설계한 내용을 따로 가르쳐주지 않아도 데이터를 혼동하지 않고 활용할 수 있도록 배려해야 한다. 무엇이 분석 대상이고 어떤 경로로 어떤 기준으로 접근 및 분석할 수 있는지 직관적으로 알 수 있어야 한다. 더 나아가 성능까지 보장하면 금상첨화이다. 이 때, 그러한 목적을 고려해서 보통 차원 테이블이 비정규화된 스타스키마 형태의 설계를 하게 된다. 물론, 사용하는 DBMS 특징과 BI 도구들에 따라 스노우플레이크 스키마 형태도 무난한 경우가 있다. 예를 들어, 전통적인 OLAP 도구들을 사용하면 최종 사용자가 관계형 테이블을 직접 바라보는 것이 아니라 별도로 설계된 OLAP 모델을 바라보는 것이기 때문에 별문제가 되지 않을

수 있다. 그럼에도 불구하고 최종 사용자가 언제든지 직접 접근할 수 있다는 것을 함께 고려하는 것이 좋다.

그런데, 모델러 스스로가 이러한 비정규화된 스타스키마 형태의 설계에 부정적인 시각을 갖는 경우가 많다. 특히, 트랜잭션 중심의 기간계 설계를 많이 했던 모델러들이 더욱 꺼리는 경향이 있다. 데이터의 중복에 따른 데이터 정합성 보장과 유지보수 측면에 대한 우려 때문이다. 이번 기사를 통해 그러한 우려를 줄여보는 계기가 되었으면 하고, 설사 여전히 동의하기 어렵더라도 사용자 중심의 설계를 충분히 고려해야 한다는 마인드를 상기하는 기회가 되었으면 한다. 읽는 내내 사용자 편이성 제공과 성능 보장을 염두에 두면 좋을 것이다.

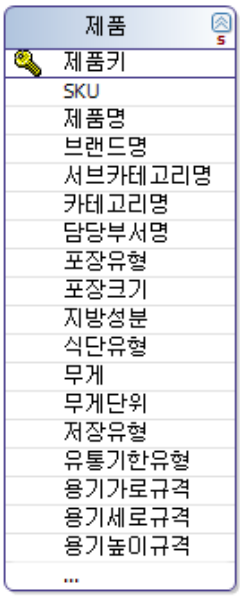



그림 1. 제품 차원

그림 1 은 제품이라는 차원 테이블로서 비정규화된 형태이다. 정규화에 익숙한 사람이라면 손이 근질근질할 것이다. 우선 반복되는 값들이 눈에 들어온다. 예를 들어, 제품이 30 만개이고 담당부서가 50 개라면 하나의 부서가 평균 6000 번 정도 반복해서 나타날 것이다. 담당부서키가 2byte 이고 담당부서명이 20byte 라면 부서라는 테이블을 만들면서 1 차 정규화를 하면, 담당부서키만 관리함으로써 꽤 많은 공간을 절약할 것이라 생각이 들 것이다. 또한 담당부서명이 변경되는 경우 평균 6000 건 정도 업데이트를 하지 않고 단 한 건만 업데이트함으로써 데이터 정합성 보장에도 유리하다. 그것이 정규화의 장점이니까 당연한 얘기다.

그럼 이제 정규화한 경우를 살펴보자. 그림 2 는 그림 1 의 제품 차원을 정규화한 모델이다. (차원 테이블을 정규화한 경우 스노우플레이크 스키마 형태가 된다.) 스노우플레이크 스키마도 정상적인 다차원모델링 기법 중에 하나이기 때문에 맞다 틀리다 논쟁하는 것은 적절치 못하다. 사실 어쩔 수 없이 부분적으로라도 스노우플레이크 스키마 형태가 나오는 경우가 많다. 그럼에도 불구하고, 최종 사용자의 사용 편이성과 성능 보장이라는 측면을 고려했을 때, 가능하면 스타스키마 형태를 지향하는 것을 권고한다. 왜 그런지 이유를 살펴보자.

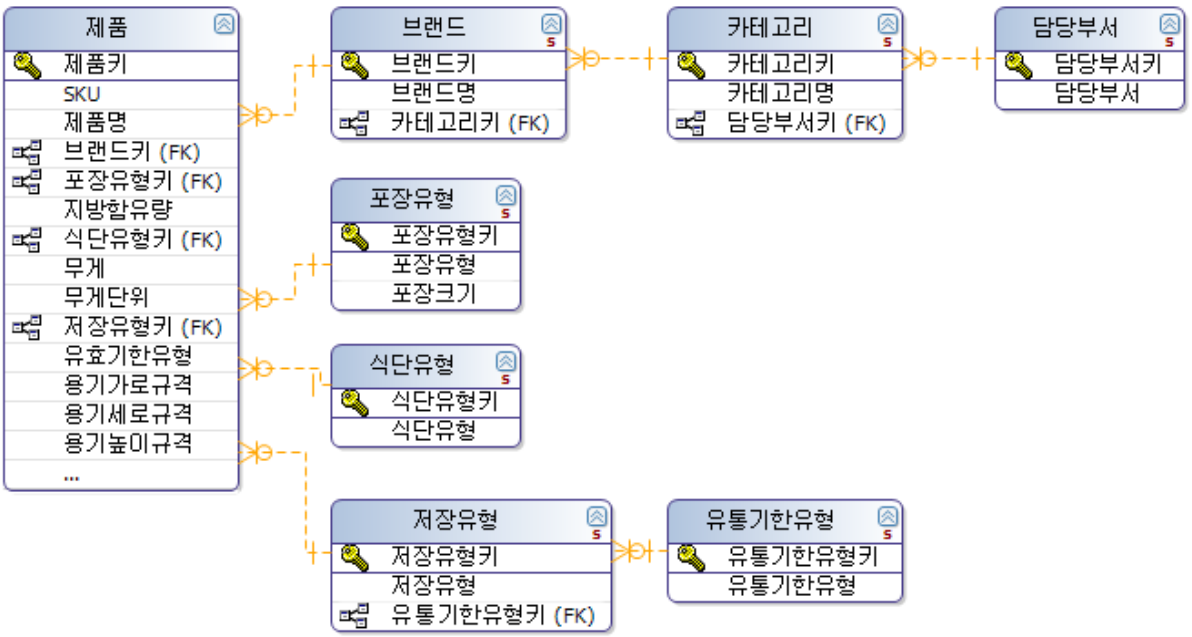



그림 2. 정규화한 제품 차원 (스노우플레이크)

첫째, 차원 정규화를 통한 디스크 절약은 대개 전체에서 1% 미만으로 미미한 수준이다. 예를 들어, 제품이 30 만개인 차원 테이블에서 정규화를 통해 20byte 인 담당부서명 대신 2byte 인 담당부서키 만을 관리할 때 5.4MB (30000 x 18 byte) 정도 절약할 수 있다. 데이터웨어하우스에서 흔하게 나타나는 10GB 이상의 팩트 테이블 등을 고려할 때 이는 매우 미미한 수준이다.

둘째, 정규화를 통한 데이터 정합성과 효율적인 유지보수는 여전히 의미가 있다. 그러나, 사용자의 편이성을 보장하기 위해 보다 전향적인 자세가 필요하다. 대안으로, 데이터웨어하우스는 소량의 데이터에 대한 빈번한 업데이트가 아닌 대량의 데이터를 처리하는 배치성 작업이 많으므로 ETL 과정을 통해서 데이터 정합성을 보장하는 것을 고려해야 한다.

셋째, 정규화를 많이 할수록 사용자는 훨씬 많은 선택을 해야 하고 혼동을 일으킬 수 있다. 그림 2 를 보면 제품 차원 하나만 정규화 했는데도 접근 경로가 4 가지 나왔다. DW Presentation 영역은 가능한 단순화하여 사용자 혼란을 줄이고 편이성을 배려하는 것이 무엇보다 중요하다.

넷째, 대부분의 DBMS 옵티마이저 관점에서 볼 때 스노우플레이크 스키마 형태는 복잡성 때문에 최적화에 불리하다. 상대적으로 더 많은 테이블과의 조인 처리를 고려해야 하므로, 쿼리 최적화 시간도 오래 걸리고 쿼리 최적화를 잘못할 확률도 높아진다.

다섯째, 하나의 차원에 속한 속성들은 서로 밀접한 상관 관계가 있는 경우가 많다. 예를 들어, 카테고리와 포장 유형을 선택하면 다른 속성의 선택 가능한 값들이 달라진다. 차원 탐색을 통해 사용자는 자연스럽게 이러한 속성 관계들을 파악하게 된다. 그러나, 정규화를 많이 하다 보면 논리적으로 하나의 차원에 속한 속성들이 여러 차원 테이블에 흩어지게 되므로 속성들간의

관계를 파악하는데 불리하다. 보다 많은 테이블에 대한 접근이 필요하고 모델을 잘 이해하지 못하면 엉뚱한 다른 차원의 속성까지 접근하려 할 수 있다.

여섯째, 스노우플레이크 스키마는 비트-맵 인덱스 사용을 어렵게 한다. 비트-맵 인덱스는 제품 차원 테이블의 담당부서나 카테고리처럼 카디널리티가 작은 칼럼에 매우 효과적인 것으로서, 해당 칼럼에 대한 제한 조건 처리에 매우 뛰어나다. 스노우플레이크 스키마는 그러한 성능 최적화를 기대하기 어렵게 한다.

이상으로 사용자 편이성과 성능 보장이라는 측면에서 DW Presentation 영역에서의 정규화에 대한 몇 가지 이슈를 살펴 보았다. 물론, 일부 DBMS 에서는 정규화된 차원 모델도 성능 저하 없이 효과적으로 지원한다고 주장한다. 그럼에도 불구하고 사용자에게는 비정규화한 단순한 형태를 제공하는 것이 좋다. 예를 들어, 관련 테이블들을 묶어서 뷰를 만들어 제공하는 것을 고려할 수 있다. 또는 OLAP 도구 등을 활용하는 경우 중간 OLAP 모델을 통해서 보다 단순화된 모습으로 제공하는 것이 바람직하다.

