

I. DBMS - Overview

- 데이터베이스: related data의 collection 이다.
- 데이터: 정보를 생산하기 위하여 처리될 수 있는 facts와 figures의 집합체 이다.

대부분의 데이터는 기록 가능한 사실을 나타낸다. 데이터는 사실에 기초한 정보를 생산하는데 도움을 준다. 예를 들어, 만일 우리가 모든 학생의 점수에 관한 데이터를 갖고 있다면, 우리는 상위자들과 평균점수를 결정할 수 있다.

- DBMS: 정보를 보다 쉽게 검색, 조정, 생산하기 위한 방법으로 데이터를 저장한다.

1. Characteristics

전통적으로 데이터는 파일 포맷으로 조직화 된다. DBMS는 새로운 개념이 아니며, 데이터 관리 분야의 전통적인 스타일에서 나타나는 단점을 극복하기 위하여 모든 관련연구가 진행되었다. 오늘날 DBMS의 특징은 다음과 같다:

1) Real-world entity

오늘날 DBMS는 설계 DBMS자인에 있어서 보다 현실적이며 실세계의 엔티티들을 사용한다. 이것은 또한 행위와 속성 역시 사용한다. 예를 들어, 학교 데이터베이스는 엔티티로서 학생을 그리고 속성으로서 그들의 나이를 사용할 수 있다.

2) Relational-based tables

DBMS는 테이블을 작성하기 위하여 엔티티들과 그것들간의 관계들을 허용하고 있다. 이용자는 이러한 테이블을 이름만 보고서도 데이터베이스의 구조를 이해할 수 있다.

3) Isolation of data and application

데이터베이스 시스템은 그것의 데이터와는 전혀 다르다. 데이터베이스는 능동적인 엔티티인 반면에 데이터는 데이터베이스에서 작동하고 조직화되므로 수동적인 것이라 말할 수 있다. DBMS는 또한 그 자체의 처리(process)를 원활하게하기 위하여 데이터의 데이터인 메타데이터를 저장하기도 한다.

4) Less redundancy

DBMS에서는 정규화(normalization) 규칙에 따라, 관계(relation)의 속성 중에서 어떤 것이 값에 있어서 중복될 경우에, 그 관계를 쪼갬다. 정규화란 데이터의 잉여성을 줄이기 위한 과학적이고도 수학적인 절차이다.

5) Consistency

일관성이란 데이터베이스의 모든 관계에서 모순이 존재하지 않는 상태를 말한다. 모순된 상태

로 데이터베이스를 남겨놓으려는 시도를 탐지할 수 있는 많은 방법과 기법이 존재한다. DBMS를 파일처리 시스템과 같은 과거 형태의 데이터 저장 어플리케이션과 비교해 보면, 훨씬 더 많은 일관성을 제공한다는 것을 알 수 있다.

6) Query Language

DBMS는 데이터를 검색하고 조정하는데 있어서 매우 효율적인 쿼리 언어를 갖추고 있다. 이용자는 다양한 데이터를 검색하는데 필요한 만큼의 다양한 filtering options를 채택할 수 있으나, 전통적인 파일처리시스템을 사용하는 곳에서 이러한 작업은 불가능하다.

7) ACID Properties

DBMS는 Atomicity, Consistency, Isolation, Durability 줄여서 ACID의 개념에 따르고 있다. 이러한 개념들은 DBMS에서 데이터를 조정하는 transactions에 적용되고 있다. 다중의 거래환경과 오류할 경우를 대비하여 ACID 성질은 데이터베이스를 건강하게 유지하는데 도움을 준다.

8) Multiuser and Concurrent Access

DBMS는 다중 사용자 환경을 지원하며 그들이 병렬적으로 데이터에 접근하여 조정할 수 있도록 허용하고 있다. 비록 이용자들이 동일한 데이터 아이템을 다루려고 시도할 때 거래상 제한이 존재하더라도, 이용자들은 항상 그런 것들에 대하여 알지 못한다.

9) Multiple views

DBMS는 다양한 이용자를 위한 다중의 뷰(views)를 제공한다. 도서관 대출실의 이용자는 수서실 이용자와는 다른 데이터베이스 뷰를 사용한다. 이러한 기능으로 이용자는 자신들의 필요에 따라 데이터베이스의 집중적인 뷰를 사용할 수 있다.

10) Security

다중의 뷰와 같은 기능은 이용자가 다른 이용자나 부서의 데이터에 접근하는 것을 방지하는 어느 정도의 보안성을 제공한다. DBMS는 데이터베이스에 데이터를 입력할 때 그리고 나중에 그 데이터를 검색할 때 제한조건을 설정하는 방법을 제공한다. DBMS에서는 다양한 차원의 보안 기능을 제공하여 다양한 이용자가 다양한 기능을 갖춘 다양한 뷰를 사용할 수 있도록 한다. DBMS는 전통적인 파일 시스템처럼 DBMS스크에 저장하지 않으므로, miscreants(악당)이 암호를 해독하기가 매우 어렵다.

2. Users

전형적으로 DBMS에는 다양한 목적으로 이것을 사용하려는 다양한 권리와 허가권을 가진 사람들이 있다. 어떤 이용자는 데이터를 검색하고 다른 이용자는 그것을 백업한다. DBMS의 이용자는 크게 다음과 같이 범주화할 수 있다:



1) Administrators

행정가는 DBMS를 유지관리하며 데이터베이스를 행정하는데 책임을 진다. 이들은 이것의 용도를 살펴보고 누가 이것을 이용해야하는지에 대한 책임을 갖는다. 이들은 이용자들의 접근 profiles를 만들고 독립성 유지와 보안 강화를 위해 제한요소를 적용한다. 행정가는 또한 시스템 라이선스, 필수 도구, 그리고 유지관리에 필요한 하드 및 소프트웨어와 같은 DBMS 자원을 감독한다.

2) Designers

DBMS 디자이너란 데이터베이스의 DBMS자인 분야에서 실제로 작업하는 사람들의 그룹이다. 이들은 유지관리되어야 할 데이터의 유형과 포맷에 대하여 밀착 감시한다. 이들은 모든 엔티티, 관계, 제한조건, 뷰 등에 대해 식별할 수 있도록 DBMS자인 한다.

3) End Users

최종 이용자인 DBMS의 이점들을 실제로 수확하는(reap) 사람들이다. 최종 이용자는 logs와 market rates에 관심을 갖고 있는 단순 이용자에서부터 기업분석가와 같은 전문 이용자까지 그 범위가 다양하다.

II. DBMS -ARCHITECTURE

DBMS의 DBMS자인은 그것의 구조에 의존한다. 이것은 중앙집중식, 분산식, 또는 계층식이 될 수 있다. DBMS의 구조는 단일 tier(층) 또는 다중의 티어처럼 보여지기도 한다. n-층 구조는 전체 시스템을 서로 관련이 있지만 독립적인 n modules로 나누는데, 이 모듈들은 독립적으로 modified, altered(개조), changed, 또는 replaced 될 수 있다.

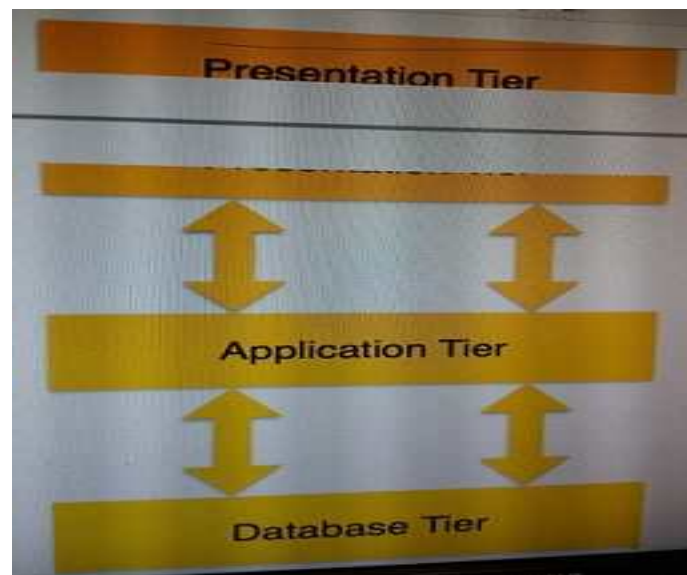
1-티어 구조에서, DBMS는 이용자가 직접적으로 DBMS에 앉아서(sit on) 사용하는 유일한 엔티티이다. 여기서 일어나는 어떤 변화들은 직접적으로 DBMS 그 자체에서 이루어질 것이다. 이것은 최종 이용자를 위한 간단한 도구들을 제공하지는 않는다. 데이터베이스 DBMS자이네

들과 프로그래머들은 대체로 단일-층 구조를 선호한다.

만일 DBMS의 구조가 2-티어이라면, 이것은 DBMS에 접근할 수 있는 어플리케이션을 갖고 있어야 한다. 프로그래머들은 그들이 어플리케이션을 이용하여 DBMS에 접근할 수 있는 2-층 구조를 사용한다. 여기 어플리케이션 티어는 운영, DBMS자인, 그리고 프로그래밍과 관련해서 데이터베이스와는 전적으로 독립적이다.

1. 3-tier Architecture

3-티어 구조는 이용자의 복잡성과 데이터베이스에 있는 데이터를 그들이 이용하는 방법을 근거로 그것의 티어들을 서로서로 분리시킨다.



1) Database *Data* Tier

이 티어에서, 데이터베이스는 쿼리 처리 언어와 함께 존재한다. 우리는 또한 이 레벨에서 데이터와 그것들의 제한조건을 정의하고 있는 관계들을 갖는다.

2) Application *Middle* Tier

이 티어에선 어플리케이션 서버와 데이터베이스에 접근하는 프로그램들이 존재한다. 이용자를 위해, 이러한 어플리케이션 티어는 데이터베이스의 요약(abstracted) 뷰를 제공한다. 최종 이용자는 어플리케이션을 벗어난 데이터베이스의 존재를 알지 못한다. 다른 쪽에 있는 데이터베이스 티어는 어플리케이션 티어를 벗어난 다른 이용자 누구도 인지하지 못한다. 따라서, 어플리케이션 layer는 중간에 존재하며 최종 이용자와 데이터베이스의 중재자로 행동한다.

3) User *Presentation* Tier

최종 이용자는 이 티어에서 활동하며 그들은 이 레이어를 벗어난 어떠한 데이터베이스의 존재

에 대하여 아무 것도 알지 못한다. 이 레이어에서, 데이터베이스의 다중의 뷰들이 어플리케이션에 의해 제공될 수 있다. 모든 뷰들은 어플리케이션 tier에 있는 어플리케이션에 의해 생성된다.

다중-tier 데이터베이스의 구조는 그것의 거의 모든 구성 요소가 독립적이어서 독립적으로 변경할 수 있으므로 highly 변경할 수 있다.

III. DBMS -DATA MODELS

데이터 모델에서는 데이터베이스의 논리적 구조를 모델화하는 방법을 정의한다. 데이터 모델은 DBMS에서 추상화(abstraction)를 소개하기 위한 기본적인 엔티티들이다. 데이터 모델은 데이터가 서로 연결되는 방법과 그것들이 시스템 내에서 처리되고 저장되는 방법을 정의한다.

최초의 데이터 모델은 평평한(flat) 데이터 모델일 수 있다. 이 모델에서 사용된 모든 데이터는 똑같은 평면(plane) 속에 들어 있다. 과거의 데이터 모델은 그렇게 과학적이지 못했기 때문에, 그것들은 많은 중복과 갱신 이상(update anomalies)을 발생시켰다(prone).

1. Entity-Relationship Model

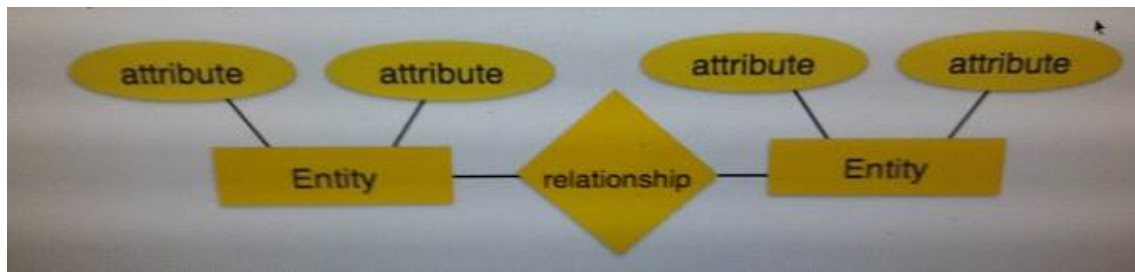
Entity-Relationship Model, 줄여서 ER 모델은 실세계의 엔티티들과 이것들 간의 관계성에 대한 개념을 근거로 삼고 있다. 실세계의 시나리오를 데이터베이스 모델로 공식화하는 동안, ER 모델은 entity set, relationship set, general attributes 그리고 constraints를 생성한다.

ER 모델은 데이터베이스의 개념적 DBMS자인용으로 가장 잘 사용되고 있다

ER 모델은 다음과 같은 것에 의존하고 있다:

- 엔티티들과 이것들의 속성(attributes)
- 엔티티들 간의 관계성(Relationships)

이들 개념들을 살펴보면 다음과 같다:



1) Entity

ER 모델에서 엔티티는 속성(attributes)라 불리는 성질을 갖고 있는 실세계의 엔티티 이다. 모든 속성은 도메인(domain)이라는 값들의 집합체에 의해 정의된다. 예를 들어, 학교도서관 데이터베이스에서 학생은 엔티티로 여겨진다. 학생은 다양한 속성 즉, 이름, 나이, 학급, 등과 같은 속성을 갖는다.

2) Relationship

엔티티들간의 논리적 조합(association)을 관계라 부른다. 관계는 다양한 방법으로 엔티티들을 표현(mapped) 한다. mapping cardinalities는 두 엔티티 간의 조합의 수를 정의한다. mapping cardinalities는 다음과 같이 세분한다:

- one to one
- one to many
- many to one
- many to many

2. Relational Model

DBMS에서 가장 인기있는 데이터 모델은 관계형 모델이다. 이것은 다른 것보다 더 과학적인 모델이다. 이 모델은 first-order predicate logic을 근거로 하고 있으며 하나의 테이블을 n-ary 관계로 정의한다.

SID	SName	SAge	SClass	SSection
1101	Alex	14	9	A
1102	Maria	15	9	A
1103	Mays	14	10	B
1104	Bob	14	8	A
1105	Newton	15	10	B

이 모델의 중요한 특징은 다음과 같다:

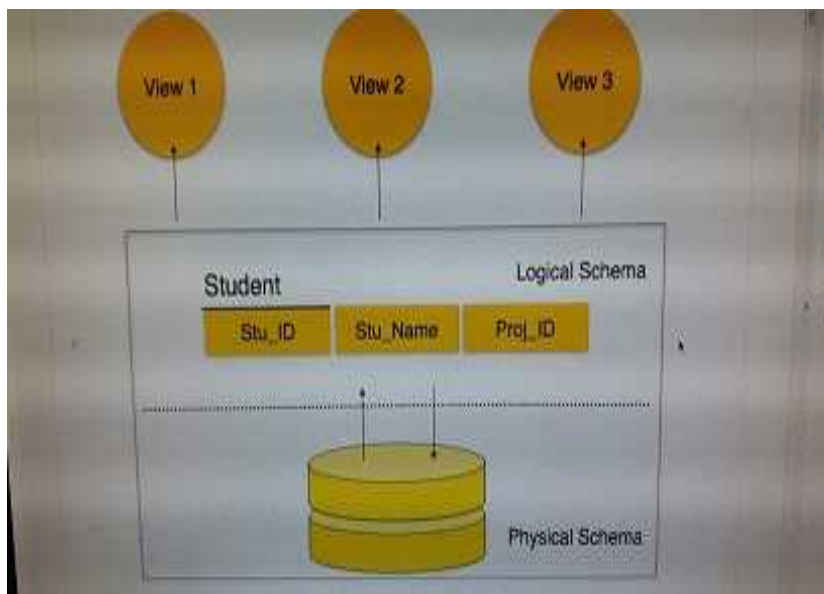
- 데이터는 관계라 부르는 테이블에 저장된다.
- 관계는 정규화될 수 있다.
- 정규화된 관계에서, 저장된 값은 원자(atomic) 값이다.
- 관계의 각 로우(row)는 유일한 값을 포함한다.
- 관계의 각 칼럼(column)은 동일한 도메인에서 나온 값을 포함한다.

IV. DBMS - DATA SCHEMAS

1. Database Schema

데이터베이스 스키마란 전체 데이터베이스의 논리적 뷰를 표현하는 골격 구조(skeleton structure)이다. 이것은 데이터를 조직하는 방법, 데이터간의 관계를 조합하는 방법을 정의한다. 이것은 데이터에 적용할 수 있는 모든 제한조건을 공식화한다.

데이터베이스 스키마는 그것의 엔티티들과 그것들의 관계를 정의한다. 이것에는 스키마의 다이어그램을 사용하여 표현할 수 있는 데이터베이스의 기술적 내역이 포함된다. 프로그래머가 데이터베이스를 이해하고 그것을 유용하게 만드는데 도움을 주도록 스키마를 디자인하는 사람은 바로 데이터베이스 디자이너이다.



데이터베이스 스키마는 크게 두 가지로 범주화할 수 있다:

■ Physical Database Schema

이 스키마는 데이터의 실제적인 저장과 파일, 색인 등과 같은 그것의 저장 형태와 관련이 있다. 이것은 데이터가 제 2차 저장소에 저장되는 방법을 정의한다.

■ Logical Database Schema

이 스키마는 저장된 데이터에 적용되어야 하는 모든 논리적 제한조건을 정의한다. 이것은 테이블, 뷰, 그리고 순수성 제한조건(integrity constraints)을 정의한다.

2. Database Instance

우리가 이들 두 용어를 개별적으로 구분하는 것은 매우 중요하다. 데이터베이스 스키마는 데이터베이스의 골격(skeleton)이다. 이것은 데이터베이스가 결코 존재하는 않을 때 디자인된다. 일단 데이터베이스가 운영되면, 그것에 어떤 변화를 가하기가 매우 힘들다. 데이터베이스 스키마에는 어떤 데이터나 정보를 포함하지 않는다.

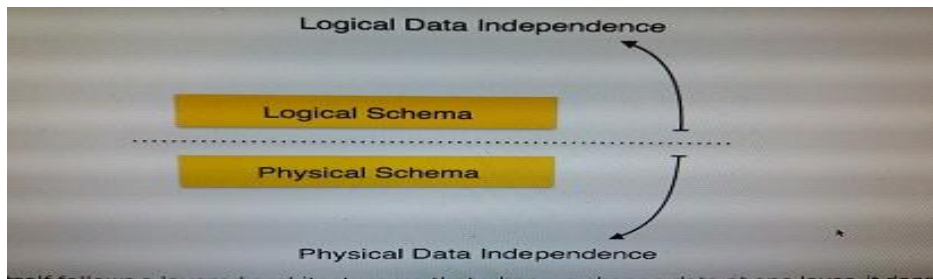
데이터베이스 instance란 어느 특정한 시간에 데이터와 함께 운영중인 데이터베이스의 상태를 말한다. 여기에는 데이터베이스의 snapshot이 포함된다. 데이터베이스 instances는 시간과 함께 변화하는 성질을 가지고 있다. DBMS는 데이터베이스 디자이너가 부여한 모든 validations, constraints, conditions를 부지런히 추적하여, 그것의 모든 instance *state*가 유효한 상태인지를 확인한다.

V. DBMS - DATA INDEPENDENCE

만일 데이터베이스 시스템이 다층 구조가 아니라면, 데이터베이스 시스템에 어떤 변화를 주기가 어렵다. 데이터베이스 시스템은 앞에서 배운 것처럼 다층 구조로 디자인 된다.

1. Data Independence

데이터베이스 시스템은 대체로 이용자의 데이터와 더불어 많은 데이터를 포함하고 있다. 예를 들어, 이것에는 쉽게 데이터를 찾아서 검색하기 위한 메타데이터라 부르는 데이터의 데이터를 포함하고 있다. 그러나 DBMS가 확대됨에 따라, 시간이 지나면서 사용자의 요구를 만족시키기 위해 이것은 변화해야 한다. 만일 모든 데이터가 의존적이라면, 이것은 따분하고 매우 복잡한 업무가 될 것이다.



메타데이터 그 자체는 계층적 구조를 가지므로, 우리가 어떤 층에 있는 데이터를 변경하려할 때, 다른 층에 있는 데이터는 영향을 받지 않는다. 이런 데이터는 독립적인 것이지만 서로서로 연결(mapped)되어 있다.

■ Logical Data Independence

논리적 데이터는 데이터를 내적으로 관리하는 방법에 대한 정보를 저장하고 있는 데이터베이스에 대한 데이터(data about database)이다. 예를 들어, 테이블 *relation* 은 데이터베이스에 저장되며, 그것의 모든 제한조건은 관련 *relation*에 적용된다.

논리적 데이터 독립성은 일종의 메카니즘이며, 이것은 디스크에 저장된 실재적인 데이터로부터 자유롭다는 것을 의미한다. 우리가 테이블 포맷을 변경하더라도, 디스크에 있는 데이터는 바뀌지 않아야 한다.

■ Physical Data Independence

모든 스키마는 논리적이며, 실재 데이터는 디스크에 비트 포맷으로 저장된다. 물리적 데이터 독립성이란 스키마나 논리적 데이터에 영향을 끼치지 않고 물리적 데이터를 변경시키는 힘(power)이다.

예를 들어, 하드 디스크를 SSD(Solid-State Drive)로 교체하는 것처럼 저장 시스템 그 자체를 갱신하거나 변경하려는 경우에, 이것이 논리적 데이터나 스키마에 어떠한 영향도 끼치지 않아야 한다.

VI. ER MODEL - BASIC CONCEPTS

ER 모델은 데이터베이스의 개념적 모양(view)를 정의한다. 이것은 실세계의 엔티티들과 이들 간의 조합과 관련된 일을 한다. view 차원에서, ER 모델은 데이터베이스를 디자인하기 위한 훌륭한 선택으로 받아들여지고 있다.

