데이터베이스(이상호)

Lee Jun Hyeok (wnsx0000@gmail.com)

June 11, 2025

목차

1. DB

1.1. DB

1.1.1. DB

1. DB
DB(Database)는 상호 관련이 있고, 컴퓨터에 저장되며, main memory에 전부 저장되기에 규모가 커이차 저장공간(HDD나 SSD)을 필요로 하는 data의 집합임. DB에서는 대용량 data 관리를 위한 여러 기법을 활용함.

이때 data는 비교적 엄격한 스키마(또는 구조체) 등에 의해 관리되는 Structured Data와, 스키마가 존재하지 않거나 느슨한 Unstructured Data로 나뉘는데, 본 수업에서의 DB는 이 둘 모두를 다루는 시스템이라고 함.

$$a_n = \sum_{i=1}^{a}(a_i^2 + 14)$$

2. DBMS
DBMS(DB Management System)은 DB가 저장된 컴퓨터에서 이를 관리하는 소프트웨어임. DBMS는 DB에 접근 및 조작하는 기능 등을 제공함.

개념적으로 DBMS는 query processor와 storage manager로 구성됨. Query Processor는 query 처리, authorization, authentication 등을 처리하는 부분이고, Storage Manager는 데이터 저장, 검색, transaction 관리 등을 수행하는 시스템의 하단 부분임.

DBMS의 사용에 따른 이점은 아래와 같음. 이는 단순히 file system을 활용한 방법과 비교하여 생각할 수 있음.

1) data abstraction.
2) easy accessing data. DB language 등을 활용한 사용자 인터페이스를 제공함.
3) redundancy/inconsistency control.
4) IC 유지. IC(Integrity Constraint, 데이터 무결성 제약조건)는 데이터가 만족해야 하는 조건임.
5) atomicity of update.
6) concurrency(동시성) control.
7) data security.
8) data backup.

참고로, 이전에 학습한 자료구조(Data Structure)는 main memory만을 활용하는 작은 규모의 data 처리를 다루는 기술임.

본 수업에서는 시스템 외부에서 보는 DB에 대해서 다루고, 이후 데이터베이스2 수업에서는 시스템 내부에서 보는 DB에 대해서 다룸. 즉, 이 수업에서는 사용자 관점에서의 DB 사용법을 배움.

1.1.2. Abstraction And Schema

1. Abstraction
전산학에서 Abstraction(추상화)는 어떤 대상을 나타내는 instance로부터 중요 개념을 분리하는 것임. 이때 어떤 level(수준)에서 abstraction하냐에 따라 활용될 수 있는 범위가 달라짐.

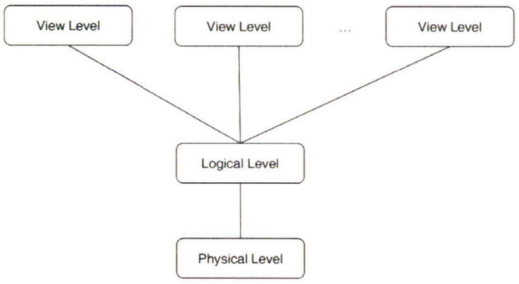
DB에 대한 abstraction은 아래와 같이 3가지 level로 나뉨. 각 level에서 이루어지는 abstraction은 DB에 따라 달라질 수 있음.

1) Physical Level : 실제 데이터가 물리적으로 어떻게 저장되는지 나타낸 것.

2) Logical Level : 데이터 사이의 관계를 논리적으로 나타낸 것.

3) View Level : logical level의 데이터 중 특정 사용자에게 필요한 데이터만을 나타낸 것. 즉, 동일한 logical level의 abstraction에 대해 사용자에 따라 여러 view level abstraction이 존재할 수 있음. DB

사용자는 view schema를 활용하여 응용 프로그램을 개발하게 됨.

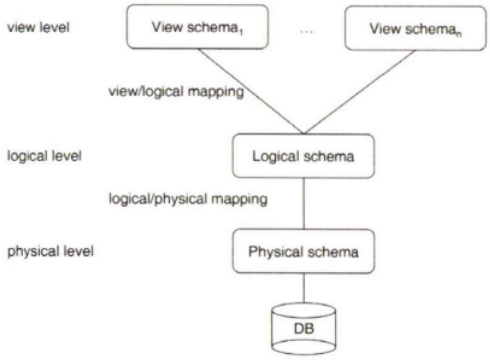


2. Schema

Schema(스키마)는 DB의 논리적 또는 물리적 구조를 나타내는 설계도 또는 표현 방식이고, Instance(인스턴스)는 DB에서 schema에 따라 저장되는 실제 값(데이터)임.

schema는 DB의 종류에 따라 다르게 정의될 수 있음. 당연하게도 DB를 활용하며 instance는 쉽게 수정/변경되지만, schema는 한 번 정해지면 잘 바뀌지 않음.

DB에 대해 3 level abstraction이 결정되면, 이에 따라 3단계 schema 구조(Three Level Schema Architecture)가 생성됨. 즉, abstraction 정도에 따른 설계도가 존재하는 것. 또한 schema 사이의 mapping이 존재함.



상용 DBS는 기본적으로 이런 3 level의 schema를 지원한다고 함.

3. Data Independence

3단계의 schema 구조를 활용하면 data에 대한 independence(독립성)를 제공할 수 있음. data independence는 physical independence와 logical independence로 구분됨. Physical Independence는 logical schema의 변경 없이 physical schema를 변경할 수 있음을 의미하고, Logical Independence은 view schema 변경 없이 logical schema를 변경할 수 있음을 의미함.

대부분의 상용 DBS는 data independence를 잘 지원한다고 함.

1.1.3. Data Model

1. Data Model

Data Model은 데이터, 데이터 간 관계, 데이터의 의미, 데이터 제약 등을 정의하는 specification(명세)임. 앞에 정리한 schema는 data model에 대한 표현 방식으로 이해할 수 있음.

아래와 같이 여러 종류의 data model들이 있음. data model이 달라지면 같은 데이터여도 DBSM이 완전히 달라진다고 함.

1) Relational Data Model

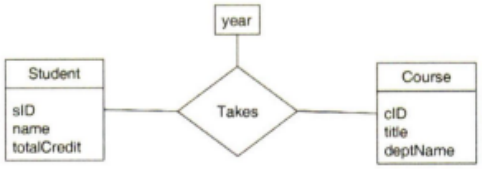
Relational Data Model(관계형 데이터 모델)은 가장 많이 사용되는 data model로, DB를 여러 대상과 그 사이의 관계로 표현함.

2) Object Relational Data Model

Object Relational Data Model(객체 관계형 데이터 모델)은 relational data model에 객체지향적 요소를 추가한 것임. 이 경우 대체로 relational data model도 지원하는 경우가 많다고 함.

3) XML Data Model
XML Data Model은 XML을 활용해 데이터를 표현하는 data model임. XML은 현재 데이터 전송을 위한 표준 기술로 사용되며, 여러 DBS에서 이를 지원한다고 함.

4) Entity Relationship Data Model
Entity Relationship Data Model(ER Model)은 데이터를 entity(object, 객체)와 relationship(관계성)으로 표현하는 data model임. 비교적 추상적인 차원의 model임.



2. Data Model의 역사
DB의 사용 이전에는 file system을 활용했고, 그 이후 data model은 network/계층 data model, relational data model, object data model, object relational data model, XML data model 등으로 발전해왔음.

3. DB Design
DB Design(설계)은 요구사항에 따른 최적의 schema를 찾는 과정으로, logcial design와 physical design으로 나뉨. Logical Design은 logical/view schema를 설계하는 것이고, Physcial Design은 physical schema를 설계하는 것임.

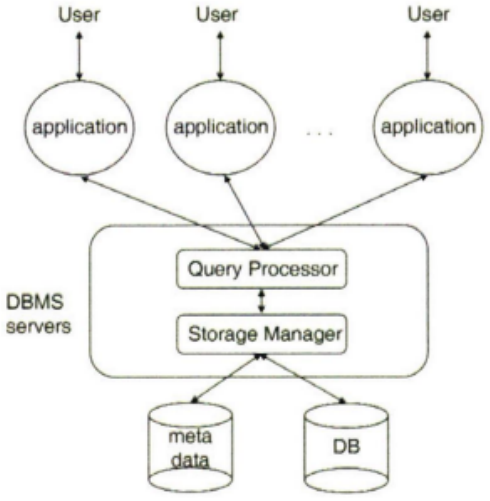
data model을 활용하여 DB design을 수행할 수 있음. 주로 ER model을 사용해 design 한다고 함.

1.1.4. DBS

DB와 DBMS를 통틀어 DBS(Database System)이라고 함. 실제로는 DB, DBMS, DBS 각 용어가 interchangeable하게 사용된다고 함.

DBS는 os의 file system으로 구현되는 경우가 일반적이고, 이 경우 DBS는 하나의 사용자 process임.

DBS는 단순히 아래와 같은 구조를 가짐.



또한 DBS는 데이터의 저장 위치에 따라 centralized, distributed, client-server 등으로 구분할 수 있음.

DBS에서 지원하는 일반적인 기능 및 서비스들은 아래와 같음.

1. DB Language

DBMS는 사용자들이 해당 시스템을 활용할 수 있도록 SQL, 관계대수 등의 DB Language를 지원함. 가장 주요하게 사용되는 것은 SQL으로, 사용자는 이를 활용해 application을 개발함.

2. Data Dictionary
Data Dictionary는 schema, 제약, authorization, 통계적 정보 등 DB에 대한 metadata를 저장하는 부분임.

3. Transaction Management
Transaction(트랙잭션)은 DB에서 특정 기능을 수행하는 operation의 집합임. DBS는 이를 잘 관리해야 함.

transaction management는 concurrency control manger와 recovery manger에 의해 수행됨. Concurrency Contorl Manager는 여러 transaction 사이의 concurrency(동시성) 문제를 처리하여 DB의 consistency(일관성)를 유지하는 부분이고, Recovery Manager는 transaction에서 작업 실패가 발생하는 경우 이를 처리하여 consistency를 유지하는 부분임.

4. DB Users
DB user(사용자)는 아래와 같이 몇 가지로 분류할 수 있음.

1) Naive User : DB를 활용하는 application을 통해 DB에 접근하는 사용자.
2) Application Progrmmer : DB를 활용하는 application을 개발하는 사용자.
3) DBA(Database Administrator) : DBS의 모든 권한을 가지고 있는 관리자. schema 정의 및 수정, authorization, 제약 등을 관리함.

참고로 meta-는 한 단계 위라는 의미임. 즉, metadata는 data의 그 윗 단계의 data로, 데이터에 대한 데이터를 의미함.

1.2. Relational Data Model

1.2.1. Relational Data Model

1. Relational Data Model
Relational Data Model(관계형 데이터 모델)은 DB를 relation와 integrity constraint의 묶음으로 표현하는 data model임. 개념적으로 단순해서 현재 사용 DBS에서 가장 많이 사용되는 data model이라고 함.

relational data model에서 Relation(관계)은 tuple의 집합(set. 중복/순서 없음.)으로, 아래와 같이 데이터에 대한 표로 표현됨. 이때의 각 행을 Tuple(터플), 열을 Attribute(속성)라고 함. relation, tuple, attribute는 더 일반적인 용어로 table, record(row), column이라고도 함. 또한 relation이 가진 attribute의 개수를 Arity라고 함.

Intergrity Constraint는 데이터의 무결성(정확성, 신뢰성, 일관성)을 위한 제약임.

sID	name	gender	deptName	year	GPA	totalCredits
152	Eric Lee	M	CS	freshman	3.54	15
201	Sam Kim	M	EE	senior	3.89	103
301	John Park	M	Media	sophomore	4.13	38
157	Alma Lee	F	CS	freshman	2.57	18
154	Joan Lee	F	CS	sophomore	1.80	58

Student relation

2. Domain
각 attribute는 domain을 가짐. Domain은 attribute의 값으로 허용될 수 있는 값의 집합임. 이때 각 domain은 값이 없음을 나타내는 null을 default로 포함하고 있고, attribute 값이 입력되지 않은 부분은 null을 값으로 가짐.

이때 domain에 속하는 값은 atomic해야 함. Atomic하다는 것은 해당 값이 더 이상 분해할 수 없다는 것임. 예를 들어, 정수/실수/문자열/시간 등은 atomic하고, set/bag/list 등 여러 데이터를 포함하는 값은 atomic하지 않음.

물론 DBS에서 모든 domain에 대한 정의가 항상 완전하게 이루어지는 것은 아님. attribute가 가지는

실제 값과 integrity constraint로 이를 유추하게 될 수 있음.

3. Relational Schema
Relational Schema는 아래와 같이 relation의 이름과 해당하는 attribute의 이름을 나열한 것임. 이는 diagram으로 쉽게 나타낼 수 있음.

$$R = (A_1, A_2, \cdots, A_n)$$

이렇게 정의된 schema에 대해 domain에 부합하는 값의 조합이 존재할 수 있는데, 이를 Relational Instance라고 함. 예를 들어, 위의 표에서 relational instance는 5개임. relational instance는 domain 값을 활용한 모든 조합에 대한 부분집합인데, 수학에서는 이러한 부분집합을 relation이라고 정의하고, 이에 따라 이런 방식이 relational data model이라고 불리게 되었다고 함.

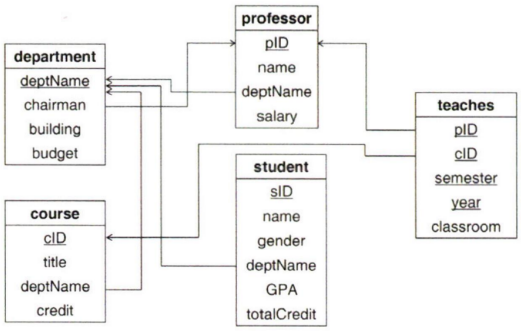
또한 relational schema는 integrity constraint도 포함함.

참고로 set/bag/list는 모두 군집 데이터를 정의하는 값임. set은 중복이 없고 순서가 존재하지 않는 데이터 묶음이고, bag은 중복이 가능하고 순서가 존재하지 않는 데이터 묶음이고, list는 중복이 가능하고 순서가 존재하는 데이터 묶음임.

1.2.2. Relational DB

1. Relational DB
Relational DBS에서 Relational DB는 relation과 integrity constraint의 집합으로 구성되는 것으로 정의함. relation을 아래와 같이 나타낼 수 있음.

relational DB에서는 relation과 tuple 모두에 대해서 순서나 상하관계가 존재하지 않음. 이에 따라 데이터 접근이 자유롭고 용이하다고 함.



2. Key
relational DB에서 Key(키)는 하나 이상의 attribute의 집합으로, relation 내부의 tuple을 식별하거나 relation 사이의 관계 설정에 사용되는 요소임.

Super Key(슈퍼 키)는 relation에서 tuple을 유일하게 식별할 수 있도록 하는 attribute의 집합임($K \subseteq R$). 또한 Candidiate Key(후보 키)는 super key에서의 유일성을 유지하면서 minimality를 확보한 key임. 이때 Minimality는 key를 구성하는 attribute 중 하나라도 제거하면 유일성을 잃는 상태를 말함. 이에 따라 서로 다른 candidiate key는 서로 다른 개수의 attribute를 가질 수 있음. 즉, minimaltiy는 minimal(최소한)이지, minimum(최소)이 아님.

relational DB에서 실제로 tuple 식별에 사용되는 key를 Primary Key(주 키)라고 함. 이는 위의 그림에서는 밑줄로 표기했음. 이때 여러 attribute를 primary key로 하는 경우의 key는 해당 attribute들의 단순 연결(concatenation)이고, 연결한 전체 값은 겹칠 수 없어도 개별 값은 서로 겹칠 수 있음. DB 설계 시에 primary key는 candidtate key 중 하나를 선택해 지정함.

relational DB에서 다른 relation의 primary key를 참조하여 두 relation 사이의 관계를 설정하는 attribute를 Foreign Key(외래 키)라고 함. 이때 foreign key는 서로 다른 relation에 대해 어떤 특정한 tuple 사이의 관계를 정의하므로 primary key를 참조해야 함. 만약 relation(A)의 attribute가 다른 relation(B)의 primary key 값을 가지고 있는 경우, A는 B를 참조하는 relation이라고 함.

3. Integrity Constraint
relational DB에서의 integrity constraint에는 구체적으로 아래와 같은 것들이 있음.

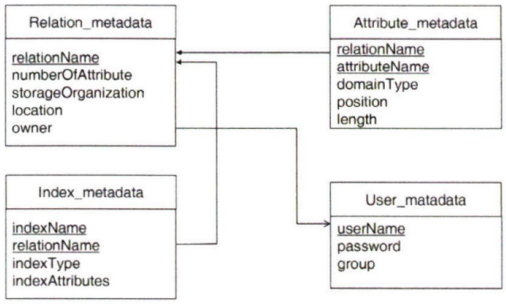
1) Domain Intergrity Constraint : 각 attribute는 정의된 domain 내의 값만을 가져야 함. 즉, 값의 자료형, 범위, 형식 등에 대한 제약임.

2) Entity Intergrity Constraint : relation 내에서 tuple이 유일하게 구별될 수 있어야 함. 즉, primary key는 null 값을 가질 수 없고, 고유해야 함.

3) Referential Intergrity Constraint : relation 사이의 관계(연결)이 항상 유효해야 함. 즉, foreign key가 참조하는 table에 존재하는 값과 일치하거나 null이어야 함.

4. Data Dictionary
Data Dictionary(데이터 사전) 또는 System Catalog는 DBS가 metadata를 관리하는 데이터 공간으로, relation, attribute, integrity constraint, 사용자 정보, 통계, physical information 등을 포함함.

data dictionary에서 metadata는 아래와 같이 table(view) 형태로 관리됨.



상용 DBS는 사용자에게 SQL을 활용한 data dictionary로의 접근(주로 read-only)을 제공함.

1.2.3. Relational Algebra : 기본 연산

Relational Algebra(관계 대수)는 relation, constraint 등에 대한 연산으로, relational data model의 일부임. relational algebra에는 기본적으로 입력/삭제/수정/검색 연산이 있는데, 여기에서는 검색 연산만 다룸.

relational algebra에는 아래와 같이 6개의 기본 연산이 존재함. 각 연산은 입력으로 하나 또는 두 개의 relation을 받아 새로운 relation을 출력하므로, 각 연산은 중첩/합성(Composite)하여 수행될 수 있음.

relational algebra에는 표준이 존재하지 않음(여기에서의 설명 또한 비표준임.). 실제로 상용 DB에서도 이를 개념적으로만 지원하고, 사용자는 relational algebra가 아닌 SQL을 사용하여 DB에 접근함.

1. Select
Select(선택) 연산은 입력 relation에 대해 주어진 조건을 만족하는 tuple만을 추출해 구성한 relation을 출력함.

$$\sigma_p(r) = \{t | t \in r \ and \ p(t)\}$$

이때 p는 Propositional Logic(명제 논리)를 표현하는 Selection Predicate(선택 조건)로, term 또는 term의 연결로 구성됨. Term(항)은 '<attribute> op <attribute>' 또는 '<attribute> op <constant>' 형태이며, op는 =, ≠, <, >, ≥, ≤로 총 6가지임. 각 term은 ∧(and), ∨(or), -(not)으로 연결될 수 있음. 참고로, propositional logic에는 '모든'이 없음.

Relation r

A	B	C	D
α	α	1	5
α	α	5	2
β	β	12	12
α	β	23	10

$\sigma_{A=B \wedge D > 4}(r)$

A	B	C	D
α	α	1	5
β	β	12	12

2. Project

Project(투영) 연산은 입력 relation에 대해 지정한 attribute만을 추출해 구성한 relation을 출력함.

$$\Pi_{A_1, A_2, \cdots, A_n}(r)$$

relation은 tuple에 대한 set이므로, 연산 이후 중복되는 tuple은 제거됨.

Relation r

A	B	C	D
α	α	1	5
α	α	5	2
β	β	12	12
α	β	23	10

$\Pi_{A,C}(r)$

A	C
α	1
α	5
β	12
α	23

$\Pi_{A,B}(r)$

A	B
α	α
β	β
α	β

3. Union

Union(합집합) 연산은 두 relation에 대해 tuple을 집합 원소로 취급하여 합집합을 수행하고, 이를 통해 구성한 relation을 출력함.

$$r \cup s = \{t | t \in r \ \ or \ t \in s\}$$

이때 두 relation이 가진 attribute의 개수가 같아야 하고, 대응되는 attribute의 domain이 호환되어야 함. 이는 intersection, difference에서도 동일함.

Relation r

A	B
α	1
α	5
α	23

Relation s

A	B
α	1
β	12

r ∪ s

A	B
α	1
α	5
α	23
β	12

4. Difference

Difference(차집합) 연산은 두 relation에 대해 tuple을 집합 원소로 취급하여 차집합을 수행하고, 이를 통해 구성한 relation을 출력함.

$$r - s = \{t | t \in r \ or \ t \notin s\}$$

앞에서 정리한 것처럼, 이때 두 relation이 가진 attribute의 개수가 같아야 하고, 대응되는 attribute의 domain이 호환되어야 함.

Relation r

A	B
α	1
α	5
α	23

Relation s

A	B
α	1
β	12

r - s

A	B
α	5
α	23

s - r

A	B
β	12

5. Cartesian Product

Cartesian Product(카타시안 곱) 연산은 두 relation에 대해 tuple을 하나씩 가져와 연결(concatenation)하여 만들 수 있는 모든 가능한 조합으로 tuple을 구성한 relation을 출력함.

$$r \times s = \{t \ q | t \in r \ or \ q \in s\}$$

이때 두 relation은 attribute가 서로 disjoint($R \cup S = \emptyset$)해야 함. 동일한 attribute가 존재하면 cartesian product가 아니라 natural join이 됨.

cartesian product 이후 arity는 두 relation의 arity를 더한 것이 되고, tuple의 개수는 두 relation의 arity를 곱한 것이 됨.