지금까지 가능하면 차원 테이블을 정규화하지 않은 스타스키마 형태를 권고하였는데 예외적으로 Outrigger 차원을 연결하는 형태를 고려할 수도 있다. 그림 3 은 제품 차원과 제품출시일자라는 Outrigger 차원을 연결한 모델이다. 이 때, 제품출시일자라는 이름과 같이 다른 일자들과 명확하게 구분되는 차원 이름과 속성 이름들을 제공하여 사용자 혼란을 방지하는 것이 필요하다.

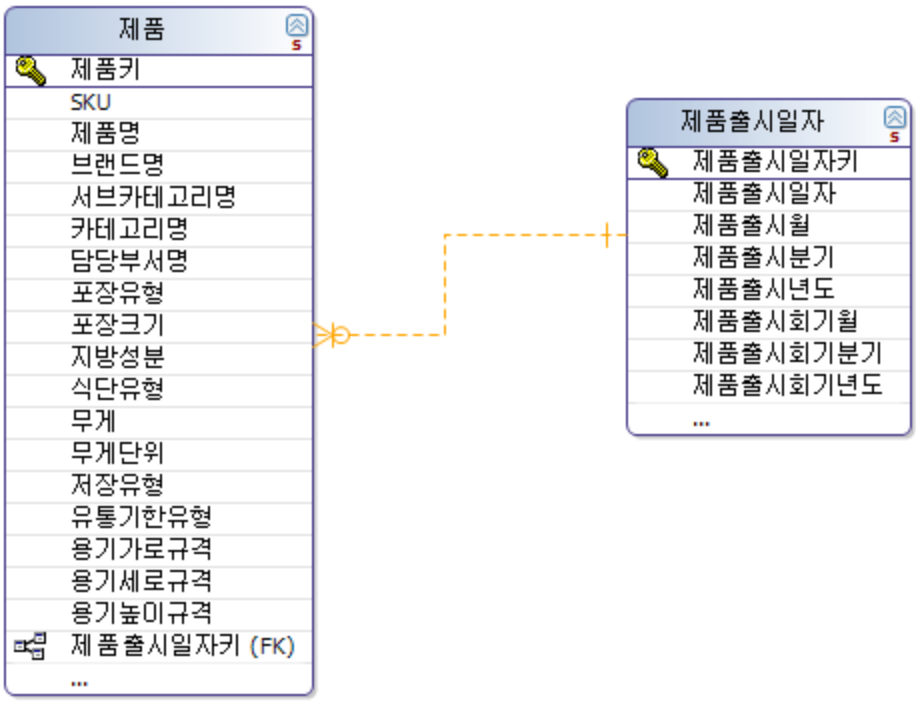



그림 3. 제품 차원에 Outrigger(제품출시일자) 차원 연결

마지막으로 하나 더 살펴 보면 차원 정규화는 하고 싶은데 스노우플레이크 스키마 형태는 피하고 싶은 유혹을 느끼는 경우이다. 이런 경우, 정규화하고 싶은 속성들을 모두 별도의 차원으로 독립시키고 팩트 테이블에서 참조하도록 설계하는 유혹을 느낀다. 결국 차원 테이블은 정규화하고 팩트 테이블을 비정규화한 꼴이 된다. 그림 4 가 그러한 예로서 마치 지네 발처럼 팩트 테이블에 매우 많은 차원 테이블이 연결되어 지네형 팩트 테이블이라 한다.

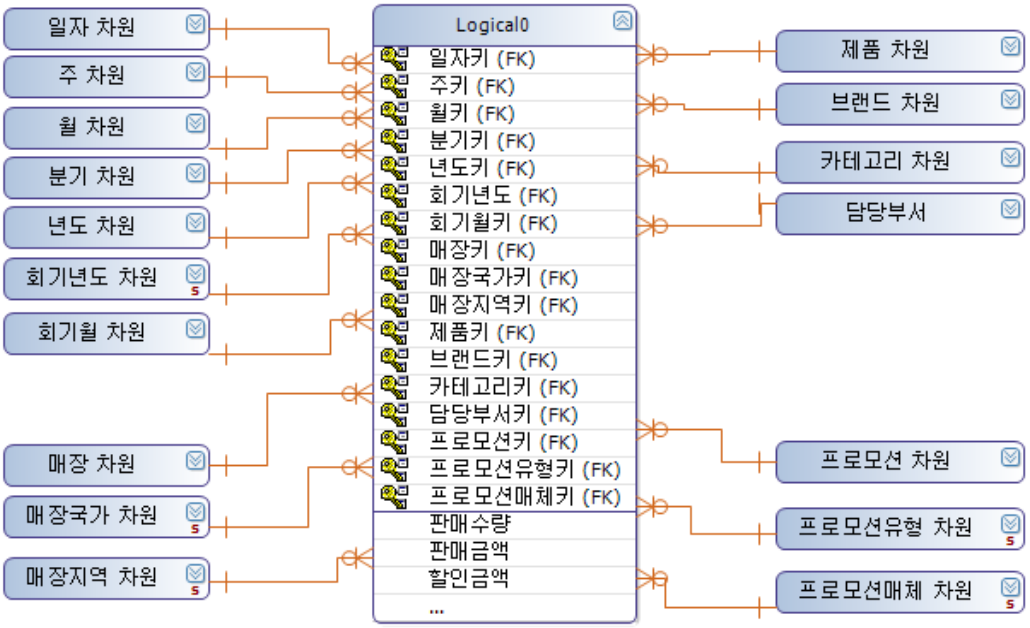



그림 4. 지네형 팩트 테이블

그러나, 지네형 팩트 테이블은 여러모로 소탐대실이라 하겠다. 우선 팩트 테이블에 칼럼이 너무 많아짐으로써 디스크 소모량이 급격히 증가할 수 있다. 많은 조인들을 고려하여 충분한 단일 칼럼 또는 복합칼럼 인덱스를 생성 및 관리하는 것도 어렵다. 너무 많은 조인은 활용 측면에서도 부정적인 영향을 줄 뿐만 아니라 옵티마이저가 쿼리를 최적화하는 것에도 많은 제약이 된다. 또한 너무 많은 차원은 사용자를 혼란하게 만들며 무엇보다도 논리적으로 하나의 차원에 속하는 많은 속성들의 관계를 파악하기도 어렵다.

물론, 최근에 각광 받고 있는 칼럼 저장형 데이터베이스를 사용하면 쿼리 및 디스크 소모와 관련된 이슈를 상당 부분 감소 시킬 수 있을 것이다. 그러나, 그것만으로 이러한 지네형 모델링을 하는 것은 명분이 부족하다 하겠다.

그런데, 예외적으로 지네형의 모델링을 부분적으로 고려 해야 하는 경우도 있다. 예를 들어서, 속성값들이 시간이 지남에 따라 변경되는 경우 발생 시점 기준으로 분석하고 싶은 경우이다. 예를 들어서, 담당부서가 작년에는 A 부서였는데 올해는 B 부서로 바뀌었을 때 작년 실적은 A 부서로 올해 실적은 B 부서로 보고 싶은 경우이다. 이런 경우 시점담당부서라는 차원을 별도로 만들어 팩트 테이블과 바로 연결하는 것이 좋은 해결책이 될 수 있다. 물론, 일차 차원 테이블인 제품 테이블에는 현재담당부서를 관리함으로써 언제든지 현재 부서 기준으로도 실적을 볼 수

있다.
이상으로 DW Presentation 영역에서 차원 테이블의 정규화 유혹에 대한 이슈와 왜 비정규화된 스타스키마 형태를 권고하는지를 살펴보았다. 다시 한번 강조하자면 전반적인 내용을 이해함에 있어 최종 사용자에 대한 사용 편이성과 성능 보장을 항상 염두에 두길 바란다.


일반적으로 최종 사용자, OLAP 도구 또는 분석 도구 등의 직접적인 접근 대상이 되는 DW Presentation 영역 설계에서 자주 나타나는 스타 스키마 모델에서 차원과 팩트 테이블 관계는 일대다 관계이다. 즉, 팩트 테이블의 데이터 한 건은 차원 테이블의 한 건만을 정확히 참조한다. 그러나, 실세계에서는 팩트 테이블의 하나의 행이 차원 테이블의 여러 행과 관계를 가지는 경우도 있다 (여기서는 논리적인 관점에서 이해하자). 결과적으로 다대다 관계가 된다. 예를 들어, 하나의 주문실적이 여러 명의 담당 사원들과 관련될 수 있다. 또한, 고객의 취미가 여러 개인 경우도 있다. 이런 경우, 담당 사원이 김성희와 조혜리인 주문실적을 보고 싶다거나, 취미가 바둑과 독서인 고객이 구매한 실적을 보고 싶을 수 있다. 이제 이러한 요구를 해결하기 위한 설계 방식에 대해 살펴 보겠다.

먼저, 팩트 테이블의 하나의 행이 차원 테이블의 여러 행과 관계를 가지는 경우를 살펴보겠다 (다중값 차원, Multi-valued Dimension). 우선 그림 1 과 같이 팩트 테이블과 차원 테이블의 관계를 복수 개로 설정하는 단순화 방식을 고려할 수 있다. 예제는 주문실적과 판매담당 사이의 다대다 관계를 여러 개의 일대다 관계로 변환시킨 것으로 이해하면 된다.