1. Entity

엔티티는 생물이든 무생물이든(animate or inanimate) 쉽게 정의할 수 있는 실세계의 객체(object)이다. 예를 들어, 학교 데이터베이스에서, 학생, 선생, 교과목은 모두 엔티티가 될 수 있다. 이 같은 모든 엔티티들은 그것들의 정체성을 나타내는 몇 가지 속성이나 성질을 갖고 있다.

엔티티 집합(entity set)은 비슷한 유형의 엔티티들의 집단이다. 엔티티 집합에는 유사한 값을 공유하고 있는 속성들을 갖고 있는 엔티티들이 포함될 수 있다. 예를 들어, 학생 집합에는 모든 학생이 포함될 수 있으며, 선생 집합에는 전체 직원들 중에서 모든 선생만을 포함할 수도 있다. 엔티티 집합은 분리(disjoint) 되지 않아야 한다.

2. Attributes

엔티티는 속성이라 부르는 성질에 의해 표현된다. 모든 속성은 값을 갖고 있다. 예를 들어, 학생 엔티티는 속성으로 이름, 학급, 나이를 가질 수 있다.

속성에 할당된 값을 정의하고 있는 도메인이 존재한다. 예를 들어, 학생 이름으로 숫자값을 사용할 수 없다. 이것은 문자여야 한다. 또한 학생 나이에는 음수(negative)가 사용될 수 없다.

3. Types of Attributes

■ Simple attribute

단일속성은 원자 값(atomic value)으로 추가로 세분될 수 없다. 예를 들어, 학생의 전화번호는 10디지트로 이루어진 원자 값이다.

■ Composite attribute

복합속성은 한 개이상의 단일속성으로 구성된다. 예를 들어, 학생의 완전이름은 성과 명으로 구성된다.

■ Derived attribute

유도속성은 물리적 데이터베이스에는 존재하지 않지만 이것들의 값이 데이터베이스에 있는 다른 속성으로부터 유도되는 속성이다. 예를 들어, 도서관에서 수서한 책들 평균구입가는 데이터베이스에 직접 저장되지는 않지만, 이것은 도서 가격 속성을 활용하여 구할 수 있다.

■ Single-value attribute

단일값속성은 단일값만을 갖는 속성이다. 예를 들어, 도서의 청구기호 이다.

■ Multi-value attribute

다중값속성은 한 개 이상의 값을 갖는 속성이다. 예를 들어, 어떤 이용자는 한 개이상의 전화번호, 이메일 어드레스 등을 갖고 있다.

이러한 속성 유형들은 다음과 같은 방법으로 묶을 수 있다.

- simple single-valued attributes
- simple multi-valued attributes
- composite single-valued attributes
- composite multi-valued attributes

4. Entity-Set and Keys

키란 속성 또는 속성의 집합이며, 엔티티 세트 중에서 특정 엔티티를 유일하게 밝히는데 사용된다. 예를 들어, 등록번호는 소자자료들 중에서 특정 책을 식별하는데 사용한다.

■ Super Key

한 개나 그 이상의 속성의 집합으로, 엔티티 세트에서 특정 엔티티를 집합적으로 식별할 때 사용한다.

■ Candidate Key

최소한의 슈퍼키를 후보키라 부른다. 엔티티 세트는 한 개 이상의 후보키를 가질 수 있다.

■ Primary Key

으뜸키는 데이터베이스 디자이너에 의해 선택된 후보키들 중의 하나이며, 이것은 엔티티 세트를 유일하게 식별하는데 사용된다.

5. Relationship

엔티티들 간의 조합을 관계라고 부른다. 예를 들어, '사서는 도서관에서 일을 하며, 학생은 이 용자로 등록한다'라는 표현에서 '일을 한다'와 '등록한다'는 관계이다.

6. Relationship Set

유사한 유형의 관계의 세트를 관계 세트라 부른다. 엔티티처럼, 관계 또한 속성을 가질 수 있다. 이러한 속성을 기술 속성(descriptive attribute)라 부른다.

7. Degree of Relationship

관계에 참여하는 엔티티의 수에 따라 관계의 정보가 정의된다.

- Binary = degree 2
- Ternary = degree 3
- n-ary = degree n

! 로마숫자 명칭:

unus=uni, duo=bi, tres=tri, quatuor=quadri, quinque=quintal, sex=si, septem=sept, octo=oct, novem=nonus, decem=deca, de

8. Mapping Cardinalities

카디널리티(집합원소의 개수)는 한 개의 엔티티 세트에 있는 엔티티의 숫자를 정의하며, 이것은 관계 세트를 통해 다른 세트의 엔티티 숫자와 조합을 이룬다.

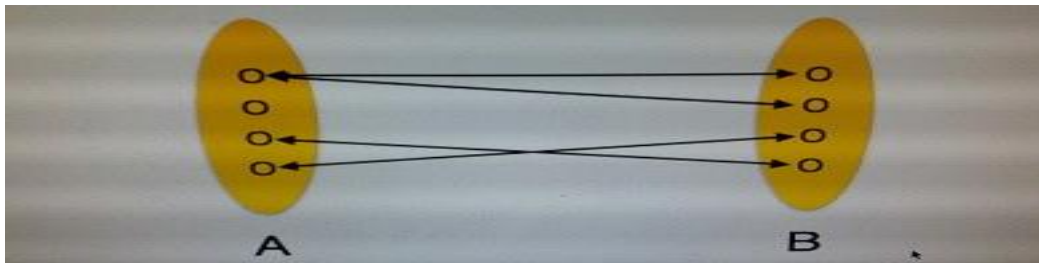
■ One-to-one

엔티티 세트 A에 있는 하나의 엔티티가 기껏해야(at most) 엔티티 세트 B에 있는 하나의 엔티티와 결합한다.



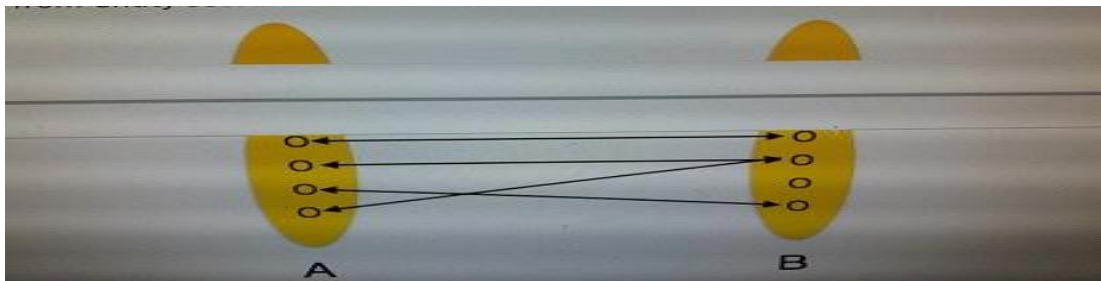
■ One-to-many

엔티티 세트 A에 있는 하나의 엔티티가 엔티티 세트 B에 있는 하나 이상의 엔티티와 결합할 수 있지만, 엔티티 세트 B에 있는 하나의 엔티티는 기껏해야 하나의 엔티티와만 결합할 수 있다.



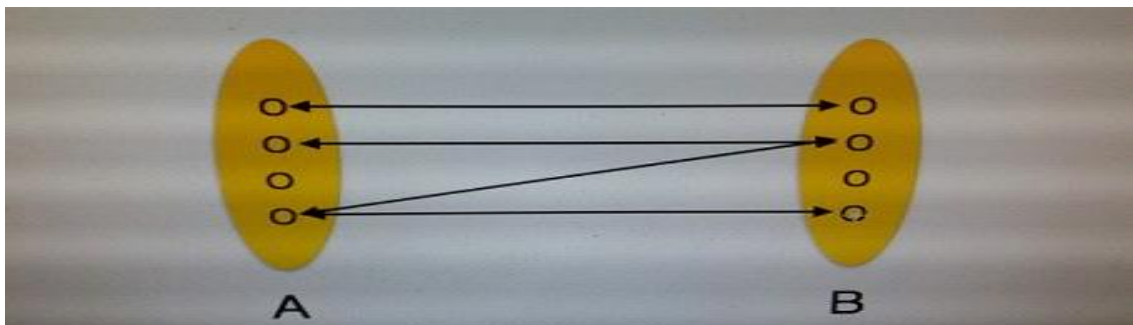
■ Many-to-one

엔티티 세트 A에 있는 하나 이상의 엔티티가 기껏해야 엔티티 세트 B에 있는 하나의 엔티티와 결합할 수 있다. 그렇지만 엔티티 세트 B에 있는 하나의 엔티티는 엔티티 세트 A에 있는 하나 이상의 엔티티와 결합할 수 있다.



■ Many-to-many

A에 있는 하나의 엔티티는 B에 있는 하나 이상의 엔티티와 결합할 수 있다.

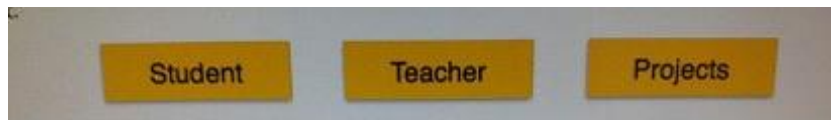


VII. ER DIAGRAM REPRESENTATION

이제 ER 모델을 ER 다이어그램을 통해 표현하는 방법에 대해 알아보자. 어떤 객체(예를 들면, 엔티티, 엔티티의 속성, 관계 세트, 관계 세트의 속성)라도 ER 다이어그램의 도움을 받아 표현할 수 있다.

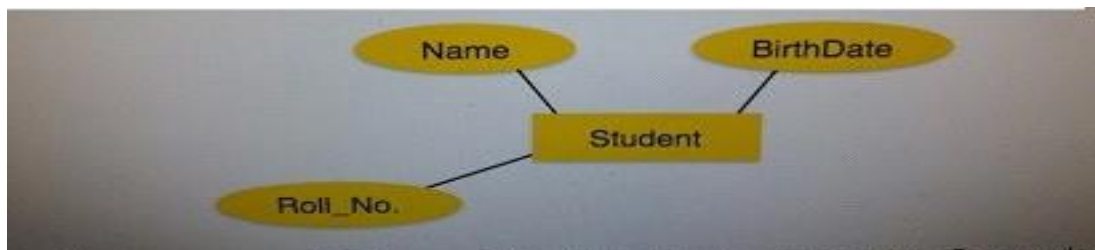
1. Entity

엔티티는 사각형으로 표현한다. 사각형은 그것이 표현하는 엔티티의 이름을 갖는다.

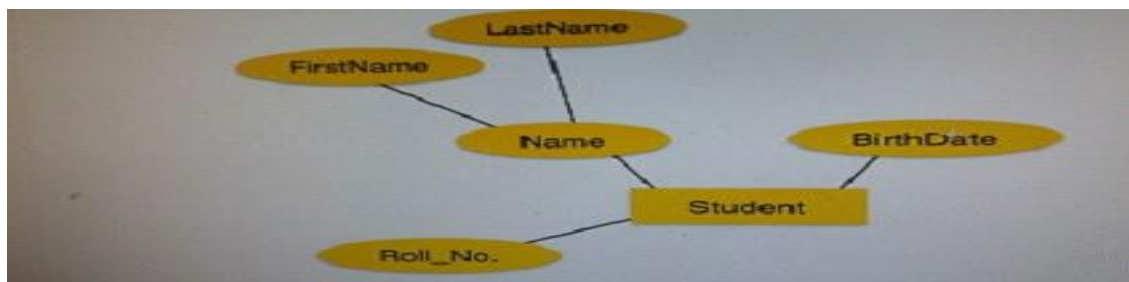


2. Attributes

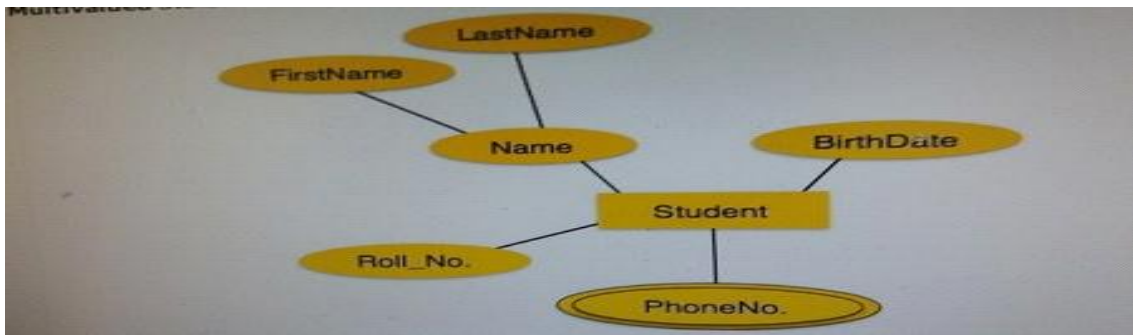
속성은 엔티티의 성질이다. 속성은 타원형(ellipse)으로 표현한다. 각각의 타원형은 하나의 속성을 표현하며 엔티티 *rectangle* 와 직접 연결된다.



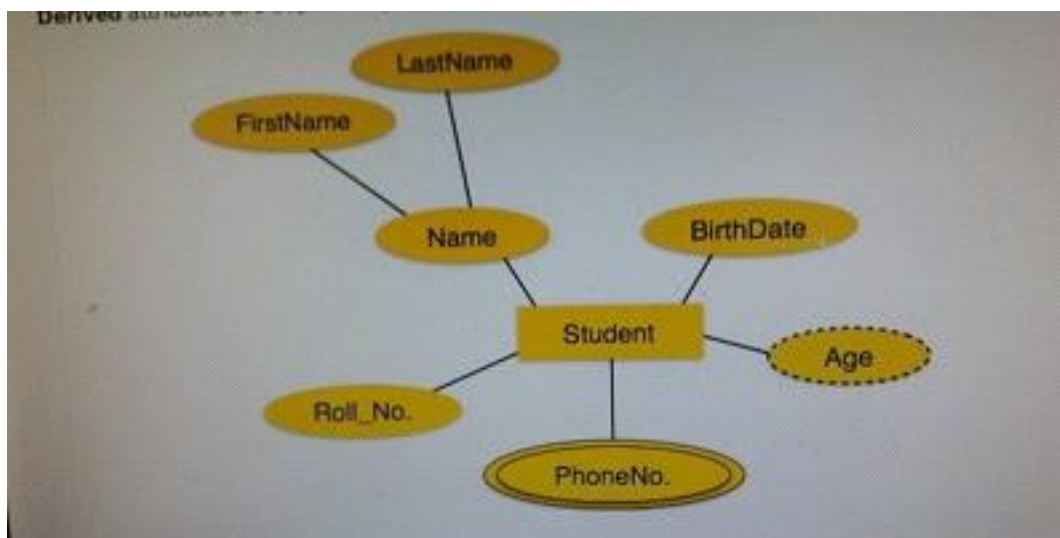
복합속성은 나무구조처럼 세분된다. 각각의 노드는 그것의 속성과 연결된다. 즉, 복합속성은 한 개의 타원형과 연결될 다중의 타원형으로 표현된다.



다중값 속성은 이중선 타원형으로 표현한다.



유도속성은 점선의 타원형으로 표현한다.



3. Relationship

관계는 다이아몬드형 박스로 표현한다. 관계의 이름은 다이아몬드 박스 안에 기록한다. 관계에 참여하는 모든 엔티티 rectangles는 한 개의 라인으로 관계에 연결된다.

■ Binary Relationship and Cardinality

두 개의 엔티티가 참여하고 있는 관계를 binary relationship이라 부른다. 카디널리티는 관계에서 그것과 결합될 수 있는 엔티티의 경우(instance)의 수 이다.

□ One-to-one

엔티티의 단지 한 개만의 경우가 하나의 관계에 결합될 때, '1:1'로 표기한다. 다음의 이미지는 각각의 엔티티에 있는 단지 한 개의 경우만이 한 개의 관계에 결합되어 있다는 것을 보여주고 있다. 이것을 one-to-one 관계라 부른다.



□ **One-to-many**

엔티티의 한 개 이상의 경우가 하나의 관계에 결합될 때, '1:N'이라 표기한다. 다음의 이미지는 왼쪽 엔티티에 있는 단지 한 개의 경우와 오른쪽 엔티티에 있는 한 개 이상의 경우가 관계에 결합되어 있다는 것을 보여주고 있다. 이것을 one-to-many 관계라 부른다.



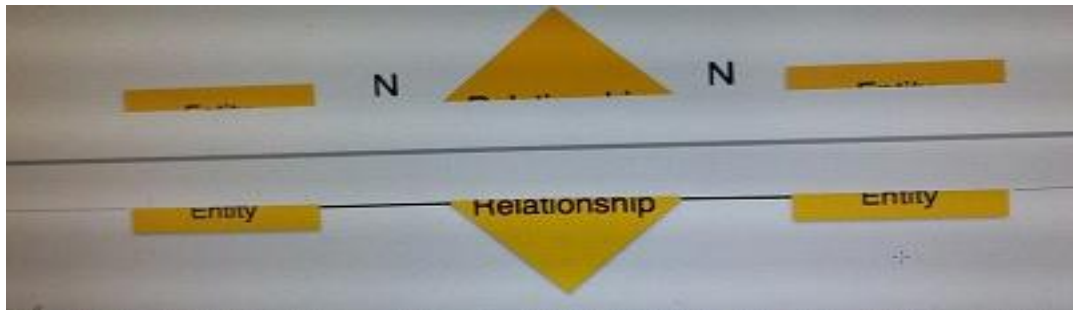
□ **Many-to-one**

엔티티의 한 개 이상의 경우가 하나의 관계에 결합될 때, 'N:1'이라 표기한다. 다음의 이미지는 왼쪽 엔티티에 있는 한 개 이상의 경우와 오른쪽 엔티티에 있는 단지 한 개만의 경우가 관계에 결합되어 있다는 것을 보여주고 있다. 이것을 many-to-one 관계라 부른다.



□ **Many-to-many**

아래의 이미지는 왼쪽 엔티티에 있는 한 개이상의 경우와 오른쪽 엔티티에 있는 한 개 이상의 경우가 관계에 결합된 것을 보여주고 있다. 이것을 many-to-many 관계라 부른다.



■ Participation Constraints

- **Total Participation** - 각 엔티티가 관계에 포함된다. 완전참여는 이중선으로 표현한다.
- **Partial participation** - 모든 엔티티가 관계에 포함되는 것은 아니다. 부분참여는 단선으로 표현한다.



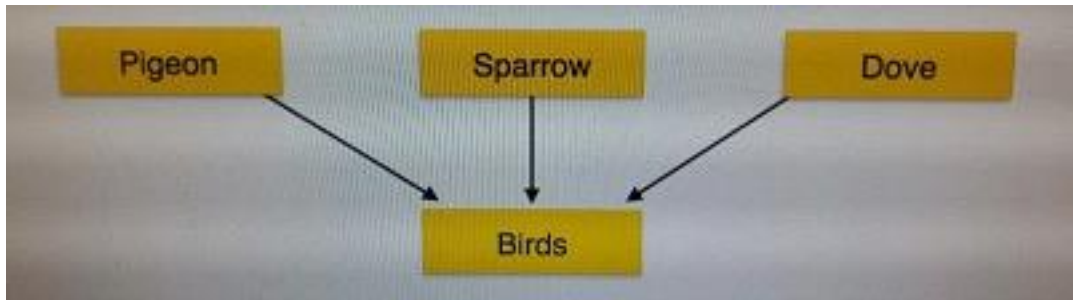
VIII. GENERALIZATION AGGREGATION

ER 모델은 개념적 계층방식으로 데이터베이스 엔티티들을 표현하는데 있어 장점을 갖고 있다. 계층이 올라갈수록, 이것은 엔티티들의 뷰를 일반화시키며, 계층을 내려갈수록 관계된 모든 엔티티의 내역을 제공한다.

이러한 구조에서 위로 올라가는 것을 일반화(generalization)라 하며, 더 많은 일반화된 뷰를 표현하려면 더 많은 엔티티들은 결합되어야 한다(clubbed). 예를 들어, 특별한 학생 이름 Mira를 가지고 모든 학생을 일반화할 수 있다. 다시 말해서, 이 엔티티는 학생일 것이고, 더 나아가면, 그 학생은 사람이다. 이러한 것의 역을 상세화(specialization)라 부르는데, 예를 들면, 사람은 학생이고, 그 학생은 Mika라는 식이다.

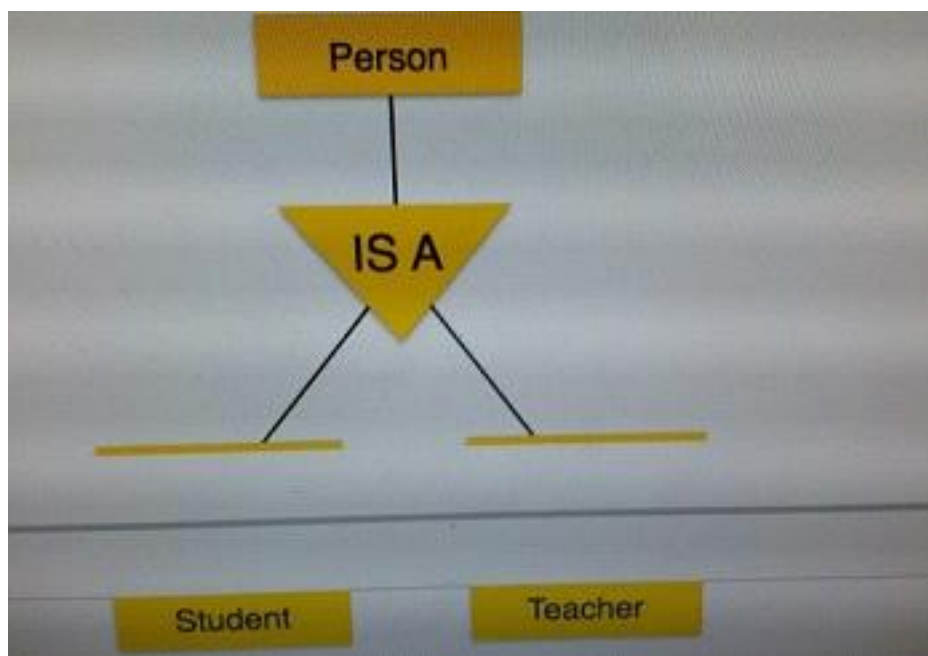
1. Generalization

위에서 말했듯이, 엔티티를 일반화하는 과정을 일반화라 부르며, 이 과정에서 일반화된 엔티티에는 모든 일반화된 엔티티의 성질이 포함된다. 일반화에서, 많은 엔티티들은 그것들의 유사한 성질을 근거로 한 개의 일반화된 엔티티로 모인다. 예를 들어, pigeon, house sparrow, crow, dove는 모두 Birds로 일반화될 수 있다.