Relation r

A	B
α	1
α	5
α	23

Relation s

C	D
α	1
β	12

r x s

A	B	C	D
α	1	α	1
α	1	β	12
α	5	α	1
α	5	β	12
α	23	α	1
α	23	β	12

6. Rename

Rename(재명명) 연산은 해당 relation(E)에 대해 relation 이름(X) 또는 attribute 이름($A_1, A_2, \cdots, A_n$)을 새로 지정한 relation을 출력함.

$$\rho_{X(A_1, A_2, \cdots, A_n)}(E)$$

relation 이름만을 변경하는 경우 attribute 이름은 생략할 수 있고, attribute 이름만을 변경하는 경우에는 relation 이름을 명시해야 함.

constant 중 정수나 실수 등은 단순 작성하면 되는데, 물론 여기에서는 표준이 없지만 자료형이 문자열이면 대체로 ""로 묶는다고 함.

문제 예시1) Retrieve course titles that are offered by both "CS" and "EE" department.
-> 이 경우 select를 $\sigma_{deptname="CS" \wedge deptname="EE"}(course)$와 같이 사용하는 실수를 할 수 있음. 그럴듯해 보이지만 이를 만족하는 tuple은 하나의 attribute가 두 개의 값을 가져야 하므로 존재할 수 없음. 대신 각 deparment에 대한 값을 추출한 뒤에 교집합해야 함.

문제 예시2) Retrieve pairs of department names where the budget of the first department is greater thant that of the second department.
-> cartesian product로 모든 pair를 생성한 뒤, select로 비교하여 적절한 것을 추출하면 됨. 이때 cartesian product에서는 attribute가 겹치면 안 되므로 rename하여 연산해야 함.

1.2.4. Relational Algebra : 추가 연산

relational algebra의 추가 연산들에 대해 알아보자. 이 연산들은 모두 기본 연산으로 대체가 가능하므로 algebra의 expressive power에 변화를 주진 않고, 단순히 편리함을 위한 것들임.

아래에서 설명하는 Join(조인)은 두 개 이상의 relation을 특정 조건에 따라 합치는 연산임. join은 크게 inner join과 outer join으로 나뉨. Inner Join(내부 조인)은 join 조건에 맞는 tuple만 활용하여 relation을 출력하는 방식으로, theta/equi/natural join 등이 있음. outer join은 아래에서 설명함.

1. Assignment
Assignment(할당) 연산은 relation을 할당하여 복잡한 query를 작성할 때 중간 결과를 저장할 수 있음.

$$A \leftarrow relation$$

또한 rename 연산과 함께 사용하여 이름을 지정하며 저장할 수 있음.

2. Interaction
Interaction(교집합) 연산은 두 relation에 대해 tuple을 집합 원소로 취급하여 교집합을 수행하고, 이를 통해 구성한 relation을 출력함. 이는 아래의 수식과 같이 두 번의 difference 연산으로 대체될 수 있으므로 추가 연산임.

$$r \cap s = \{t | t \in r \ and \ t \in s\} = r - (r - s)$$

앞에서 정리한 것처럼, 이때 두 relation이 가진 attribute의 개수가 같아야 하고, 대응되는 attribute의 domain이 호환되어야 함.

Relation r

A	B
α	1
α	5
α	23

Relation s

A	B
α	1
β	12

r ∩ s

A	B
α	1

3. Theta/Equi Join
Theta Join(세타 조인)은 두 relation에 대해 cartesian product를 수행한 뒤, 지정한 조건으로 select하는 join임. 이에 따라 이름이 겹치는 attribute는 결과 relation에서 서로 다른 attribute로 등장하게 되는데, 여기에서는 relation의 이름을 앞에 붙이는 것으로 두 attribute를 구분했음. 이는 가장 일반적인 형태의 join임.

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

Equi-Join(동등 조인)은 join 조건 모두가 동등 조건(=)으로만 이루어진 join임. theta join과 마찬가지로 이름이 겹치는 attribute는 relation에서 서로 다른 attribute로 등장하게 되고, 이를 개선하여 project 연산을 활용해 이런 attribute는 하나로 합친 것이 natural join임.

myProfessor

pID	name	deptName
10101	Sam	Computer
12121	Wu	Finance
15151	Mozart	Music

myTeaches

pID	courseID
10101	CS-101
12121	FIN-201
86868	BIO-101

myProfessor $\bowtie_{myProfessor.pID>myTeaches.pID}$ myTeaches

myProfessor.pID	name	deptName	myTeaches.pID	courseID
15151	Mozart	Music	10101	CS-101
15151	Mozart	Music	12121	FIN-201
12121	Wu	Finance	10101	CS-101

myProfessor $\bowtie_{myProfessor.pID=myTeaches.pID}$ myTeaches

myProfessor.pID	name	deptName	myTeaches.pID	courseID
10101	Sam	Computer	10101	CS-101
12121	Wu	Finance	12121	FIN-201

4. Natural Join

Natural Join(자연 조인)은 equi join의 결과에서 join attribute를 하나의 attribute로 합친 join임. 구체적으로는, 동일한 이름의 attribute를 join attribute로 지정하고, 두 relation에 대해 join attribute에 동일한 값을 가지는지를 join 조건으로 함.

- R = (A, B, C, D) S = (B, D, E)
- $r \bowtie s : \prod_{r.A,\ r.B,\ r.C,\ r.D,\ s.E}(\sigma_{r.B = s.B \wedge r.D = s.D}(r \times s))$

Relation r

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

Relation s

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	δ

r⋈s

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

위의 수식에서도 알 수 있듯이 별도의 join 조건을 지정하지 않고(정해져 있음.), 결과 relation에서 join attribute가 한 번씩만 나옴.

당연하게도 natural join에 대해서는 아래와 같이 associative(결합성)와 commutative(교환성)이 성립함.

$$((A \bowtie B) \bowtie C) = (A \bowtie (B \bowtie C))$$

$$(A \bowtie B) = (B \bowtie A)$$

natural join은 join 중 가장 흔하게 사용되는 연산이라고 함. 두 relation을 합치는 경우 말 그대로 자연스럽게 사용이 가능함.

5. Outer Join

Outer Join(외부 조인)은 join 조건을 맞지 않는 tuple도 활용하여 relation을 출력하는 join임. 즉, join을 수행한 결과에 join 조건에 맞지 않는 tuple들을 포함시키는데, 이에 따라 정보 손실을 막을 수 있음.

join 조건에 맞지 않는 tuple을 포함시키는 방식에 따라 아래와 같이 3가지로 나눌 수 있음. tuple을 포함시킬 때 값이 존재하지 않는 attribute에는 null을 넣음.

1) Natural Left Outer Join : 연산 작성 시에 왼쪽 relation의 모든 tuple은 결과에 포함시킴.
2) Natural Right Outer Join : 연산 작성 시에 오른쪽 relation의 모든 tuple은 결과에 포함시킴.
3) Natural Full Outer Join : 연산 작성 시에 두 relation의 모든 tuple을 결과에 포함시킴.

myProfessor ⟖ myTeaches

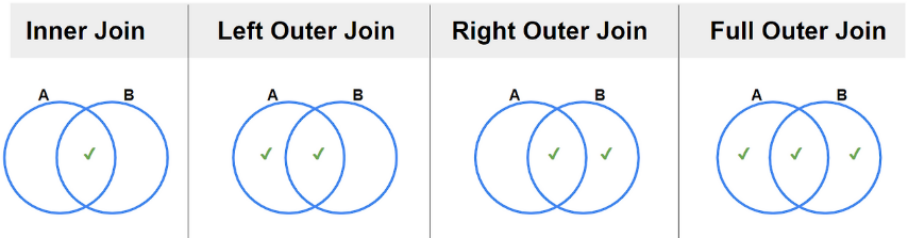
pID	name	deptName	courseID
10101	Sam	Computer	CS-101
12121	Wu	Finance	FIN-201
15151	Mozart	Music	null

myProfessor ⟕ myTeaches

pID	name	deptName	courseID
10101	Sam	Computer	CS-101
12121	Wu	Finance	FIN-201
86868	null	null	BIO-101

myProfessor ⟗ myTeaches

pID	name	deptName	courseID
10101	Sam	Computer	CS-101
12121	Wu	Finance	FIN-201
15151	Mozart	Music	null
86868	null	null	BIO-101



6. Division

Division(나눔)은 동일한 이름의 attribute를 가지는 두 relation에 대해서(순서대로 A, B), A의 tuple 중 B의 모든 tuple과의 조합이 존재하는 tuple만을 도출하는데, 이때 겹치는 attribute는 제외하고 출력함.

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

B의 모든 tuple과의 조합이 존재해야 하므로, 적절한 결과를 얻으려면 B의 attribute 집합은 A의 attribute 집합의 부분집합이어야 함.

예를 들어, division 연산은 $r(sID, cID) \div s(cID)$ 등으로 특정 과목을 모두 수강한 학생의 id를 추출하는 상황에서 사용할 수 있음.

A	B	C	D	E
α	1	α	a	1
α	1	γ	a	1
α	1	γ	b	1
β	1	γ	a	1
β	1	γ	b	3
γ	1	γ	a	1
γ	1	γ	b	1
γ	1	β	a	1

r

D	E
a	1
b	1

s

A	B	C
α	1	γ
γ	1	γ

r ÷ s

문제 예시1)

- Retrieve professor names who teach in the Fall semester of 2020 together with the course titles that the professors teach
- myCourse ← $\rho_{myCourse(cID,\ title,\ newDeptName,\ credit)}$(course)
- $\Pi_{name,\ title}(\sigma_{semester='Fall'\ \wedge\ year\ =\ '2020'}$(professor ⋈ teaches ⋈ myCourse))
- $\Pi_{name,\ title}((\sigma_{semester='Fall'\ \wedge\ year\ =\ '2020'}$(teaches)) ⋈ professor ⋈ myCourse)

-> 이 문제는 join을 활용하여 쉽게 해결할 수 있지만, natural join을 사용하여 여러 relation을 합치는 경우 의도치 않게 같은 이름의 attribute로 join될 수 있음. 여기에서도 deptName을 적절히 바꿔주지 않으면 잘못된 결과가 도출됨(deptName은 professor와 course의 소속을 나타내는 것이므로 항상 일치하지 않을 수 있음). **join 시에는 이름이 겹치는 attribute의 의미를 잘 살피고, join되면 안되는 경우 rename해줘야 함.**

2. SQL

2.1. SQL

2.1.1. DB Language

DB Language는 DB와 그 사용자 사이의 의사소통을 위한 언어임. 이는 기능적 관점과 표현 방식 관점을 사용해 분류할 수 있음.

1. 기능적 관점에서의 분류
기능적 관점에서 DB language는 DDL, DML, DCL로 구분할 수 있음.

1) DDL
DDL(Data Definition Language)는 DB schema의 조작을 담당하는 영역임. schema(relation과 integrity constraint)에 대한 생성/삭제/수정 등을 수행함. 해당 내용은 data dictionary에 저장되므로, DDL은 이를 조작하는 기능을 함.

2) DML
DML(Data Manipulation Language)는 instance의 조작을 담당하는 영역임. instance에 대한 생성/삭제/조회/수정 등을 수행함. 사용자는 이를 활용해 DB에 query를 전달하므로 이를 query language라고도 함.

3) DCL
DCL(Data Control Language)는 DBS의 schema와 instance를 제외한 다른 객체를 조작하는 영역임. transaction, session, recovery, authorization, user 관리 등을 수행함.

2. 데이터 표현 방식 관점에서의 분류
데이터 표현 방식 관점에서 DB language는 절차적 언어와 비절차적 언어로 분류할 수 있음. 절차적 언어(Procedure Language)는 데이터 처리에 대한 구체적인 방법과 절차를 명시하고, 비절차적 언어(Non-procedure/Declarative Language)는 방법이나 절차 없이 필요한 데이터만을 명시함.

비절차적 언어는 방법과 절차를 알아서 판단하여 처리하여야 하므로 그 구현이 어렵지만, 당연하게도 사용자 입장에서는 절차적 언어보다 편리함. SQL은 비절차적 언어임.

2.1.2. SQL

1. SQL
SQL(Structured Query Language)은 relational data model을 위한 DB language임. 이는 DDL, DML, DCL을 모두 포함하며, ISO에 의한 표준이 사용되고 있음. 이 필기에서는 ISO 표준인 SQL2를 기반으로 설명함.

여기에서 Query는 DB에 대한 검색이나 조작 등을 수행하기 위해 작성하는 명령문임.

relational data model을 위한 언어로는 relational algebra와 같은 pure relational DB language도 존재하지만, 상용 시스템에서 구현되어 있고 가장 많이 사용되는 언어는 SQL임.

기본적으로 SQL에서는 대소문자 구분이 존재하지 않고, 따옴표 안에 작성하는 문자열에 대해서는 구분이 존재함. 따옴표는 기본적으로 작은따옴표를 쓴다고 함.

여기에서 설명하는 SQL의 각 문장은 ;로 끝남.

SQL에서 ()는 연산 우선순위 지정, 서브 query 정의, 함수 호출 등이나 특정 문법에 필수적으로 사용됨.

DBS에 따라 문법 상 조금씩 다른 부분이 존재하기도 함.

2. 중복 처리
SQL에서 table은 relational algebra의 relation과는 달리, tuple에 대한 multiset(중복을 허용하는 set)임. 즉, SQL에서는 relational algebra의 select/project/cartensian product가 모든 tuple에 대해 연산을 수행하며 중복에 상관없이 결과에 포함시킴.

하지만 SQL에서는 union(합집합), intersect(교집합), except(차집합)과 같은 tuple에 대한 set operation을 제공하는데, 이것들은 set을 가정한 연산이므로 중복을 제거함. 만약 multiset으로 처리하고 싶다면 대신 union all, intersect all, except all을 사용해야 함. ***집합 연산을 잘 활용하자.***

만약 어떤 한 tuple이 r에서 m번, s에서 n번 나온다고 하면 all 연산을 했을 때 결과에서는 아래와 같은 횟수만큼 나옴. 즉, 마치 각 tuple이 서로 다른 것처럼 취급되어 결과에 포함됨.

1) r union all s -> m + n
2) r intersect all s -> min(m, n)
3) r except all s -> max(0, m - n)

2.1.3. Schema와 Catalog

SQL2 표준에서 schema는 table, 자료형, constraint, view, doamin 등을 포괄하는 개념임.

Catalog는 SQL 환경에 존재하는 schema의 집합임. catalog는 하나 이상의 schema를 포함하는데, meta-data 조회를 위한 view인 information schema는 항상 포함함.

오라클 시스템에서 catalog는 DBMS가 관리하는 전체 DB를 의미하고, schema는 사용자 계정에 1대1로 대응되어 존재하며 해당 사용자의 id와 이름이 같음.

2.1.4. Oracle DBMS

1. Oracle DBMS
Oracle DBMS는 Oracle에서 개발한 RDBMS임. Oracle에서 제공하는 DB, DBMS, 클라이언트 도구로 이를 활용할 수 있음.

해당 수업에서는 무료 DBMS인 Oracle XE version DBMS와, GUI 기반의 무료 DB 개발(클라이언트) 도구인 Oracle SQL Developer를 활용함.

Orcale DBMS는 여러 process들에 의해 동작하게 됨. 특히, server process는 서버로서 클라이언트를 처리하는 역할을 함.

2. DBA
Oracle에서 대표적인 DBA으로는 system과 sys가 있음. DBMS 설치 시에 이런 DBA에 대한 암호 등을 지정하게 됨. system은 대부분의 DBA 기능을 가진 계정임. sys는 dynamic performace table, data dictionary를 구성하는데 사용되는 table과 view 등을 소유하고 있고, 이에 대한 변경은 일반적인 경우 수행하지 않으므로 sys 계정을 사용할 일은 잘 없음.

Oracle 시스템을 사용하려면 DBA로 시스템에 사용자를 생성하고 권한을 부여해야 함. 이는 아래와 같이 수행할 수 있음. 이 경우 hodori라는 이름, tooshytotell이라는 암호를 가진 사용자를 생성하고, connect/resource 권한 묶음을 부여함.

```
Create user C##hodori identified by tooshytotell
default TABLESPACE users
temporary TABLESPACE temp;

Grant connect, resource to C##hodori;
```

3. 연결

클라이언트 프로그램이 Oracle 서버에 접근할 때는, 아래와 같이 지역 연결이냐 원격 연결이냐에 따라 클라이언트에서 명시해야 하는 정보가 다름. 지역(local) 연결의 경우 id와 암호만 명시하면 되고, 원격(remote) 연결의 경우 id와 암호, 호스트 이름, 포트 번호를 명시해야 함. 원격 호스트에서는 Oracle Net Listener가 해당 포트에 대해 listen하고 있으며, 요청이 오면 이를 Oracle 서버에 전달함.

```
Connect sys/mypassword                              // local

Connect sys/mypassword@dbserver.ssu.ac.kr:1522      // remote
```

4. Data Dictionary

Oracle 시스템에서는 user(user view), all(extended user view), dba(dba view)를 접두사로 하고, 그 뒤에 table을 명시하여 data dictionary의 table에 접근할 수 있음.

```
Select * from user_tables;
Select * from all_tables;
```

5. Dynamic Performace Table

Dynamic Performace Table은 DBMS의 활동 현황을 기록하는 가상 table(view)로, sys가 소유하고 있음.

6. Sequence

Sequence는 유일한 값을 생성하는 데에 사용되는 객체임. 특히 primary key의 값을 생성하는 데에 사용할 수 있고, 여러 사용자의 의해 사용될 수 있음. 참고로, 이는 transaction의 성공 여부와 상관없이 생성된다고 함.

sequence는 아래와 같이 Create/Alter/Drop으로 생성/수정/삭제하고, nextvalue로 유알한 값을 얻음.

```
Create sequence seqname
[start with n]
[increment by n]
[maxvalue n | nomaxvalue]
[minvalue n | nominvalue]
[cycel | nocycle]
[cache | nocache]

Create sequence seq1 start with 100 increment by 10;
Alter sequence seq1 maxvalue 100000;
Insert into dept1 values(seq1.nextvalue, 'hello');
Drop sequence seq1;
```

7. 자료형

varchar(n)이 표준이지만, Oracle 시스템에서는 이와 같이 작성해도 varchar2()로 저장한다고 하며, 최대 길이는 4000byte임.

Oracle 시스템에서는 ”와 같이 빈 문자열을 null로 인식함.

2.2. DDL SQL

SQL DDL 영역의 기본 키워드는 Create/Drop/Alter임.

2.2.1. Create

1. Create

Create로 아래와 같이 table을 생성할 수 있음. table 이름, attribute와 domain, integrity constraint 순서대로 작성함.

```
Create table R (
    A1 D1,
    ...,
    An Dn,
    (integrity-constraint1),
    ...,
    (integrity-constraint1)
);
```

또한 기존의 table의 구조와 데이터를 가져와 table을 생성할 수 있음.

```
Create table myTable as
select *
from professor;
```

생성한 table은 Describe로 그 구조(schema)를 확인할 수 있음.

```
Describe R;
```

2. Domain 자료형

domain이 가질 수 있는 자료형(Data Type)은 여러 가지가 존재하고, 대표적인 것들은 아래와 같이 지정함.

1) char(n) : n자리의 고정된 문자열.
2) varchar(n) : 최대 길이가 n인 가변 길이 문자열.
3) int : 4바이트 크기의 정수.
4) smallint : 2바이트 크기의 정수.
5) numeric(p,d) : 유효숫자 개수가 p이고, 소수점 아래 숫자 개수가 d인 고정소수점 표현.
6) float(n) : precision이 n인 부동소수점.
7) double precision 또는 real : 배정밀도 부동소수점.

3. Integrity Constraint

Create 시에 아래와 같이 integrity constraint를 지정할 수 있음.

1) not null은 해당 attribute가 null 값을 허용하지 않음을 나타냄.
2) unique는 해당 값이 tuple들에 대해 유일해야 함을 나타냄.
3) primary/foreign key를 지정할 수 있음. 이때 foreign key는 어차피 해당 table의 priamry key를 참조하므로, foreign key를 지정할 때는 table명 뒤에 작성하는 attribute를 명시하지 않아도 됨.
4) check로 해당 attribute가 가질 수 있는 값에 대한 constraint를 정의할 수 있음.

```
Create table professor (
    pID         char(5),
    cID         char(5),
    deptName    varchar(20),
    salary      numeric(8,2) not null,
    gender      char(1),
    primary key (pID, cID),
    unique (pID, cID, deptName),
    foreign key (cID) references course,
    foreign key (deptName) references department(pID),
    check (gender in ('F', 'M'))
);
```

primary key가 하나의 attribute로 구성되는 경우 아래와 같이 자료형 뒤에 명시할 수도 있음. 또한 unique도 동일하게 명시할 수 있음.

```
pID     char(5) primary key,
cID     char(5) unique,
```

아래와 같이 따로 작성할 때 constraint에 이름을 붙여 삭제/수정에 용이하도록 할 수 있음. 이름을 붙이지 않으면 DBMS가 자동으로 이름을 붙임.

```
constraint myForeignKey foreign key (deptName) references department
```

2.2.2. Drop/Alter

Drop과 Alter로 table을 삭제/수정할 수 있음.