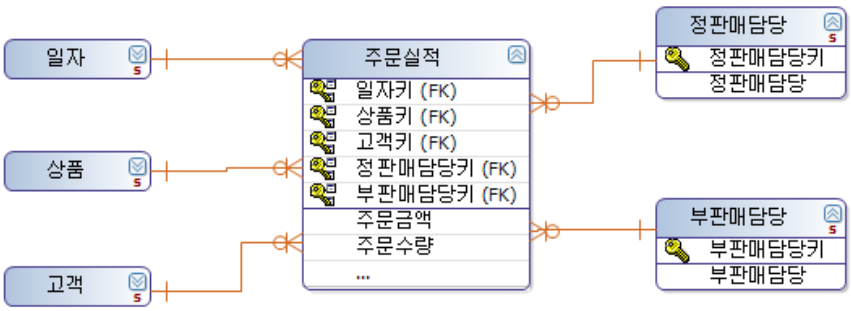
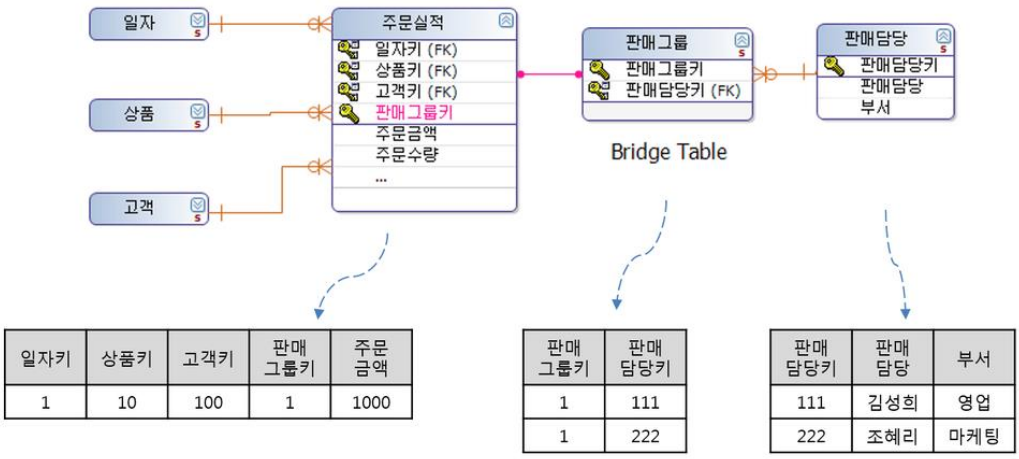



그림 1. 다중값 차원 설계 (1)

이 방식은 단순한 것이 장점이다. 예를 들어, 두 명의 정/부판매담당 사이의 조합별로 주문실적을 자유롭게 볼 수 있음을 직관적으로 알 수 있다. 따라서 BI 도구들을 이용하여 쉽게 접근 가능하다.

그러나, 한계점 또한 명확하다. 우선 고정된 수의 관계일 때만 가능하다. 고정된 수라도 관계가 너무 많으면 적용하기 어렵다. 예제에서 실적이 상위 10 위 안의 판매담당사원을 구하려면 주문실적과 정판매담당, 주문실적과 부판매담당을 따로 조인 처리하여 각각의 결과를 합친 후 2 차 처리를 해야 한다. 단순히 특정 담당사원의 실적을 보고자 하는 경우에도 같은 사원이 주문에 따라 정판매담당도 되었다가 부판매담당도 될 수 있기 때문에 역시 주문실적과 정판매담당, 주문실적과 부판매담당을 따로 조인 처리하여 각각의 결과를 합친 후 2 차 처리를 해야 한다.

이러한 애로사항을 해결하기 위하여 브릿지 테이블을 이용할 수 있다. 그림 2 에서 판매그룹이 브릿지 테이블이다. 판매그룹키는 구분되는 판매담당 조합별로 유일하게 생성한다. 예제에서 판매그룹키가 1 인 그룹은 김성희와 조혜리라는 담당사원으로 구성된 그룹이다. 또한, 판매담당이 김성희 또는 조혜리 1 명인 경우도 각각 별도의 판매그룹키를 생성한다. (판매그룹키는 ETL 과 같은 별도의 과정을 통해 생성 및 관리한다.) 팩트 테이블인 주문실적에는 판매그룹키를 추가 관리한다.




| 일자키 | 상품키 | 고객키 | 판매 그룹키 | 주문 금액 |
|---|---|---|---|---|
| 1 | 10 | 100 | 1 | 1000 |

| 판매 그룹키 | 판매 담당키 |
|---|---|
| 1 | 111 |
| 1 | 222 |

| 판매 담당키 | 판매 담당 | 부서 |
|---|---|---|
| 111 | 김성희 | 영업 |
| 222 | 조혜리 | 마케팅 |

그림 2. 다중값 차원 설계 (2)

브릿지 테이블을 이용하는 방식은 앞에서의 단순화 방식이 가진 단점을 극복하고 매우 유연하며 다양한 분석이 가능하다. 그러나, 그 만큼 비용과 위험도 커서 신중하게 적용해야 한다.

예제에서 김성희와 조혜리의 개별 주문 실적은 세 테이블을 조인하여 쉽게 구할 수 있다. 그러나, 김성희와 조혜리가 함께 담당하는 전체 주문금액을 구하는 경우는 중복 계수 문제가 발생하여 2000 이 나온다. 이는 주문실적과 판매그룹이 다대다 관계이기 때문이다.

중복 계수 문제를 해결하기 위해 몇 가지 제한적인 방식을 고려할 수 있다. 첫째, 그룹별로 구성원 수를 카운트하여 나눠준다. 예제에서 그룹 1 의 구성원 수가 2 이므로 전체 주문금액은 1000 (2000/2)가 된다. 둘째, 판매그룹 테이블에 배부율을 관리한다. 예를 들어, 김성희와 조혜리가 각각 0.75, 0.25 의 배부율을 가지면 각각의 실적은 750, 250 이 되고 전체실적은 1000 이 된다. 물론, 개별 실적도 1000, 전체 실적도 1000 이 나오도록 쉽게 처리할 수 있다. 그러나, 이 방법에는 풀어야 할 숙제가 있다. 배부율은 부서간 또는 개인간 이해가 걸려 있기 때문에 쉽게 정하기 어렵다. 설사 정했다 하더라도 장기간에 걸쳐 안정적인 유지관리가 쉽지 않다. 셋째, 브릿지 테이블을 사용하지 않고 팩트 테이블의 데이터 관리 수준(Grain)을 조정하여 아예 배부된 데이터를 적재하는 것이다. 이를 위해 주문시점에 배부율을 미리 반영하여 처리하거나 ETL 과정에서 반영한다. 그러나, 이 역시 배부율 관리 이슈는 남아 있다. 팁으로 판매그룹에 정담당여부를 추가로 관리하거나 별도의 정판매담당 테이블을 추가하여 주문실적에서 참조하도록 함으로써, 활용 능력이 낮은 사용자에게는 정담당사원 기준으로 중복 계수가 되지 않은 실적을 제공할 수 있다.

언뜻 생각하기에 매번 판매그룹키를 새로 생성하면 판매그룹과 주문실적 사이의 다대다 관계가 일대다 관계로 전환되어 중복 계수 문제를 해결할 수 있는 것으로 보일 수 있다. 예를 들어, 김성희와 조혜리라는 조합에 대해 매번 새로운 판매그룹키를 채번하는 것이다. 그러나, 여전히 하나의 주문실적이 판매그룹의 여러 행을 참조하므로 다대다 관계가 해소되지 않는다. 오히려, 판매그룹에서 관리하는 그룹 수만 많아져서 테이블이 매우 커질 수 있다.

브릿지 테이블을 활용할 때 SCD(Slowly Changing Dimension)까지 반영한다면 보다 주의가 필요하다. 먼저 차원 테이블이 SCD Type 1 이라고 하면 브릿지 테이블은 영향 받지 않는다. 따라서, 팩트 테이블도 영향이 없다. 그러나, 차원 테이블이 SCD Type 2 라고 하면 브릿지 테이블도 영향을 받는다. 예를 들어, 판매담당인 조혜리의 부서가 마케팅부에서 기획부로 바뀌면 판매담당키 333 으로 새로운 행이 추가되고, 판매그룹에도 새로운 판매담당 조합 {111, 333}에 해당하는 새로운 판매그룹키가 생성되야 한다. 이 때, 판매담당에 유효시작일자, 유효종료일자 같은 타임스탬프와 현재여부를 추가 관리할 수 있다. 이때, 기존 주문실적은 영향을 받지 않는다.

예제에서, 판매그룹 구성원 자체가 변동되는 경우 상황은 또 달라진다. 예를 들어, 조혜리가 퇴사하고 새로이 곽기영, 김종회가 그룹에 조인했다고 하자. Type 1 을 적용하면 판매담당에서 조혜리를 삭제 또는 삭제로 마크하고 곽기영, 김종회를 추가한다. 추가된 2 명의 판매담당키가 444, 555 라고 하면 판매그룹의 그룹키 1 의 구성원이 {111, 222}에서 {111, 444, 555}로 바뀌는 것이므로 판매그룹키 1 을 삭제하고 새로운 판매그룹키를 생성해야 한다. (물론, 판매그룹키 1 을 유지하면서 222 구성원을 삭제하고 444, 555 구성원을 추가할 수도 있다.) 결국 과거 주문실적까지 업데이트해야 하므로 일이 커진다. 반면에 Type 2 를 적용하면 판매그룹의 판매그룹키 1 은 유지되면서 {111, 444, 555}에 해당하는 새로운 판매크룹키 2 를 생성하면 되므로 주문실적은 영향을 받지 않는다.