2. Specialization

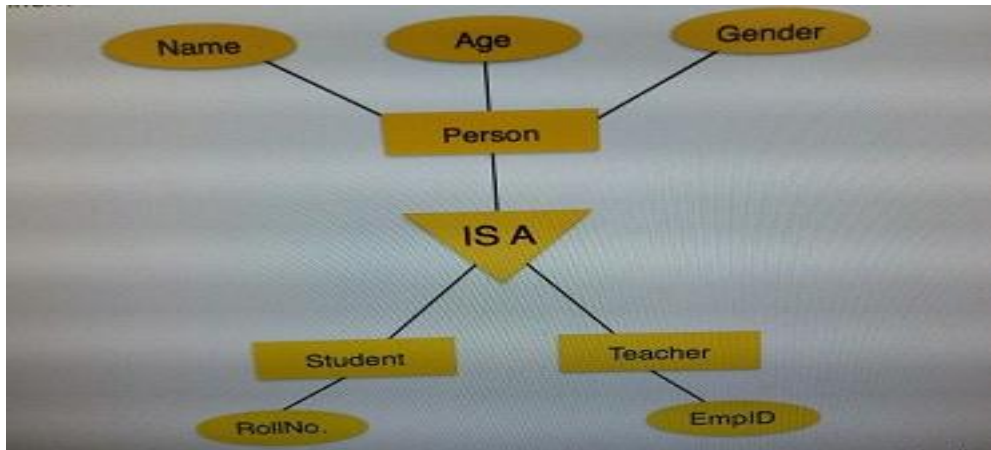
상세화는 일반화의 반대이다. 상세화에서, 한 무리의 엔티티들은 그것들의 특징을 근거로 하위그룹으로 세분된다. 예를 들어, 'Person' 그룹을 살펴보자. 사람은 이름, 생일, 성, 등을 가지고 있다. 이러한 성질은 모든 사람에겐 공통인 것이다. 그러나 회사에서, 사람은 자신들의 역할에 따라 직원, 고용주, 소비자, 또는 판매자로 구분할 수 있다.



이것과 유사하게, 학교 데이터베이스에서, 사람들은 엔티티로서 학교에서 자신들의 역할을 근거로 선생, 학생, 직원으로 세분될 수 있다.

3. Inheritance

우리는 위에서 설명한 ER 모델의 특징 모두를 사용하여 object-oriented programming으로 된 objects의 classes를 만들 것이다. 엔티티들의 내역은 일반적으로 이용자는 알 수 없다: 이러한 과정은 추상화(abstraction)라 부른다.



예를 들어, 이름, 나이, 성별과 같은 Person class의 속성들은 Student나 Teacher와 같은 lowed-level 엔티티들에서 Person class의 유산으로 물려받을 수 있다(inherited).

IX. CODD'S 12 RULES

데이터베이스 시스템의 관계형 모델에 대한 많은 연구를 통해, Dr. Edgar F. Codd는 스스로 12개의 규칙을 마련하였다. Codd에 따르면, 데이터베이스는 참된 관계형 데이터베이스로 간주되기 위해서는 복종(obey)해야 한다는 것이다.

이 규칙들은 그것의 관계형 성능만을 사용하여 저장된 데이터를 관리하는 어떠한 데이터베이스 시스템에도 적용될 수 있다. 이것은 모든 다른 규칙의 기본으로 사용될 수 있는 기초적 규칙이다.

Rule 1: Information Rule

데이터베이스에 저장된 데이터는 이용자 데이터나 메타데이터일 수 있으며, 어떤 테이블 셀은 하나의 값을 가져야 한다. 데이터베이스에 있는 모든 것은 테이블 포맷에 저장되어야 한다.

Rule 2: Guaranteed Access Rule

모든 단일 데이터 요소 즉, *value* 는 테이블 이름, 오픈키 즉, *rowvalue*, 그리고 속성이름

즉, *columnvalue* 와의 결합하여 논리적 접근이 보장되어야 한다. pointers와 같은 어떠한 다른 수단도 데이터 접근에 사용되지 않아야 한다.

Rule 3: Systematic Treatment of NULL Values

데이터베이스에서 NULL 값은 체계적이고 획일적으로(uniform) 취급되어야 한다. 이것은 매우 중요한 규칙인데, 그 이유는 NULL은 데이터가 분실되었거나, 데이터가 알려지지 않았거나, 데이터를 적용할 수 없는 것 중의 하나로 해석될 수 있기 때문이다.

Rule 4: Active Online Catalog

데이터베이스 전체에 대한 구조설명서(structure description)는 접근권한을 가지고 있는 이용자만 접근할 수 있는 data dictionary라고 하는 온라인 카탈로그로 저장되어야 한다. 이용자는 자신들이 데이터베이스 그 자체에 접근하는데 사용하는 이 같은 카탈로그에 접근하기 위해 똑같은 쿼리 언어를 사용할 수 있다.

Rule 5: comprehensive Data Sub-Language Rule

데이터베이스는 data definition, data manipulation transaction management 기능을 지원하는 linear syntax를 가지고 있는 언어를 사용하여 접근할 수 있어야 한다. 이 언어는 직접 또는 특정한 어플리케이션을 통해 사용할 수 있다. 만일 데이터베이스가 이러한 언어의 도움없이 데이터에 접근하도록 한다면, 이것은 규정위반(violation)으로 간주될 것이다.

Rule 6: View Updating Rule

데이터베이스의 모든 뷰들은 이론적으로 갱신될 수 있으므로, 시스템에 의해 또한 갱신될 수 있어야만 한다.

Rule 7: High-Level Insert, Update, and Delete Rule

데이터베이스는 고차원의 insertion, updation, deletion을 지원해야 한다. 이것은 단일 row로 제한되지 않아야 한다. 다시 말해서, data records의 세트들에 적용되는(yield) union, intersection, minus 기능들도 지원해야 한다.

Rule 8: Physical Data Independence

데이터베이스에 저장된 데이터는 데이터베이스에 접근하는 어플리케이션과는 독립적이어야 한다. 데이터베이스의 물리적 구조에 대한 어떠한 변화도 외부 어플리케이션의 데이터 접근 방법에 영향을 끼치지 않아야 한다.

Rule 9: Logical Data Independence

데이터베이스의 논리적 데이터는 이것의 이용자 뷰 즉, *application* 과는 독립적이어야 한다. 논리적 데이터에서의 어떠한 변화라도 그것을 사용하는 어플리케이션에 영향을 끼치지 않아야 한다. 예를 들어, 두 개의 테이블이 통합되거나 한 개가 두 개의 서로 다른 테이블로 쪼개지더라도, 이것이 이용자 어플리케이션에 어떠한 영향이나 변화를 주지 않아야 한다. 이것은 적용해야 할 규칙들 중에서 가장 까다로운 것들 중의 하나이다.

Rule 10: Integrity Independence

데이터베이스는 그것을 사용하는 어플리케이션과 독립적이어야 한다. 이것의 모든 순수성 제한조건은 어플리케이션에서의 변화와 상관없이 독립적으로 변경될 수 있다. 이 규칙은 데이터베이스는 the front-end application 그리고 이것의 interface와 독립적이어야 한다는 것이다.

Rule 11: Distribution Independence

최종 이용자는 데이터가 다양한 장소로 분산되어 있다는 것을 알지 못해야 한다. 이용자는 항상 데이터가 단지 한 곳에만 저장되어 있다는 인상을 가도록 해야 한다. 이 규칙은 분산형 데이터베이스 시스템의 기초로 간주되어 있다.

Rule 12: Non-subversion Rule

만일 시스템이 low-level records에 접근할 수 있는 인터페이스를 가지고 있다면, 이 인터페이스가 시스템을 파괴(subvert)하지도, 그리고 보안과 순수성 제한조건을 무시하지(bypass)도 못하도록 해야 한다.

X. RELATION DATA MODEL

관계형 데이터 모델은 으뜸이 되는 데이터 모델이며, 데이터 저장과 처리를 위해 전 세계에서 널리 사용되고 있다. 이 모델은 간단하며 저장 효율성을 높이기 위하여 데이터를 처리하는데 필요한 모든 성질과 성능을 가지고 있다.

1. Concepts

■ Tables

관계형 모델에서, 관계는 테이블의 형태로 저장된다. 이 포맷에는 엔티티들 사이에 있는 관계를 저장한다. 테이블은 로우와 컬럼을 가지고 있으며 로우는 레코드를 컬럼은 속성을 나타낸다.

■ Tuple

관계용의 단일 레코드를 가지고 있는 테이블의 단일 로우를 터플이라 부른다.

■ Relation instance

관계형 데이터베이스 시스템에서 터플들의 제한된(finite) 세트들은 관계 인스턴스로 표현한다. 관계 인스턴스는 중복된 터플을 갖지 않는다.

■ Relation schema

관계 스키마에는 관계이름 즉, tablename, 속성, 그리고 그것들의 이름을 기술한다.

■ Relation key

각 로우는 관계 table에 있는 로우를 유일하게 식별할 수 있는 관계 키라 하는 한 개 이상의 속성을 갖는다.

■ Attribute domain

모든 속성들은 속성 도메인이라고 하는 미리 정의된 값의 범위를 갖는다.

2. Constraints

모든 관계는 유효한 관계일 경우에 가져야할 어떤 조건들이 있다. 이러한 조건들을 relational integrity constraints라 부른다. 여기에는 3 가지의 주요 순수성 제한조건이 있다:

- Key Constraints
- Domain Constraints
- Referential integrity Constraints

□ Key Constraints

관계에는 속성들의 적어도 한 개의 최소한의 subset이 존재하여야 한다. 이것으로 유일하게 튜플을 식별할 수 있어야 한다. 속성들의 이러한 최소한의 subset를 관계용 키라 부른다. 만일 이러한 최소한의 subset이 한 개 이상이라면, 이것들을 candidate keys라고 부른다.

key 제한조건에서 강조하는 것은 다음과 같다:

- ▲ 키 속성이 있는 관계에서, 어떠한 두 개의 터플이라도 키 속성용의 식별가능한 값을 가질 수 없다.
- ▲ 키 속성은 NULL 값을 가질 수 없다.

키 제한조건을 또한 Entity Constraints라고도 부른다.

□ Domain Constraints

속성은 실세계 시나리오에서 특별한 값을 갖는다. 예를 들어, 나이는 단지 양의 정수만 가질 수 있다. 똑같은 제한조건이 관계의 속성들에서 적용될 수 있다. 모든 속성은 특정한 범위의 값을 가질 수 있다. 예를 들어, 나이는 0보다 작을 수 없으며, 전화번호는 0-9이외의 숫자를 가질 수 없다.

□ Referential integrity Constraints

참조무결성 제한조건은 Foreign Keys의 개념과 연관이 있다. 외래 키란 다른 관계를 참조할 수 있는 관계의 키 속성이다.

참조무결성 제한조건에서는 만일 관계가 다른 또는 동일한 관계의 키 속성을 참조한다면, 그 키 요소가 존재해야 한다고 말하고 있다.

XI. RELATIONAL ALGEBRA

관계형 데이터베이스 시스템에서는 이용자가 데이터베이스 인스턴스를 쿼리하는 것을 돕기 위하여 쿼리 언어를 구비하고 있다. 두 가지 종류의 쿼리 언어가 있는데, 하나는 relational algebra 이고 나머지는 relational calculus 이다.

1. Relational Algebra

관계대수란 input으로 관계의 인스턴스를 취하고 output으로 관계의 인스턴스를 제공하는 절차중심의 쿼리언어이다. 이것은 쿼리를 수행하기 위한 여러 가지 연산자를 사용한다. 연산자는 unary이거나 binary일 수 있으며, 이것들은 자신들의 input으로 관계를 받아들이고 다음에 output으로 관계를 제공한다. 관계형 대수는 관계와 관련에서 반복적으로 수행되며, 중간(intermediate) 결과물도 관계로 여겨진다.

관계형 대수의 기본적 기능들은 다음과 같다:

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

1) Select Operation σ

이것은 관계로부터 주어진 조건을 만족시키는 튜플들을 선택한다.

표기법: $\sigma_p r$

σ : 선택 조건;

r : 관계

p : and, or, not과 같은 연산자를 사용할 수 있는 전치사적 논리공식(propositional logic)

formula). 이러한 용어들은 $=$, \neq , \geq , $<$, $>$, \leq 같은 관계형 연산자로 사용할 수도 있다.

예제: Books인 테이블에서,

(1) $\sigma_{subject='database'}(Books)$

Output - 주제가 'database'인 책들을 터플로 선택한다.

(2) $\sigma_{subject='database' \text{ and } price='450'}(Books)$

Output - 주제가 'database'이고 'price'가 450인 책들을 터플로 선택한다.

(3) $\sigma_{subject='database' \text{ and } price < '450' \text{ or } year > '2010'}(Books)$

Output - 주제가 'database'이고, 'price'가 450이거나 출판연도가 2010인 책들을 터플로 선택한다.

2) Projection Operaton Π

이것은 주어진 조건을 만족시키는 칼럼들을 불러온다(project).

표기법: $\Pi A_1, A_2, \dots, A_n \text{ } r$

A_1, A_2, \dots, A_n : 관계 r 의 속성이름들.

관계는 하나의 집합이므로, 중복된 로우들은 자동적으로 제거된다.

예제:

$\Pi_{subject, author} (Books)$

관계 Books로부터 주제와 저자로 명명되어 있는 칼럼들을 선택하여 불러온다.

3) Union Operation \cup

이것은 두 개의 주어진 관계들 간에 binary union을 수행하며 다음과 같이 정의한다:

$$r \cup s = \{ t \mid t \in r \text{ or } t \in s \}$$

표기법: $r \cup s$

r 과 s 는 데이터베이스 관계이거나 관계의 결과 세트 즉, *temporary relation* 이다.

합집합 연산이 유효하려면, 다음과 같은 조건을 충족시켜야 한다:

- ▲ r 과 s 는 동일 수 의 속성을 가지고 있어야 한다.
- ▲ 속성도메인은 양립할(compatible) 수 있어야 한다.
- ▲ 중복된 터플들은 자동적으로 제거되어야 한다.

$\Pi_{author} (Books) \cup \Pi_{author} (Articles)$

Output: 책의 저자이거나 학술기사의 저자이거나 두 가지 모두의 저자의 이름을 불러온다.

4) Set Difference

차집합의 결과는 한 개의 관계에는 나타나 있으나 두 번째 관계에는 없는 터플들이다.

표기법: $r - s$

r 에는 있으나 s 에는 없는 모든 터플들을 찾는다.

$\Pi \text{ author (Books)} - \Pi \text{ author (Articles)}$

Output: 책의 저자이지만 학술기사의 저자가 아닌 터플들을 찾는다.

5) Cartesian Product X

이것은 두 개의 서로다른 관계에 있는 정보를 하나로 결합시킨다.

표기법: $r \times s$

r 과 s 는 관계들이며, 이것들의 결과는 다음과 같이 정의될 수 있다:

$$r \times s = \{ q \mid q \in r \text{ and } t \in s \}$$

$\Pi \text{ author} = \text{'tutorialspoint'}(\text{Books} \times \text{Articles})$

Output: tutorialspoint에 의해 작성된 모든 책과 기사를 보여주는 관계를 제공한다.

6) Rename Operation ρ

관계형 대수의 결과도 역시 관계이지만 이들은 어떠한 이름도 갖고 있지 않다. 재명명 기능은 결과 관계에 이름을 붙이도록 한다. 'rename' 기능은 작은 그리스 문자 ρ 와 함께 명명한다.

표기법: $\rho \times E$

표현 E 의 결과는 x 란 이름으로 저장된다.

이밖에도 추가적인 기능으로는 다음과 같은 것들이 있다:

- Set intersection
- Assignment
- Natural join

2. Relational Calculus

관계대수와 대조적으로, 관계미적은 비 절차적인 쿼리언어이다. 특, 이것은 무엇을 해야하는지

를 말하지만 결코 그것을 해야하는 방법을 설명하진 않는다.

1) Tuple Relational Calculus, TRC

터플 변수의 범위를 filtering 한다.

표기법: $\{ T \mid \text{Condition} \}$

조건을 만족시키는 모든 터플들 T를 돌려준다.

예제:

$\{ T.name \mid \text{Author}(T) \text{ AND } T.article = 'database' \}$

Output: 'database'라는 기사를 쓴 Author의 'name'에 들어 있는 모든 터플들을 돌려준다.

TRC는 계량화할 수 있다. 우리는 Existential \exists and Universal Quantifiers \forall 를 사용할 수 있다.

예제:

$\{ R \mid \exists T \in \text{Authors}(T.article = 'database' \text{ AND } R.name = T.name) \}$

Output: 위의 쿼리는 바로 이전과 동일한 결과를 돌려준다.

2) Domain Relational Calculus, DRC

DRC에서, filtering 변수는 모든 터플 값 대신에, 위에서 TRC에서 했던 것처럼, 속성 도메인을 사용한다.

표기법: $\{ a_1, a_2, a_3, \dots, a_n \mid P(a_1, a_2, a_3, \dots, a_n) \}$

a_1, a_2 는 속성들이고, p 는 내부 속성들에 의해 구축된 공식을 나타낸다.

예제:

$\{ \langle article, page, subject \rangle \mid \langle \rangle \in \text{TutorialsPoint} \wedge subject = 'database' \}$

관계 Tutorialspoint로부터 주제가 'database'인 Article, Page, Subject를 제공한다.

TRC처럼, DRC 역시 existential and universal quantifiers를 사용하여 작성할 수 있다. DRC 역시 관계형 연산자를 포함시킬 수 있다.

Tuple Relation calculus와 Domain Relation Calculus의 표현력은 Relational Algebra와 동등하다.

XII. ER MODEL TO RELATIONAL MODEL

ER 모델이 다이어그램으로 개념화될 때, 이것은 이해하기 쉽도록 엔티티-관계에 대한 멋진 전체모습(overview)을 제공한다. ER 다이어그램은 관계형 스키마와 판박이(mapped) 일 수 있다. 다시 말해서, ER 다이어그램을 사용하여 관계형 스키마를 만들 수 있다는 것이다. 우리는 관계형 모델에 모든 ER 제한조건을 적용시킬 수는 없지만, 근접한 스키마를 제작할 수 있다.

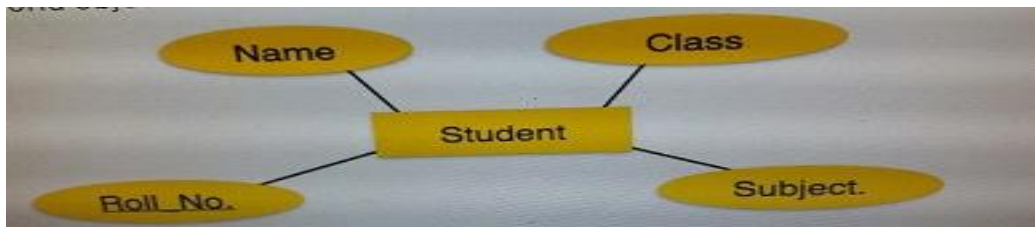
ER Diagram을 Relational Schema로 변환시킬 수 있는 여러 가지 처리절차와 알고리즘이 존재한다. 이것들 중 어떤 것은 자동화되어 있으며, 또 어떤 것은 수동으로 처리해야 한다. 우리는 이제 diagram contents를 relational basics에 그대로 적용하는데(mapped)하는데 초점을 맞출 것이다.

ER 다이어그램의 주요 구성요소는 다음과 같다:

- Entity와 이것의 Attributes.
- Relationship- 엔티티들을 결합시킴.

1. Mapping Entity

엔티티란 몇 개의 속성들을 갖고 있는 실세계의 객체(object) 이다.



□ Mapping Process Algorithm

- ▲ 각각의 엔티티마다 테이블을 만든다
- ▲ 엔티티의 속성들은 테이블의 필드가 되어야 하며, 각각 자신들의 독립된 data types을 가져야 한다.
- ▲ 으뜸키를 선언한다.

2. Mapping Relationship

관계는 엔티티들의 결합이다.

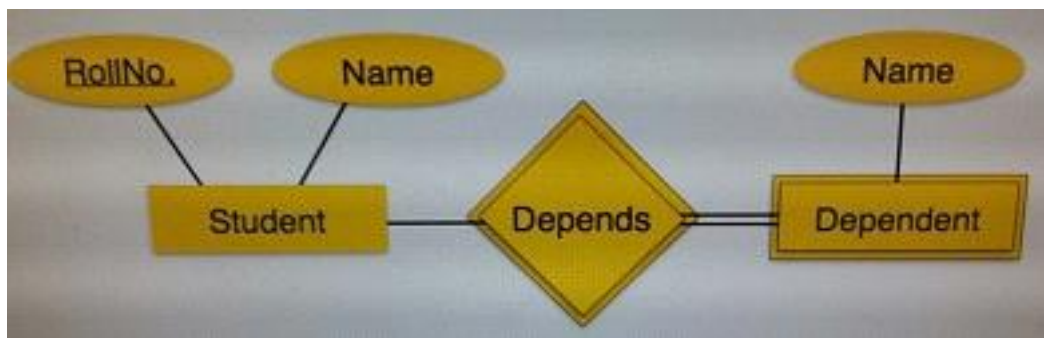


□ Mapping Process

- ▲ 관계용 테이블을 만든다
- ▲ 자신들의 고유한 데이터 타입을 가지고 있는 테이블의 필드로 참여하고 있는 모든 엔티티들의 으뜸키를 추가한다.
- ▲ 만일 관계가 속성을 가지고 있다면, 각각의 속성들을 테이블 필드처럼 추가한다.
- ▲ 참여하고 있는 엔티티들의 모든 으뜸키를 구성하고 있는 한 개의 으뜸키를 선언한다.
- ▲ 모든 외래키의 제한조건을 선언한다.

