

I. DBMS - Overview

<https://www.tutorialspoint.com/dbms/index.htm>

- 데이터베이스: related data의 collection 이다.
- 데이터: 정보를 생산하기 위하여 처리될 수 있는 facts(사실)와 figures(숫자)의 집합체 이다. 데이터는 기록 가능한 사실을 포함하고 있으므로, 사실에 기초한 정보를 생산하는데 도움을 준다. 예를 들어, 만일 우리가 모든 책의 대출에 관한 데이터를 갖고 있다면, 우리는 상위 대출자들과 평균 대출권수를 구할 수 있다.
- DBMS: 정보를 보다 쉽게 검색, 조정, 생산할 수 있는 방법으로 데이터를 저장하는 시스템이다.

1. Characteristics

전통적으로 데이터는 파일 포맷으로 조직화 된다. DBMS는 새로운 개념이 아니며, 데이터 관리 분야의 전통적인 스타일에서 나타나는 단점을 극복하기 위한 것이다.

오늘날 DBMS의 특징은 다음과 같다:

1) Real-world entity

오늘날 DBMS는 디자인에서 보다 현실적인 개체(엔티티)들을 사용한다. 또한 개체들의 행위와 속성도 사용한다. 예를 들어, 도서관 데이터베이스는 개체로서 학생을 그리고 속성으로서 그들의 학과를 사용할 수 있다.

2) Relational-based tables

DBMS에서는 테이블을 작성하기 위하여 개체들과 그것들간의 관계들을 사용한다. 이용자는 이러한 테이블의 이름만 보고도 관련 데이터베이스의 구조를 이해할 수 있다.

3) Isolation of data and application

DBMS는 그것의 데이터와는 전혀 다르다. 데이터베이스는 능동적인 개체인 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, relation의 속성 중에서 중복된 값이 있는 것은 그 관계를 decomposition 한다. 여기서 정규화란 데이터의 잉여성을 줄이기 위한 과학적이고도 수학적인 절차를 말한다.

5) Consistency

일관성이란 데이터베이스의 모든 관계에서 모순이 존재하지 않는 상태를 말한다. 모순된 상태로 데이터베이스를 남겨놓으려는 시도를 탐지할 수 있는 많은 방법과 기법이 존재한다. DBMS를 파일처리 시스템과 같은 과거의 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 갖고 있다. 이용자는 DBMS에서 다양한 데이터를 검색하는데 필요한 다양한 filtering options를 선택할 수 있으나, 전통적인 파일처리시스템에서는 이러한 작업이 불가능하다.

7) ACID Properties

DBMS는 Atomicity, Consistency, Isolation, Durability 줄여서 ACID의 개념에 의존한다. 이 개념들은 DBMS에서 데이터를 다루는 transactions에 적용된다. 다중의 거래환경과 오류가 발생할 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 허용한다. 비록 다수의 이용자들이 동일한 데이터 아이템을 다루고자할 경우, 데이터 처리에 한계가 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 타 부서에서 데이터에 접근하는 것을 방지하는 어느 정도의 보안성을 제공한다. DBMS는 데이터베이스에 데이터를 입력할 때 그리고 나중에 그 데이터를 검색할 때 제약조건을 설정하는 방법을 제공한다. DBMS에서는 다양한 차원의 보안 기능을 제공하여 다양한 이용자가 다양한 기능을 갖춘 다양한 뷰를 사용할 수 있도록 한다. DBMS는 전통적인 파일 시스템처럼 디스크에 저장하지 않으므로, miscreants(악당)가 암호를 해독하기가 매우 어렵다.

2. Users

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



1) Administrators

행정가는 DBMS를 유지관리하며 데이터베이스의 행정업무에 책임을 진다. 또한 이들은 이것의 용도를 파악한 다음 이용자의 이용에 대해 책임을 진다. 이들은 이용자들의 접근 profiles를 만들며, 독립성 유지와 보안성 강화를 위한 제약요소를 설정한다. 행정가는 또한 시스템의 라이선스, 필수 도구, 그리고 유지관리에 필요한 하드 및 소프트웨어와 같은 DBMS 자원을 관리감독한다.

2) Designers

DBMS 디자이너란 데이터베이스의 디자인 분야에서 실제로 작업하는 사람들이다. 이들은 유지관리해야 할 데이터의 유형과 포맷에 대하여 꼼꼼하게 확인한다. 이들은 모든 개체, 관계, 제약조건, 뷰 등에 대해 식별 가능하도록 디자인 한다.

3) End Users

최종 이용자인 DBMS의 이익들을 실제로 수확하는(reap) 사람들이다. 최종 이용자는 단순 이용자에서부터 기업분석가와 같은 전문가까지 그 범위가 다양하다.

II. DBMS -ARCHITECTURE

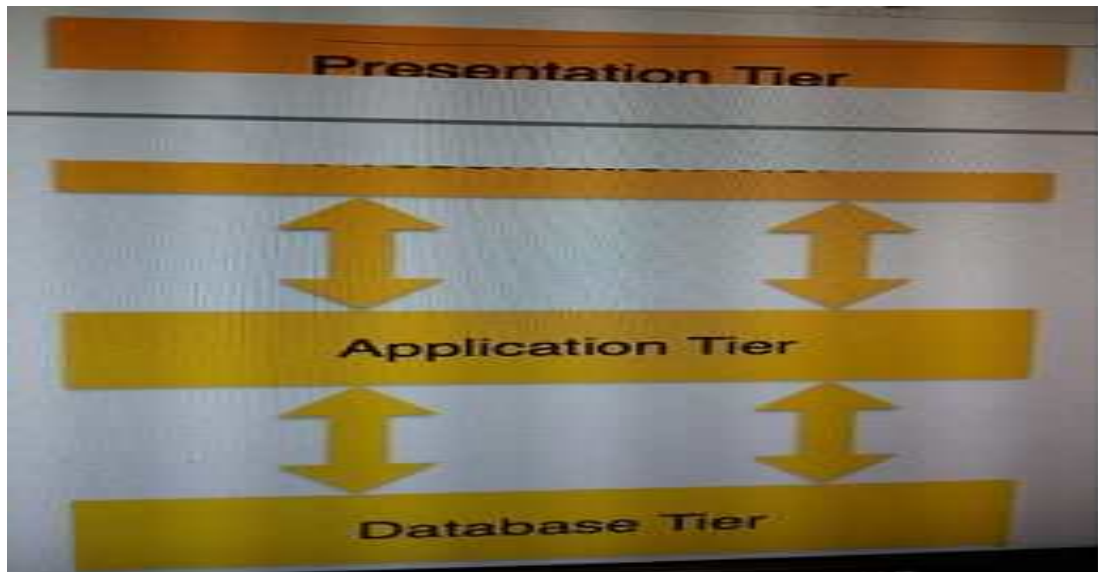
DBMS의 디자인은 그것의 구조를 결정한다. 이것은 중앙집중식, 분산식, 또는 계층식이 될 수도 있다. DBMS의 구조는 단일 티어(tier:층) 또는 다중의 티어로 표현되기도 한다. n-티어 구조는 전체 시스템을 서로 연관되는 독립적인 n-modules로 나누는데, 이 모듈들은 독립적으로 modified(변경), altered(개조), changed(교환), 또는 replaced(대체) 될 수 있다.

1-티어 구조의 DBMS에서 이용자는 직접 DBMS를 사용하는 유일한 개체이다. 이 구조에서 일어나는 변화들은 직접적으로 DBMS 그 자체에서 발생한다. 이것은 최종 이용자를 위한 간단한 도구들조차 제공하지 않는다. 데이터베이스 디자이너들과 프로그래머들은 대체로 단일-티어 구조를 선호한다.

만일 DBMS의 구조가 2-티어이라면, 이것은 DBMS에 접근할 수 있는 어플리케이션을 갖고 있다는 것이다. 프로그래머들은 그들이 어플리케이션을 사용하여 DBMS에 접근할 수 있는 2-티어 구조를 주로 사용한다. 이 때 어플리케이션 티어는 운영, 디자인, 그리고 프로그래밍과 관련해서 데이터베이스와는 완전하게 독립적이다.

1. 3-tier Architecture

3-티어 구조는 이용자의 복잡성과 데이터베이스에 있는 데이터를 사용하는 방법에 의거하여, 그것의 티어들을 서로 분리시킨다.



1) Database *Data* Tier

이 티어에서, 데이터베이스는 쿼리 처리 언어와 함께 존재한다. 또한 이 레벨에는 데이터와 그것들의 제약조건을 정의하는 관계들을 포함한다.

2) Application *Middle* Tier

이 티어에선 어플리케이션 서버와 데이터베이스에 접근하는 프로그램들이 포함된다. 이용자를 위해, 이 어플리케이션 티어는 데이터베이스의 요약(abstracted) 뷰를 제공한다. 최종 이용자는 어플리케이션의 범위를 벗어난 데이터베이스의 존재에 대해서는 알지 못한다. 다시 말해서, 이 어플리케이션 티어를 벗어난 이용자 누구도 다른 데이터베이스 티어를 인지하진 못한다. 어플리케이션 레이어는 중간에 존재하며 최종 이용자와 데이터베이스의 중재자로 행동한다.

3) User *Presentation* Tier

최종 이용자는 이 티어에서 활동하며 그들은 이 레이어를 벗어나 있는 어떠한 데이터베이스의

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

다중-tier 데이터베이스의 구조는 그것의 거의 모든 구성 요소를 독립적으로 변경할 수 있으나 비용이 많이 들어간다(highly).

III. DBMS -DATA MODELS

데이터 모델에서는 데이터베이스의 논리적 구조를 모델화하는 방법을 정의한다. 데이터 모델은 DBMS 디자인의 기본적인 개체들을 추상적으로 표현한다. 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 중복(redundancy)과 갱신 이상(update anomalies)을 발생시켰다.

1. Entity-Relationship Model

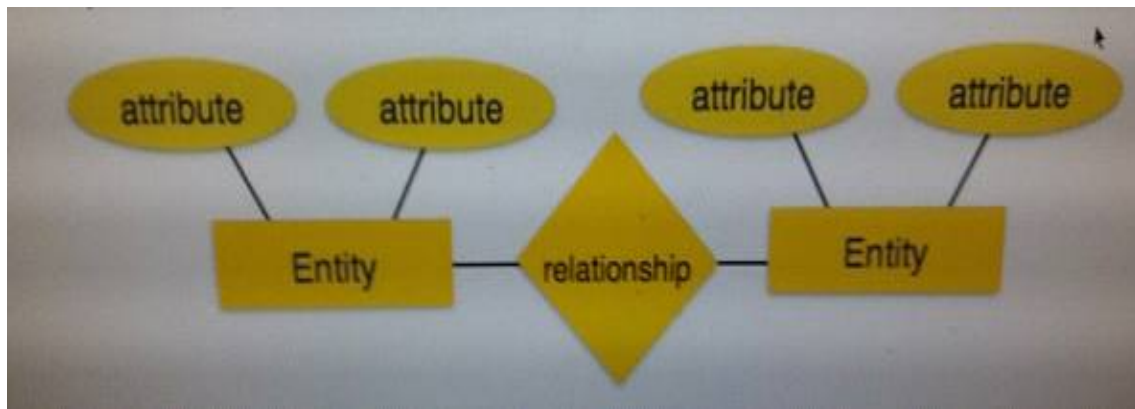
Entity-Relationship Model, 줄여서 ER 모델은 실세계의 개체들과 이것들 간의 연관성에 대한 개념을 근거로 삼고 있다. 실세계의 시나리오를 데이터베이스 모델로 공식화하는 동안, ER 모델은 entity set, relationship set, general attributes 그리고 constraints를 생성한다.

ER 모델은 데이터베이스의 개념적 디자인용으로 가장 널리 사용되고 있다

ER 모델은 다음과 같은 것을 근거로 작성된다:

- 개체와 이것의 속성(attributes)
- 개체간의 관계(Relationships)

이 개념들을 살펴보면 다음과 같다:



1) Entity

ER 모델에서 개체는 속성(attributes)이라 부르는 성질을 갖고 있는 실세계의 개체 이다. 모든 속성은 도메인(domain)이라는 값들의 집합체에서 정의된다. 예를 들어, 학교도서관 데이터 베이스에서 학생은 개체로 여겨진다. 학생은 다양한 속성 즉, 이름, 나이, 학급, 등과 같은 속성을 갖고 있다.

2) Relationship

개체들간의 논리적 결합(association)을 관계라 부른다. 관계는 다양한 방법으로 개체들을 연결(mapped) 한다. mapping cardinalities(연결 농도)는 두 개체 간의 결합의 수를 나타낸다. mapping cardinalities는 다음과 같이 나눈다:

- one to one(단사함수: 미스매칭발생 또는 전단사함수:모든 요소가 일대일로 매칭)
- one to many
- many to one(전사함수)
- 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.

<first-order predicate logic?>

문장의 주어가 개별 객체 (individual object)일 때 일차 술어논리(first order predicate logic)을 사용한다고 한다. 예를 들면 "Socrates is mortal (소크라테스는 죽는다)" 과 같은 경우이다. 반면에 주어가 또다른 술어로 구성되어 있을 때 우리는 second order logic 또는 higher order logic을 사용한다고 말한다. 예 들들면 "Being mortal is tragic (죽는다는 것은 비극이다)" 에서 'Being mortal' 과 같은 경우이다.

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

이 모델의 중요한 특징은 다음과 같다:

- 데이터는 relation이라 부르는 테이블에 저장된다.
- relation은 normalization(정규화)될 수 있다.
- 정규화된 릴레이션에서, 저장된 값은 원자(atomic) 값이다.
- relation의 각 로우(row)는 유일한 값을 나타낸다.
- relation의 각 칼럼(column)은 동일한 도메인에서 설정한 값으로 나타낸다.

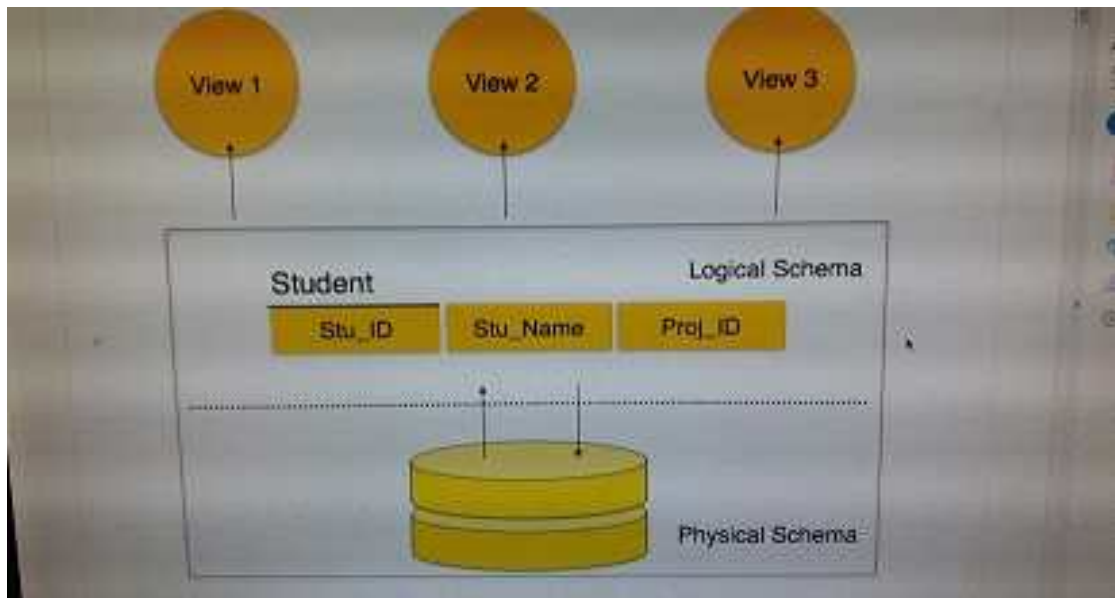
IV. DBMS - DATA SCHEMAS

1. Database Schema

데이터베이스 스키마란 전체 데이터베이스의 논리적 뷰를 표현하는 뼈대(skeleton structure)이다. 이것은 데이터를 조직하는 방법과 데이터간의 관계를 결합하는 방법을 정의한다. 이것에서 데이터에 적용할 수 있는 모든 제약조건을 공식화한다.

