

# 13 장. 관계 데이터베이스의 함수적 종속성과 정규화 기본 이론

## Contents

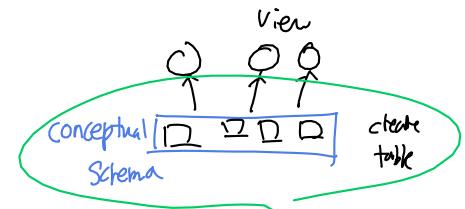
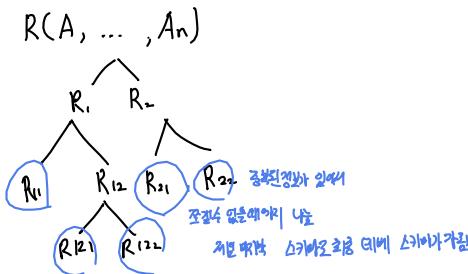
- 릴레이션 스키마를 설계하는 몇 가지 개략적인 지침
- 함수적 종속성 (functional dependencies, FDs)**
- 기본 키를 기반으로 한 정규형
- 제 2 정규형과 제 3 정규형의 일반적인 정의
- BCNF (Boyce-Codd Normal Form)

Relational Database design의 가장 중요한 목표 2개

- Information preservation
- minimum redundancy

## 릴레이션 스키마를 설계하는 몇 가지 개략적인 지침

- 관계형 데이터베이스 설계란?
  - “좋은” 릴레이션 스키마를 생성하기 위하여 **애트리뷰트들을 그루핑하는 과정**
  - “좋은” 릴레이션에 대한 기준은?
- 릴레이션 스키마의 두 가지 수준
  - 논리적인 “사용자 뷰(user view)” 수준
  - 저장이 되는 “기본 릴레이션(base relation)” 수준
- 데이터베이스 설계는 주로 기본 릴레이션을 대상으로 함
- 내용
  - 여기서는 좋은 릴레이션 설계에 관한 개괄적인 지침을 논한 후, 함수적 종속성과 정규형 개념에 관해 논의함
  - 1NF (제 1 정규형)
  - 2NF (제 2 정규형)
  - 3NF (제 3 정규형)
  - BCNF (Boyce-Codd 정규형)



one information at one place

No Redundancy

① EMP(ssn, name, ..., salary)

② EMP(ssn, name, name) \*

EMP\_SAL(ssn, salary)

③ Space를 약제 사용한다.

④ 계약 조건에 모든 충족 가능해야 한다.



## 릴레이션 애트리뷰트들의 의미

설계자는 데이터베이스에 포함될 전체 애트리뷰트들을 대상으로 실세계에서 어떤 연관성이 있는 애트리뷰트들을 묶어서 하나의 릴레이션 스키마를 만들어 나간다.

- 여러 엔티티(EMPLOYEE, DEPARTMENT, PROJECT)의 애트리뷰트들이 하나의 릴레이션에 혼합되면 안된다.  
    <sup>불이되어 있어야 함.</sup>
- 다른 엔티티를 참조하기 위해서는 외래키만을 사용해야 한다.

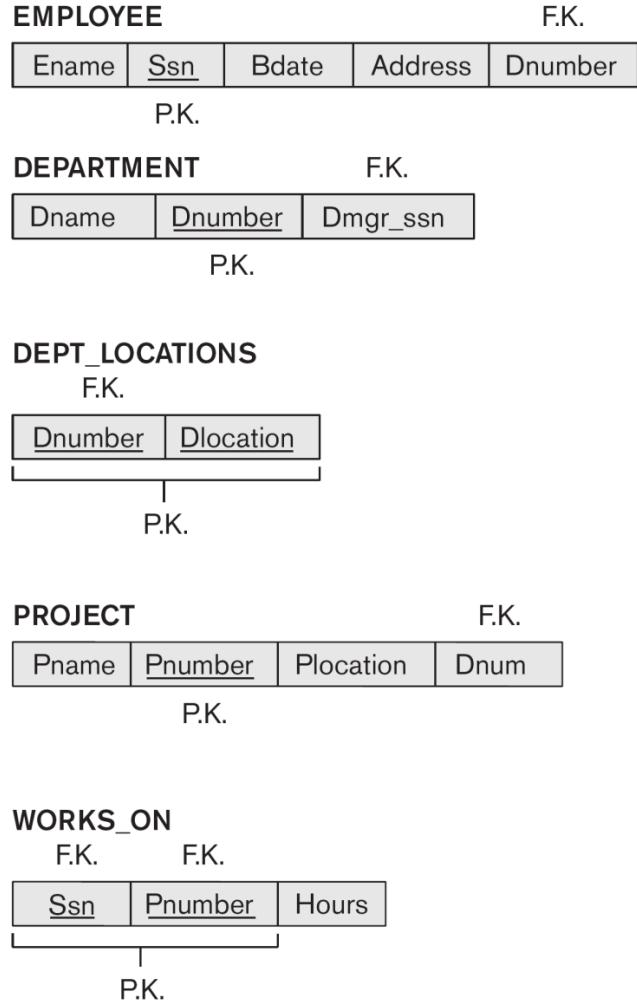
예 :

foreign keys 만에 서로 다른 entity들을  
참조할 수 있어야 한다

- 잘 설계된 경우 (Fig 15.1)
- 데이터베이스 인스턴스 (Fig 15.2)
- 여러 엔티티의 속성들이 하나의 릴레이션에 혼합되어 문제 발생 (Fig 15.3)

**Figure 15.1**

A simplified COMPANY relational database schema.



**Figure 15.2**

Sample database state for the relational database schema in Figure 15.1.

**EMPLOYEE**

Ename	Ssn	Bdate	Address	Dnumber
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1

**DEPARTMENT**

Dname	Dnumber	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

**DEPT\_LOCATIONS**

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

**WORKS\_ON**

Ssn	Pnumber	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	Null

**PROJECT**

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

**Figure 15.3**

Two relation schemas suffering from update anomalies. (a) EMP\_DEPT and (b) EMP\_PROJ.

(a)

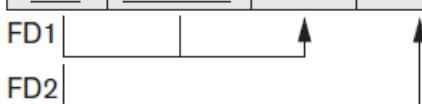
EMP\_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn

(b)

EMP\_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation



*Insert anomaly*

1. Employee 가 할당되지 않은 project 정보는 Insert 할 수 없다.  
Insert 하기 되면 Ssn의 사이즈로, Entity Constraint 위반

2. Project를 부여받지 않은 employee 정보는 Insert 할 수 있다.  
Insert 하기 되면 Pnumber가 Null이므로, Entity constraint 위반

*Delete Anomaly*  $\Rightarrow$  2개의 다른 정보들이 함께  
제거되어 발생하는 문제

- Project가 삭제되면 해당 project에 참여하고 있는 모든 employees 정보 또한 삭제될 것이다.
- 어떤 employee가 어떤 project에 참여하는 유일한 직원인 경우, 해당 employee 정보를 삭제한 그 project 정보 또한 삭제되게 된다.

## 튜플에서 중복된 정보와 이상(anomaly)

하나의 릴레이션에 하나 이상의 엔티티의 애트리뷰트들을 혼합하는 것은 여러 가지 문제를 일으킨다. (Fig 15.4.)

- 정보가 중복 저장되며, 저장 공간을 낭비하게 된다 (Fig 15.2의 EMPLOYEE와 DEPARTMENT  $\leftrightarrow$  15.3 및 15.4의 EMP\_DEPT 비교)
- 갱신 이상이 발생하게 된다; 동일한 정보를 한 릴레이션에는 변경하고, 나머지 릴레이션에서는 변경하지 않은 경우 어느 것이 정확한지 알 수 없게 된다.
- 갱신 이상의 종류
  - 삽입 이상 (insertion anomalies)
    - EMP\_DEPT에 객체를 삽입할 때 부서가 정해지지 않은 직원이나 직원이 없는 부서를 insert 하는데 문제가 발생함
  - 삭제 이상 (deletion anomalies)
    - 부서의 마지막 직원을 삭제하면 부서 정보도 없어짐
  - 수정 이상 (modification anomalies)
    - 부서 정보를 변경하면 부서의 모든 직원 투플에서 동일하게 변경해야 함

project number 10의 name은

"computerization"  $\&$  "Customer-Accounting"으로 변경하는 경우

project number 10에 종사하는 모든 employees를 update 해야 한다.

일관성의 원칙

abnormal 비정상적

anomaly 비정상

one information one place 原則

5번처럼 이름 Research → 4번만 복붙임.

Redundancy

EMP_DEPT						
Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

EMP_PROJ						
Ssn	Pnumber	Hours	Ename	Pname	Plocation	
123456789	1	32.5	Smith, John B.	ProductX	Bellaire	
123456789	2	7.5	Smith, John B.	ProductY	Sugarland	
666884444	3	40.0	Narayan, Ramesh K.	ProductZ	Houston	
453453453	1	20.0	English, Joyce A.	ProductX	Bellaire	
453453453	2	20.0	English, Joyce A.	ProductY	Sugarland	
333445555	2	10.0	Wong, Franklin T.	ProductY	Sugarland	
333445555	3	10.0	Wong, Franklin T.	ProductZ	Houston	
333445555	10	10.0	Wong, Franklin T.	Computerization	Stafford	
333445555	20	10.0	Wong, Franklin T.	Reorganization	Houston	
999887777	30	30.0	Zelaya, Alicia J.	Newbenefits	Stafford	
999887777	10	10.0	Zelaya, Alicia J.	Computerization	Stafford	
987987987	10	35.0	Jabbar, Ahmad V.	Computerization	Stafford	
987987987	30	5.0	Jabbar, Ahmad V.	Newbenefits	Stafford	
987654321	30	20.0	Wallace, Jennifer S.	Newbenefits	Stafford	
987654321	20	15.0	Wallace, Jennifer S.	Reorganization	Houston	
888665555	20	Null	Borg, James E.	Reorganization	Houston	

Figure 15.4

Sample states for EMP\_DEPT and EMP\_PROJ resulting from applying NATURAL JOIN to the relations in Figure 15.2. These may be stored as base relations for performance reasons.

## 튜플에서 널 값

Null

- 릴레이션의 튜플들이 널 값을 가지지 않도록 설계해야 함
  - 널 값은 저장 단계에서 공간을 낭비하게 되고 이미 값이 있으면 의미를 명확히 하기 어렵다.
  - 논리적 차원에서는 조인 연산들을 지정하기 힘들고 해석의 어려움
  - 애트리뷰트들의 의미를 이해하기 어려움
  - COUNT나 AVG와 같은 집단 함수들이 적용되었을 때 널 값의 해석이 모호함
  - 널 값은 다음과 같이 여러 가지로 해석이 가능함
    - 그 애트리뷰트가 이 튜플에는 적용되지 않는다. (존재 여부를 모른다)
    - 이 튜플에서 애트리뷰트의 값이 아직 알려져 있지 않다 (존재하지만 모른다).
    - 애트리뷰트 값이 알려져 있지만 DB에 기록되지는 않았다.
  - 모든 널 값을 동일하게 표현하면 널 값이 갖는 여러 의미를 훼손하게 된다.

## 튜플에서 널 값

- 널 값의 방지 기법 - 릴레이션의 분리

- 널 값이 많이 나타나는 애트리뷰트들은 별개의 릴레이션으로 분리함

1:1			
I	K	30	100

Employee(ssn, ename, age, office\_no)

↳ 한 번에 여러 개의 열(행)으로 office\_no 있는 업무를 개별적 office 할당된 4값

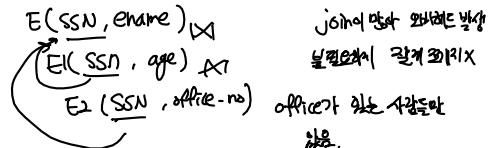
분리하는 경우 1:1 관계

대부분의 Employee는 개별 office\_no를 가지고 있지 않다. 위의 Employee를 아래와 같이 두 개로 분리하는 것이 좋다.

- Employee(ssn, ename, age)

→ 모든 행은 의미가 많아서 좋겠다.

Emp\_Office(ssn, office\_no)



→ 각각의 행은 의미가 많아서 좋겠다.

↑ 각각의 행은 의미가 많아서 좋겠다.

# 가짜 투플 (spurious tuples)

- 관계 데이터베이스 설계를 잘못하게 되면, 조인 연산들이 틀린 결과를 생성할 수 있다.
- 조인 연산의 결과가 올바르기 위해서는, 릴레이션들이 “무손실 조인(lossless join)” 조건을 만족하도록 설계되어야 한다.

(a)

EMP\_LOCS

Ename	Plocation

P.K.



EMP\_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation

P.K.

join attribute  
Plocation.

아예 합계가 아니요

이상한 내용들이 들어온다.

Figure 15.5

Particularly poor design for the EMP\_PROJ relation in Figure 15.3(b). (a) The two relation schemas EMP\_LOCS and EMP\_PROJ1. (b) The result of projecting the extension of EMP\_PROJ from Figure 15.4 onto the relations EMP\_LOCS and EMP\_PROJ1.

(b)

EMP\_LOCS  $\leftarrow \pi_{...}(\text{EMP\_PROJ})$



Ename	Plocation
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelaya, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

EMP\_PROJ1  $\leftarrow \pi_{...}(\text{EMP\_PROJ})$

설계와 맞지 않아

Ssn	Pnumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
999887777	30	30.0	Newbenefits	Stafford
999887777	10	10.0	Computerization	Stafford
987987987	10	35.0	Computerization	Stafford
987987987	30	5.0	Newbenefits	Stafford
987654321	30	20.0	Newbenefits	Stafford
987654321	20	15.0	Reorganization	Houston
888665555	20	NULL	Reorganization	Houston

따라서, join operation의 meaningful results를 보장하기 위해 "loseless join property" (join이, 소외는 경과 있는 투영)  
\* 사용한다.