1. Drop

Drop은 아래와 같이 단순히 table명을 작성하여 삭제함.

```
Drop table r;
```

2. Alter

Alter는 아래와 같이 statement에 해당하는 내용을 반영함. add에 attribute와 domain을 지정하여 attribute를 추가할 수 있고, integrity constraint를 지정하여 추가할 수 있음. 또한 drop에 attribute를 지정하여 attribute를 제거할 수 있음. 이때 추가한 attribute의 값에는 null이 들어감.

constraint는 이름을 지정하여 disable/enable할 수 있음.

```
Alter table statement;

Alter table r add A D;
Alter table r add constraint myConst
                                foreign key (deptName) references DEPARTMENT;
Alter table r drop A;

Alter table r disable constraint myConst;
```

Delete 키워드는 Drop과는 달리 tuple에 대한 삭제를 수행하므로 DML에 속함. 참고로, Truncate는 tuple을 삭제하지만 DDL이라고 함.

2.3. DML SQL

SQL DML 영역의 기본 키워드는 Insert/Select/Delete/Update임.

2.3.1. Select

Select는 relation을 작성하고 지정한 조건에 부합하는 데이터를 찾아 relation을 구성하여 반환하는데, 아래와 같이 총 6개의 순차적인(순서가 맞아야 함.) clause(절)로 구성될 수 있음. 이때 select와 from을 제외한 clause는 생략이 가능하지만, having은 group by가 등장해야 사용이 가능함.

```
select A1, A2, ..., An
from R1, R2, ..., Rm
where P
group by <grouping attributes>
having <conditions>
order by <ordering attributes>
```

이는 아래와 같은 실행 과정을 가짐(물론 DB에 따라 다르게 구현되었을 수 있음.). 생략된 clause에 대해서는 작업을 수행하지 않음.

1) from에 작성한 table들에서 각각 tuple을 하나씩 추출하여 tuple 조합을 구성함. 아래의 과정은 이렇게 추출한 모든 tuple 조합에 대해 수행됨.
2) tuple 조합 중 where의 조건을 참으로 만족시키는 것들만 추출하고, 거짓인 것들은 제거함.
3) group by에 명시한 attribute에 따라 tuple 조합을 group으로 묶음.
4) aggregate function 계산.
5) group에 having에 명시한 조건 적용.
6) tuple 조합에서 select에 의해 최종 attribute 선택.

추가로 유의할 점은, SQL select문에서 실제로 최종 결과로 추출하지 않는다면, from에 의해 모호함이 발생해도 오류가 아니라는 것임.

1. select
select clause에는 사용자가 보고 싶은 attriubte들의 이름을 작성함. 즉, relational algebra에서의 project에 대응되는 연산임.

**는 모든 attriubte를 의미함. 또한 각 attribute에 대해 단순 연산자(+, -, / 등)나 연산 함수(log, sqrt 등), aggregate function 등을을 함께 작성하여 해당 attribute의 값을 일괄적으로 변경할 수 있음.*

```
select *
from professor;

select pID, salary / 12
from professor;
```

결과 relation에서 tuple들 간의 중복을 허용하지 않으려면 distinct를 작성할 수 있고, 중복을 허용함을 명시하려면 all을 작성할 수 있음. select는 기본적으로 중복 tuple을 허용함(all이 default임.).

```
select distinct deptName
from professor;
```

2. from
from clause에는 해당 query와 관련된 table들의 이름을 작성함. 즉, relational algebra에서의 cartesian product에 대응되는 연산임.

```
select *
from professor, teaches;
```

3. where
where clause에는 결과 tuple이 만족해야 하는 조건(논리식)을 작성함. 즉, relational algebra에서의 select에 대응되는 연산임. 조건은 비교 연산자(=, !=, <, >, <=, >=)와 관계 연산자(and, or, not)를 활용하여 작성할 수 있음. 또한, !=와 동일한 기능을 하는 <> 연산도 지원함.

```
select name
from professor
where deptName = 'CS' and salary > 1000;
```

이때 두 문자열의 비교에는 like를 사용할 수 있음(물론 이것 말고도 SQL은 여러 문자열 처리 연산을 지원함.). 두 문자열이 일치하는 경우 true, 일치하지 않는 경우 false임. 이렇게 like를 사용하는 경우 문자열 안에 %를 작성하여 임의 길이의 임의 문자를 가지는 문자열을, _를 작성하여 길이 1의 임의 문자를 가지는 문자열을 나타낼 수 있음. 이때 %나 _ 등을 와일드카드가 아니라 단순 문자로 쓰려면 escape를 쓰고 그 뒤에 지정한 문자를 escape 문자로 사용할 수 있음. 즉, 해당 문자를 % 앞에 작성하여 %를 단순 문자로 사용하도록 escape할 수 있음.

```
select name
from professor
where deptName like 'C_' or deptName like '100\%' escape '\';
```

값의 범위를 확인할 때는 between을 사용할 수 있음. 이 경우 경계값도 포함함.

```
select name
from professor
where salary between 5000 and 6000;
```

where에서는 tuple끼리의 비교도 가능함. 아래와 같이 ()를 사용하여 임시 tuple을 생성하여 비교할 수 있음.

```
select name
from professor, teaches
where (professor.pID, deptName) = (teaches.pID, 'CS');
```

query의 실행 과정 상 where에서는 aggregate function이 계산되지 않았으므로, 이를 사용하여 조건을 구성할 수 없음. 대신 조건에 ()로 서브 query를 작성해 그 안에서 aggregate function을 계산하거나, having에서 적용하는 식으로 구현할 수 있음.

또한 where에서 null 여부에 대한 조건은 =" 등이 아닌, is null과 is not null로 지정함.

```
select name
from professor
where salary is null;
```

4. group by

group by clause에서는 attribute를 작성하여 해당 attribute 대해 동일한 값을 가지는 tuple 조합들을 group으로 묶음.

이때 null도 값으로 고려되어, null이 존재한다면 null에 대한 group도 생성됨.

group by를 사용한 경우 select에는 group by에 명시한 attribute 또는 특정 attribute에 대한 aggregate function만이 등장할 수 있음(물론 둘 다 반드시 등장해야 하는 것은 아님.). 이때 aggregate function은 각 group에 대해 계산됨.

```
select deptName, avg(salary)    // pID 등을 작성하면 오류.
from professor
group by deptName;
```

5. having

having clause에서는 group by에서 묶은 group에 대한 조건을 작성함. 즉, group 중 조건을 참으로 만족시키는 것만을 추출하도록 함. 이때 조건은 aggregate function을 사용하여 작성하는 것이 일반적임.

```
select deptName, avg(salary)
from professor
group by deptName
having avg(salary) > 1000
```

이에 따라 tuple 단위의 조건은 where에서, group 단위의 조건은 having에서 지정하게 됨.

6. order by
order by는 attribute를 지정하여 tuple 조합을 해당 attribute의 값을 기준으로 정렬함. 오름차순(asc)이 default이고, 문자열의 경우 영어 알파벳 순서대로 정렬됨. 내림차순으로 정렬하려면 desc를 뒤에 작성함.

여러 개의 attribute를 작성하는 경우 맨 앞 attribute부터 사용하여 정렬하고, 동일한 값을 가지는 것들끼리는 그 이후에 작성한 attribute를 사용하여 정렬함. 이 경우 내림차순을 지정하려면 각 attribute 바로 뒤에 desc 작성함.

```
select deptName
from professor
order by deptName desc, name;
```

2.3.2. Insert/Delete/Update

SQL DML 영역의 Insert/Delete/Update 키워드에 대해 알아보자.

1. Insert
Insert는 table에 tuple을 입력하는 연산임. 아래와 같이 작성하며, values로 작성해도 되고 attribute명을 명시해도 됨. 값이 없는 부분은 null을 작성함.

```
Insert into r values ('437', 'abc', null, 4);

Insert into r (cID, title, deptName, credit) values ('437', 'abc', null, 4);
```

또한 select-from-where문을 사용해 입력할 tuple들을 지정할 수 있음. 이때 select-from-where문이 우선 수행되고, 그 결과가 insert에 사용됨. 즉, 아래의 예시는 professor table이 2배가 됨.

```
Insert into professor
select * from professor;
```

입력하려는 tuple에 foreign key가 존재하는 경우 입력이 까다로울 수 있음. 이에 따라 table A에 대한 foreign key를 가지는 table B에 tuple을 입력하는 것은 아래의 3가지 방법으로 수행할 수 있음.

1) A에 tuple을 입력하고, 그 뒤에 B에 tuple을 입력함.
2) B 생성 시에 foreign key에 대한 constraint를 작성하는 대신, tuple 입력 이후 Alter로 constraint를 추가함.
3) Alter로 B의 foreign key에 대한 constraint를 일시적으로 disable했다가 다시 enable함.

2. Delete
Delete는 table에서 tuple을 삭제하는 연산임. 아래와 같이 table명만을 지정하면 해당 table의 모든 tuple이 삭제되고, where로 조건을 지정하면 해당 조건이 true인 tuple들을 삭제함.

이때 tuple의 특정 attribute 값을 삭제 또는 변경하려면 Update를 사용해야 함.

```
Delete
from professor;

Delete
from professor
where deptName = 'EE';

Delete
from professor
where deptName In (select deptName
                   from department
                   where building = 'Vision Hall');
```

여기에서 In은 특정 attribute의 값이 주어진 값 목록에 포함되면 true, 포함되지 않으면 false를 반환하는 연산임.

```
Select name
From employees
Where department In ('HR', 'IT');   // department 값이 'HR' 또는 'IT'이면 true.
```

아래와 같이 aggregate function을 활용하여 tuple을 삭제하는 경우 매번 새로운 집계 결과를 내놓는다면 삭제된 tuple에 의해 값이 변하게 되는데, SQL에서는 매번 집계 결과를 계산하는 대신 query 초기에 한 번 계산한 값을 계속 활용하도록 되어 있다고 함.

```
Delete
from professor
where salary < (Select avg(salary) from professor);
```

전체 tuple을 삭제할 때는 Truncate를 사용하면 더 빠르다고 함.

```
Truncate table myTable;
```

3. Update

Update는 table의 특정 attribute를 수정/삭제하는 연산임. 아래와 같이 attribute와 그 값을 지정하고, where로 tuple을 특정지어 값을 수정/삭제할 수 있음.

```
Update professor
set salary = salary * 2
where salary < 2000;
```

또한 SQL에서는 아래와 같이 case문을 작성하여 조건에 따라 값을 반환하도록 할 수 있음.

```
Update professor
set salary = case
             when salary <= 5000 then salary * 2
             else salary * 1.5
             end;
```

2.3.3. Rename

Select에서는 as로 attribute와 table에 대한 rename을 수행할 수 있음.

아래와 같이 attribute에 대해서 rename할 수 있고, as를 생략할 수도 있음.

```
select name as profName, cID
from professor natural join teaches;

select name profName, cID
from professor natural join teaches;
```

아래와 같이 table에 대해서도 rename할 수 있음. 이때 동일한 table을 서로 다른 이름으로 rename하여 하나의 table에서 여러 번 tuple을 가져올 수 있음.

```
select name
from professor as T, professor as K;
```

2.3.4. null and Three-valued Logic

1. null
attribute 값에서 null은 값이 존재하지 않거나 값을 알 수 없음을 나타냄.

SQL에서 null이 포함된 산술 연산의 결과는 항상 null로 정의되어 있음.

2. Three-valued Logic
null이 포함된 비교 연산의 결과는 항상 unknown으로 정의되어 있음. 이에 따라 SQL에서는 논리값이 true, false, unknown으로 3가지인데, 이를 Three-valued Logic이라고 함. where에서는 이 3가지 중 true일 때만 해당 tuple을 추출함.

논리값이 3가지이므로, unknown에 대한 OR, AND, NOT이 아래와 같이 추가로 정의되어 있음. true > unknown > false인데, OR는 큰 것을, AND는 작은 것을 도출하는 것으로 암기하자.

1) OR
(unknown or true) = true
(unknown or false) = unknown
(unknown or unknown) = unknown

2) AND
(unknown and true) = unknown
(unknown and false) = false
(unknown and unknown) = unknown

3) NOT
(not unknown) = unknown

2.4. Advanced SQL

2.4.1. Aggregate Function

Aggregate Function(집계 함수)는 attribute의 여러 값들에 대한 정보를 반환하는 함수임. 아래와 같은 aggregate function들이 있음.

avg() : 평균값 반환.
min() : 최솟값 반환.
max() : 최댓값 반환.
sum() : 총합 반환.
count() : 값의 개수 반환.

aggregate function은 null값을 무시함. 예를 들어, 모든 값이 null이면 count() 값은 null이 아니라 0임. 참고로, primary key가 아닌 경우 null일 수 있으므로 count()로 tuple 개수를 셀 때에는 primary key를 사용하는 것이 좋음.

distinct한 요소들의 개수를 세는 등의 작업은 아래와 같이 query를 작성할 수 있음.

```
select count(distinct pID)
from teaches
where sememster = 'Spring' and year = 2010;
```

2.4.2. Join

1. Inner Join
Inner Join(Theta Join)은 아래와 같이 from에 inner join과 on을 사용하여 수행할 수 있음. 이때 inner는 생략하고 join만 작성할 수 있음. 조건으로 동등 조건(=)을 작성하면 Equi Join임.

이때 동일한 이름의 attribute가 존재하면 어떤 table의 것인지 알 수 없으므로, select와 where 등에서 attribute 앞에 table명과 .을 작성하여 모호함을 배제해야 함.

```
select name, d.cID, s.cID
from  d inner join s on d.cID = s.cID;
```

theta/epui join은 아래와 같이 단순히 where에 조건을 작성하여 수행할 수도 있지만, inner join을 사용하는 것이 가독성 등의 측면에서 좋음.

```
select name, cID
from professor, teaches
where professor.pID = teaches.pID;
```

2. Natural Join
natural join은 아래와 같이 from에 작성할 수 있음. natural join의 앞뒤로 오는 table들에 대해서 join이 적용됨.

이때 using에 attribute들을 지정하여 해당 attribute에 대해서만 natural join이 수행되도록 할 수 있음. 이때 attribute는 괄호로 묶어줘야 함. using을 사용하는 경우 natural을 생략해도 natural join이 수행됨. 지정하지 않은 attribute 중 이름이 같은 것이 있다면 오류가 날 수 있으므로 rename하는 것이 좋음. 특히 natural join 시에는 동일한 이름의 attribute에 대해 자동으로 연산이 수행되므로 주의가 필요한데, using을 사용하여 문제 상황을 방지할 수 있음.

```
select name, cID
from professor natural join teaches;

select name, cID
from professor natural join teaches, course // course는 단순히 from이 적용됨.
where teaches.cID = course.cID;

select name, cID
from (professor natural join teaches) join course using(cID, deptName);
```

3. Outer Join
Outer Join은 inner join과 동일하게 작성하는데, 그 종류에 따라 아래와 같이 키워드만 바꿔주면 됨.

left/right/full outer join -> outer join 수행. on을 사용해 조건 명시해야 함.
natural left/right/full outer join -> natural outer join 수행. 조건을 명시하지 않음.

```
select name, d.cID, s.cID
from d, s
where d left outer join s on d.cID = s.cID;
```

2.4.3. Nested Subquery

1. Nested Subquery
SQL에서는 아래와 같이 (select-from-where)로 구성되는 Nested Subquery를 지원함. 이는 이론상으로

table이 들어갈 수 있는 위치에 사용될 수 있으며, 주로 where, from에 사용하여 집합 처리를 할 때 활용함.

```
select name
from professor
where salary = (select salary from professor where pID='10') and pID <> '10';
```

from에 subqeury를 작성해 having을 대체할 수도 있음. 참고로 from에 subquery를 작성하는 것은 대부분의 DBMS에서 지원하지만, 그렇지 않은 것도 존재함.

2. Correlated Subquery
Correlated Subquery(상관 서브질의)는 외부 query의 table을 참조하는 subquery를 말함.

이 경우 해당 table의 각 tuple에 대해 subquery를 수행하게 되므로 자원을 많이 사용하게 됨.

```
SELECT employee_id, first_name, salary, department_id
FROM employees e
WHERE salary > (
                SELECT AVG(salary)
                FROM employees e2
                WHERE e2.department_id = e.department_id
                );
```

3. Scalar Subquery
하나의 tuple만을 가지는 single-row table을 반환하는 subquery를 Scalar Subquery라고 함. 즉, scalar subquery는 하나의 값만을 반환하므로, 이는 하나의 값을 반환하는 어떤 곳에도 활용될 수 있음.

```
select deptName, (select count(*)
                  from professor P1
                  where d1.deptName = P1.deptName)
from department d1;
```

subquery가 단일 값을 반환하는 경우 아래와 같이 =로 비교할 수 있음. select의 실행 과정에서는 작업이 tuple 별로 수행되고, 당연하게도 where에서도 하나의 tuple이 사용됨. 이 경우 subquery로 단순 비교를 수행하려면 subquery도 single row여야 함. 여러 개의 row를 가지는 table을 비교에 사용하면 runtime error가 발생함. 그래서 subquery에서 실제로 row가 하나가 되도록 primary key 등으로 select하거나, aggregate function으로 하나의 값을 전달하거나 하는 식으로 사용함. 특히 where에서 aggregate function이 나올 수 없으므로 이를 계산하는 데에 subquery를 쓸 수 있음.

```
select name
from professor
where salary = (select salary from professor where pID='10') and pID <> '10'
```

2.4.4. Nested Subquery : 관련 연산자들

1. IN 연산자
IN은 아래와 같이 특정 값이 집합에 속하는지를 판단하는 연산자로, 집합에 속하면 true, 아니면 false를 반환함. 이는 특히 subquery와 함께 사용할 수 있음.

```
select name, salary
from professor
where pID in (10, 21, 22);
```

아래와 같이 tuple을 구성해 비교할 수도 있음.

```
select count(distinct cID)
from takes
where (cID, semester, year) in (select cID, semester, year
                                from teaches
                                where pID = 10);
```

2. All/Some/Any

All/Some/Any는 하나의 값과 여러 개의 값을 비교하는 데에 사용하는 연산자임. 이는 아래와 같이 작성하는데, all은 모든 값과의 비교가 참이어야 참이고, some/any는 하나의 값과의 비교가 참이면 참임. 이때 some과 any는 동일함.

```
(1 > 2)             // false
(3 >all {2, 3})     // false
(3 >some {2, 3})    // true
```

하나의 값끼리의 비교는 기본 연산자들로 수행할 수 있지만, 여러 개의 값과 비교하는 경우는 이 연산자들을 활용해야 함. 아래와 같은 경우에 사용할 수 있음.

```
select name
from professor
where salary >all (select salary from professor where deptName='CS');
```

이때 =some과 in은 동일하게 동작하지만 !=some과 not in은 그렇지 않으므로, some과 in은 의미적으로 동일하지 않음. all도 유사한 이유로 not in과 동일하지 않음.

3. Exists

Exists는 subquery의 결과가 존재하면 true를, 존재하지 않으면 false를 반환함.

exists는 사용하는 subquery가 correlated subquery가 아니라면 항상 참 또는 거짓이므로 의미가 없음. 이에 따라 correlated subquery에 사용하여 외부의 각 tuple에 대해 연산하는 것으로 주로 사용됨. 물론 매 tuple에 대해 연산되어야 하는 경우 자원을 많이 사용하게 됨.

```
select S.cID
from teaches as S
where S.semester = 'Fall' and S.year = 2009 and
exists( select *
        from teaches as T
        where T.semester = 'Fall' and T.year = 2010 and S.cID = T.cID);
```

sql에서는 for all을 명시적으로 구현하는 연산자를 제공하지 않음. exists는 아래와 같이 이를 구현하는 데에 주로 활용함. 특히 exists를 활용하면 relational algebra의 division을 sql에서 구현할 수 있음. 아래의 예시는 모든(for all) 과목을 수강한 학생을 찾는 query이고, 차집합(except)를 활용함.

```
select S.sID, S.name
from student as S
where not exists(
                    (select cID from course where deptName = 'CS')
                    except
                    (select T.cID from takes as T where S.sID = T.sID)
                );
```

4. Unique

Unique는 subquery의 각 tuple이 유일하면 true를, 유일하지 않으면 false를 반환함. 이때 각 null은 서로 다른 값으로 취급되고, 공집합은 true로 처리됨.

```
unique( <1, 2>, <1, 2> )        -> false
unique( <1, 2>, <1, 3> )        -> true
unique( <1, 2>, <1, null> )     -> true
unique( <1, null>, <1, null> )  -> true
```

아래와 같이 subquery의 결과에 중복이 존재하는지를 판단할 수 있음.

```
select c.cID
from course as C
where unique(select T.cID
             from teaches as T
             where C.cID = T.cID and T.year = 2009);
```

5. Lateral

Lateral은 from 절에서 correlated subquery를 사용할 수 있도록 함. 즉, from에 사용되어 앞에서 포함된 table을 가져와서 subquery의 from에 포함시킴.

```
select P1.name
from professor P1, lateral (select avg(P2.salary)
                            from professor P2
                            where P1.deptName = P2.deptName);
```

6. With

With은 해당 query에서만 사용할 수 있는 임시 view를 생성함. 즉, sql 문장의 결과를 임시로 저장하여 복잡한 query를 단순화할 수 있음.

아래와 같이 view 이름과 tuple 이름을 지정해 생성하고, 한 번에 여러 개 생성할 수 있음. 또한 하나의 view에서 다른 것을 활용할 수도 있음.

```
with    deptTotal(deptName, value) as
        (select deptName, sum(salary)
        from professor
        group by deptName)
        deptTotalAvg(value) as
        (select avg(valuue)
        from deptTotal)
select deptName
from deptTotal, deptTotalAvg
where deptTotal.value = deptTotalAvg.value;
```

2.4.5. Ranking

sql에서는 여러 가지 ranking 기능을 제공함.