이와 같이 브릿지 테이블을 적용하는 경우, SCD 관리는 보다 복잡해지므로 보다 주의가 필요하다. 경우에 따라서는 브릿지 테이블 자체도 SCD 관리 대상이 될 수 있다. 예를 들어,

판매그룹별로 지역이 할당되어 있다면, 시간이 지남에 따라 담당지역이 변경될 수 있다. 이 때, Type 1 은 변경된 담당지역을 업데이트하면 되고, Type 2 는 새로운 그룹키를 추가하면 된다. 둘 다 기존 주문실적에 영향을 주지 않는다. Type 2 의 경우 유효시작일자, 유효종료일자 같은 타임스탬프와 현재여부를 추가 관리할 수 있다.

브릿지 테이블을 이용할 때 추가적인 애로사항이 있다. 예제에서 보이는 주문실적과 판매그룹 사이의 다대다 관계를 DBMS 나 BI 도구들이 지원하지 못하는 경우이다. 대부분 테이블간의 관계를 다른 테이블의 주키를 외래키로 참조하는 방식으로 구현하기 때문이다. 해결 방법으로 다대다 관계는 논리적으로만 파악하고 실제로는 관계를 설정하지 않는 것을 고려할 수 있다. 그러나, BI 도구 입장을 고려할 때 이보다는 그림 3 과 같이 판매그룹을 정규화하는 것이 더 바람직하다. 예제에서 판매그룹은 판매그룹키만 관리하는 더미 테이블이지만 DBMS 나 BI 도구에서 정상적으로 지원할 수 있도록 도와준다. 팁으로 판매그룹에 판매그룹구성 내역을 속성으로 추가하여 "{111,222}" 와 같이 문자열로 만들어 활용하는 것도 고려할 수 있다.

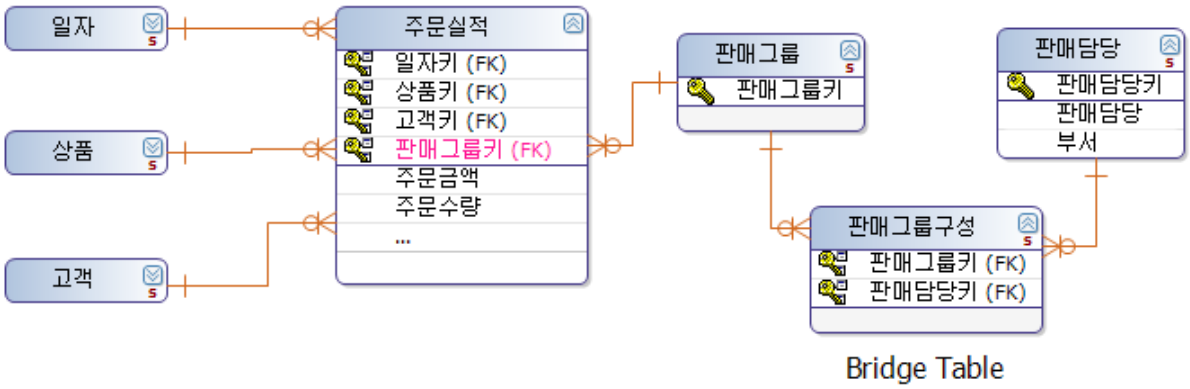



그림 3. 브릿지 테이블 정규화

지금까지는 팩트 테이블의 하나의 행이 차원 테이블의 여러 행과 관계를 가지는 경우를 살펴보았다 (다중값 차원). 유사한 경우로 팩트 테이블의 하나의 행이 차원 테이블의 하나의 행과 관계를 가지지만 속성값이 여러 개 인 경우가 있다 (다중값 속성, Multi-valued Attribute). 예를 들어, 고객의 취미가 여러 개인 경우다. 해결 방식은 다중값 차원의 경우와 유사하다.

먼저, 그림 4 와 같이 차원 테이블에 여러 개의 칼럼으로 반영하는 단순화 방식을 고려할 수 있다. 취미 1, 취미 2, 취미 3 등이 이에 해당한다. 단순화 방식의 장단점은 다중값 차원의 경우와 같다. 또한 칼럼별로 NULL 값 분포가 큰 경우 비트-맵 인덱스 적용에 유리하며 칼럼 저장 방식 데이터베이스의 경우 이점을 더 활용할 수 있다. 변형으로 칼럼명을 속성값으로 할 수도 있다. 예를 들어, 칼럼명을 바둑, 독서, 동전모으기 등으로 생성하는 것이다. 만약에 BI 도구가 지원한다면 하나의 칼럼으로 관리하며 Name1:Value1, Name2:Value2, … 형태의 "이름:값" 리스트를 문자열로 저장하는 것도 고려할 수 있다.

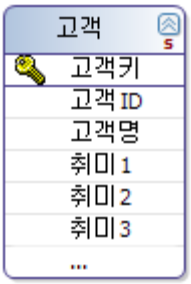



그림 4. 다중값 속성 설계 (1)

단순화 방식의 단점을 해결하기 위해 다중값 차원처럼 브릿지 테이블을 이용할 수 있다. 가장 큰 차이점이라면 브릿지 테이블의 위치이다. 다중값 차원은 그림 2 처럼 차원과 팩트 테이블 사이에 위치하는데 비해 다중값 속성은 그림 5 처럼 차원과 Outrigger(취미) 테이블 사이에 존재한다.
또한 해결해야 할 다대다 관계가 전자는 팩트 테이블과 브릿지 테이블 사이에 있고, 후자는 차원 테이블과 브릿지 테이블 사이에 있는 점도 다르다.

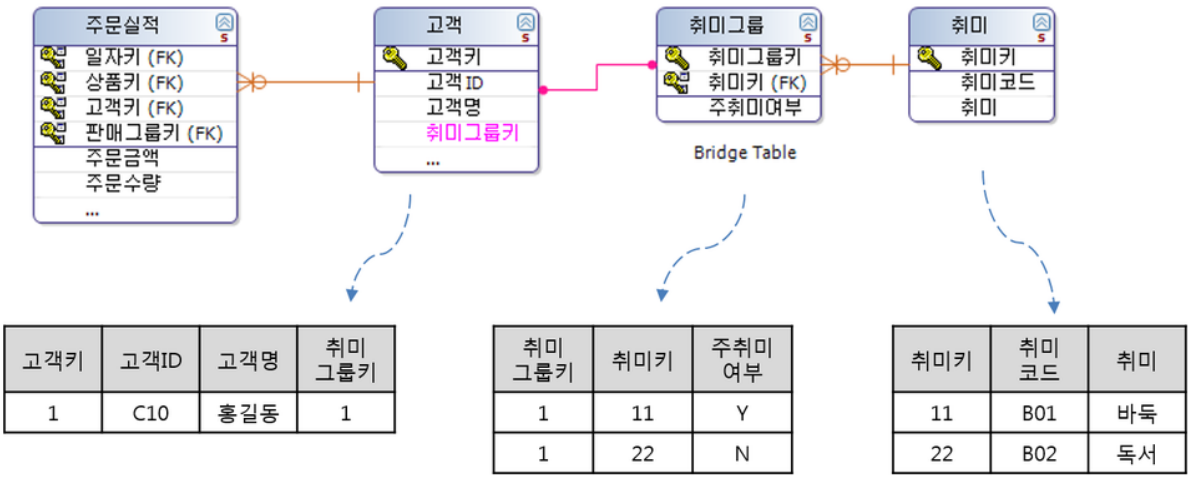



그림 5. 다중값 속성 설계 (2)

그러나, 궁극적으로는 동일한 형태로 이해할 수 있다. DBMS 나 BI 도구들에서 지원 가능한 형태가 되도록 그림 5 를 그림 6 처럼 정규화해 보면 그림 3 과 논리적으로 동일한 형태가 된다.

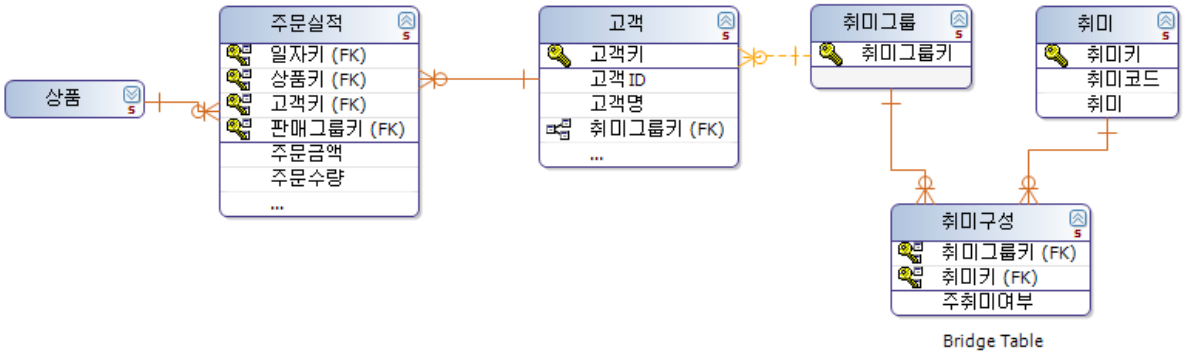



그림 6. 브릿지 테이블 정규화 (2)

사실 그림 3 에서 판매그룹도 차원의 하나로 볼 수 있다. 판매그룹구성도 별도의 팩트를 저장할 수 있는 팩트 테이블로 볼 수 있다. 또한 그림 6 에서 취미그룹도 고객 차원의 Outrigger 차원 형태로 볼 수 있고, 취미도 하나의 차원으로 볼 수 있다. 취미구성 역시 별도의 팩트 테이블로 볼 수 있다. 따라서 넓게 보면 "차원 1-팩트 A-차원 2-팩트 B-차원 3" 형태가 되어 양방향으로 브릿지 테이블을 경유하는 형태의 분석을 기대할 수 있다. 즉, 차원 1 를 기준으로 팩트 B 를 분석하고 차원 3 을 기준으로 팩트 B 를 분석하는 것으로서 중간 다대다 관계를 거치게 된다. 이때 차원 2 는 공통 차원으로서 연결고리 역할을 하게 되고, 팩트 A 와 팩트 B 는 팩트 테이블과 브릿지 테이블 역할을 동시에 하게 된다.