3. Mapping Weak Entity Sets

약한 엔티티 세트는 어떤 것도 으뜸키와 연결되지 않는 엔티티 세트이다.

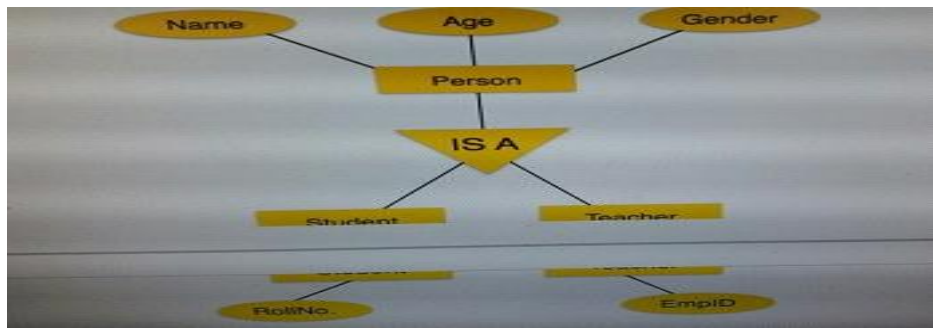


□ Mapping Process

- ▲ 약한 엔티티 세트용의 테이블을 만든다.
- ▲ 이것의 모든 속성들을 필드처럼 테이블에 추가한다.
- ▲ 엔티티 세트를 식별하기 위한 으뜸키를 추가한다.
- ▲ 모든 외래키의 제한조건을 선언한다.

4. Mapping Hierarchical Entities

ER의 specialization과 generalization은 계층적 엔티티 세트들의 형태로 표현된다.



□ Mapping Process

- ▲ 모든 고차원 엔티티용의 테이블들을 만든다.
- ▲ 저차원 엔티티용의 테이블을 만든다.
- ▲ 저차원 엔티티의 테이블에 고차원 엔티티의 으뜸키를 추가한다.
- ▲ 저차원 테이블에 저차원 엔티티들의 모든 속성들을 추가한다.
- ▲ 고차원 테이블의 으뜸키와 저차원 테이블의 으뜸키를 선언한다.
- ▲ 외래키의 제한조건을 선언한다.

XIII. SQL OVERVIEW

SQL이란 관계형 데이터베이스를 위한 프로그래밍 언어이다. 이것은 relational algebra and tuple relational calculus보다 우선하여(over) 디자인되었다. SQL은 모든 중요한 RDBMS와 하나의 패키지를 이루고 있다.

SQL은 데이터 정의어와 데이터 조정어로 구성되어 있다. SQL의 데이터 정의 속성들을 사용하여, 누구나 데이터베이스 스키마를 디자인하고 변경할 수 있으며, 데이터 조정 속성으로는 데이터베이스로부터 데이터를 저장하고 검색할 수 있다.

1. Data Definition Language

SQL에서는 데이터베이스의 스키마를 정의하기 위하여 다음과 같은 명령어들을 사용한다:

1) CREATE

RDBMS로부터 새로운 데이터베이스, 테이블, 뷰를 만든다

예제:

Create database tutorialspoint;

Create table article;
Create view for_students;

2) DROP

RDBMS로부터 명령어, 뷰, 테이블, 그리고 데이터베이스를 제외시킨다.

예제:

Drop object_type object_name;
Drop database tutorialspoint;
Drop table article;
Drop view for_students;

3) ALTER

데이터베이스 스키마를 변경시킨다.

Alter object_type object_name parameters;

예제:

Alter table article add subject varchar;

이 명령어는 관계 **article**에 문자열로 된 **subject**라는 이름을 가진 한 개의 속성을 추가한다.

2. Data Manipulation Language

SQL에는 data manipulation language, DML 가 포함되어 있다. DML은 inserting, updating, deleting을 사용하여 데이터베이스 인스턴스를 변경시킨다. DML은 데이터베이스에서 데이터 변형과 관계된 모든 것에 책임을 진다. SQL은 자신의 DML 부분에 다음과 같은 명령어 세트를 가지고 있다:

- SELECT/FROM/WHERE
- INSERT INTO/VALUES
- UPDATE/SET/WHERE
- DELETE FROM/WHERE

이러한 기본적인 구조는 데이터베이스 프로그래머와 이용자로 하여금 데이터와 정보를 데이터베이스에 입력한 다음, 수많은 검색 기능을 선택하여 효율적으로 검색할 수 있도록 한다.

1) SELECT/FROM/WHERE

□ SELECT

이것은 SQL의 기본적인 쿼리 명령어중의 하나이다. 이것은 관계형 대수의 projection 기능과

유사하다. 이것은 WHERE 조건 절에서 설정한 조건을 근거로 속성들을 선택한다.

□ FROM

이 절(clause)에서는 선택하려거나 프로젝티드하려는 속성들의 근거(argument)가 되는 관계의 이름을 설정한다.

□ WHERE

이 절은 프로젝티드하려는 속성의 품질을 결정하는 술어(predicate) 또는 조건을 정의한다.

예제:

```
Select author_name
From book_author
Where age > 50;
위의 명령어는 관계 book_author로부터 나이가 50세 이상인 저자의 이름을 선택한다.
```

2) INSERT INTO/VALUES

이 명령어는 테이블 relation의 로우에 값을 입력할 때 사용한다.

구문(syntax):

```
INSERT INTO table (column n1 [, column n2, column n3 ... ]) VALUES (value1 [,
value2, value3
... ])
```

또는

```
INSERT INTO table VALUES (value1, [value2, ... ])
```

예제:

```
INSERT INTO tutorialspoint (Author, Subject) VALUES ("anonymous", "computers");
```

3) UPDATE/SET/WHERE

이 명령어는 테이블 relation에 있는 칼럼들의 값을 갱신하거나 변경하고자 할 때 사용한다.

구문:

```
UPDATE table_name SET column_name = value [, column_name = value ...]
[WHERE condition]
```

예제:

```
UPDATE tutorialspoint SET Author="webmaster" WHERE Author="anonymous";
```

4) DELETE FROM/WHERE

이 명령어는 테이블 relation에 있는 한 개 이상의 로우를 제거할 때 사용한다.

구문:

```
DELETE FROM table_name [WHERE condition];
```

예제:

```
DELETE FROM tutorialspoints  
WHERE Author="unknown";
```

XIV. DBMS-NORMALIZATION

1. Functional Dependency

함수 종속성(FD)이란 한개의 관계에 있는 두 개의 속성들 간의 제한조건 세트이다. 함수 종속의 의미를 살펴보면, 이것은 만일 두 개의 튜플들이 속성 A_1, A_2, \dots, A_n 으로 동일한 값들 갖고 있다면, 이러한 두 개의 튜플들은 속성 B_1, B_2, \dots, B_n 에서도 동일한 값을 가져야 한다는 것이다.

함수 종속성은 화살표 즉, \rightarrow 로 표현한다. 즉, $X \rightarrow Y$ 라는 함수 종속 표현은 X 가 함수적으로 Y 를 결정한다는 것을 의미한다. 다시 말해서, 왼쪽에 있는 속성이 오른쪽에 있는 속성의 값을 결정한다는 것이다.

2. Armstrong's Axioms

만일 F 가 함수 종속성의 세트라면, F^+ 로 명명된 F 의 closure(결과, 마감, 종결)는 F 에 의해 논리적으로 의미가 부여된(imply) 모든 함수 종속성의 세트이다. 암스트롱의 공리는 한 세트의 규칙으로 이것이 반복적으로 적용된다면, 함수 종속성의 closure를 생산한다.

■ Reflexive rule(반사의 공리)

Y 가 X 의 부분 집합이면, $X \rightarrow Y$ 이다.

■ Augmentation rule(확대의 공리)

만약 $X \rightarrow Y$ 이면, $XZ \rightarrow YZ$ 이다. 이것은 의존하고 있는 속성을 추가하는 것이지만 기본적인 의존성은 변하지 않는다.

■ Transitivity rule(이행의 공리)

대수의 이행적 규칙과 같다. 만약 $X \rightarrow Y$ 이고 $Y \rightarrow Z$ 이면 $X \rightarrow Z$ 이다.

이 공리에 의해 다음과 같은 부수적 법칙을 유도해 낼 수 있다.

- 합집합의 성질(Union): If $X \rightarrow Y$ 이고 $X \rightarrow Z$ 이면 $X \rightarrow YZ$ 이다.
- 분해의 성질(Decomposition): $X \rightarrow YZ$ 이면 $X \rightarrow Y$ 이고 $X \rightarrow Z$ 이다.
- 유사 이행적 성질(Pseudotransitivity): 만약 $X \rightarrow Y$ 이고 $YZ \rightarrow W$ 이면, $XZ \rightarrow W$ 이다.

3. Trivial Functional Dependency

■ Trivial

만일 함수 종속성 FD $X \rightarrow Y$ 가 유지되고, Y가 X의 부분집합이라면, 이것을 trivial FD 라 부른다. Trivial FD는 항상 유지된다(hold).

■ Non-trivial

만일 FD $X \rightarrow Y$ 가 유지되고, Y가 X의 부분집합이 아니라면, 이것을 non-trivial FD라 부른다.

■ Completely non-trivial

만약에 FD $X \rightarrow Y$ 가 유지되고 x 가 $Y = \Phi$ 를 intersect한다면, 이것을 completely non-trivial라고 부른다.

4. Normalization

데이터베이스 디자인이 완벽하지 않다면, 그것에는 데이터베이스 운영자에게는 악몽과 같은 이상현상(anomalies)이 포함될 수 있다. 이상현상을 갖고 있는 데이터베이스를 운영하는 것은 거의 불가능하다.

■ Update anomalies

만일 데이터 아이템이 흩어져 있고 서로서로 올바르게 링크되어 있지 않다면, 상황을 이상하게 이끌 수 있다. 예를 들어, 우리가 여러 장소에 흩어져 있는 사본을 갖고 있는 한 개의 데이터 아이템을 갱신하고자 할 때, 약간의 인스턴스만을 올바르게 갱신할 수 있는 반면에 소수의 또 다른 것들은 기존의 값을 유지하게 된다. 이 같은 인스턴스들은 데이터베이스를 일상이 없는(inconsistent) 상태로 남겨 둔다.

■ Deletion anomalies

우리가 레코드를 삭제하려 하지만, 그것의 일부가 삭제되지 않는 채로 남아 있는데, 그 이유는 우리가 깨닫지 못하거나, 데이터 또한 어떤 다른 곳에 저장되어 있기 때문이다.

■ Insert anomalies

우리는 결코 존재하지 않은 레코드에 데이터를 삽입하려고 한다.

정규화란 모든 이러한 이상현상을 제거하는 방법이며 데이터베이스를 일관성 있는 상태로 구성하는 것이다.

1) First Normal Form(1 NF)

1 NF에서는 관계인 tables 그 자체의 정의를 정의한다. 이 규칙에서 정의하는 것은 관계의 모든 속성들은 원자 도메인을 가져야 한다는 것이다. 원자 도메인에 있는 값들은 쪼갤 수 없는 단위이다.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

우리는 이것을 1NF로 변경하기 위하여, 아래처럼 관계 table로 재정리할 수 있다.

Course	Content
Programming	Java
Programming	C++
Web	HTML
Web	PHP
Web	ASP

각 속성은 이미 정의된 자신의 도메인으로부터 단지 하나의 단일 값을 갖는다.

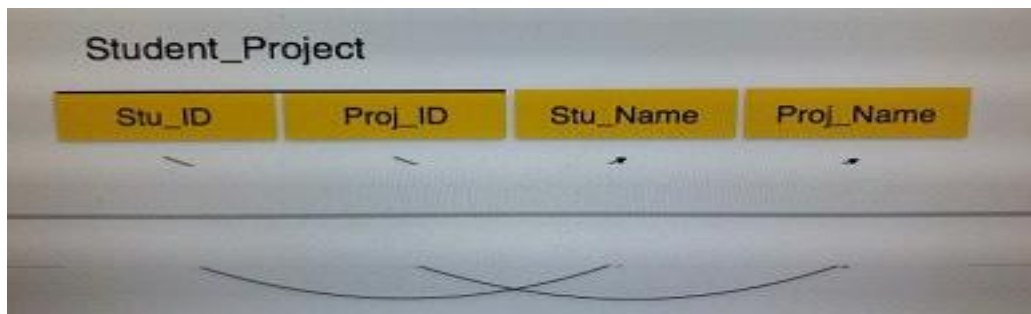
2) Second Normal Form(2 NF)

2NF를 배우기 전에, 다음과 같은 것을 이해하여야 한다:

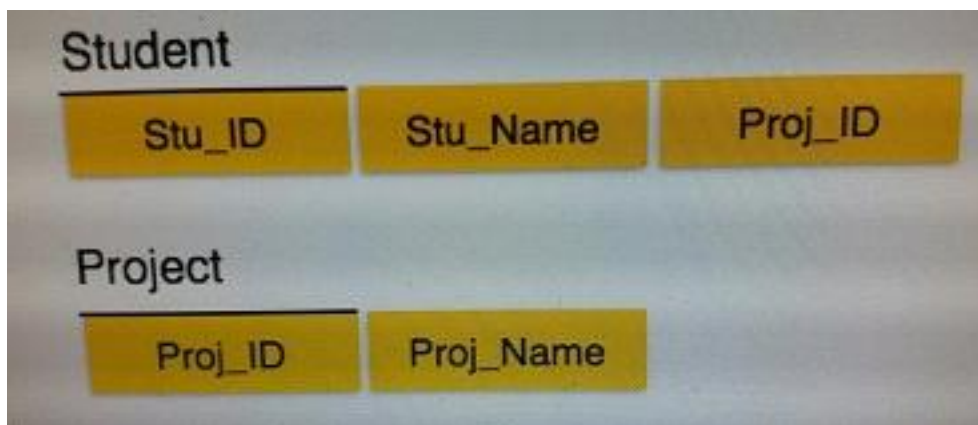
■ Prime attribute - 기본 키(prime-key)의 일부분인 속성을 기본 속성이라 한다.

- **Non-prime attribute** - 기본 키의 일부분이 아닌 속성은 비-기본 속성이라 한다.

만일 우리가 2 NF를 구성하려면, 모든 비-기본 속성이 기본 키 속성에 완전하게 함수적으로 의존하여야 한다. 즉, 만일 $X \rightarrow Y$ 라면, X 의 부분집합 Y 가 올바르게 존재하지 않아야 하는데, 왜냐하면 $Y \rightarrow X$ 역시 참으로 존재하기 때문이다.



우리는 Student_Project 관계를 보고 있으며, 이것의 기본 키 속성들은 Stu_ID와 Proj_ID이다. 규칙에 따라, 비-키 속성 즉, Stu_Name과 Proj_Name은 양쪽에 의존해야 하지만 개별적으로 어떠한 기본 키 속성에만 의존하진 않는다. 그러나, 우리가 알 수 있는 것은 Stu_Name은 Stu_ID에 의해, 그리고 Proj_Name은 Proj_ID에 의해 각자 식별 가능하다. 이러한 것을 partial dependency라 부르지만, 이것은 2 NF에서 허용되지 않는다.



우리는 위의 그림처럼 관계를 두 개로 쪼개었다. 따라서 어떠한 부분적 의존성도 존재하지 않는다.

3) Third Normal Form(3 NF)

관계가 3 NF가 되기 위해서는 2 NF에 있어야만 하고, 아래의 내용을 만족시켜야 한다:

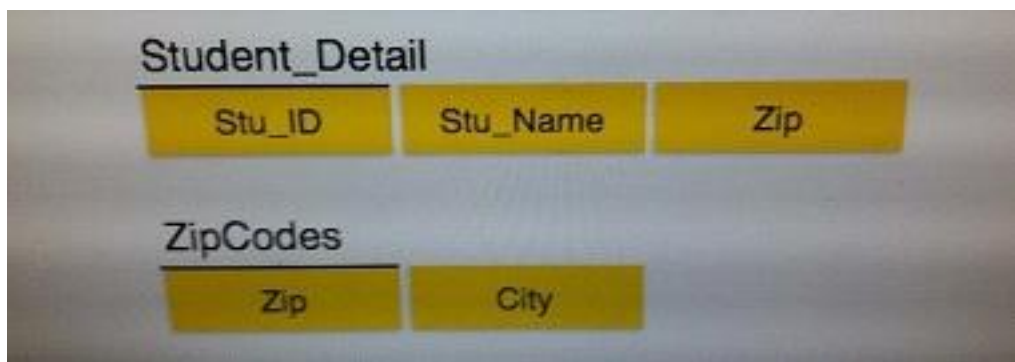
- 어떠한 비-기본 속성도 기본 키 속성에 전이적으로 의존하지 않아야 한다.
- 어떠한 non-trivial 함수 종속성이 $X \rightarrow Y$ 일 경우에, 다음의 둘 중의 하나여야 한다:
 - X 는 슈퍼키이거나,

□ A는 기본 속성이다.



우리가 알 수 있는 것은 위의 Student_detail 관계에서, Stu_ID는 키이면서 유일한 기본 키 속성이라는 것이다. 또한 우리가 알 수 있는 것은 City는 Stu_ID뿐만 아니라 Zip 그 자체에 의해서 식별 가능하다. Zip은 슈퍼키가 아니며, City 역시 기본 속성이 아니다. 따라서, $\text{Stu_ID} \rightarrow \text{Zip} \rightarrow \text{City}$ 가 성립되며, 이것이 transitive dependency 이다.

이러한 관계를 3 NF로 만들기 위해서, 우리는 다음과 같이 이 관계를 두 개의 관계로 나누었다:



4) Boyce-Codd Normal Form(BCNF)

BCNF는 엄격히 말해서 3 FN의 확장형이다. BCNF에서는 다음과 같이 주장한다:

- 어떠한 non-trivial 함수 종속성이 $X \rightarrow A$ 일 경우에, X는 슈퍼-키여야 한다.

위의 그림에서, Stu_ID는 관계 Student_Detail의 슈퍼-키이며, Zip는 관계 ZipCodes에서 슈퍼-키이다. 따라서,

$\text{Stu_ID} \rightarrow \text{Stu_Name}, \text{Zip}$

그리고

$\text{Zip} \rightarrow \text{City}$

이들 두 개의 관계는 이제 BCNF에 있다.

XV. DBMS - JOINS

우리는 두 관계들의 Cartesian product을 사용함으로써 얻는 장점 즉, 함께 쌍을 이루는 모든 잠재적 튜플들을 우리에게 제공한다는 것에 대해 잘 알고 있다. 그러나 무수히 많은 속성들을 갖고 있는 수천개의 튜플로 구성된 커다란 관계를 만날 때처럼, 어떤 경우에는 카르티안적을 사용하기가 쉽지 않을 수 있다.

Join은 selection 절차에 수반되는 카르티안 적의 결합이다. Join 기능은 만일 그리고 오직 만일에(if and only if) 주어진 조건을 만족시킨다면, 서로 다른 관계들로부터 두 개의 터플을 짝(pair)으로 만든다.

간단하게 여러 가지 종류의 Join에 대해 알아보자.

1. Theta θ Join

췌타 조인은 세타 조건을 만족시키는 서로 다른 관계로부터 튜플들을 결합시킨다. 이 조인의 조건은 부호 θ 로 표시한다.

표기법:

$$R1 \bowtie_{\theta} R2$$

$R1$ 과 $R2$ 는 속성 $A1, A2, \dots, A_n$ 그리고 $B1, B2, \dots, B_n$ 을 가지고 있는 관계들이며, 이 속성들이 공동으로 (in common) 어떠한 것도 갖지 않으므로, 이것은 $R1 \cap R2 = \Phi$ 이다.

췌타 조인은 모든 종류의 비교 연산자를 사용할 수 있다.

예제:

Student

SID	Name	Std
101	Alex	10
102	Maria	11

Subjects

Class	Subject
10	Math
10	English

11 Music
11 Sports

Student_Detail =
STUDENT ⋈_{Student.Std = Subject.Class} SUBJECT

Output:

Student_detail

SID	Name	Std	Class	Subject
101	Alex	10	10	Math
101	Alex	10	10	English
102	Maria	11	11	Music
102	Maria	11	11	Sports

2. Equijoin

세타 조인이 단지 동등(equality) 비교연산자만을 사용할 때, 동등 조인(equijoin)이라 부른다. 위의 예제는 동등 조인과 일치한다.

3. Natural join

자연 조인에서는 어떠한 비교 연산자도 사용하지 않는다. 이것은 카르티잔 적이 하는 방법과 연관(concatenate)이 없다. 우리는 두 관계 간에 적어도 한 개의 공동 속성만이 존재하는 경우에만 자연 조인을 수행할 수 있다. 따라서 이 속성들은 동일한 이름과 도메인을 가져야만 한다.

자연 조인은 양쪽의 관계에서 속성의 값이 동일해서 매칭되는 속성들을 대상으로 이루어진다.

예제:

Courses

CID	Course	Dept
CS01	Database	CS
ME01	Mechanics	ME
EE01	Electronics	EE

HoD

Dept	Head
CS	Alex
ME	Maya

EE Mira

Courses ⋈ HoD

Dept	CID	Course	Head
CS	CS01	Database	Alex
ME	ME01	Mechanics	Maya
EE	EE01	Electronics	Mira

4. Outer Joins

췌타조인, 동등조인, 자연조인은 inner join이라 부른다. 이너조인에는 단지 대칭되는 속성들을 가지고 있는 튜플들만을 포함하며 그 나머자들은 결과 관계에서 버려진다. 그러므로 결과 관계에 참여하는 관계들의 모든 튜플들을 포함시키기 위해서는 outer join을 사용하여야 한다.

이것에는 3 종류가 있다.

- left outer join
- right outer join
- full outer join

■ Left Outer Join($R \bowtie S$)

Left 관계인 R에 있는 모든 튜플들이 결과 관계에 포함된다. 만일 R의 튜플들이 Right 관계인 S에 있는 튜플들과 어떠한 매칭도 이루어지지 않는다면, 결과 관계의 S-속성들은 NULL로 처리된다.

