Normalization In Database

Database Survey Report

Prof. Jeonyoung Lee

TEAM : JOE

Computer Science & Engineering
9725002 Joon-Myung Kang
9725018 Taeho Oh
9725029 Hyunho Jung

email : joe@ohhara.postech.ac.kr

1999년 5월 8일

목 차

제 1 절 Introduction

Normalization plays a key role of Relational Database Design. The general purpose of database design are

- Elimination of redundant data storage.
- To avoid certain update anomalies
- Close modeling of real world entities, processes, and their relationships.
- Structuring of data so that the model is flexible.

To make a set of relational schemes which satisfy these is a final purpose. One way is to design a proper normal form. We need some information to know whether Relational schemes are one of Normal Form. And, those information is about real-world we are modeling, a set of constraint called data dependency. We surveyed the kind of these dependency and normal form. Furthermore, we will explain which constraint is in each normal form. Until recently, data base sepecialists have lacked a comprehensive technique for logical data base design. As a result, data base design has historically been an intuitive and often haphazard process. However, the techniques of normalization now provide a foundation for logical data base design. Let's define normalization simply.

1 **Normalization**

Normalization is the *analysis of functional dependencies between attributes (or data items)* and a formal process for *determining which fields belong in which tables in a relational database.*

Normalization is to make a set of entity specification[1] to satisfy conditions of normal form defined. Until now, there are six normal form as 1NF(First Normal Form), 2NF, 3NF, BCNF(Boyce-Codd Normal Form), 4NF, 5NF. These normal form will be explained from next section.

제 2 절 Need of Normalization

2.1 절 Merit of Normalization in Database Design

Relational schema created by Normalization can improve data integrity because of decreasing data redundancy and update anomalies. We will explain why data redundancy and update anomalies is a problem, show examples that problems are solved in separated table.

2.2 절 Data redundancy

[1] entity specification is to name attribute of entity. That is, one attribute can be included in several entity specification. This will be explained on update anomaly.

schema design의 목표중에 하나는 base relations(files)이 차지하는 storage space를 최소화하는 것이다. 한 가지 방법은 attributes을 relation schema들로 grouping함으로써 이런 목표를 달성할 수 있다. 우선 고용자와 분야를 나타내는 EMP_DEPT Relational schema와 이것을 둘로 쪼개 놓은 EMPLOYEE, DEPARTMENT 각각의 table들을 비교해보자. EMP_DEPT에서는 (DNUMBER, DNAME, DMGRSSN)(department number, 이름등...)에 관한 attribute value들이 그 department에서 일하는 모든 employee에 반복되는 것을 볼 수 있다. 반대로 DEPARTMENT table에서는 각 department에 대해서 한 번만 나타난다. 단지 DNUMBER만 EMPLOYEE table의 각 tuple들에 나타나고 있다.

2.3 절 Update anomalies

EMP_DEPT나 EMP_PROJ와 같은 relation들을 base relation으로 사용하면 발생하는 또 다른 문제 중에 하나는 Update anomalies이다. Update anomalies는 insertion anomalies, deletion anomalies, modification anomalies로 구분할 수 있다. 각각에 대해서 차례로 알아보자.

1. Insertion Anomalies
   먼저, EMP_DEPT에 새로운 employee tuple을 insert하는 것을 생각해 보자. 각 tuple은 employee가 일하는 department에 대한 attribute들을 포함하거나, 만약 employee가 속한 department가 없다면 null값을 포함해야 한다. 새로운 employee가 일하는 department의 DNUMBER가 5라면 EMP_DEPT에 있는 tuples들중 DNUMBER가 5인 tuple들의 DNAME, DMGRSSN 와 일치하는 attribute value들을 insert해야 한다. insert를 할 때마다 이런 것을 확인하는 것은 쉬운 일이 아닐 것이다. 하지만 fig2의 EMPLOYEE, DEPARTMENT table을 따로 만들 경우 EMPLOYEE에 5만 insert해 주면 되므로 앞에서와 같은 걱정은 안해도 된다.
   다음은 employee가 없는 department를 EMP_DEPT에 삽입하는 경우를 생각해 보자. 한 가지 방법은 employee에 null value를 넣는 것이다. 하지만 SSN이 EMP_DEPT의 primary key이므로 문제가 발생한다. 처음 employee가 이 department에 한 명 들어 오면 더 이상 null value는 필요 없게 된다. EMPLOYEE, DEPARTMENT를 따로 만들 경우는 이런 문제가 발생하지 않는다. employee가 없는 department가 생기더라도 department table에 저장 되므로 null value는 필요없고 employee가 그 department에 배당되더라도 employee table에 insert된다.

2. Deletion Anomalies
   relation table EMP_DEPT에서 마지막 employee를 삭제하게 되면 department 1에 대해 고유한 정보이기에 department 1에 대한 정보가 손실되게 된다. 하지만 각각의 EMPLOYEE, DEPARTMENT table에서는 이런 문제가 발생하지 않는다.

3. Modification Anomalies
   EMP_DEPT에서 department 5의 manager를 바꾼다면, 그 department에서 일하고 있는 모든 employee의 tuple들을 Update해야 한다. 그렇지 않으면 integrity가 깨진다. 따라서, 작업량이 훨씬 늘어나게 되는 것이다. fig2의 EMPLOYEE, DEPARTMENT table에서는 DEPARTMENT의 DNUMBER 5인 tuple하나만 바꾸면 된다.

2.4 절 좋은 Relational schema design

Normalization의 중요성을 부각시키기 위해서 잠시 좋은 Relational schema design이 어떤 것인지 언급해 보기로 하겠다. Relational schema 의 "goodness"를 두 Level에서 볼 수 있다. 첫번째는 logical level이고 두번째는 manipulation(or storage) level이다. logical level은 user가 Relational schema를 interpret하는 방법과 Relational table에 있는 data tuples들의 의미에 관한 것이다. logical level에서 좋은 Relational schema는 relation들의 data tuples의 의미를 사용자가 쉽게 이해할 수 있도록 해 준다.
manipulation level은 base relation에서 tuples이 어떻게 저장되고, Update되는지에 관한것이다. logical level에서는 base relation과 view(virtual relaiton), 둘 다에 관심이 있는 반면, storage level은 단지 물리적으로 저장된 file인 base relation에만 관심이 있다. Normalization은 base relation에 관한 것이다.

1. Semantics of the Relation Attributes
   우선 Relational schema는 meaning을 쉽게 설명할 수 있어야 한다.

   The easier it is to explain the semantics of the relaiton, the better the relation schema design.