최근에는 이러한 다대다 관계를 지원하는 OLAP 엔진이나 BI 도구가 나오면서 보다 쉽게 다중값 문제를 해결할 수 있다. 즉, 그림 7 처럼 별도로 취미그룹을 생성하지 않고도 취미별로 주문실적을 분석할 수 있다. 반대로 상품별로 고객취미 분포를 분석할 수도 있다. 단, 이 상태에서는 취미나 상품에 대한 AND 분석이 번거롭다. 즉, 취미가 바둑 또는 독서인 고객의 주문실적은 쉽게 처리하지만, 취미가 바둑과 독서인 고객의 주문실적을 보고 싶으면 처리가 복잡해진다. (다대다 분석을 지원하는 대부분의 BI 도구는 OR 분석을 지원한다.) 이를 보완하기 위해 취미조합별로 그룹 리스트를 관리하는 테이블을 추가하고 고객 테이블에서 이를 참조하도록 고려할 수 있다.


그림 7. 그룹 생성 없는 다대다 관계 모델

이상으로 다중값 차원과 다중값 속성 기준으로 분석하고자 하는 요건을 해결하기 위한 접근 방법들을 살펴 보았다. 크게 보면 차원과 팩트 사이의 다대다 관계를 해소하는 방식으로 볼 수 있다. 여기에서 일반적으로 익숙하지 않은 그룹키를 관리하는 브릿지 테이블을 이용한 것은 사용자들의 요구를 보다 효과적으로 충족시키기 위한 방안에서 나온 것이다. 이는 정답은 아니란 의미이다. 보다 더 쉽고 나은 방법이 당연히 그 방법으로 가면 된다. 아마도, 대부분 그림 7 과 같은 형태가 더 친숙하고 이해도 쉬울 것이다. 다행히도 최근 OLAP 엔진이나 BI 도구들에서 우리가 더 친숙한 이러한 형태의 모델을 지원하는 추세다. 따라서, 시간이 더 흐르면 굳이 복잡하게 설계하지 않아도 다중값 차원과 다중값 속성 문제를 보다 더 쉽게 해결할 수 있을 것으로 본다.


데이터 모델링에서 모델러를 가장 고민하게 만드는 것 중 하나가 계층구조 설계일 것이다. 계층구조 중에서도 다양한 레벨 깊이를 가지는 트리 구조가 특히 그러하다. 계층구조는 조직, 사원, 계정과목 등 다양하게 나타나는데, 데이터 처리 중심의 운영계와 달리 DW Presentation 영역은 최종 사용자의 편이성과 성능을 최대한 고려해야 한다는 점에서 또 다른 고민이 있다. 이번 기사에서는 계층구조의 형태와 룩다운/업 등을 효율적으로 처리하기 위한 방법들을 살펴 보도록 하겠다.

가장 단순한 형태의 계층구조는 그림 1 의 예제처럼 고정된 깊이를 가지는 형태이다. 이 때, 각각의 레벨은 년도, 분기, 월, 일과 같이 의미 있는 이름을 부여해서 사용자가 직관적으로 이해하고 접근할 수 있도록 하는 것이 중요하다. 예를 들어, 최종 사용자는 별도의 설명이 없어도 일 레벨의 데이터를 롤업하면 월 레벨의 데이터가 나온다는 것을 직관적으로 이해한다.


그림 1. 고정 깊이 계층구조 (1)

많은 경우 고정된 깊이 형태를 가질 때 그림 2 와 같이 정규화 하는 것에 대한 유혹을 느끼게 되는데 가능하면 정규화하지 않은 형태를 권고한다. 여기에서 "가능하면" 이라고 표현한 것은 어쩔 수 없이 정규화해야 하는 경우도 있기 때문이다. 예를 들어, 실적은 제품 레벨에서 관리하지만 계획은 브랜드 레벨에서 관리할 수 있다. 이 경우, 제품 테이블을 정규화하여 실적 테이블은 제품 테이블을 참조하고 계획 테이블은 브랜드 테이블을 참조하도록 하는 것이 더 합리적이다. 물론, 이 경우에도 대표 제품을 지정하여 계획 데이터도 제품 레벨에서부터 관리하는 것으로 설계함으로써 정규화를 피해가는 방법도 있다. 정규화에 관련된 이슈는 *DW Presentation 영역에서의 정규화 유혹* 기사를 참고하기 바란다.

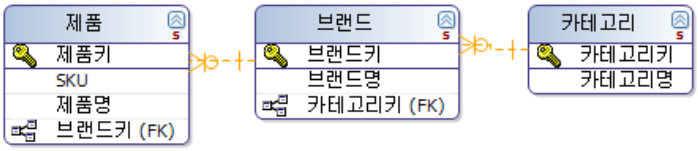
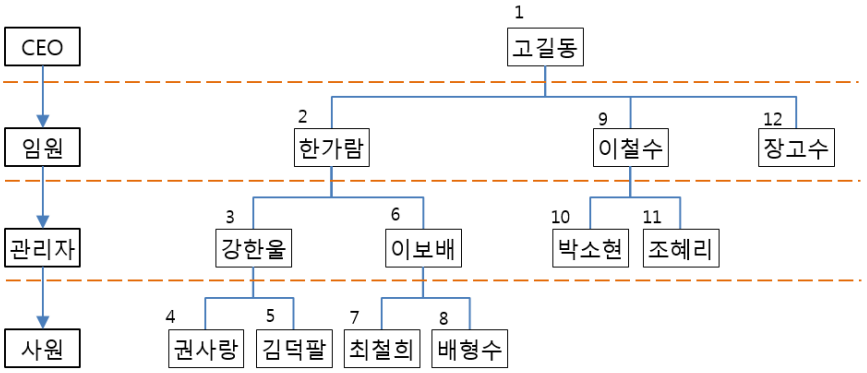



그림 2. 고정 깊이 계층구조 (2)

고정 깊이 계층구조로 설계하는 경우, 인위적으로 고정 깊이로 가공하는 경우가 있다. 그림 3 을 보면 사원들의 계층구조를 구성하는 최하위 멤버들을 보면 레벨 깊이가 다르다. 이를 테이블의 주황색 부분처럼 인위적으로 가공하여 레벨을 맞추는 것이다. 보통 상위 멤버와 동일한 값을 가지도록 가공한다.




| 번호 | CEO | 임원 | 관리자 | 사원 |
|---|---|---|---|---|
| 4 | 고길동 | 한가람 | 강한울 | 권사랑 |
| 5 | 고길동 | 한가람 | 강한울 | 김덕팔 |
| 7 | 고길동 | 한가람 | 이보배 | 최철희 |
| 8 | 고길동 | 한가람 | 이보배 | 배형수 |
| 10 | 고길동 | 이철수 | 박소현 | 박소현 |
| 11 | 고길동 | 이철수 | 조혜리 | 조혜리 |
| 12 | 고길동 | 장고수 | 장고수 | 장고수 |

그림 3. 고정 깊이 계층구조 (3)

이 때, 최종 사용자가 보는 뷰를 그림 3 의 트리 구조처럼 보이게 하는 것은 클라이언트 도구의 몫이다. 그런데, 이러한 방식의 경우 풀어야 할 이슈가 잠재해 있다. 예를 들어, 그림 4 에서 장고수는 사장 비서로서 직급이 임원이 아니라 관리자 레벨일 수 있다. 이러한 것을 나타내기 위해 부모 멤버로부터 건너뛰는 레벨 수를 함께 관리할 수 있는데, 클라이언트 도구에서 이를 표현하는 것은 별개의 몫으로 남는다.

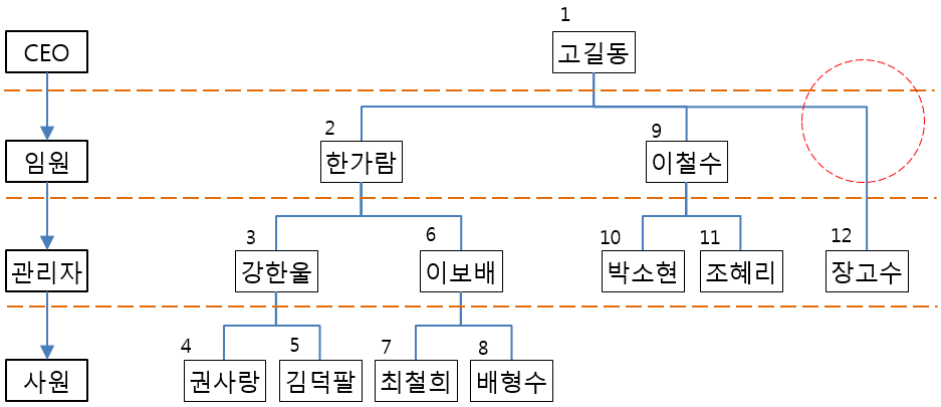



그림 4. 고정 깊이 계층구조 (4)

지금까지의 예제는 계층구조 트리에서 최하위 멤버들에 해당하는 데이터만 가지고 상위 레벨까지 표현하는 것이었다. 하지만, 연결되는 팩트들이 상위 레벨의 임의의 멤버들에서도 발생할 수 있다면 보완이 필요하다. 계층구조를 만들기 위한 사원 테이블에 상위 멤버들 데이터도 모두 포함시켜야 하는데 이쯤 되면 그림 5 처럼 당연히 재귀 참조를 사용하고 싶을 것이다. 최근에는 SQL Server 의 CTE(Common Table Expressions)나 Oracle 의 Connect By 처럼 재귀 참조 테이블을 이용하여 계층구조를 쉽게 풀어낼 수 있도록 DBMS 에서 관련 SQL 을 지원하고 있다. 또한, OLAP 엔진이나 BI 도구들에서도 재귀 참조를 지원하는 것이 일반화되어 모델러의 어깨를 한층 가볍게 해주고 있다.