데이터베이스 스키마는 그것의 개체들과 그것들의 관계를 정의한다. 이것에는 스키마의 diagram(일람표)를 사용하여 표현할 수 있는 데이터베이스의 기술적 내역이 포함된다. 프로그래머가 데이터베이스를 이해하고 그것을 유용하게 만드는데 도움을 주도록 스키마를 디자인하는 사람이 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 제약조건을 정의한다. 이것은 테이블, 뷰, 그리고 순수성 제약조건(integrity constraints)을 정의한다.

2. Database Instance

위의 두 용어를 별도로 구분하는 것은 매우 중요하다. 데이터베이스 스키마는 데이터베이스의 골격(skeleton)이다. 이것은 데이터베이스가 분명히 존재하는 않을 때 디자인된다. 일단 데이터베이스가 기동되면, 그것에 어떤 변화를 가하기는 매우 힘들다. 또한 데이터베이스 스키마에는 어떤 데이터나 정보를 포함하지 않는다.

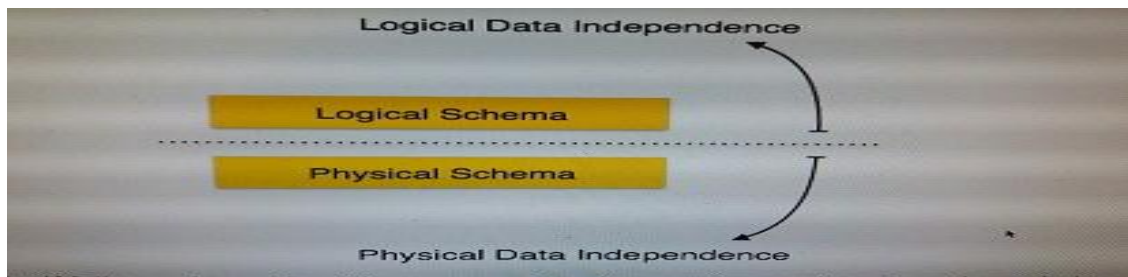
database instance란 어느 특정한 시간에 데이터를 운영하고 있는 데이터베이스의 상태를 말한다. 여기에는 데이터베이스의 snapshot(특정 시간에 파일시스템을 복사해서 (Point-In-Time Copy) 보관하다가 나중에 원본에 문제가 생겼을 때 복원을 해주는 기능)이 포함된다. 데이터베이스 instances는 시간에 따라 변하는 성질을 가지고 있다. DBMS는 데이터베이스 디자이너가 부여한 모든 validations, constraints, conditions를 부지런히 추적하여, 그것의 모든 instance(state)가 유효한 상태인지를 확인한다.

V. DBMS - DATA INDEPENDENCE

만일 데이터베이스 시스템이 다층 구조가 아니라면, 데이터베이스 시스템에 어떤 변화를 주기가 어렵다. 그러므로 데이터베이스 시스템은 앞에서 배운 것처럼 다층 구조로 디자인되어야 한다.

1. Data Independence

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



메타데이터 그 자체는 계층적 구조를 가지므로, 특정 층에 있는 데이터를 변경하려할 때, 다른 층에 있는 데이터는 영향을 받지 않는다. 이러한 데이터들은 서로 독립적이지만 서로 연결(mapped)되어 있다.

■ Logical Data Independence

논리적 데이터는 데이터를 내적으로 관리하는 방법에 관한 정보를 저장하고 있는 데이터베이스에 대한 데이터(data about database)이다. 예를 들어, relation은 데이터베이스에 저장되며, 그것의 모든 제약조건은 해당 relations에 적용된다.

논리적 데이터의 독립성은 일종의 기계적인 것이므로, 이것의 의미는 디스크에 저장된 실제적인 데이터로부터 자유롭다는 것이다. 다시 말해서, 테이블 포맷을 변경하더라도, 디스크에 있는 데이터는 바뀌지 않는다는 것이다.

■ Physical Data Independence

모든 스키마는 논리적이며, 실제 데이터는 디스크에 비트 포맷으로 저장된다. 물리적 데이터의 독립성이란 스키마나 논리적 데이터에 영향을 끼치지 않고 물리적 데이터를 변경시키는 원동력(power)을 말한다.

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

VI. ER MODEL - BASIC CONCEPTS

ER 모델은 데이터베이스의 개념적 모습(view)을 정의한다. 이것은 실세계의 개체들과 이들간의 결합에 관한 일을 한다. view 차원에서, ER 모델은 데이터베이스를 디자인하기 위한 훌륭한 선택으로 인식되고 있다.

1. Entity

개체는 생물이든 무생물이든(animate or inanimate) 쉽게 정의할 수 있는 실세계의 객체(object)이다. 예를 들어, 도서관 데이터베이스에서, 책, 이용자, 사서 등은 모두 개체가 될 수 있다. 그리고 이 개체들은 그것들의 정체성을 나타내는 몇 가지 속성이나 성질을 갖고 있다.

개체 집합(entity set)은 비슷한 유형의 개체들의 집합체이다. 개체 집합에는 유사한 값을 공유하고 있는 속성들을 갖고 있는 개체들이 포함될 수 있다. 예를 들어, 이용자 집합에는 모든 이용자가 포함될 수 있으며, 사서 집합에는 전체 직원들 중에서 단지 사서만을 포함할 수도 있다. 개체 집합은 분해(disjoint)되지 않아야 한다.

2. Attributes

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

속성에 할당된 값은 도메인에서 정의한다. 예를 들어, 학생 이름으로 숫자를 사용할 수 없고 문자만을 사용해야 한다면, 그 내용이 도메인에 정의되어 있어야 한다.

3. Types of Attributes

■ Simple attribute

단일속성은 원자 값(atomic value)으로 더 이상 세분되지 않는 값을 갖는다. 예를 들어, 이용

자의 전화번호는 10개의 디지털로 이루어진 원자 값이다.

■ Composite attribute

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

■ Derived attribute

유도속성은 데이터베이스에 실제로 존재하지 않는 속성이며, 자신들의 값을 다른 속성의 값을 유도하여 얻는 속성이다. 예를 들어, 도서관에서 수서한 책들 평균구입가는 데이터베이스에 직접 저장되지는 않지만, 이것은 도서 가격 속성의 값을 유도하여 구할 수 있다.

■ Single-value attribute

단일값 속성은 단일값만을 갖는 속성이다. 예를 들어, 도서의 청구기호 이다.

■ Multi-value attribute

복수값 속성은 다수의 값을 갖는 속성이다. 예를 들어, 어떤 이용자는 한 개이상의 전화번호를 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 집합체이며, 개체 세트 중에서 특정 개체를 유일하게 식별하는데 사용된다. 예를 들어, 등록번호는 소장자료들 중에서 특정 책을 유일하게 식별하는데 사용한다.

■ Super Key

한 개나 그 이상의 속성의 집합체로, 개체 세트에서 특정 개체를 집단적으로 식별할 때 사용한다.

■ Candidate Key

슈퍼 키 중 더 이상 줄일 수 없는(irreducible) 형태를 가진 것을 말한다. 더 이상 줄일 수 없다는 것은 슈퍼 키를 구성하는 속성(열) 중 어느 하나라도 제외될 경우 유일성을 확보할 수 없게 되는 것을 말한다. 이를 최소(minimal)라고도 한다. 즉, 행의 식별을 위해 필요한 특성 또는 그 집합이 후보 키이다.

■ Primary Key

기본키는 데이터베이스 디자이너에 의해 선택된 후보키들 중의 하나이며, 이것은 개체 세트를 유일하게 대표하는 속성이다.

5. Relationship

개체들 간의 결합을 관계라고 부른다. 예를 들어, '사서는 도서관에서 일하며, 학생은 이용자로 등록한다'라는 표현에서 '일한다'와 '등록한다'는 관계이다.

6. Relationship Set

비슷한 형태의 관계들로 집단화된 것을 관계 세트라 부른다. 개체처럼, 관계 또한 속성을 가질 수 있다. 이러한 속성을 기술 속성(descriptive attribute)이라 부른다.

7. Degree of Relationship

관계에 참여하는 개체의 수에 따라 관계의 정도가 정의된다.

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

! 라틴어 숫자 기수/서수 = 그리스어 명칭 알고 가기:

unus/primus=mono, duo/secundus=di, tres/tertius=tri, quatuor/quartus=tetra,
quinque/quintus=penta, sex/sextus=hexa, septem/septimus=hepta,
octo/octavus=octa, novem/nonus=nona, decem/decimus=deca.

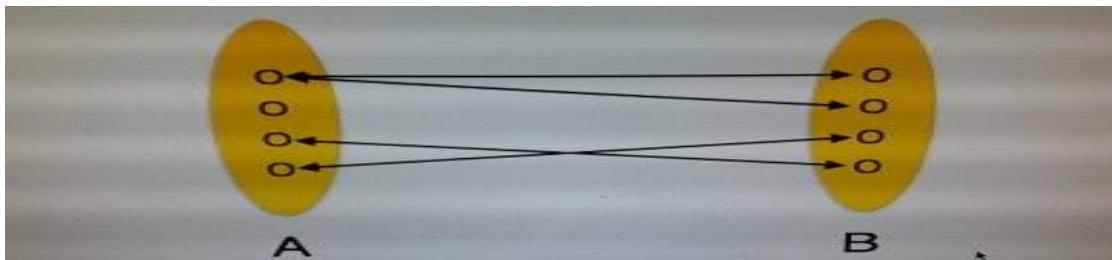
8. Mapping Cardinalities

카디널리티(집합원소의 갯수)는 한 개의 개체 세트에 있는 개체의 숫자를 정의하며, 이것은 관계 세트를 통해 다른 세트의 개체 숫자와 조합을 이룬다.



■ One-to-one

개체 세트 A에 있는 하나의 개체가 기껏해야(at most) 개체 세트 B에 있는 하나의 개체와 결합한다.



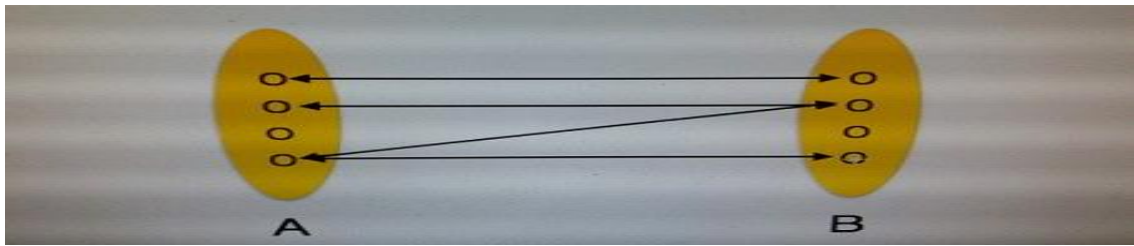
■ One-to-many

개체 세트 A에 있는 하나의 개체가 개체 세트 B에 있는 하나 이상의 개체와 결합할 수 있지만, 개체 세트 B에 있는 하나의 개체는 기껏해야 하나의 개체와만 결합할 수 있다.



■ Many-to-one

개체 세트 A에 있는 하나 이상의 개체가 기껏해야 개체 세트 B에 있는 하나의 개체와 결합할 수 있다. 그렇지만 개체 세트 B에 있는 하나의 개체는 개체 세트 A에 있는 하나 이상의 개체와 결합할 수 있다.



■ Many-to-many

A에 있는 하나의 개체는 B에 있는 하나 이상의 개체와 결합할 수 있다.

VII. ER DIAGRAM REPRESENTATION

이제 ER 모델을 ER 다이어그램으로 표현하는 방법에 대해 알아보자. 어떤 객체(예를 들면, 개체, 개체의 속성, 관계 세트, 관계 세트의 속성)라도 ER 다이어그램을 사용하여 표현할 수 있다.

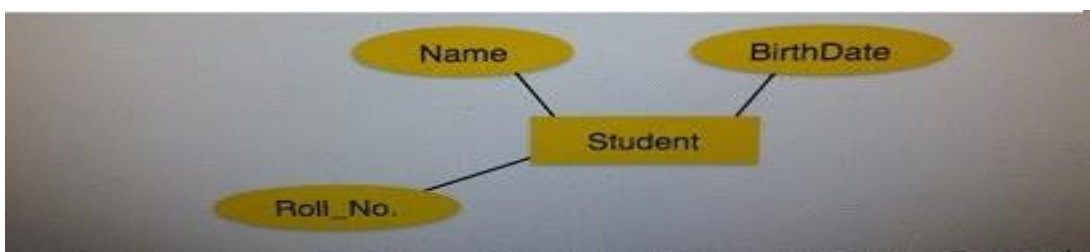
1. Entity

개체는 사각형으로 표현한다. 사각형은 그것이 의미하는 개체의 이름을 갖는다.

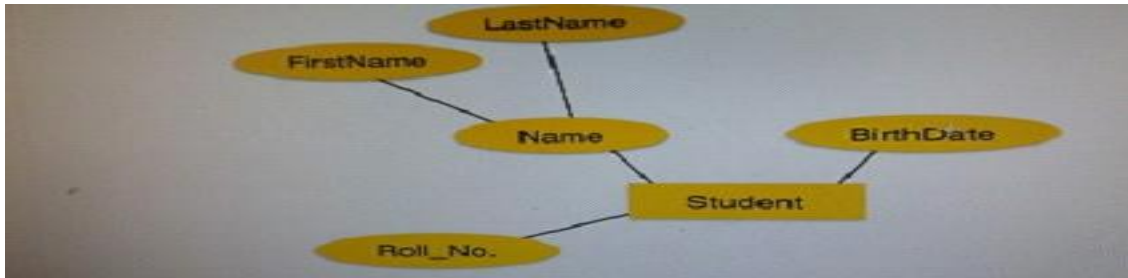


2. Attributes

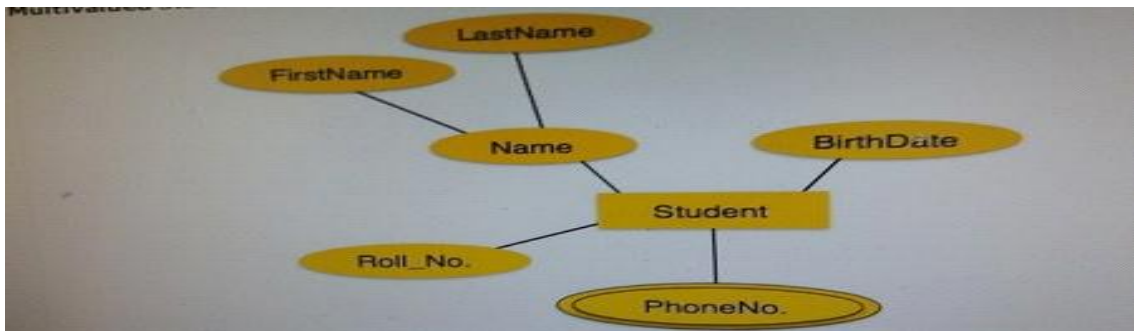
속성은 개체의 성질이다. 속성은 타원형(ellipse)으로 표현한다. 각각의 타원형은 하나의 속성을 표현하며 개체 *rectangle* 와 직접 연결된다.



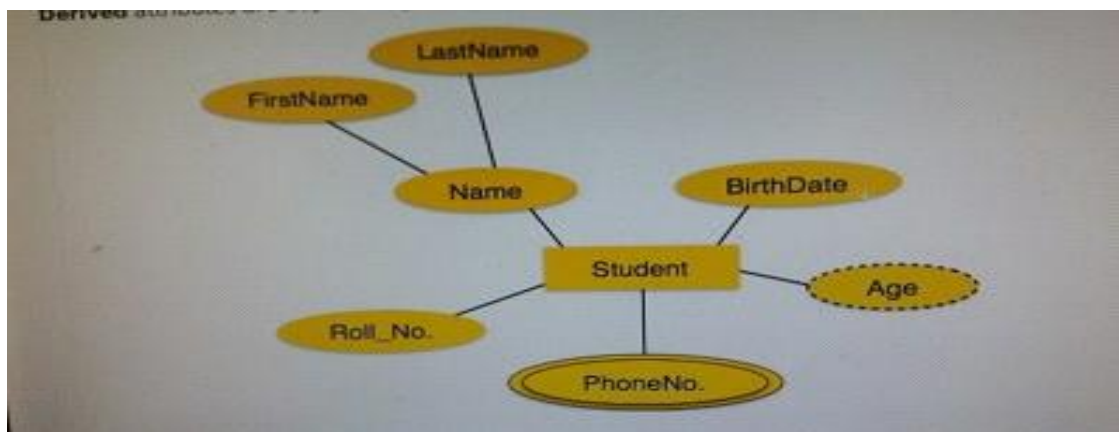
복합속성은 나무구조처럼 세분된다. 각각의 노드는 그것의 속성과 연결된다. 즉, 복합속성은 한 개의 타원형과 연결될 다중의 타원형으로 표현된다.