   EMPLOYEE Relational schema의 meaning은 매우 단순하다. 각 tuple은 employee를 나타낸다. DNUMBER는 EMPLOYEE와 DEPARTMENT사이의 relationship을 나타내는 foreign key이다. DEPARTMENT tuple은 department entity를 나타내고, PROJECT tuple은 project entity를 나타낸다. DEPARTMENT의 attribute DMGRSSN은 department와 그 department의 manager인 employee를 연결시킨다. PROJECT의 DNUM은 project와 그 project를 맡고 있는 department를 연결시킨다. DEPT_LOCATIONS와 WORKS_ON relation schema의 semantic은 앞에서 언급한 것 보다는 약간 복잡하다.

2. Reducing the redundant values in tuples
   이것에 관한 것은 앞에서 충분히 설명되었다. 중요한 것은 insertion, deletion, modification anomalies가 relation에서 일어나지 않는 base relation을 design하여야 한다는 것이다.

3. Reducing the null values in tuples
   schema design에서 많은 attritue을 모아서 "fat" relation을 만들 수도 있다. 그런 경우는 많은 attribute들이 그 relation 의 모든 tuples에 대해서 값을 안 가질 수 있다. 따라서 relation의 많은 값들이 null 이 되어야한다. 이것은 storage의 낭비이고, attribute의 meaning을 이해하는데 문제를 일으킬 수도 있다. 다른 문제는 COUNT나 SUM과 같은 aggregate operation을 사용할 때 null에 대한 count 문제이다. 더구나 null은 여러가지 의미를 가질 수도 있다.

   - The attribute does not apply to this tuple.
   - The attribute value for this tuple is unknown.
   - The value is known but absent, that is, has not been recorded yet.

   따라서 좋은 Relational schema design이 되기 위해서는 base relation의 값이 가능한 null이 되지 않는 방향으로 attribute들을 적절히 base relation으로 만들어야 한다.

4. Disallowing spurious tuples
   fig5의 EMP_LOCS와 EMP_PROJ1을 보자. 이것은 fig4의 EMP_PROJ 대신 쓸 수 있다. fig5의 table EMP_PROJ1과 EMP_LOCS는 EMP_PROJ의 project operation에 의해 얻어 질 수 있다. EMP_PROJ1과 EMP_LOCS이 EMP_PROJ대신 base relation에 있다고 가정하자. 하지만 이것은 좋은 schema design이 아니다. EMP_PROJ1과 EMP_LOCS로 원래 information인 EMP_PROJ를 얻을 수 없기 때문이다. EMP_PROJ1과 EMP_LOCS를 NATURAL-JOIN한다면, EMP_PROJ보다 더 많은 tuples이 생긴다. join의 결과는 fig6에서와 같다. EMP_PROJ에는 없는데 EMP_PROJ1과 EMP_LOCS의 join의 결과에는 있는 잘못된 tuple을 *spurious tuples*이라고 한다. spurious tuples은 fig6에서 *로 표시되어 있는 tuple이다. 이런 잘못된 결과가 나타나는 이유는 EMP_PROJ를 자를 때 EMP_PROJ1과 EMP_LOCS를 연결시켜 주는 attribute으로 PLOCATION을 선택했기 때문이다. PLOCATION는 EMP_PROJ1나 EMP_LOC에서 primary key도 foreign key도 아니다. 따라서 좋은 Relational schema design을 위해서는 primary key나 foreign key를 선택해서 JOIN을 할 수 있도록 relation schema를 design한다.

지금까지 우리는 table을 쪼개서 여러가지 문제점들이 해결될 수 있음을 살펴 보았다. 하지만 normal form들을 적용시킨 조직적인 방법이 아니었다. 이제부터 우리는 좀 더 조직적이고, 잘 정의되어 있는 rule들을 소개 할 것이다.

제 3 절 Entity Attributes 사이의 관계

두 attributes 사이에 세가지 가능한 연관이 있을 수 있다. 1:1, 1:N , M:N 연관이 그것이다. 즉, 한 attribute가 하나 또는 그 이상의 다른 attributes 의 값을 선택하거나 결정할 수 있는 것이다. Attribute A 가 attribute B를 결정한다고 가정해보자. 이것은 한 entity의 두 instance가 A 에 대해 같은 값을 가질 때 반드시 B에 대해서도 같은 값을 가져야 한다는 것을 의미한다. attribute A 가 attribute B의 값을 결정할 때 그 사이의 1:1 association에 관해 얘기 할 수 있다.record의 primary key는 나머지 attributes의 determinant이다. 따라서 1:1 association 은 single-valued dependency의 example이고 1:N 과 M:N association은 multivalued dependencies의 example이다. 만약 attribute A 와 B 사이에 1:N multivalued dependency 가 있다면 A 의 값이 많은 B의 값을 결정할 수 있다. M:N association이 있다면 attribute A 의 값이 유한개의 B 값을 결정할 수 있고 attribute B 의 값이 또다른 유한개의 A 값을 결정할 수 있는 것이다.

3.1 절 Single–Valued Dependencies

entity specification에서 한 attribute가 나머지 다른 모든 attribute들을 결정할때, determinant는 entity에 대한 candidate key이다. primary key는 candidate key중에서 선택된다. entity는 많은 candidate key을 가질수 있지만, 정의에 의해 단지 하나의 primary key를 가진다. 여기서 우리는 nonkey attributes가 candidate key를 포함한다고 가정한다. 왜냐하면 primary key가 이미 선택되었기 때문이다.
한 entity의 attributes들 사이에는 네 종류의 single–valued dependencies가 있다.

	Attributes	Dependencies
1	key → Nonkey	Functional
2	Part of key → Nonkey	Partial
3	Nonkey → Nonkey	Transitive
4	Nonkey → Part of key	Boyce-Codd Where det-value → attrib-value

첫번째는 바람직한 single-valued dependency로서 primary key가 nonkey를 결정하는 것으로 determinant가 primary key attribute이기 때문이다.따라서 이 dependency는 file과 DBMS environment에서 알려져있고 선언될 수 있다. 나머지 세 종류의 dependencies는 semantics가 application programs에 의해 유지되는 association을 나타내고 있다.
다음의 entity specification을 생각해 보자.

professor Table			
composite key is (pid, office)			
PID	OFFICE	PNAME	DEPT
20	413	Barnes	CS
21	414	Smith	CS
25	404	Gumb	CS
34	304	Wong	EE
55	402	Katz	EE
57	402	Schwartz	EE
27	414	Schwartz	CS

Member	
key is office	
OFFICE	DEPT
414	CS
413	CS
404	CS
304	EE
402	EE

Dependenciesis는 다음과 같이 된다.

