Chapter

0

데이터베이스 설계의 기본 원리

이 강좌에서는 데이터베이스 프로젝트의 분석과 설계에 대해서 기본적인 지식을 살펴보고자 한다. 또한, 프로젝트 전반적인 관점보다는 데이터베이스 분석과 설계에 그리고 이를 델파이 프로젝트에 적용하는 초기 단계에 대한 설명을 간략하게 살펴보고자 한다.

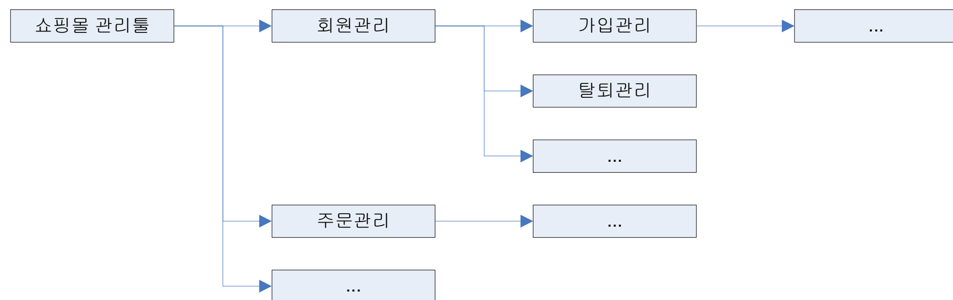
프로젝트 전반적인 프로세스에 대한 자세한 설명은 http://cafe.naver.com/codeway/540를 참고하기 바란다.

기능분석

시스템을 분석할 때 필자는 제일 우선적으로 기능분석을 실시해야 한다고 생각한다. 그 이유는 기능분석을 통해서 우리는 구축해야 할 전체 시스템의 규모와 기능을 일목요연하게 정리할 수 있기 때문이다. 또한, 이를 통해서 시스템을 구축하기 위해 필요한 최소 단위의 프로세스를 판별해 낼 수가 있다.

기능분석은 우선 개발해야 할 시스템이 가져야 할 기능을 Tree 구조로 도식화하여 정리한다. 이것은 Top-Down 방식을 통해서 개발해야 할 시스템의 기능 구성을 보다 쉽게 정리할 수 있기 때문이다. 이때 기능의 각 항목은 짧은 제목으로 표시한다.

또한 기능설계를 통해서 작성된 Tree는 개발일정 산출의 근거가 되기도 한다. (http://cafe.naver.com/codeway/142 참고)



[그림 1] PBS (Process Breakdown Sheet)

PBS의 Tree 구조에서 가장 낮은 단계의 노드(네모박스)가 바로 시스템을 구축하기 위해 필요한 최소단위의 프로세스들이다. 이제부터 이것들을 "프로세스" 또는 "단위 프로세스"라고 부르겠다.

엔티티 도출

우선 단위 프로세스를 분석하게 되면 엔티티 도출은 상당히 쉬워진다. 엔티티란 정보를 저장하는 최소단위 객체를 뜻한다. 여기서는 단순히 데이터베이스의 테이블이라 명칭 하겠다.

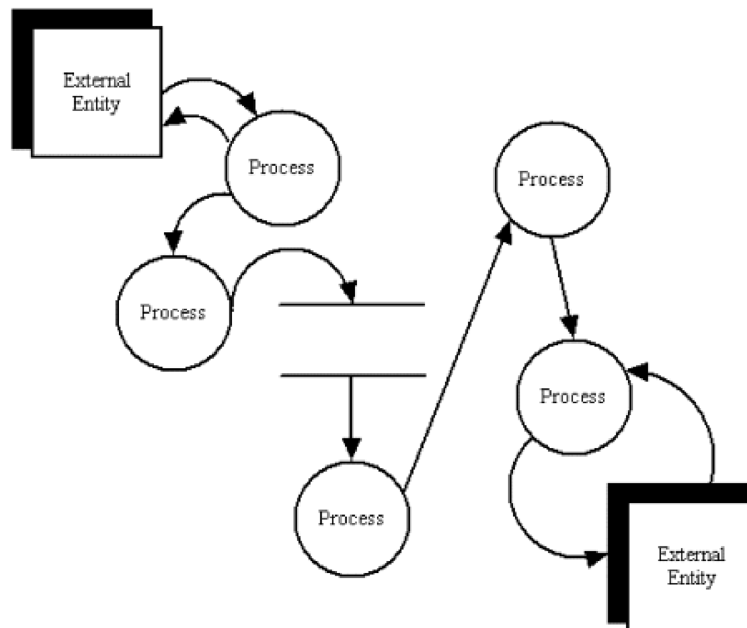
기본적으로 프로세스 또는 프로그램이라고 하면 크게 세가지 구조로 나뉘게 된다. 그것은 "입력 → 처리 → 출력"으로 표현된다. 즉, 모든 프로그램은 입력된 데이터를 처리하고 그 가공한 결과 데이터를 출력하게 된다.