Ssn	Pnumber	Hours	Pname	Plocation	Ename
123456789	1	32.5	ProductX	Bellaire	Smith, John B.
*	123456789	1	32.5	ProductX	Bellaire English, Joyce A.
123456789	2	7.5	ProductY	Sugarland	Smith, John B.
*	123456789	2	7.5	ProductY	Sugarland English, Joyce A.
*	123456789	2	7.5	ProductY	Sugarland Wong, Franklin T.
666884444	3	40.0	ProductZ	Houston	Narayan, Ramesh K.
*	666884444	3	40.0	ProductZ	Houston Wong, Franklin T.
*	453453453	1	20.0	ProductX	Bellaire Smith, John B.
453453453	1	20.0	ProductX	Bellaire English, Joyce A.	
*	453453453	2	20.0	ProductY	Sugarland Smith, John B.
453453453	2	20.0	ProductY	Sugarland English, Joyce A.	
*	453453453	2	20.0	ProductY	Sugarland Wong, Franklin T.
*	333445555	2	10.0	ProductY	Sugarland Smith, John B.
*	333445555	2	10.0	ProductY	Sugarland English, Joyce A.
333445555	2	10.0	ProductY	Sugarland Wong, Franklin T.	
*	333445555	3	10.0	ProductZ	Houston Narayan, Ramesh K.
333445555	3	10.0	ProductZ	Houston Wong, Franklin T.	
333445555	10	10.0	Computerization	Stafford Wong, Franklin T.	
*	333445555	20	10.0	Reorganization	Houston Narayan, Ramesh K.
333445555	20	10.0	Reorganization	Houston Wong, Franklin T.	

\* \* \* EMP\_PROJ1을 EMP\_LOCS와 EMP\_PROJ1로 병합

• EMP\_LOCS \* EMP\_PROJ1 → 중복발생, 이중한 Tuples 발생 가능

Figure 15.6

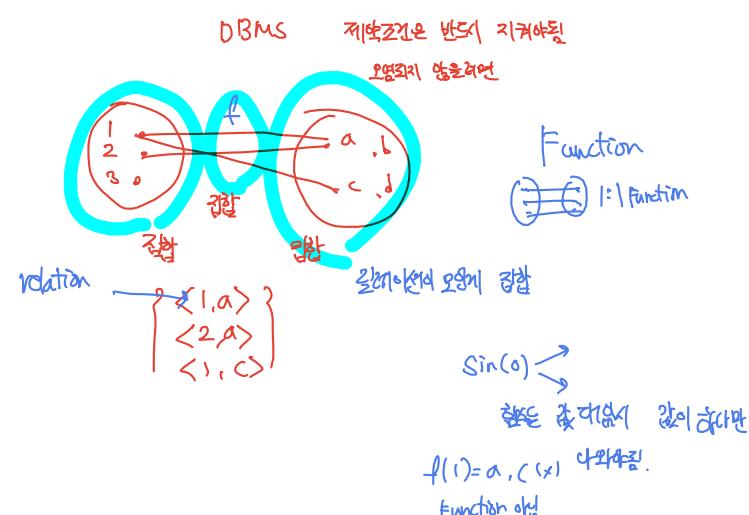
Result of applying NATURAL JOIN to the tuples above the dashed lines in EMP\_PROJ1 and EMP\_LOCS of Figure 15.5. Generated spurious tuples are marked by asterisks.

"Inconsistency". 두 relation의 incorrect original Information을 생성하기 때문이다.  
• Location은 두 relation에서 PK, FK로 선택된다.  
• 이는 곧 적절한 Attribute를 PK나, FK로 선택해야 한다는 것이다

## 함수적 종속성

key or candidate key PK(X)

- 함수적 종속성(FD: functional dependency)은 좋은 릴레이션 설계의 정형적 기준으로 사용된다.
- FD와 키는 릴레이션의 정규형을 정의하기 위해 사용된다.
- FD는 데이터 애트리뷰트들의 의미와 애트리뷰트들 간의 상호 관계로부터 유도되는 제약조건(constraints)의 일종이다.
- 구성
  - 함수의 종속성(functional dependency)의 정의
  - 함수적 종속성의 추론 규칙
  - 함수적 종속성 집합의 동등성
  - 함수적 종속성의 최소집합



keys는 relations의 normal form을 정하는게 사용

X의 value가 Y의 "unique" value를 결정하면

"attribute X의 값은 attribute Y의 값을 functionally determines"

## 함수적 종속성의 정의

{ SSN, Pnumber }

### • 함수적 종속성

- X와 Y를 임의의 애트리뷰트들의 집합이라고 할 때, X의 값이 Y의 값을 유일하게(unique) 결정한다면

"X는 Y를 함수적으로 결정한다(functionally determines)"라고 함 (X, Y는 R의 subset)

- X → Y로 표기하고, "Y는 X에 함수적으로 종속된다"라고 함 (값은 X의 값에만 관계)

- 함수적 종속성은 모든 릴레이션 인스턴스 r(R)에 대하여 성립해야 함

### • 함수적 종속성의 검사 방법

- 릴레이션 인스턴스 r(R)에 속하는 어떠한 임의의 두 투플에 대해서도 속성들의 집합 X에 대해 동일한

값을 가질 때마다 Y에 대해서도 동일한 값을 가진다면 X → Y라는 함수적 종속성이 성립한다.

- 즉, r(R)에서의 임의의 두 투플  $t_1$ 과  $t_2$ 에 대해  $t_1[X] = t_2[X]$ 이면,  $t_1[Y] = t_2[Y]$ 이다.

### • FD는 특정 릴레이션 인스턴스보다는 실세계에서 존재하는 애트리뷰트들 사이의 제약조건으로부터 유도된다.

### • FD 제약조건의 예제

- 주민등록번호는 사원의 이름을 결정한다.

○ SSN → ENAME

- 프로젝트 번호는 프로젝트 이름과 위치를 결정한다.

○ PNUMBER → {PNAME, PLOCATION}

- 사원의 주민등록번호와 프로젝트 번호는 그 사원이 일주일동안 그 프로젝트를 위해서 일하는 시간을 결정한다.

○ {SSN, PNUMBER} → HOURS

- FD는 스키마 R에 있는 애트리뷰트들의 특성이며, 모든 릴레이션 인스턴스 r(R)에서 성립해야 하는 성질이다.

- K가 R의 키라면 K는 R의 모든 애트리뷰트들을 함수적으로 결정한다 → 키에 대해서는 동일한 값을 갖는 두개의  $(t_1[K] = t_2[K])$ 인 서로 다른 두개의 투플이 존재하지 않기 때문에). "distinct" tuple을 가질 수 없기 때문이다.

$X \rightarrow Y$ 라고 해서 반드시  $Y \rightarrow X$ 인 것은  $K$ 가 R의 candidate key라면  $K \rightarrow R$

아니다

## 함수적 종속성의 추론 규칙

- 설계자는 주어진(알려진) FD의 집합 F를 가지고, 추가로 성립하는 FD들을 추론할 수 있다.

### • 암스트롱의 추론 규칙들

- A1. (재귀성 규칙)  $Y \subseteq X$ 이면,  $X \rightarrow Y$ 이다.
- A2. (부가성 규칙)  $X \rightarrow Y$ 이면,  $XZ \rightarrow YZ$ 이다. (표기:  $XZ$ 는  $X \cup Z$ 를 의미)
- A3. (이행성 규칙)  $X \rightarrow Y$ 이고  $Y \rightarrow Z$ 이면,  $X \rightarrow Z$ 이다.

- A1, A2, A3는 sound하고 complete 추론 규칙 집합을 형성한다.

### • 추가적으로 유용한 추론 규칙들

- (분해 규칙)  $X \rightarrow YZ$ 이면,  $X \rightarrow Y$ 이고  $X \rightarrow Z$ 이다.
- (합집합 규칙)  $X \rightarrow Y$ 이고  $X \rightarrow Z$ 이면,  $X \rightarrow YZ$ 이다.
- (의사이행성 규칙)  $X \rightarrow Y$ 이고  $WY \rightarrow Z$ 이면,  $WX \rightarrow Z$ 이다.

- 위의 세 규칙 뿐 아니라 다른 추론 규칙들도 A1, A2, A3으로부터 추론 가능하다 (완전성 특성).

- FD의 집합 F의 폐포(closure) :  $F^+$  F가 함수적 종속성의 집합이라 유도되며 모든 집합을 증거화 시킬 범위
  - F로부터 추론할 수 있는 모든 가능한 함수적 종속성들의 집합

- F 하에서 속성 집합 X의 폐포(closure of X under F) :  $X^+$  X^+ under F 증거화
  - 함수적 종속성 집합 F를 사용하여 X에 의해 함수적으로 결정되는 모든 애트리뷰트들의 집합

$X^+ \text{ under } F = \{X, Y, W, Z\}$   
① 이행성 규칙  
예제:  $X \rightarrow Y$ ,  $Y \rightarrow W$ ,  $W \rightarrow Z$   
여기서  $Z$ 를  $W$ 로  $W$ 를  $Y$ 로  $Y$ 를  $X$ 로 증명할 수 있다.  
② 예제:  $X \rightarrow Y$ ,  $Y \rightarrow Z$ ,  $Z \rightarrow W$   
여기서  $W$ 를  $Z$ 로  $Z$ 를  $Y$ 로  $Y$ 를  $X$ 로 증명할 수 있다.  
③ 예제:  $X \rightarrow Y$ ,  $Y \rightarrow Z$ ,  $Z \rightarrow W$ ,  $W \rightarrow V$   
여기서  $V$ 를  $Z$ 로  $Z$ 를  $Y$ 로  $Y$ 를  $X$ 로 증명할 수 있다.  
④ 예제:  $X \rightarrow Y$ ,  $Y \rightarrow Z$ ,  $Z \rightarrow W$ ,  $W \rightarrow V$ ,  $V \rightarrow U$   
여기서  $U$ 를  $Z$ 로  $Z$ 를  $Y$ 로  $Y$ 를  $X$ 로 증명할 수 있다.

주어진 relation instance에 대해서, FD는

특정 attributes 사이에 조건할 수도 있다:

in CAR, ? State, Driver\_license\_number? → ? SSN?

dependencies의 위법을 보는 tuples로 인해 특정

$A \rightarrow B$   
 $B \rightarrow C$

Sech, IPN → 강의실

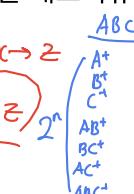
①  $\{A \rightarrow B, B \rightarrow C\}$  두번만 감지되었음.  
증거화 → 강의실 탐지.  
 움드렸을 수 있음.

②  $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$

$R(A, B, C)$

$F: A \rightarrow B, B \rightarrow C$

$F^+ = \{A \rightarrow B, B \rightarrow C, C \rightarrow C\}$   
 $AB \rightarrow A, AB \rightarrow B, A \rightarrow C$



## 알고리즘 : F하의 $X^+$ 를 구하는 알고리즘

프로그램의 앞에서 결정한다.

이 알고리즘은 complete 하다.

일부러 Input 시킬 필요 없어

$$X^+ := X \text{ 재귀성 규칙}$$

repeat

$$oldX^+ := X^+ \text{ 지금까지 구한 } X^+ \text{을 } oldX^+ \text{에 Save}$$

for each functional dependency  $Y \rightarrow Z$  in  $F$  do 5번 반복문에  
if  $Y \subseteq X^+$  then  $X^+ := X^+ \cup Z$  돌아갔다.

until ( $oldX^+ = X^+$ )

파이썬에 Set 이용해 고대로 쪽으면 됨

$$FD = \{AB \rightarrow C, AC \rightarrow AD, CD \rightarrow F, A \rightarrow E, D \rightarrow E\}$$

$$Y \rightarrow Z$$

$$AB^+ \text{ under } F = ?$$

$$Y \rightarrow Z$$

$$X^+ \rightarrow Y \text{ 재귀성 (복잡성을 줄여야 함)}$$

$$\textcircled{1} \quad X^+ = ?A, B, C, A, D, F, E, ?$$

$$oldX^+ = ?A, B ?$$

$$X^+ \neq oldX^+ = ?A, B$$

$$X^+ \rightarrow oldX^+ = ?A, B, C, D, E, F ?$$

$$X^+ = ?A, B, C, D, E, F ?$$

$$X^+ = oldX^+ \text{ closure}$$

$AB^+ = A$ 가 수퍼키인 사실은 이전부터 알았음.  
 $R(A, B, C, D, E, F)$  수퍼키

Example :

$F =$

$$SSN \rightarrow ENAME,$$

$$PNUMBER \rightarrow PNAME, PLOCATION,$$

$$SSN, PNUMBER \rightarrow HOURS$$

알고리즘을 사용하여  $F$ 하에서 다음과 같은 폐포 집합들을 구할 수 있다.

- $SSN^+ = SSN, ENAME$

- $PNUMBER^+ = PNUMBER, PNAME, PLOCATION$

- $SSN, PNUMBER^+ = SSN, PNUMBER, ENAME, PNAME, PLOCATION, HOURS$

$$\textcircled{2} \quad AB^+ = ABCDEF$$

$$AD \not\subseteq A$$

$$A^+ = ?A, E ?$$

$$\textcircled{3} \quad B^+ = ?B, E ?$$

$$AB^+ = ABCDEF \rightarrow S.K$$

$$A^+ = AE \rightarrow C.K$$

$$B^+ = BE \rightarrow C.K$$

$$F^+ = G^+ \text{로부터 유도되어지는 것들이 같아서}$$

동등하고 학습 가능.