- PID → PNAME, OFFICE, DEPT
- OFFICE → DEPT

이에 따른 Dependency graph는 다음과 같이 된다.

두 specification은 학교에서 faculty에 대한 data을 가지는 entities을 정의하고 있다. professor은 유일한 identification number(PID)를 가지고 그들의 office는 같은 department의 동료와 공유한다. 위 그림은 주어진 functional dependency에 대해 가장 좋은 specification은 아니다. 사실, entity specification에서

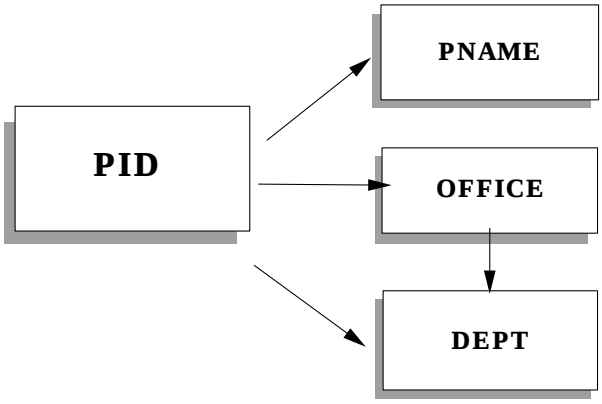

그림 1: Dependency Diagram

네가지의 single-valued dependencies를 증명하는데 문제가 있다. 지금부터 이러한 문제들을 생각해 보자. 보기에서 PID는 PROF entity에 대한 candidate key이고 OFFICE는 MEMBER entity의 유일한 primary key이다. 여기서 PID 와 OFFICE가 PROF에 대한 composite key이고 OFFICE는 MEMBER의 key라고 가정하자. 만약 PID가 PROF의 candidate key 이면 PID 와 OFFICE는 PROF의 unique한 record instance로 선택된다. Dependencies는 entities 사이의 association을 이르는 말이다. 각각의 A 값에 대해 그와 연관된 B의 값이 오직 하나 있을 때 attribute(or set of attributes) B 가 attribute(or set of attributes) A에 functionally dependent 하다고 한다. 따라서, 한 entity의 두 instances가 같은 A 의 값을 가지면 B의 값도 같아야 한다. 이상적으로는 모든 nonkey attribute가 entity의 key 에 functionally dependent하다. 앞의 표에서도 볼 수 있듯이 일반적인 single-valued dependencies의 네 종류는 functional dependencies, partial dependencies, transitive dependencies, and Boyce-Codd dependencies 이다.

- **Functional dependencies**

  2 **F.D**

  Let R be a relation variable, and let X and Y be arbitrary subsets of the set of attributes of R. Then we say that Y is functionally dependent on X–in symbols,

  $X \rightarrow Y$

  (read "X functionally determines Y" simple "X arrow Y") - iff, in every possible legal value of R, each X-value has associated with it precisely one Y-value.

- **Partial dependencies**
  entity specification의 nonkey attribute B가 composite primary key attribute A의 subset에 dependent할때 이러한 dependency를 partial이라고 한다. 아래에서 C의 값을 선택하기 위해 A1 과 A3 값만이 필요하므로 attribute C는 primary key(A1, A2, A3)에 partially dependent하다. 즉, attribute C는 composite key (A1, A2, A3)의 부분 또는 모두에 의해 결정되어지게 된다.

- **Transitive dependencies**
  nonkey attributes 들 사이에 functional dependencies가 존재할 때 transitive dependency가 존재한다. 예를 들어 E3 과 E4의 specification을 생각해보자. E4의 경우 primary key H 는 attribute I 를 결정한다. E3 의 경우 primary key F 는 attributes G, H, I를 결정한다. 여기서 F는 E4에 선언된 functional dependency를 통해 I 를 선택할 수 있으므로 E3에는 transitive dependency가 존재한다. 따라서 F 로부터 H 로, H 에서 I 로의 직접 또는 간접적 방법을 통해 F의 값에 근거한 I의 instance에 접근할 수 있다.

- **Boyce-Codd dependencies**
  nonkey attribute가 composite primary key attribute set의 부분을 결정지을 때 그 entity specification에는 Boyce-Codd dependency가 존재한다.그 예로서 E5를 들 수 있다. M 이 composite primary key의 부분이고 P가 nonkey 이므로 P 와 M 사이에 Boyce-Codd dependency가 존재한다. 만약 P가 nonkey가 아닌 candidate key라면 Boyce-Codd dependency가 존재하지 않는 것이 된다.

3.2 절 Multivalued Dependencies

attribute가 다른 attribute에 대해 하나 이상의 값을 결정할 때 multivalued dependency가 존재한다. 이 경우, determinant 와 set of determined attributes사이에 one-to-one association이 성립하지 않는다. 이런 multivalued dependency는 그래프상 double headed arrow로 표시한다. multivalued dependencies의 예로서 다음의 entity specification TEXTS를 생각해보자.

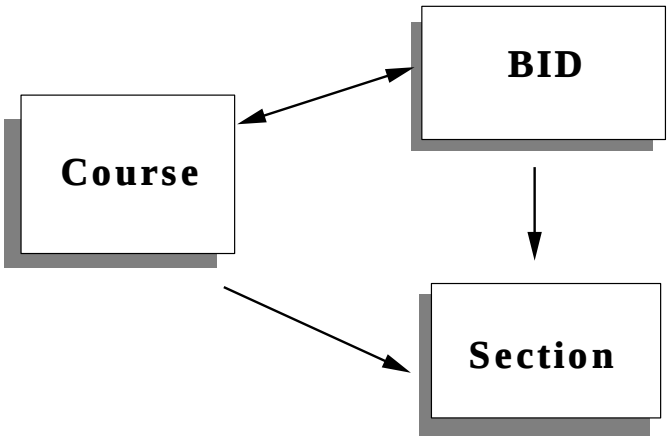

그림 2: Dependency Diagram

TEXTS		
composite key is (bid, section)		
BID	COURSE	SECTION
20	CS132	456
20	CS132	457
20	CS182	458
30	CS182	458
20	CS182	459
30	CS182	459
38	CS242	461
42	CS242	461
38	CS242	463
42	CS242	463
42	CS440	540

Dependencies는 다음과 같다.

- BID → COURSE
- COURSE → BID
- COURSE → SECTION

이에 따른 Dependency graph는 다음과 같다.