따라서 우리는 도출된 각각의 단위 프로세스가 필요한 입력데이터의 종류를 찾아내고 처리 후 결과를 저장해야 할 종류를 분석해 나가면, 이 시스템 전체가 필요한 엔티티의 종류를 알아낼 수가 있는 것이다.

엔티티는 이 밖에도 장표분석, 업무분석, 동적분석 등 모든 수단을 통하여 세밀하게 검토되고 도출되어야 한다.

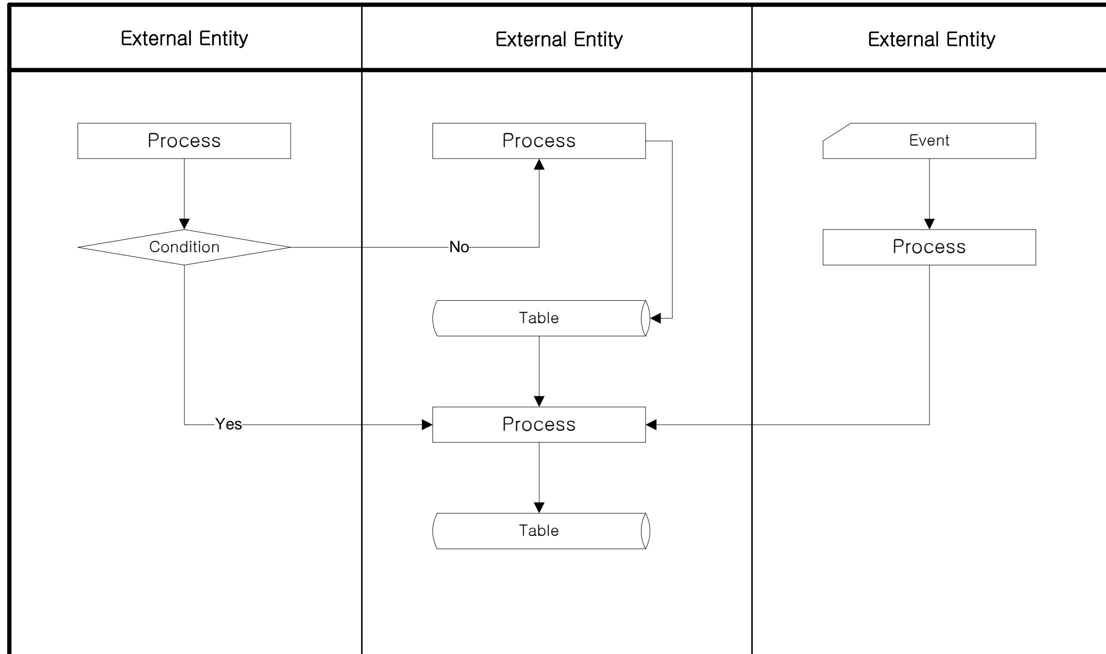
동적분석

데이터베이스 시스템에서 동적분석의 가장 대표적인 예는 DFD이다.



[그림 2] DFD

DFD를 통해서 우리는 프로세스와 데이터 흐름의 관계를 분석하고 검증할 수가 있다. 필자는 아래와 같은 형식으로 DFD를 대체해서 사용한다.

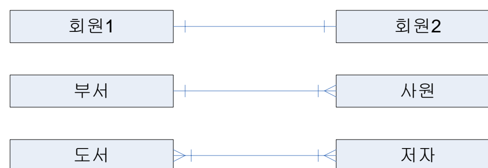


[그림 3] Job Flow

Job Flow를 통해서 우리는 프로세스의 흐름과 데이터베이스의 흐름을 분석하고 검증할 수 있으며, 이를 통해서 고객이 설명하고자 하는 요구사항을 도면을 통해서 확인하고 시스템에 적용할 수가 있다. (http://cafe.naver.com/codeway/543 등 참고)

관계 정의

엔티티간의 관계를 정의하는 단계이다.



[그림 4] ERD

[그림 4]에서 보듯이 엔티티간에는 주로 1:1, 1:N, N:M의 관계로 분류된다. 여기서 N값이 또는 M값이 '0' 이 포함될 수 있느냐 없느냐를 다시 구분하게 된다.