두개가 의미하는 바가 같다면 동등하다는 것을

확인할 수 있음.

주제로 검사해야 하는 계약조건에서 계산하기 험수도 있는 것을 빼고

최소화하고 싶을 때.

## 함수적 종속성 집합의 동등성

• 정의: cover

▪  $G$ 의 모든 FD가  $F$ 로부터 추론될 수 있다면(즉,  $G^+ \subseteq F^+$ ) “ $F$ 가  $G$ 를 덮는다(cover)”라고 말한다.  $minimality$  참고.

• 두 FD 집합의 동등성

▪ FD의 집합  $F$ 와  $G$ 에 대하여,  $F$ 의 모든 FD가  $G$ 로부터 추론될 수 있고,

•  $G$ 의 모든 FD가  $F$ 로부터 추론될 수 있으면 “ $F$ 와  $G$ 는 동등하다(equivalent)”라고 한다

▪  $F$ 와  $G$ 가 다르더라도  $F^+ = G^+$  이면  $F$ 와  $G$ 는 동등하다.

▪  $F$ 가  $G$ 를 덮고  $G$ 가  $F$ 를 덮으면  $F$ 와  $G$ 는 동등하다.

• FD 집합의 동등성을 검사하는 알고리즘이 존재함

$$F = ?A \rightarrow B, B \rightarrow C, A \rightarrow C, A \rightarrow A ?$$

$$G = ?A \rightarrow B, B \rightarrow C, AB \rightarrow A ?$$

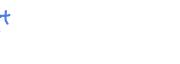
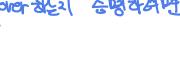
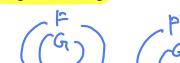
$$H = ?A \rightarrow B, B \rightarrow C ?$$

▪  $H$ 를 유지하고 싶음. 주제로 검사해야 하는 계약조건에서 계산하기 험수도 있는 것을 빼고

최소화하고 싶을 때.

▪ 두개가 의미하는 바가 같다면 동등하다는 것을

확인할 수 있음.



## 함수적 종속성의 최소집합

- 함수적 종속성들의 집합 F의 최소(minimal)
  - 다음 세 조건을 만족하는 FD 집합을 최소라고 함  $X \rightarrow Y$
  - F의 모든 함수적 종속성들의 오른쪽면 애트리뷰트가 하나이다.
  - F로부터 어떤 함수적 종속성을 제거했을 때, F와 동등한 함수적 종속성들의 집합이 될 수 없다.
  - F에서 X → A를 X의 진부분집합 Y에 대하여 Y → A로 교체했을 때, F와 동등한 함수적 종속성들의 집합이 될 수 없다.
- 함수적 종속성들의 집합 F의 최소 덮개(minimal cover)는 F와 동등한 함수적 종속성들의 최소 집합  $F_{min}$ 을 의미함
  - 함수적 종속성들의 최소 덮개는 여러 개 존재할 수 있다.
  - 또한, 임의의 함수적 종속성들의 집합 F에 대해 알고리즘을 사용하여 적어도 하나의 최소 덮개를 구할 수 있다

$E : \{B \rightarrow A, D \rightarrow A, AB \rightarrow D\}$   
E'에서 유도되어질수 있다.  
 $E' : \{B \rightarrow A, D \rightarrow A, A \rightarrow D, D \rightarrow A\}$   
 $AB^+ \text{ under } E' = ABD$   
E'에서 유도되어질수 있다.  
 $A^+ \text{ under } E = A$   
 $AB \rightarrow ABD \rightarrow D$   
 $B^+ \text{ under } E = BAD$   
E'에서 유도되어질수 있다.  
 $E'' : \{D \rightarrow A, B \rightarrow D\}$   
 $E'' \text{에서 유도되어질수 있다. } D \rightarrow A \text{이다.}$   
 $B^+ \text{ under } E'' = B, D, A$   
 $B \rightarrow BDA \rightarrow A$   
 $F = \{X \rightarrow Y, V \rightarrow V\}$   
 $F' = \{U \rightarrow V\}$

## 기본 키를 기반으로 한 정규형

- 정규화 소개
- 제1 정규형(First Normal Form ; 1NF)
- 제2 정규형(Second Normal Form ; 2NF)
- 제3 정규형(Third Normal Form ; 3NF)

## 정규화 소개

- 정규화(normalization)
  - 불만족스러운 “나쁜” 릴레이션의 애트리뷰트들을 나누어서 더 작은 “좋은” 릴레이션으로 분해하는 과정
- 정규형(normal form)
  - 특정 조건을 만족하는 릴레이션 스키마의 형태
- 제 2 정규형, 제 3 정규형, BCNF
  - 릴레이션 스키마의 FD와 키에 기반하여 정의됨
- 좋은 릴레이션 설계를 위해서는 정규형 이외에도 추가 특성이 필요함 (즉, 무손실 조인과 종속성 보존; 다음 장 참고)

## 제 1 정규형(1NF)

atomic ×

- 제 1 정규형
  - 복합 애트리뷰트(composite attribute), 다치 애트리뷰트(multivalue attribute), 그리고 중첩 릴레이션(nested relation) 등 비원자적(non-atomic) 애트리뷰트들을 허용하지 않은 릴레이션의 형태
- 제 1 정규형은 릴레이션 정의의 일부분을 이루고 있음

(a)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations

(b)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

1NF

D1(Dnumber, Dlocation)

D2(Dname, Dnumber, Dmgrssn)

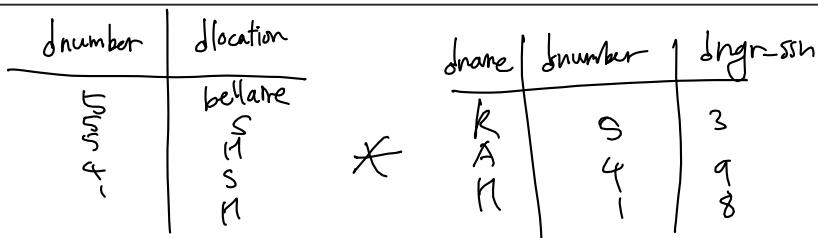
(c)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	<u>Dlocation</u>
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 15.9

Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.



**(a)**

EMP_PROJ		Projs	
Ssn	Ename	Pnumber	Hours

**(b)**

EMP_PROJ			
Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

**Figure 15.10**

Normalizing nested relations into 1NF. (a) Schema of the EMP\_PROJ relation with a *nested relation* attribute PROJS. (b) Sample extension of the EMP\_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP\_PROJ into relations EMP\_PROJ1 and EMP\_PROJ2 by propagating the primary key.

**(c)**

EMP_PROJ1		
Ssn	Ename	
<b>EMP_PROJ2</b>		
Ssn	Pnumber	Hours

1NF Nested relation 복합 키 attribute

$D^+ \subset E$

## 제 2 정규형(2NF)

- 제 2 정규형은 FD와 기본키 개념을 이용한다.

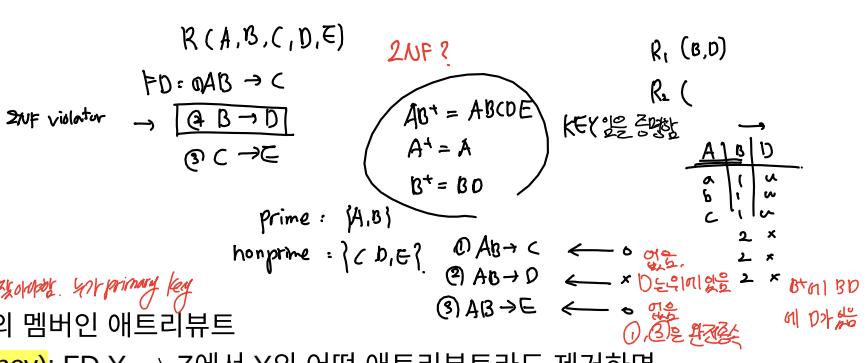
- 정의:

① 기본 애트리뷰트를 포함하는都不是 primary key

- 기본 애트리뷰트(prime attribute): 기본키 K의 멤버인 애트리뷰트
- 완전 함수적 종속성(full functional dependency): FD  $Y \rightarrow Z$ 에서 Y의 어떤 애트리뷰트라도 제거하면 더이상 성립하지 않는 경우

- 예제:

- {SSN, PNUMBER}  $\rightarrow$  HOURS는 SSN  $\rightarrow$  HOURS와 PNUMBER  $\rightarrow$  HOURS가 성립하지 않기 때문에 완전 함수적 종속성이다.
- {SSN, PNUMBER}  $\rightarrow$  ENAME은 SSN  $\rightarrow$  ENAME이 성립하기 때문에 완전 함수적 종속성이 아니다 (이는 부분 함수 종속성(partial function dependency)이라고 부름).
- 릴레이션 스키마 R의 모든 비기본(non-prime) 애트리뷰트들이 기본키에 대해서 완전 함수적 종속이면, R은 제 2 정규형(2NF)에 속한다.
- R은 제 2 정규형 정규화 과정에 의해서 항상 제 2 정규형 릴레이션으로 분해될 수 있다.



SCR  $S \rightarrow Y$  Y의 전부분집합을 제거하면 결점x

무언을 빼도 상관없으면

2NFx

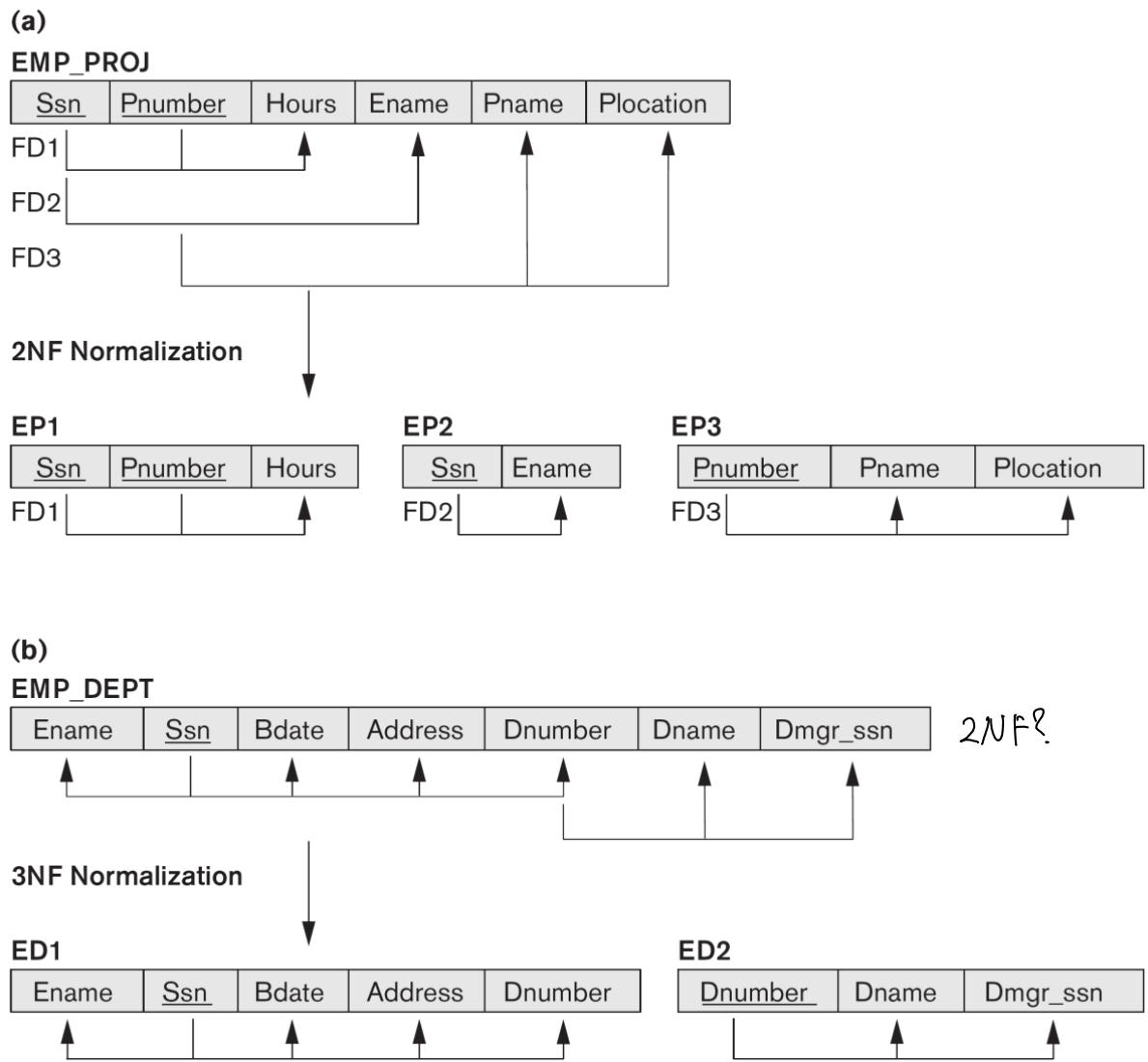


Figure 15.11

Normalizing into 2NF and 3NF. (a) Normalizing EMP\_PROJ into 2NF relations. (b) Normalizing EMP\_DEPT into 3NF relations.