이 specification은 두 개의 독립된 multivalued dependencies 를 포함하고 있다. courses 와 textbooks에 관한 정보는 book identification number(BID), course number(COURSE), course section number(SECTION)에 따라 저장된다. 책은 여러 course에서 사용될 수 있고, course는 여러 책을 사용할 수 있으므로 COURSE 와 BID 사이의 dependency는 M:N 이 된다. 또, course는 다수의 section을 가지므로 COURSE 와 SECTION 사이의 dependency는 1:N 이 된다. 이 때 , 같은 course의 모든 section은 같은 textbook를 쓴다고 가정한다. 이러한 semantics와 multivalued dependencies때문에 TEXTS의 모든 instance 를 표현하는 데에는 많은 불필요한 data item이 필요하다. TEXTS에는 모두 11 instances 가 있는데 attributes를 위해 각각의 data values가 총 33개가 필요하다. 이 33개의 data가운데 단지 15개 만이 unique하다. 나머지는 모두 data redundancy를 나타내고 있다. 이러한 redundant values는 서로 독립된 multivalued dependencies의 cross product에 의해 발생한 결과이다. 두개의 multivalued dependencies (COURSE –>> BID, BID –>> COURSE), (COURSE –>>SECTION)은 서로 독립적인데 이유는 SECTION과 BID 사이에 직접적인 dependency가 없기 때문이다.

3.3 절 Lossless Decompositions

dependency에 따라 table을 하나 또는 그 이상의 작은 table로 decompose하는 것이 Normalization인데, 이렇게 decompose하는 것만이 중요한 것이 아니라, decomposed part들을 다시 join했을 때 처음의 table을 얻을 수 있어야 제대로 된 Normalization 과정인 것이다.

3 Decomposition

주어진 table T 에서, k 개의 tables 로의 decomposition은 $\{T_1, T_2, T_3, ..., T_k\}$ table의 모임이다. 즉, Head(T) = Head(T_1) $\cup$ Head(T_2) $\cup \cdots \cup$Head(T_k)가 된다.

T의 rows들은 T_j의 column들로 project된다. functional dependency의 set F를 가지고 table T를 decomposition하는 것을 **lossless decomposition**이라고 부른다.

제 4 절 Theory of Functional Dependencies

functional dependencies에 대하여 많은 theory들이 발전해 왔다. 여기서는 BCNF 와 3NF Relational database의 design에 적용되는 theory의 가장 유용한 algorithm에 초점을 맞추고자 한다. 일반적으로 functional dependency가 주어졌다면 이 functional dependency는 다른 functional dependency를 나타낼 수도 있다는 것을 의미한다. 예를 들어, R = (A, B, C, G, H, I)라는 relation scheme가 주어졌다고 가정하자. 그리고 여기에 따른 functional dependency set은 다음과 같다.

- A $\rightarrow$ B
- A $\rightarrow$ C
- CG $\rightarrow$ H
- CG $\rightarrow$ I

- B → H

단지 주어진 functional dependencies만을 고려하는 것은 충분하지 않고 이 functional dependencies 가 가진 모든 functional dependencies를 고려할 필요가 있다. 곧 보이겠지만 주어진 functional dependencies set은 또 다른 functional dependencies를 가진다는 것을 알수 있다. 이러한 functional dependency를 F에 의해 logically implied 되었다고 한다. 위의 functional dependency에서 functional dependency A→H 는 logically implied 되었다.즉 functional dependency 의 set이 주어질 때마다 A→H 의 functional dependency도 반드시 있다는 것이다. 그럼 $t_1[A] = t_2[A]$ 인 t_1 과 t_2 tuples를 가정해 보자. A→B 라는 functional dependency가 주어졌기 때문에 functional dependency 의 정의에 따라 $t_1[B] = t_2[B]$ 가 성립한다. 또, B→H 의 functional dependency가 주어졌기 때문에 $t_1[H] = t_2[H]$ 도 역시 functional dependency 의 definition에 의해 성립한다. 따라서 t_1 과 t_2 가 $t_1[A] = t_2[A]$ 을 성립시킬 때마다 $t_1[H] = t_2[H]$ 이 성립함을 볼 수 있는데 이것은 A→H 를 의미한다.
F를 functional dependency의 set이라고 하자. F의 closure를 F^+ 로 표시하는데 F 의 closure란 F에 의해 logically implied 된 모든 functional dependency의 set을 말한다. F 가 주어졌을 때 functional dependency의 formal definition으로 부터 직접 F^+를 계산할 수 있다. 만약 F의 크기가 크다면 물론 이 작업이 쉬운 것은 아닐 것이다. F^+의 계산은 위에서 A–>H 가 closure에 있다는 것을 보이기 위해 주어진 type의 argument를 필요로 한다. 지금부터 functional dependency에 대하여 좀 더 쉽게 생각할 수 있는 기법을 보이기로 하겠다. inference rules 또는 axioms에 기초를 둔 기법이 그것이다. 이 rules을 반복적으로 적용시킴으로서 F 의 F^+를 찾을 수 있다.

1. **Reflexivity rule**
   X가 attributes의 set이고 $Y \subseteq X$이면 X→Y가 성립한다.

2. **Augmentation rule**
   X→Y 가 성립하고 W 가 attributes의 set이면 WX→WY 가 성립한다.

3. **Transitivity rule**
   X→Y 가 성립하고 Y→Z 가 성립하면 X→Z 가 성립한다.

   이 rules 들을 사용했을 때 잘못된 functional dependency가 발생하지 않기 때문에 인정받고 있다. 그리고 이 규칙들은 완벽한데, 이유는 functional dependency 의 set F가 주어지면 F^+의 모든 것을 찾게 해 주기 때문이다. 이러한 rule을 Armstrong's axioms이라고 한다. 이 Armstrong's axioms가 완전한 것이라 할지라도 F^+를 계산하는데 직접 사용하기는 힘들다. 그래서 좀 더 간소화하기 위해 몇가지 부가적인 rule을 첨가 했다. 물론 이 부가적인 rule들은 Armstrong's axioms을 사용하여 정당성을 증명할 수 있다.

4. **Union rule**
   X→Y 가 성립하고 X→Z 가 성립하면 X→YZ 가 성립한다.

5. **Decomposition rule**
   X→YZ 가 성립하면 X→Y 가 성립하고 X→Z 가 성립한다.

6. **Pseudotransitivity rule**
   X→Y 가 성립하고 WY→Z 가 성립하면 XW→Z 가 성립한다.

그러면 이 rules들을 앞에서 제시한 보기에 적용시켜보자. 아래에 F^+의 몇 member를 적어본다.

- A→H : A→B 와 B→H 가 성립하므로 rule 3(transitivity rule)을 적용한다.
- CG→HI : CG→H 와 CG→I 가 성립하므로 union rule(rule 4)를 적용한다.
- AG→I : 이것을 보이기 위해선 몇 단계가 필요하다. 먼저 A→G 가 성립하므로 augmentation rule(rule 2)를 사용하여 AG→CG를 얻고 CG→I가 성립하므로 transitivity rule(rule 3)에 의해 AG→I가 된다.