예제:

Left

A	B
100	Database
101	Mechanics
102	Electronics

Right

A	B
100	Alex
102	Maya
104	Mira

Courses \bowtie HoD

A	B	C	D
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya

■ Right Outer Join($R \bowtie_{\text{right}} S$)

Right 관계 S에 있는 모든 튜플들이 결과 관계에 포함된다. S의 튜플들과 R의 튜플들이 서로 매칭되지 않는다면, 결과 관계의 R-속성들은 NULL로 처리된다.

예제:

Courses \bowtie_{right} HoD

A	B	C	D
100	Database	100	Alex
102	Electronics	102	Maya
---	---	104	Mira

■ Full Outer Join($R \bowtie_{\text{full}} S$)

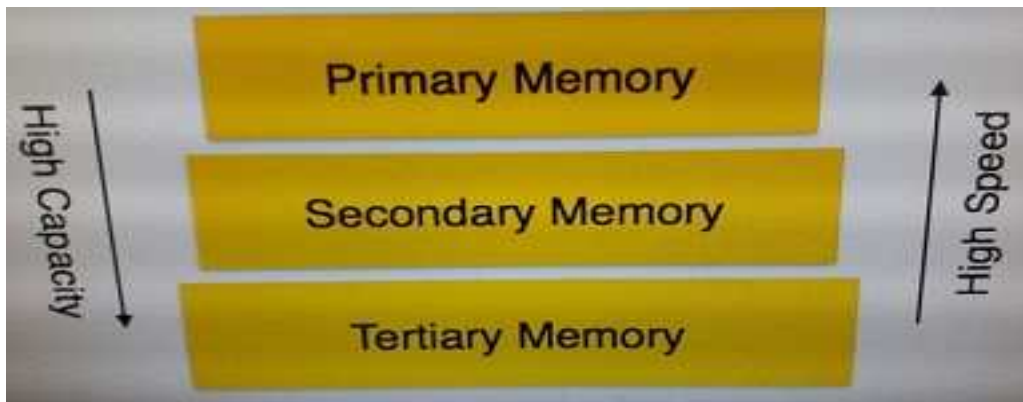
양쪽에서 참여하는 관계들의 모든 튜플들이 최종 관계에 포함된다. 양쪽 관계에서 매칭되는 튜플들이 없다면, 이것들의 각각의 비-매칭 속성들은 NULL로 표시된다.

Courses \bowtie_{full} HoD

A	B	C	D
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya
---	---	104	Mira

MVI. DBMS - STORAGE SYSTEM

데이터베이스는 레코드를 포함하고 있는 파일 포맷에 저장된다. 물리적 차원에서 말하면, 실제적인 데이터는 어떤 기기의 전자기적 포맷에 저장된다. 이러한 저장장치들은 크게 3가지 유형으로 나눌 수 있다:



■ Primary Storage

CPU에 직접 접근할 수 있는 메모리 저장기기(storage)가 이 범주에 속한다. CPU의 내부 메모리인 *register*, 빠른 메모리인 *cache*, 그리고 메인 메모리인 *RAM*은 CPU에 직접 접근할 수 있으며, 이것들은 모두 motherboard나 CPU chipset에 자리 잡고 있다. 이 저장기기는 전형적으로 매우 작으며, 엄청나게 빠를 뿐만 아니라 휘발성(volatile)도 가지고 있다. 1차 저장기기는 그것의 상태를 유지하기 위하여 지속적인 전력공급을 필요로 한다. 전원이 꺼지게 되면, 그것의 모든 데이터는 날라 간다.

■ Secondary Storage

2차 저장기기는 미래를 위한 데이터 저장용으로 또는 백업용으로 사용된다. 2차 저장기기에는 CPU chipset이나 마더보드의 일부가 아닌 즉, 마그네틱 디스크, 광학 디스크인 DVD, CD 등, 하드 디스크, flash drives, 그리고 마그네틱 테이프와 같은 메모리 기기들이 포함된다.

■ Tertiary Storage

3차 저장기기는 막대한 양의 데이터를 저장하기 위하여 사용된다. 이러한 저장기기는 컴퓨터 시스템의 외부에 있으므로, 이것들은 속도에서 가장 느리다. 이러한 저장기기들은 대체로 시스템의 전반적인 백업용으로 사용된다. 광학 디스크와 마그네틱 테이프 등이 3차 저장기기로 널리 사용되고 있다.

1. Memory Hierarchy

컴퓨터 시스템은 잘 정의된 계층 구조의 메모리를 가지고 있다. CPU는 메인 메모리뿐만 아니라 내장되어있는 레지스터에 직접 접근할 수 있다. 메인 메모리로의 접근시간은 분명히 CPU 속도보다는 느리다. 이러한 속도간의 불일치(mismatch)를 최소화하기 위하여, 캐쉬 메모리를 도입하였다. 캐쉬 메모리는 가장 빠른 접근 시간을 제공하며, 이것은 CPU에 의해 가장 자주 접근하는 데이터를 포함하고 있다.

가장 빠른 접근 속도의 메모리는 가장 값이 비싸다(costliest). 대용량 저장기기는 속도가 느리며, 이것들은 비싸지 않다. 그렇지만 이것들은 CPU 레지스터나 캐쉬 메모리에 비해 막대한

양의 데이터를 저장할 수 있다.

2. Magnetic Disk

하드 디스크 드라이브 현존하는 컴퓨터 시스템에서 가장 일반적인 2차 저장기기 이다. 이것들을 마그네틱 디스크라 부르는데, 왜냐하면 이것들은 정보를 저장하기 위하여 전자기의 개념을 사용하기 때문이다. 하드 디스크는 전자기 물질이 코팅된 금속 디스크이다. 이 디스크들은 한 개의 중심축(spindle)에 수직으로 쌓여있다. 읽기/쓰기 헤드가 디스크 사이를 움직여서 그것의 밑에 있는 반점(spot)에 자성을 입히거나 지우거나 한다. 자성을 띤 반점은 0이나 1로 인식된다.

하드 디스크는 효율적으로 데이터를 저장하기 위하여 잘 정의된 순서로 포맷된다. 하드 디스크는 *tracks*이라 부르는 많은 동심원을 가지고 있다. 모든 트랙은 *sectors*로 나뉘어지며 하드 디스크에 있는 한 섹터에는 전형적으로 512 바이트의 데이터를 저장할 수 있다.

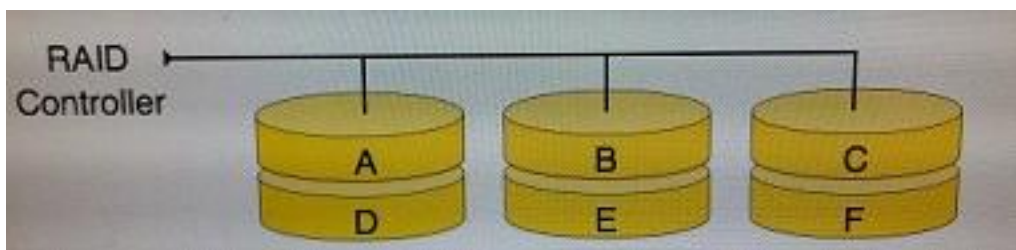
3. RAID

RAID란 Redundant Array of Independent Disks의 준말이며, 다수의 2차 저장기기를 결합시키는 기술이며, 이것들을 단일저장매체처럼 사용한다.

RAID는 여러 가지 목적을 달성하기 위하여, 다중의 디스크가 서로 결합된 배열 형태를 갖고 있다. RAID의 단계(level)에 따라 디스크의 배열이 결정된다.

□ RAID 0

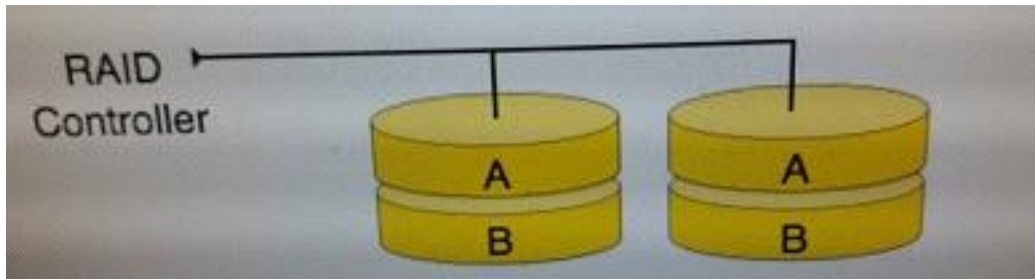
이 단계에서, 디스크의 줄무늬 배열(striped array)로 구성된다. 데이터는 blocks로 쪼개지며 블록들은 여러 디스크로 분배된다. 각 디스크는 병렬적으로 읽기/쓰기 위하여 데이터의 블록을 접수한다. 이것은 저장기기의 속도와 성능을 높인다. 레벨 0에서는 어떠한 parity와 backup도 존재하지 않는다.



□ RAID 1

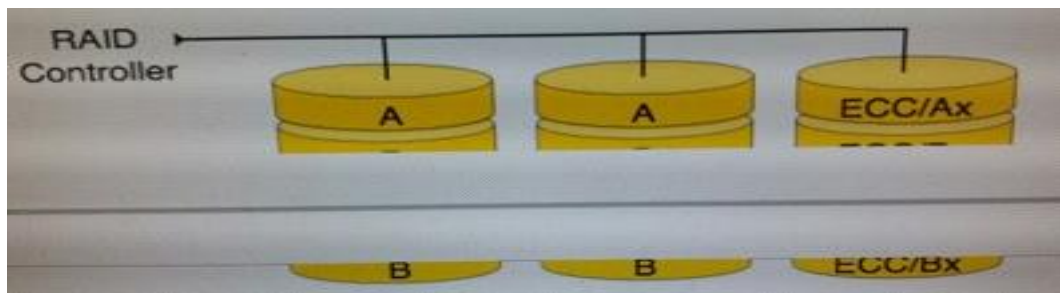
RAID 1에서는 mirroring 기술을 사용한다. 데이터가 RAID controller에 보내지면, 이것은

자신의 배열에 있는 모든 디스크에 데이터의 사본을 보낸다. RAID 레벨 1을 mirroring이라 부르며, 오류할 경우에 100%의 대리기능성(redundancy)을 제공한다.



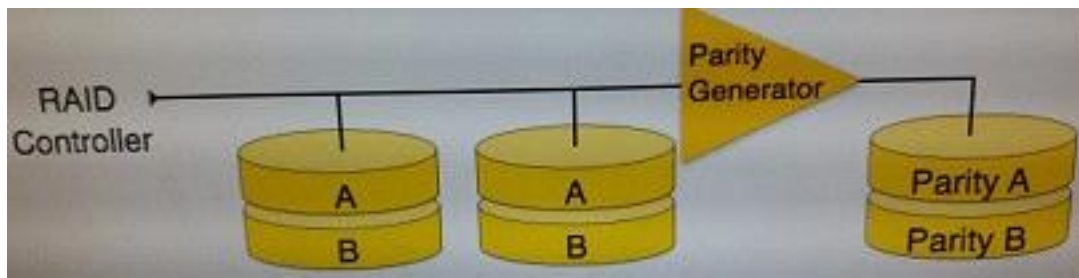
□ RAID 2

RAID 2에서는 여러 가지 디스크에 striped되어 있는 데이터와 관련해서 Hamming distance를 사용하여 Error Correction Code를 기록한다. 레벨 0처럼, 하나의 단어에 있는 각 데이터 비트는 별도의 디스크에 기록되며 그 데이터 단어들의 ECC codes는 다른 set disks에 저장된다. 이것의 복잡한 구조와 높은 비용으로 인해, RAID 2는 상업적으로 이용되지 않고 있다.



□ RAID 3

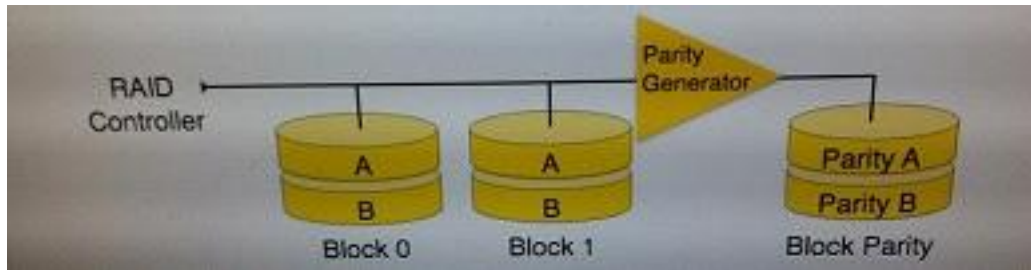
RAID 3에서는 다중의 디스크에 데이터를 stripes 한다. 데이터 단어로 생산된 parity bit가 다른 디스크에 저장된다. 이 기술은 단일 디스크의 오류를 극복하기 위하여 사용한다.



□ RAID 4

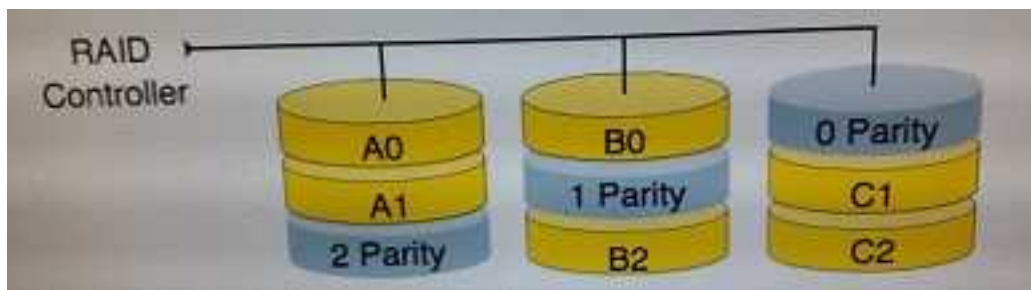
이 단계에서, 데이터의 전체 블록이 데이터의 디스크에 씌여진 다음에, 패리티가 생산되어 다른 디스크에 저장된다. 주목할 것은 레벨 3에서는 byte-level striping을 사용하지만 레벨 4

에서는 block-level striping을 사용한다는 것이다. 레벨 3과 레벨 4 모두 RAID를 실행하려면 적어도 3개의 디스크가 필요하다.



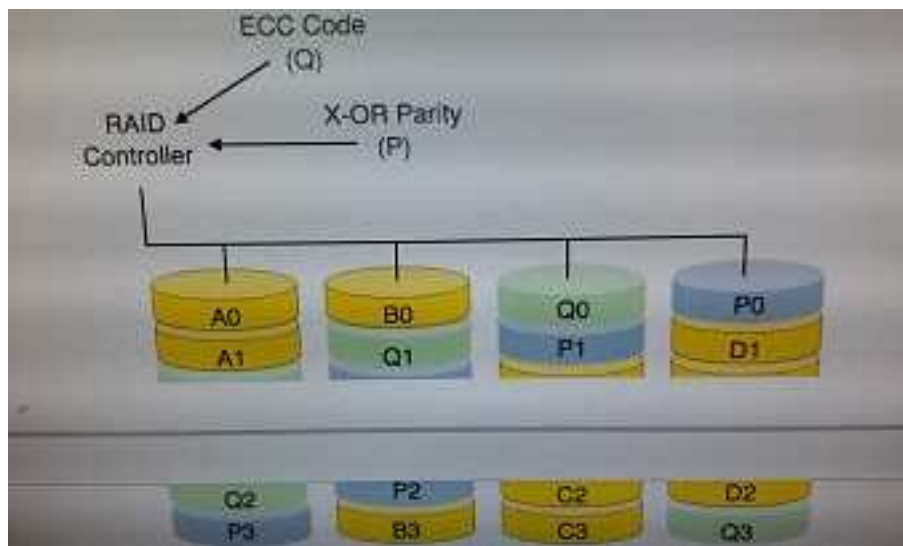
□ RAID 5

RAID 5에서는 다양한 디스크에 모든 데이터 블록들을 쓰지만, data block stripe용으로 생산된 parity bits는 또 다른 전용 디스크에 그것들을 저장하기 보다는 모든 데이터 디스크에 분산 시킨다.



□ RAID 6

이것은 RAID 5의 확장형이다. 이 단계에서, 두 개의 독립된 parities가 생산되어 다중의 디스크에 분산형식으로 저장된다. 두 개의 패리티들은 추가로 fault tolerance를 제공한다. 이 단계에서는 적어도 RAID를 실행하는데 4개의 디스크 드라이브가 필요하다.



XVII. DBMS - FILE STRUCTURE

서로 관련된 데이터와 정보는 파일 포맷에 집단적으로 저장된다. 파일이란 이진 포맷으로 저장된 순차적인(a sequence of) 레코드들이다. 디스크 드라이브는 레코드를 저장하기 위하여 여러 개의 블록으로 포맷된다. 파일 레코드들은 이러한 디스크 블록들에 복사된다(mapped).

1. File Organization

FO에서는 디스크에 파일 레코드를 매핑하는 방법을 정의한다. 파일 레코드를 조직하는 데는 4가지의 유형이 있다:



■ Heap File Organization

Heap File Organization을 이용하여 파일을 만들 때, Operation System에서는 추가적인 내역을 고려하지 않고 해당 파일용의 메모리 지분을 할당한다. 파일 레코드는 그러한 메모리 지분 어디엔가 자리를 잡는다. 이러한 레코드를 관리하는 것은 바로 소프트웨어의 책임이다. Heap File은 그 자체에 대해 어떠한 ordering, sequencing, 또는 indexing을 지원하지 않는다.

■ Sequential File Organization

모든 파일 레코드에는 그러한 레코드를 유일하게 식별할 수 있는 데이터 필드인 *attribute*가 포함되어 있다. 순차적 파일 조직에서, 레코드들은 unique key field나 search key를 근거로 하는 순차적 방식으로 파일에 자리 잡는다. 사실상, 물리적 형태로 모든 레코드들을 순차적으로 저장하는 것은 불가능하다.

■ Hash File Organization

Hash File Organization은 레코드의 특정 필드에 대하여 Hash 함수 계산법을 사용한다. 해쉬 함수의 결과를 가지고 레코드가 자리잡을 디스크 블록의 위치를 결정한다.

■ Clustered File Organization

클러스트 방식의 파일 조직은 대규모 데이터베이스에는 맞는 것으로 여겨지고 있다. 이 메카니즘에서, 한 개나 그 이상의 관계에 있는 관련된 레코드들은 동일한 디스크 블록에 있어야 한다. 다시 말해서, 레코드의 순서가 오픈키나 탐색키에 의존하지 않는다는 것이다.

2. File Operations

데이터베이스 파일을 작업하는 데는 크게 두 가지로 범주화할 수 있다:

■ Update Operations

■ Retrieval Operations

갱신 작업에서는 insertion, deletion, 또는 update를 사용하여 데이터 값을 변경한다. 한편으로, 검색 작업에서는 데이터를 변화시키지는 않지만, 선택 조건에 따라 그것들을 검색한다. 두 가지 작업 유형 모두에서, selection은 중요한 역할을 한다. 파일의 제작과 삭제 이외에도, 파일들에서 이루어질 수 있는 여러 가지 작업들이 있다.

□ **Open** - 파일은 두 가지 모드 즉, **read mode**와 **write mode** 중의 하나로 열 수 있다. 읽기 모드에서, 운영 시스템은 누구에게도 데이터의 변경을 허용하지 않는다. 다른 말로 해서, 데이터를 단지 읽을 수만 있다. 읽기 모드에서 열린 파일은 여러 엔티티들 간에 공유될 수 있다. 쓰기 모드에서는 데이터의 변경을 허용한다. 쓰기 모드에서 열린 파일은 읽을 수는 있지만 공유할 수는 없다.

□ **Locate** - 모든 파일들은 읽거나 쓰려고 하는 데이터가 존재하고 있는 현재의 위치를 알려

주는 파일 포인터(pointer)를 가지고 있다. 이 포인터는 추가로 조정이 가능하다. 찾기 *seek* 기능을 사용하면, 포인터를 앞으로 또는 뒤로 이동시킬 수 있다.

□ **Read** - 디폴트로서, 파일이 읽기 모드에서 열릴 때, 파일 포인터는 그 파일의 시작부분을 지적한다. 이용자가 파일을 열고자 할 때, 파일 포인터의 위치를 결정할 수 있는 여러 가지 옵션이 존재한다. 파일 포인터의 바로 다음 데이터부터 읽혀진다.

□ **Write** - 이용자는 쓰기 모드에서 내용을 편집하고자 하는 파일의 열기를 선택할 수 있다. 또한 deletion, insertion, 또는 modification을 할 수 있다. 파일 포인터는 열고자하는 시간에 설정될 수 있거나 운영 시스템에서 그렇게 할 수 있도록 허용한다면 역동적으로 변경될 수 있다.

□ **Close** - 이것은 운영시스템 입장에서 가장 중요한 작업이다. 파일 닫기가 요구될 때, 운영 시스템은 다음과 같은 것을 수행한다:

- ▲ 만일 공유 모드에서라면, 모든 잠금장치(locks)를 제거한다.
- ▲ 데이터가 변경되었다면, 2차 저장매체에 그 데이터를 저장한다.
- ▲ 파일과 결합되어 있는 모든 buffers와 file handlers를 해금(release)한다.

파일에 있는 데이터의 조직은 중요한 역할을 한다. 파일에 있는 원하는 레코드에 파일 포인터를 위치시키는 방법은 레코드를 순차적으로 또는 클러스트 방식으로 정렬하는지에 따라 다양하다.

XVIII. DBMS - INDEXING

데이터는 레코드의 형태로 저장된다. 모든 레코드는 하나의 key field를 가지고 있으며, 이것은 유일하게 관련 레코드를 식별하는데 사용된다.

색인화이란 색인으로 사용되는 어떤 속성을 근거로 데이터베이스 파일로부터 효율적으로 데이터를 검색하는 데이터 구조 기법을 말한다. 데이터베이스 시스템에서 색인화는 우리가 책에서 보는 것과 유사하다.