관계가 명확하게 정의되어 있지 않을 경우에는 데이터의 무결성을 보장받을 수 없다. 또

한 ERD를 통해서 분석/설계된 시스템의 효율성도 검토하게 된다.

식별자 정의

주식별자 정의는 전체 데이터 모델의 복잡성을 결정하는 중요한 요소이다.

- 해당 업무에서 자주 이용되는 속성을 주식별자로 지정한다.
- 속성값의 길이가 가변적인 속성은 주식별자로서 적당하지 않다.
  “부서이름” 보다는 “부서코드”를 주식별자로 지정하라
- 속성값이 자주 변하는 속성은 주식별자로서 적당하지 않다.
- 주식별자를 선정하기 위한 속성(필드)의 수를 적게 한다.
- 주식별자에는 Null 데이터가 들어와서는 안 된다.

세부사항 정의

이제 설계된 각각의 엔티티에 입력될 세부사항(속성)을 정의할 단계이다. 아래는 속성을 정의할 때 주의해야 할 원칙이다.

- 각각의 속성은 반드시 하나의 엔티티에 속해야 한다.
- 각각의 속성은 전체 데이터 모델에서 하나의 의미만을 가지고 있어야 한다.

정규화

정규화란 데이터 모델을 보다 효율적으로 개선시켜나가는 과정을 뜻한다. 일반적으로 정규화는 중복된 데이터를 삭제하는 것이 주 목적이다. 정규화는 1차에서 5차까지의 단계로 나누어 지며, 5차 정규화는 실무에서 거의 사용하지 않고 있다.

- 1차 정규화
  하나의 주식별자를 기준으로 여러 값을 가진 속성은 존재할 수 없다.

고객번호	고객명	구매물품
1	류종택	면도기
		비누
2	이미정	화장품

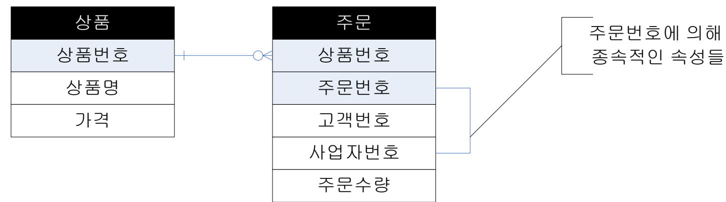
[그림 5] 1차 정규화 대상

고객번호(PK)	고객명
1	류종택
2	이미정

고객번호(PK)	일련번호(PK)	구매물품
1	1	면도기
1	2	비누
2	1	화장품

[그림 6] 1차 정규화 이후

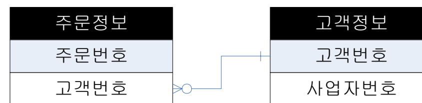
- 2차 정규화
  모든 속성은 주식별자에 종속적이어야 한다.



[그림 7] 2차 정규화 대상

[그림 8] 2차 정규화 이후

- 3차 정규화
  다른 속성에 종속적인 속성은 분리되어야 한다.



[그림 9] 3차 정규화 이후

- 4차 정규화
  N:M의 관계 해소

- 5차 정규화
  설명 생략

반정규화

반정규화란 정규화를 통해서 제거된 중복데이터를 고의로 입력하는 작업을 뜻한다. 정규화가 잘되어 있는 모델의 경우 무결성이 보장되는 장점이 있지만, 정규화가 잘되어 있을 경우 성능이 오히려 떨어질 수 있다. 이때, 성능 자체가 큰 이슈가 되었을 때는 반정규화를 통해서 성능을 향상시킬 수 있다.

정규화와 반정규화는 시스템의 무결성과 성능이라는 두 가지 이슈 사이에서 적절한 선택을 통하여 균형을 잡았을 때 빛이 나게 된다. 정규화는 정합성과 무결성을 보장하는 대신 성능에 저하를 가져올 수 있고, 반정규화는 성능과 모델의 단순화에 대한 이점이 있지만 무결성

저하로 인하여 시스템의 안정성을 해칠 수가 있다.

이에 대한 간단한 예는 "데이터베이스 설계 시 유의사항"에서 다루겠다.

검증

이제는 지금까지 분석 과정을 통하여 설계된 데이터베이스 구조가 의도된 것처럼 진행이 되었는 지를 파악하여야 한다. "어떻게 또는 무엇을 검증해야 하는 가?"에 대하여 상당히 많은 방법들이 존재하겠지만 여기서는 CRUD 분석을 통한 검증방법만을 소개하도록 하겠다.