1. Top/Limit/Rownum Clause

일반적인 상용 DBMS에서는 상위 tuple들을 반환하는 기능을 제공함. 여기에서는 일부 DBMS에서의 문법을 살펴보자.

1) SQL Server : Top

SQL Server에서 top clause는 상위 tuple들을 추출함. persent는 퍼센트로 취급하는 옵션이고, with ties는 순위권에 드는 tuple과 동일한 값을 가지는 tuple은 모두 추출하는 옵션임.

```
Select top(3) * from professor order by salary;
Select top(10) persent with ties * from professor order by salary;
```

2) MySQL : Limit

MySQL에서는 limit과 offset을 활용해 상위 tuple들을 추출할 수 있음. limit은 해당 개수만큼 추출하는 것이고, offset은 해당 개수만큼의 최상위 tuple들은 건너뛰는 것임.

```
select * from professor order by salary desc limit 3;
select * from professor order by salary desc limit 3 offset 1;
```

3) Oracle : Rownum
Oracle에서는 Rownum을 사용해 상위 tuple들을 추출할 수 있음. rownum은 oracle에서 각 tuple에 대해 부여하는 가상의 번호로, where을 통과한 tuple들에 대해 1부터 순차적으로 부여함. 즉, where에 rownum을 활용한 조건을 사용하여 상위 tuple들을 추출할 수 있음.

```
select * from professor where rownum <= 3 order by salary desc;
```

2. rank()
표준 sql에서는 ranking 정보를 얻도록 하기 위해 rank()를 제공함. 아래와 같이 사용하여 ranking 정보를 얻을 수 있음. 물론 rank()만 사용하는 경우 실제로 정렬이 되진 않으므로, 실제로 정렬하려면 order by를 추가로 사용해야 함.

```
select ID, rank() over (order by GPA desc) as sRANK
from studentGrades;
```

물론 아래와 같이 rank() 대신 sql을 직접 작성할 수도 있는데, 이는 correlated subquery이므로 비효율적임.

```
select ID, (1 + (select count(*)
            from studentGrades B
            where B.GPA < A.GPA)) as sRank
from studentGrades A
order by sRank;
```

이때 null의 경우 가장 큰 값으로 처리되는 게 default임. 또한 SQL:1999부터는 아래와 같이 order by에 nulls last, nulls first를 지정해 각각 null을 맨 뒤에 넣도록, 또는 맨 앞에 넣도록 할 수 있음.

```
select ID, rank() over (order by GPA desc nulls last) as sRANK
from studentGrades;
```

rank()에는 partition by clause를 함께 사용하여 특정 attribute에 대해 partition을 나누고, 각 partition에 대한 ranking 정보를 얻을 수 있음.

```
select ID, rank() over (partition by deptName order by GPA desc) as sRANK
from studentGrades;
```

rank()는 select에 여러 번 등장할 수 있음.

3. rank()와 유사한 함수들
rank()와 유사한 함수들은 아래와 같은 것들이 있음. 기본적으로 사용 방법은 rank()와 동일한데, 결과가 일부 다름.

1) dense_rank()
rank()에서는 값이 같은 tuple은 동순위로 처리하고 해당 숫자만큼 등수를 뛰어넘게 되는데, dense_rank()를 사용하면 뛰어넘는 등수가 없도록 처리함.

- rank(): <10, 1> < 10, 1> <30, 3>, <30, 3>, <50, 5> ...
- dense_rank(): <10, 1> < 10, 1> <30, 2>, <30, 2>, <50, 3> ...

2) row_number()
rank()에서는 동일한 값의 tuple은 동순위로 처리하는데, row_number()에서는 모든 tuple이 서로 다른 순위를 가지도록 처리함. 이때 동일한 값의 tuple의 경우 어떤 것의 순위가 더 높게 지정되는지는

deterministic하지 않음.

3) percent_rank()
ranking 정보를 실수로 나타냄. n개가 있고 rank가 r이라면 $\frac{r-1}{n-1}$으로 계산함. 즉, rank가 높은 쪽부터 0에서 1까지의 값을 순차적으로 가짐.

또한 n이 1이라면 null이 반환됨.

4) cume_dist()
ranking 정보를 실수로 나타냄. n개가 있고 rank가 p라면 $\frac{p}{n}$으로 계산함. 즉, rank가 높은 쪽부터 $\frac{1}{n}$에서 1까지의 값을 순차적으로 가짐. 이때 p는 정렬된 상태에서의 앞쪽 순위 및 동일 순위 tuple의 개수임.

percent_rank()와 달리 0부터 시작하지 않는다는 차이점이 있음.

4. ntile() ranking
ntile()은 rank()와 동일한 형태로 사용할 수 있는데, order by에 지정한 attribute를 기준으로 n개의 bucket(나눠진 단위)으로 나눠서 값을 부여함. 여기에서 각 bucket이 가지는 tuple의 개수는 서로 최대 1개까지만 차이가 날 수 있음.

이때 동일한 순위의 tuple이 서로 다른 bucket에 들어가게 될 수도 있는데, 어떤 tuple이 어디에 들어갈지는 deterministic하지 않음.

예를 들어, 아래와 같은 결과를 가짐.

다음과 같은 테이블을 가정하자.

ID	GPA
S1	2.8
S2	3.5
S3	4.0
S4	3.5
S5	4.2

NTILE(2):

ID	NTILE	/* GPA */
S5	1	/* 4.2 */
S3	1	/* 4.0 */
S2	1	/* 3.5 */
S4	2	/* 3.5 */
S1	2	/* 2.8 */

2.4.6. 기타 기능

SQL에서 제공하는 기타 기능들을 살펴보자.

1. Schema/data 재활용
아래와 같이 create에 like 키워드를 사용하여 특정 table과 동일한 schema의 table을 생성할 수 있음. 이때 모든 integrity constraint가 동일하게 복제되는 것은 아닌데, 특히 primary/foreign key 등은 복제되지 않음.

```
Create table t1 like professor;
```

아래와 같이 as 키워드를 사용하여 특정 table의 schema 및 데이터를 복제해 table을 생성할 수 있음. sql:2003에는 with data를 작성해야 데이터까지 복제하도록 되어 있지만, 많은 상용 DBSM에서 이를 작성하지 않아도 데이터를 복제하도록 한다고 함.

```
Create table t1 as (select * from professor where deptName='SW') with data;
Create table t1 as (select * from professor where deptName='SW');
```

2. 대용량 객체 자료형
sql에서는 사진, 비디오 등과 같이 용량이 큰 객체(하나의 column에 들어가는 데이터)를 저장할 수 있도록 blob, clob을 지원함.

blob(Binary Large Object)은 큰 용량의 이진 데이터를 저장하는 자료형이고, clob(Character Large Object)은 큰 용량의 문자 데이터를 저장하는 자료형임. 사용법은 단순한 자료형과 동일함.

큰 객체라는 것은, 하나의 column에 들어가는 데이터를 말한다. blob은 transactional semantics(ACID properties)를 제공하지 않는다.

3. 내장 날짜 타입
sql에서 제공하는 내장 날짜 타입에는 아래와 같은 것들이 있음.

1) date : 특정 시점의 연, 월, 일을 나타내는 자료형. (ex. '2015-7-27')

2) time : 특정 시점의 시, 분, 초를 나타내는 자료형. (ex. '09:30:21')

3) timestamp : 특정 시점의 data, time을 모두 나타내는 자료형. (ex. '2015-7-27 09:30:21')

4) interval : 시간 간격을 나타내는 자료형. date, time, timestamp 자료형의 값에 interval 값을 더하면 해당 간격만큼의 새로운 date, time, timestamp 데이터가 생성됨.

```
timeObject + interval '1' day
```

4. 사용자 정의 자료형
아래와 같이 create type으로 사용자 정의 자료형을 정의할 수 있음. 이렇게 정의한 자료형은 다른 기본 자료형과 동일한 방식으로 활용할 수 있음.

이때 final을 지정하면 해당 자료형으로 다른 자료형(subtype)을 정의하지 못하도록 하고, not final은 정의할 수 있도록 함.

```
create type Dollars as numeric (12, 2) final;
```

5. domain
아래와 같이 create domain으로 domain을 정의하여 자료형과 동일한 방식으로 활용할 수 있음.

자료형은 integrity constraint를 가질 수 없는 반면, domain은 가질 수 있음. 하지만 대체로 type을 사용함. domain이 더 옛날에 나왔고, 그 이후에 type이 등장했음.

```
create domain degreeLevel
varchar(20)
constraint degreeLevelTest check (value in ('Master', 'Doctorate'));
```

6. Transaction
Transaction은 ACID(Atomic, Consistent, Isolated, Durable) property(성질)를 가지는 DB 연산의 sequence임.

transaction은 DML 문장의 사용과 함께 암시적으로 시작됨. 이후 암시적으로 종료되거나, commit/rollback work에 의해 명시적으로 종료될 수 있음. 암시적으로 종료되는 경우 commit/rollback 여부는 시스템에 의해 결정됨.

대부분의 상용 시스템은 auto_commit이 on으로 지정되어 있는데, 이 경우 sql 문장 각각이 transaction으로 처리됨.

7. Index
Index는 table에 대한 검색 기능을 향상시키는 데에 사용되는 데이터 구조임. sql 표준에서는 index에 대한 내용을 정의하지는 않지만, 많은 상용 DBMS에서 index를 지원함.

index는 아래와 같이 정의함. 즉, table의 특정 attribute에 대해 지정함.

```
create index myIndex on course(cID);
```

transaction에 대한 구체적인 내용은 DB2에서 다룸.

3. DBMS 주요 기능들

대부분의 DBMS는 아래의 기능들을 제공함. 이는 주로 SQL로서 지원됨.

3.1. View

3.1.1. View

1. View
특정 사용자는 DB의 모든 볼 필요가 없을 수 있으므로, RDBMS에서는 view를 사용하여 해당 사용자가 relation의 일부분만을 보도록 함.

이때 view는 다른 특정 table을 참조하므로, 일반적인 table과는 달리 tuple을 physical하게 실제로 가지지 않음. 이에 따라 실제 tuple을 가지는 table을 Base Relation(Table), view를 Virtual Relation(Table)이라고도 함. 또한 view가 다른 table을 참조하므로, 당연하게도 해당 table이 변경되면 view에서의 값도 자동으로 갱신됨. 즉, view는 항상 최신 데이터를 보유함.

view는 단순 참조이므로, index와 integrity constraint를 지정할 수 없음.

2. View 생성
view는 아래와 같이 create view와 임의의 유효한 query를 작성하여 생성할 수 있고, 이렇게 생성된 view는 해당 이름으로 일반 table과 동일하게 활용될 수 있음.

```
create view myProfessor as
select pID, name, degree
from professor;
```

또한 view는 다른 view로도 생성할 수 있음. 이때 직접/간접적으로 자기 자신을 참조하는 view도 존재할 수 있는데, 이를 Recursive View(순환뷰)라고 함. recursive view는 non-recursive view와는 다른 처리 방법을 필요로 하는데, 이는 뒤에서 다룸.

3. View Expansion
View Expansion(뷰 확장)은 view를 활용하는 query를 해당 view에 대한 정의를 사용하여 base table에 대한 query로 치환하는 과정임. 이 치환 과정은 view가 존재하지 않고 base table만이 존재할 때까지 수행됨.

- Create view myFaculty as
  select pID, name, deptName
  from professor
  where salary > 50000;
- Create view myFacultyCS as
  select pID, name
  from myFaculty
  where deptName = CS;
- The above view can be expanded:
  Create view myFacultyCS as
  select pID, name
  from professor
  where deptName = 'CS' and salary>50000;

3.1.2. View Modification

1. View Modification
view에 대해서는 select만이 아니라 Modification(insert/delete/update)도 가능함. 당연하게도 그 결과는 base table에 반영됨.

아래와 같이 단순 table에 대한 작업과 동일한 형태로 modification을 수행할 수 있음. 이때 값이 지정되지 않은 attribute는 base table에 null로 반영됨.

```
insert into myProfessor values ('12345', 'Lee', 'CS');
```

2. Updatable View
view는 다른 table을 참조할 뿐이므로 modification에 따른 동작이 ambiguous할 수 있는데, 이 경우 정상적으로 처리되지 않음. 이때 modification이 가능한 view를 Updatable View라고 함. 일반적으로 updatable view는 생성에 사용하는 query가 아래의 조건들은 모두 만족시키는 것임.

1) from에 하나의 relation만이 존재함. 2) select에 attribute만이 존재하고, expression, aggregation, distinct가 존재하지 않음. 3) select에 지정되지 않은 attribute들이 모두 null 값이 가능함. 4) query가 group by, having, order by, set operation을 포함하지 않음.

또한 view 정의에 join이 사용된 경우 한 번에 하나의 base table만을 modification할 수 있음.

3. With Check Option
특정 query에 의해 생성된 view에 insert하는 등의 modification을 적용했음에도 view의 정의에 의해 그 결과가 사용자에게 보이지 않을 수 있는데, 이 경우 사용자는 정상적인 상황인지를 판단할 수 없음. 이 경우 view 정의 마지막에 with check point를 지정해 해당 modification의 결과를 사용자가 확인할 수 있을 때에만 modification을 허용하도록 할 수 있음.

```
create view myProfessor as
select pID, name, degree
from professor
where deptName='CS'
with check option;
```

3.2. Integrity Constraint

3.2.1. Integrity Constraint

1. Integrity Constraint
앞에서 정리한 것처럼 relational DB에서의 IC(Integrity Constraint)에는 구체적으로 아래와 같은 것들이 있고, 이는 accuracy와 consistency를 위한 것임.

1) Domain Intergrity Constraint : 각 attribute는 정의된 domain 내의 값만을 가져야 함. 즉, 값의 자료형, 범위, 형식 등에 대한 제약임.

2) Entity Intergrity Constraint : relation 내에서 tuple이 유일하게 구별될 수 있어야 함. 즉, primary key는 null 값을 가질 수 없고, 고유해야 함.

3) Referential Intergrity Constraint : relation 사이의 관계(연결)이 항상 유효해야 함. 즉, foreign key가 참조하는 table에 존재하는 값과 일치하거나 null이어야 함. 참고로 이는 값을 기준으로 관계를 나타내기 때문에 존재하는 constraint로, 포인터를 사용한다면 존재하지 않을 것임.

2. 기본 IC
create에서 지정하는 등으로 단일 table에 적용 가능한 기본적인 IC로는 아래와 같은 것들이 있음.

1) not null : 해당 attribute가 null 값을 허용하지 않음을 나타냄.
2) unique : 해당 값이 tuple들에 대해 유일해야 함을 나타냄. null값도 가질 수 있음.
3) primary key : primary key 지정. 해당 값은 null일 수 없음.
4) foreign key : foreign key 지정. 이때 참조하는 대상이 실제로 존재하거나, 또는 null을 값으로 가져야 함. 또한 foreign key는 어차피 priamry key를 참조하므로, foreign key를 지정할 때는 table명 뒤에 작성하는 attribute를 명시하지 않아도 됨.
4) check() : 해당 table이 항상 만족해야 하는 조건을 지정함.

```
Create table professor (
    pID         char(5),
    cID         char(5),
    deptName    varchar(20),
    salary      numeric(8,2) not null,
    gender      char(1),
    primary key (pID, cID),
    unique (pID, cID, deptName),
    foreign key (cID) references course,
    foreign key (deptName) references department(pID),
    check (gender in ('F', 'M'))
);
```

primary key가 하나의 attribute로 구성되는 경우 아래와 같이 자료형 뒤에 명시할 수도 있음. 또한 unique도 동일하게 명시할 수 있음. 참고로, primary key로 지정되면 null 값을 가질 수 없지만, unique로 지정된 경우에는 null값을 가질 수 있음.

```
pID     char(5) primary key,
cID     char(5) unique,
```

아래와 같이 따로 작성할 때 constraint에 이름을 붙여 삭제/수정에 용이하도록 할 수 있음. 이름을 붙이지 않으면 DBMS가 자동으로 이름을 붙임.

```
constraint myForeignKey foreign key (deptName) references department
```

3. on delete/update

foreign key 정의 시에 delete/update에 의해 referential integrity constraint가 위반된 경우에 대한 동작을 정의할 수 있음. 동작이 정의되지 않았다면 delete/update가 수행되지 않고, 동작이 정의되었다면 해당 동작에 따라 delete/update가 수행됨. 사용자는 동작으로 cascade, set null, set default를 지정할 수 있음.

아래와 같이 referential integrity constraint에 대한 부분은 참조하는 table에서 작성하며, 참조되는 table에서는 선언하는 부분이 없음. 또한 on delete/update는 참조되는 table에 대한 delete/update 시에 수행하는 동작임.

```
create table teaches(
...
foreign key(pID) references professor,
on delete cascade,
on update cascade);
```

참조되는 table에서 tuple이 delete/update되면 referential integrity constraint가 위배됨. cascade가 지정되었다면 delete/update 연산이 참조하는 table에 파급되어 대응되는 두 tuple이 모두 delete/update됨. set null이 지정되었다면 참조되는 tuple을 delete/update하고 대응되는 foreign key 값을 null로 함.

물론 해당 foreign key가 primary key에 포함되어 있다면 null로 지정할 수 없음.

3.2.2. Initially Deferred

IC는 기본적으로 즉시 검사되지만, foreign key 등을 지정하는 경우 이 조건이 번거로울 수 있으므로, sql에서는 아래와 같이 IC 검사를 미뤘다가(deferred) transaction이 끝날 때만 검사하도록 지정할 수 있음.

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id INT,
    CONSTRAINT fk_dept FOREIGN KEY (dept_id)
        REFERENCES departments(dept_id)
        DEFERRABLE INITIALLY DEFERRED
);
```

3.2.3. Assertion

check와 assertion을 사용하여 복잡한 IC를 정의할 수 있음. 물론 SQL 표준에서는 check에 subquery를 포함한 임의의 조건을 허용하고, SQL2에는 assertion이 정의되어 있지만, 일반적으로 상용 DBMS에서는 check에서의 subquery와, assertion을 비용과 성능 등의 이유로 지원하지 않는다고 함.

assertion은 아래와 같이 정의할 수 있고, DB 전체에 대해 해당 조건이 참이어야 함을 나타냄. 전체 DB에 대해 적용되는 조건이므로 관련 table이 수정될 때마다 검사를 수행해야 하는 overhead가 존재함. 이런 자원 낭비에 의해 현재는 assertion 대신 trigger를 사용함.

```
CREATE ASSERTION limit_total_salary
CHECK ((SELECT SUM(salary) FROM employees) <= 10000000);
```

3.3. Trigger

3.3.1. Trigger

Trigger는 상용 DBMS에서 IC 관리를 위해 사용하는 기능임. 여기에서는 표준 sql에서의 trigger를 알아봄.

trigger는 ECA(event, condition, action) rule으로, 이 3가지를 모두 포함하도록 구성됨. 즉, DB에 event(insert/delete/update)가 발생하면 condition을 평가하여 action을 수행하도록 함. trigger는 아래와 같이 create trigger로 정의하는데, before/after를 지정해 event 발생 전 또는 후에 처리할지를 지정하고, insert/delete/update로 event를 지정하고, of와 on으로 attribute와 table을 지정하고, when으로 condition을 지정하고, begin과 end 사이에 action을 지정함(문장이 하나인 경우 begin과 end 생략 가능.).

for each row는 변화가 발생한 각 tuple에 대해 trigger를 개별적으로 실행하도록 지정하는 것임. for each row 대신 for each statement를 지정하면 해당 문장(table 조작)에 대해 trigger를 한 번만 실행하도록 지정하는 것임.

begin에 atomic을 지정하면 begin부터 end까지의 작업을 하나의 transaction으로 처리함.

```
create trigger myCred after update of grade on takes
referencing old row as orow
referencing new row as nrow
for each row
    when nrow.grade <> 'F' and nrow.grade is not null
begin atomic
    Update student
    ...
end;
```

이때 변화가 발생한 tuple을 지칭하는 문장을 아래와 같이 trigger에 작성하여 조건 등에 활용할 수 있고, row 대신 table을 지정해 table 단위 처리도 가능함. 이때 당연하게도 delete에서는 변화 후 tuple을 사용할 수 없고, insert에서는 변화 전 tuple을 사용할 수 없음.

```
    referencing old row as oRow
    referencing new row as nRow
```

원래 trigger는 통계 정보를 유지하거나 table의 복사본 관련 처리를 할 때 사용했는데, 현재는 materialized view와 replication 등 더 편리한 built-in 기능들이 존재한다고 함. 또한 obejct relation DBMS 등은 데이터에 대한 연산을 메소드로 구현하므로 굳이 trigger를 사용하지 않아도 됨.

3.4. Authorization

3.4.1. Authorization

1. Authorization
사용자가 query를 제출하면 DBMS는 해당 사용자의 Authorization(권한)을 우선 검사함. DBA는 모든 authorization을 가지고 있고, 다른 사용자에게 authorization을 부여함.

instance에 대한 authorization으로는 read/insert/update/delete authorization이 있음.

schema에 대한 authorization으로는 아래와 같은 것들이 있음.
1) Index Authorization : index에 대한 생성과 삭제
2) Resources Authorization : relation 생성
3) Alteration Authorization : relation에 대한 attribute 추가 및 삭제
4) Drop Authorization : relation 삭제

sql에서 지원하는 Privilege(권한)로는 아래와 같은 것들이 있음.
1) select/insert/update/delete
2) references : 해당 table에 대한 foreign key 생성. grant 시에 해당 attribute명도 지정함.
3) usage : 특정 domain 사용
4) all privileges : 모든 권한

2. Grant
Grant는 authorization을 부여하는 문장으로, 아래와 같은 형식을 가짐. 즉, 특정 relation이나 view에 대한 어떤 권한을 지정한 사용자에게 부여함. user list에는 단순히 사용자의 나열을 작성할 수도 있고, 특정 role을 지정할 수도 있고, public(모든 사용자)을 지정할 수도 있음.

grant를 작성하는 사용자는 해당 privilege를 이미 가지고 있어야 함. 또한 with grant option을 작성하면 해당 privilege가 부여된 사용자가 다른 사용자에게 그 privilege를 부여할 수 있도록 지정함.