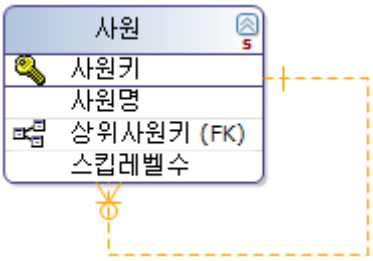



그림 5. 재귀 참조

그럼에도 불구하고, 2% 부족한 부분이 있어서 재귀 참조와 별도로 추가 설계를 고민해야 할 경우가 있다. 우선, 재귀 참조 방식은 사전에 계층구조를 구성하는 레벨별로 의미 있는 레벨명을 부여하기 어렵다. 또한, 계층구조가 2 개 이상인 경우 효과적이지 못하다. 예를 들어, 사원의 경우 직무상의 계층구조와 자기계발을 관리하기 위한 계층구조가 다를 수 있다. 즉, 재귀 참조에서의

부모와 자식 관계가 다대다 관계가 되는 것이다. 또한 SCD Type 2 변화 관리를 해야 하는 경우, 변화가 발생한 멤버의 모든 하위 멤버들에도 영향을 미치므로 유지보수에 어려움이 발생한다. 이러한 것에 대한 보완책으로 브릿지 테이블을 이용하는 것을 고려할 수 있다.

| 상위사번키 | 하위사번키 | 부모로부터깊이 | 루트Flag | 바텀Flag |
|---|---|---|---|---|
| 1 | 1 | 0 | Y | N |
| 1 | 2 | 1 | Y | N |
| 1 | 3 | 2 | Y | N |
| 1 | 4 | 3 | Y | Y |
| 1 | 5 | 3 | Y | Y |
| 1 | 6 | 2 | Y | N |
| 1 | 7 | 3 | Y | Y |
| 1 | 8 | 3 | Y | Y |
| 1 | 9 | 1 | Y | N |
| 1 | 10 | 2 | Y | Y |
| 1 | 11 | 2 | Y | Y |
| 1 | 12 | 1 | Y | Y |
| 2 | 2 | 0 | N | N |
| 2 | 3 | 1 | N | N |
| 2 | 4 | 2 | N | Y |
| 2 | 5 | 2 | N | Y |
| 2 | 6 | 1 | N | N |
| 2 | 7 | 2 | N | Y |
| 2 | 8 | 2 | N | Y |
| 3 | 3 | 0 | N | N |
| 3 | 4 | 1 | N | Y |
| 3 | 5 | 1 | N | Y |
| 4 | 4 | 0 | N | Y |
| 5 | 5 | 0 | N | Y |
| 6 | 6 | 0 | N | N |
| 6 | 7 | 1 | N | Y |
| 6 | 8 | 1 | N | Y |
| 7 | 7 | 0 | N | Y |
| 8 | 8 | 0 | N | Y |
| 9 | 9 | 0 | N | N |
| 9 | 10 | 1 | N | Y |
| 9 | 11 | 1 | N | Y |
| 10 | 10 | 0 | N | Y |
| 11 | 11 | 0 | N | Y |
| 12 | 12 | 0 | N | Y |

그림 6. 사원 브릿지 테이블

그림 6 은 그림 3 에 해당하는 사원 브릿지 테이블이다. 각 사원 기준으로 모든 하위 사원들과의 관계를 기술하였다. 루트 Flag 는 계층구조에서 최상위 멤버 여부를, 바텀 Flag 는 더 이상 자식이 존재하지 않는 최하위 멤버 여부를 관리하는 것이다. 그림 7 은 사원 브릿지 테이블을 활용한 설계 예제인데 상위멤버와 하위멤버 간의 다대다 관계까지 관리가 가능하다. 비중 칼럼은 하나의 멤버가 2 개 이상의 부모 멤버를 가지는 경우 상위 레벨로 롤업될 때의 가중치를 관리하는 용도이다.

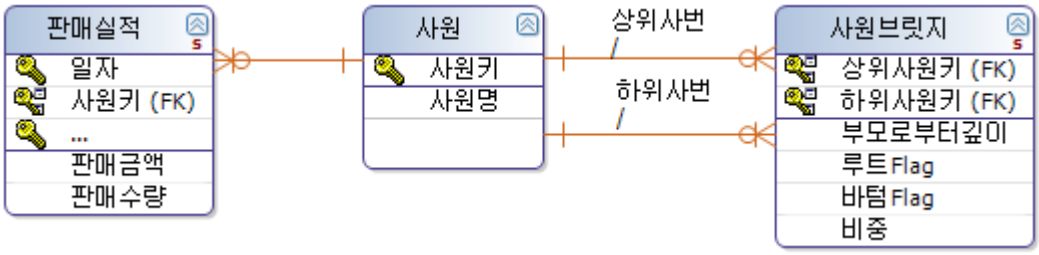



그림 7. 브릿지 테이블을 활용한 계층구조 설계

그런데, 실제로 사원 기준으로 판매실적을 조회하고자 하는 경우 사원브릿지 테이블이 판매실적과 사원 테이블 중간에서 다리 역할 하는 방식으로 활용된다. 경우의 수는 크게 3 가지인데 각각의 경우를 살펴보겠다.

첫째, 단순히 사원 기준으로 판매 실적을 조회하는 경우이다. 단순히 판매실적과 사원 테이블을 조인하면 된다. (필요에 따라 조인 결과를 사원키로 Group By 처리하면 된다.) 이 때는 브릿지 테이블이 필요 없다.

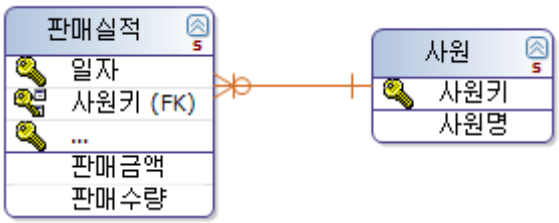
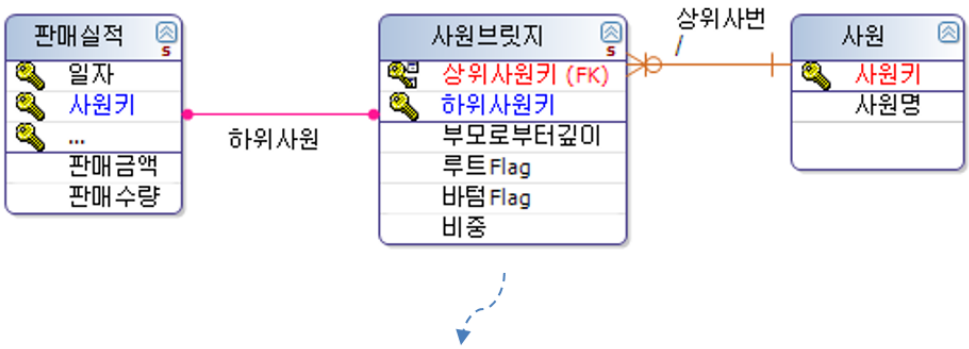



그림 8. 사원 기준 실적 조회

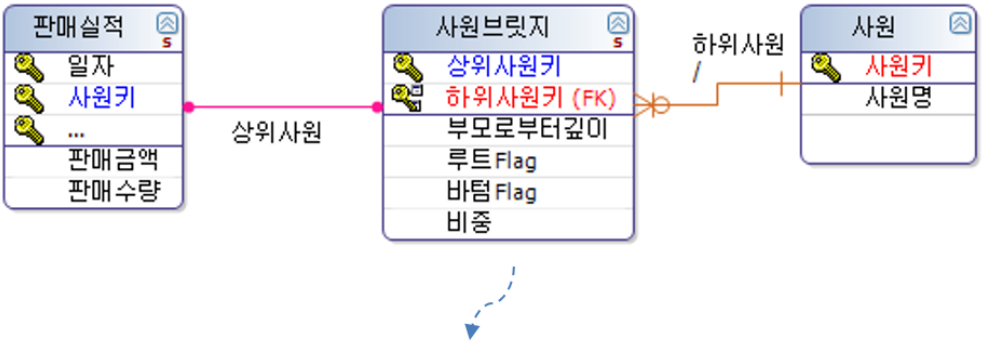
둘째, 계층구조 상의 특정 사원 기준으로 자신을 포함한 하위 모든 사원들의 판매 실적을 조회하는 경우이다 (Look Down). 이 때는 사원과 판매실적 테이블 중간에서 브릿지 테이블이 다리 역할을 한다. 그림 9 에서, 오른쪽부터 해석하면 먼저 조회 기준이 되는 상위사원을 선택하고, 선택된 사원을 포함하여 하위 모든 사원들을 브릿지 테이블에서 뽑아낸다. 그리고, 그 결과를 판매실적과 조인 처리한다. 이 때 기준 사원으로 하나만 선택하면 하위사원들은 중복이 발생하지 않으므로 판매실적도 중복 계수되지 않는다. 예를 들어, 상위사번으로 2 하나만 선택하면 나타나는 하위사번들은 2, 3, 4, 5, 6, 7, 8 로 단 한번씩만 나타나서 중복이 발생하지 않는다. 그러나, 기준 사원으로 2 개 이상을 선택하면 하위 사원들은 중복이 발생할 수 있으므로 기준 사원 기준으로 2 차 Group By 를 해야 한다. 예를 들어, 상위사번으로 2, 3 을 선택하면 하위사번 중 3, 4, 5 는 중복이 발생하므로 실적도 중복 계수된다. 이 때, 상위사번키로 Group By 를 하면 기준 사원별로 중복되지 않은 집계 결과를 얻을 수 있다. 단, 전체 합계를 구하고자 할 때는 여전히 중복이 발생하므로 별도의 *"Distinct 하위사원키"*를 서브 쿼리로 구해서 활용해야 한다.