다중값 속성은 이중선 타원형으로 표현한다.



유도속성은 점선의 타원형으로 표현한다.



3. Relationship

관계는 다이아몬드형으로 표현한다. 관계의 이름은 다이아몬드 안에 표시한다. 관계에 참여하

는 모든 개체는 한 개의 직선으로 관계와 연결한다.

■ Binary Relationship and Cardinality

두 개의 개체가 참여하고 있는 관계를 binary relationship이라 부른다. 카디널리티는 관계에서 그것과 결합될 수 있는 개체의 현 상태(instance)의 수 이다.

□ One-to-one

개체의 단지 한 개만이 하나의 관계에 결합된 상태는 '1:1'로 표기한다. 아래의 이미지는 각각의 개체에 있는 단지 한 개만의 관계에 결합되어 있다는 것을 보여주고 있다. 이것을 one-to-one 관계라 부른다.



□ One-to-many

한 개 이상의 개체가 하나의 관계에 결합될 때, '1:N'이라 표기한다. 다음의 이미지는 왼쪽의 한 개 개체와 오른쪽에 있는 한 개 이상의 개체가 관계와 결합되어 있다는 것을 보여주고 있다. 이것을 one-to-many 관계라 부른다.



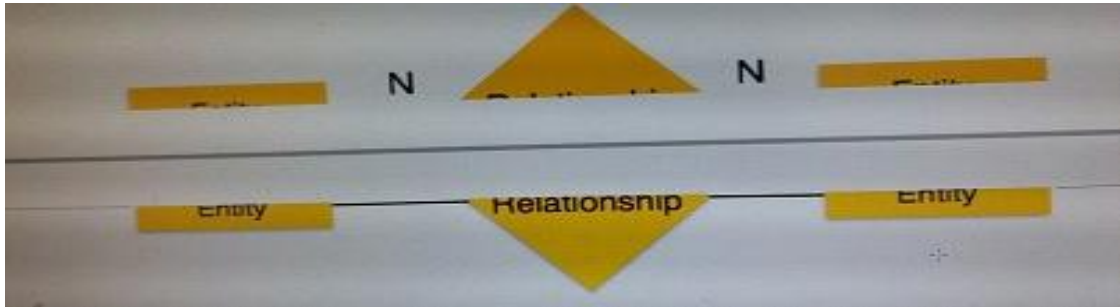
□ Many-to-one

한 개 이상의 개체가 하나의 관계에 결합될 때, 'N:1'이라 표기한다. 다음의 이미지는 왼쪽에 있는 한 개 이상의 개체와 오른쪽에 있는 단지 한 개의 개체가 관계와 결합되어 있다는 것을 보여주고 있다. 이것을 many-to-one 관계라 부른다.



□ Many-to-many

아래의 이미지는 왼쪽에 있는 한 개이상의 개체와 오른쪽에 있는 한 개 이상의 개체가 관계에 결합된 것을 보여주고 있다. 이것을 many-to-many 관계라 부른다.



■ Participation Constraints:

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



VIII. GENERALIZATION AGGREGATION

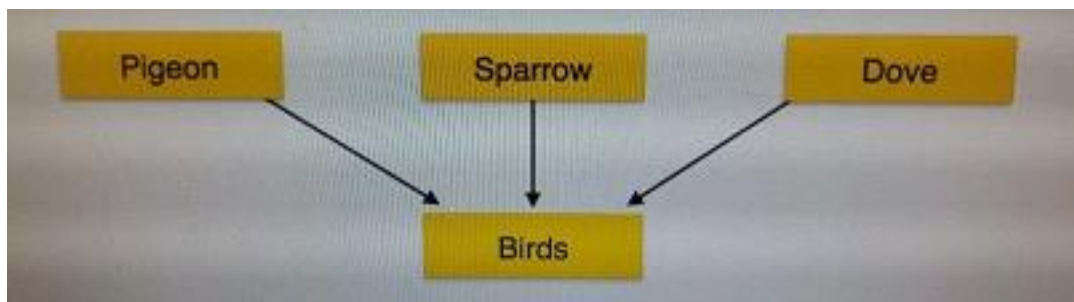
ER 모델은 개념적 계층방식으로 데이터베이스 개체들을 표현하는데 있어 장점을 갖고 있다. 계층이 올라갈수록, 이것은 개체들의 뷰를 전체적으로 일반화시키는 것이고, 계층이 내려갈수록 관계된 모든 개체의 내역을 상세하게 제공한다.

이러한 구조에서 위로 올라가는 것을 일반화(generalization)라 하며, 더 많은 일반화된 뷰를 표현하기 위해 더 많은 개체들을 결합시킨다. 예를 들어, 특별한 학생 이름 '철수'를 가지고 모든 학생을 일반화할 수 있다. 다시 말해서, '철수'는 학생 개체에 속하고, 더 나아가면, 그

학생은 '사람'개체에 속한다고 표현하는 것이다. 그리고 이러한 것의 역을 상세화 (specialization)라 부르는데, 예를 들면, '사람'은 '학생'이고, 그 학생은 '철수'라는 식이다.

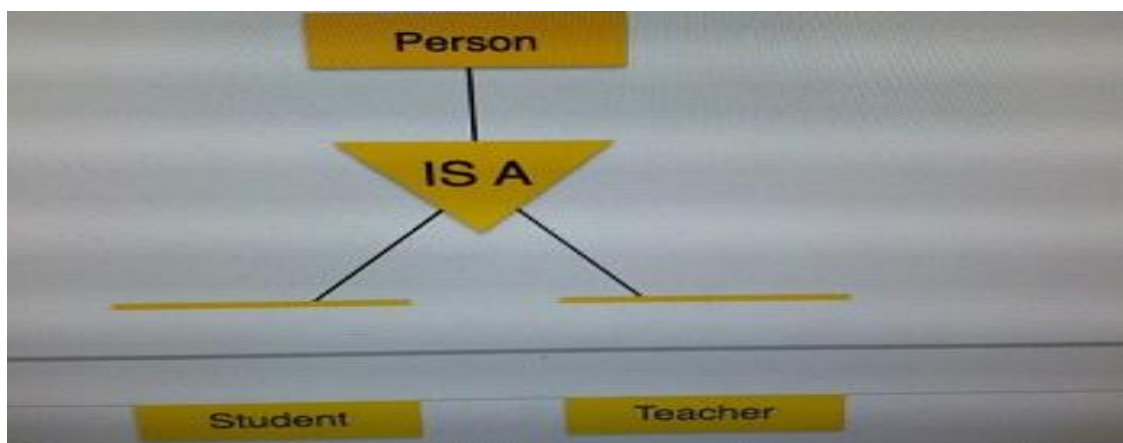
1. Generalization

위에서 말했듯이, 개체를 일반화하는 과정을 일반화라 부르며, 이 과정에서 디자인된 일반화 개체에는 그것과 연결된 모든 개체의 성질이 포함된다. 일반화에서, 많은 개체들은 그것들의 유사한 성질을 근거로 한 개의 일반화된 개체로 디자인한다. 예를 들어, pigeon, house sparrow, crow, dove는 모두 Birds 개체로 일반화될 수 있다.



2. Specialization

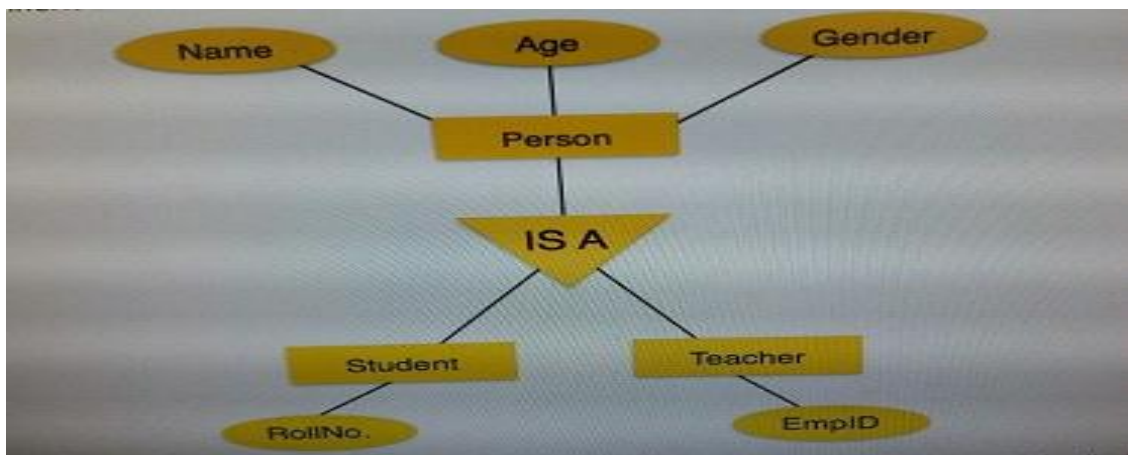
상세화는 일반화의 반대이다. 상세화에서, 한 무리의 개체들은 그것들의 특징을 근거로 하위 그룹으로 세분된다. 예를 들어, '사람' 그룹을 살펴보자. 사람은 이름, 생일, 성 등을 가지고 있다. 이러한 성질은 모든 사람에겐 공통인 것이다. 그러나 회사에서, 사람은 자신들의 역할에 따라 직원, 고용주, 소비자, 또는 판매자로 구분할 수 있다.



이것과 유사하게, 도서관 데이터베이스에서, 사람들은 개체로서 도서관에서 자신들의 역할을 근거로 사서, 이용자, 직원으로 세분될 수 있다.

3. Inheritance(상속)

위에서 설명한 ER 모델의 모든 특징을 사용하여 object-oriented programming으로 된 objects의 classes를 만들어 보자. 개체들의 상세한 내역을 이용자는 일반적으로 알 수 없다: 이러한 과정은 추상화(abstraction)라 부른다.



위의 그림에서, Name, Age, Gender와 같은 Person superclass의 속성들은 Student나 Teacher와 같은 subclasses(low-level) 개체들은 Person class의 속성들을 상속 받는다 (inherited).

IX. CODD'S 12 RULES

데이터베이스 시스템의 관계형 모델에 대한 많은 연구를 통해, Dr. Edgar F. Codd는 스스로 12개의 규칙을 마련하였다. Codd는 데이터베이스가 참된 관계형 데이터베이스가 되기 위해서는 다음과 같은 규칙에 따라야한다고 주장하였다.

이 규칙들은 관계형 기능을 사용하여 데이터를 관리하는 모든 데이터베이스 시스템에 적용 가능하며, 또한 기타의 모든 다른 규칙의 기본으로도 사용될 수 있는 기초이다.

Rule 1: Information Rule

데이터베이스에 저장된 데이터는 이용자 데이터나 메타데이터일 수 있으며, 테이블의 셀은 하나의 값을 가져야 한다. 데이터베이스에 있는 모든 값은 테이블 포맷으로 저장되어야 한다.

Rule 2: Guaranteed Access Rule

모든 한 개의 데이터 값, 즉, *value* 는 테이블 이름, 기본키, *rowvalue*, 그리고 *columnvalue* 와의 연결되어 논리적 접근이 보장되어야 한다. pointers와 같은 기타 방법 등이 데이터 접근에 사용되지 않아야 한다.

Rule 3: Systematic Treatment of NULL Values

데이터베이스에서 NULL은 체계적이고 확일적으로(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 사용자만 접근할 수 있는 data dictionary라고 하는 온라인 카탈로그로 저장되어야 한다. 이용자는 자신들이 데이터베이스에 접근하는데 사용하는 쿼리언어로 이 카탈로그에 접근할 수 있다.

Rule 5: comprehensive Data Sub-Language Rule

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

Rule 6: View Updating Rule

이론적으로 데이터베이스의 모든 뷰들은 갱신할 수 있다 하더라도, 시스템에 의해 갱신될 수 있어야만 한다.

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

데이터베이스는 고차원의 insertion, update, deletion을 지원해야 한다. 이것은 single row에만 적용되지 말아야 한다. 또한 data records의 세트들에 union, intersection, minus 기능들도 적용할 수 있어야 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. 이 규칙은

여러 가지 규칙들 중에서 가장 까다로운 것들 중의 하나이다.

Rule 10: Integrity Independence

데이터베이스는 그것을 사용하는 어플리케이션과 독립적이어야 한다. 이것의 모든 순수성 제약조건은 어플리케이션에서의 변화와 상관없이 독립적으로 변경될 수 있다는 것이다. 따라서 데이터베이스는 the front-end application 그리고 이것의 interface와는 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 가지도록 해야 한다. 이 규칙은 분산형 데이터베이스 시스템의 기초로 여겨지고 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 제약조건을 무시하지(bypass)도 못하도록 해야 한다.

X. RELATION DATA MODEL

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

1. Concepts

■ Tables

관계형 모델에서, relation은 테이블 이다. 이것의 포맷에는 개체들 사이의 관계를 저장한다. 테이블은 로우와 컬럼을 가지고 있으며 로우는 레코드를, 컬럼은 속성을 나타낸다.

■ Tuple

관계용 테이블의 단일 로우를 터플이라 부른다.

■ Relation instance

관계형 데이터베이스 시스템에서 터플들의 특정한(finite) 세트들은 릴레이션 인스턴스로 표현할 수 있다. 그렇지만, 릴레이션 인스턴스에는 중복된 터플이 없어야 한다.

■ Relation schema

릴레이션 스키마에는 릴레이션이름 즉, tablename, 속성, 그리고 그것들의 이름을 기술한다.

■ Relation key

각 로우는 테이블에 있는 로우를 유일하게 식별할 수 있는 relation key라는 한 개 이상의 속성을 갖는다.

■ Attribute domain

모든 속성들은 속성 도메인에서 미리 정의한 자신들의 값의 범위를 갖고 있다.

2. Constraints

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

- Key Constraints
- Domain Constraints
- Referential integrity Constraints

□ Key Constraints

릴레이션의 튜플(로우)에는 적어도 한 개 이상의 key가 존재해야 한다. 이것으로 유일하게 튜플들을 식별할 수 있기 때문이다. 이러한 속성들을 릴레이션의 키라 부른다. 만일 이러한 키가 한 개 이상이라면, 이것들을 candidate keys라고 부른다.

key 제약조건에서 강조하는 것은 다음과 같다:

- ▲ 한 개의 릴레이션에는 적어도 한 개의 키가 존재해야 한다.

키 제약조건을 또한 Entity Constraints라고도 부른다.

□ Domain Constraints

속성은 실세계 시나리오에서 특별한 값을 갖는다. 예를 들어, 나이는 단지 양의 정수만 가질 수 있다. 똑같은 제약조건이 모든 릴레이션의 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 제약조건은 외래키(Foreign Keys)의 개념과 연관이 있다. 외래키란 다른 릴레이션의 키를 참조하는 속성이다.

- ▲ 외래키는 NULL을 가질 수 없다.
- ▲ 참조 릴레이션의 기본키 값과 동일해야 한다

참조무결성 제약조건에서는 만일 릴레이션이 다른 또는 동일한 릴레이션의 키를 참조한다면, 그 키 요소가 존재해야 한다고 말하고 있다.