```
    Grant <privilege list> on <relation or view name> to <user list>
    [with grant option];
```

또한 update, references에 대해서 ()로 특정 attribute를 지정하여 authorization을 부여할 수 있음. 이와 같이 특정 attribute에 대한 처리는 존재하지만, sql DBMS에서는 특정 tuple에 대한 authorization 처리는 지원하지 않아 이는 application 수준에서의 구현이 필요함.

```
    Grant select on professor to U1, U2 with grant option;
    Grant references(deptName) on professor to U3;
```

3. Revoke
Revoke는 authorization을 철회(revoke)하는 문장으로, 아래와 같은 형식을 가짐. user list에는 단순히 사용자의 나열을 작성할 수도 있고, 특정 role을 지정할 수도 있고, public(모든 사용자)을 지정할 수도 있음.

cascade는 함께 철회되어야 하는 privilege가 있으면 함께 철회하도록 지정하는 옵션이고, restrict는 함께 철회되어야 하는 privilege가 있으면 철회하지 않도록 지정하는 옵션임. cascade가 default임. 이때 어떤 사용자의 privilege가 철회되면 해당 사용자가 부여한 privilege도 철회되어야 하고, 이는 authorization graph로 쉽게 파악할 수 있음.

이때 여러 사용자가 각각 authorization을 부여했다면, 한 사용자가 철회해도 다른 사용자의 authorization이 남아있음. privilege list에 all을 작성하여 모든 privilege에 대한 철회를 적용할 수 있음.

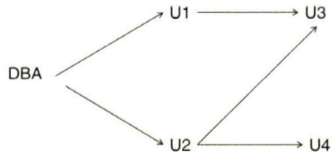
```
Revoke <privilege list> on <relation or view name> from <user list>
[restrict | cascade];
```

아래와 같이 grant option만 철회하는 것도 가능함.

```
Revoke grant option for select on professor from U5;
```

3.4.2. Authorization Graph

Authorization Graph는 DBA가 root, 각 authorization이 edge(특정 사용자가 다른 사용자에게 부여한 것.), 각 사용자가 node인 graph임. DBMS의 모든 authorization은 DBA로부터 부여가 시작되므로 각 node는 root로부터 접근이 가능함.



- A graph for select authorization on "professor"
- If DBA revokes grant from U2
  - Grant must be revoked from U4 since U2 no longer has authorization
  - Grant must not be revoked from U3 since U3 has another authorization path from DBA through U1

3.4.3. View Authorization

view는 일반적인 table과 동일하게 authorization을 관리할 수 있지만, 완전히 동일하게 처리되지는 않음. 아래는 view authorization에서의 특징임.

1) view와 base table의 authorization은 별도로 관리됨. 이에 따라 base table에 대한 authorization을 갖지 않아도 view에 대한 authorization을 가질 수 있음.
2) view는 base table에 대한 authorization을 능가하는 authorization을 가질 수 없음.
3) view는 read 권한만 있으면 생성이 가능함. view는 relation이 아니므로 생성 시에 resources authorization이 필요 없음.
4) table 생성자는 해당 table에 대한 authorization을 모두 가지지만, view의 생성자는 그렇지 않고 base table에 대해 가지고 있던 authorization만큼만을 view에 대한 authorization만을 가짐.

특정 사용자들에 대해 view를 생성하고 적절한 authorization을 부여하여 접근을 제어할 수 있음. 이때 view와 base table의 authorization은 별도로 관리되므로, view에 대한 authorization 검사는 view expansion이 수행되기 전에 처리됨.

3.4.4. Role

Role은 사용자의 집합임. role에 대해 authorization을 부여하여 해당 사용자들을 일괄적으로 처리할 수 있음. role에는 사용자들을 추가할 수 있고, role 또한 role에 추가할 수 있어 사용자를 계층적으로 관리할 수 있음.

아래와 같이 role을 생성하고, 사용자 또는 role에게 부여하고, grant/revoke할 수 있음. 당연하게도 어떤 role을 부여하면 해당 role의 모든 authorization이 부여됨.

```
create role teller;
create role manager;

grant select on branch to teller;
grant update(balance) on account to manager;

grant teller to manager;
grant teller to U1, U2;
```

3.5. Recursive View

3.5.1. Recursive View

Recursive View는 recursive query를 사용해서 정의되는 view임. view를 recursive하게 정의하여 임의의 길이의 데이터를 처리하도록 할 수 있음. 이때 recursive query가 monotonic해야 하는데, 이는 확장 시에 그 결과가 일관되고 ambigious하지 않아야 하는 것을 의미함.

recursive view는 아래와 같은 논리식을 가짐. 여기에서 prereq(x, y)는 x를 위해 y가 존재해야 함을 나타냄. 예를 들어, 과목 x를 위한 선수과목 y를 생각할 수 있음. 이를 활용해 recursive view인 recPrereq를 정의할 수 있음.

$$(\forall x)(\forall y)\ (\text{recPrereq}(x,y) \leftarrow \text{prereq}(x,y))$$
$$(\forall x)(\forall y)(\forall z)\ (\text{recPrereq}(x,y) \leftarrow \text{recPrereq}(x,z) \wedge \text{prereq}(z,y))$$

recursive view는 단순히 recursive한 관계를 발견하지 못할 때까지 반복하는 반복문으로 이해할 수 있음. 아래와 같이 table을 사용해 반복하며 결과를 생성함. 이런 과정을 Transitive Closure라고 하고, 반복이 종료된 형태를 Fixed Point라고 함. recursive view가 지원되지 않는다면 transitive closure를 직접 구현해야 함.

- Create view myFaculty as
  select pID, name, deptName
  from professor
  where salary > 50000;
- Create view myFacultyCS as
  select pID, name
  from myFaculty
  where deptName = CS;
- The above view can be expanded:
  Create view myFacultyCS as
  select pID, name
  from professor
  where deptName = 'CS' and salary>50000;

sql에서는 아래와 같이 recursive view를 활용할 수 있음. 이와 같이 transitive closure를 활용할 수 있음.

```
With recursive rePrereq(courseID, prereqID) as (
    (
        select courseID, prereqID
        from prereq
    )
    union
    (
        select recPrereq.courseID, prereq.prereqID
        from recPrereq, prereq
        where recPrereq.prereqID = prereq.courseID
    )
)
select *
from recPrereq;
```

3.6. Oracle SQL

3.6.1. 다양한 함수들

oracle에서는 아래와 같이 다양한 함수들을 지원함. 여기에서는 디테일한 사용법보다는 예시로 정리함. 참고로 dual은 attribute와 tuple이 모두 1개인 sys 소유의 table로, 여기에서는 단순히 예시를 위해 사용함.

1. 문자열 처리 함수

```
select sID from student where lower(name) = 'kim'; -- lower case

select length('hello~~') from dual; -- 문자열 길이

select concat('hell', 'o~~') from dual; -- 문자열 concat

/*
해당 문자열에서 문자 찾기. 1번째 위치부터 검색하여 2번째로 등장하는 위치 반환.
위치는 1부터 시작하고, 존재하지 않으면 0 반환.
*/
select instr('MILLAR', 'L', 1, 2) from dual;

select replace('jack and jue', 'j', 'bl') from dual; -- 특정 문자열 치환

/*
부분문자열 추출. 8번째 위치부터 2개의 문자 추출.
위치는 1부터 시작함.
*/
select substr('202048-1291239', 8, 2) from dual;
```

2. 숫자형 처리 함수
숫자형 처리 함수로는 ROUND, TRUNC, MOD, CEIL, FLOOR, SIGN, ABS 등이 있음.

```
    /*
    해당 숫자를 소수 두 번째 자리로 반올림.
    -2로 지정하면 정수 두 번째 자리로 반올림하고, 0으로 지정하면 반올림하여 정수만 반환
함.
    0이 default임.
    */
    select round(456.789, 2) from dual;

    /*
    round()와 동일한데, 반올림 대신 절삭함.
    */
    select trunc(456.789, 2) from dual;

    select sign(-102) from dual -- 부호 반환
```

3. 날짜 자료형 처리 함수

아래와 같이 sysdate나 날짜 처리 함수 등을 활용할 수 있음.

```
    /*
    sysdate를 사용하면 현재 시점에서의 날짜가 'YY/MM/DD' 형식으로 반환됨.
    sysdate-1, sysdate+1 등으로 전날과 그 다음 날에 대한 값도 얻을 수 있음.
    */
    select sysdate, sysdate+1, sysdate-1 from dual;

    /*
    아래와 같이 round()를 사용하면 월에 대해서 반올림한 결과가 반환됨.
    */
    select round(hiredate, 'month') from emp;

    /*
    두 날짜 사이의 달 수를 반환. 남는 일수는 소수점으로 나타냄.
    */
    select months_between(sysdate, hiredate) from emp;
```

4. 형변환 함수

to_char()를 사용해 날짜 데이터 또는 숫자형을 문자열로 변환할 수 있고, to_number()를 사용해 문자열을 숫자형으로 변환할 수 있음.

```
/*
sysdate 등의 날짜 데이터를 지정한 형식의 문자열로 반환함.
YYYY는 연도를 4자리로, YY는 2자리로 나타냄.
MON, DAY(또는 DY)로 달, 요일을 문자로 나타낼 수 있음.
*/
select to_char(sysdate, 'YYYY-MM-DD'), to_char(sysdate, 'YY-MM-DD DAY')
from dual;

/*
숫자형을 문자열로 변환함.
뒤에 지정한 문자열에서 0에 해당되는 위치에 숫자가 없으면 0으로 채워짐.
뒤에 지정한 문자열에서 9에 해당되는 위치에는 숫자가 없어도 그냥 비워둠.
뒤에 지정한 문자열에 ,가 존재하면 해당 위치에는 ,를 넣어 변환함.
*/
select to_char(1234), to_char(1234, '000000'), to_char(1234, '999,999,999')
from dual;

select to_number('1234') from dual;  -- 해당 문자열을 숫자형으로 변환
```

3.6.2. View

1. View

oracle에서는 아래와 같이 view를 생성할 수 있음.

```
create [or replace] [force | noforce] view <view_name>
as <subquery>
[with check option]
[with read only];
```

or replace를 지정하면 이름이 겹치는 view가 존재할 경우 해당 오류가 나는 대신 view를 대체함. force를 지정하면 base table의 존재 여부와 상관 없이 view를 생성하는데, noforece가 default임. with check point는 앞에서 다룬 것처럼 그 결과를 사용자가 확인할 수 있을 때에만 modification을 허용하도록 함. with read only를 지정하면 해당 view에 대해서는 read만 가능함.

2. Materialized View

Materialized View는 view에 실제 tuple이 존재하도록 하는 기능임.

view expansion 대신 즉시 처리가 가능하므로 더 빠른 처리가 가능하지만, 해당 tuple이 항상 최신 정보를 반영하지는 못할 수 있음. materialized view는 base table이 수정되는 경우나, 특정 주기나 사용자의 요청 등에 의해 최신화됨.

표준 sql에서는 materialized view에 대한 정의를 하고 있지 않음. oracle에서는 아래와 같은 형식으로 materialized view를 정의할 수 있음.

```
create materialized view <view_name>
build [immediate | deferred]
refresh [fast | complete | force]
on [commit | demand]
[ [enable | disable] query write ]
[on prebuilt table]
as <subquery>;
```

build에는 해당 view에 대한 instance를 구하는 시점을 지정함. immediate이면 즉시 instance를 구하고, deferred는 첫 번째 refresh 시점에 구함.

refresh 방식에는 fast, complete, force가 있음. fast는 미리 정의된 materialized view log를 활용해 변

경된 부분만을 refresh하고, complete는 table을 초기화한 뒤 instance를 다시 구하고, force는 fast를 시도하고 불가능하면 complete를 시도함.

refresh 시점은 commit 또는 demand로 지정함. commit은 해당 table에 대한 commit이 존재하면 refresh하고, demand는 주기적으로 또는 사용자 요청에 의해 refresh함.

3.6.3. 기타

1. LOB 자료형
LOB 자료형은 대용량 데이터 저장을 위한 자료형으로, BLOB, CLOB, NCLOB, BFILE이 있음.

앞에서 설명한 것처럼 blob(Binary Large Object)은 큰 용량의 이진 데이터를 저장하는 자료형이고, clob(Character Large Object)은 큰 용량의 문자 데이터를 저장하는 자료형이고, NCLOB는 unicode 문자 데이터를 저장하는 자료형임. 이 자료형들에 대해서는 DBMS에 의해 transaction property가 지원됨. 사용법은 단순한 자료형과 동일함.

BFILE은 구조화되지 않은 binary data를 저장하기 위해 DBMS 밖의 os file system을 활용하는 자료형으로, transaction property가 지원되지 않으며 read 연산만 적용할 수 있음.

2. Set Operation
oracle에서는 union/intersect/minus가 지원됨. 또한 multiset(중복 허용 set) 연산인 union all은 지원되지만 intersect/minus all은 지원되지 않음.

3. Recursive Query
oracle에서는 recursive query를 정의할 때 아래와 같이 recursive 키워드를 작성하지 않고, union 대신 union all을 사용함.

```
With rePrereq(courseID, prereqID) as (
    (
        select courseID, prereqID
        from prereq
    )
    union all
    (
        select recPrereq.courseID, prereq.prereqID
        from recPrereq, prereq
        where recPrereq.prereqID = prereq.courseID
    )
)
select *
from recPrereq;
```

추가로, prereqID와 courseID를 비교하는 경우 lower()등을 활용할 수 있음.

4. Privilege
oracle에서의 privilege는 schema에 대한 privilege인 System Privilege와, 구체적인 object(table, view 등)에 대한 privilege인 Object Privilege가 있음.

system privilege로는 아래와 같은 것들이 있음. system privilege를 grant할 때 with admin option을 작성하여 해당 privilege를 다른 사용자에게 부여할 수 있도록 지정할 수 있음.

admin, alter any index, alter any procedure, alter any table, create any table, create any view, create any synonym, create any sequence, drop any table, drop any view, execute any procedure, insert any table, select any table, select any sequence, update any table, etc.

object privilege로는 아래와 같은 것들이 있음. obejct privilege를 지정할 때는 해당되는 object가 명시되

어야 함. object privilege를 grant할 때 with grant option을 작성하여 해당 privilege를 다른 사용자에게 부여할 수 있도록 지정할 수 있음.

권한	객체 타입
Delete	Table
Execute	PL/SQL package, procedure, function
Index	Table or materialized view
Insert	Table or synonym
References	Table or materialized view
Select	Table, sequence, view, materialized view, synonym
Update	Table

참고로, cluster, index, trigger 등은 object이지만 system privilege로 처리함.

5. Synonym

Synonym(동의어)는 schema.object 형식으로 명시하는 대신 alias처럼 편리하게 사용할 수 있도록 한 기능임.

기본적으로 다른 사용자의 object에 접근하려면 schema.object 형식으로 명시해야 하고, schema를 지정하지 않으면 자신의 schema에서 object를 찾음. synonym을 정의하여 편리하면서도 보안 상 안전하게 다른 사용자의 object를 활용하도록 할 수 있음. 예를 들어, dual도 sys 소유의 table이므로 sys.dual로 명시해야 하지만, public synonym으로 정의되어 있어 dual로도 사용이 가능함.

oracle에서는 아래와 같이 정의함. public을 지정하면 모든 사용자가 해당 synonym을 사용할 수 있음. default는 정의한 사용자만이 사용할 수 있는 것임.

```
create [public] synonym <synonym_name> for <schema>.<object>
```

6. Trigger

oracle에서 trigger는 아래와 같이 작성함. PL/SQL 형식으로 작성하며, 마지막에 /를 작성해야 trigger가 실행됨.

```
create [or replace] trigger <trigger_name>
[before | after | instead of]
[insert | update | delete] on <object_name>
[for each row]
[when <comdition>]
[declare <declaration>]
begin
    <execuation part>
[exception <exception handling part>]
end;
/
```

여기에서는 referencing old row as orow를 작성하지 않아도 단순히 old, new를 바로 사용할 수 있음. 물론 referencing old row as orow를 직접 작성해 이름을 지정할 수도 있음.

or replace를 지정하면 이름이 겹치는 trigger가 존재할 경우 해당 오류가 나는 대신 trigger를 대체함.

begin과 end 사이에 execution part에는 각 문장 끝에 ;를 작성해야 함.

4. Application Development

DBMS에 접근하는 것은 sql을 직접 DBMS에 직접 입력하는 Direct SQL 방식도 가능하지만, database application을 구현할 때 PL에서 sql을 활용하는 방식에는 아래와 같은 2가지 방식 중 하나를 주로 사용함.

1) Static Approach : application code에 sql을 작성하는 방식. embedded SQL 등이 있음. 전처리 과정이

필수적이며, 이에 따른 syntax/authorization 검사 등이 가능해 실제 연산이 빠름.

2) Dynamic Approach : sql api를 활용하는 방식. ODBC/JDBC 등이 있음. flexible하지만 사용하기 어렵고 느림.

과거에는 static approach가 주로 사용되었지만, ODBC/JDBC가 등장한 이후로는 dynamic approach가 주로 사용된다고 함.

4.1. Embedded SQL

4.1.1. Embedded SQL

Embedded SQL(내장 SQL)은 application의 PL 중간에 sql을 직접 삽입하는 방식임. sql 표준에서는 c/c++, java 등 여러 PL에 대한 embedding을 정의함.

embedded sql을 사용한 프로그램은 compile 이전에 전처리(pre-processing)가 적용되어야 함. 전처리기는 syntax/authorization 검사 및 최적화를 수행하고, 해당 sql 문장을 api(함수) 호출로 치환함.

물론 embedded sql은 그 종류에 따라 syntax가 다양한데, 한 예시로 아래와 같은 형식이 존재할 수 있음. 이때 embeded sql 내에서 변수를 사용하는 경우 변수명 앞에 :을 붙여야 함(ex. :fname).

```
EXEC SQL CONNECT ...; // DBMS 연결

EXEC SQL BEGIN DECLARE SECTION ... // 변수 선언
    char fname[FNAME_LEN + 1];
EXEC SQL END DECLARE SECTION;

EXEC SQL <statement>; // sql 문장 실행
```

4.1.2. Cursor

Cursor는 embedded sql의 사용에 따른 impedence mismatch를 해결하기 위한 개념임. cursor는 sql 결과에 대해 각 tuple을 가져와 개별적으로 처리할 수 있도록 함. 여기에서 Impedence Mismatch는 데이터 처리 방식 불일치를 의미함. 예를 들어, 대응되는 자료구조가 존재하지 않는다면 select의 결과를 해당 PL에서 그대로 받아 활용할 수 없음.

cursor는 declare/open/fetch/close 과정을 거쳐 활용됨. declare에서는 sql 문장을 포함하는 cursor를 정의하고, open에서는 정의된 cursor를 실행하여 그 결과를 임시 table에 저장하고, fetch에서는 임시 table에서 tuple을 하나씩 검색하여 가져와 활용하도록 하고, close에서는 임시 table의 결과를 삭제함.

물론 이 또한 언어 등에 따라 그 형식이 달라질 수 있는데, 아래는 c언어에서의 한 예시임.

```
#include <stdio.h>
EXEC SQL define FNAME_LEN 15;
EXEC SQL define LNAME_LEN 15;
main() {
EXEC SQL BEGIN DECLARE SECTION;
        char fname[ FNAME_LEN + 1 ];
        char lname[ LNAME_LEN + 1 ];
EXEC SQL END DECLARE SECTION;
EXEC SQL connect to 'myServer';

EXEC SQL DECLARE demoCursor cursor for
        select firstName, lastName
        from customer
        where :lname < 'C';
EXEC SQL open demoCursor;
for (;;) {
        EXEC SQL fetch demoCursor into :fname, :lname;
        if (strncmp(SQLSTATE, "00000", 5) != 0) break;
        printf("%s %s₩n", fname, lname);
}
if (strncmp(SQLSTATE, "02", 2) != 0)   // 02000 means "no more tuples"
        printf("SQLSTATE after fetch is %s₩n", SQLSTATE);  // shouldn't be printed
EXEC SQL close demoCursor;
EXEC SQL disconnect current;
}
```

cursor의 sql 문장을 실행하면 SQLSTATE라는 변수에 값이 지정되는데, 실행이 성공했으면 '00000'

이, 더 이상 검색되는 tuple이 없으면 '02000'이 지정됨.

cursor를 활용해 update를 하려면 cursor 선언 마지막에 for update를 작성하여 update를 위한 것임을 명시해야 함.

4.1.3. Dynamic SQL

sql 문장은 런타임에 문장이 생성되는 Dynamic SQL과, 런타임 이전에 문장이 생성되는 Static SQL이 있음. dynamic sql에서는 동적으로 sql 문장을 생성하고, compile하고, 실행함.

static sql에 대해서는 sytax/authorization 검사 등이 가능하지만, dynamic sql에 대해서는 이런 작업이 불가능하고, 또한 성능이 떨어짐. 대신 dynamic sql을 사용하면 flexibility를 확보할 수 있음.

dynamic sql의 수행은 아래와 같이 prepare 단계와 execute 단계로 나뉨. query를 나타내는 문자열을 prepare에 넣으면 compile이 수행되는데, 이때 placeholder(변수)를 ?로 사용할 수 있음. 이후 execute로 해당 query를 실행하는데, using에 지정한 변수의 값이 ?에 들어감.