| 상위사번키 | 하위사번키 | 부모로부터깊이 | 루트Flag | 바텀Flag |
|---|---|---|---|---|
| ... | ... | ... | ... | ... |
| 2 | 2 | 0 | N | N |
| 2 | 3 | 1 | N | N |
| 2 | 4 | 2 | N | Y |
| 2 | 5 | 2 | N | Y |
| 2 | 6 | 1 | N | N |
| 2 | 7 | 2 | N | Y |
| 2 | 8 | 2 | N | Y |
| 3 | 3 | 0 | N | N |
| 3 | 4 | 1 | N | Y |
| 3 | 5 | 1 | N | Y |
| ... | ... | ... | ... | ... |

그림 9. 사원 기준 실적 조회 (Look Down)

둘째, 계층구조 상의 특정 사원 기준으로 자신을 포함한 상위 모든 사원들의 판매 실적을 조회하는 경우이다 (Look Up). 이 때도 Look Down 시나리오와 마찬가지로 사원과 판매실적 테이블 중간에서 브릿지 테이블이 다리 역할을 하는데 역할이 바뀐다. 그림 10 에서, 오른쪽부터 해석하면 먼저 조회 기준이 되는 하위사원을 선택하고, 선택된 사원을 포함하여 상위 모든 사원들을 브릿지 테이블에서 뽑아낸다. 그리고, 그 결과를 판매실적과 조인 처리한다. 이 때 기준 사원으로 하나만 선택하면 상위사원들은 중복이 발생하지 않으므로 판매실적도 중복 계수되지 않는다. 예를 들어, 하위사번으로 3 하나만 선택하면 나타나는 상위사번들은 1, 2, 3 으로 단 한번씩만 나타나서 중복이 발생하지 않는다. 그러나, Look Down 시나리오에서와 마찬가지로 기준 사원으로 2 개 이상을 선택하면 상위 사원들은 중복이 발생할 수 있으므로 기준 사원 기준으로 2 차 Group By 를 해야 한다. 단, 전체 합계를 구하고자 할 때는 여전히 중복이 발생하므로 별도의 *"Distinct 상위사원키"*를 서브 쿼리로 구해서 활용해야 한다.




| 상위사번키 | 하위사번키 | 부모로부터깊이 | 루트Flag | 바텀Flag |
|---|---|---|---|---|
| 1 | 1 | 0 | Y | N |
| 1 | 2 | 1 | Y | N |
| 1 | 3 | 2 | Y | N |
| ... | ... | ... | ... | ... |
| 1 | 12 | 1 | Y | Y |
| 2 | 2 | 0 | N | N |
| 2 | 3 | 1 | N | N |
| ... | ... | ... | ... | ... |
| 2 | 8 | 2 | N | Y |
| 3 | 3 | 0 | N | N |
| 3 | 4 | 1 | N | Y |
| 3 | 5 | 1 | N | Y |
| ... | ... | ... | ... | ... |

그림 10. 사원 기준 실적 조회 (Look Up)

이상으로 계층구조와 브릿지 테이블을 활용하는 방법을 간단하게 살펴 보았다. 브릿지 테이블은 필요에 따라 추가적인 변형이 가능하다. 예를 들어, 그림 11 과 같이 시작일자, 종료일자와 같은 태그를 두어 특정 시점의 계층구조를 기준으로 Look Down/Up 처리를 할 수 있다.

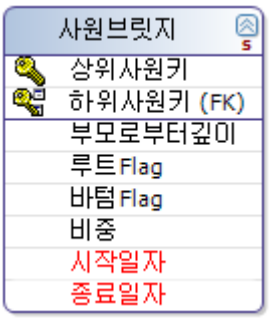



그림 11. 계층구조 변경 관리

사실 브릿지 테이블 활용과 관련하여 시간에 따른 변경 관리는 그림 11 과 같은 간단한 그림 하나로 끝날 사항은 아니다. 브릿지 테이블은 지금까지 살펴본 바와 같이 상위키와 하위키로 차원 테이블을 이중 참조하고 있다. 따라서, 차원 테이블의 변경 관리 유형에 따라 영향을 받는데 두 가지 참조 역할을 모두 고려해야 한다. 또한, 브릿지 테이블에서의 자체 속성이나 계층 구조 변경 관리도 필요할 수 있다. 따라서, 이와 관련된 내용은 너무 길어질 수 있으므로 기회가 될 때 별도의 기사에서 다루도록 하겠다.


스타 스키마 모델에서 팩트 테이블은 조회 및 분석의 대상이 되는 팩트를 저장한다. 여기에서 팩트란 주문금액, 주문수량 처럼 관리가 필요한 측정 가능 수치들을 의미한다. 그런데, 그러한 수치들이 없어도 팩트 테이블을 만들어야 할까? 대개는 필요 없지만 필요에 따라 만드는 경우가 있다. (팩트가 없는 팩트 테이블을 Factless Fact Table 이라 부른다.) 첫째, 차원의 브릿지 테이블 용도로 사용하는 경우가 있다. 브릿지 테이블에 대한 내용은 이전의 *다중값 차원 (Multi-valued Dimension)*과 *계층구조와 브릿지* 기사를 참고하길 바란다. 둘째, 비즈니스 활동을 분석하고 관련된 환경 여건들을 비교하고자 하는 경우이다. 이번 기사에서는 후자의 경우에 해당하는 내용을 살펴보도록 하겠다.

먼저, 비즈니스 활동 분석과 관련된 경우로 이벤트를 카운트하여 분석 대상으로 활용한다. 예를 들어, 고객 콜 센터에서 민원 접수에서 처리까지의 상태별 카운트를 하여 민원 접수건수, 처리 중 건수, 처리건수 등을 분석할 수 있다. 이 외에도 광고의 노출 횟수, 웹 사이트 및 페이지의 방문 횟수, 수업의 출석 및 결석 횟수 등 카운트를 활용하는 사례는 주변에서 쉽게 접할 수 있다.

이 때, 팩트 테이블의 그레인 즉, 데이터를 관리하는 상세 수준은 "하나의 행은 하나의 이벤트를 나타낸다"와 같이 정의할 수 있다. 또한, 각 행은 다양한 차원 테이블을 참조함으로써 이벤트에 의미를 부여하게 된다. 그림 1 에서, 콜센터접수내역의 각 행은 하나의 민원접수 이벤트를 나타내며 별도의 팩트를 가지지 않는다. 또한, 각 행은 일자, 시간, 고객, 접촉유형 등을 참조함으로써 이벤트를 카운트하여 해당 차원으로 다양하게 분석할 수 있다.

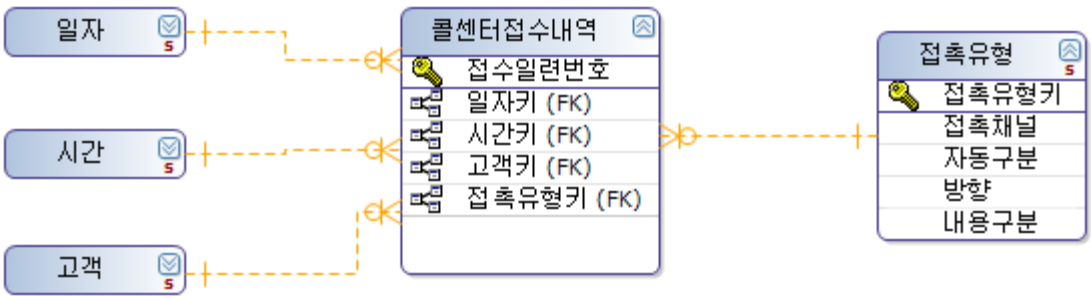



그림 1. 콜센터 접수 내역 (1)

그림 2 처럼 카운트와 같은 칼럼을 추가하여 Count() 대신에 Sum()을 활용함으로써 동일한 분석이 가능하면서 SQL 쿼리를 작성하거나 읽는데 용이하도록 할 수도 있다. 그런데, 여기에는 장단점이 있다. 관계형 DBMS 입장에서는 칼럼을 추가하는 게 공간만 더 소모하고 실제 쿼리 처리도 더 비효율적일 수 있다. Sum()은 구체적인 데이터 페이지를 읽어야 하지만, Count()는 인덱스 페이지만 읽어도 충분하기 때문이다. 그렇지만, 스타-조인 최적화를 지원하면 팩트 테이블에서 이미 충분히 필터링된 일부 데이터만 읽기 때문에 I/O 측면에서 Count()를 쓸 때의 장점이 미미할 수 있다. 반면에 전통적인 OLAP 엔진은 하위 레벨의 데이터를 읽어서 상위 레벨의 집계 데이터를 구하는데 Count 보다는 Sum 방식에 더 최적화되어 있다. 따라서, 어느 방식으로 할지는 상황에 따라 판단하는 것이 더 합리적이라 하겠다.