## 제 3 정규형(3NF)



- 정의
  - 이행적 함수적 종속성(transitive functional dependency): 두 FD  $Y \rightarrow X$ 와  $X \rightarrow Z$ 에 의해서 추론될 수 있는 FD  $Y \rightarrow Z$
- 예제
  - $SSN \rightarrow DMGRSSN$ 은  $SSN \rightarrow DNUMBER$ 과  $DNUMBER \rightarrow DMGRSSN$ 이 성립하기 때문에 이 행적 함수적 종속성이다.
  - $SSN \rightarrow ENAME$ 은  $SSN \rightarrow X$ 이고  $X \rightarrow ENAME$ 인 애트리뷰트 집합 X가 존재하지 않기 때문에 이 행적 종속성이 아니다.
- 릴레이션 스키마 R이 제 2 정규형을 갖고 R의 어떤 비기본 애트리뷰트도 기본키에 대해서 이행적으로 종속되지 않으면 R은 제 3 정규형을 갖는다고 한다.
- R은 제 3 정규형 정규화 과정에 의해서 항상 제 3 정규형 릴레이션으로 분해될 수 있다

## 제 2 정규형과 제 3 정규형의 일반적인 정의

- 여기서부터는 여러개의 후보 키를 가진 릴레이션들을 고려한다.
- 릴레이션 스키마 R의 모든 비기본 애트리뷰트 A가 R의 모든 후보키에 완전 함수적 종속이면 R은 제 2 정규형(2NF)을 갖는다고 한다.
- 정의:
  - 기본 애트리뷰트(prime attribute): 임의의 후보키 K의 멤버인 애트리뷰트
  - 릴레이션 스키마 R의 수퍼키(superkey): R의 후보키를 포함한 R의 애트리뷰트들의 집합 S
  - 릴레이션 스키마 R의 FD  $X \rightarrow A$ 가 성립할 때마다
    - (a) X 가 R의 수퍼키 이거나
    - (b) A가 R의 기본 애트리뷰트이면 R은 제 3 정규형(3NF)을 갖는다고 한다.
- Boyce-Codd 정규형은 위의 조건 중 (b)의 경우를 허락치 않는 정규형을 의미한다.  
반드시 만족해야 함.

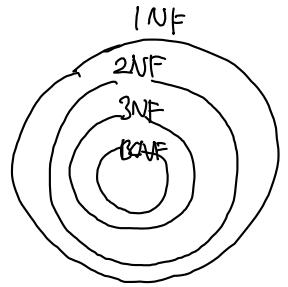


Figure 15.12

Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.

(a) **다익부지, 땅**

**Candidate Key**

LOTS					
Property_id#	County_name	Lot#	Area	Price	Tax_rate
FD1					
FD2					
FD3					
FD4					

CK: {Property\_id#, County-name, Lot#}

Prime.: {pid, cn, lot#}       $pid \rightarrow TR$

2NF?

$CN^+$

Example.

①  $R(A, B, C, D)$

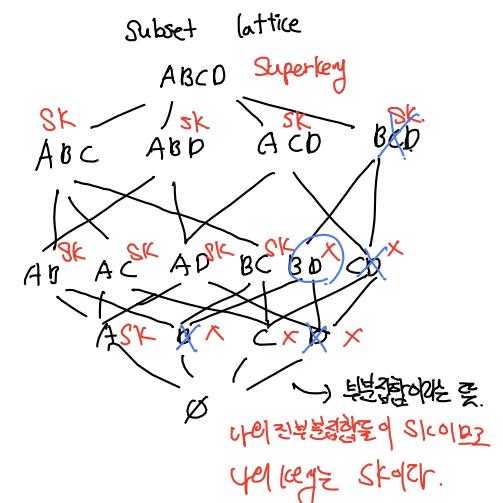
FD:  $A \rightarrow B$

$A \rightarrow C$

$A \rightarrow D$

$BC \rightarrow A$

CK?



(b)

**LOTS1**

<u>Property_id#</u>	County_name	Lot#	Area	Price
FD1				
FD2				
FD4				

**LOTS2**

County_name	Tax_rate
FD3	

(c)

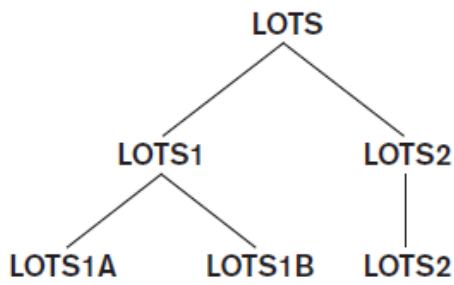
**LOTS1A**

<u>Property_id#</u>	County_name	Lot#	Area
FD1			
FD2			

**LOTS1B**

Area	Price
FD4	

(d)



1NF

2NF

3NF

## BCNF (Boyce-Codd Normal Form)

- 릴레이션 스키마 R에서 성립하는 임의의 FD  $X \rightarrow A$ 에서 X가 R의 수퍼키이면 R은 Boyce-Codd 정규형 (BCNF)을 갖는다고 한다.
- 각 정규형은 그의 선행 정규형보다 더 엄격한 조건을 갖는다. 즉,
  - 모든 제 2 정규형 릴레이션은 제 1 정규형을 갖는다.
  - 모든 제 3 정규형 릴레이션은 제 2 정규형을 갖는다.
  - 모든 BCNF 릴레이션은 제 3 정규형을 갖는다.
- 제 3 정규형에는 속하나 BCNF에는 속하지 않는 릴레이션이 존재한다.
- 관계 데이터베이스 설계의 목표는 각 릴레이션이 BCNF(또는 제 3 정규형)를 갖게 하는 것이다.
- “좋은” 관계형 데이터베이스의 릴레이션을 설계하기 위해서는 추가적인 특성이 만족되어야 한다.
  - 무손실 조인(lossless join) 특성
  - 종속성 보존(dependency preservation) 특성

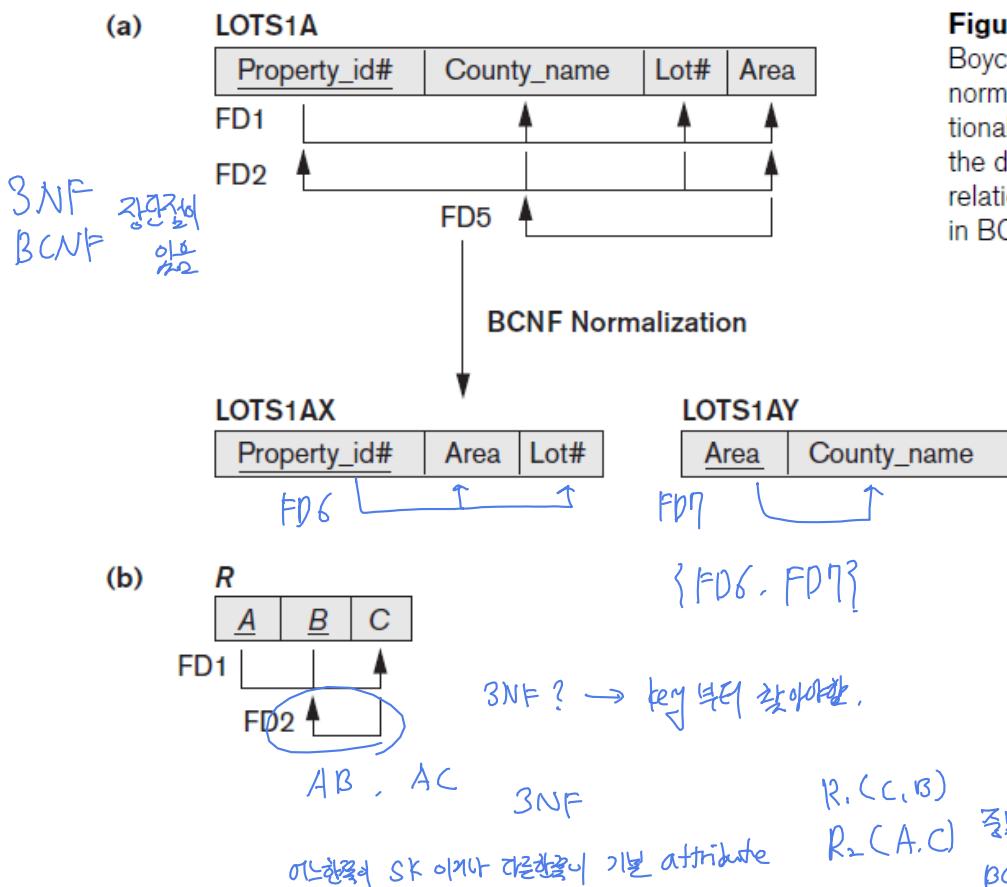


Figure 15.13

Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

## Relation in 3NF but not BCNF

**Figure 15.14**

A relation TEACH that is in 3NF but not BCNF.

**TEACH**

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

FD:

$\text{Student}, \text{Course} \rightarrow \text{Instructor}$

$\text{Instructor} \rightarrow \text{Course}$

기본적임      허수적      종속성찾기

## **Exercise 1**

Consider a relation  $R(A, B, C, D)$ , with FDs  $AB \rightarrow C$ ,  $BC \rightarrow D$ ,  $CD \rightarrow A$ .

- (a) Find the closure of  $AB$ .
- (b) Is  $R$  a good schema?
- (c) If we decompose  $R$  as  $R1(A,B,C)$  and  $R2(A,C,D)$ . Is it a good decomposition?

## **Exercise 2**

- Consider relation  $R(A,B,C,D,E)$  with the following functional dependencies:  $AB \rightarrow C$ ,  $D \rightarrow E$ ,  $DE \rightarrow B$ .
- Is  $R$  in BCNF? If not, decompose  $R$  into a collection of BCNF relations.

## **Exercise 3**

- “Any two-attribute relation is in BCNF.” Is it correct?

## **Exercise 4**

Compute the closure of the following set  $F$  of functional dependencies for relation schema  $R = \{A, B, C, D, E\}$ .

$A \rightarrow BC$   
 $CD \rightarrow E$   
 $B \rightarrow D$   
 $E \rightarrow A$

List the candidate keys for  $R$ .

## **Exercise 5**

Consider a relation  $R(A,B,C,D,E)$  with the following dependencies:

$\{AB \rightarrow C, CD \rightarrow E, DE \rightarrow B\}$

List all candidate keys.

## **Exercise 6**

$R(A,B,C,D)$  and FDs  $\{AB \rightarrow C, C \rightarrow D, D \rightarrow A\}$ .

- (1) List all nontrivial FDs that can be inferred from the given FDs.
- (2) Find all candidate keys.
- (3) Find all BCNF violations.
- (4) Decompose  $R$  into relations in BCNF.
- (5) What FDs are not preserved by BCNF.

# 14장. 관계형 데이터베이스 설계 알고리즘(Relational Database Design Algorithms)

## 관계형 데이터베이스 스키마 설계 알고리즘

- 좋은 데이터베이스 스키마 설계를 위해서는 정규형만으로는 불충분함
- 예제: 2개의 애트리뷰트를 갖는 릴레이션은 BCNF이다. 그러면, 모든 릴레이션을 2개 애트리뷰트로 설계하면 좋은 설계인가?
- 좋은 데이터베이스 설계를 보장하기 위해서는 다음의 추가적인 조건들이 필요함
  - 종속성 보존 특성 (dependency preservation property)
  - 무손실 조인 특성 (lossless join property)

## 릴레이션 분해와 정규형의 부족한 점

- 전체 릴레이션 스키마(universal relation schema)  $R = A_1, A_2, \dots, A_n$ 
  - 데이터베이스의 모든 애트리뷰트들을 포함하는 릴레이션이다.
  - 분해는 전체 릴레이션  $R$ 로부터 시작한다.
- 설계목표 I:  $R$ 을  $m$  개의 릴레이션 스키마의 집합인 분해집합  $D = R_1, R_2, \dots, R_m$ 로 분해한다.
  - 각 릴레이션 스키마  $R_i$ 는  $R$ 의 애트리뷰트들의 부분집합이다.
  - 애트리뷰트 보존 조건:  $R$ 의 모든 애트리뷰트들은 적어도 하나의 릴레이션  $R_i$ 에 나타나야 한다.
- 설계목표 II:  $D$ 의 각 릴레이션  $R_i$ 는 BCNF 혹은 3NF에 속하도록 한다.
  - 좋은 데이터베이스 설계는 이를 정규형만으로 부족하다.
  - 예제: 애트리뷰트가 둘 뿐인 릴레이션은 자동적으로 BCNF에 속하게 된다. 이러한 애트리뷰트가 둘인 릴레이션을 조인하게 되면 가짜 투플이 만들어질 수 있다.

## 분해와 종속성 보존

- 데이터베이스 설계자는  $R$ 의 애트리뷰트들에 대해 성립하는 함수적 종속성의 집합  $F$ 를 정의한다.
- 분해집합  $D$ 는 종속성을 보존해야 한다
  - 즉, 각 릴레이션  $R_i$ 에서 성립하는 모든 함수적 종속성들의 합집합이  $F$ 와 동등 (equivalent)해야 한다.
- 종속성 보존은 정형적으로 다음과 같이 정의한다.
  - $R_i$  상으로  $F$ 의 프로젝션( $\Pi_{R_i}(F)$ )은  $F^+$ 에 속하는 함수적 종속성  $X \rightarrow Y$ 들의 집합이며  $(X \cup Y) \subseteq R_i$ 를 만족한다.
  - 만약  $(\Pi_{R_1}(F) \cup \Pi_{R_2}(F) \cup \dots \cup \Pi_{R_m}(F))^+ = F^+$ 를 만족하면, 분해 집합  $D = R_1, R_2, \dots, R_m$ 은 종속성을 보존한다고 정의한다.
  - 이런 특성으로 인해 각 릴레이션  $R_i$  상의 함수적 종속성들이 개별적으로 성립함을 보임으로써,  $F$ 의 함수적 종속성들이 성립함을 보장할 수 있다

## Claim 1:

- 종속성들의 집합  $F$ 에 대해서 종속성을 보존하면서 각 릴레이션  $R_i$  가 제 3 정규형인 한 분해집합  $D$ 를 항상 구할 수 있다.
- 관계 합성(relational synthesis) 알고리즘

- Find a minimal set of FDs  $G$  equivalent to  $F$
- For each  $X$  of an FD  $X \rightarrow A$  in  $G$ ,  
create a relation schema  $R_i$  in  $D$  with the attributes  
 $\{X \cup \{A_1\} \cup \{A_2\} \cup \dots \cup \{A_k\}\}$ , where  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$  are the  
only dependencies in  $G$  with  $X$  as left-hand-side;
- Place any remaining attributes in a single relation schema to ensure the attribute preservation  
property.