이제 set X 가 superkey인지 확인하기 위해 X에 의해 functionally 결정되는 attributes set을 계산해 봐야한다. 이 계산 역시 functional dependency의 set F의 closure를 계산하는데 유용한 일부이다. 그럼 X가 attributes set이라고 가정하자. functional dependency의 set F하에서 X에 의해 functionally determined 되는 모든 attributes의 set을 F하에서의 X의 closure라고 하고 X^+로 나타낸다. F^+를 계산하는 방법은 inference rule을 이용하는 것이다. 다행히 좀 더 효율적인 접근이 있다. 아래의 algorithm은 X^+를 계산하기 위한 것이다.

4 Algorithm

```
result := X;
while ( changes to result ) do
for each functional dependency Y->Z in F do
begin
if Y ⊆ result then result := result ∪ Z ;
end
```

input은 functional dependency 의 set F 와 attributes set X이고 output은 변수 result에 저장된다. 이 algorithm이 어떻게 동작하는지 보이기 위해 $(AG)^+$를 계산해 보겠다. 먼저 result = AG 로 시작한다. 각 functional dependency를 test하기 위해 while loop를 실행시키는데 첫 loop에서 다음을 알수 있다.

- A→B 때문에 B는 result에 포함된다. A→B가 F에 있고 $A \subseteq result (AG)$이고 따라서 result := $result \cup B$인 것으로 알수 있다.
- A→C 는 result가 ABCG가 되게 한다.
- CG→H는 result가 ABCGH가 되게 한다.
- CG→I는 result가 ABCGHI가 되게 한다.

while loop를 두번째 돌 때 result에 새로운 attribute가 더해지지 않고 algorithm은 끝나게 되는 것이다. 어떻게 이 algorithm이 옳다고 할 수 있는지 알아보자. 첫 단계로 X→X 가 reflexivity rule에 의해 항상 성립한다. 그래서 result의 임의의 subset Y에 대해 X→Y라고 말할 수 있다. while loop를 시작할 때 X→result였기 때문에 Z를 result에 더할 수 있는 유일한 방법은 $Y \subseteq result$이고 Y→Z가 성립할 때이다. result→Y가 reflexivity rule에 의해 성립하고 transitivity에 의해 X→Y 가 성립한다.
transitivity의 또 다른 적용은 X→Z 임을 보여준다.(X→Y이고 Y→Z임을 이용하여). Union rule은 $X \rightarrow result \cup Z$를 보여주고 X는 while loop에서 생기는 임의의 새로운 result를 functionally determine한다. 따라서 이 algorithm에 의해 되돌려지는 어떠한 attribute도 X^+에 있게 된다. 또 이 algorithm이 X^+의 모든 것을 찾는다는 것도 보이기 쉽다. 만약 X^+에는 있고 result에는 없는 어떤 attribute가 있다면 $Y \subseteq result$이고 F의 적어도 한 attribute가 result에 없는 것에 대해 Y→Z의 functional dependency가 반드시 존재한다. 밝혀진 바에 의하면 F의 size에 따라 최악의 경우 이 algorithm은 네번의 loop를 돌게된다.

4.1 절 Desirable Properties of Decomposition

database system을 design하는데 있어 relation을 작은 여러개의 relation으로 decompose하는 것이 필요하다. 이 개념을 보이기 위해 Lending–scheme을 생각해 본다.
Lending–scheme = (branch-name, assets, branch-city, loan-number, customer-name,amount)
그리고 functional dependency set F는 다음과 같다.

branch-name → assets
branch-name → branch-city
loan-number → amount
loan-number → branch-name

이것을 다섯개의 relation으로 다음과 같이 decompose한다고 가정해 보자.

R_1 = (branch-name, assets)
R_2 = (branch-name, branch-city)
R_3 = (loan-number, amount)
R_4 = (loan-number, branch-name)
R_5 = (loan-number, customer-name)

Lossless–join decomposition

위의 decomposition은 lossless인데 이것을 증명하기 위해 먼저 이를 결정하는 criterion을 제시해야 한다. R은 relation scheme이고 F는 R에서의 functional dependency set이라고 할 때 R_1 과 R_2는 R의 decomposition을 lossless–join decompose한다.

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

lossless–join decomposition임을 보이기 위해 decomposition의 단계를 아래에 보이겠다.먼저

$$R_1 = (\text{ branch-name, assets })$$
$$R_1^{'} = (\text{ branch-name, branch-city, loan-number, customer-name, amount })$$

branch-name $\rightarrow$ assets 이기 때문에 augmentation에 의해 branch-name $\rightarrow$ branch-name assets이 된다. $R_1 \cap R_1^{'} =$ branch-name 이기 때문에 초기의 decomposition은 lossless–join decompose하다. 다음으로 $R_1^{'}$을

$$R_2 = (\text{ branch-name, branch-city })$$
$$R_2^{'} = (\text{ branch-name, loan-number, customer-name, amount})$$

로 decompose 한다. branch-name$\rightarrow$ branch-city이므로 이 단계도 lossless–join decompose이다. $R_2^{'}$를

$$R_3 = (\text{ loan-number, amount })$$
$$R_3^{'} = (\text{ loan-number, branch-name })$$

으로 decompose하고 $R_3^{'}$를

$$R_4 = (\text{ loan-number, branch-name })$$
$$R_5 = (\text{ loan-number, customer-name })$$

으로 decompose한다.

제 5 절 Theory of Multivalued Dependencies

D를 functional and multivalued dependencies set이라고 하자. 그러면 D의 closure D^+는 D에 의해 logically implied 되는 모든 functional and multivalued dependencies의 set을 말한다. functional dependencies의 경우처럼 functional dependencies and multivalued dependencies의 definition을 이용하여 D로 부터 D^+를 계산할 수 있다. 다음의 functional and multivalued dependencies에 대한 inference rules은 완벽하다. 처음 세 rules은 앞에서 보았던 Armstrong's axioms이다.

1. **Reflexivity rule**
   X가 attributes의 set이고 $Y \subseteq X$일 때, X$\rightarrow$Y가 성립한다.

2. **Augmentation rule**
   X$\rightarrow$Y가 성립하고 W가 attributes의 set이면 WX$\rightarrow$WY가 성립한다.

3. **Transitivity rule**
   X$\rightarrow$Y가 성립하고 Y$\rightarrow$Z가 성립하면 X$\rightarrow$Z가 성립한다.