<간단요약>

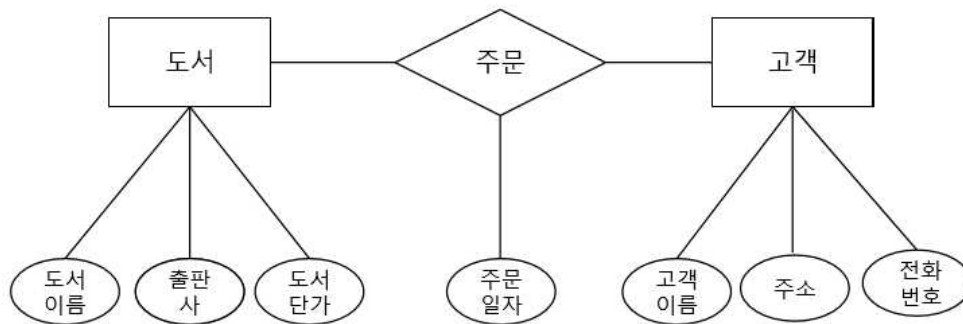


그림 6-5 개념적 모델링의 예



도서 (도서번호, 도서이름, 출판사이름, 도서단가)
 고객 (고객번호, 고객이름, 주소, 전화번호)
 주문 (주문번호, 고객번호(FK), 도서번호(FK), 주문일자, 주문금액)

그림 6-6 논리적 모델링의 예

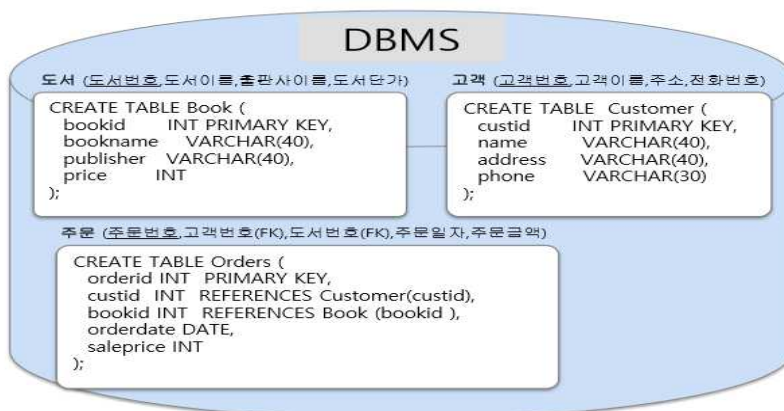


그림 6-7 물리적 모델링의 예

XI. RELATIONAL ALGEBRA

관계형 데이터베이스 시스템에서는 이용자의 데이터베이스 인스턴스 탐색을 돕기 위하여 쿼리 언어를 제공하고 있다. 두 가지 종류의 쿼리 언어가 있는데, 하나는 relational algebra(릴레이션 대수)이고 나머지는 relational calculus(릴레이션 해석)이다.

1. Relational Algebra

릴레이션 대수란 input으로 릴레이션의 인스턴스를 설정한 다음, output으로 릴레이션의 인스턴스를 얻는 절차지향적인 쿼리 언어이다. 이것에서는 쿼리를 진행하기 위해 여러 가지 연산자를 사용한다. 연산자는 unary이거나 binary 형일 수 있으며, 이것들은 자신들의 input으로 릴레이션을 사용한 다음에 output으로 릴레이션을 제공한다. 릴레이션 대수는 릴레이션과 관련에서 반복적으로 수행되며, 중간(intermediate) 결과물도 릴레이션으로 나타난다.

릴레이션 대수의 6가지 기본적 기능들은 다음과 같다:

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

1) Select Operation σ

이것은 릴레이션에서 설정된 조건을 만족시키는 터플들을 선택한다.

<표기법: $\sigma_p(r)$ >

σ : 선택 조건

r : 릴레이션

p : and, or, not과 같은 연산자를 사용할 수 있는 전치사적 논리공식(prepositional logic formula). 이러한 용어들은 $=$, \neq , \geq , $<$, $>$, \leq 같은 릴레이션형 연산자로 사용할 수도 있다.

<<예제>> Books인 테이블에서,

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

Output - 주제가 'database'인 책들을 터플로 선택한다.

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

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

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

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

2) Projection Operaton Π

이것은 주어진 조건을 만족시키는 칼럼들을 불러온다(project).

<표기법: $\Pi_{A_1, A_2, \dots, A_n}(r)$ >

A_1, A_2, \dots, A_n : 릴레이션 r 의 속성이름들.

릴레이션은 하나의 집합이므로, 중복된 로우들은 자동적으로 제거된다.

<<예제>>

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

Output - 릴레이션 Books로부터 subject와 author로 명명되어 있는 칼럼들을 선택하여 불러온다.

3) Union Operation \cup

이것은 두 개의 주어진 릴레이션들 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 릴레이션이거나 릴레이션의 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 제거된다.

<<예제>>

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

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

4) Set Difference

차집합의 결과는 한 개의 릴레이션에는 있으나 두 번째 릴레이션에는 없는 터플들이다.

<표기법: $r - s$ >

r 에는 있으나 s 에는 없는 모든 터플들을 찾는다.

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

Output - 책의 저자이지만 학술기사의 저자가 아닌 터플들을 찾는다.

5) Cartesian Product X

이것은 두 개의 서로 다른 릴레이션에 있는 정보를 하나로 결합시킨다.

<표기법: $r \times s$ >

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

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

<<예제>>

$\Pi_{author} = 'sunh'(Books \times Articles)$

Output - Books.author & Articles.author의 컬럼의 값이 'sunh'인 릴레이션을 제공한다.

6) Rename Operation ρ

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

<표기법: $\rho_x E$ >

표현 E 의 결과는 x 란 이름으로 저장된다.

이밖에도 추가적인 기능으로는 다음과 같은 것들이 있다:

- Set intersection
- Assignment
- Natural join

2. Relational Calculus

릴레이션 대수와 대조적으로, 릴레이션 해석은 비절차적 쿼리언어이다. 특히, 이것은 무엇을 해야 하는지는 말하지만, 그것을 해야 하는 방법에 대해서는 결코 설명하진 않는다.

1) Tuple Relational Calculus, TRC

터플 변수의 범위를 filtering 한다.

<표기법: {T | Condition }>

조건을 만족시키는 모든 터플들 T를 돌려준다.

<<예제>>

{T.name | Author(T) AND T.article = 'database'}

Output - 'database'라는 기사를 쓴 Author의 'name'에 들어 있는 모든 터플들을 결과로 보여준다.

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

<<예제>> 처 | 처 쓴

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

Output - 위의 쿼리는 바로 이전과 동일한 결과를 보여준다.

2) Domain Relational Calculus, DRC

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

<표기법: { a1, a2, a3, ..., an | P (a1, a2, a3, ... ,an) }>

a1, a2는 속성들이고, p는 내부 속성들에 의해 구축된 공식을 나타낸다.

<<예제>>

{< article, page, subject > | $\in \text{TutorialsPoint} \wedge \text{subject} = \text{'database'}$ }

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

TRC처럼, DRC 역시 existential and universal quantifiers를 사용하여 작성할 수 있다. DRC 역시 릴레이션형 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 개체-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 제약조건을 적용시킬 수는 없지만, 근접한 스키마를 제작할 수는 있다.

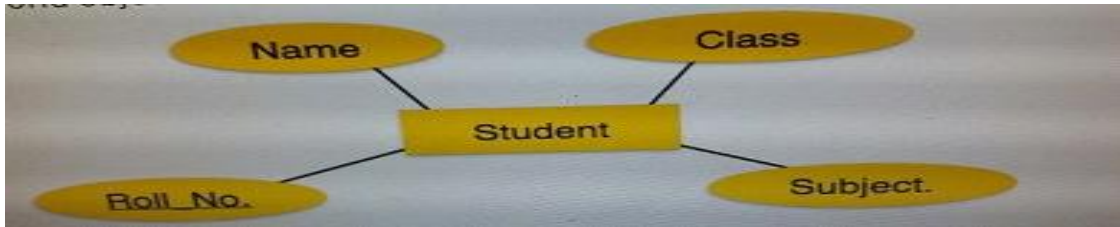
ER Diagram을 Relational Schema로 변환시킬 수 있는 여러 가지 처리절차와 알고리즘이 존재한다. 이것들 중 어떤 것은 자동화되어 있으며, 또 어떤 것은 수동으로 처리해야 한다. 우리는 이제 diagram contents를 relational basics에 맞추어 적용시킬(mapped) 것이다.

ER 다이어그램의 주요 구성요소는 다음과 같다:

- 개체와 이것의 속성.
- 관계- 개체의 결합.

1. Mapping Entity

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

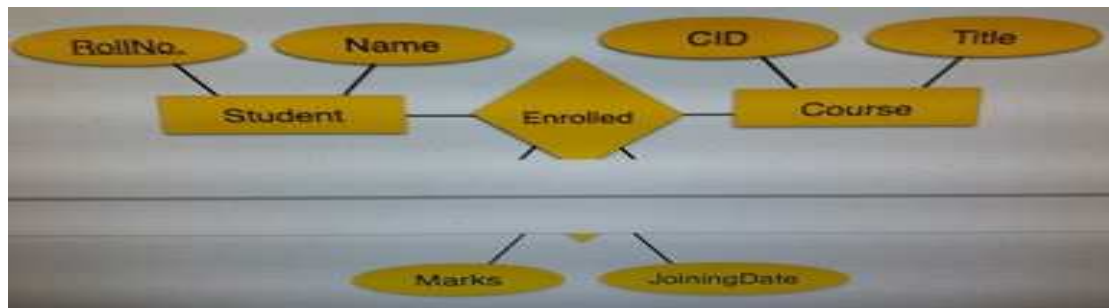


□ Mapping Process Algorithm

- ▲ 각각의 개체마다 테이블을 만든다;
- ▲ 개체의 속성들은 테이블의 필드가 되어야 하며, 각각 자신들의 독립된 data types을 갖는다;
- ▲ 기본키를 선언한다.

2. Mapping Relationship

관계는 개체들의 결합이다.

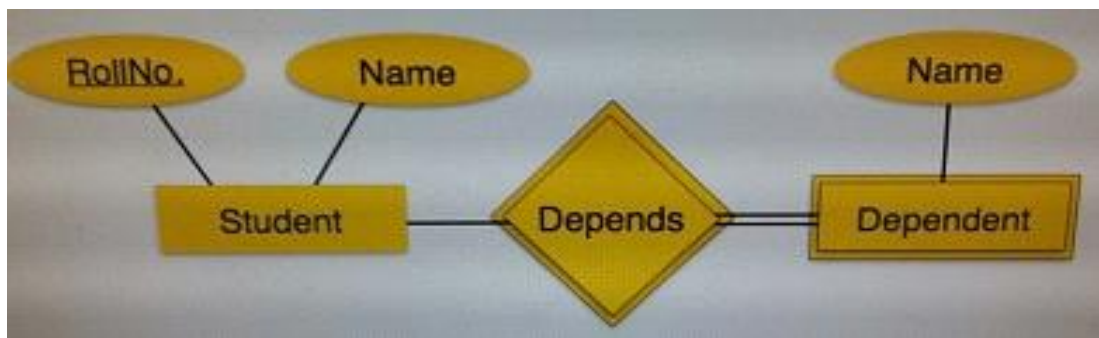


□ Mapping Process

- ▲ 릴레이션용 테이블을 만든다
- ▲ 자신들의 고유한 데이터 타입을 가지고 있으면서 테이블의 필드로 참여하고 있는 모든 개체들의 기본키를 지정한다.
- ▲ 만일 관계가 속성을 가지고 있다면, 그러한 속성들을 튜플로 디자인한다.
- ▲ 참여하고 있는 개체들의 모든 기본키를 대표하는 한 개의 기본키를 선언한다.
- ▲ 모든 외래키의 제약조건을 선언한다.

3. Mapping Weak Entity Sets

약한 개체 세트는 어떤 것도 기본키와 연결되지 않는다.

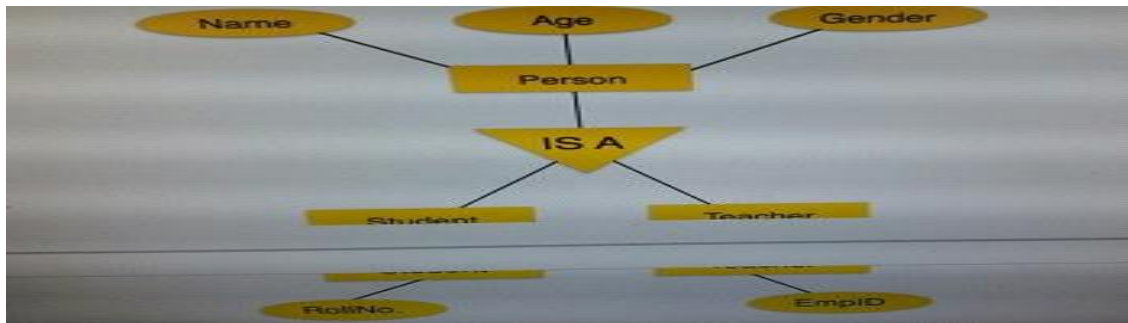


□ Mapping Process

- ▲ 약한 개체 세트용의 테이블을 만든다.
- ▲ 이것의 모든 속성들을 필드처럼 테이블에 추가한다.
- ▲ 개체 세트를 식별하기 위한 기본키를 디자인 한다.
- ▲ 모든 외래키의 제약조건을 선언한다.

4. Mapping Hierarchical Entities

ER의 specialization과 generalization은 계층적 개체 세트들의 형태로 표현된다.



□ Mapping Process

- ▲ 모든 고차원 개체에 대한 테이블들을 만든다.
- ▲ 저차원 개체용의 테이블도 만든다.
- ▲ 저차원 개체의 테이블에 고차원 개체의 기본키를 디자인 한다.
- ▲ 저차원 테이블에 저차원 개체들의 모든 속성들을 디자인 한다.
- ▲ 고차원 테이블의 기본키와 저차원 테이블의 기본키를 선언한다.
- ▲ 외래키의 제약조건을 선언한다.

XIII. SQL OVERVIEW

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

SQL은 데이터 정의어와 데이터 조작어로 구성되어 있다. SQL의 데이터 정의 속성들을 사용하면, 누구나 데이터베이스 스키마를 디자인하고 변경할 수 있으며, 데이터 조작 속성으로 는 데이터베이스를 대상으로 데이터를 저장하거나 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 DU_library;  
CREATE TABLE Books;
```

2) DROP

RDBMS로부터 명령어, 뷰, 테이블, 그리고 데이터베이스를 제거한다.

예제:

```
DROP DATABASE DU_library;  
DROP TABLE Books;
```

3) ALTER

데이터베이스 스키마를 변경시킨다.

```
Alter object_type object_name parameters;
```

<<예제>>

```
ALTER TABLE article ADD subject;
```

이 명령어는 릴레이션 **article**에 **subject**라는 이름을 가진 한 개의 속성을 추가한다.