## Claim 1A:

- 관계 합성 알고리즘에 의해 생성되는 모든 릴레이션 스키마는 3NF에 속한다.
- 문제점:
  - $F$ 에 대한 최소 폐쇄(minimal cover)를 찾아야 한다.
  - 최소 폐쇄를 찾는 효율적인 알고리즘이 존재하지 않는다.
  - $F$ 에 대한 여러 개의 최소 폐쇄가 존재할 수 있다. 따라서, 알고리즘의 결과는 어떤 함수적 종속성이 선택되었는가에 따라 다르다.

## 분해와 무손실 (비부가적) 조인

- 무손실 조인의 비정형적 정의:
  - 분해집합의 릴레이션들을 조인했을 때 어떠한 가짜 투플도 생성되지 않음을 보장하는 특징이다.  $\Rightarrow$  원래 티비 블로 조합화한다
- 무손실 조인의 정형적 정의:
  - 함수적 종속성의 집합  $F$ 를 만족하는 모든 릴레이션 상태  $r$ 에 대해 다음의 성질이 만족되면,  $R$ 의 분해집합  $D = \{R_1, R_2, \dots, R_m\}$ 은  $F$ 에 대해서 무손실 조인 특성을 갖는다. (\*는  $D$ 에 포함된 모든 릴레이션의 자연조인이다.)  
*만약  $m$ 개를 자연조인시켰더라면 원래대로 돌아온다*  
$$*(\Pi_{R_1}(r), \dots, \Pi_{R_m}(r)) = r$$
- 분해집합이 위의 성질을 만족하면, 프로젝션과 조인 연산 후에 가짜 투플이 추가되지 않음이 보장된다.
- 분해된 릴레이션들은 실제로는 기본 릴레이션으로 저장하기 때문에, 조인을 수반한 질의가 의미있는 결과를 생성하기 위해서는 위와 같은 조건이 필요하다.
- 주어진 분해집합  $D$ 가 함수적 종속성 집합  $F$ 에 대해서 무손실 조인 특성을 가지는 것을 검사하는 알고리즘이 존재한다.

R<sub>1</sub>

$e_1$	100	1	2
$e_2$	200	10	3

R<sub>2</sub>

p <sub>1</sub>	p <sub>x</sub>	s
p <sub>2</sub>	p <sub>y</sub>	s

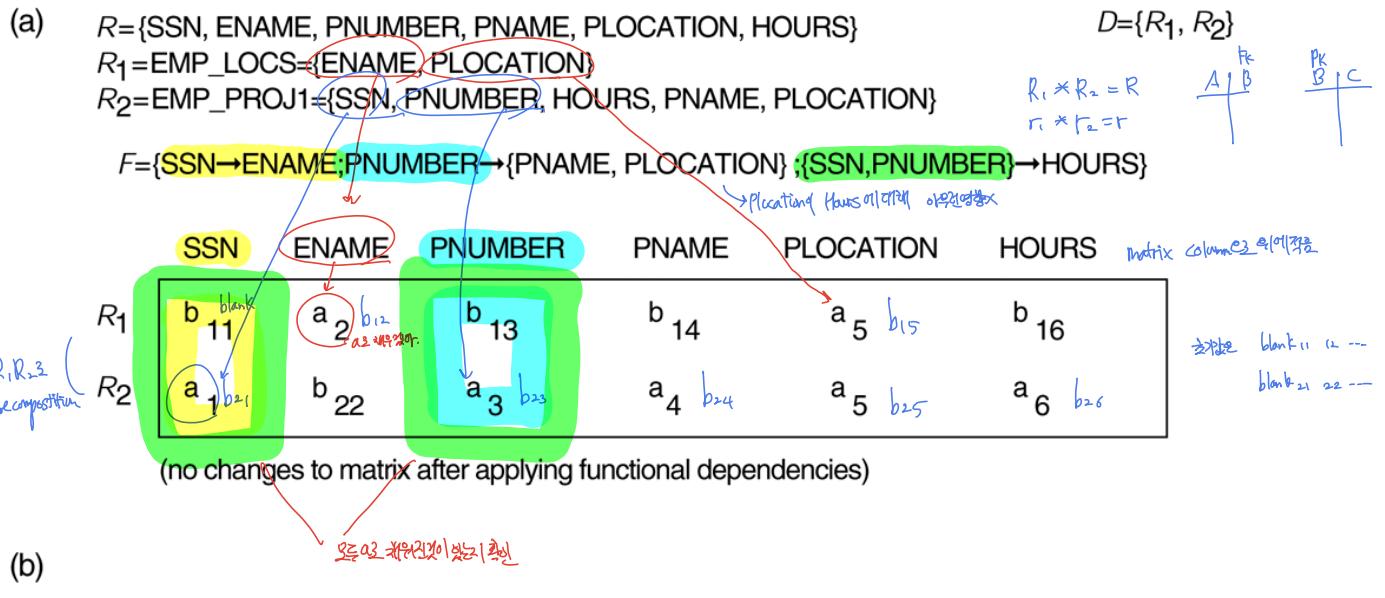
## 무손실 (비부가적) 조인 특성의 검사

- 검사 알고리즘 (다음의 그림을 이용하여 설명함)

(a) EMP\_PROJ를 EMP\_PROJ1과 EMP\_LOCS로 분해한 결과가 무손실조인 특성을 갖는지 검사하기 위하여 알고리를 적용함

(b) EMP\_PROJ의 또 다른 분해집합

(c) 그림 (b)의 분해집합이 무손실 조인 특성을 갖는지 검사하기 위하여 알고리즘을 적용함



(b)

EMP		PROJECT			WORKS_ON		
SSN	ENAME	PNUMBER	PNAME	PLOCATION	SSN	PNUMBER	HOURS

(c)  $R = \{\text{SSN}, \text{ENAME}, \text{PNUMBER}, \text{PNAME}, \text{PLOCATION}, \text{HOURS}\}$   $D = \{R_1, R_2, R_3\}$

$R_1 = \text{EMP} = \{\text{SSN}, \text{ENAME}\}$

$R_2 = \text{PROJ} = \{\text{PNUMBER}, \text{PNAME}, \text{PLOCATION}\}$

$R_3 = \text{WORKS\_ON} = \{\text{SSN}, \text{PNUMBER}, \text{HOURS}\}$

$F = \{\text{SSN} \rightarrow \{\text{ENAME}; \text{PNUMBER} \rightarrow \{\text{PNAME}, \text{PLOCATION}\}; \{\text{SSN}, \text{PNUMBER}\} \rightarrow \text{HOURS}\}$

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
$R_1$	a <sub>1</sub> <sup>PK</sup>	a <sub>2</sub>	b <sub>13</sub>	b <sub>14</sub>	b <sub>15</sub>	b <sub>16</sub>
$R_2$	b <sub>21</sub>	b <sub>22</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>	b <sub>26</sub>
$R_3$	a <sub>1</sub> <sup>FK</sup>	b <sub>32</sub>	a <sub>3</sub>	b <sub>34</sub>	b <sub>35</sub>	a <sub>6</sub>

(original matrix S at start of algorithm)

Stop line  
여기까지는 허용  
다음은 허용X  
종료  
 $R_3$ 은  $R_1$ 의  
운영상태로 되돌아감

	SSN	ENAME	PNUMBER	PNAME	PLOCATION	HOURS
$R_1$	a <sub>1</sub>	a <sub>2</sub>	b <sub>13</sub>	b <sub>14</sub>	b <sub>15</sub>	b <sub>16</sub>
$R_2$	b <sub>21</sub>	b <sub>22</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>	b <sub>26</sub>
$R_3$	a <sub>1</sub>	b <sub>32</sub>	a <sub>3</sub>	b <sub>34</sub>	b <sub>35</sub>	a <sub>6</sub>

(matrix S after applying the first two functional dependencies - last row is all "a" symbols, so we stop)

## 분해와 무손실 (비부가적) 조인

- 릴레이션  $R$ 을 분해한 릴레이션들이 BCNF 정규형에 속하며, 분해집합이 함수적 종속성의 집합  $F$ 에 대해 무손실 조인 특성을 갖게 하는 알고리즘이 존재한다.

Set  $D \leftarrow R$

While there is a relation schema  $Q$  in  $D$  that is not in BCNF do

{

choose one  $Q$  in  $D$  that is not in BCNF

find a FD  $X \rightarrow Y$  in  $Q$  that violates BCNF

replace  $Q$  in  $D$  by two relation schemas  $(Q - Y)$  and  $(X \cup Y)$

}

$$\{A, B, C\} - \{C\}$$

$$Q - (X^+ - X) \quad X^+$$

$$\begin{array}{c} R_{11}(AB) \\ R_{12}(B,D) \\ \text{① ②} \end{array}$$

$$R_{11}(AB) + R_{12}(B,D) = R$$

BCNF를 경유하여 적어도 2개의 허용되는 조인

$$R_1 \rightarrow R_2 \quad R_2 \rightarrow R$$

$$Q \rightarrow R(A, B, C), 0$$

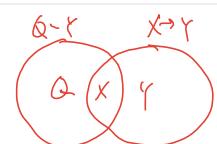
$$\textcircled{1} \quad A \rightarrow B$$

$$\begin{array}{c} \textcircled{2} \\ \textcircled{3} \end{array} \quad \begin{array}{l} B \rightarrow C \\ B \rightarrow D \end{array}$$

$$R_1(A, B)$$

$$R_2(B, C)$$

	A	B	C
$R_1$	$\alpha_1$	$\alpha_2$	$\alpha_3$ copy
$R_2$	$\beta_1$	$\beta_2$	$\beta_3$
	$\alpha_1$	$\alpha_2$	$\alpha_3$



- 위 알고리즘은 다음 두 가지 무손실 조인 분해 특성에 기반한다:

- (1)  $R$ 의 분해집합  $D = \{R_1, R_2\}$ 가 함수적 종속성의 집합  $F$ 에 대해 무손실 조인 특성을 가지기 위한 필요충분조건은 다음의 두 가지 중 반드시 하나를 만족하는 것이다.
  - 함수적 종속성  $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ 가  $F_+$ 에 속한다.
  - 함수적 종속성  $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ 가  $F_+$ 에 속한다.
- (2)  $R$ 의 분해집합  $D = \{R_1, R_2, \dots, R_m\}$ 이 함수적 종속성 집합  $F$ 에 대해 무손실 조인 특성을 가지고,  $R_i$ 의 분해집합  $D_1 = \{Q_1, Q_2, \dots, Q_m\}$ 가  $F$ 의  $R_i$  상에의 프로젝션에 대해 무손실 조인 특성을 가진다면,  $R$ 의 분해집합  $D_2 = \{R_1, R_2, \dots, R_{i-1}, Q_1, Q_2, \dots, Q_k, R_{i+1}, \dots, R_m\}$  또한  $F$ 에 대해 무손실 조인 특성을 가진다.

- 분해집합이 종속성을 보존하며 BCNF 정규형에 속하게 하는 알고리즘은 존재하지 않는다.
- 다음의 수정된 합성 알고리즘은 1) 무손실 조인 특성과 2) 종속성 보존 특성을 만족하며, 3) 제 3 정규형 릴레이션으로 분해하는 것을 보장한다. (주의: BCNF 정규형으로의 분해는 보장 못함)
- 많은 제 3 정규형 릴레이션들은 BCNF 정규형에도 속한다.

## 무손실 조인과 종속성 보존을 보장하는 제 3 정규형 릴레이션으로 분해 Algorithm

1. Find a minimal cover for  $F$ .

2. For each  $X$  of an FD  $X \rightarrow Y$  in  $G$

create a relation schema  $R_i$  in  $D$  with the attributes  $\{X \cup \{A_1\} \cup \{A_2\} \cup \dots \cup \{A_k\}\}$ ,

where  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$  are the only dependencies in  $G$  with  $X$  as left-hand-side

3. If none of the relations schemas in  $D$  contains a key of  $R$ , 구조를 헛하고 있지 않다면

then create one more relation schema that contains

attributes that form a key for  $R$ . key를 헛지 않게

Relational  
synthesis Algo

## 널값과 허상 투플이 야기하는 문제점

- 널값(null values):
  - 널값이 조인 애트리뷰트에 존재할 때 문제가 발생한다.
  - 널 값이 존재하는 경우 질의를 명기할 때 정규 조인(regular join)과 OUTER 조인의 결과 사이의 차이가 중요해진다.
  - 어떤 질의는 정규 조인을 필요로 하고 어떤 질의는 OUTER 조인을 필요로 한다.
  - 널값 조인의 문제점: (a) 조인 애트리뷰트에 널값이 존재하는 데이터베이스

(a)

### EMPLOYEE

Ename	Ssn	Bdate	Address	Dnum
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	NULL
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX	NULL

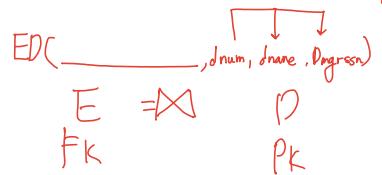
### DEPARTMENT

Dname	Dnum	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

Figure 16.2

Issues with NULL-value joins. (a) Some EMPLOYEE tuples have NULL for the join attribute Dnum. (b) Result of applying NATURAL JOIN to the EMPLOYEE and DEPARTMENT relations. (c) Result of applying LEFT OUTER JOIN to EMPLOYEE and DEPARTMENT.