```
stcopy("select name from professor where salary > ?", myText);
mySalary = 1000;

EXEC SQL prepare my1 from :myText;
EXEC SQL execute my1 using :mySalary;
```

또는 아래와 같이 execute immediate로 작성하면 prepare와 execute를 한 번에 수행함. 이 경우 변수는 사용할 수 없음.

```
stcopy("select name from professor where salary > 1000", myText);

EXEC SQL execute immediate from :myText;
```

dynamic sql을 사용하는 경우 SQL injection의 위험성이 존재함.

4.2. ODBC

4.2.1. ODBC

1. ODBC

ODBC(Open Database Connectivity)는 application에서 DB server와 연결하고, server에 query를 전송하고, 그 결과를 tuple 단위로 fetch할 수 있도록 하는 api임.

참고로, ISO 표준으로 제정된 CLI(Call Level Interface)는 ODBC를 기반으로 만들어졌음.

2. Connect

ODBC 2.0에서의 connection 생성은 아래와 같이 environment handle을 할당하고, 이를 활용해 connection handle을 할당하고, connection handle과 server 주소, 사용자와 password를 사용해 DB server와 연결함.

```
int ODBCexample() {
RETCODE     error;
HENV        env;    /* environment handle */
HDBC        conn;   /* connection handle */
SQLAllocEnv(&env);
SQLAllocConnect(env, &conn);
SQLConnect(conn, "db.ssu.ac.kr", SQL_NTS, "myUserID", SQL_NTS
"myPassword", SQL_NTS);

... Do actual work ...

SQLDisconnect(conn);
SQLFreeConnect(conn);
SQLFreeEnv(env);
}
```

ODBC 3.0에서의 connection 생성 시에는 environment/connection/statement handle로 총 3가지 변수 영역을 할당함.

```
SQLHENV env;
SQLHDBC dbc;
SQLHSTMT stmt;
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
&env);       /* allocate environment handle first */
SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);
            /* connect to the data source */
SQLAllocHandle(SQL_HANDLE_ENV, dbc, &stmt);

/* do something with the statement handle e.g. issue sql */

SQLFreeHandle(SQL_HANDLE_STMT, stmt); /* disconnect */
SQLFreeHandle(SQL_HANDLE_DBC, dbc);
SQLFreeHandle(SQL_HANDLE_ENV, env);
```

3. Data Access

ODBC에서는 기본적으로 dynamic하게 data에 접근함. 아래와 같이 SQLExecDirect()로 sql 문장을 수행하고, SQLBindCol()로 해당 결과를 local 변수와 연결한 뒤, SQLFetch()를 사용해 tuple 단위로 그 결과를 받아옴.

```
char deptName[80];
float salary;
int lenOut1, lenOut2;
HSTMT stmt;
char * sqlquery = "select deptName, sum(salary)
                   from professor
                   group by deptName";
SQLAllocStmt(conn, &stmt);

error = SQLExecDirect(stmt, sqlquery, SQL_NTS);   // dynamic SQL
if (error == SQL_SUCCESS) {
      SQLBindCol(stmt, 1, SQL_C_CHAR, deptName, 80, &lenOut1);
      SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0, &lenOut2);
      while (SQLFetch(stmt) == SQL_SUCCESS) {
           printf (" %s %g₩n", deptName, salary);
      }
}
SQLFreeStmt(stmt, SQL_DROP);
```

4. Metadata

ODBC에서는 아래와 같이 DBMS에 대한 metadata를 확인할 수 있는 함수들을 제공함.

```
SQLTables(), SQLColumns(), SQLTablePrivileges(),
SQLStatistics(), SQLSpecialColumns(), SQLProcedures(),
SQLProcedureColumns(), SQLPrimaryKeys(),
SQLForeignKeys(), SQLColumnPrivileges(), et. al
```

5. Transaction

ODBC에서는 default로 각 sql 문장을 transaction 취급하는데, 이런 방식을 AutoCommit이라고 함.

AutoCommit을 off하면 여러 문장으로 transaction을 구성할 수 있음. 이 경우 아래와 같이 commit 또는 rollback을 직접 지정해줘야 함.

- SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)
- Transactions must then be committed or rolled back explicitly by
  - SQLTransact(conn, SQL_COMMIT) or
  - SQLTransact(conn, SQL_ROLLBACK)

6. Conformance Level

COnformance Level은 표준 준수 정도를 나타내는 level로, ODBC에는 아래와 같은 3가지 conformance level이 존재함.

1) Core Level

2) Level 1 : metadata querying을 지원함.
3) Level 2 : catalog에 대한 더 자세한 정보 등을 지원함.

참고로, 여러 개의 schema가 모여 catalog를 구성함.

4.2.2. 기타

1. JDBC
JDBC(Java Database Connectivity)는 java에서 DBMS와 통신하기 위한 api임.

connection을 open하고, statement object를 생성하고, 해당 statement object를 사용해 query를 보내고 이후 결과를 fetch함.

2. SQLJ
SQLJ(The Java Embedding of SQL)는 java에서의 embedded sql을 지원함. 이 경우 PL에 sql을 포함시키는 것이므로 compile 이전에 전처리가 필수적임.

JDBC는 dynamic sql만을 지원하므로 embedded sql이 유의미할 수 있지만, 현재는 잘 사용되지 않고 사장된 기술임.

3. ADO.NET
ADO.NET은 .NET 등에 대해 ODBC/JDBC와 유사하게 DB에 대한 접근을 지원하는 api임.

이는 relational DBMS에 대해서도 접근을 지원하고, OLE-DB, XML, entity 등을 포함하는 non-relational DBMS에 대해서도 접근을 지원함.

4.3. Application Architecture

DB에 접근하는 사용자는 DB를 직접적으로 접근하기보다는 application을 활용하여 접근함. DB를 활용한 application이 가지는 구조를 알아보자.

4.3.1. Application Architecture

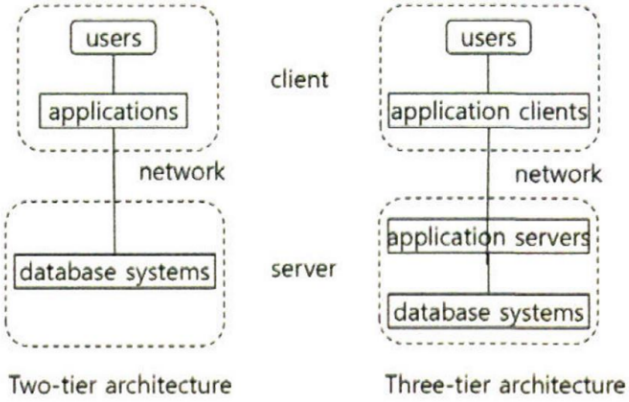
DB를 활용하는 application architecture는 기본적으로 사용자 인터페이스를 담당하는 frontend, 비지니스 로직을 구현하는 middle layer, DBMS와의 연동을 담당하는 backend로 구성됨.

초창기의 architecture는 하나의 중앙 서버의 mainframe이 모든 연산을 처리하며 더미 터미널을 사용하여 여러 사용자의 접속을 관리하는 형태였음. 이후 client-server 구조와 web 구조가 등장했음.

1. Client-Server
client-server 구조는 two-tier 구조와 three-tier 구조로 나눌 수 있음.

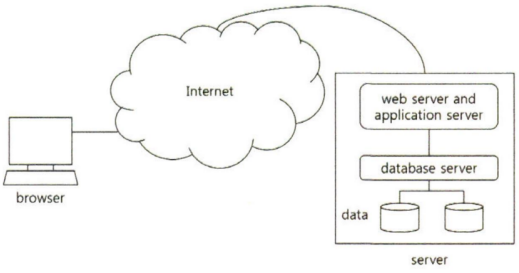
two-tier 구조에서는 application이 client에만 존재하며, server에서는 DB만 관리함. client에서는 ODBC/JDBC 등을 활용해 DB에 접근함. 이 경우 application에 변화가 발생하면 각 client의 application을 모두 수정해야 함.

three-tier 구조에서는 application이 client와 server에 분리되어 존재함. 이때 server의 application은 비지니스 로직을 담당하여, 각 client의 application 대신 server의 application만 조작해 수정할 수 있도록 함. client는 직접적인 DB 접근을 수행하지 않음.



2. Web Server
web server를 활용하는 구조에서는 client가 script language와 HTML 등을 지원하는 web browser를 사용하여 사용자 인터페이스가 구현되고, web server는 JSP, PHP, Python 등을 활용해 DB 로직을 구현함.

주로 아래와 같이 2층 웹 구조가 활용됨. 즉, web server가 비즈니스 로직을 포함하는 application 기능도 함께 수행함. web server와 application server가 분리된 3층 웹 구조도 존재하지만, 이는 오버헤드가 커 잘 사용되지 않는다고 함.



web의 기반인 HTTP protocol은 connectionless이므로, client와 server의 연결을 위해 짧은 text인 cookie를 사용함.

5. SQL Extension

5.1. Procedural Extension

5.1.1. Procedural Extension

1. Procedural Extension
DB 사용자는 SQL/PSM 또는 외부 PL을 사용하여 함수/프로시저를 개발함.

표준 sql은 절차적 확장(Procedural Extension)을 지원하는데, 이를 SQL/PSM이라고 함. 또한 상용 DBMS도 나름의 절차적 확장을 지원하고, 특히 oracle의 경우 PL/SQL이 있음.

2. Stored Procedure
Stored Procedure는 DB에 저장된 프로그램 모듈로, sql 확장을 통해 비즈니스 로직 연산을 구현한 것임. 일반 사용자는 이를 호출하여 활용할 수 있음. 이는 DB object로 관리됨.

stored procedure의 사용은, 비즈니스 로직을 캡슐화할 수 있고, network traffic을 줄일 수 있고, SQL injection을 방지하는 등의 여러 이점이 존재함.

stored procedure를 개발할 수 있는 방식은 DBMS마다 상이함.

5.1.2. 외부 PL을 사용한 함수/프로시저

sql:1999에서는 외부 PL을 사용한 함수/프로시저의 개발을 정의했음. 예를 들어, c 언어를 사용하는 경우 아래와 같이 작성할 수 있음.

이렇게 외부 PL을 사용하는 경우 성능적인 측면에서의 이점은 존재할 수 있지만, 해당 프로그램이 DB에 올라와 실행되다가 오류로 DB가 죽는 문제가 발생하는 등 보안 상 문제가 많음. 이에 따라 코드를 sandbox에서 돌려 안전하게 PL을 사용하거나, 아예 다른 process로 처리하고 IPC를 통해 데이터를 주고받는 식으로 구현하기도 함.

```
- Create procedure deptCountProc
        (in deptName varchar(20), out count integer)
  language C
  external name '/usr/shlee/bin/deptCountProc';
- Create function deptCountProc2(deptName varchar(20))
  returns integer
  language C
  external name '/usr/shlee/bin/deptCount';
```

5.1.3. SQL:1999에서의 함수/프로시저

sql:1999에서 정의하는 함수/프로시저에 대해 알아보자. 당연하게도 이런 정의가 oracle 등 임의의 DBMS에서 호환되는 것은 아님.

1. SQL:1999에서의 함수/프로시저
아래와 같이 함수를 정의할 수 있음. 이때 반환값의 자료형도 명시함.

```
Create function profC(deptName varchar(20)) returns intege
begin
        Declare pCount integer;
        Select count(*) into pCount
        from professor
        where professor.deptName = profC.deptName;
        Return pCount;
end;
```

아래와 같이 table을 반환하는 Table Function을 정의할 수 있음. 반환값의 자료형을 table로 명시함.

```
Create function myProf(deptName varchar(20))
 returns table(   pID char(5),
                  name varchar(20),
                 deptName varchar(20),
                  salary numeric(10,2))
 return table(    select pID, name, deptName, salary
                  from professor
                  where professor.deptName =
                  myProf.deptName);
```

아래와 같이 프로시저를 정의할 수 있음. sql 프로시저는 직접 값을 반환하는 대신 입력과 출력 모두를 인자에 대해서 처리함.

```
Create procedure profC2 (in deptName varchar(20), out pCount integer)
begin
  Select count(*) into pCount from professor
  where professor.deptName = profC2.deptName;
end
```

2. PSM
PSM(Persistent Storage Module)은 sql에서 절차적 확장에 대한 표준을 정의하는 부분임.

PSM에서는 begin과 end를 사용하여 복합문을 작성하는 것을 지원함. begin과 end 사이에 declare로 지역변수를 선언하고, while문, repeat문, for문, if문, 예외 처리 등을 작성할 수 있음. 특히 함수/프로시저 정의에서 이를 활용할 수 있음.

물론 이런 표준 문법을 외울 필요는 없음. 필요한 경우 찾아보자.

5.1.4. PL/SQL

PL/SQL은 oracle에서 지원하는 절차적 확장임.

PL/SQL 프로그램은 아래와 같은 형식을 가짐. 각 문장은 ;로 끝남. 변수 선언, if문, loop문, while문, for문, 예외 처리, 사용자 정의 예외, 프로시저, 함수 등을 활용할 수 있음.