```
ALTER TABLE article DROP subject;
```

이 명령어는 릴레이션 **article**에서 **subject**라는 이름을 가진 한 개의 속성을 제거한다.

2. Data Manipulation Language

SQL에는 data manipulation language, DML이 포함되어 있다. DML은 inserting, updating, deleting을 사용하여 데이터베이스 인스턴스를 조작하는데 사용한다. DML은 데이터베이스에서 데이터 변형과 연관된 모든 것에 책임을 진다. SQL은 DML에 대한 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 검색을 가능하도록 한다.

1) SELECT/FROM/WHERE

- SELECT

이것은 SQL의 기본적인 쿼리 명령어중의 하나이다. 이것은 릴레이션 대수의 projection 기능과 유사하다. 이것은 WHERE 조건 절에서 설정한 조건을 근거로 속성들을 선택한다.

□ FROM

이 어구(clause)에서는 선택하려거나 프로젝트하려는 속성들의 근거(argument)가 되는 릴레이션의 이름을 지정한다.

□ WHERE

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

<<예제>>

```
SELECT author_name
FROM book_author
WHERE age > 50;
```

위의 명령어는 릴레이션 **book_author**로부터 나이가 50세 이상인 author_name을 테이블로 보여준다.

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 Library (Author, Subject) VALUES ("sunh", "digital library");
```

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 n_name = value ...]
[WHERE condition];
```


<<예제>>