→ 허상 tuple Null 값이 있는 자연조인 결과에 사라지는 tuple



- 허상 투플(dangling tuples):

- 정규화 과정에서 릴레이션을 너무 분해하면 허상 투플의 문제가 발생함
- 분해된 릴레이션을 조인하면 사라지는 투플을 허상투플이라고 함
- 널값 조인의 문제점:
  - (b) EMPLOYEE와 DEPARTMENT 릴레이션에 자연조인 연산을 적용한 결과
  - (c) EMPLOYEE 릴레이션을 DEPARTMENT 릴레이션과 외부조인한 결과

**(b)**

Ename	Ssn	Bdate	Address	Dnum	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

**(c)**

Ename	Ssn	Bdate	Address	Dnum	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	NULL	NULL	NULL
Benitez, Carlos M.	888665555	1963-01-09	7654 Beech, Houston, TX	NULL	NULL	NULL

# Ch15. 데이터베이스 파일 조직: 레코드들의 비순서, 순서 및 해시된 파일

File

- 디스크 저장 매체
- 레코드들로 구성된 파일
- 파일에 대한 연산
- 비순서 파일(Heap)
- 순서 파일
- 해시 파일
  - 동적이고 확장 가능한 해싱 기술
- RAID 기술

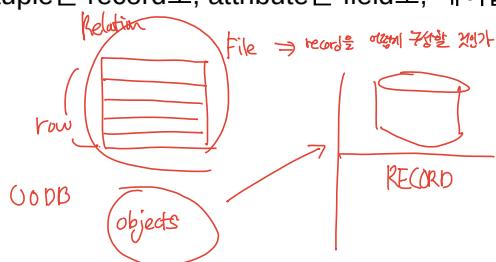
Access  
Method      Indexing

covers ( $f, g$ )

$F$  covers  $G \rightarrow \text{True}$

## 물리적 DB에서 혼동을 일으키는 용어들

- FILE: OS File이 아니다. Disk-based Data structure로 이해해야 한다. 레코드의 집합체
- Data structure라는 용어보다 Data organization이라는 용어를 전통적으로 사용한다.
- Key: 논리적 데이터 모델에서 이야기하는 키가 아니다. search term이면서, 파일에서 탐색을 위해 사용되는 필드이다. key는 주고 value를 가져온다
- tuple은 record로, attribute는 field로, 테이블은 파일로 매핑된다.



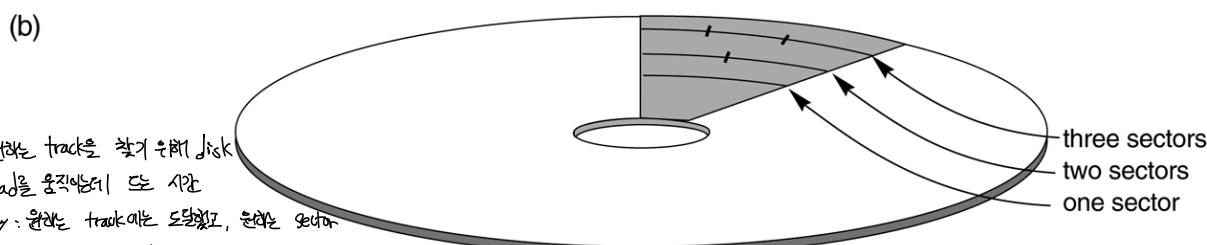
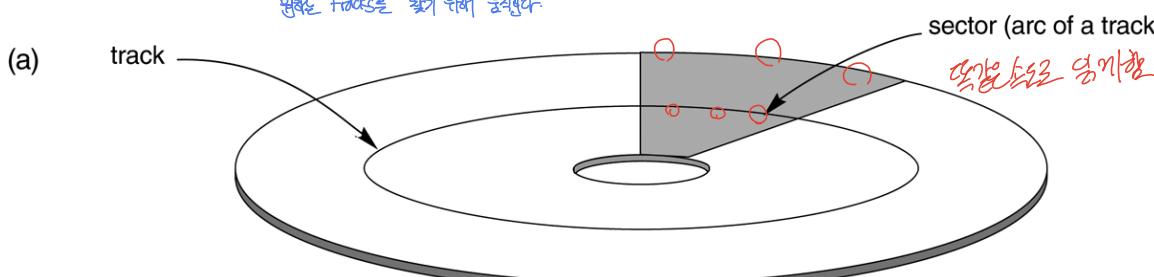
# 디스크 기억 장치

- 자기 디스크는 적은 비용으로 방대한 양의 데이터를 저장하기 위하여 사용된다.
- 디스크에 저장하는 가장 기본적인 데이터의 단위는 비트이며, 특정 방법으로 디스크상의 한 영역을 자기화함으로써 0 또는 1의 비트값을 표현
- 디스크 팩은 여러 장의 자기 디스크를 뮤어서 구성
- 디스크는 디스크 표면상의 동심원인 트랙으로 나누어지며 각 트랙은 4 ~ 50 Kbytes를 기록할 수 있음.
- 트랙은 블록으로 나누어지며 블록 크기는 한 시스템 내에서 고정되어 있으며 일반적인 블록 크기는 512byte에서 4096byte이다. 주기억장치와 디스크간의 데이터 이동은 블록 단위로 수행
- 디스크 상의 여러 섹터 구조 :
  - (case a) 일정각으로 분할된 섹터,
  - (case b) 동일한 기록 밀도를 유지하는 섹터

Block (or page) : sectors의 그룹 (read/write의 Unit)  
일반적으로 4K bytes

- Disk는 magnetic platters의 stack
- 각 platter의 표면은 여러개 tracks로 구성
- tracks는 sectors로 나눠진다.
- Sector: Storage의 기본 Unit

disk heads는 spindle의 원통을 찾기 위해 회전하는 동안 (동일한 cylinder내)  
원하는 tracks를 찾기 위해 움직인다.



- 판독/기록 헤드가 전송할 블록이 있는 트랙으로 이동하고 자기 디스크면이 회전하여 원하는 블록이 판독/기록 헤드 아래에 위치하면 그 블록을 읽거나 쓴다. ↗
- 디스크 접근 시간 = 탐색시간 (seek time) + 회전 지연시간 (또는 지연시간 rotational delay) + 블록 전송 시간(block transfer time)  
↑↑↑ 시간의 합계임. ↗ 한바퀴 돌아와서 일정시간 Ram Buffer로 보내는시간
- 물리적 디스크 블록 주소는 표면번호, 표면내 트랙번호, 트랙내 블록번호로 구성
- 디스크 블록의 읽기나 쓰기에서 탐색시간과 회전 지연 시간이 대부분의 시간을 차지한다.

전송시간 Commit发生在 read 완료되었으면  
Reto

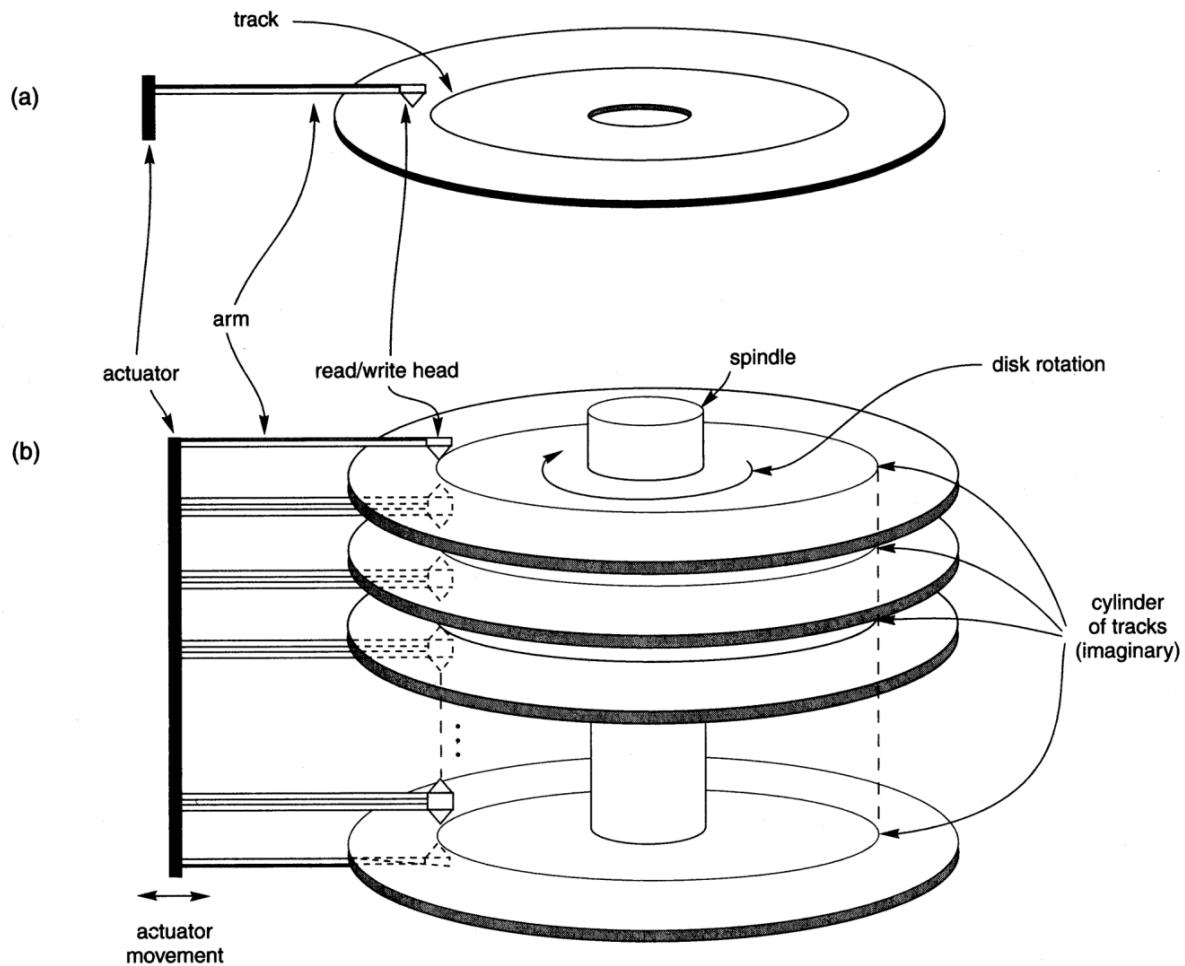
## Disk read

- 처리할 disk block의 LBA가 disk drive buffer(cache)에 복사
- cache에 이미 해당 LBA가 있다면 copy X

## Disk write

- buffer (main memory buffer)의 contents가 LBA와 함께 disk block로 복사됨
- Disk read와 write는 Synchronous 하여 성능이 같지만, flash의 경우는 두 경로의 성능이 차이가 있다.

(a) 판독/기록 하드웨어를 장착한 단일면 디스크 (b) 판독/기록 하드웨어를 장착한 디스크



## Making Data Access More Efficient on Disk - 7th Edition in English

We list some of the commonly used techniques to make accessing data more efficient on HDDs.

1. Buffering of data
2. Proper organization of data on disk:
  - keep related data on contiguous blocks
  - Doing so avoids unnecessary movement of the read/write arm and related seek times
3. Reading data ahead of request
4. Proper scheduling of I/O requests:
  - **elevator algorithm**
5. Use of log disks:
  - A single disk may be assigned to just one function called logging of writes.
  - All blocks to be written can go to that disk sequentially, thus eliminating any seek time.
6. Use of SSDs or flash memory for recovery purposes

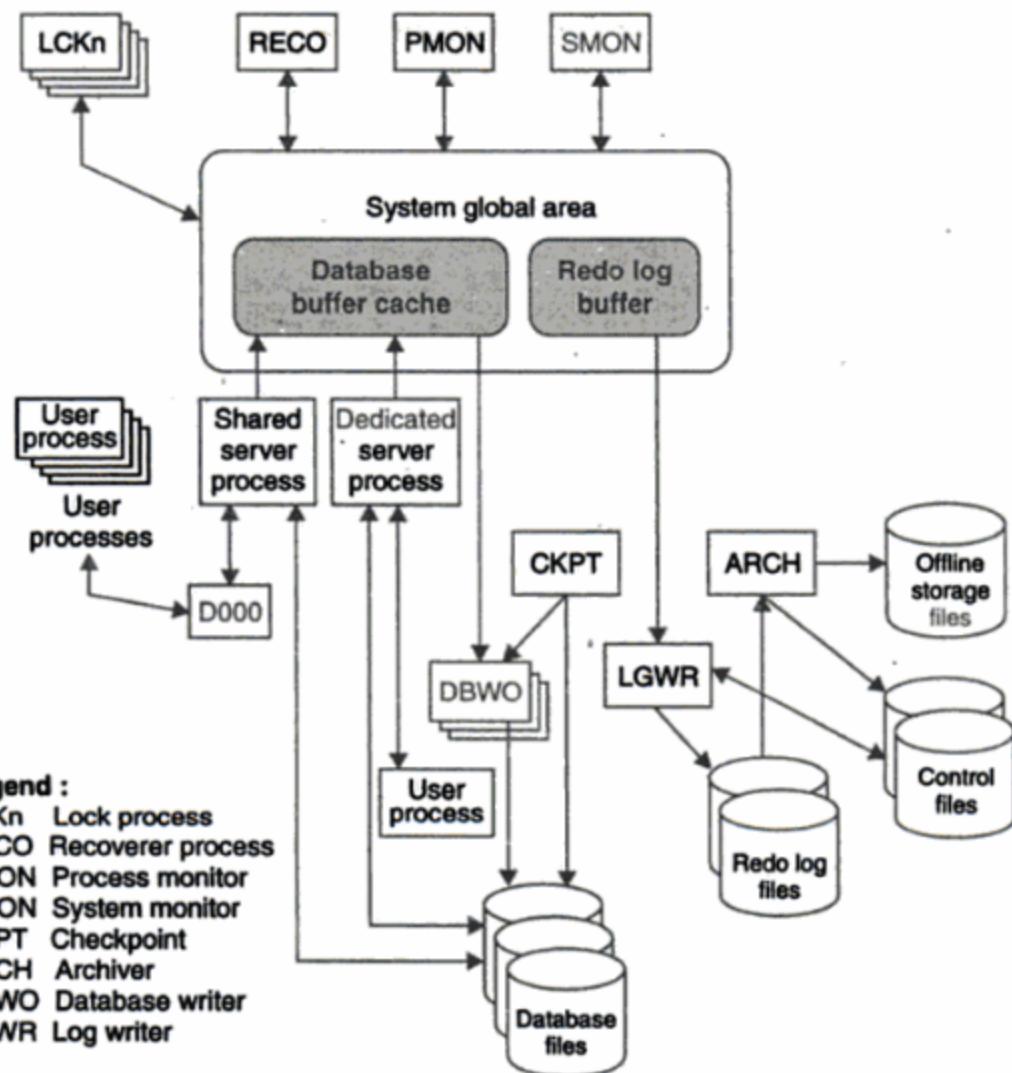
## Oracle DBA Guide

[https://docs.oracle.com/cd/B28359\\_01/server.111/b28310/toc.htm](https://docs.oracle.com/cd/B28359_01/server.111/b28310/toc.htm)  
([https://docs.oracle.com/cd/B28359\\_01/server.111/b28310/toc.htm](https://docs.oracle.com/cd/B28359_01/server.111/b28310/toc.htm)).