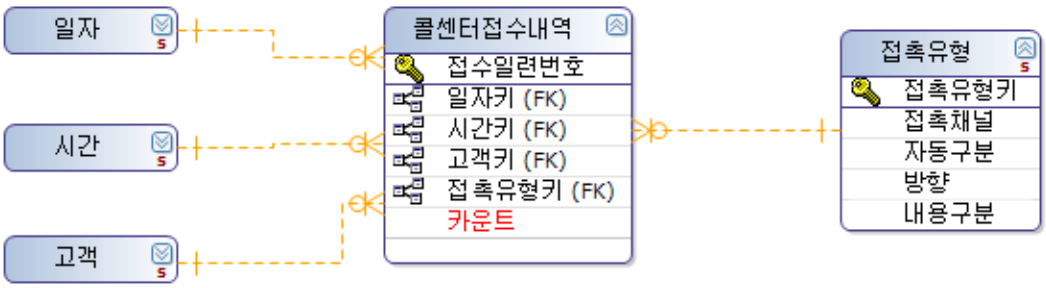



그림 2. 콜센터 접수 내역 (2)

다음으로 살펴볼 것은 비즈니스 환경 여건들을 비교하고자 하는 경우이다. 예를 들어, 특정 시점에 고객에 할당된 판매사원들과 실제 판매실적을 가지고 있는 판매사원들을 비교 분석하는 것이다. 이를 위해, 고객에 할당된 판매사원들을 관리하는 테이블이 필요하다. 이벤트를 카운트하는 것은 그 자체가 측정값으로 의미를 가지는 반면에 고객에 할당된 판매사원과 같은 경우는 다른 무엇과 비교를 위한 기준 데이터로서 의미를 가진다.

그림 3 은 정상적인 팩트 테이블이 포함된 일반적인 스타 스키마이다. 주문 실적을 일자, 상품, 판매담당, 고객의 조합별로 분석할 수 있다. 이 때, 주문실적이 판매담당과 고객의 연결 고리 역할을 하는데 실제로 주문이 발생한 경우에 한한다. 이 경우, 고객을 할당 받았지만 해당 고객에 대한 판매 실적이 없는 판매사원을 파악하고자 할 때 방법이 없다. 따라서, 추가적으로 고객에 할당된 판매사원들을 관리하는 테이블이 필요한데 팩트가 없는 Factless Fact Table 형태이다.

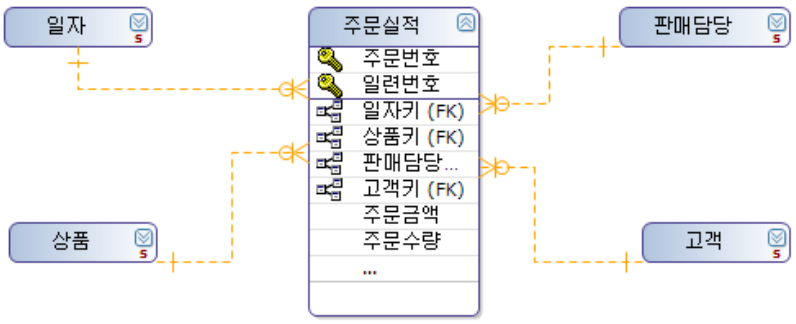



그림 3. 주문 실적 스타 스키마

그림 4 는 판매담당별로 고객할당 이력을 관리하는 예이다. 고객할당내역을 보면 별도의 팩트가 없는 Factless Fact Table 이다. 이제 그림 4 의 고객할당내역과 그림 3 의 주문실적을 함께 활용하면 고객을 할당 받았는데 해당 고객에 대한 실적이 없는 판매사원을 쉽게 알아낼 수 있다.

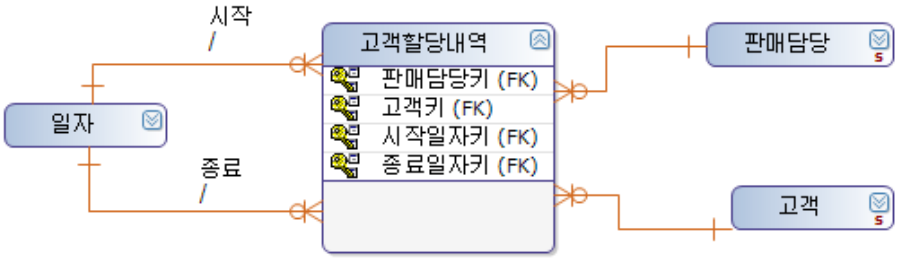



그림 4. 고객 할당 내역 (Factless Fact Table)

비교를 할 수 있는 기준이 있다는 것은 데이터를 분석하는 관점에서 매우 매력적이다. 그림 5 를 보면 왜 매력적인지 쉽게 알 수 있다. 고객을 할당 받았지만 실적이 없는 경우(①), 고객을 할당 받고 실적까지 있는 경우(②), 고객을 할당 받지 않았지만 실적이 있는 경우(③) 등 다양한 상태를 분석할 수 있다.

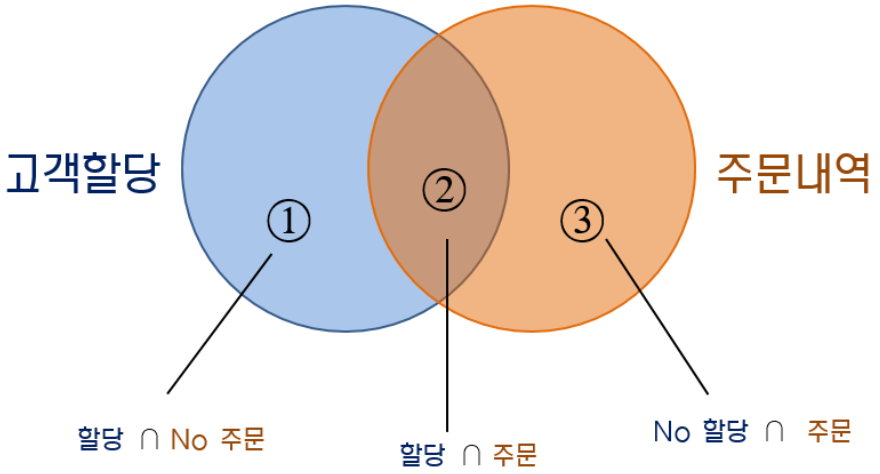



그림 5. 고객/판매담당의 관계

우선, 고객을 할당 받고 실적까지 있는 경우는 고객할당내역과 주문실적을 조인하면 쉽게 파악할 수 있다.

고객을 할당 받았지만 실적이 없는 경우도 쉽게 파악할 수 있다. MINUS 와 같은 집합 연산자나 상관 서브쿼리 등을 이용하면 된다. 예를 들어, 2014 년도 1 사분기(2014-01-01 ~ 2014-03-31)에 고객을 할당 받았지만 실적이 없는 경우를 파악하고자 한다면 다음과 같이 쿼리문을 작성할 수 있다. (편의상 분기 시작 전에 고객을 할당하고 분기 중에는 변동이 없는 것으로 가정하겠다.)

1) MINUS 집합 연산자 이용

```
SELECT B.판매담당자, C.고객명
FROM 고객할당내역 A
INNER JOIN 판매담당 B ON A.판매담당키 = B.판매담당키
INNER JOIN 고객 C ON A.고객키 = C.고객키
INNER JOIN 일자 D ON A.시작일자키 = D.일자키
INNER JOIN 일자 E ON A.종료일자키 = D.일자키
WHERE D.일자 >= '2014-01-01' AND E.일자 <= '2014-03-31'
MINUS
SELECT B.판매담당자, C.고객명
FROM 주문실적 O
INNER JOIN 판매담당 B ON O.판매담당키 = B.판매담당키
INNER JOIN 고객 C ON O.고객키 = C.고객키
INNER JOIN 일자 D ON O.일자키 = D.일자키
WHERE D.일자 >= '2014-01-01' AND D.일자 <= '2014-03-31'
```

2) 상관 서브쿼리 이용

```
SELECT B.판매담당자, C.고객명
FROM 고객할당내역 A
INNER JOIN 판매담당 B ON A.판매담당키 = B.판매담당키
INNER JOIN 고객 C ON A.고객키 = C.고객키
INNER JOIN 일자 D ON A.시작일자키 = D.일자키
INNER JOIN 일자 E ON A.종료일자키 = D.일자키
```

WHERE D.일자 >= '2014-01-01' AND E.일자 <= '2014-03-31'

**AND NOT EXISTS**

( SELECT *

FROM 주문실적 O

INNER JOIN 판매담당 X ON O.판매담당키 = X.판매담당키

INNER JOIN 고객 Y ON O.고객키 = Y.고객키

INNER JOIN 일자 Z ON O.일자키 = Z.일자키

WHERE Z.일자 >= '2014-01-01' AND Z.일자 <= '2014-03-31'

**AND X.판매담당 ID = A.판매담당 ID**

**AND Y.고객 ID = C.고객 ID**

)

2 번째 상관 서브쿼리를 이용하는 예제에서 판매담당 ID, 고객 ID 는 자연키(Natural Key)를 의미한다. 이는 판매담당과 고객 테이블을 관리할 때 SCD Type 2 형태로 속성 변경 관리하는 경우를 염두에 둔 것이다. 엄격하게 본다면 1 번째 MINUS 연산자를 이용하는 예제는 변경 관리를 고려한다면 잠재적인 위험성이 있다. 그럴 리는 거의 없겠지만 판매담당자나 고객의 이름이 변경될 수 있다면 문제가 될 수 있다. 따라서, 더 확실하게 하려면 이름 보다는 자연키를 이용하는 것이 바람직하다. 고객을 할당 받지 않았지만 실적이 있는 경우도 마찬가지 방식으로 구하면 되므로 별도로 언급하진 않겠다.

이상으로 간단하게 Factless Fact Table 의 의미와 활용에 대해서 살펴보았다. 어찌 보면 용어를 모르더라도 일상적인 데이터 모델링에서 자주 나올 만한 내용들이기 때문에 별도로 다룰 만한 내용은 아니라고 느낄 수도 있다. 그러나, 데이터를 활용하는 입장에서 볼 때 이것이 과연 어떠한 의미를 가지는지 다시 생각해 보는 측면에서는 나름 의미 있다고 생각한다.