```
UPDATE Library SET Author="sunh" 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 Library WHERE Author="unknown";
```

XIV. DBMS-NORMALIZATION

1. Functional Dependency

함수종속성(FD)이란 한 개의 릴레이션에 있는 두 개의 속성간의 제약조건이다. 함수종속의 의미를 살펴보면, 만일 두 개의 튜플들이 속성 A_1, A_2, \dots, A_n 으로 동일한 값들을 갖고 있다면, 이 두 개의 튜플들은 속성 B_1, B_2, \dots, B_n 에서도 동일한 값을 가져야 한다는 것이다.

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

2. Armstrong's Axioms

만일 F가 함수종속성의 세트라면, F^+ 라는 F의 closure(결과)는 F에 의해 논리적으로 모든 의미가 부여된(imply) 함수종속성의 세트이다. 암스트롱의 공리는 한 세트의 규칙으로 이러한 것이 반복적으로 적용된다면, 함수 종속성의 closure가 거듭해서 생산된다는 것이다.

■ Reflexive rule(반사의 공리)

Y가 X의 부분집합이면, $X \rightarrow Y$ 이다.

■ Augmentation rule(확대의 공리)

만약 $X \rightarrow Y$ 이면, $XZ \rightarrow YZ$ 이다. 이것은 의존하고 있는 속성을 추가하는 것이지만 기본적

인 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 공리들에 따라, 다음과 같은 부수적 법칙을 유도해 낼 수 있다.

- 합집합의 성질(Union): 만일 $X \rightarrow Y$ 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(명확한)**

만일 FD $X \rightarrow Y$ 가 유지되고, Y 가 X 의 부분집합이라면, 이것을 trivial FD라 부른다.

■ **Non-trivial(명확하지 않은)**

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

4. Normalization

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

■ **Update anomalies(갱신이상)**

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

■ **Deletion anomalies(삭제이상)**

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

■ **Insert anomalies(추가이상)**

우리는 결코 존재하지 않은 레코드에 데이터를 추가하려고 한다.

정규화란 모든 이러한 이상현상을 제거하는 방법이며, 데이터베이스를 일관성 있는 상태로 유지하기 위한 것이다.

1) First Normal Form(1 NF)

1 NF에서는 릴레이션인 그 자체를 정의한다. 이 규칙에서 정의하는 것은 릴레이션의 모든 속성들은 원자값을 가지고 있어야 한다는 것이다. 원자값이란 쪼갤 수 없는 단일 값이다.

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

우리는 위의 테이블을 1 NF로 변경하기 위하여, 아래처럼 릴레이션으로 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:

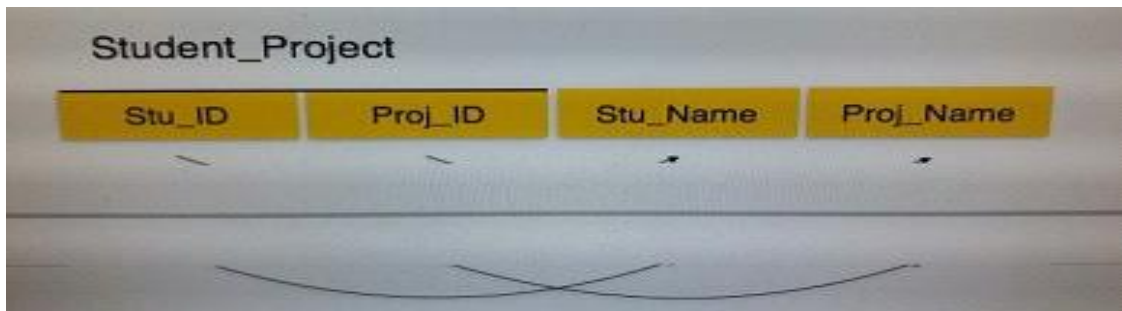
■ Prime attribute

기본키(prime-key)의 일부분의 속성을 기본 속성이라 한다.

■ Non-prime attribute

기본키의 기본 속성이 아닌 속성을 비기본 속성이라 한다.

만일 우리가 2 NF를 구성하려면, 모든 비기본 속성이 기본키 속성에 완전하게 함수적으로 의존하여야 한다. 다시 말해서, 만일 $X \rightarrow Y$ 라면, X의 부분집합 Y는 참으로 존재하지만, $Y \rightarrow X$ 는 거짓이다.



우리는 Student_Project 릴레이션을 보고 있으며, 이것의 기본키들은 Stu_ID와 Proj_ID이다. 규칙에 따라, 비-키 속성 즉, Stu_Name과 Proj_Name은 양쪽에 의존해야 하지만 개별적으로 하나의 기본키에만 의존하진 않는다. 그렇지만 우리가 알 수 있는 것은 Stu_Name은 Stu_ID에 의해, 그리고 Proj_Name은 Proj_ID에 의해 각각 식별이 가능하다는 것이다. 이러한 것을 partial dependency라 부르지만, 이것은 2 NF에서 허용되지 않는다.



우리는 위의 그림처럼 Student_Project 릴레이션을 두 개의 릴레이션 Student와 Project로 나누었다. 따라서 이제 각 릴레이션의 속성에서는 어떠한 부분적 의존성도 존재하지 않는다.

3) Third Normal Form(3 NF)

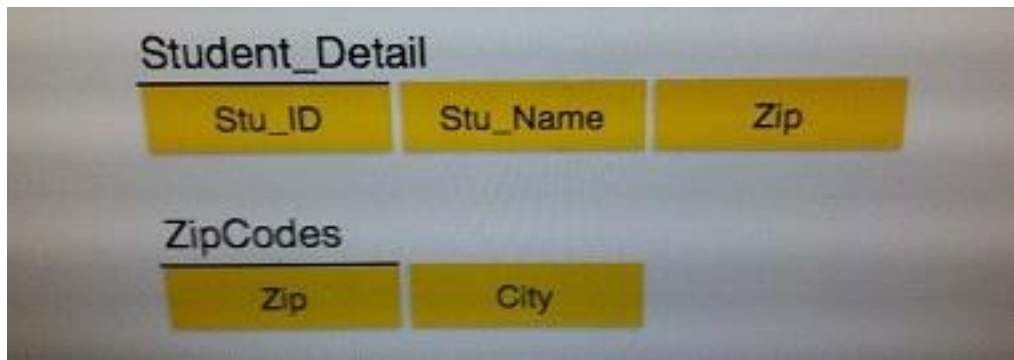
릴레이션이 3 NF가 되기 위해서는 2 NF에 있으면서, 동시에 아래의 내용을 만족시켜야 한다:

- 어떠한 비-기본 속성도 기본키에 전이적(transitive)으로 의존하지 않아야 한다.
- 어떤 non-trivial 함수 종속성이 $X \rightarrow Y$ 일 경우에, 다음의 둘 중의 하나여야 한다:
 - X는 슈퍼키이다.
 - Y는 기본 속성이 아니다.



위의 Student_Detail 릴레이션에서, Stu_ID는 키이면서 유일한 기본 키이다. 또한 City는 Stu_ID뿐만 아니라 Zip에도 명확하게 의존하고 있다. Zip은 슈퍼키이지만, City는 기본 속성이 아니다. 따라서 $\text{Stu_ID} \rightarrow \text{Zip} \rightarrow \text{City}$ 가 성립되며, 이것을 transitive dependency 이라 부른다.

이러한 릴레이션을 3 NF로 만들기 위해서, 우리는 다음과 같이 이러한 릴레이션을 두 개의 릴레이션으로 나누었다:



4) Boyce-Codd Normal Form(BCNF)

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

- 어떠한 non-trivial 함수 종속성이 $X \rightarrow Y$ 일 경우에, X는 슈퍼키여야 한다.

BCNF 를 위반하는 릴레이션에 대한 분해과정은 아래와 같다.

- BCNF 를 위반하는 nontrivial FD $X \rightarrow Y$ 를 찾는다.
- 두 개의 릴레이션으로 분해한다.
 - XY 로 구성된 릴레이션 하나
 - X 와 나머지 속성들로 구성된 릴레이션 하나

위 과정을 Student, Course, Instructor 예제에 적용시키면 아래와 같다.

- nontrivial FD Instructor \rightarrow Course 를 찾았다.
- 두 개의 릴레이션으로 분해한다.
 - Instructor, Course 하나
 - Instructor 와 나머지 속성들로 구성된 Instructor, Student

분해한 두 개의 릴레이션에서 기존 릴레이션에서 결정자역할을 했던 속성을 키로 해준다. 그러면 BCNF 까지 만족시키는 릴레이션 두 개가 생기게 된다.

이들 두 개의 릴레이션은 이제 BCNF에 있다.

XV. DBMS - JOINS

우리는 두 릴레이션들의 Cartesian product을 사용함으로써 얻을 수 있는 장점(모든 튜플들을 잠재적으로 pair로 제공한다는 것)에 대해 잘 알고 있다. 그러나 무수히 많은 속성들을 갖고 있는 수 천개의 튜플로 구성된 커다란 릴레이션을 만날 때처럼, 어떤 경우에는 이러한 카르티안 적을 사용하기가 쉽지 않다.

Join은 selection 절차에 수반되는 카르티안 적의 결합을 의미한다. Join 기능은 iff(if and only if = iff; 참이면)에 주어진 조건을 만족시킨다면, 서로 다른 릴레이션으로부터 두 개의 튜플을 짝(pair)으로 결합시킨다.

간단하게 여러 가지 종류의 Join에 대해 알아보자.

1. Inner Joins

두개의 집합 (A, B) 의 교집합 이다.

A {1,2,3}, B {2,3,4 } 이면 {2,3}이 inner join이다

1) Theta θ Join

췌타 조인은 췌타 조건을 만족시키는 서로 다른 릴레이션의 튜플들을 결합시킨다. 이 조인의 조건은 부호 췌타 : 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 릴레이션들이며, 이 속성들이 서로 어떠한 것도 공유하지 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

<<<입력>>>

Student_Detail = **STUDENT** $\bowtie_{\text{Student.Std} = \text{Subject.Class}}$ **SUBJECT**

<<<출력>>>

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) Natural join

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

자연 조인은 양쪽의 릴레이션에서 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

<<<입력>>>

Courses ⋈ **HoD**

<<<출력>>>

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

2. Outer Joins

썸타조인, 자연조인은 inner join이라 부른다. 이너조인에는 단지 대칭되는 속성들을 가지고 있는 튜플들만을 포함하며 그 나머지들은 결과 릴레이션에서 제외된다. 그러므로 결과 릴레이션에 참여하는 릴레이션들의 모든 튜플들을 포함시키기 위해서는 outer join을 사용하여야 한다.

여기에는 3 종류가 있다.

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

■ Left Outer Join($R \bowtie^L S$)

Left 릴레이션인 R에 있는 모든 튜플들이 결과 릴레이션에 포함된다. 만일 R의 튜플들이 Right 릴레이션인 S에 있는 튜플들과 어떠한 매칭도 이루어지지 않는다면, 결과 릴레이션의 S-속성들은 NULL로 처리된다.

<예제>:

Left

Courses

A	B
100	Database
101	Mechanics
102	Electronics

Right

HoD

A	B
100	Alex
102	Maya
104	Mira

<<입력>>

Courses \bowtie HoD

<<출력>>

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

■ Right Outer Join($R \bowtie = S$)

Right 릴레이션 S에 있는 모든 튜플들이 결과 릴레이션에 포함된다. S의 튜플들과 R의 튜플들이 서로 매칭되지 않는다면, 결과 릴레이션의 R의 속성들은 NULL로 처리된다.

<예제>:

<<입력>>

Courses $\bowtie =$ HoD

<<출력>>

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

■ Full Outer Join($R \bowtie = S$)

양쪽에서 참여하는 릴레이션들의 모든 튜플들이 최종 릴레이션에 포함된다. 양쪽 릴레이션에

서 매칭되는 튜플들이 없다면, 이것들의 각각의 비-매칭 속성들은 NULL로 표시된다.

<예제>:

Courses = ⋈ = HoD

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

XVI. DBMS - STORAGE SYSTEM

데이터베이스는 레코드를 포함하고 있는 파일 포맷으로 저장한다. 물리적으로 말하면, 데이터는 실제로 특정 기기의 전자기적 포맷에 저장된다. 이러한 저장장치의 유형들은 크게 3가지로 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 작지만, 속도가 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 사라진다.

■ Secondary Storage

2차 저장기기는 미래를 위한 데이터 저장용이나 백업용으로 사용된다. 2차 저장기기는 CPU

chipset이나 마더보드의 일부가 아니며, 마그네틱 디스크, 광학 디스크인 DVD, CD, 하드 디스크, flash drives, 그리고 마그네틱 테이프와 같은 메모리 기기들이 포함된다.

■ Tertiary Storage

3차 저장기기는 막대한 양의 데이터를 저장하기 위하여 사용한다. 이러한 저장기기는 컴퓨터 시스템의 외부에 있으므로, 이것들은 속도에서 가장 느리다. 이러한 저장기기는 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, 이것에는 CPU가 가장 자주 접근하는 데이터가 내장되어 있다.

가장 빠른 접근 속도의 메모리는 값이 가장 비싸다(costliest). 대용량 저장기기는 속도가 느리지만, 이것들은 비싸지 않다. 그렇지만 이것들은 CPU 레지스터나 캐쉬 메모리에 비해 막대한 양의 데이터를 저장할 수 있다.

2. Magnetic Disk

하드 디스크 드라이브는 현존하는 컴퓨터 시스템에서 가장 일반적인 2차 저장기기이다. 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 결합

시키는 기술이며, 이것들을 단일저장매체처럼 사용한다. RAID는 여러 가지 목적을 달성하기 위하여, 다중의 디스크가 서로 결합된 배열 형태를 갖고 있다.

XVII. DBMS - FILE STRUCTURE

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

1. File Organization

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



1) 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.

2) Sequential File Organization

모든 파일 레코드에는 해당 레코드를 유일하게 식별할 수 있는 데이터 필드인 *attribute*가 포함되어 있다. 순차적 파일 조직에서, 레코드들은 unique key field나 search key를 근거로

순차적으로 파일에 저장된다. 사실상, 물리적 형태로 모든 레코드들을 순차적으로 저장하는 것은 불가능하다.