## Oracle Performance Tuning Guide

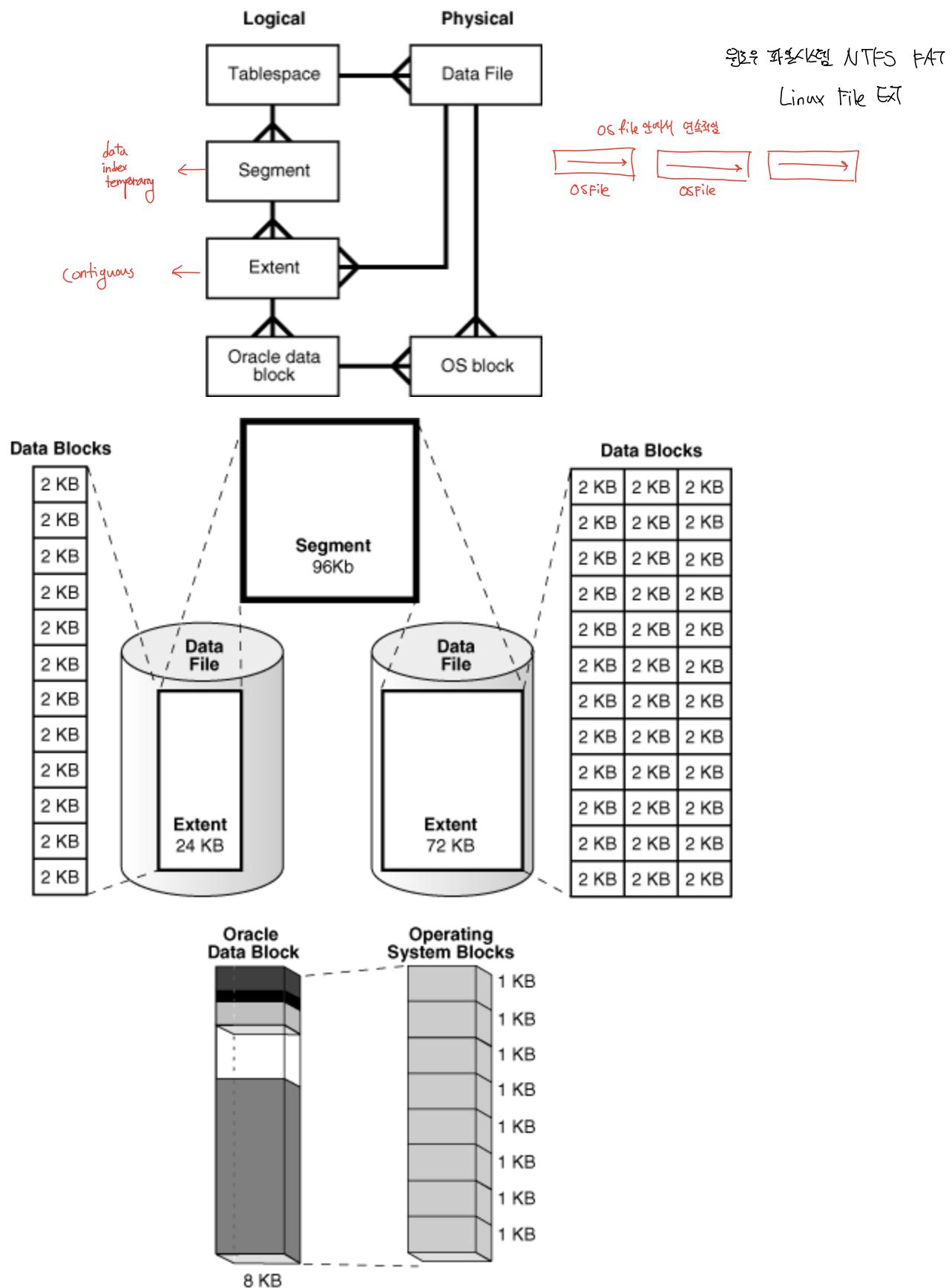
[https://docs.oracle.com/cd/B28359\\_01/server.111/b28274/toc.htm](https://docs.oracle.com/cd/B28359_01/server.111/b28274/toc.htm)  
([https://docs.oracle.com/cd/B28359\\_01/server.111/b28274/toc.htm](https://docs.oracle.com/cd/B28359_01/server.111/b28274/toc.htm)).

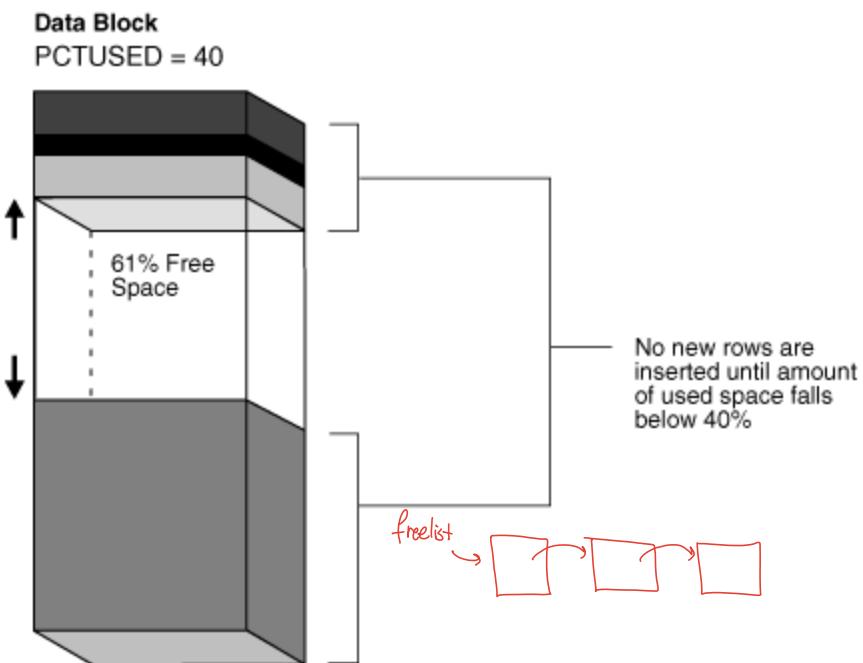
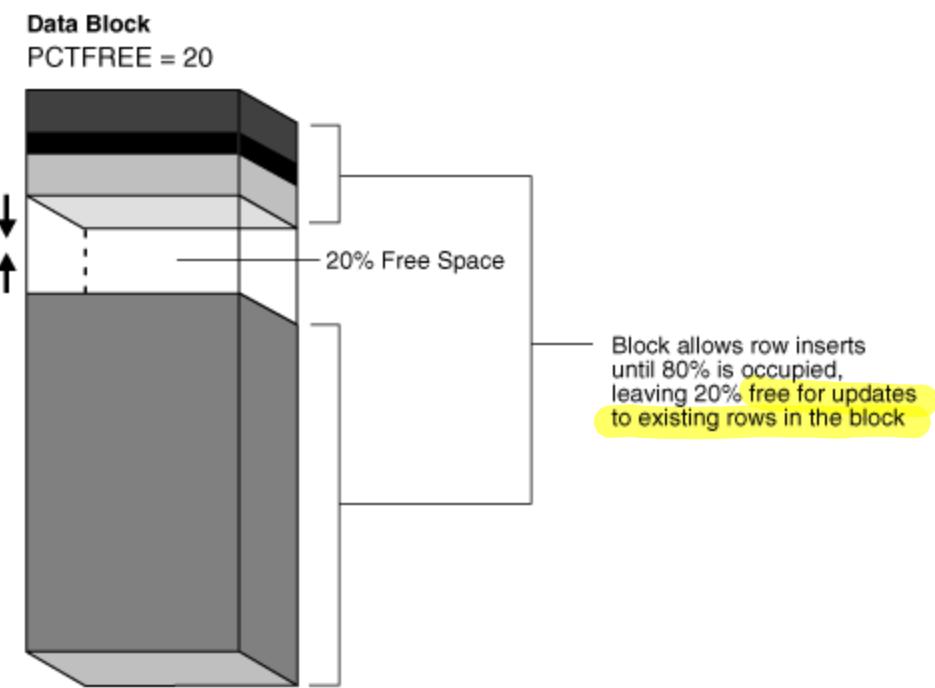
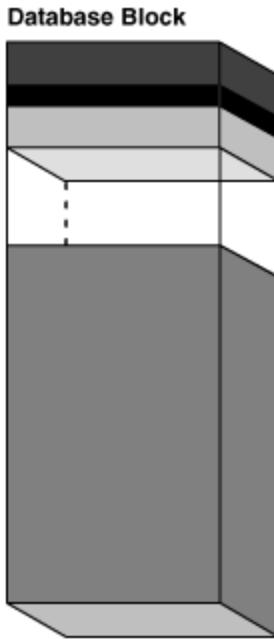
## Oracle Memory, Disk and Processes



- SMON is responsible for system recovery at instance start-up
- PMON is responsible for user transaction recovery, reclaim cache, and other resources when user transaction fails
- RECO is responsible for distributed transactional recovery
- A dedicated server process, which services only one user process
- A shared server process, which can service multiple user processes

# Oracle Storage Structure





```

Create Tablespace my_tbl_space
Datafile
' /data/Bigtbsa01.dbf '
Size 100M
AUTOEXTEND On
Next 10M
MAXSIZE 200M,

' /data/Bigtbsa02.dbf '
Size 100M
AUTOEXTEND On
Next 10M
Maxsize 200M,

' /data/Bigtbsa03.dbf '
Size 100M
AUTOEXTEND On
Next 10M
MAXSIZE 200M,

' /data/Bigtbsa04.dbf '
Size 100M
AUTOEXTEND On
Next 10M
Maxsize 200M
Blocksize 16K;

```

## Record and Record Type

- 데이터는 일반적으로 레코드 형태로 저장되며, 각 레코드는 연관된 데이터 값이나 항목들로 구성된다.
- 각 값은 하나 이상의 바이트로 구성되며 레코드의 특정 필드에 해당된다.
- 레코드는 엔티티와 엔티티의 애트리뷰트들을 기술한다.
- 한 필드의 데이터 타입은 프로그래밍에서 사용되는 표준 데이터 타입들 중의 하나이다.
- 표준 데이터 타입에는 수치, 문자열, 부울(0과 1 또는 TRUE와 FALSE) 데이터 타입이 있고, 때로는 특별하게 코드화된 날짜, 시간 데이터 타입들도 사용된다.
- 이미지, 비디오, 오디오, 긴 텍스트를 나타내는 방대한 양의 비구조적인 객체들은 BLOB(Binary Large Object)라고 하며, 대개 별도의 디스크 블럭들의 풀(pool)에 저장되고 레코드에는 그 포인터만 저장한다.

*enum { 'Spring', 'Summer', 'Autumn', 'Winter'*  
              6        1        2        3

*Epoch      시즌의 행정집*

## Blocking

- 디스크와 주기억 장치 사이의 데이터 전송 단위는 하나의 블록이므로 한 파일의 레코드들을 디스크 블록들에 할당하여야 한다.
- 블록 크기가 레코드 크기보다 크다면 각 블록에 여러 개의 레코드를 저장한다.
- 블록크기가  $B$ 바이트이라 하고 크기가  $R$ 바이트인 고정 길이 레코드들로 구성된 파일에 대하여  $B \geq R$ 인 경우에 블록당  $bfr = \lfloor B/R \rfloor$  개의 레코드를 저장할 수 있다.  $B$ : block size  $R$ : record size  $bfr$ : blocking factor for the file
- 일반적으로  $B$ 를  $R$ 로 정확히 나눌 수 없으므로 각 블록마다  $B - (bfr \times R)$ 바이트만큼의 사용되지 않는 공간이 있다.
- 비사용 공간을 사용하기 위하여 레코드의 일부분을 한 블록에 저장하고 나머지 부분을 다른 블록에 저장할 수 있다.