이어서 설명할 "데이터베이스 설계 시 유의사항"에 대한 점검도, 검증의 한 방법이라고 할 수 있겠다.

CRUD 분석은 프로세스와 엔티티의 상관관계를 이용하여 구축된 데이터베이스 시스템을 검증할 수 있는 방법이다. 아래의 토표처럼 각 프로세스 마다 사용하는 엔티티를 표기하고, 각각의 프로세스가 해당 엔티티를 생성(C), 조회(R), 변경(U), 삭제(D) 하는가에 대한 여부를 표기한다.

프로세스/엔티티	고객	주문	제품	사용후기
가입신청	C			
제품주문	R	C	RU	
제품등록			C	
제품정보 보기			R	R

이후 아래와 같은 항목들을 점검하여 설계에 이상이 없는 가를 확인하게 된다.

- 모든 엔티티 타입에 CRUD가 한 번 이상 표기되었는가?
- 모든 엔티티 타입에 C가 한 번 이상 존재하는가?
- 모든 엔티티 타입에 R이 한 번 이상 존재하는가?
- 모든 단위 프로세스 하나 이상의 엔티티 타입에 표기가 되었는가?
- 두 개 이상의 단위 프로세스가 하나의 엔티티 타입을 생성하는가?
  (이 경우 반드시 잘못되었다기 보다 로직의 검토 대상이 된다)

Primary Key를 가볍게

사용자 테이블을 구축할 때 주로 “Primary Key”로 사용하는 것은 UserID일 것이다. 하지만, 문자열은 숫자에 비해 그 데이터 사이즈가 크기 때문에 인덱스 효율이 떨어진다. 일반적인 코드 테이블에서는 그 레코드 숫자가 작기 때문에 이러한 것을 무시할 수 있으나, 사용자 테이블의 레코드 수가 많아진다면 문제가 될 수가 있다.

이러한 경우에는 일련번호 필드를 새로 추가해서 “Primary Key”로 사용하는 것이 효율적이다.

또한, 필드를 조합하여 “Primary Key”로 사용할 경우에도 그 크기가 너무 커지지 않도록 조심해야 한다.

주민번호와 사원번호 등에 대한 고정관념

가끔 필드타입을 결정하는 데 있어서, 사원번호를 문자열 형태로만 만드는 경향이 있다. 이것들을 숫자형태로 바꾼다면 인덱스의 효과 면에서나 테이블이 차지하는 용량 면에서나 많은 이점을 얻을 수 있다.

- 주민번호는 가운데 '-'를 필드에 반드시 넣을 필요가 있을 까?
- 사원번호는 '000001'처럼 '0'이라는 문자가 반드시 필드에 있어야 할 까?
- 사원번호는 영문과 함께 반드시 부서이름을 설정해줘야 할 까?

테이블이 중복되면 무조건 역효과를 가져올 것인 가?

통계처리와 같은 경우를 생각해보면 해당 월이든 기본 단위 외에는 데이터가 변하지 않는다. 이런 경우 이미 고정된 범위를 미리 집계한 테이블을 생성하고 추가될 부분만 필요 시마다 생성한다면, 퍼포먼스는 매우 향상될 것이다. 만약 빈번한 사용이 없는 통계데이터라면 분리할 필요는 없다.

만약 월마다 또는 기본 단위마다 서로 상관관계가 없는 데이터가 생성된다면 기본단위 별로 테이블을 잘라내서 변경 시 걸리는 부하를 줄일 수 있다.

테이블의 분할과 통합

- 만약 빈번하게 조인을 해야 하는 테이블이라고 한다면 통합에 대한 재 검토가 필요하다.
- 분할과 조합에 영향을 받는 것은 Record의 변경 시 보다는 조회가 더 심하다.
  예를 들어 "학과테이블 + 출석테이블"과 같은 경우 만약 전체학과에 학생에 대한 출석현황을 빈번하게 필요로 한다면 분할보다는 조합하는 것이 좋다. 반대로 학과별의 경우라면 분할이 유리하다.
- 분할이 필요한 경우
  - 테이블들을 조인해야 하는 경우가 적은 경우 (따로 사용을 많이 할 경우)
  - 테이블 마다 사용권한 등의 설정이 다른 경우
- 테이블 통합의 장점
  - 조회하는 작업이 간편해 진다.
  - 엔티티타입간 중복성이 제거된다.
  - 동일한 규칙을 가진 업무처리를 단일 엔티티로 표현이 가능하다.
  - 구조가 단순해 진다.
- 테이블 통합의 단점
  - 확장성의 침해 받을 수 있다.
  - 업무흐름을 이해하는데 어려워진다.
  - 시스템 성능이 저하될 수도 있다.
  - 속성에 제약을 걸지 못하는 경우가 발생한다.
  - 검색조건이 늘어날 가능성이 많다.
  - SQL문이 복잡해 지거나 작성하기 힘들어진다.