3) Hash File Organization

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

4) Clustered File Organization

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

2. File Operations

데이터베이스 파일작업은 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, 파일들을 대상으로 하는 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 개체들 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 알려주는 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 지정한다. 이용자가 파일을 열고자 할 때, 파일 포인터의 위치를 결정할 수 있는 여러 가지 옵션이 존재하며, 파일 포인터의 바로 다음에 있는 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 중요하다. 파일에 있는 원하는 레코드에 파일 포인터를 지정하는 방법은 레코드의 정렬방식인 순차적인지 또는 클러스트 방식인지에 따라 다르다.

XVIII. DBMS - INDEXING

색인 메카니즘은 원하는 데이터에 접근하는 속도를 높이는데 사용된다. 예를 들면, 도서관의 저자목록이다.

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

Search key란 파일에 있는 레코드들을 찾는데 사용되는 속성이거나 속성의 세트를 말한다.

Index file은 아래와 같은 형태의 레코드들(index entries라 부름)로 구성된다:

search-key	pointer
------------	---------

index file은 전형적으로 original file보다 그 크기가 매우 작다.

기본적으로 색인에는 두 종류가 있다

- 1) Ordered indices: search keys가 정리된 순서로 저장되어 있다.
- 2) Hash indices: search keys가 'hash function'를 사용하는 'buckets' 간에 동일하게 분

산되어 있다.

1) Ordered(정렬) Indices

■ 정렬색인에서, 색인 엔트리는 search key value에 따라 정렬되어 저장된다. 예 - 도서관의 저자 목록.

■ Primary Index

순차적으로 정렬된 파일에서, 탐색기가 그 파일의 순차적 순서를 특정하고 있는 색인이다. clustering index라고도 부른다. primary index의 탐색기는 primary key가 일반적이긴 하지만 절대적인 것은 아니다.

Primary index는 잘 정돈된 색인파일에서 나타난다. 이 데이터파일은 key field에 의해 정돈된다. key field는 일반적으로 릴레이션의 기본키 이다.

▶ Clustering Index

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

■ Secondary Index

탐색기가 파일의 순차적 순서와는 다른 순서를 특정화하고 있는 색인이다. non-clustering index라 부르기도 한다.

2차 색인은 후보키인 필드로 생산될 수 있으며, 모든 레코드에서 또는 중복된 값을 가지고 있는 non-key에서 유일한 값을 갖는다.

■ Index-sequential file이란 primary index로 질서정연하게 순차적으로 되어 있는 파일을 말한다.

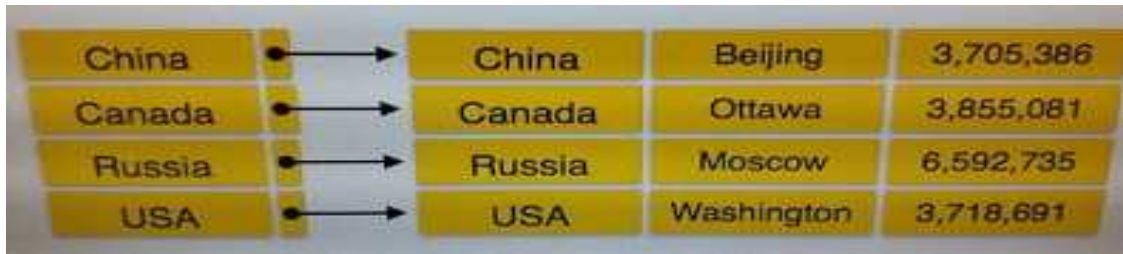
잘 정돈된 색인에는 두 가지 유형이 있다:

- Dense Index
- Sparse Index

1) Dense Index(조밀색인)

조밀색인에서는 파일에 있는 모든 탐색기의 값에 매치되는 색인레코드가 존재 한다. 이것은 탐색을 보다 빠르게 만들지만 색인 레코드 그 자체를 보관하는데 더 많은 공간을 필요로 한

다. 색인 레코드에는 탐색키의 값과 디스크에서 실제로 데이터를 지정하는 포인터가 포함된다.

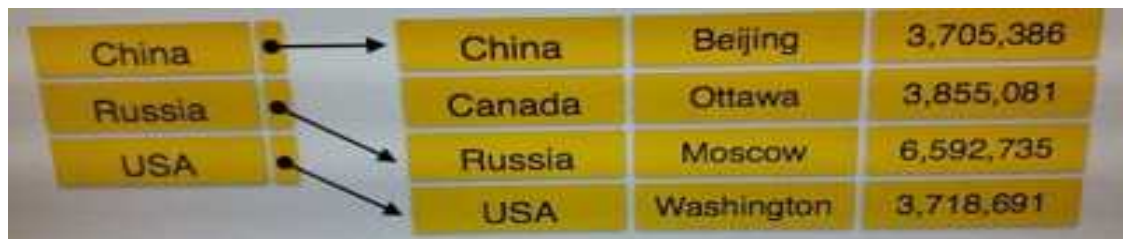


2) Sparse Index(희소색인)

희소색인에서는 단지 몇 개의 탐색키 값만을 위한 색인 레코드들을 포함한다. 이것은 레코드들이 탐색키에 따라 순차적으로 정돈될 때 적용될 수 있다.

탐색키 값 K 를 갖고 있는 레코드를 찾고자 할 때, 우리는 K 보다 작은 가장 큰 탐색키 값을 갖고 있는 색인 레코드를 찾은 다음에, 해당 색인 레코드에서 포인트하고 있는 레코드에서부터 순차적으로 파일을 탐색한다.

희소색인은 추가와 삭제 시에 적은 공간 사용과 적은 유지관리비가 들어가지만, 조밀색인에 비하여 레코드를 찾는 속도가 일반적으로 느리다.



■ Multilevel Index

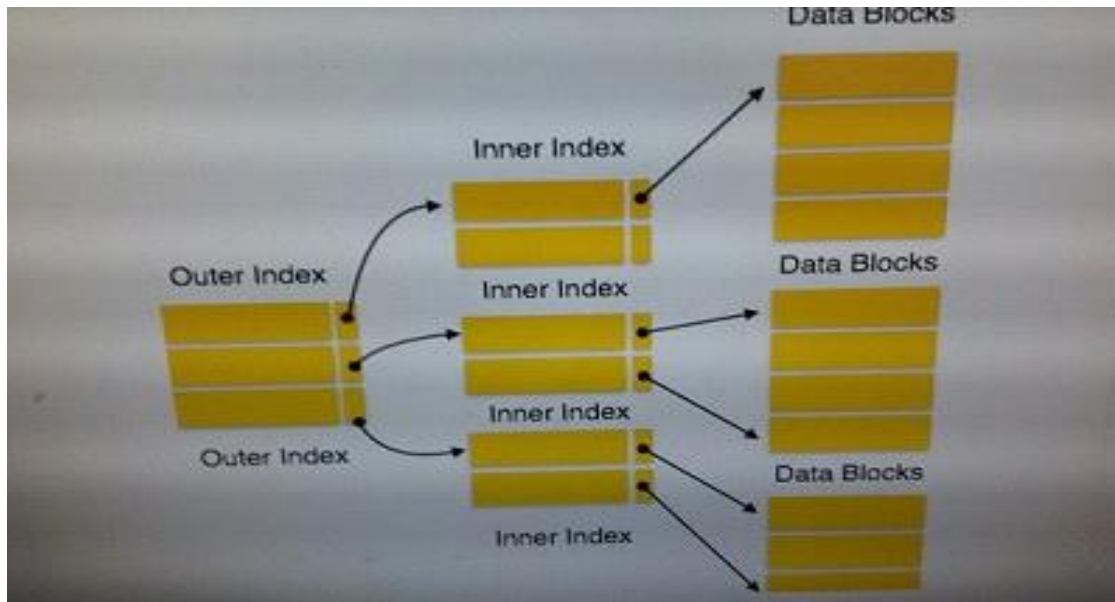
만일 primary index이 메모리에 적합하지 않다면, 접근 비용이 늘어난다. 색인 레코드에 접근하기 위한 디스크의 숫자를 줄이기 위하여, 디스크에 있는 primary index를 순차적 파일로 취급한 다음에 그곳에 희소색인을 만든다.

- ▶ outer index - primary index의 희소 색인.
- ▶ inner index - primary index file.

만일 outer index조차 너무 커서 주 메모리에 적합하지 않다면, 또다른 차원의 색인을 만들

어야 한다. 모든 차원의 색인들은 파일에서의 추가와 삭제가 갱신될 수 있어야 한다.

색인 레코드는 탐색키 값과 데이터 포인터로 구성되어 있다. 다단계 색인은 실제적인 데이터베이스 파일들에 맞춰서 디스크에 저장된다. 데이터베이스 규모가 늘어남으로써, 색인의 크기도 늘어난다. 탐색 작업의 속도를 높이기 위하여 주 메모리에 색인 레코드를 보관해야 한다는 필요성이 크게 대두되었다. 만일 단일 단계의 색인만을 사용한다면, 대규모 색인을 다중의 디스크 접근이 가능한 메모리에 보관할 필요는 없다.



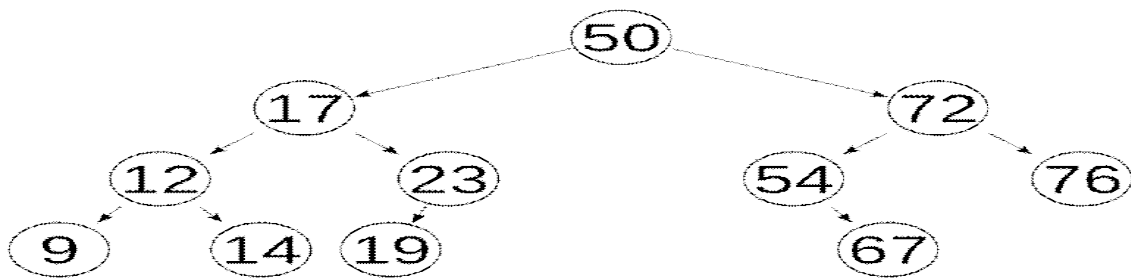
다단계 색인은 가장 바깥쪽 단계(the outermost level)에서 데이터 블록을 저장하기 위한 여러 개의 작은 색인으로 구성된다. 이것들은 주 메모리의 어디에서나 쉽게 사용할 수 있도록 도움을 준다.

■ B-Tree

B-Tree는 검색을 위한 자료구조 중에 하나이다. 기본적으로 B-Tree는 이진 탐색 트리의 단점을 보완하기 위해서 개발되었다. 이진 탐색 트리보다 더욱 더 효율적이고 빠르면서도 여러 개의 자식 노드를 가질 수 있게 해준다.

<이진(binary) 탐색 트리>

(※ 이진 탐색 트리의 자료구조 이미지)



- 가장 대표적인 비선형적인 자료구조로서 데이터의 삽입, 삭제, 탐색 등이 자주 일어나는 경우에 효율적인 자료구조이다.
- 왼쪽 서브트리에 있는 모든 데이터는 현재 노드의 값보다 작으며, 오른쪽 서브 트리에 있는 모든 노드의 데이터는 현재 노드의 값보다 커야 한다.
- 같은 값을 가지는 노드가 없어야 한다.

위와 같은 이진 탐색 트리는 균형이 잘 잡히지 않는 경우, 최악의 선형 검색 시간(하나의 데이터를 찾기 위해 전체 노드의 개수만큼 검색해야 되는 경우)을 소비할 수 있다. 따라서 그러한 단점을 보완하기 위해서 B-Tree라는 개념이 나오게 된 것이다. B-Tree는 이진 탐색 트리의 구조를 확장하여 더욱 많은 수의 자식을 가질 수 있도록 하였다. 따라서 같은 숫자의 자료를 저장하려고 할 때 B-Tree가 이진 탐색 트리보다 훨씬 트리의 높이가 낮아질 수 밖에 없다.

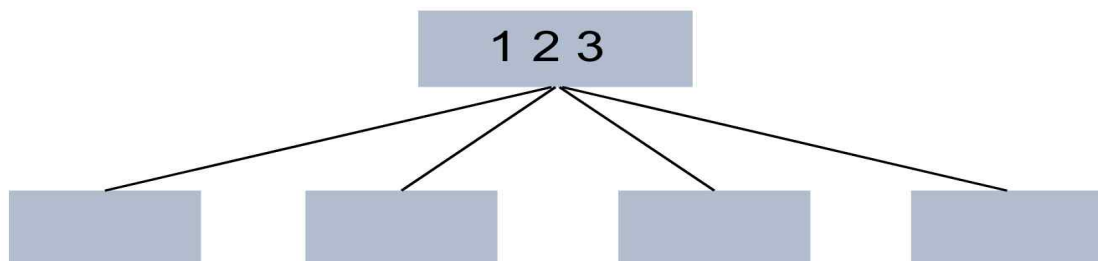
<B(Balanced)-Tree>

이진 트리가 자식 노드가 최대 2개인 노드를 말하는 것이라면 B-Tree는 자식 노드의 개수가 2개 이상인 트리를 말한다. 또한 노드내의 데이터가 1개 이상일 수가 있다. 노드내 최대 데이터 수가 2개라면 2차 B-Tree, 3개라면 3차 B-Tree 라고 말한다. '1, 2, 3, ... M차 B-Tree' 라고 부른다.

차수가 홀수인지 짝수인지에 따라 알고리즘이 많이 달라진다.

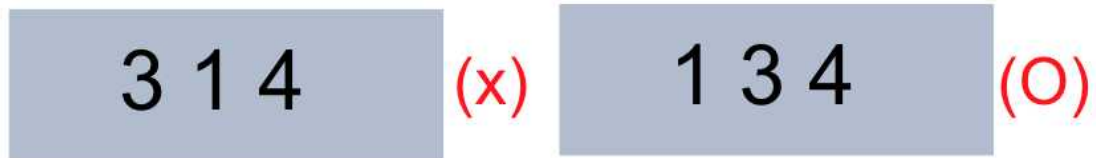
B-Tree 성립 조건에 대해서 자세히 알아보자.

- ▶ 노드의 데이터수가 n개라면 자식 노드의 개수는 n+1 개이다.

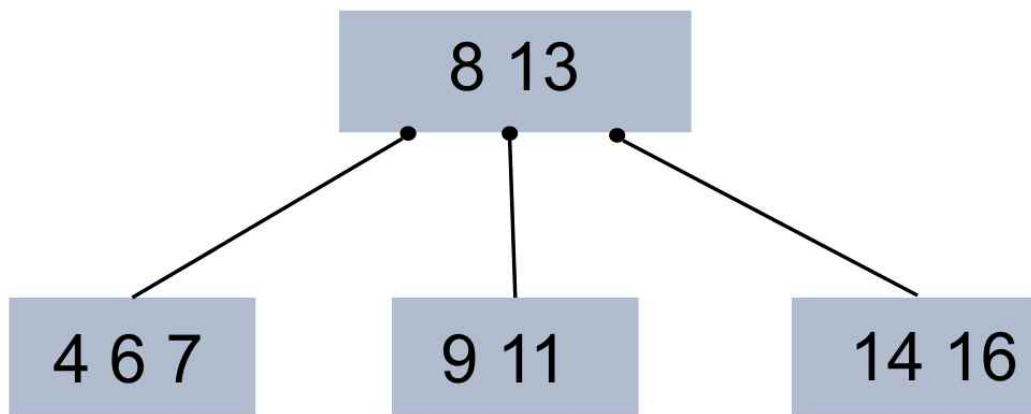


- root 노드에 데이터가 1, 2, 3 3개이므로 자식 노드의 개수는 4개 이다.

▶ 노드의 데이터는 반드시 정렬된 상태여야 한다.



▶ 노드의 자식노드의 데이터들은 노드 데이터를 기준으로 데이터보다 작은 값은 왼쪽 서브 트리에, 큰값들은 오른쪽 서브 트리에 이루어 져야 한다.

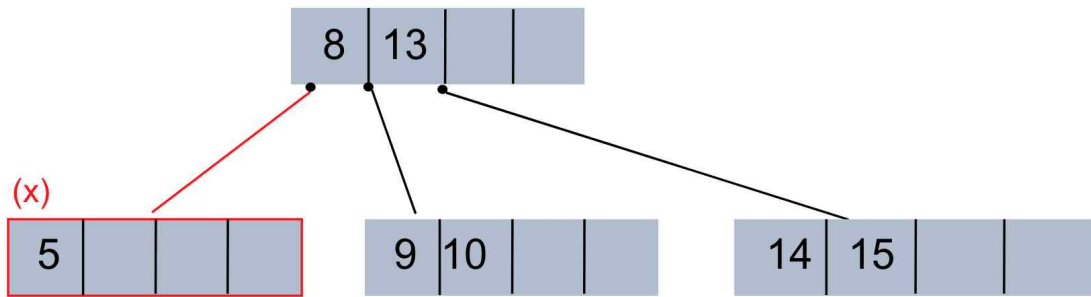


- root노드의 데이터는 8, 13 이다.

- 8보다 작은 데이터는 8의 왼쪽 서브트리에, 8과 13 사이의 값은 8의 오른쪽 13의 왼쪽 서브트리 (중간)에 , 13보다 큰값은 13 오른쪽 서브 트리에 값을 이루고 있다.

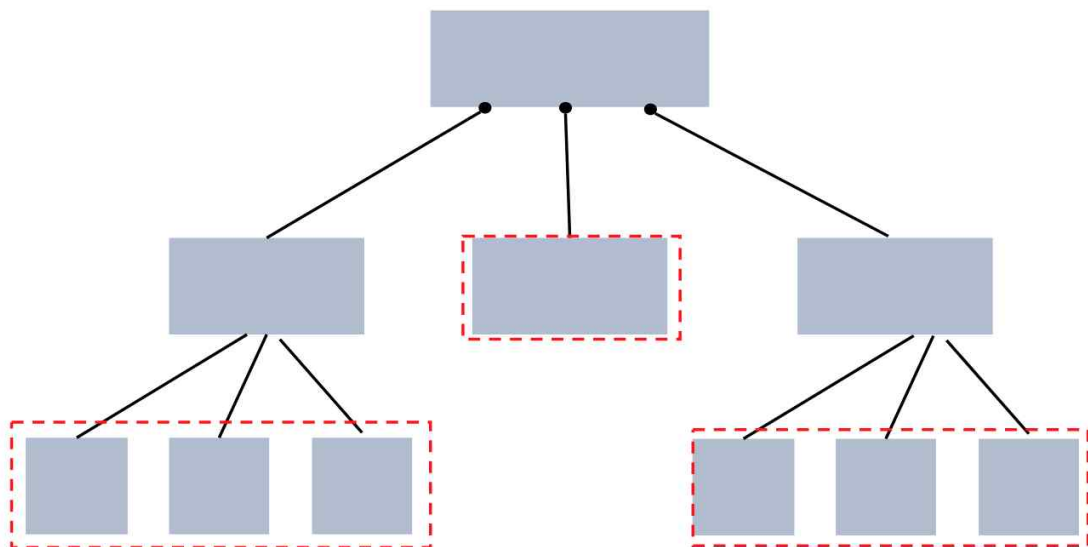
▶ Root 노드가 자식이 있다면 2개이상의 자식을 가져야 한다.

▶ Root 노드를 제외한 모든 노드는 적어도 $M/2$ 개의 데이터를 갖고 있어야 한다. 3차 B-Tree 까지는 1개의 데이터를 갖고 있어야 하니 고려하지 않아도 되는 조건이다. 4차 부터는 Root 노드를 제외하고 노드가 최소 2개의 데이터를 갖고 있어야 한다.



- 위와 같은 4차 B-Tree 에서 Root 노드의 데이터 8의 왼쪽 서브트리 노드가 데이터가 1개이므로 조건에 맞지 않다.

▶ Leaf 노드로 가는 경로의 길이는 모두 같아야 한다. 즉 Leaf 노드는 모두 같은 레벨에 존재해야 한다.



▶ 입력 자료는 중복될 수 없다.

■ 탐색

B-Tree 는 이진트리와 마찬가지로 작은 값은 왼쪽 서브트리, 큰 값은 오른쪽 서브트리에 이루어져있다. 탐색 하고자하는 값을 root 노드 부터 시작해 하향식으로 탐색해 나간다.

또한 B-Tree는 거기다가 자동으로 자료구조의 높이를 맞추어 준다는 점에서 검색 시간을 더욱 빠르게 해준다. 현존하는 자료구조가 균형이 맞다는 소리는 다시 말하면 검색할 때 더욱 빠른 시간에 검색이 가능하다는 것을 의미한다. 무엇보다 하나의 노드가 다량의 데이터를 가

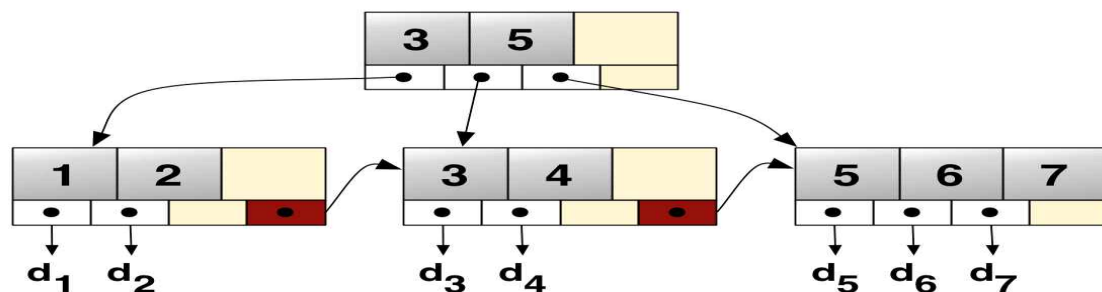
질 수 있다는 점은 많은 데이터가 들어 있는 경우에는 아주 효율적인 강점이라고 할 수 있다. 대표적으로 다양한 데이터베이스들은 내부적으로 이러한 B-Tree의 구조를 채택하고 있다. 그만큼 데이터가 많을수록 효율적이라는 의미이다. 많은 수의 데이터는 당연히 보조기억장치(SSD, 하드 디스크 등)에 저장되는데 이러한 외부 기억 장치들은 블록 단위로 입출력을 하는 것이 핵심이다. 한 블록의 바이트 수에 상관없이 불러오는 블록의 크기가 동일하기 때문에 노드의 크기를 대략 그 블록의 크기와 비슷하게 맞출 수 있다. 그러면 보조기억장치(외부 기억 장치)에 접근하는 횟수가 줄어들기 때문에 당연히 빠른 시간 데이터를 처리할 수 있다. 즉, 노드의 크기를 확장함에 따라 레벨이 낮아지고 데이터 처리 속도가 증가하게 된다는 것이다.

- 내부 검색 : 메모리 안에 들어있는 데이터를 검색할 때
- 외부 검색 : 보조기억장치 안에 들어있는 데이터를 검색할 때

B-Tree는 '외부 검색'에 굉장히 효율적이며, 특히나 저장하고 있는 데이터가 광범위하다면 외부 검색을 할 수 밖에 없는데 이 때 최대한 빠르게 검색할 수 있는 방법론을 제시한다. 이진 탐색 트리 같은 경우는 '내부 검색' 구조이기 때문에 메모리에 모든 데이터를 올려놓고 검색을 하게 되는데 이런 경우 데이터가 크게 되면 오류가 발생하지만 '외부 검색'은 하드 디스크 등을 이용할 수 있기에 데이터가 많아질 수 있으며, 많아도 커버가 가능하다.

<B⁺ Tree: quaternary tree>

B-트리는 특성상 순회 작업이 상당히 어렵다. B+ 트리는 색인구조에서 순차접근에 대한 문제의 해결책으로 제시되었다. B-트리에서는 특정 key 값이 하나의 노드에만 존재할 수 있지만 B+ 트리에서는 leaf 노드와 leaf의 부모 노드에 공존할 수 있다. B+ 트리의 비잎 노드(not leaf)들은 데이터의 빠른 접근을 위한 인덱스 역할만 한다. 잎(leaf) 노드들은 연결 리스트 형태로 서로 연결되어 있고 이를 순차집합(sequence set)이라고 하며 오름차순으로 정렬되어 있다. 고로 B+ 트리는 (기존의 B-트리 + 데이터의 연결 리스트)로 구현된 색인구조라고 할 수 있다. B⁺ Tree 색인은 indexed-sequential files의 대안이다.



- 키에 의해 각각 식별되는 레코드의 효율적인 삽입, 검색과 삭제를 통해 정렬된 데이터를 표현하기 위한 트리자료구조의 일종
- 각각의 인덱스 세그먼트 (블록 or Node) 내에 최대와 최소범위 키의 개수를 가지는 다계

층 인덱스 (multilevel index)로 구성된다

- B트리와 대조적으로 B+트리는 모든 레코드들이 가장 하위레벨에 정렬되어 있다. 오직 키들만 내부 블록에 저장된다.
 - 중간 레벨의 노드는 데이터를 찾기 위한 인덱스 세트(index set) 이다.

결론적으로, 이것은 B-트리의 변형 구조로 index 부분과 잎 노드로 구성된 순차 data 부분으로 이루어져 있다. Index 부분의 key 값은 잎에 있는 key 값을 직접 찾아 가는데 사용하고 모든 key 값은 잎 노드에 나열된다. 즉, index 부분의 key 값도 잎 노드에 다시 한 번 더 나열된다. 잎 노드는 순차적으로 linked list를 구성하고 있어서 순차적 처리가 가능하다.

B⁺ Tree는 아래와 같은 성질을 만족시키는 rooted tree이다:

- 1) 뿌리에서 잎까지 모든 paths가 동일한 길이를 갖는다;
- 2) 뿌리나 잎이 아닌 각각의 노드는 $n/2$ 개와 n 개 사이의 자녀 노드를 갖는다;
- 3) 잎 노드는 $(n-1)/2$ 와 $n-1$ 사이의 값을 갖는다;
- 4) 예외적으로: 만일 뿌리가 잎이 아니라면, 그것은 적어도 2개의 자녀 노드를 가지며, 만일 뿌리가 잎이라면, 즉 해당 트리에 어떠한 기타 노드도 없다면 그것은 0과 $(n-1)$ 사이의 값을 갖는다.

■ B⁺ Tree Node Structure

▶ 전형적인 노드