4. **Complementation rule**
   X→Y이면 X→ R−Y− X가 성립한다.

5. **Multivalued augmentation rule**
   X→Y가 성립하고 $W \subseteq R$ 이고 $V \subseteq W$이면 WX→VY가 성립한다.

6. **Multivalued transitivity rule**
   X→Y이고 Y→Z이면 X→ Z−Y가 성립한다.

7. **Replication rule**
   X→Y가 성립하면 X→Y가 성립한다.

8. **Coalescence rule**
   X→Y 이고 $Z \subseteq Y$이고 $W \subseteq R$, $W \cap Y = \phi$, W →Z가 성립하는 W가 존재하면 X→Z가 성립한다.

그러면 위의 rule들에 대한 example을 보기로 하자. R = (A, B, C, G, H, I)의 relation scheme이라고 한다. 그리고 A→BC 가 성립한다고 가정한다.multivalued dependencies 의 definition은 $t_1[A] = t_2[A]$ 이면 다음을 만족하는 tuples t_3 과 t_4가 존재한다는 것을 의미한다.

$$t_1[A] = t_2[A] = t_3[A] = t_4[A]$$
$$t_3[BC] = t_1[BC]$$
$$t_3[GHI] = t_2[GHI]$$
$$t_4[GHI] = t_1[GHI]$$
$$t_4[BC] = t_2[BC]$$

complementation rule은 A → BC 이면 A → GHI 임을
말해준다. 간단히 subscript만을 바꾼다면 A→GHI 의 정의를 t_3과 t_4가 만족시킨다는 것을 알 수 있다. 비슷한 단정을 rule 5, 6에 대해서도 multivalued dependencies의 definition을 이용하여 내릴 수 있다. Rule 7, replication rule은 functional and multivalued dependencies를 포함한다. 가령 R에서 A→BC 가 성립한다고 가정해 보자. $t_1[A] = t_2[A]$, $t_1[BC] = t_2[BC]$의 관계가 있고 t_1과 t_2는 각각 자신에게 관계가 있을 때 multivalued dependency A→BC의 정의에 의해 tuples t_3, t_4가 요구된다. Rule 8,coalescence rule,은 여덟가지 rule중에서 보이기 가장 힘든 것이라 여기서 생략하겠다.
다음의 rules을 이용하여 D의 closure를 간단히 계산할 수 있다.

- **Multivalued union rule**
  X→Y, X→Z 가 성립하면 X→YZ 가 성립한다.

- **Intersection rule**
  X →Y, X→Z가 성립하면 X→ $Y \cap Z$가 성립한다.

- **Difference rule**
  X→Y, X→Z 가 성립하면 X→ Y−Z , X→ Z−Y가 성립한다.

위의 rule을 다음의 보기에 적용시켜보자. R = (A, B, C, G, H, I) 이고 이것의 dependencies set D는 아래와 같다.

- A→B
- B→HI
- CG→H

아래에는 D^+의 몇가지 member를 보인 것이다.

- A→CGHI : A→B이기 때문에 complementation rule(rule 4)는 A→ R−B−A를 나타낸다. 여기서 R−B− A = CGHI 이므로 A→CGHI가 된다.
- A→HI : A→B , B→HI 이기 때문에 multivalued transitivity rule(rule 6)은 A→ HI−B를 나타낸다. HI−B = HI이기 때문에 A→HI가 된다.
- B→H : 이것을 보이기 위해 coalescence rule(rule 8)을 적용하여야 한다. B→HI가 성립한다. $H \subseteq HI, CGrightarrowH, CG \cap HI = \phi$이기 때문에 X는 B로 Y는 HI로, W는 CG로, Z는 H로 하여 coalescence rule을 만족시킨다.따라서 B→H의 결론을 내릴 수 있다.
- A→CG : 이미 A→CGHI , A→HI가 성립함을 알고 있다. difference rule에 의해 A→CGHI − HI 가 되고 CGHI − HI = CG 이므로 A→CG가 성립한다.

제 6 절 Normal form

Normalization을 한마디로 요약하면 가능한 한 중복성을 배제하여 한가지 사실은 한 곳에서만 나타나게 하여 정보의 무결성을 보장하게 하는 이론이라 할 수 있다. 따라서 Normalization의 궁극적인 목적은 real world를 보다 효율적이고 현실에 가깝게 표현하는데 있다고 하겠다.
Real world는 계속 변하고 있고, 따라서 real world를 표현하는 방법은 계속 발전할 수 있으나, 현재까지 체계가 확립된 Normal form은 6가지이다. 이들의 성립도 한꺼번에 이루어 진것이 아닌것에 착안을 하면 앞으로 Normal form의 발전이 더 진전되리라 예상되지만 실제 응용에의 한계에 의해 4,5 정규형은 거의 쓰여지지 않은 상태이다.
Normal form의 간단한 특징에 대해 알아보자.

- 다음 그림에서 볼 수 있듯이 Normalize되어 가는 각 단계는 hiararchy가 있다. 즉 정규화된 모든 relation은 1NF에 속하고, 2NF는 1NF에 각각 포함된다는 사실을 알 수 있다.

- 대부분의 data base는 BCNF까지만 만족하면 별무리 없이 사용 할 수 있다. 즉 4NF와 5NF는 real world에 적용되는 경우도 있을 수 있지만 이론상의 발전으로 이해하면 될 것이다.

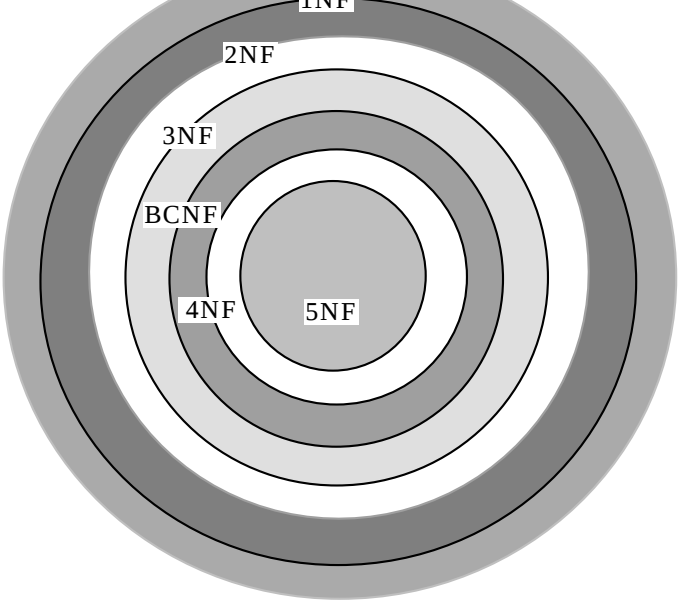

그림 3: NF Hierarchy.