색인화는 색인하려는 속성을 근거로 정의할 수 있다. 색인화에는 다음과 같은 유형이 존재한다:

■ Prime Index

Prime index는 잘 정돈된 색인파일에서 나타난다. 이 데이터파일은 key field에 의해 정돈된다. key field는 일반적으로 관계의 primary key 이다.

■ Secondary Index

2차 색인은 후보 키인 필드로부터 생산될 수 있으며, 모든 레코드에서 또는 중복된 값을 가지고 있는 비-key에서 유일한 값을 갖는다.

■ Clustering Index

clustering 색인은 잘 정돈된(ordered) 데이터 파일에서 나타난다. 이 데이터 파일은 non-key field에 맞춰서 정돈된다.

잘 정돈된 색인에는 두 가지 유형이 있다:

- Dense Index
- Sparse Index

1. Dense Index

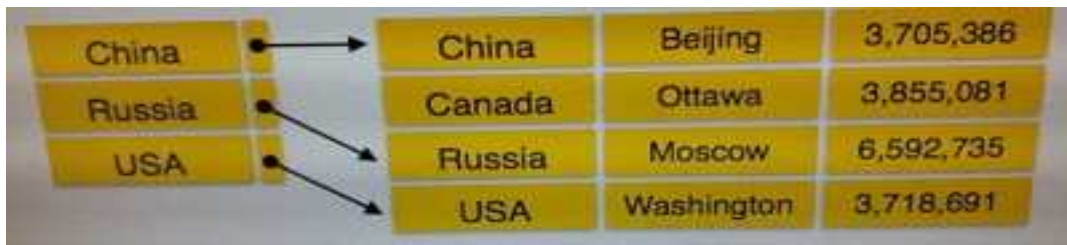
조밀색인에서는 데이터베이스에 있는 모든 탐색 키 값을 위한 색인 레코드가 존재 한다. 이것은 탐색을 보다 빠르게 만들지만 색인 레코드 그 자체를 보관하는데 더 많은 공간을 필요로 한다. 색인 레코드에는 탐색 키 값과 디스크에서 실재적인 데이터를 지정하는 포인터가 포함된다.

The diagram illustrates a dense index structure. On the left, a column of yellow boxes contains the country names: China, Canada, Russia, and USA. To the right of each country name is a small black dot, and a horizontal arrow points from each dot to a corresponding row in a table on the right. The table has four columns: the country name, the capital city, and the population. The rows are: China (Beijing, 3,705,386), Canada (Ottawa, 3,855,081), Russia (Moscow, 6,592,735), and USA (Washington, 3,718,691).

China	China	Beijing	3,705,386
Canada	Canada	Ottawa	3,855,081
Russia	Russia	Moscow	6,592,735
USA	USA	Washington	3,718,691

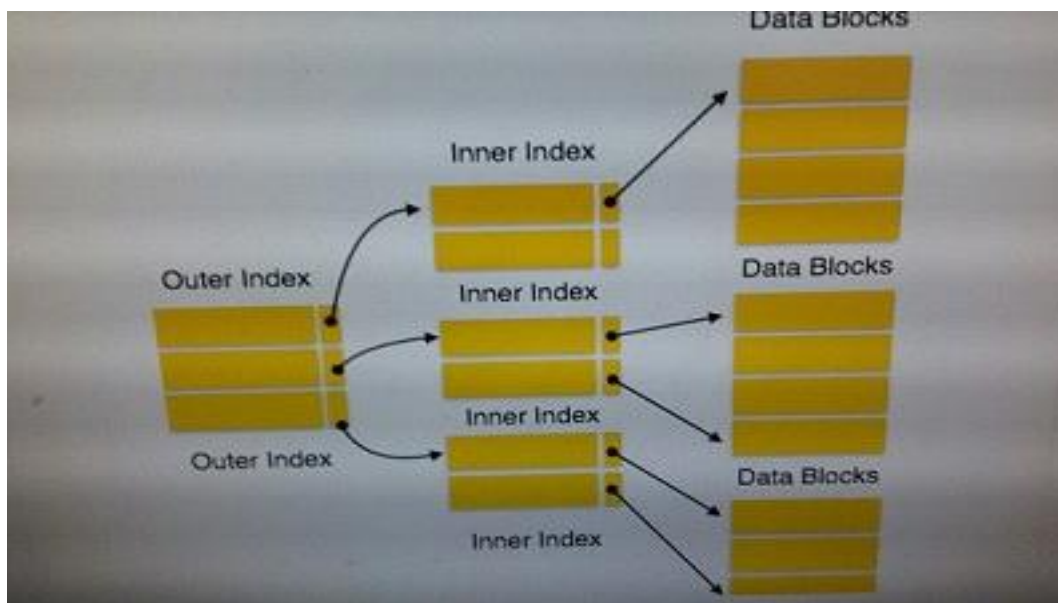
2. Sparse Index

희소색인에서, 색인 레코드는 모든 탐색 키용으로 제작되지 않는다. 여기에 있는 색인 레코드에는 탐색 키와 디스크에 있는 데이터의 실재적 포인터가 포함된다. 우리는 레코드를 탐색하기 위하여, 먼저 색인 레코드를 처리한(proceed) 다음에 데이터의 실재적 위치에 도달한다.



3. Multilevel Index

색인 레코드는 탐색 키 값과 데이터 포인터로 구성되어 있다. 다단계 색인은 실제적인 데이터베이스 파일들에 맞춰서 디스크에 저장된다. 데이터베이스 규모가 늘어남으로써, 색인의 사이즈도 늘어난다. 탐색 작업의 속도를 높이기 위하여 메인 메모리에 색인 레코드를 보관해야 한다는 커다란 필요성이 존재한다. 만일 단일 단계의 색인을 이용한다면, 커다란 색인을 다중의 디스크 접근이 가능한 메모리에 보관할 필요가 없다.



다단계 색인은 가장 바깥쪽 단계(the outermost level)를 단일 디스크 블록을 저장할 수 있을 정도로 작은 것으로 만들기 위하여 색인을 여러 개의 보다 작은 색인으로 쪼개서, 논리적 도움을 준다. 메인 메모리의 어떠한 곳에서도 쉽게 적응할 수 있도록(accommodated) 도움을 준다.

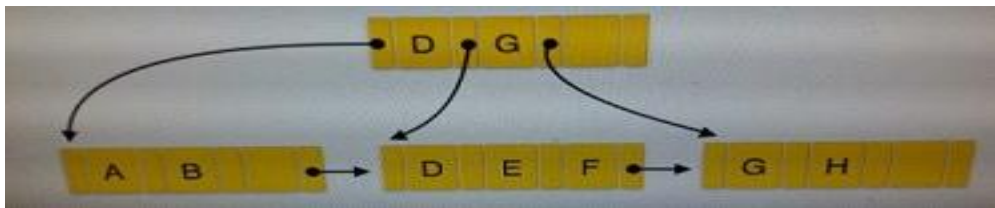
4. B⁺ Tree

B⁺ Tree는 균형 잡힌 이진 탐색 트리(balanced binary search tree)이다. 이것은 다단계 색

인 포맷으로 되어 있다. B⁺ Tree의 잎 노드(leaf node)는 실제적인 데이터 포인터를 나타낸다. B⁺ Tree에서 보장하는 것은 모든 잎 노드들이 동일 높이에 있어서 균형이 잡혀있다는 것이다. 추가로 잎 노드들은 link list를 사용하여 링크되어 있다: 그러므로 B⁺ Tree는 random access 뿐만 아니라 sequential access도 지원할 수 있다.

1) Structure of B⁺ Tree

모든 잎 노드는 루트 노드로부터 동일한 거리에 있다. B⁺ Tree는 n 이 모든 B⁺ Tree용으로 고정되어 있다면 order n 에 속한다.



■ Internal nodes

- 내부 non-leaf 노드들에는 루트 노드를 제외하더라도 적어도 $\lceil n/2 \rceil$ 포인터가 포함된다.
- 기껏해야, 한 개의 내부 노드에는 n 포인터들이 포함될 수 있다.

■ Leaf nodes

- 잎 노드에는 적어도 $\lceil n/2 \rceil$ 레코드 포인터와 $\lceil n/2 \rceil$ 키 값이 포함된다.
- 기껏해야, 한 개의 잎 노드에는 n 레코드 포인터들과 n 키 값들이 포함된다.
- 모든 잎 노드에는 다음의 잎을 지정하기 위한 한 개의 블록 포인터 p 가 포함되어, 링크된 리스트를 만든다.

2) B⁺ Tree Insertion

- B⁺ Tree는 밑바닥에서부터 채워지며 각 엔트리는 잎 노드에서 끝난다(done).
- 만일 잎 노드가 넘쳐흐르면(overflow):
 - 노드를 두 개의 파트로 쪼갬다.
 - Partition at $i = \lfloor m+1/2 \rfloor$.
 - 첫 번째 i 엔트리들은 한 개의 노드에 저장된다.
 - 나머지 엔트리들 $i+1$ onward는 새로운 노드로 이동한다.
 - i^{th} 키는 잎 노드의 부모 노드에서 중복된다.
- 만일 비-잎 노드가 넘쳐흐르면:
 - 노드를 두 개의 파트로 쪼갬다.
 - Partition at $i = \lfloor m+1/2 \rfloor$.
 - i 까지의 엔트리들은 한 개의 노드에 저장된다.
 - 나머지 엔트리들은 새로운 노드로 이동한다.

3) B⁺ Tree Deletion

- B⁺ Tree 엔트리는 잎 노드에서 삭제된다.

- 타겟 엔트리를 찾아서 삭제한다.
 - 만일 그것이 내적 노드라면, 왼쪽에서부터 엔트리를 삭제하여 대체한다
- 삭제한 다음에, underflow를 검증한다.
 - 만일 underflow가 발생한다면, 그것에 남아있는 노드들로부터 엔트리들을 분산시킨다.
- 만일 분산이 왼쪽부터 불가능하다면,
 - 그것에 오른쪽에 있는 노드부터 분산시킨다.
- 만일 분산이 왼쪽이나 오른쪽으로부터 불가능하다면,
 - 그것의 왼쪽과 오른쪽에 있는 노드를 통합한다.

XIX. DBMS - HASHING

거대한 데이터베이스 구조로 인하여, 그것의 모든 단계를 통해 모든 색인 값을 탐색한 다음에 원하는 데이터를 검색하기 위하여 목적지 데이터 블록에 도달하는 것은 거의 불가능에 가깝다. hashing은 디스크에서 색인구조를 사용하지 않고 데이터 레코드의 직접적인 위치를 계산하는 효과적인 기법이다.

해싱에서는 데이터 레코드의 어드레스를 생산하기 위한 파라미터로서 탐색키와 함께 해쉬 함수를 사용한다.

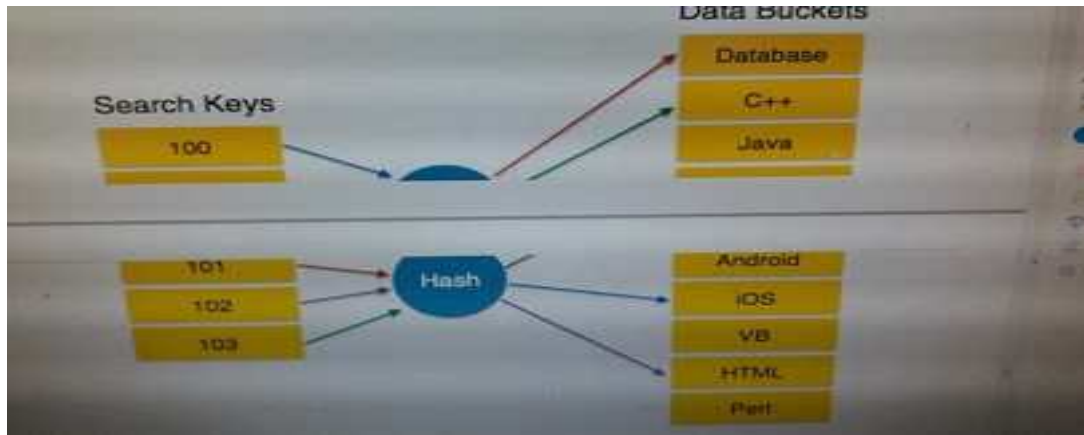
1. Hash Organization

■ **Bucket** - 해쉬 파일은 버킷 포맷으로 데이터를 저장한다. 버킷이란 저장의 단위이다. 전형적으로 하나의 버킷은 한 개의 완전한 디스크 블록을 저장하며, 차례차례(in turn) 한 개 이상의 레코드를 저장할 수 있다.

■ **Hash Function** - 해쉬 함수 h 는 모든 탐색키의 세트를 실제적인 데이터가 자리잡고 있는 어드레스에 매핑시키는 매핑 함수 이다. 이것은 탐색 키에서부터 버킷 어드레스까지를 다루는 함수이다.

2. Static Hashing

정적 해싱에서, 탐색-키 값이 제공될 때 그 해쉬 함수는 항상 동일한 어드레스를 계산한다. 예를 들어, 만일 mod-4 해싱 함수가 사용되었다면, 이것은 단지 5개의 값만을 생산할 것이다. 결과 어드레스도 항상 그 함수에 맞춰 동일할 것이다. 제공된 버킷의 번호는 항상 변하지 않고 남아 있다.



■ Operation

□ Insertion - 레코드를 정적 해쉬를 사용하여 입력하고자 할 때, 해쉬 함수 h 는 탐색 키 k 용으로 그 레코드가 저장될 버킷 어드레스를 계산한다.

$$\text{Bucket address} = hK$$

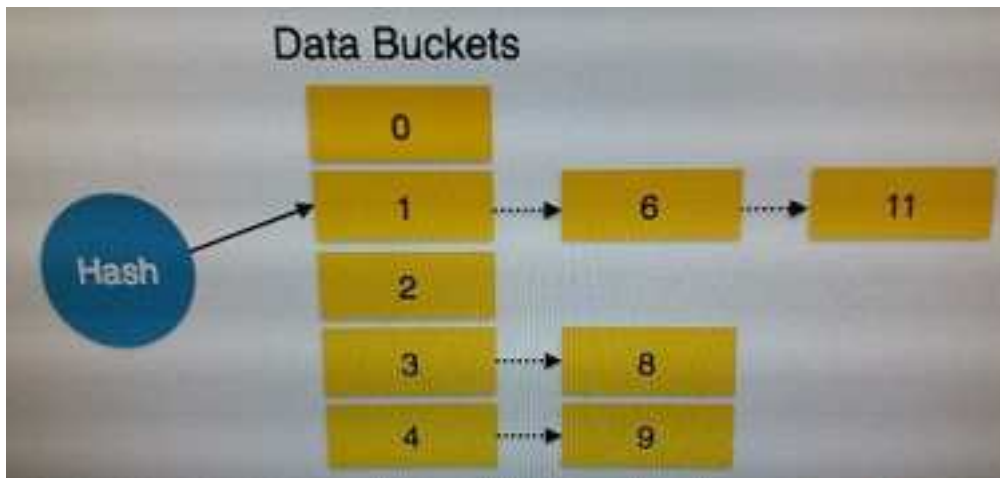
□ Search - 레코드를 검색하려할 때, 동일한 해쉬 함수를 사용하여 데이터가 저장된 버킷의 어드레스를 검색한다.

□ Delete - 이것은 삭제 작업에 의해 수반되는 간단한 탐색이다.

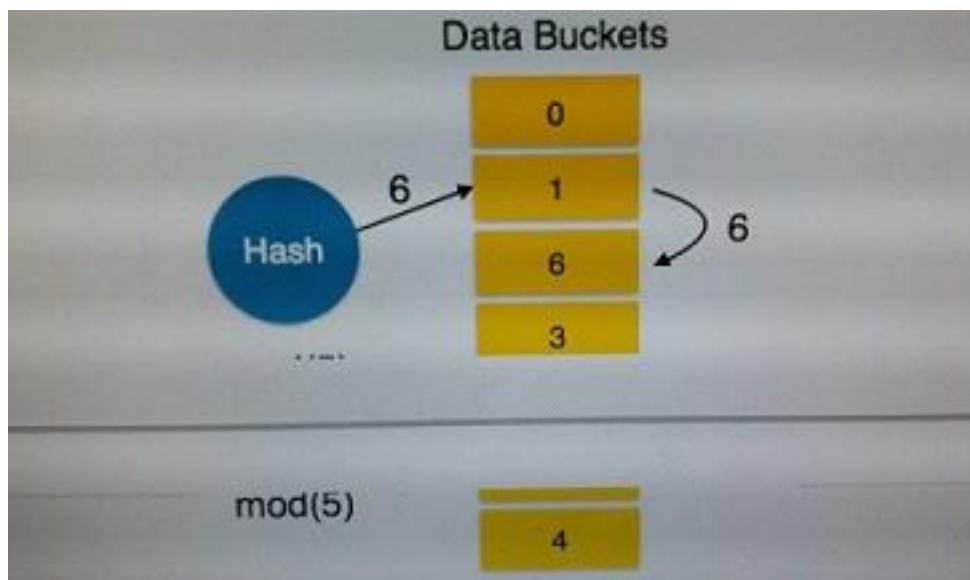
■ Bucket Overflow

bucket-overflow 조건을 collision이라 부른다. 이것은 어떠한 정적 해쉬 함수에서도 치명적인 상태이다. 이러한 경우에, overflow chaining이 사용될 수 있다.

□ Overflow chaining - 버킷이 가득 차면, 새로운 버킷이 동일한 해쉬 결과에 할당되어 이전의 것에 이어서 링크된다. 이러한 메카니즘을 Closed Hashing이라 부른다.



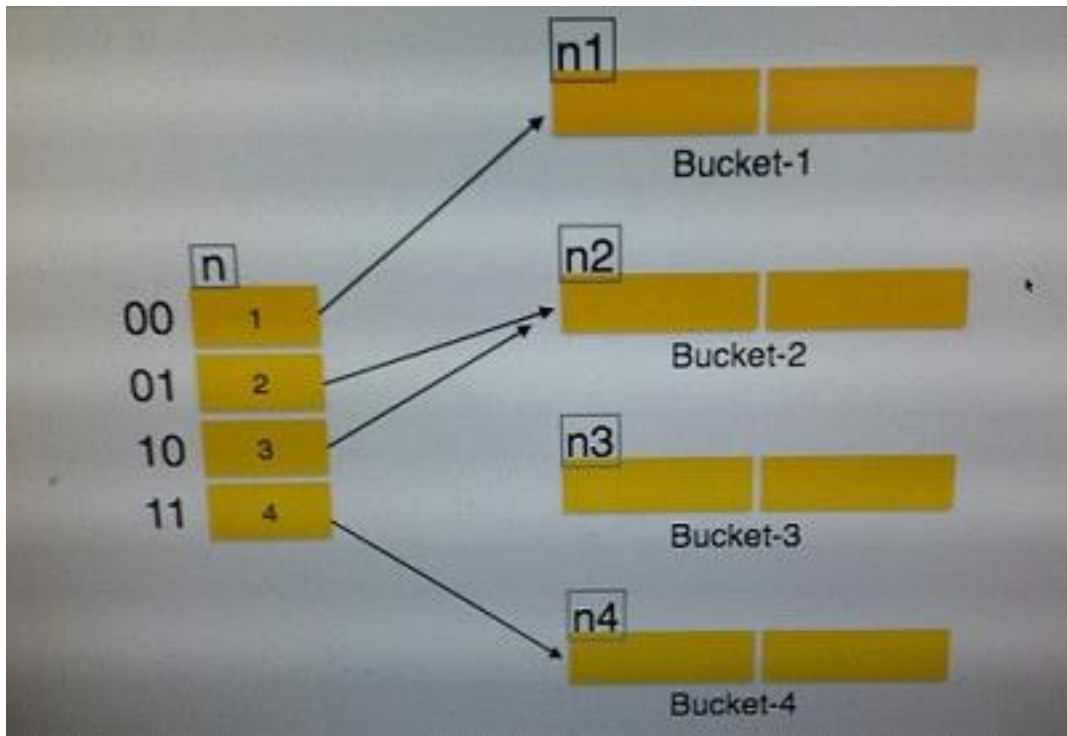
□ Linear Probing - 해쉬 함수가 데이터가 이미 저장된 어드레스를 생산할 때, 그 다음의 자유로운 버킷이 그것에 할당된다. 이러한 메카니즘을 Open Hashing이라 부른다.



3. Dynamic Hashing

정적 해싱의 문제는 데이터베이스가 성장하거나 줄어들 때 역동적으로 확장하거나 축소할 수 없다는 것이다. 역동적 해싱에서는 데이터 버킷을 역동적으로 또는 on-demand에 따라 추가하거나 제거할 수 있는 메카니즘을 제공한다. 역동적 해싱을 또한 extended hashing이라고도 한다.

역동적 해싱에서, 해쉬 함수는 많은 수의 값을 생산하는데 사용되지만, 단지 소수만을 처음엔 사용할 수 있다.



■ Organization

모든 해쉬 값의 접두어(prefix)는 해쉬 색인으로 그 역할을 한다. 해쉬 값의 portion(한 조각)만이 컴퓨터 버킷 어드레스용으로 사용된다. 각각의 모든 해쉬 색인은 많은 비트가 해쉬 함수를 계산하는데 사용되는 방법을 보여주기 위하여 depth value를 갖는다. 이러한 비트들은 $2n$ 버킷들을 어드레스 할 수 있다. 모든 이러한 비트들이 소비됐을 때, 즉, 모든 버킷들이 가득 찾을 때, depth value가 선형적으로 증가하며 2배(twice)의 버킷들이 할당된다.

■ Operation

- Querying - 해쉬 색인의 depth value를 찾아서 이러한 비트들을 사용하여 버킷 어드레스를 계산한다.
- Update - 위에서처럼 쿼리를 수행한 다음에 데이터를 갱신한다.
- Deletion - 쿼리를 사용하여 찾고자하는 데이터에 접근한 다음에 그 데이터를 삭제한다.
- Insertion - 버킷의 어드레스를 계산한다.

- ▲ 만일 버킷이 이미 가득 차다면,
 - △ 더 많은 버킷을 추가한다.
 - △ 해쉬 값에 additional bits를 추가한다.
 - △ 해쉬 함수를 재-계산한다.