모든 테이블은 반드시 서버에 있어야 한다?

만약 코드에 관련된 테이블들이 변경횟수가 매우 적다면 구태여 서버에 두려고 할 필요는 없다. 클라이언트에 복사해서 사용하는 방법을 적극적으로 검토한다.

특히, 우편번호와 같이 레코드 수가 테이블을 빈번히 사용해야 하는 경우라면 클라이언트에 복사본을 두고 작업하는 것이 훨씬 능률적이다.

필드의 추가가 고려되는 경우

- 자주 계산되는 필드
  계산필드는 View 테이블이나 델파이의 Calculated Field 등을 이용하는 경우가 많지만 계산에 의한 부하가 많은 경우에는 아예 계산된 필드를 생성한다. 이는 기존에 계산된 데이터가 필요한 경우에도 도움이 된다.
- Flag를 이용하여 조인 등의 시간을 절약할 수 있는 경우
  미수요금이 있는 지 없는 지를 검사해야 할 경우. 전체 입금과 부과금을 계산하여 마이너스인지를 항상 점검해야 한다면 Flag필드를 만들어 주는 것이 좋다. 특히, 이러한 조건이 Case문과 같이 다양한 값을 가져야 할 때 효과적이다.
  "최근 1개월 이전에 물건을 산 적이 있는 사용자"에 대한 작업을 한다고 가정하면 그 효과는 극적이다. 매 레코드 마다 이것을 계산하는 것과 미리 Flag를 사용하여 작업하는 것의 차이는 엄청나게 크다.
- 날짜 필드의 중복
  Flag의 경우와 동일하다고 볼 수 있다. 만약 매주 시작되는 유료강좌가 있다고 가정하고 사용자가 이것을 등록하기 위한 프로세스를 생각하자. 이때, 각 주 별로 통계를 내거나 하는 프로세스라면 날짜를 통해서 몇 번째 주인지를 항상 계산하는 것은 효율이 없다.

Index 설계 시 유의 사항

- 한 Field에 입력될 내용의 종류가 적으면 그 Field는 Index를 만들지 않는다.
  예를 들면 성별 Field와 같은 경우이다.
- Data의 양(=Record 수)이 적으면 Index를 만들지 않는다.
- 변경이 많은 Field는 Index를 만들기를 조심한다.
- 변경이 적고, 검색이 많은 Field는 Cluster 생성을 고려한다.
- 변경이 주로 되며 Batch작업이 많은 Table의 경우는 Index를 만들지 않는다.
  Batch 작업 전에 인덱스를 삭제하고 종료 후에 인덱스를 생성하는 방법도 고려할 수 있다
- 결합 Index를 생성할 때는 검색이 많은 Field를 항상 먼저 쓴다.
- 결합 Index를 생성할 때는 인덱스 효율이 좋은 필드를 먼저 쓴다.
  내용의 종류가 많은 필드가 인덱스 효율이 좋다.
- 하나의 Table에 5개 이상의 Index를 생성해야 하는 경우라면 설계를 재검토 한다.
- 빈번하게 Join을 할 필요가 있을 때, 해당 Field의 Index를 생성한다.
- Index가 가해지는 필드는 가능한 Null값이 없어야 한다.

Join 시 유의 사항

- Table Join 시에는 Record의 수가 적은 것부터 Join한다.
- 다량의 Table과 다중 Join이 필요할 시에는 Cluster를 생성한다.

검색 시 필드는 연산하지 않는다

- 인덱스가 적용되지 못하는 경우

  Select * from Table1 where SUBSTR(Name, 1, 2) = '류'

  Select * from Table1 where Score * 10 >= 90
- 인덱스가 적용되는 경우

  Select * from Table1 where Name Like '류%'

  Select * from Table1 where Score >= 90 / 10

계산이 필요한 숫자형 필드는 0을 디폴트로 지정한다

Null Data로 인해 Sum이나 Avg와 같은 숫자 연산이 되지 않아서 생기는 논리적이 오류를 방지할 수 있다. 쿼리문을 통한 결과값을 확인하는 조회의 경우에는 큰 문제가 되지 않으나, 이것을 토대로 계산을 하는 "Stored Procedure" 등을 작성할 경우에는 그 과정에 대한 가시성을 확보할 수가 없어, 찾기 어려운 에러를 유발하게 된다.