앞에서 설명한 여러가지 이론을 가지고, 지금부터는 구체적인 예를 보면서 현재까지 발전한 6가지 normal form을 소개하겠다.
예에 대해 간단히 설명을 하면 5NF까지 바꿀 수 있는 것으로 생각해 보았으나 쉽지 않았다. 그래서 4NF까지 가능한 것으로 하나 들었고, 4NF에서 5NF로 가는 것은 따로 별도의 예를 들었다.

Normalization이 되기 전에는 다음 그림 6와 같은 Table이다.

6.1 절 1 Normal Form

5 First Normal Form

First Normal Form : All domain contain **atomic** values only.

보통의 table에서 볼 때 각 tuple당 attribute에 대해 하나의 value만 들어가면 1NF라 할 수 있다. domain value가 주어진 경우에는 그 범위 안에 들어가야 하며, 제약이 없는 한 Null value도 가능하다. table1을 1NF하면 FIRST가 된다. table1에 있는 repeating group을 없애 FIRST 모든 domain이 atomic value만을 포함하므로 1NF이다. 하지만, 이 relation은 너무 많은 redundancy를 가지고 있다. 따라서 다음과 같은 anomaly가 일어날 수 있다.

- Insert : supplier가 한 개 이상의 part를 공급할 때까지 supplier가 특정한 city에 위치한다는 것을 insert할 수 없다. 예를 들면 Athens에 있는 supplier S5를 insert할려고 해도 S5가 part를 공급하지 않는다면 insert 할 수 없다.

- Delete : supplier의 tuple이 하나일 때 그 supplier가 공급한 part를 delete할 경우, supplier가 특정

S1

P#	QTY	STATUS	CITY
P1	300	20	London
P2	200		
P3	400		
P4	200		
P5	100		
P6	100		

S2

P#	QTY	STATUS	CITY
P1	300	10	Paris
P2	400		

S3

P#	QTY	STATUS	CITY
P2	200	10	Paris

S4

P#	QTY	STATUS	CITY
P2	200	20	London
P4	200		
P5	100		

한 city에 위치한다는 정보까지 같이 없어진다. 예를 들면 S#가 S3, P#가 P2인 tuple을 delete하면 S3가 Paris에 위치한다는 것까지 없어진다.

- Update : supplier가 city를 옮길 경우, 그 supplier의 모든 tuple을 찾아서 update해 주어야한다. 예를 들면 supplier S10이 London에서 Paris로 옮길 경우, S#가 S1인 모든 tuple을 찾아서 고쳐주어야 한다. 이와 같은 문제점이 발생하므로, 이것을 없앤 것이 second normal form이다.

S#	STATUS	CITY	P#	QTY
S1	20	London	P1	300
S1	20	London	P2	200
S1	20	London	P3	400
S1	20	London	P4	200
S1	20	London	P5	100
S1	20	London	P6	100
S2	10	Pairs	P1	300
S2	10	Pairs	P2	400
S3	10	Pairs	P2	200
S4	20	London	P2	200
S4	20	London	P4	300
S4	20	London	P5	500

6.2 절 2 Normal Form

6 Second Normal Form

Second Normal Form : Every nonkey attribute is fully dependent on primary key.

Relation이 1NF이고, 모든 nonkey attribute는 primary key키에 irreducibly dependent하면 Second normal form이다. FIRST에서 city와 status는 S#에 functional dependent하므로 이것은 2NF 정의에 어긋난다. 따라서 이것을 분리해 냄으로 2NF 정의를 만족할 수 있다. 따라서 table2를 2NF하면 SECOND과 SP가 된다. 하지만, 이 relation도 문제점이 있다.

- Insert : city에 supplier가 위치할 때까지 city의 status를 insert할 수 없다. 예를 들면, seoul에 suppler가 한명도 없다면 seoul의 status를 insert할 수 없다.

- Delete : city에 supplier가 한 명 있을 경우, 그 supplier를 없앤다면, 그 city의 status도 같이 없어진다. 예를 들면, S5를 없앤다면 Athens의 status까지 같이 없어진다.

- Update : city의 status를 update할 경우, 그 city의 모든 tuple을 찾아서 update해 주어야한다. 예를 들면, Paris의 status가 10에서 20으로 바뀔 경우, city가 Paris인 모든 tuple을 찾아서 update해야 한다.

  이와 같은 문제점이 발생하므로, 이것을 없앤 것이 Third normal form이다.

SECOND

S#	STATUS	CITY
S1	20	London
S2	10	Pairs
S3	10	Pairs
S4	20	London

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

6.3 절 3 Normal Form

7 Third Normal Form

Third Normal Form :A relation is in 3NF, iff it is in 2NF and every nonkey attribute is *non-transitively* dependent on the primary key.

Relation이 2NF이고 모든 nonkey attribute는 primary key에 nontransitively dependent하면 Third normal form이다. SECOND에서 status는 S#에 functional dependent하면서 city에도 functionally dependent하다. 따라서 이것은 3NF의 정의에 어긋난다. 따라서 이것을 분리해 냄으로 3NF 정의를 만족할 수 있다. 따라서 table3을 3NF하면 SC와 CS이 된다.

SC

S#	CITY
S1	London
S2	Pairs
S3	Pairs
S4	London

CS

CITY	STATUS
London	20
Paris	10

6.4 절 Boyce-codd Normal Form

8 BCNF

Boyce-Codd Normal Form : All determinant must be *Candidate* key.

3NF까지는 Codd가 제안한 것인데, 3NF는 부적절한 성질을 가지고 있다.
자세히 말하자면, relation이 다음과 같을 때는 적당하지 않다.

- Multiple candidate key를 가질 경우
- 이런 candidate key들이 복합속성을 가질 경우
- candidate key들이 overrapped할 경우

이런 경우에 대응하기 위해 원래의 3NF 정의가 Boyce와 Codd에 의해 좀더 강한 의미로 대체된다. 이 새로운 정의는 원래의 3NF보다 강한 정규형을 의미하므로, 3NF라고 부르는 대신 BCNF(Boyce/Codd Normal Form)라고 이름지어 졌다.
예를 들면 SCT에서 T는 S, C에 functionally dependent하고, C는 T에 functionally dependent하다. 따라서 문제점이 생긴다.

- Insert : course에 student가 없다면, course에 teacher의 정보를 넣을 수 없다.

- Delete : course에 한 명의 student가 있고, 그 student를 없앨 경우, course의 teacher의 정보까지 같이 사라진다.