P ₁	K ₁	P ₂	...	P _{n-1}	K _{n-1}	P _n
----------------	----------------	----------------	-----	------------------	------------------	----------------

- K_i는 탐색키 값이다.
- P_i는 자녀 노드(비-잎 노드용인)를 가리키는 포인터들이거나 (잎 노드용인) 레코드들의 레코드나 buckets을 가리키는 포인터들이다.

▶ 노드에 있는 탐색키는 다음과 같이 정돈되어 있다:

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

■ Leaf Nodes in B⁺ Trees

잎노드의 속성은 다음과 같다:

- ▶ $i = 1, 2, \dots, n-1$ 일 경우에, P_i는 탐색키 값 K_i를 갖고 있는 파일 레코드를 가리키거나 탐색키 값 K_i를 가지고 있는 각각의 레코드인 파일 레코드를 지정하는 a bucket of pointers를 가리킨다.
- ▶ 만일 L_i, L_j가 잎 노드들이고 $i < j$ 라면, L_i의 탐색키 값은 L_j의 탐색키 값보다 작다.
- ▶ P_n은 탐색키 순서에 있는 그 다음의 잎 노드를 가리킨다.

■ B⁺ Tree의 개요

▶ B+Tree의 필요성

데이터가 순차적으로 처리되어야 할 때는 트리 구조 내에서 노드 사이를 오르내리는데 많은 처리 시간이 필요하게 되는데 이런 경우 B+트리를 사용한다.

- m-차수 트리의 한 종류로 노드의 Balance 를 맞춘 트리로 효율적인 균형 탐색 알고리즘을 제공한다.
- B 트리의 변형으로 인덱스 세트(index set)와 순차세트(sequence set)로 구성된다.

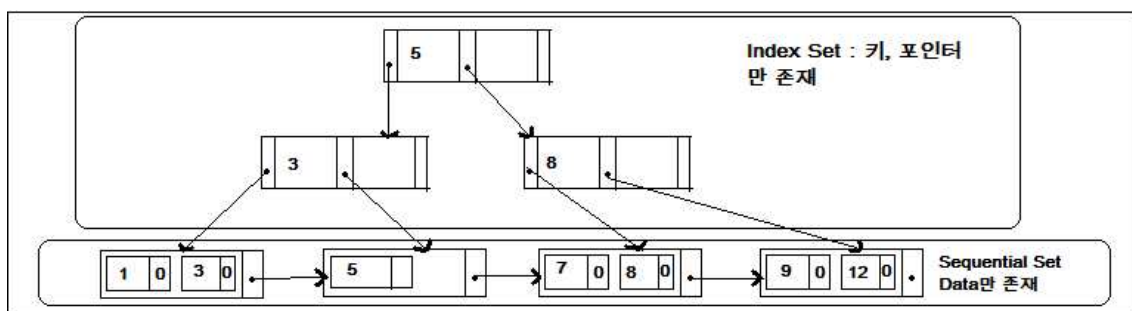
▶ B Tree와 차이점

- 인덱스 세트에 있는 키 값 : 리프 노드에 있는 키 값을 찾아가는 경로만 제공하기 위해서 사용한다.
- 인덱스 세트의 노드와 순차 세트의 노드는 그 내부 구조가 서로 다르다.
- 인덱스 세트에 있는 노드 : 키 값만 저장, 리프 노드 : 키 값과 포인터가 함께 저장된다.
- 순차 세트의 모든 노드가 순차적으로 서로 연결되어있다(순차 접근이 효율적).

▶ B Tree의 특징

- 루트는 0 또는 2에서 m개 사이의 서브트리를 갖는다.
- 루트와 리프를 제외한 모든 내부 노드는 최소 $\lceil m/2 \rceil$ 개, 최대 m개의 서브트리를 갖는다.
- 리프가 아닌 노드에 있는 키 값의 수는 그 노드의 서브트리 수보다 하나 적다.
- 모든 리프 노드는 같은 레벨이며, 한 노드 안에 있는 키 값들은 오름차순 이다.
- 리프 노드는 모두 링크로 연결되어 있다
- 리프 노드의 키 값의 수는 $\lceil m/2 \rceil$ 이상이다

■ B+tree의 구성



▶ B+ Tree의 구성 및 동작

- Index Set : 키, 포인터만 존재
- Index Set에 있는 Node는 Leaf를 찾아갈 수 있는 Key 값과 Pointer 값만 가지고 있다.
- Leaf Node관련 키값은 오름차순으로 정렬되어 있으며 삽입으로 인한 분기시 Sequence

Set에 연결되면서 순차성을 유지한다: 성능저하해결

- 삭제는 Leaf에서만 수행, 키 값은 Index Set에 유지하되 탐색 시에는 사용되지 않고 split 값으로만 사용한다.

▶ B+tree의 data 구조/특징

- 인덱스 세트 : 실제적인 키 값(Leaf node)를 찾아갈 수 있는 경로제공이 목적이다.
 - 인덱스 세트에 있는 노드는 키 값만 존재
 - 각 노드는 키 값과 Sub-tree에 대한 포인터를 포함
 - 인덱스 세트의 키 값은 Leaf node에 있는 키 값을 찾아 갈 수 있는 직접 경로를 제공
- 순차세트(Sequence Set)
 - Leaf Node값은 오름차순으로 정렬되어 있으며, 삽입으로 인한 분기 시 Sequence Set에 연결되면서, 순차성 유지 : 성능 저하 해결
 - 각각의 노드는 순차접근을 할 수 있도록 링크되어 순차접근이 용이

XIX. DBMS - HASHING

<https://www.db-book.com/db4/slide-dir/ch12.pdf>

데이터베이스 구조의 거대화로 인하여, 그것의 모든 계층을 거쳐서 모든 색인 값을 탐색한 다음, 원하는 데이터를 검색하여 목적지 데이터 블록에 도달하는 것은 거의 불가능에 가깝다. 해싱은 디스크에서 색인구조를 사용하지 않고 데이터 레코드의 직접적인 위치를 계산하는 효과적인 검색기법이다.

<해싱의정의>

해싱이란 데이터의 신속한 탐색을 위해 데이터를 해시테이블이라는 배열에 저장하고, 데이터의 키값을 주면 이를 적절한 해시함수를 통하여 테이블의 주소로 변환하여 원하는 데이터를 찾는 방법이다.

<Key-to-address>

특정규칙에 따라 주어진 키값을 주소로 변환하여 해시 테이블이라는 메모리 공간에 키의 레코드를 저장한다. 또한 해시함수를 이용하여 필요한 레코드의 주소를 산출하여 검색작업을 수행한다.

해싱에서는 데이터 레코드의 어드레스를 생산하기 위한 파라미터로서 탐색키와 더불어 해시함수를 사용한다. 해시(Hash)함수는 평문을 해시 알고리즘을 통하여 해시값으로 출력시켜준다. 이 함수는 어떤 길이의 데이터를 입력해도 정해진 길이의 결과를 제공해 주는 함수 이다.

해시함수(hash function)란 데이터의 효율적 관리를 목적으로 임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수이다. 이 때 매핑 전 원래 데이터의 값을 키(key), 매핑 후 데이터의 값을 해시값(hash value), 매핑하는 과정을 해싱(hashing)라고 한다.

특정한 키를 사용하지 않기 때문에, 동일한 문자들은 동일한 해시값으로 얻게 되며, 문자가 조금만 변해도 해시값이 크게 달라지게 된다. 해시 함수의 종류에 따라 출력되는 해시값의 길이가 달라진다.

예를 들어, '안녕'의 해시값은 '69F0BA16B5ACE257553E2FD23F74D6B4' 이나, '안 녕'의 해시값은 '48ADE1F88E07F872CDBA5073A9A557C9' 으로, 약간의 변화에도 완전 다른 해시값이 출력된다.

그렇기 때문에 해시값은 주로 데이터의 위변조여부를 확인하는데 사용되며, 또한 해시함수는 평문을 해시 알고리즘을 통하여 해시값으로 출력할 수는 있지만, 해시값을 다시 평문으로 되돌릴 수는 없다.

해시함수의 종류는 MD5, SHA-1, SHA 224 등이 있으며, 국내에서 개발된 HAS-160도 있다. MD5는 이미 해시충돌이 발견되어 SHA-1 이상의 해시함수 사용이 권장되고 있다.

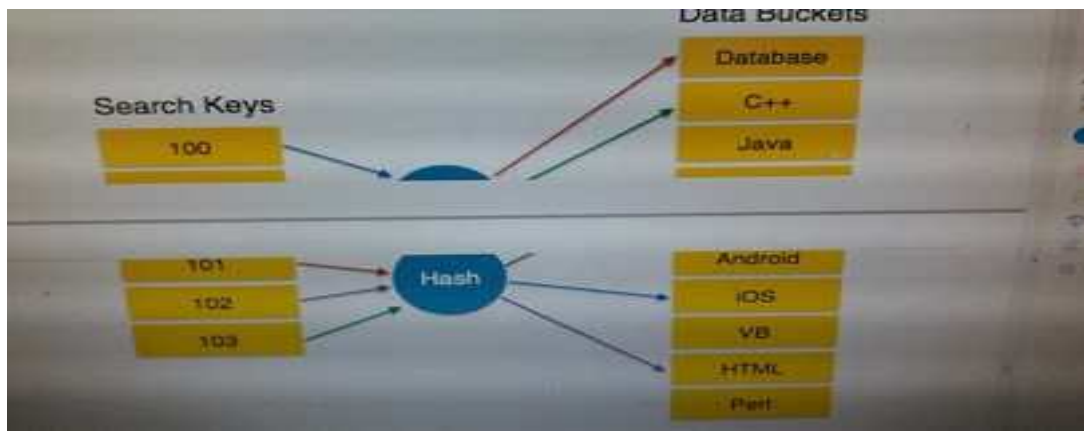
1. Hash Organization

■ **Bucket** - 해시 파일은 버킷 포맷으로 데이터를 저장한다. 버킷이란 저장 단위이다. 전형적으로 하나의 버킷은 한 개의 완전한 디스크 블록을 저장한 다음, 추가로 다른 레코드를 저장할 수 있다.

■ **Hash Function** - 해시 함수 h 는 모든 탐색키를 실재적인 데이터가 자리잡고 있는 어드레스에 연결시키는 매핑 함수 이다. 이 함수는 탐색키에서부터 버킷 어드레스까지를 다룬다.

2. Static Hashing

정적 해싱에서는 탐색키의 값이 제공될 때 그것의 해시 함수는 항상 동일한 어드레스를 산정한다. 예를 들어, 만일 mod-4 해시 함수가 사용되었다면, 이것은 단지 5개의 값만을 생산할 것이다. 그 결과 어드레스도 항상 그 함수에 맞춰 동일하게 나타난다. 그리고 제공된 버킷의 번호는 항상 변하지 않고 남아 있다.



■ Operation

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

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

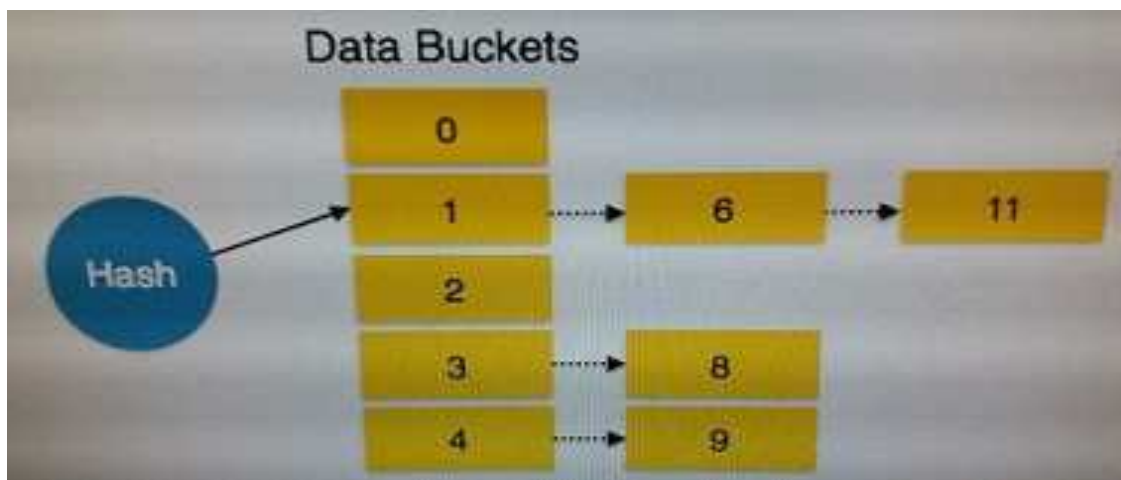
□ Search - 레코드를 검색하려할 때, 동일한 해시 함수를 사용하여 데이터가 저장된 버킷의 어드레스를 검색한다.

□ Delete - 이것은 삭제 작업에 의해 수반되는 간단한 탐색이다.

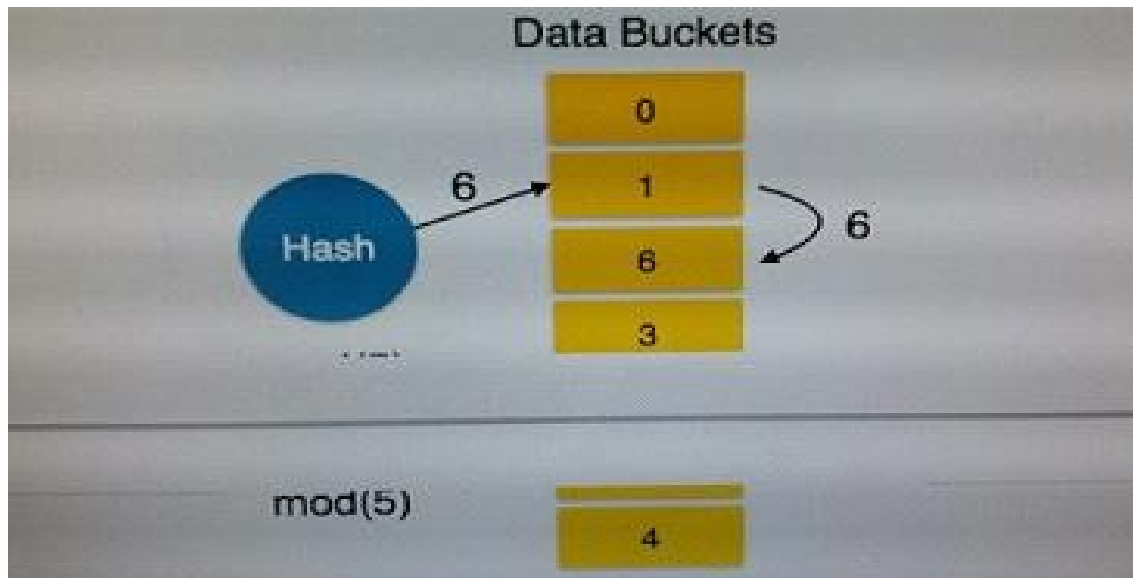
■ Bucket Overflow

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

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



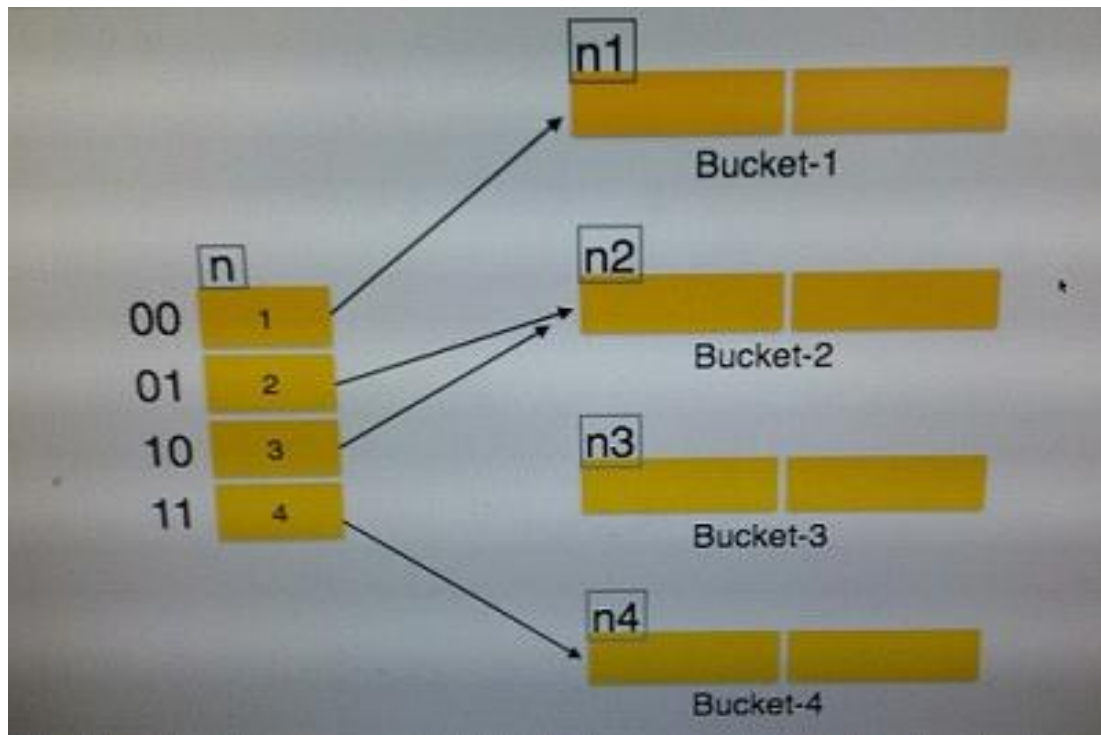
□ Linear Probing - 해시 함수가 데이터가 이미 저장된 어드레스를 생산할 때, 그 다음에 있는 자유로운 버킷이 그것에 할당된다. 이러한 메카니즘을 Open Hashing이라 부른다.



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, 해시 함수는 많은 수의 값을 생산하는데 사용되지만, 단지 소수만을 처음엔 사용할 수 있다.



■ Organization

모든 해시 값의 접두어(prefix)는 해시 색인으로 그 역할을 한다. 해시 값의 portion(한 조각)만이 컴퓨터 버킷 어드레스용으로 사용된다. 각각의 모든 해시 색인은 많은 비트가 해시 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 찼을 때, depth value가 선형적으로 증가하며 2배(twice)의 버킷들이 할당된다.

■ Operation

- Querying - 해시 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 찼다면,
 - △ 더 많은 버킷을 추가한다.
 - △ 해시 값에 additional bits를 추가한다.
 - △ 해시 함수를 재-계산한다.

- ▲ 그렇지 않다면,
 - △ 버킷에 데이터를 추가한다.

- ▲ 모든 버킷이 가득 찼다면, 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 해시 기능(operations)은 항구적인 것이다(in constant time).

XX. DBMS - DEADLOCK

다중-처리 시스템에서, 교착상태는 자원을 공유하는 환경에서 원치 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 X가 필요하다. 자원 X는 T_1 에 의해 유지되고(held) 있으며, T_1 은 자원 Y를 기다리고 있다. T_2 는 T_0 에 의해 유지되고 있는 자원 Z를 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 되돌아가기) 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 발생하게 될 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 스키마는 보다 오래된 트랜잭션으로 하여금 기다리도록 하지만 보다 최신의 것을 없애지는 않는다.

2) Wound-Wait Scheme

이 스키마에서, 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.

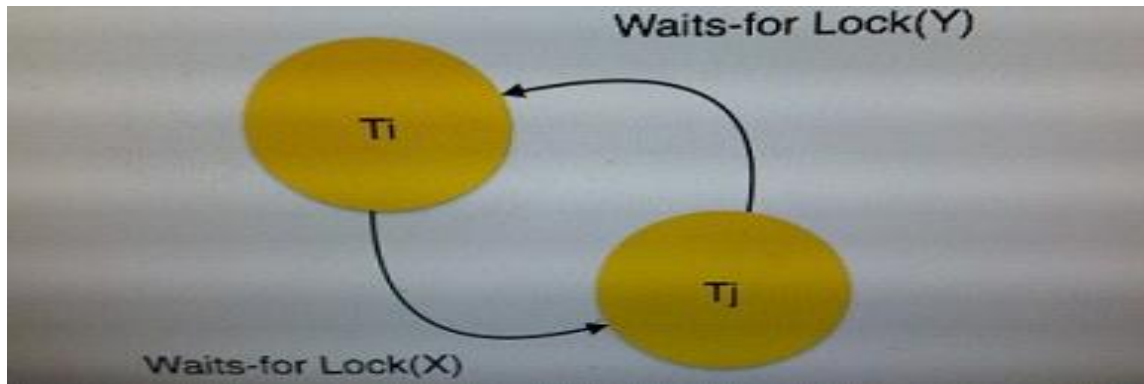
이 스키마는 최신의 트랜잭션이 기다릴 것을 허용하고 있지만, 오래된 트랜잭션이 보다 최신의 것에 의해 유지되는 아이템을 요청할 때, 보다 오래된 트랜잭션은 보다 최신의 것이 강제로 포기하도록 한 다음 그 아이템을 해제한다. 이러한 양쪽 모두의 경우에, 보다 나중의 단계에 시스템에 입력되는 트랜잭션은 중단된다(aborted).

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 X인 아이템에 대해 록을 요청할 때, 하나의 직선(directed edge)이 T_i 에서 T_j 까지 만들어진다. 만일 T_j 가 아이템 X를 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.

XXI. 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 넘겨

야(dumped) 한다. 이 저장기기에는 또한 logs, active transactions, buffer blocks이 포함되기도 한다.

- <dump>는 데이터베이스 콘텐츠가 비-휘발성 메모리로부터 안전한 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

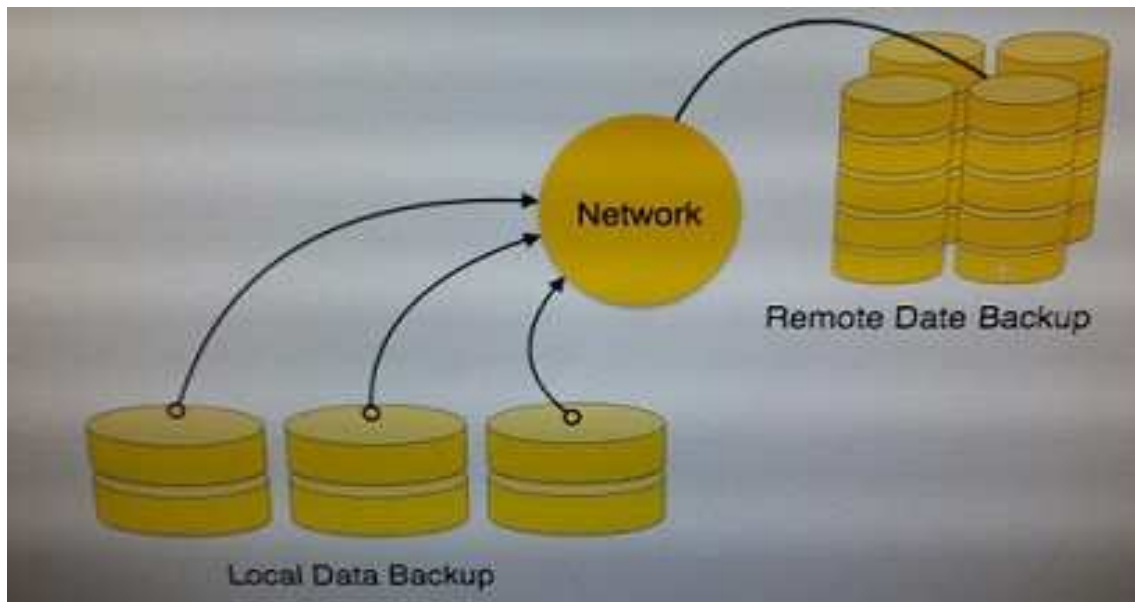
재앙적 실패는 안정되고 부차적인 저장기기가 훼손(corrupt) 되었을 때 발생한다. 저장 기기와 함께, 그 안에 저장된 모든 가치있는 데이터가 사라진다. 우리는 이러한 재앙적 실패로부터 데이터를 복구할 수 있는 두 가지 서로 다른 전략을 가지고 있다:

- Remote backup & minu - 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 투자자에겐 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.

XXII. DBMS - DATA RECOVERY

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 중지되

어야 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 저장 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:

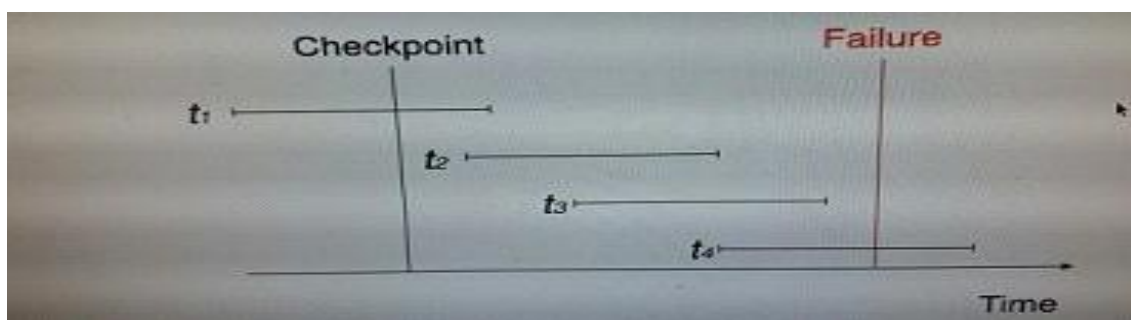
- Deferred database modification - 모든 로그들은 안정된 저장매체에 작성된 다음에, 데이터베이스가 트랜잭션이 완료되면 갱신된다.
- Immediate database modification - 각 로그들이 실제적인 데이터베이스 변경 이후에 작성된다. 즉, 데이터베이스는 모든 operation이 끝난 다음에 변경된다.

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 실제의 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 <Tn, Start>와 <Tn, Commit> 또는 단지 <Tn, Commit>로 된 로그를 본다면, 그 트랜잭션을 redo-list에서 갖다 놓는다.
- 만일 복구 시스템이 <Tn, Start>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!!