```
[Declare
    declaration _ statements]
Begin
    executable _ statements
[Exception
    exception _ handling _ statements]
End;
```

또한 select가 하나 이상의 tuple을 반환하는 경우 cursor를 사용함. 참고로 for문과 cursor를 함께 사용하면 open, close 등을 명시하지 않아도 자동 처리됨.

또한 함수/프로시저 집합입 패키지를 사용할 수 있는데, 패키지.함수명()으로 해당 함수를 활용함.

6. ER Data Model

지금까지는 sql 문법에 대해 알아봤고, 지금부터는 어떻게 schema를 잘 design할 것인지를 알아봄.

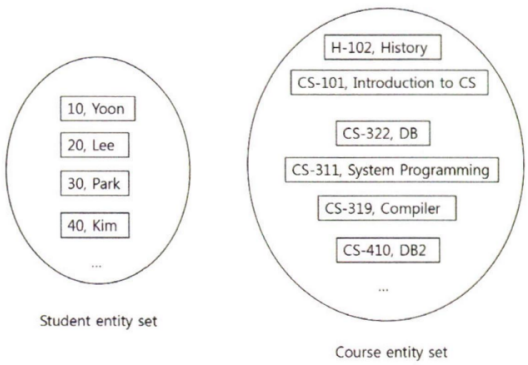
6.1. Entity And Relationship

6.1.1. ER Data Model

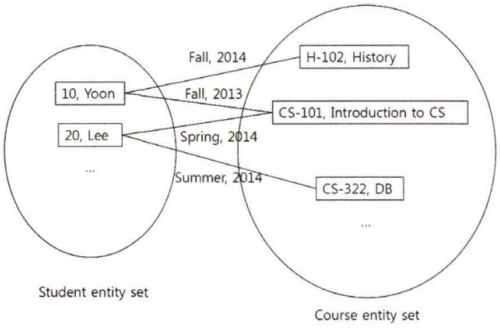
1. ER Data Model
ER Data Model은 DB를 entity와 relationship만으로 구성되는 집합으로 보는 data model임.

ER data model은 DB 설계 시에 가장 많이 활용되는 방식임.

2. Entity/Relationship
Entitiy는 서로 구분이 되는 object임. entity가 가지는 property를 attribute라고 하고, 같은 attribute를 가지는 entity들의 집합을 Entity Type 또는 Entity Set이라고 함.



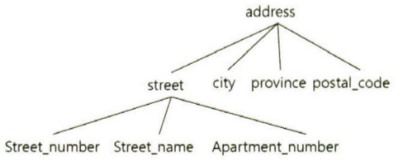
Relationship은 entity들 간의 연관성으로, entity의 나열로도 이해할 수 있음. 당연하게도 하나의 entity set 간에 여러 개의 relationship이 존재할 수 있음. relationship이 가지는 property 또한 attribute라고 하고, 같은 attribute를 가지는 relationship들의 집합을 Relationship Set이라고 함.



relationship의 Degree(차수)는 해당 relationship에 관련되는 entity의 개수로, binary, ternary, quaternary, quinary 등이 존재함. relationship은 대체로 binary임.

3. Attribute
entity와 relationship은 attribute를 가질 수 있음.

attribute는 더 작은 단위로 나눌 수 있는지에 따라 Simple Attribute와 Composite(복합) Attribute로 구분할 수 있음. 아래는 composite attribute의 예시임.

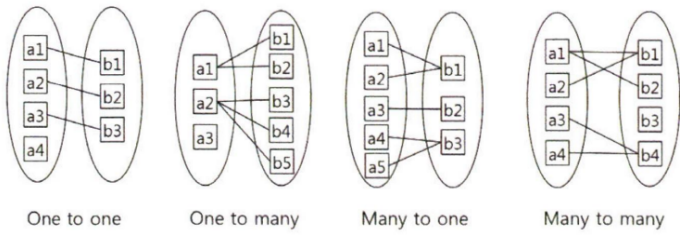


attribute는 attribute가 가지는 값의 개수에 따라 Single Valued Attribute와 Multivalued Attribute로 구분할 수 있음. 예를 들어, multivalued attribute phoneNumber는 "123-4567", "312-1512" 등 여러 개의 값을 가질 수 있음.

Derived Attribute는 다른 attribute의 값을 활용해 그 값을 구할 수 있는 attribute임.

4. Cardinality Constraint
Cardinality Constraint는 ER data model에서 어떤 relationship set에 대해 각 entitiy가 다른 entity와 연결될 수 있는 개수에 대한 제약임.

binary relationship set에서 cardinality constraint로는 아래와 같이 one-to-one(1:1), one-to-many(1:N), many-to-one(N:1), many-to-many(M:N)가 있음. 예를 들어, professor와 student가 many-to-one으로 지정되어 있다면 한 명의 student는 여러 명의 professor와 연관될 수 있음을 의미함.



5. Participation Constraint
Participation Constraint는 어떤 relationship set에 대해, 관련 entity set에서 부분이 참여하냐, 전체가 참여하냐에 대한 constraint임.

1) Total Participation : entity set의 각 entity가 해당 relationship set에서 적어도 하나의 relationship에 참여하고 있는 경우.

2) Partial Pariticipation : entity set 중 일부 entity는 relationship set의 relationship에 참여하고 있지 않은 경우.

6.1.2. Key

ER data model에서 Key는 하나 이상의 attribute의 집합으로, 각 entity를 식별하는 등에 사용되는 요소임. ER data model에서의 super key, candidate key, primary key는 relational DB에서의 key와 그 개념이 동일함.

Super Key(슈퍼 키)는 entity를 유일하게 식별할 수 있도록 하는 attribute의 집합임.

Candidiate Key(후보 키)는 super key에서의 유일성을 유지하면서 minimality를 확보한 key임. 이때 Minimality는 key를 구성하는 attribute 중 하나라도 제거하면 유일성을 잃는 상태를 말함. 이에 따라 서로 다른 candidiate key는 서로 다른 개수의 attribute를 가질 수 있음. 즉, minimaltiy는 minimal(최소한)이지, minimum(최소)이 아님.

Primary Key(주 키)는 실제로 entity 식별에 사용되는 key임. 이때 여러 attribute를 primary key로 하는 경우의 key는 해당 attribute들의 단순 연결(concatenation)이고, 연결한 전체 값은 겹칠 수 없어도 개별 값은 서로 겹칠 수 있음. primary key는 candidtate key 중 하나를 선택해 지정함.

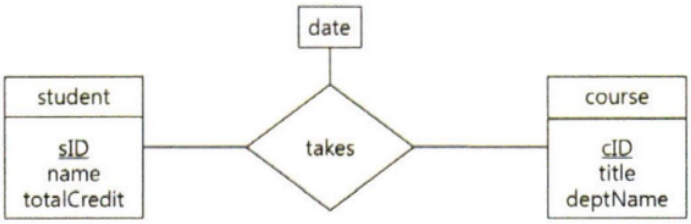
relationship set의 super key는 관련된 entity set의 primary key를 단순 합성하여 구성함. 즉, 특정 entity 사이의 relationship은 하나만 존재할 수 있음. relationship set의 primary key를 지정할 때(relationship을 생성할 때)는 cardinality constraint와 participation constraint(참여 제약)를 고려해야 함.

6.2. ERD

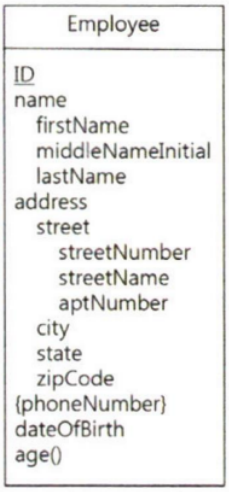
6.2.1. ERD

1. ERD

ERD(Entity Relationship Diagram)은 ER data model을 시각화한 diagram임. 이는 기본적으로 아래와 같이 사각형으로 entity set을, 마름모로 relationship set을, 밑줄로 primary key를 나타냄. 또한 relationship이 attribute를 가지는 경우 사각형으로 표기함.

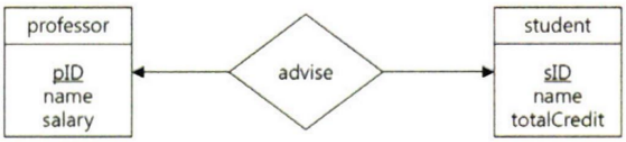


entity set이 가지는 composite attribute는 아래와 같이 들여쓰기로 작성하고, multivalued attribute는 {}로 묶어서 작성하고, derived attribute는 뒤에 ()를 붙여 작성함.

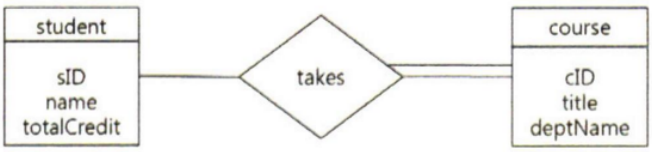


2. Cardinaryity/Participation Constraint

relationship set에 대한 cardinaryity constraint는 아래와 같이 화살표로 나타냄. 화살표가 있는 부분은 one을, 화살표가 없는 부분은 many를 의미함.

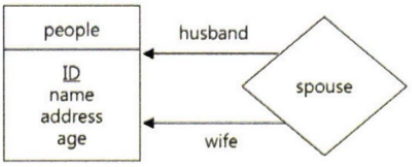


relationship set에 대한 participation contraint는 아래와 같이 single line 또는 double line으로 나타냄. single line은 partial participation을, double line은 total participation을 나타냄.



3. Role

Role은 entity set에 대한 label임. 특히 하나의 entity set이 relationship set에 여러 번 등장할 때, 그 의미를 구분해줘야 함. role은 아래와 같이 line 위에 작성함.



4. Strong/Weak Entity Set

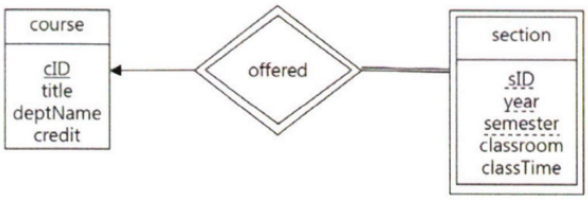
Strong Entity Set은 자체적으로 primary key를 가지고 있는 entity set을 말하고, Weak Entity Set은 자체적으로 primary key를 가지고 있지 않은 entity set을 말함.

weak entity set과 strong entity set 사이에 존재하는 relationship set을 Identifying Relationship Set이라고 하고, 이때의 strong entity set을 Identifying Entity Set이라고 함. 이때 weak entity set은 identifying entity set 없이는 존재할 수 없으므로 weak entity set은 total participation면서 many이고, identiying entity set은 key로 구분하는 역할이므로 one임.

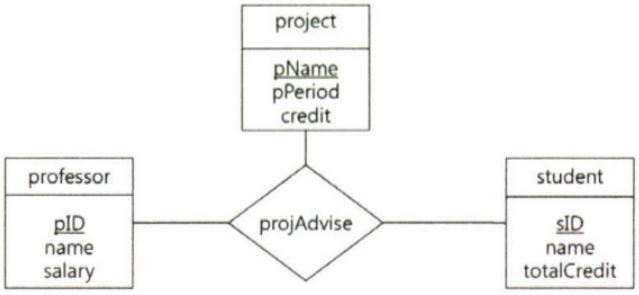
weak entity set은 해당 set 내에서 entity를 구분하기 위한 key를 가지는데, 이를 Discriminator 또는 Partial Key라고 함. 즉, weak entity set의 primary key는 identifying entity set의 primary key와 weak entity set의 partial key의 concatenation임.

identifying entity set의 primary key는 weak entity set에 명시적으로 저장되지 않음. 명시적으로 저장되는 경우 strong entity set이 되며, relationship이 필요하지 않게 됨.

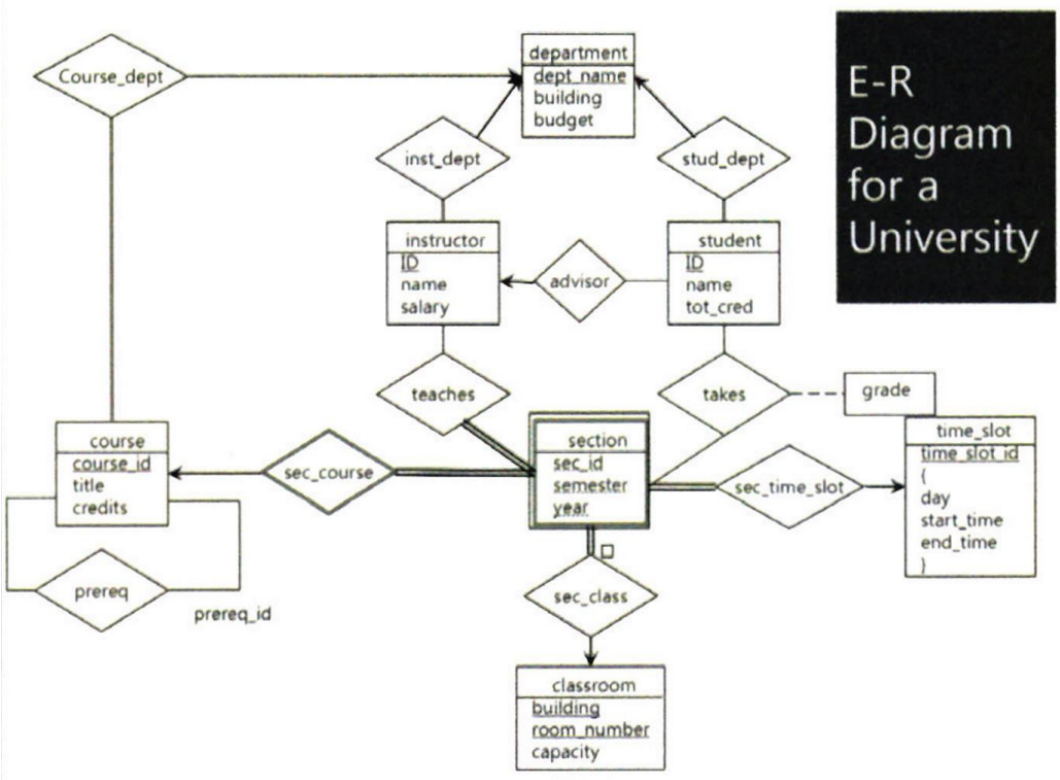
아래와 같이 identifying relation set은 이중 마름모로, weak entity set은 이중 사각형으로 나타냄. partial key는 점선 밑줄로 나타냄.



ternary 이상의 degree를 가지는 relationship set에 대한 ERD도 아래와 같이 동일하게 나타낼 수 있음. 이때 cardinality constraint를 나타내는 경우 one에 해당되는 entity set은 하나만 존재할 수 있음. one이 여러 개 존재하면 의미적 혼동이 존재할 수 있다고 함.



ERD 작성 방식을 종합하면 아래와 같이 대학교 DB에 대한 ERD를 그릴 수 있음.



6.3. Reduction to Relation Schema

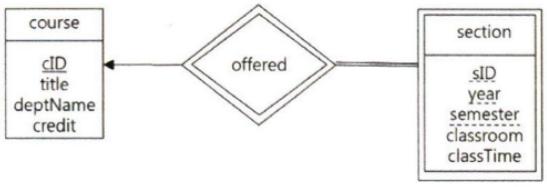
6.3.1. Reduction to Relation Schema

ERD를 작성했으면 이를 relation schema로 변환함. DB design의 목적은 좋은 relation schema를 얻는 것임. 여기에서는 ERD가 어떤 규칙에 따라 relation schema로 변환되는지를 살펴봄.

1. Entity/Relationship Set의 변환
entity/relationship set은 기본적으로 포함하는 attribute와 함께 각각 동명의 table(relation)로 변환됨.

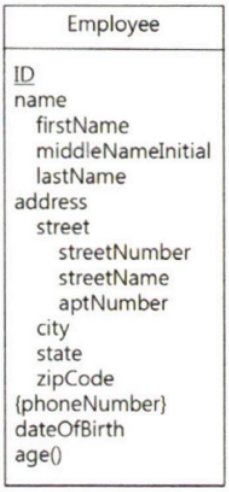
strong entity set은 table로 변환되고, weak entity set은 identifying entity set의 primary key를 포함하는 table로 변환됨. 이에 따라 이미 primary key를 포함하므로 relation schema로의 변환 시에 identifying relationship set은 table로 변환되지 않음. 예를 들어, 아래와 같이 변환됨.

- course(cID, title, deptName, credit)
- section(cID, sID, year, semester, classroom, classTime)
- No table for "offered"



2. Attribute의 변환
기본적으로 entity attribute는 단순히 table attribute로 변환됨.

composite attribute는 flattern되어 attribute로 변환되는데, 이때 composite attribute a가 b, c, d로 구성된다면 b, c, d만 활용되고 a는 제외됨. 예를 들어, 아래와 같은 composite attribute address는 streetNumber, streetName, aptNumber, city, state, zipCode로 변환됨.



multivalued attribute는 해당 attribute를 가지는 단일 table로 변환되는데, 기존 entity set의 primary key를 포함하여 attribute와 primary key를 table의 primary key로 함. 예를 들어, 위의 그림과 같은 entity set의 경우 employPhone(ID, phoneNumber)로 변환되고 이때 primay key는 ID, phoneNumber임.

derived attribute는 단순 변환됨. 또는 특정 data model에서는 method로 변환된다고 함.

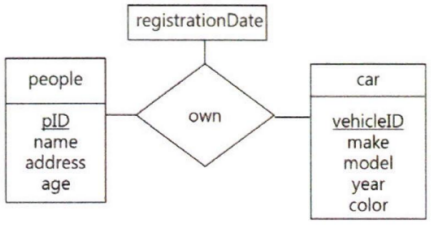
6.3.2. Cardinality Constraint에 따른 변환

relationship set은 cardinality constraint에 따라 아래와 같은 규칙으로 변환이 수행됨.

1. Many-to-Many

many-to-many relationship set은 항상 독립적인 table로 변환됨. 이때 관여하는 entity set들의 primary key들로 primary key가 구성되고, relationship set이 포함하는 attriubte가 table의 attribute로 변환됨.

Example: Own(pID, vehicleID, registrationDate)

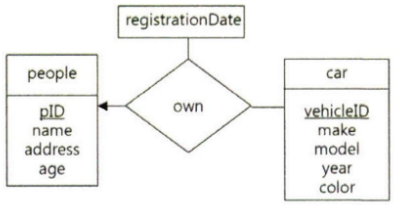


2. Many-to-One(One-to-Many)

many-to-one(one-to-many) relationship set은 독립된 table로 변환하거나, many쪽에 병합시킬 수 있음.

독립된 table로 변환하는 경우 primary key는 entity set들의 primary key 중 하나로 함.

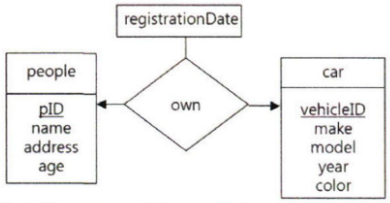
many 쪽에 병합하는 경우, 병합하는 table에서는 relationship set에 해당하는 attribute(다른 entity set들의 primary key, 기존 attribute)를 추가적으로 가지게 됨. 또한 partial participation의 경우 존재하지 않는 값은 null로 처리됨.



- people(pID, name, address, age)
  own(pID, vehicleID, registrationDate)
  car(vehicleID, make, model, year, color)
- people(pID, name, address, age)
  car(vehicleID, make, model, year, color, registrationDate, pID)

3. One-to-One

one-to-one relationship set의 경우 many-to-one에서와 동일하게 처리되는데, 둘 중에 한 쪽을 선택해 many에 해당하는 것으로 처리함.



- people(pID, name, address, age)
  own(pID, vehicleID, registrationDate)
  car(vehicleID, make, model, year, color)
- people(pID, name, address, age)
  car(vehicleID, make, model, year, color, registrationDate, pID)
- people(pID, name, address, age, registrationDate, vehicleID)
  car(vehicleID, make, model, year, color)

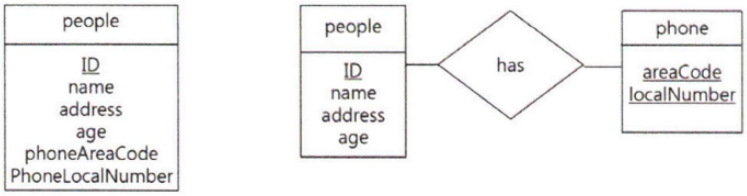
6.4. DB Design Issue

6.4.1. DB Design Issue

DB design 시에는 아래와 같은 고려 사항들이 존재함. 당연하게도 이에 대한 정답은 존재하지 않으며, tradeoff만이 존재함.

1. Entity Set vs. Attribute

정보는 독립된 entity set으로 나타낼 수도 있고, 이미 존재하는 entity set의 attribute로 나타낼 수도 있음. 예를 들어, 아래와 같이 전화번호 정보를 entity set 또는 attribute로 나타낼 수 있음.



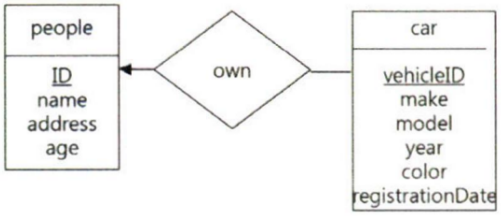
2. Entity vs. Relationship

어떤 정보를 entity로 표현할 수도 있고, relationship으로 표현할 수도 있음. 일반적으로 정보 중 명사는 entity로, 동사는 relationship으로 표현함.

3. Relationship Attribute의 위치

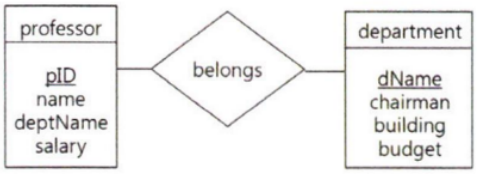
relationship이 가지는 attribute는 cardinality에 따라 그 저장 위치를 결정할 수 있음. 이는 변환에서의 위치가 아니라 design에 따른 위치임. many-to-many인 경우 relationship이 가지고 있어야 하고, one-to-one인 경우 임의의 entity에서도 가지고 있을 수 있고, many-to-one(one-to-many)인 경우 many 쪽에서도 가지고 있을 수 있음.

예를 들어, registrationDate가 relationship own의 attribute인 경우, 이를 아래와 같이 car에 저장할 수 있음.



4. Redundant Attribute

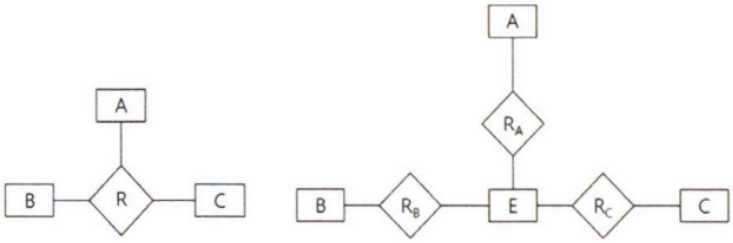
redundant한 attribute는 제거하는 것이 적절함. 예를 들어, 아래와 같은 경우 deptName은 redundant함.



5. Non-binary Relationship

다수의 entity set 사이의 연관성은 non-binary relationship을 활용해야 명확히 표현할 수 있음. 물론 임의의 non-binary relationship은 여러 개의 binary relationship으로 변환할 수 있음. non-binary relationship set R이 있다고 하면, entity set E를 R 대신에 활용하여 기존에 R로 연결되어 있던 entity set들을 E와의 relationship set으로 연결하여 이를 대체할 수 있음. 이때 E는 identifying attribute와 R의 attribute들을 가짐.

- Replace R between entity sets A, B and C by an entity set E, and three relationship sets
  - R_A, relating E and A
  - R_B, relating E and B
  - R_C, relating E and C
- Create a special identifying attribute for E
- Add any attributes of R to E
- For each relationship (a_i, b_i, c_i) in R
  - Create a new entity e_i in the entity set E
  - Add (e_i, a_i) to R_A
  - Add (e_i, b_i) to R_B
  - Add (e_i, c_i) to R_C



하지만 이렇게 binary relationship으로 변환하는 경우, 기존의 constraint를 완전히 표현하지 못해 정보가 손실될 수 있음.

6.5. Extended ER Feature

6.5.1. Specialization/Generalization

1. Specialization/Generalization

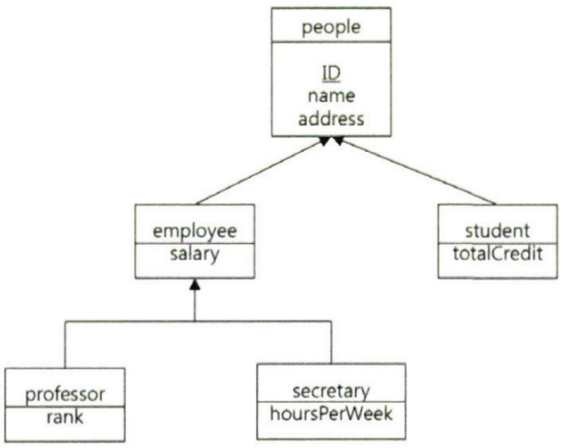
ER data model은 객체지향의 inheritance와 유사한 기능으로, entity 간의 계층적 관계를 정의하는 specialization/generalization을 제공함. 이 두 개념은 단순히 반대 과정임.

1) Specialization : 상위 entity를 더 구체적인 하위 entity로 나누어 특정 attribute를 추가하는 작업. 즉, top-down design 과정임.

2) Generalization : 하위 entity들에서 공통적 attribute를 추출해 상위 entity를 구성하는 작업. 즉, bottom-up design 과정임.

이렇게 구성된 entity 계층에서, 하위 entity는 상위 entity의 attribute와 participation constraint를 상속받음.

이때 하나 이상의 specialization/generalization이 존재할 수 있음. 즉, 여러 개의 부모를 가지는 다중 상속이 가능함.



2. Constraint on Specialization/Generalization
specialization/generalization에 대해 지정할 수 있는 constraint로는 아래와 같은 것들이 있음.

1) 특정 entity에 대해, 어떤 entity가 해당 entity의 하위 entity가 될 수 있는지에 대한 constraint. condition-defined 또는 user-defined로 지정할 수 있음.

2) 특정 entity에 대해, 얼마나 많은 하위 entity가 존재할 수 있는지에 대한 constraint. disjoint(하나만 가능) 또는 overlapping(여러 개 가능)으로 지정할 수 있음.

3) 특정 entity에 대해, 하위 entity가 존재해야 하는지에 대한 constraint. Completeness Constraint라고 함. total(반드시 존재해야 함) 또는 partial(존재하지 않아도 됨)으로 지정할 수 있음. partial이 default임.

3. Reduction to Relation Schema
specialization/generalization은 relation schema로의 변환 시에 아래와 같은 두 가지 방법 중 하나를 활용할 수 있음.

1) 하위 entity에 대해서 해당 entity 자체에 속하는 attribute와 상위 entity의 primary key만으로 table을 구성함.

이 경우 상위 entity에 속하는 attribute를 활용하려면 매번 해당 상위 entity까지 접근해야 함.

2) 하위 entity에 대해서 해당 entity 자체에 속하는 attribute와 상위 entity의 attribute까지 모두 활용하여 table을 구성함.

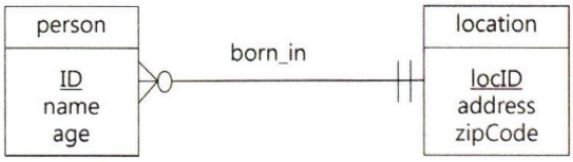
이 경우 total constraint가 지정되어 있다면 상위 entity에 대한 table은 생략하고 하위 entity에 대한 table만을 사용하여 view를 제공할 수 있음. 물론 이때 상위 entity와 관련된 데이터 constraint를 지정할 수는 없음.

6.6. Notation

ERD를 나타내는 notation으로는 여러 가지가 있음.

Alternative ER Notation은 최초에 ERD가 등장하면서 함께 제시된 notation으로, 학계에서 주로 활용됨.

Crow's Foot은 까마귀 발 모양으로 many를, 수직 바로 one을 나타내는 notation으로, 실무에서 주로 사용됨.



(min, max) Notation은 relationship에 대해 entity의 participate 횟수를 명시하는 notation임.

UML(Unified Modeling Language)에서도 나름의 notation을 제공함.

7. DB Design Theory

ERD를 만든 뒤 변환해서 relational schema를 생성했으면, 이후 여기에서 다루는 DB design theory를 활용해 개선함. 여기에서의 개념과 설명은 relational schema에 대한 것임.

7.1. Functional Dependency Theory

어떤 schema에 대한 good/bad 여부를 판단하고, bad schema라면 적절히 decompose하는 방법을 알아보자.

7.1.1. Bad Schema

아래와 같은 bad schema들이 존재할 수 있음. 이는 update/insert/delete 작업에 anomaly가 존재하는지로 확인할 수 있음.

1. Bad Schema 1 : 종속성이 존재하는 경우
아래의 예시는 이는 course와 department를 natural join하여 얻은 것으로, 3가지 종류의 anomaly가 존재함.

cID	title	deptName	credit	chairman	building	budget
202	Java	CS	2	Lee	IT building	67000