- Update : course의 teacher가 바뀔 경우, 그 course의 모든 tuple을 찾아서 고쳐주어야한다.
  이와 같은 문제점이 생기는 것은 attribute teacher가 candidate key가 아님에도 determinet이기 때문인데, 이것은 BCNF의 정의에 어긋난다. 따라서 이것을 분리해 냄으로 BCNF 정의를 만족할 수 있다. 따라서 SCT를 BCNF하면 ST, TC로 분리된다.

SCT

STUDENT	COURSE	TEACHER
Smith	Math	Prof. White
Smith	Physics	Prof. Green
Jones	Math	Prof. White
Jones	Physics	Prof. Brown

ST

STUDENT	TEACHER
Smith	Prof. White
Smith	Prof. Green
Jones	Prof. White
Jones	Prof. Brown

TC

TEACHER	COURSE
Prof. White	Math
Prof. Green	Physics
Prof. Brown	Math

6.5 절 4 Normal Form

9 Fourth Normal Form

Relation에서 Multivalued dependense A→B가 존재할 때, relation의 모든 속성은 A에 함수적 종속이면 4NF이다. 이때는 그 역도 성립하는데, 즉 reltion의 유일할 종속성은 K→X 형태이다.

BP를 보면 갱신이상의 요인이 되는 중복성을 지니고 있음은 매우 명백하다. 모든 속성의 결합인 (Name, Author, Publisher)를 제외하고는 함수적 종속성이 없으므로 BP는 BCNF이다.
BCNF에도 문제가 자주 발생한다. 예를 들어 Name이 Parallel이고 Author가 DAVIS에서

DAVIT으로 바뀌졌을때 Publisher에 대해 각각 하나씩 새로운 tuple 두 개가 생성되야 한다. 이는 Author, Publisher가 서로 독립적이면서 Name에 대해서는 다치종속성(mulivalues dependency) 이 있기 때문이다. 우선 MVD(mulivalues dependency)의 정의를 보면

10 Multivalued Dependense

Relation R이 속성 A, B와 C로 구성되어 있고 A→B가 성립하면, 주어진 (A, C)쌍에 관계되는 일련의 B값들이 A값에만 종속하면 C값에는 독립적이어야 한다.

이 된다. Relation BP에는

1. Name → Year
2. Name → Author
3. Name → Publisher

의 세 가지 다치 종속성이 있다.
위의 relation BP에서 생기는 문제는 함수적 종속성이 아닌 다치종속성을 포함하기 때문이다. 따라서 relation BP를 다치종속성이 없도록 세가지의 relation으로 projection시키면 이러한 문제가 없어진다. BA(Name, Year), BB(Name, Author), BC(Name, Publisher)
그런데 여기서 BA(Name, Year) relation은 이미 table B내에 존재하기 때문에 또 쓸필요가 없다.

6.6 절 5 Normal Form

11 Fifth Normal Form

Candidate key에 포함되지 않는 종속성을 제거하기 위하여 4NF reltion의 projection을 취한 것이 제 5정규형이다.

2개의 projection으로 nonloss-decomposition 되지 않고 세 개나 그 이상으로 decomposition 되는 경우가 존재하는 경우 제 4정규형으로는 해결이 되지 않는다.
예에서 NSM을 두개의 relation으로 projection시킬 경우에 (즉, NS와 SM 또는 SM과 MN 또는 MN과 NS) 이것을 join시켜 보면 실제로는 data에 없는 tuple이 생기게 된다. 그러나 나머지 하나의 relation 과 join하게 되면 원래의 NSM을 얻을 수 있다.
이렇게 3개 이상의 relation만으로 projection이 가능한 이유는 join dependency가 재약조건으로 작용하기 때문이다. 여기서 join dependency를 정의하여 보자.

12 join dependency

X, Y, ... , Z 는 R의 속성의 부분집합이다. relation R이 join dependency ∗(X, Y, ... , Z)를 만족하면, X, Y, ... , Z에 대한 projection들의 join과 R이 같다. 이 역도 성립한다.

5NF에 대해 쉽게 설명하면 현실세계에 이런 제약 조건이 존재한다.

* C++ 책에 Object라는 주제가 있다.
* Object라는 개념이 전산과와 관련되어 있다.
* C++ 책이 전산과 관련 서적이라고 가정하면 C++ 책은 Object에 관한 전산과 관련 서적이다.

이런 제약조건이 가능한 real world에서

(C++, Object, Math)
(C++, Class, CS)

라는 data가 존재하면 (Parallel, Object, CS) 라는 정보가 insert될 때 (C++, Object,CS) 라는 tuple이 함께 추가 되어 져야 한다는 것이다.
이런 문제점이 relation NSM을 세개의 relation으로 projection시키면 자연스럽게 해결된다.

제 7 절 Implementation of Normalization

We will make normalization tool, and this tool is an independent application tool.

7.1 절 The purpose of Program

We decided several program rules for developing normalization tool.

* Project Name is < JOE Project >.
* To develop the general database design tool independent of special DBMS.
* To give the user the most efficient normal form by automatical normalization process to 2NF, 3NF, BCNF.
* Development Program Language : C++ (Visual C++)
* Graphic Library : MFC

7.2 절 Schedules of developing program

Program Plan

* **First Week**
  To decide an whole structure and libraries, tools which are needed.
* **Second Week**
  To decide an Algorithm and Data Structure.
* **Third Week**
  To implement a given structure, decide inheritances and internal functions.
* **Fourth Week**
  To define internal class memeber function, coding directly.

* **Fifth Week**
  To make user interface by MFC, to bind coded program and user interface.
* **Sixth Week**
  To test program by using many testing data, to modifiy and debug.
* **Last Week**
  DEMO day and end.

7.3 절 Division of working

* **Attribute and Attribute set class implementation** : Taeho Oh
* **Functional Dependency and Functional Dependency class implementation** : Hyunho Jung
* **RelationSchema Class implementation** : Joon-Myung Kang
* **2NF Function** : Taeho Oh
* **3NF Function** : Hyunho Jung
* **BCNF Function** : Joon-Myung Kang

참고 서적

[1] Fred R. McFadden and Jeffrey A.Hoffer, "Data Base Management",(1985), The Benjamin/Cummings publishing company.

[2] George Gardarin and Patrick Valduriez, "Relational Databases and Knowledge Bases", (1986), Addison Wesley, 2nd-edition.

[3] Paolo Atzeni and Valeria De Antonellis, "Relational Database Theory", (1993), The Benjamin/Cummings publishing company.

[4] Alan F.Dulka and Howard H.Hanson, "Fundamentals of Data Normalization",(1989), Addison Wesley,2nd-edition.