- ▲ 그렇지 않다면,
 - △ 버킷에 데이터를 추가한다.

▲ 모든 버키트가 가득 차다면, 정적 해싱의 remedies(치료)를 수행한다.

해싱은 데이터가 어떤 순서로 조직되고 쿼리가 데이터의 범위를 요구할 때 선호되지 않는다. 데이터가 이산적이고 무작위적일 때, 해싱은 가장 잘 수행된다.

해싱 알고리즘은 색인보다 매우 복잡하다. 모든 해쉬 기능(operations)은 항구적인 것이다(in constant time).

XX. DBMS - TRANSACTION

트랜잭션이란 작업의 그룹(a group of tasks) 이다. 따라서 단일 작업은 더 이상 쪼갤 수 없는 최소한의 처리 단위다.

간단한 트랜잭션의 예를 살펴보자. 은행원이 A의 계좌에서 B의 계좌로 500 달러를 이전했다고 가정해 보자. 이러한 매우 간단하고 조그만 트랜잭션에는 여러 가지 low-level 작업이 포함된다.

■ A의 계좌

```
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)
```

■ B의 계좌

```
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)
```

1. ACID Properties

트랜잭션은 매우 작은 단위의 프로그램이며 이것에는 여러 가지의 낮은 수준의 작업들이 포함될 수 있다. 데이터베이스 시스템에서 트랜잭션은 Atomicity, Consistency, Isolation,

Durability, 즉 ACID 성질이라는 것을 유지해야만 한다. 왜냐하면 정확성, 완전성, 그리고 데이터 순수성을 보장해야하기 때문이다.

■ Atomicity

이것은 트랜잭션이 원자 단위로 취급되어야 한다는 성질이다. 다시 말해서, 이것의 기능 모두가 수행되거나 아무 것도 수행되지 않는다는 것이다. 트랜잭션이 불완전하게 남아 있는 데이터베이스에는 어떠한 상태도 존재하지 않아야 한다. 이러한 상태들은 트랜잭션이 수행되기 전이나 트랜잭션의 execution/abortion/failure 이후에 정의되어야 한다.

■ Consistency

데이터베이스는 어떠한 트랜잭션이 수행되더라도 그 다음에는 일관된 상태를 유지하여야 한다. 어떠한 트랜잭션도 데이터베이스에 있는 데이터에 역효과를 초래해서는 안 된다. 만일 데이터베이스가 트랜잭션 수행 이전에 일관된 상태였다면, 트랜잭션이 수행된 이후에도 그 상태 역시 일관적이어야 한다.

■ Durability

데이터베이스는 비록 시스템이 실패하거나 재-시작되더라도 모든 최신의 갱신 데이터를 충분히 유지할 수 있도록 내구적(durable)이어야 한다. 만일 트랜잭션이 데이터베이스의 데이터 일부를 갱신한다면, 데이터베이스는 변경된 데이터를 유지해야 한다. 만일 트랜잭션이 수행되었으나 데이터가 디스크에 기록되기 전에 그 시스템이 오류한다면, 시스템이 다시 기동될 때, 그 데이터는 갱신되어야 할 것이다.

■ Isolation

한 개 이상의 트랜잭션이 동시에 또는 병렬적으로 수행되고 있는 데이터베이스 시스템에서, 독립이란 시스템에서 그것이 유일한 트랜잭션인 것처럼 모든 트랜잭션이 전달되어 처리되어야 한다는 성질이다. 어떠한 트랜잭션도 기타 다른 존재의 트랜잭션에 영향을 끼쳐서는 안 된다.

2. Serializability

다중의 트랜잭션들이 멀티-프로그래밍 환경에서 운영시스템에 의해 수행될 때, 하나의 트랜잭션의 명령(instruction)이 어떤 다른 트랜잭션과 인터리브(interleave)할 가능성이 있다.

□ Schedule

트랜잭션의 연대순(chronological) 처리 순서를 스케줄이라 부른다. 스케줄은 그 속에 많은 트랜잭션을 가질 수 있다. 이들 트랜잭션 각각은 많은 instruction/tasks로 구성되어 있다.

□ Serial Schedule

스케줄은 하나의 트랜잭션이 먼저 수행되는 방식으로 정돈되어 있다. 첫 번째 트랜잭션이 자신의 순환주기를 완성할 때, 다음 트랜잭션이 수행된다. 트랜잭션들이 앞뒤로 정돈되어 있다. 마치 트랜잭션이 직렬방식으로 수행되기 때문에, 이런 종류의 스케줄을 직렬 스케줄이라 부른다.

다.

다중의 트랜잭션 환경에서, 직렬 스케줄은 benchmark로 여겨지고 있다. 트랜잭션에서 지시(instruction)의 수행 순서는 변할 수 없지만, 두 개의 트랜잭션은 무작위 방식으로 수행되는 자신들만의 지시들을 가지고 있다. 만일 두 개의 트랜잭션이 서로 독립적이고 데이터의 서로 다른 부분에서 작업을 한다면, 이러한 수행이 결코 해가 되지는 않는다; 그러나 경우에 따라서 두 개의 트랜잭션이 동일한 데이터에서 작업을 하면 그 결과는 변할 것이다. 이처럼 지속적으로 변하는(ever-varying) 결과는 데이터베이스에 비-일관성을 가져올 수도 있다.

이런 문제를 해결하기 위하여, 만일 그것의 트랜잭션들이 직렬적이거나 그것들 간에 어떤 동등한 관계를 가지고 있다면, 트랜잭션 스케줄의 병렬적 수행이 이루어질 수 있다.

■ Equivalence Schedules

동치 스케줄은 다음과 같은 유형들 중의 하나이다:

1) Result Equivalence

만일 두 개의 스케줄이 수행이 이루어진 다음에 동일한 결과를 생산한다면, 이것들을 동치적 결과(result equivalent)라 부른다. 이것들은 어떤 값에 대해 동일한 결과를 그리고 다른 값에는 다른 결과를 제공할 것이다. 이러한 이유 때문에 이러한 동치는 일반적으로 중요하게 여겨지지 않고 있다.

2) View Equivalence

만일 양쪽 스케줄의 트랜잭션들이 유사한 방식으로 동일한 행동을 취한다면, 두 개의 스케줄은 동치 보기일 수 있다.

예를 들어,

- 만일 T가 S1에 있는 시작(initial) 데이터를 읽는다면, 그것은 또한 S2에 있는 시작 데이터를 읽는다.
- 만일 T가 S1에서 J에 의해 작성된 값을 읽는다면, 그것은 또한 S2에서 J에 의해 작성된 값도 읽는다.
- 만일 T가 S1에 있는 데이터 값에 대한 최종 지시(write)를 수행한다면, 그것은 또한 S2에 있는 데이터 값에 대한 최종 지시를 수행한다.

3) Conflict Equivalence

만일 그것들이 다음과 같은 성질을 갖고 있다면, 두 개의 스케줄은 충돌할 것이다.

- 양쪽 모두 분리 트랜잭션(separate transactions)에 속한다.
- 양쪽 모두 똑같은 데이터 아이템에 접근한다.
- 적어도 이것들중의 하나는 “write” operation이어야 한다.

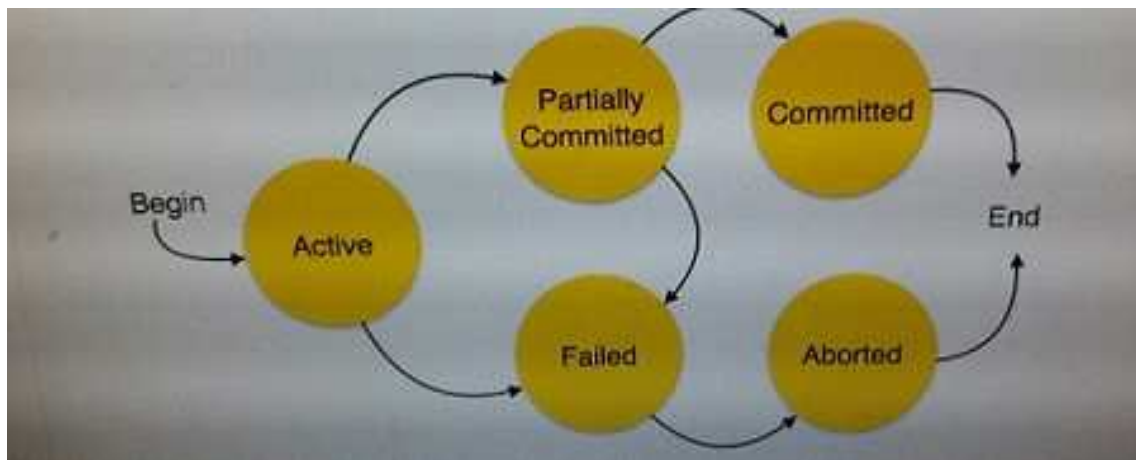
충돌 기능과 함께 다중의 트랜잭션을 가지고 있는 두 개의 스케줄이 다음과 같은 조건을 갖고 있을 때만 충돌 동치라 부른다:

- 양쪽 스케줄 모두 똑같은 세트의 트랜잭션을 포함하고 있다.
- operation에서 충돌하는 짝들(pairs)의 순서는 양쪽 스케줄에서 관리된다.

NOTE - 동치보기 스케줄은 연속적(serializable) 보기이며 충돌 동치 스케줄은 연속적 충돌이다. 모든 연속적 충돌 스케줄 역시 연속적 보기이다.

■ States of Transactions

데이터베이스에서 트랜잭션은 다음과 같은 상태들 중의 하나일 것이다:



- Active - 이 상태에서, 트랜잭션은 수행될 것이다. 이것은 모든 트랜잭션의 시작(initial) 상태이다.
- Partially Committed - 트랜잭션이 자신의 최종 운영을 수행할 때, 부분적으로 완수된 상태(partially committed state)라 부른다.
- Failed - 만일 데이터베이스 복구 시스템에 의해 진행되는 체크들 중의 어떤 것이 오류한다면, 이 트랜잭션은 오류 상태에 있다.
- Aborted - 체크들의 어떤 것이 오류하고, 그 트랜잭션이 오류 상태에 도달한다면, recovery manager는 트랜잭션이 수행되기 이전의 데이터베이스의 원래 상태를 불러 오기 위하여 그것에 관한 모든 write 운영을 소환한다. 이런 상태에 있는 트랜잭션을 폐기되었다고 말한다. 데이터베이스 복구 모듈은 트랜잭션이 폐기된 이후에, 다음과 같은 두 가지의 운영 중 하나를 선택할 수 있다:

- ▲ 해당 트랜잭션을 다시 시작한다.
- ▲ 해당 트랜잭션을 없앤다.

- Committed - 트랜잭션이 자신의 모든 operations를 성공적으로 수행한다면, committed 라고 부른다. 이것의 모든 효과는 이제 영원히 데이터베이스 시스템에서 이루어진다.

XXI. DBMS - CONCURRENCY CONTROL

다중의 트랜잭션이 동시에 진행되는 다중의 프로그래밍 환경에서 , 트랜잭션들의 병행성 (concurrency: 동시성)을 통제하는 것은 매우 중요하다. 우리는 병행적 트랜잭션의 atomicity, isolation, serializability를 보장하기 위하여 병행성 제어 프로토콜을 가지고 있다. 병행성 제어 프로토콜은 크게 다음과 같이 두 가지로 범주화할 수 있다:

- Lock based protocols
- Time stamp based protocols

1. Lock-based Protocols

록 기반 규약에 따르는 데이터베이스 시스템은 어떠한 트랜잭션도 그것에 대한 올바른 록을 입수(acquire)할 때까지 데이터를 읽고 쓸 수 없는 메카니즘을 사용한다. 두 가지 종류의 록이 존재한다:

- **Binary Locks** - 데이터 아이템에 대한 록은 두 가지의 상태에 있다. 잠겨 있거나 열려 있거나(locked or unlocked).
- **Shared/exclusive** - 이런 유형의 록 메카니즘은 자신들이 사용에 근거가 되는 록과는 다르다. 만일 록이 write 기능을 수행하려는 데이터 아이템에 걸린다면, 이것은 배타적 록(exclusive lock)이다. 한 개 이상의 트랜잭션으로 하여금 동일한 데이터 아이템을 쓰도록 허락하는 것은 데이터베이스를 비-일관성 상태를 초래할 것이다. 어떤 데이터 값도 변하지 않는다면, Read 록은 공유될 수 있다.

이용 가능한 록 규약에는 4가지의 유형이 있다:

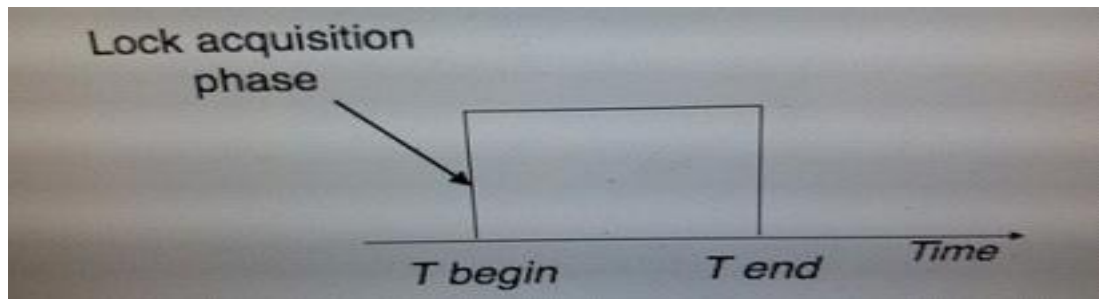
1) Simplistic Lock Protocol

단순 록 기반 규약은 '쓰기' 기능의 수행되기 전에, 트랜잭션으로 하여금 모든 객체에 대한 록을 걸 수 있도록 한다. 트랜잭션은 '쓰기' 기능이 끝나면, 데이터 아이템을 unlock할 수 있다.

2) Pre-claiming Lock Protocol

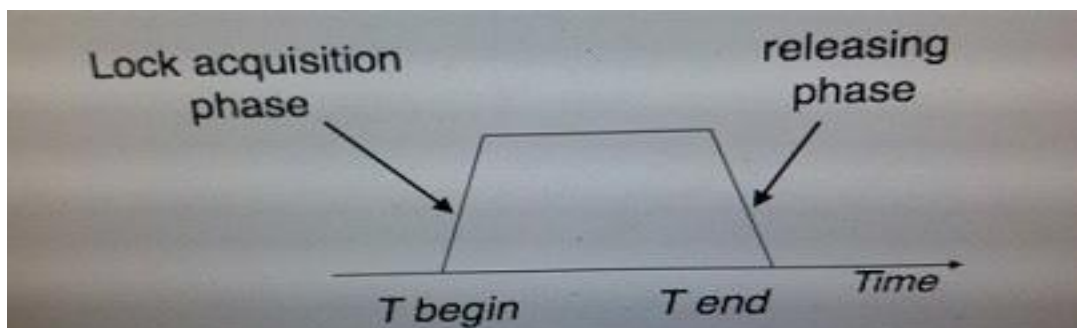
Pre-claiming Lock Protocol(선행조건 록 규약)은 자신들의 운영을 평가한 다음에 록이 필요한 데이터 아이템의 리스트를 만든다. 실행이 시작하기 전에, 트랜잭션은 사전에 필요한 모든 록을 시스템에 요청한다. 모든 록이 접수된다면, 트랜잭션은 수행되며 모든 운영이 종료되

면 그 록들은 해제 된다. 만일 모든 록이 제공되지 않는다면, 트랜잭션은 다시 처음으로 되돌아가서 모든 록이 제공될 때까지 기다린다.



3) Two-Phase Locking, 2PL

이 록 규약은 트랜잭션의 수행 단계를 3 부분으로 나눈다. 첫 번째 부분에서 트랜잭션의 수행이 시작될 때, 록에 필요한 허가권(permission)을 찾는다. 두 번째 부분은 트랜잭션이 모든 록들을 입수하는 곳이다. 트랜잭션이 첫 번째 록에서 해제되자마자 세 번째 단계가 시작된다. 이 단계에서, 트랜잭션은 어떠한 새로운 록도 요구할 수 없다: 이것은 단지 입수된 록들을 해제하기만 한다.

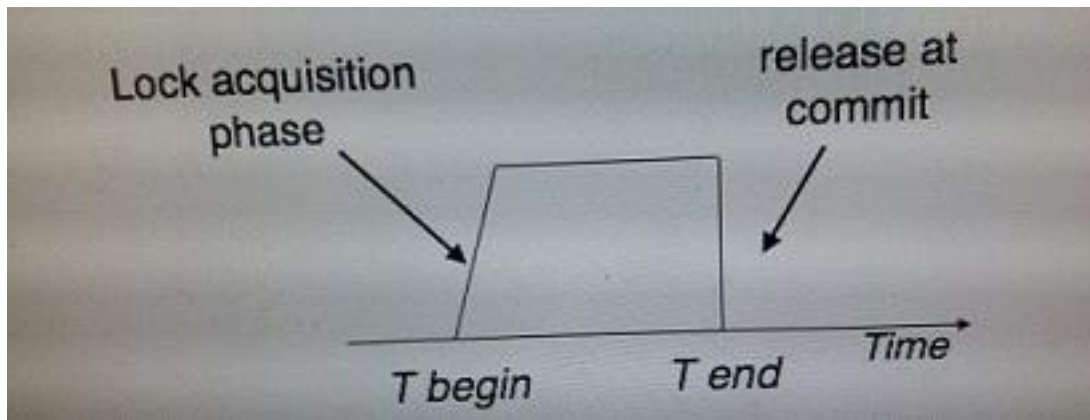


2PL은 두 개의 단계를 가지고 있다. 하나는 **growing**으로 모든 록들이 트랜잭션에 의해 입수되는 부분이고, 두 번째는 **shrinking**으로, 트랜잭션에 의해 유지되는 록들이 해제되는 부분이다.

배타적인 write 록을 주장하기 위하여, 트랜잭션은 먼저 공유되는 read 록을 입수한 다음에, 그것을 배타적인 록에서 갱신하여야 한다.

4) Strict Two-Phase Locking

엄한-2PL의 첫 번째 단계는 2PL과 같다. 첫 번째 단계에서 모든 록이 입수된 다음에, 트랜잭션의 정상적인 수행이 계속된다. 그러나 2PL과 다른 것은 엄격-2PL에서는 사용이 끝난 이후에도 록을 해제하지 않는다. 엄한-2PL에서는 commit point까지 모든 록을 유지한 다음, 한꺼번에 모든 록을 해제한다.



엄한-2PL은 2PL처럼 cascading abort를 갖진 않는다.

2. Timestamp-based Protocols

가장 일반적으로 사용되는 병행성 규약은 타임스탬프 기반 규약 이다. 이 규약은 타임스탬프처럼 시스템 타임이나 논리적 카운터(counter)를 사용한다.

록-기반 규약들은 수행이 시작될 때 트랜잭션들 중에서 충돌 짝들 간의 순서를 관리한다. 반면에 타임스탬프 기반 규약들은 트랜잭션이 발생하자마자 작업을 시작한다.

모든 트랜잭션은 자신과 결합된 타임스탬프를 가지고 있으며, 순서는 트랜잭션의 나이(age)에 의해 결정된다. 0002 clock time에 만들어진 트랜잭션은 그것 다음에 만들어진 모든 다른 트랜잭션보다 나이가 많을 것이다. 예를 들어, 0004에 시스템에 입력된 어떤 트랜잭션 'Y'는 2초 젊을 것이므로 우선권이 보다 이전의 것에 주어질 것이다.

추가로, 모든 데이터 아이템에는 최신의 읽기와 쓰기 타임스탬프가 주어진다. 이것은 시스템에게 해당 데이터 아이템에 마지막 '읽기와 쓰기' 운영이 언제 수행되었는지를 알려주는 역할을 한다.

1) Timestamp Ordering Protocol

Timestamp Ordering Protocol에서는 충돌하는 읽기/쓰기 운영에 있는 트랜잭션들 간에 연속성(serializability)을 보증한다. 이것은 작업들(tasks)의 충돌 짝이 해당 트랜잭션의 타임스탬프 값에 따라 수행되어야만 한다는 규약 시스템의 책임이다.

- 트랜잭션 T_i 의 타임스탬프는 $TS(T_i)$ 로 표시한다.
- 데이터 아이템 X 의 Read 타임스탬프는 R-timestamp X 로 표기한다.
- 데이터 아이템 X 의 Write 타임스탬프는 W-timestamp X 로 표기한다.

Timestamp Ordering Protocol은 다음과 같이 작업한다:

■ 만일 트랜잭션 T_i 가 read X 운영을 시작(issue)한다면,

□ 만일 $TS\ T_i < W\text{-timestamp}\ X$,

▲ 운영이 거부된다.

□ 만일 $TS\ T_i \geq W\text{-timestamp}\ X$,

▲ 운영이 수행된다.

□ 모든 데이터-아이템 타임스탬프가 갱신된다.

■ 만일 트랜잭션 T_i 가 write X 운영을 시작(issue)한다면,

□ 만일 $TS\ T_i < R\text{-timestamp}\ X$,

▲ 운영이 거부된다.

□ 만일 $TS\ T_i \geq R\text{-timestamp}\ X$,

▲ 운영이 거부되고 T_i 가 되돌려 진다(roll back).

□ 그렇지 않다면, 운영이 수행된다.

2) Thomas' Write Rule

이 규칙에서는 만일 $TS\ T_i < W\text{-timestamp}\ X$ 라면, 이 운영은 거부되며 T_i 는 roll back 한다고 말하고 있다.

Timestamp Ordering 규칙은 스케줄 보기를 연속적으로 만들기 위해 변경될 수 있다.

T_i 를 roll back(원래상태로 되돌아 가기) 하도록 만드는 대신에, '쓰기' 운영은 스스로 무시된다.

XXII. DBMS - DEADLOCK

다중-처리 시스템에서, 교착상태는 자원을 공유하는 환경에서 바라지 않는(unwanted) 상황이다.