203	Data Structure	CS	3	Lee	IT building	67000
301	Databases	CS	3	Lee	IT building	67000
241	Logic Circuit	EE	2	Han	HN Eng	82500
341	LAN	EE	3	Han	HN Eng	82500
342	Electronics	EE	3	Han	HN Eng	82500
222	Computer Art	Media	3	Paik	IT building	55500
223	Game Theory	Media	3	Paik	IT building	55500

1) update anomaly : 특정 attribute에 종속적인 attribute들이 존재하여, 해당 attribute와 관련된 값이 update되는 경우 종속적인 attribute들 모두에 대해 update해야 함. 이 예시에서 chairman, building, budget이 deptName에 종속적인 경우, chairman 값에 수정이 발생하면 해당 deptName과 관련된 모든 chairman 값을 수정해야 함.

2) insert anomaly : key가 적절히 지정되지 않아, 특정 정보를 insert할 수 없음. cID(course ID)가 key이므로, 새로운 department가 생겨도 해당 department에 course가 생성되지 않으면 table에 나타날 수 없음. 과목이 없어도 department가 생성되었다면 table에 반영되는 것이 바람직함.

3) delete anomaly : key가 적절히 지정되지 않아, 특정 정보에 대한 delete 시에 다른 주요 데이터까지 delete됨. insert anomaly와 같은 맥락으로 어떤 department에 대해 마지막으로 남은 course를 delete하는 경우, course 하나가 delete되는 것인데 department가 delete될 수 있음. 과목이 없어도 department가 존재하는 것이 바람직함.

2. Bad Schema 2 : 관련성이 없는 경우
서로 관련이 없는 course와 room을 join해 생성한다고 하자. 이 예시에서는 그 결과로 myTable(cID, title, deptName, credit, roomID, building, capacity)가 도출되었다고 가정함. 이때 primay key는 (cID, roomID)임.

1) update anomaly : 하나의 cID와 서로 다른 roomID를 가지는 tuple들로 table이 구성되므로, course에 대한 수정 시에 모든 관련 tuple들을 전부 수정해야 함.

2) insert anomaly : cID와 roomID 모두가 있어야 입력할 수 있음.

3) delete anomaly : room을 삭제했는데 관련이 없는 course까지 함께 삭제됨.

이렇게 관련되지 않은 데이터들끼리 묶이는 경우가 더 심각하다고 함. course에 대한 정보를 알아도, 이와 관련없는 room에 대한 정보가 있어야 이를 적절히 처리할 수 있음.

7.1.2. Functional Dependency

Functional Dependency(FD)은 어떤 일부 attribute의 값이 다른 attribute의 값이 유일하게 결정되는 종속성을 말함.

더 구체적으로, relational schema R에 대해 attribute $\alpha \subseteq R$와 $\beta \subseteq R$가 존재할 때, 임의의 두 tuple의 α 값이 같으면 β 값도 같은 경우 β가 α에 대해 FD를 가진다고 함. 또한 이는 $\alpha \rightarrow \beta$로 표기함.

superkey는 모든 속성과 관련된 FD를 가지는 attribute set이고, candidate key는 superkey의 성질을 만족하면서 해당 성질의 subset이 존재하지 않는 attribute set임(minimal superkey).

FD는 특정 tuple에 대한 유효성 검사나, table의 constraint 지정 등에 사용할 수 있음.

7.1.3. Functional Dependency Theory

FD를 활용할 때에 고려할 수 있는 이론들을 살펴보자.

1. Trivial Functional Dependency
Trivial Functional Dependency는 임의의 table의 tuple에 대해서 항상 성립하는 FD를 말함. 즉, 무의미한 FD임.

예를 들어, attibute set α가 있을 때, α와 그 subset 사이에 존재하는 FD는 항상 성립하므로 trivial FD임.

2. Armstrong's Axiom
Armstrong's Axiom(공리)은 FD로부터 다른 FD를 유추할 수 있는 inference rule로, 아래와 같은 3가지 rule로 구성됨.

1) Reflexivity : if $\beta \subseteq \alpha$ then $\alpha \rightarrow \beta$. 즉, trivial을 나타냄.
2) Augmentation : if $\alpha \rightarrow \beta$ then $\gamma\alpha \rightarrow \gamma\beta$. 즉, 양쪽에 동일한 attribute를 추가할 수 있음.
3) Transivity : if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ then $\alpha \rightarrow \gamma$. 즉, 연쇄적인 특징을 나타냄.

당연하게도 FD의 대상은 attribute의 set이므로 $\alpha\gamma$와 $\gamma\alpha$가 동일하고, 동일한 attribute가 여러 개 존재하면 단순히 하나로 표기함.

arstrong's axiom을 통해 증명될 수 있는 추가적인 inference rule로는 아래와 같은 것들이 있음.

1) Union : if $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ then $\alpha \rightarrow \beta\gamma$. 즉, 동일한 attribute에 대한 FD는 합쳐질 수 있음.
2) Decomposition : if $\alpha \rightarrow \beta\gamma$ then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$. 즉, 동일한 attribute에 대한 FD는 분해될 수 있음.
3) Pseudotransitivity : if $\alpha \rightarrow \beta$ and $\beta\gamma \rightarrow \delta$ then $\alpha\gamma \rightarrow \delta$. 즉, attribute는 해당 attribute를 결정하는 attribute로 대체될 수 있음.

armstrong's axiom은 sound하고 complete하다는 것이 증명되어 있음. Sound(건전)하다는 것은 armstrong's axiom을 통해 생성되는 FD가 항상 유효함을 뜻함. Complete(완전)하다는 것은 armstrong's axiom을 통해 모든 유효한 FD를 생성할 수 있음을 뜻함.

3. Closure of FD Set
어떤 FD에 대한 set인 F가 존재할 때, F에 속하는 FD로부터 유추할 수 있는 모든 FD의 set을 해당 FD set에 대한 Closure라고 하고, F^+로 표기함. armstrong's axiom은 sound하고 complete하므로 FD set에 대한 closure를 구할 수 있음.

어떤 FD set F에 대해 armstrong's axiom으로 closure를 구하려면, reflexivity와 augmentation을 적용해 얻은 FD를 F에 추가하고, F의 FD들 중 2개를 뽑아 transitivity를 적용할 수 있는 것은 적용해 F에 추가하는 과정을 반복할 수 있음. 이를 통해 항상 closure를 구할 수 있지만, 당연하게도 이는 실질적으로는 사용되지 않는 이론적 방법임.

4. Closure of Attribute Set
어떤 attribute set A와 그에 대한 FD set F가 있다고 하면, F를 활용하여 A가 functional하게 결정할 수 있는 모든 attribute들을 해당 attribute set의 Closure라고 하고, A^+로 표기함. 당연하게도 이는 FD set에 대한 closure와는 아예 다른 개념임.

attribute set A의 closure는, 단순히 A에 속하는 원소들에 대한 FD를 활용해 attribute를 추가하는 과정을 반복하여 얻을 수 있음. 예를 들어, 아래와 같이 R에 대한 closure ABCGHI를 구할 수 있음.

R = (A, B, C, G, H, I)
- F = {A → B A → C CG → H
  CG → I B → H}

$(AG)^+$
- closure = AG
- closure = ABCG (A → C and A → B)
- closure = ABCGH (CG → H and CG ⊆ AGBC)
- closure = ABCGHI (CG → I and CG ⊆ AGBCH)

attribute set의 closure는 아래와 같이 활용할 수 있음.

1) super key인지 확인
어떤 attribute set A의 closure가 해당 relation의 전체 attribute를 포함한다면, A는 super key임. 당연하게도 이때 A의 subset이 super key가 아니면 A는 candidate key임.

2) FD의 유효성 검증
$\alpha \rightarrow \beta$라는 FD에 대한 유효성은, attribute α의 closure에 β가 존재하는지를 확인하여 검증할 수 있음.

3) FD에 대한 closure 구하기
attribute α의 closure가 가지는 attribute β가 존재한다면, FD $\alpha \rightarrow \beta$를 얻을 수 있음.

5. Canonical Cover
Canonical Cover는 주어진 FD set과 동일한 표현력을 가지면서(closure가 동일함.), 가장 최소한의 attribute 및 FD를 가지는 FD의 set으로, F_c로 표기함. 즉, 어떤 FD set에 대한 minimal로, 당연하게도 canonical cover가 되도록 DB를 design하는 것이 이상적임.

즉, FD set F에 대한 canonical cover F_c는 아래와 같이 3가지 조건을 만족시키는 FD set임.

1) $F^+ = F_c^+$
2) F_c의 FD는 extraneous(불필요한) attribute를 가지지 않음.
3) F_c의 FD들이 가지는 각 left side는 유일(unique)함.

canonical cover는 union을 활용해 FD들을 합치고, extraneous attribute를 제거하는 과정을 반복하여 구할 수 있음. 아래와 같은 예시가 존재함. 구체적인 알고리즘은 복잡하여 다루지 않음.

- R = (A, B, C)
  F = {A → BC, B → C
  A → B, AB → C}
- Combine A → BC and A → B into A → BC
  - Then we have {A → BC, B → C, AB → C}
- A is extraneous in AB → C
  - Check if the result of deleting A from AB → C is implied by the other dependencies
    - Yes, in fact B → C is already there!
  - Then we have {A → BC, B → C}
- C is extraneous in A → BC
  - Check if A → C is logically implied by A → B and the other dependencies
    - Yes, using transitivity on A → B and B → C
- The canonical cover is {A → B, B → C}

7.2. Normalization

7.2.1. Normalization

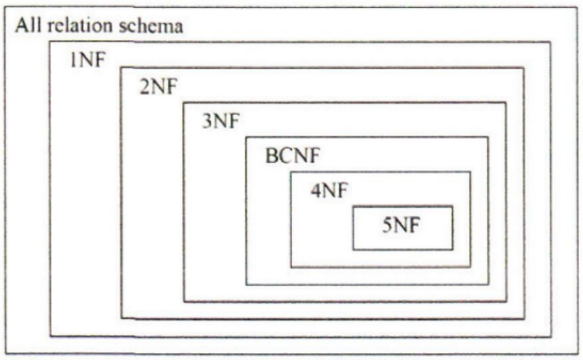
Normalization은 DB design 과정에서 어떤 schema가 good schema인지 검사하고, good schema가 아니면 decomposition을 수행하여 더 작은 규모의 good schema로 변환하는 작업을 의미함.

normalization은 normal form에 따라 진행됨. 즉, normal form의 특정 단계임을 판단하거나, 특정 단계로 decomposition함.

7.2.2. Normal Form

Normal Form(NF, 정규형)은 DB table이 특정 구조적 특징을 만족하도록 정의된 단계별 규칙으로, 1NF, 2NF, 3NF, BCNF, 4NF, 5NF로 총 6개의 단계로 나뉨.

각 NF 단계들은 서로 subset 관계로, 1NF로 갈수록 큰 범위의 set임. 어떤 schema가 특정 단계라고 하는 것은 적용되는 단계 중 가장 높은 것을 의미함. 실세계에서는 3NF 또는 BCNF를 사용하는 것이 권장됨.



1NF부터 BCNF까지는 FD를 사용하여 정의됨. 4NF와 5NF는 잘 사용되지 않으므로 생략함.

Prime Attribute는 임의의 candidate key에 속하는 attribute이고, Non-prime Attribute는 어떤 canndiate key에도 속하지 않는 attribute임.

Complete FD는 attribute set의 subset이 아닌 전체에 대해서만 종속성이 존재하는 FD를 말함. 즉, complete FD인 $\alpha \rightarrow \beta$에서 β는 α의 subset에 대해서는 종속성을 가지지 않음. complete FD가 아닌 FD는 Partial FD라고 함.

Transitive FD는 armstrong's axiom 중 transitivity에 의한 FD로, $\alpha \rightarrow \beta$와 $\beta \rightarrow \gamma$가 존재하는 경우 α와 γ 사이에 존재하는 종속성을 말함.

1. 1NF

1NF(First Normal Form)은 모든 attribute의 domain이 atomic한 schema임. 이는 relational data model에서의 기본 정의의므로, 모든 relational schema는 이를 만족함.

앞에서도 정리했지만, atomic하다는 것은 더 작은 단위로 나눌 수 없음을 의미함. atomic하지 않은 domain으로는 set, list, bag, composite attribute 등이 있음.

2. 2NF

2NF는 1NF 중에서 모든 non-prime attribute가 각 candidate key에 대해 complete FD를 가지는 schema임. 즉, 해당 candidate key는 모든 attribute에 대해 더 작게 나뉠 수 없음.

예를 들어, 아래에서 위의 table은 2NF가 아니고, 그것을 분해한 아래의 table은 각각 2NF임.

R(SSN, pNumber, hours, eName, pName, pLocation)
SSN pNumber → hours
SSN → eName
pNumber → pName pLocation

decomposed into

R1(SSN, pNumber, hours)
R2(SSN, eName)
R3(pNumber, pName, pLocation)

3. 3NF

3NF는 2NF 중에서 모든 non-prime attribute가 각 candidate key에 대해서 transitive FD를 가지지 않는 schema임. 즉, $\alpha \rightarrow \beta$에서 α가 super key(다른 것들을 유일하게 결정)이거나, β가 prime attribute여야 함.

transitive FD를 가지지 않는다는 것은 직접 의존한다는 것으로 이해할 수 있음.

예를 들어, 아래에서 위의 table은 3NF가 아니고, 그것을 분해한 아래의 table은 각각 3NF임.

myEmpDept(SSN, Ename, Bdate, Addr, D#, Dname, Dmgr)
SSN → Ename Bdate Addr D#
D# → Dname Dmgr

Since Dname and Dmgr are transitively dependent on SSN
(not in 3NF)

ED1(SSN, Ename, Bdate, Addr, D#)
ED2(D#, Dname, Dmgr)

4. BCNF

BCNF(Boyce/Codd NF)는 3NF 중에서 $\alpha \rightarrow \beta$에서 α가 super key(다른 것들을 유일하게 결정)인 schema임. 즉, β가 prime attribute인 경우를 제외한 것임.

이때 F^+가 아니라 주어진 FD set F에 대해서만 super key인지를 확인하면 됨. F에 대해서 조건이 성립하면 F^+에 대해서도 성립한다고 함.

attribute의 개수가 2개인 schema는 항상 BCNF이므로, 더 이상 정규화할 필요가 없음. 아래와 같이 경우를 나누어 증명할 수 있음. attribute는 각각 A, B라고 하자.

1) FD가 없는 경우, (A, B)가 candidate key가 되고, BCNF임.
2) $A \rightarrow B$가 성립하고, $B \rightarrow A$가 성립하지 않는 경우, A가 candidate key가 되고 BCNF임.
3) $B \rightarrow A$가 성립하고, $A \rightarrow B$가 성립하지 않는 경우, B가 candidate key가 되고 BCNF임.
4) $A \rightarrow B$와 $B \rightarrow A$가 모두 성립하는 경우, FD의 왼쪽이 모두 super key이므로 BCNF임.

어떤 schema에 대한 NF를 판단하는 것은 아래의 과정을 거침. 이때 속성을 화살표와 함께 그려놓으면 쉬움.

1) 모든 candidate key를 찾음. 이때 어떤 attribute를 결정해 주는 FD가 없다면 해당 attribute는 그대로 candidate key에 추가하면 됨.
2) prime/non-prime attribute를 구분함.
3) NF를 판별함.
-> 2NF. 모든 non-prime attribute이 각 candidate key에 대해 complete FD를 가짐.
-> 3NF. FD에서 왼쪽이 super key이거나, 오른쪽이 prime attribute임.
-> BCNF. FD에서 왼쪽이 super key임.

key가 attribute 하나를 가질 때는 일반적으로 문제가 잘 안 되고, 키가 2개 이상인 경우에 문제가 주로 발생함.

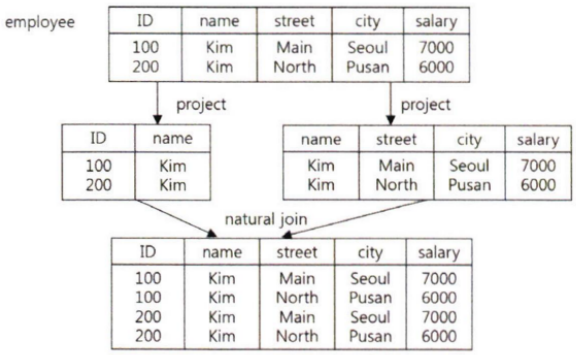
7.2.3. Decomposition

Decomposition은 bad schema를 더 작은 크기를 가지는 여러 개의 good schema로 분해하는 과정임. 이때 decomposition에 의해 생성된 각 schema는 good schema여야 하고, decomposition 과정은 lossless 여야 함. 또한 FD를 보존하는 것이 이상적임.

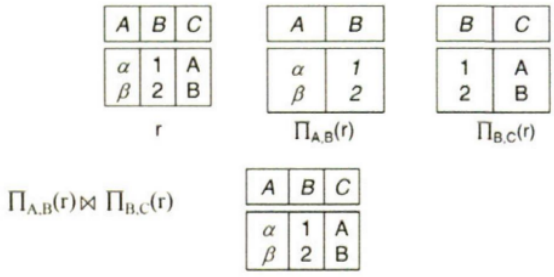
1. Lossy vs. Lossless Decomposition

여기에서 decomposition 시에는 기존 table에 projection 연산을 적용해 분해된 table을 구성함.

Lossy Join Decomposition 또는 Lossy Decomposition은 분해 결과로 생성된 두 table을 natural join 하면 원본 table에는 없던 tuple인 Spurious tuple이 생성되는 decomposition임.



Lossless Join Decomposition 또는 Lossless Decomposition은 분해 결과로 생성된 두 table을 natural join하면 원본 table이 생성되는 decomposition임.



decomposition이 lossless이려면 분해된 두 table의 공통 attribute가 두 table 중 적어도 하나에서 super key여야 함. 물론 이는 2개로 분해되는 상황에 대한 조건이고, 3개 이상(일반적이지 않은 상황.)에 대해서는 또 다른 조건들이 존재한다고 함.

2. FD Perservation
table R이 $R_1, \cdots, R_n$으로 분해될 때, 기존 table의 FD set closure F^+ 중에서 R_i에 속하는 attribute만을 다루는 FD set을 F_i라고 하자. decomposition에 의해 FD가 보존되려면, 각 F_i의 합집합의 closure가 F^+와 동일하면 됨. 즉, 분해되는 각 table이 가지는 attribute 간의 FD가 존재하지 않아야 함.

FD가 보존되지 않는 decomposition에 대해서는 FD를 확인하기 위해 natural join을 해야 하므로, 비용이 많이 듦.

FD preservation을 확인하는 구체적인 알고리즘은 아래와 같음. 이는 attribute set의 closure를 활용함.

- result = α
  while (changes to result) do
  for each R_i in the decomposition
  $t = (result \cap R_i)^+ \cap R_i$
  $result = result \cup t$
- If result contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved

아래는 decomposition의 예시임.

- R = (A, B, C)
  F = {A → B, B → C}
  - Can be decomposed in two different ways
- R_1 = (A, B), R_2 = (B, C)
  - Lossless-join decomposition: $R_1 \cap R_2$ = {B} and B → BC
  - Dependency preserving
- R_1 = (A, B), R_2 = (A, C)
  - Lossless-join decomposition
  - Not dependency preserving
    (cannot check B → C without computing $R_1 \bowtie R_2$)

7.2.4. Decomposition과 NF

decomposition 이후에는 분해된 각 table이 good schema를 가지는지 확인해야 함. 또한, 어떤 단계의 table로 분해할 지도 잘 결정해야 함.

임의의 table에 대해 BCNF 또는 3NF로 분해하는 것이 가능함. 이때 BCNF는 더 이상적이지만 분해 시에 FD가 보존되지 않을 수 있고, 3NF는 덜 이상적이지만 FD가 보존됨. 실제 DB 설계 시에는 이 두 NF를 선택적으로 활용한다고 함.

1. BCNF 판단 방법
어떤 table이 BCNF인지는 F^+ 전체가 아닌, 주어진 FD set F에 대해서만 super key인지를 확인하면

됨. 하지만 table이 decomposition되는 경우에는, 주어진 FD가 소실될 수 있음. 이에 따라 분해된 table에 대해 존재하는 FD만을 활용해서 BCNF인지를 판단할 수 없는 경우가 존재함. 분해된 table이 BCNF인지는 아래와 같은 방법으로 확인할 수 있음.

1) F^+에서 분해된 table의 attribute에 해당하는 모든 FD를 가져와 확인함. 당연하게도 이 과정은 실용적이지 않음.

2) 아래와 같이 attribute set의 closure를 활용해 확인함.

- Or use the original set of dependencies F that hold on R_i but with the following test
  - For every set of attributes $\alpha \subseteq R_i$, check that α^+ either includes no attribute of R_i-α or includes all attributes of R_i
  - If the condition is violated by some $\alpha \rightarrow \beta$ in F, the dependency $\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$ can be shown to hold on R_i and R_i violates BCNF
  - We use above dependency to decompose R_i further

아래의 예시와 같이, decomposition을 통해 BCNF인 table들을 얻는 경우 항상 FD가 보존되는 것은 아님. 대신 3NF로 분해하는 경우에는 항상 FD를 보존할 수 있음.

R = (J, K, L)
F = {JK → L
L → K }

- Candidate keys: JK and JL
- R is not in BCNF
- Any decomposition of R fails to preserve JK → L

2. 3NF 판단 방법

3NF로의 decomposition은 값의 중복에 의한 anomaly가 존재할 수 있지만, 특정 기법들을 활용하면 항상 FD가 보존되도록 할 수 있음. 즉, 3NF로의 decomposition은 주로 FD의 보존이 필요한 경우 사용되고, 이 경우 모든 FD는 분해된 각 table에 대해서 활용이 가능함.

분해한 table이 3NF인지를 확인하는 경우에도 F^+가 아닌 주어진 F에 대해서 검증을 수행하면 됨. 이때 앞에서 다룬 것처럼 FD $\alpha \rightarrow \beta$에 대해서 α가 super key이거나, β가 prime attribute인지를 확인해야 함. 이 과정은 모든 candidate key를 찾아 확인해야 하므로 비용이 많이 드는 NP-hard 연산임.