예를 들어, 한 세트의 트랜잭션 $\{T_0, T_1, T_2, \dots, T_n\}$ 을 가정해 보자. T_0 는 자신의 임무를 완수하기 위하여 자원 X 가 필요하다. 자원 X 는 T_1 에 의해 유지되고(held) 있으며 T_1 은 자원 Y 를 기다리고 있다. T_2 는 T_0 에 의해 유지되고 있는 자원 Z 를 기다리고 있다. 이처럼, 모든 처리과정들이 자원을 해제하기 위하여 서로를 기다리고 있다. 이런 상황에선 어떠한 처리과정도 자신들의 임무를 마칠 수 없다. 이러한 상황을 데드록(교착상태)라 부른다.

데드록은 시스템에 해가 되는 것이다. 시스템이 데드록에 걸리는 경우에, 데드록에 포함되어 있는 트랜잭션들은 roll back(원래상태로 되돌아 가기) 되거나 다시 시작하여야 한다.

1. Deadlock Prevention

시스템의 데드록 상황을 예방하기 위하여, DBMS는 트랜잭션이 막 시작하려는 자신의 모든 운영을 공격적으로(aggressively) 조사한다. DBMS는 운영들을 조사한 다음 만일 그것들이 데드록 상황을 만드는지를 분석한다. 만일 데드록 상황이 발생할 것 같다면, 그 트랜잭션은 결코 수행을 허락하지 않는다.

데드록 상황을 미리 결정하기 위하여 트랜잭션의 timestamp ordering mechanism을 사용하는 데드록 예방 스킴(scheme)이 존재한다.

1) Wait-Die Scheme

이 스킴에서, 만일 트랜잭션이 이미 다른 트랜잭션에 의해 충돌 록과 함께 유지되고(held) 있는 자원인 데이터 아이템을 록하도록 요청한다면, 다음과 같은 두 가지의 가능성 중 한 가지가 발생할 것이다.

- 만일 $TS(T_i) < TS(T_j)$ 즉, 충돌 록을 요청하고 있는 T_i 가 T_j 보다 오래되었다면, T_i 는 데이터-아이템을 이용할 수 있을 때까지 기다린다.
- 만일 $TS(T_i) > TS(T_j)$ 즉, T_i 가 T_j 보다 최신이라면, T_i 는 없어진다(die). T_j 는 무작위로 연기되지만 동일한 타임스탬프에 따라 나중에 다시 시작한다.

이 스킴은 보다 오래된 트랜잭션으로 하여금 기다리도록 하지만 보다 최신의 것을 없애지는 않는다.

2) Wound-Wait Scheme

이 스킴에서, 만일 트랜잭션이 이미 다른 트랜잭션에 의해 충돌 록과 함께 유지되고(held) 있는 자원인 데이터 아이템을 록하길 요청한다면, 다음과 같은 두 가지의 가능성 중 한 가지가 발생할 것이다.

- 만일 $TS(T_i) < TS(T_j)$ 라면, T_i 는 T_j 를 강제로 원상태로 복귀 시킨다. 즉, T_i 는 T_j 를 타격(wounds)한다. T_j 는 무작위로 연기되지만 동일한 타임스탬프에 따라 나중에 다시 시작한다.
- 만일 $TS(T_i) > TS(T_j)$ 라면, T_i 는 자원을 이용할 때까지 강제로 기다린다.

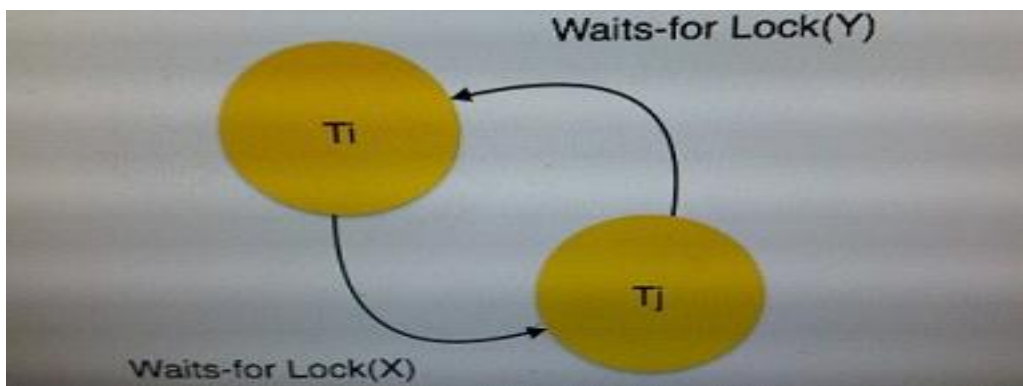
이 스킴은 최신의 트랜잭션이 기다릴 것을 허용하고 있지만, 오래된 트랜잭션이 보다 최신의 것에 의해 유지되는 아이템을 요청할 때, 보다 오래된 트랜잭션은 보다 최신의 것이 강제로 포기하도록 한 다음 그 아이템을 해제한다. 이러한 양쪽 모두의 경우에, 보다 나중의 단계에 시스템에 입력되는 트랜잭션은 중단된다(aborted).

2. Deadlock Avoidance

트랜잭션의 중단이 항상 실재적인 방법(approach)은 아니다. 대신에, deadlock avoidance mechanisms는 미리 어떤 데드록 상황을 탐지하는데 사용한다. “wait-for graph”와 같은 방법들을 이용할 수 있지만, 이것들은 자원의 인스턴스가 거의 없는 가벼운(lightweight) 트랜잭션의 시스템에만 적당하다. 덩치가 큰(bulky) 시스템에서, 데드록 예방 기법들이 잘 작동할 것이다.

■ Wait-for Graph

이것은 만일 어떤 데드록 상황이 발생한다면, 이것을 추적(track)하는데 사용할 수 있는 간단한 방법이다. 각 트랜잭션이 시스템에 입력되는 동안에, 하나의 노드가 만들어진다. 트랜잭션 T_i 가 어떤 다른 트랜잭션 T_j 에 의해 유지되는 X 인 아이템에 대해 록을 요청할 때, 하나의 직선(directed edge)이 T_i 에서 T_j 까지 만들어진다. 만일 T_j 가 아이템 X 를 해제한다면, 이들간의 선이 탈락(dropped)하며 T_i 는 그 데이터 아이템을 록한다.



여기서, 우리는 다음과 같은 두 가지의 approaches 중 어떤 것을 사용할 수 있다:

첫째, 이미 다른 트랜잭션에 의해 록되어 있다면, 그 아이템에 대한 어떠한 요청도 허용하지 않는다. 이것이 항상 가능한(feasible) 것은 아니며, 트랜잭션이 데이터 아이템을 무한정으로 기다리지만 결코 그것을 입수할 수 없는 starvation(굶주림)의 원인이 될 수 있다.

두 번째 옵션은 트랜잭션들 중의 하나를 원상태로 되돌려 놓는(roll back) 것이다. 보다 최신의 트랜잭션을 원 상태로 되돌려 놓은 것이 항상 가능한 것은 아니다. 어떤 relative algorithm의 도움을 받아, 중지해야 할 트랜잭션을 선택할 수 있다. 이런 트랜잭션을 **victim**이라 부르며, 그 처리과정을 **victim selection**이라 부른다.

XXIII. DBMS - DATA BACKUP

1. Loss of Volatile Storage

RAM과 같은 휘발성 저장기기는 모든 active logs, disk buffers, related data를 저장한다. 추가로, 이것은 현재 수행되고 있는 모든 트랜잭션도 저장한다. 만일 이러한 휘발성 저장기기가 느닷없이(abruptly) 박살난다면 무슨 일이 벌어질까? 분명한 것은 데이터베이스의 모든 logs와 active copies가 사라진다는 것이다. 이것은 데이터 복구에 필요한 모든 것을 잃게 함으로 복구는 거의 불가능하다.

휘발성 저장기기의 손실이 발생한 경우에, 다음의 기법들을 채택할 수 있다:

- 우리는 정기적으로 데이터베이스 콘텐츠의 안전을 위하여 다중의 단계에서 checkpoints를 가져야 한다.
- 휘발성 메모리 안에서 활동하는 데이터베이스의 상태는 정기적으로 안정된 저장기기로 넘겨야(dumped) 한다. 이 저장기기에는 또한 logs, active transactions, buffer blocks이 포함되기도 한다.
- <dump>는 데이터베이스 콘텐츠가 비-휘발성 메모리로부터 안전된 메모리로 넘겨질 때마다 log file에 표기될 수 있다.

2. Recovery

- 시스템이 실패로부터 복구될 때, 최신의 dump로 회복(restore)할 수 있다.
- checkpoints처럼 redo-list와 undo-list를 유지관리할 수 있다.
- 마지막 체크포인트까지 모든 트랜잭션의 상태를 회복하기 위하여, undo-redo list를 참고하여 시스템을 복구할 수 있다.

3. Database Backup & Recovery from Catastrophic Failure

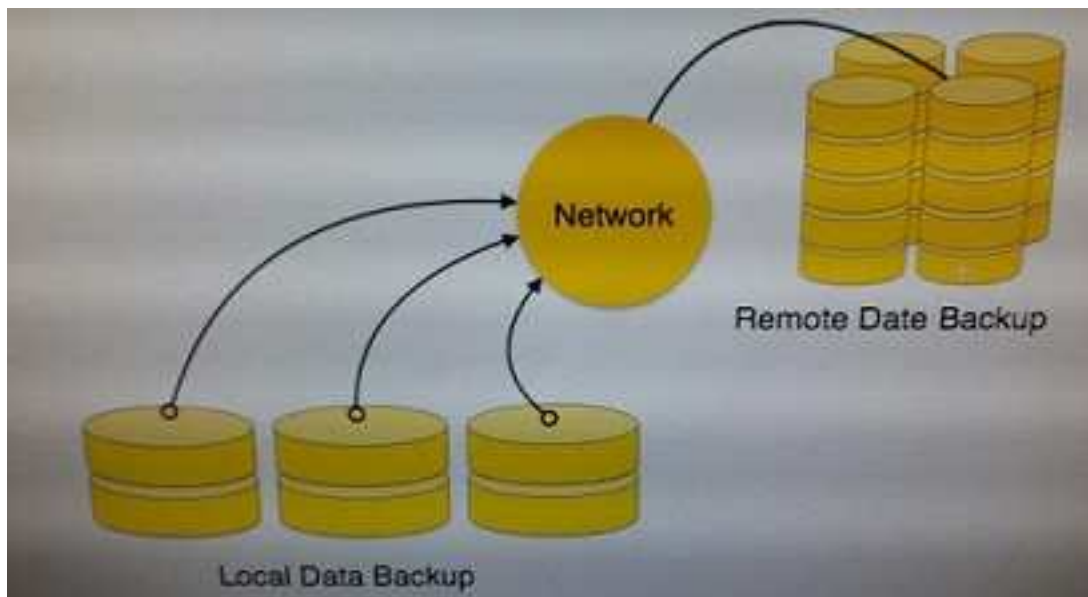
재앙적 실패는 안정되고 부차적인 저장기기가 훼손(corrupt) 되었을 때 발생한다. 저장 기기와 함께, 그 안에 저장된 모든 가치있는 데이터가 사라진다. 우리는 이러한 재앙적 실패로부터 데이터를 복구할 수 있는 두 가지 서로 다른 전략을 가지고 있다:

- Remote backup & minus - 이것은 데이터베이스 백업 사본을 재앙이 발생했을 때 다중할 수 있도록 원거리 장소에 저장하는 것이다.
- 대안적으로, 데이터베이스 백업은 마그네틱 테이프를 사용한 다음에, 그것들은 안전한 장소에 보관하는 것이다. 이러한 백업은 나중에 새롭게 설치된 데이터베이스에 백업 포인트를 사용하여 이관할 수 있다.

대용량(grown-up) 데이터베이스는 너무나 부피가 커서 빈번하게 백업할 수 없다. 이러한 경우에, 우리는 단지 그것의 logs만 살펴보고도 데이터베이스를 복구시킬 수 있는 여러 기법을 가지고 있다. 우리가 이 때 필요로 하는 모든 것은 모든 logs를 자주 백업하는 것이다. 데이터베이스는 한 주에 한번 백업되어야 하며, 매우 작은 logs라도 매일 또는 가능한 한 자주 백업되어야 한다.

4. Remote Backup

원격 백업은 데이터베이스의 제 1차적인 위치가 파괴될 경우에 대비하여 안전성하다는 느낌을 제공한다. 원격 백업은 offline, 또는 real-time, 또는 online을 사용할 수 있다. 오프라인인 경우에, 이것은 수작업으로 이루어진다.



온라인 백업 시스템은 보다 더 실시간적이며, 데이터베이스 운영자와 투자자에게는 생명의 은인(lifesaver)이다. 온라인 백업 시스템은 실시간 데이터의 모든 비트들이 서로 떨어진 두 곳에 동시에 백업되는 메카니즘이다. 이것들 중의 하나는 직접 시스템에 연결되어 있으며, 나머지 하나는 백업용으로 원격 장소에 보관되어 있다.

XXIV. DBMS - DATA RECOVERYC

1. Crash Recovery

DBMS는 매초에 수백 개의 트랜잭션이 이루어지는 매우 복잡한 시스템이다. DBMS의 durability(내구성)와 robustness(강건성)은 그것의 복잡한 구조와 기본이 되는 하드웨어와 시스템 소프트웨어에 달려있다. 만일 그것이 트랜잭션 중에 오류하거나 충돌(고장)한다면, 기대할 수 있는 것은 그 시스템이 손실된 데이터를 다중시킬 수 있는 어떤 유형의 알고리즘이나 기법을 따라야 한다는 것이다.

2. Failure Classification

문제가 발생한 곳을 알기 위하여, 우리는 오류를 다음과 같은 다양한 범주로 나눌 것이다:

1) Transaction Failure

트랜잭션이 만일 수행하는 것을 오류하거나 더 이상 갈 수 없는 포인터에 도달했을 때 중지되어야 한다. 이것을 단지 소수의 트랜잭션이나 처리과정들만이 손상되는 트랜잭션 오류라 부른다. 트랜잭션의 오류 원인은 다음과 같다:

- Logical errors - 트랜잭션을 완성할 수 없다. 왜냐하면 어떤 code error나 어떤 internal error condition이 있기 때문이다.
- System errors - DBMS가 그것을 수행할 수 없거나 또는 어떤 시스템 조건 때문에 데이터베이스가 멈춰야만 하기 때문에, 데이터베이스 시스템 스스로 활발한 트랜잭션을 중지시킨다.

2) System Crash

시스템을 느닷없이 중지시켜서 충돌의 원인이 되는 시스템 외부의 여러 가지 문제가 존재한다. 예를 들어, 전원공급의 휘방은 중요한 하드웨어의 오류뿐만 아니라 소프트웨어의 오류를 유발한다.

운영시스템(operation system)의 에러도 좋은 예이다.

3) Disk Failure

기술발달의 초기에, 이것은 매우 흔한 문제였다. 하드 디스크 드라이브와 저장 드라이브는 자주 오류하곤 했다.

디스크 오류에는 bad sections의 formation, 디스크로의 unreachability, disk head crash, 그리고 디스크 저장공간의 전체 또는 부분을 파괴하는 기타 오류 등이 포함된다.

3. Storage Structure

이미 앞에서 저장 시스템을 설명하였다. 간단하게 말해서, 저장 구조는 두 개의 범주로 나눌

수 있다:

- Volatile storage - 이름이 의미하는 것처럼, 휘발성 저장기기는 시스템 충돌에서 회생할 수 없다. 휘발성 저장기기는 CPU 바로 옆에 자리잡고 있다: 대체로 이것들은 chipset 그 자체에 내재되어 있다. 예를 들어, 메인 메모리와 캐쉬 메모리는 휘발성 저장기기의 좋은 예이다. 이것들은 빠르지만 단지 매우 작은 양만의 정보만을 저장할 수 있다.

- Non-volatile storage - 이러한 메모리는 시스템 충돌로부터 재생할 수 있도록 만들어졌다. 이것들은 데이터 저장 용량이 대규모이지만, 접근성에서 속도가 떨어진다. 이것들의 예로는 하드 디스크, 마그네틱 디스크, flash memory, 비-휘발성 *battery backed up* RAM이 있다.

4. Recovery and Atomicity

시스템에서 충돌이 발생할 때, 수행 중인 여러 가지의 트랜잭션과 데이터 아이템을 변경하기 위하여 열려 있는 다양한 파일이 있을 수 있다. 트랜잭션은 성질이(in nature) 원자인 여러 가지 operations로 만들어 진다. DBMS의 ACID 성질에 따라, 트랜잭션의 원자성은 하나의 전체로 유지되어야 한다. 다시 말해서, 모든 operations가 수행되거나 또는 어떠한 operations도 수행되지 않아야 한다.

DBMS가 충돌로부터 복구될 때, 다음과 같은 것들을 유지해야 한다:

- 수행되고 있는 모든 트랜잭션의 상태를 체크해야 한다.
- 트랜잭션이 어떤 operation의 한 가운데 있을 수 있다; DBMS는 이러한 경우에 트랜잭션의 원자성을 보장해야만 한다.
- 트랜잭션이 이제 완성되었는지 또는 원상태로 되돌려야 하는지를 체크해야 한다.
- 어떠한 트랜잭션도 비-일관성 상태로 DBMS에 남겨지지 않도록 해야 한다.

트랜잭션의 원자성을 복구할 뿐만 아니라 유지하는데 도움을 줄 수 있는 두 종류의 기법이 있다:

- 각 트랜잭션의 logs를 유지하기, 그리고 데이터베이스를 실제로 변경하기 전에 어떤 안정된 저장기기에 그것들을 기록하기(writing).
- 휘발성 메모리에서 변화가 이루어진 다음에, 현재(actual) 데이터베이스를 갱신하는 shadow paging을 유지하기.

5. Log-based Recovery

로그는 레코드들의 순서이며 트랜잭션에 의해 수행된 활동들에 대한 레코드들을 유지관리한다. 로그들이 실제로 변화가 일어나기 전에 작성(written)되서, 오류에서 자유로운 안정된 저

장 매체에 저장된다는 것은 매우 중요한 것이다.

로그-기반 복구는 다음과 같이 이루어진다:

- 로그 파일은 안정된 저장 매체에 보관된다.

- 트랜잭션이 시스템에 입력되어 수행이 시작될 때, 그것에 대한 로그가 작성된다.

<Tn, Start>

- 트랜잭션이 아이템 X를 변경할 때, 이것은 다음과 같이 로그에 작성된다:

<Tn, X, V1, V2>

이것은 Tn이 V1에서부터 V2까지 X의 값이 변했다는 것을 읽는다.

- 트랜잭션이 종료되었을 때, 이것은 다음과 같이 log한다:

<Tn, commit>

데이터베이스는 두가지 어프로치를 사용하여 변경할 수 있다:

- Deferred database modification - 모든 로그들은 안정된 저장매체에 작성된 다음에, 데이터베이스가 트랜잭션이 완료되면 갱신된다.

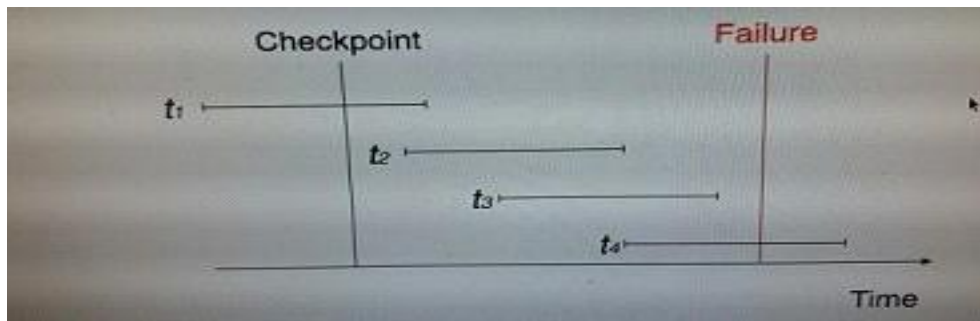
- Immediate database modification - 각 로그들이 실제적인 데이터베이스 변경 이후에 작성된다. 즉, 데이터베이스는 모든 operation이 끝난 다음에 변경된다.

6. Recovery with Concurrent Transactions

한 개 이상의 트랜잭션이 병렬적으로 수행될 때, 로그들은 인터리브드(interleaved) 된다. 다중 시에, 이것은 복구 시스템이 모든 로그들을 역추적(backtrack)하는 것을 어렵게 한 다음에, 복구를 시작한다. 이러한 상황을 완화하기 위하여, 현대의 대부분의 DBMS에서는 'checkpoints' 개념을 사용한다.

- Checkpoint - 실시간으로 그리고 실재의 환경에서 로그들을 보관하고 유지관리하는 것은 시스템에서 이용할 수 있는 모든 메모리를 가득 채울 수 있다. 시간이 흘러가면, 로그 파일이 너무 커져서 다룰 수 없게 될 수도 있다. checkpoint는 모든 이전의 로그들을 시스템에서 제거한 다음에 저장 디스크에 영구적으로 저장하는 메카니즘이다. checkpoint에서는 과거에 DBMS가 일관된 상태에 있었고 모든 트랜잭션이 완료됐던 지점인 포인트를 선언한다.

- Recovery - 동기성이 있는 트랜잭션의 시스템이 충돌한 다음 복구될 때, 다음과 같은 방식으로 취한다:



- 복구 시스템은 마지막 체크 포인트의 끝에서부터 역으로 로그들을 읽는다.
- 두 개의 리스트, undo-list와 redo-list를 유지한다.
- 만일 복구 시스템이 $\langle T_n, \text{Start} \rangle$ 와 $\langle T_n, \text{Commit} \rangle$ 또는 단지 $\langle T_n, \text{Commit} \rangle$ 로 된 로그를 본다면, 그 트랜잭션을 redo-list에서 갖다 놓는다.
- 만일 복구 시스템이 $\langle T_n, \text{Start} \rangle$ 가 있는 로그를 봤지만, 어떠한 완료 또는 중지 로그를 찾지 못한다면, 그 트랜잭션을 undo-list에 갖다 놓는다.

undo-list에 있는 모든 트랜잭션들은 그런 다음 undone된 다음, 그것들의 로그들은 제거된다. redo-list에 있는 모든 트랜잭션들과 이것들의 이전 로그들은 제거된 다음, 자신들의 로그를 저장하기 전에 redone 된다.

DBMS DONE!!