$bfr$  many records per block을 사용할 수 있다.

$$b = \lceil r/bfr \rceil \text{ blocks } (r: \text{file} \text{ read } \ell)$$

Q How many blocks are needed to store a file of 100 reads?

Record size : 16 bytes

Block size : 4KB bytes

Unused space : 0 bytes

$bfr$ ?

$$0 = 4096 - (bfr * 16)$$

$$bfr = 4096 / 16 = 256$$

# Record and File

- 파일은 데이터 값이나 항목들로 구성된 레코드의 집합
- 파일 내의 레코드들이 모두 같을 때 이 파일이 고정 길이 레코드들로 구성되어 있다고 말한다.
- 파일 내의 레코드들의 크기가 서로 다를 때 이 파일은 가변 길이 레코드들로 구성되어 있다고 말한다.
- 가변 길이 레코드 파일에서는 가변 길이 필드의 바이트 수를 결정하기 위해 분리 문자를 사용하거나, 필드값 앞에 가변 길이 필드의 바이트 길이를 기록한다.
- 레코드를 하나 이상의 블록에 걸쳐서 저장하는 방식을 신장(spanned)조직이라 한다
- 레코드 하나의 크기가 한 블록의 크기보다 클 경우 신장조직을 사용하여야 한다.
- 레코드가 블록 경계를 넘지 않도록 저장하는 방식을 비신장(unspanned) 조직이라 한다.
- 가변길이 레코드의 경우에는 신장 또는 비신장 조직을 사용할 수 있다.
- 평균 레코드 크기가 클 경우에는 각 블록내의 손실 공간을 줄이기 위하여 신장조직을 사용하는 것이 유리하다.

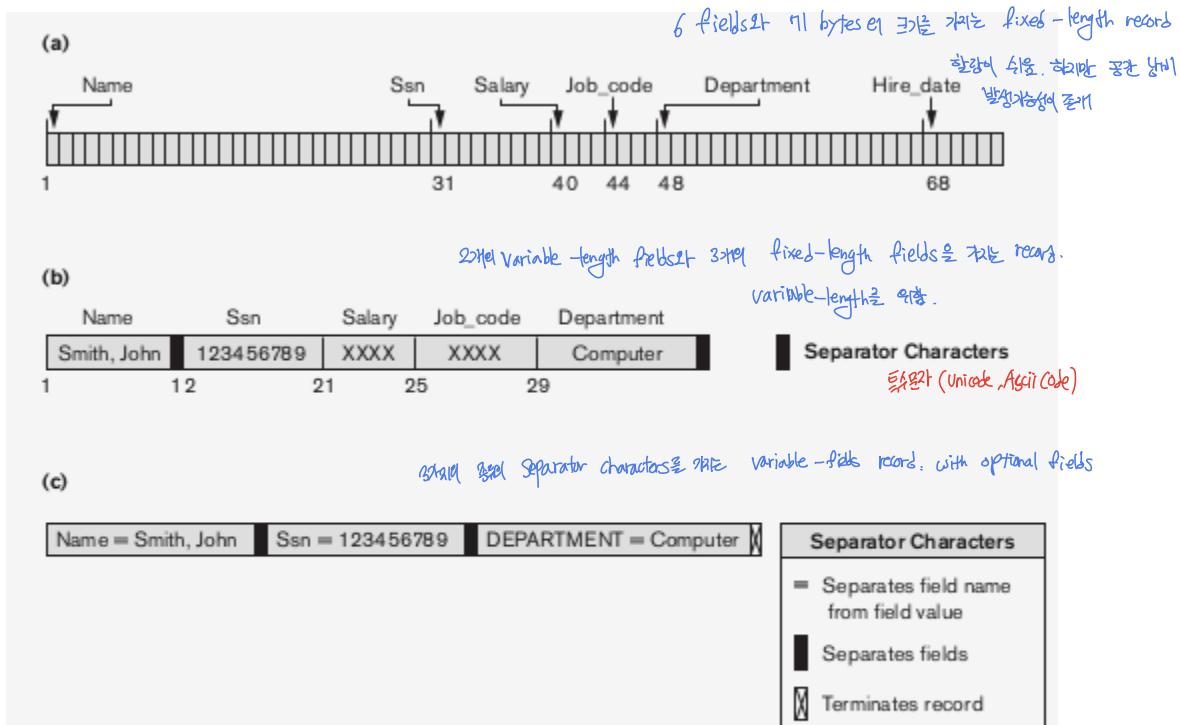


Figure 16.5

Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

record의 크기가 block보다 작은 경우

pointer를 사용하지 않아 빠르지만, 공간이 낭비된다.

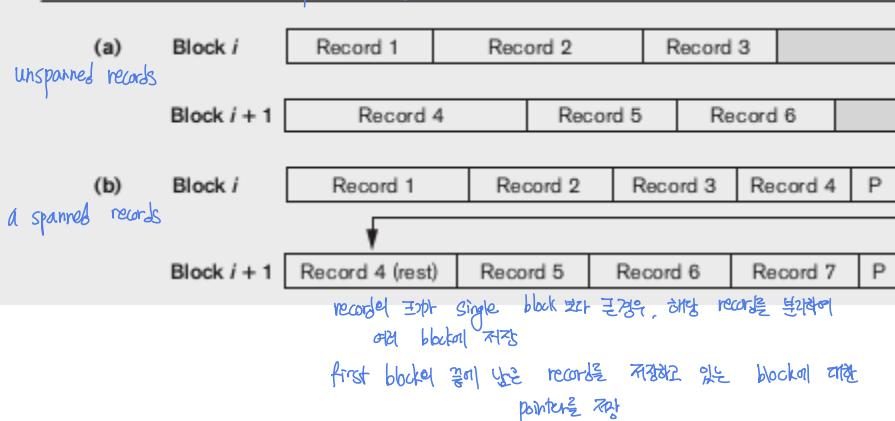


Figure 16.6

Types of record organization.

(a) Unspanned.

(b) Spanned.

file의 records는 무조건 disk blocks에 할당되어야 한다.

• block은 disk와 memory 사이의 data transfer의 기본위

• block의 크기가 record의 크기와 크거나 같다면, 한 block에 여러 개의 records가 저장될 수 있다

# 화일에 대한 연산

65/8 처음보는거

- 한 번에 한 레코드 연산
  - Open : 판독과 기록을 위해 화일을 개방하고, 디스크로부터 파일 블록들을 유지하기 위해 적절한 버퍼들을 할당한 후 파일 헤더를 검색한다. 파일 포인터를 파일의 시작 위치로 설정한다.
  - Reset : open된 파일의 파일 포인터를 파일의 시작 위치로 설정한다.
  - Find : 조건을 만족하는 첫 번째 레코드를 탐색하고, 탐색된 레코드를 현재 레코드로 지정한다.
  - FINDNEXT : 현재 레코드로부터 조건을 만족하는 다음 레코드를 탐색하고, 탐색된 레코드를 현재 레코드로 지정한다.
  - READ : 현재 레코드를 프로그램 변수로 복사한다.
  - Delete : 현재 레코드를 삭제하고, 파일을 갱신한다.
  - Modify : 현재 레코드의 필드값을 수정하고, 파일을 갱신한다.
  - Insert : 새로운 레코드를 파일에 삽입한다.
  - Close : 버퍼 해제와 기타 필요한 제거 연산을 수행하여 파일 접근을 완료한다.
- 한 번에 레코드들의 집합 연산 : 데이터베이스 시스템에서 사용하는 고수준의 연산
  - FindAll : 탐색 조건을 만족하는 파일 내의 모든 레코드를 찾는다.
  - Find n : 검색 조건을 만족하는 첫 번째 레코드를 검색한 후 동일한 조건을 만족하는 나머지 n-1개의 레코드를 연속해서 찾는다.
  - FindOrdered : 특정 순서대로 파일 내의 모든 레코드를 검색한다.
  - Reorganize : 파일을 재조직한다.
- **화일 조직(File Organization)**은 기억 장치 매체상에 레코드와 블록들을 저장하고 서로 연결시키는 방법.
- **접근 방법(Access Method)**은 파일에 적용할 수 있는 연산들의 그룹을 제공 - scan or indexed access
- 파일에 자주 사용되는 연산들을 가능한 한 효율적으로 수행해야 성능이 우수한 파일 조직이라 하겠다.

## HEAP FILE

file의 어느곳이든 record를 저장 가능

- 비순서(unordered) 파일. pile이라 불리기도 함.
- 새로운 레코드는 파일의 마지막에 삽입하므로 **삽입은 매우 효율적이다**.
- 레코드를 탐색하기 위해서는 선형 탐색 기법을 사용해야 한다. 이 기법은 **평균 파일 블록의 반**을 탐색하기 때문에 비효율적이다.
- 어떤 필드값의 순서대로 레코드를 판독하기 위해서는 해당 파일의 레코드들을 정렬시켜야 한다.
- 비저장 블록과 연속할당 방식을 사용하는 비순서 고정길이 레코드들로 구성된 파일에서는 레코드의 위치(순번)을 이용하여 레코드를 접근할 수 있다. 이런 방식으로 접근할 수 있는 파일을 상대 파일이라고 한다.

un internal sorting

외부 External "

## Oracle create table syntax

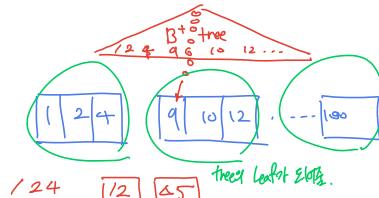
```
CREATE TABLE [schema.]tablename
  ((columnname datatype [DEFAULT (default_constant|NULL)] [col_constr {col_constr. . .}])
   | table_constr) -- choice of either columnname-definition or table_constr
  [, (columnname (repeat DEFAULT clause and col_constr list) | table_constr) . . .])
  [ORGANIZATION HEAP | ORGANIZATION INDEX (this has clauses not covered)]
  [TABLESPACE tblspacename]
  [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS {n|UNLIMITED}]
            [PCTINCREASE n]) (additional STORAGE options not covered)]
  [PCTFREE n] [PCTUSED n]
  [disk storage and other clauses (not covered, or deferred)]
  [AS subquery]
```

**ORACLE** Relational Create Table Statement Syntax with Partial Indexing Syntax

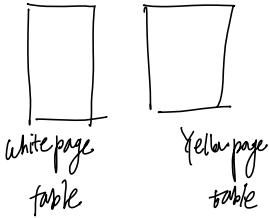
## Ordered File

각 record Search key를 값에 따라 순차적으로 records를 저장

- 순서 파일. 순차화일 (sequential file)로 불리우기도 한다.
- 파일 내의 레코드들을 순서 필드의 값에 따라 정렬된다.
- 레코드들이 정렬된 형태로 추가되어야 하기 때문에 삽입 비용이 비싸다. 더욱 효율적으로 작업하기 위해 오버플로 파일 또는 트랜잭션 파일을 사용한다. 단, 이 방식은 정기적으로 주 파일에 합병시켜야 한다.
- 이진 탐색은 레코드의 순서 필드값에 따른 탐색의 경우 사용된다.
- 평균적으로  $\log_2 b$ 개의 블록을 접근하는 이진 탐색은 선형 탐색보다 더 좋은 성능을 보인다.(b는 블록의 총 개수).
- 정렬된 키 값의 순서대로 판독하는 연산이 매우 효율적이다. (why?)
- (dynamic file) 레코드 삽입과 삭제는 레코드를 물리적으로 정돈된 상태를 유지해야 하므로 비용이 매우 크다. (static file인 경우, no problem. ISAM)
- 순서 파일 방식에서 비순서 필드 값을 기반으로 레코드들을 임의 접근하거나 순서 접근하기 어렵다. 이 경우 임의 접근 (random access)은 선형 탐색을 사용하고, 순서 접근(sequential access)은 파일을 정렬한 복사본을 사용해야 한다.
- 삽입을 효율적으로 수행하기 위해 오버플로 파일 또는 트랜잭션 파일이라는 임시 비순서 파일에 기록한 후 주 파일 또는 마스터 파일과 정기적으로 합병
- 레코드 삭제시 빈 공간을 메우기 위해 레코드들을 이동하거나 삭제 표시자를 사용할 수 있다. 생각해보면 맹거야짐. 시간이 되면 merge할 때 삭제 표시자를 찾면서 merge
- 순서 필드 값을 수정하는 연산은 그 레코드를 삭제하고 새로운 순서 필드 값을 갖는 레코드를 삽입한다.
- 기본 인덱스라고 부르는 부가적인 접근 경로와 함께 사용되는 것이 일반적이다. (Oracle primary index)



white page and Yellow page example  
only one ordered file



정렬되며 잊을 경우. 블록의 개수만큼 끌어져야함  
정렬되지 않거나 놓을 경우. 레코드 개수만큼 끌어져야함



## NAME 필드를 순서키로 갖는 EMPLOYEE 레코드들로 구성된 순서 화일의 일부 목록

정렬되지 않은 경우

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
		⋮				
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
		⋮				
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
		⋮				
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
		⋮				
	Anderson, Rob					
Block 5	Anderson, Zach					
	Angeli, Joe					
		⋮				
	Archer, Sue					
Block 6	Arnold, Mack					
	Arnold, Steven					
		⋮				
	Atkins, Timothy					
	⋮					
Block $n-1$	Wong, James					
	Wood, Donald					
		⋮				
	Woods, Manny					
Block $n$	Wright, Pam					
	Wyatt, Charles					
		⋮				
	Zimmer, Byron					

**Figure 16.7**

Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

## 기본 파일 구조에 대한 평균 접근 시간

TABLE 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS

TYPE OF ORGANIZATION	ACCESS/SEARCH METHOD	AVERAGE TIME TO ACCESS A SPECIFIC RECORD
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$ $\frac{10^6}{10} = 10^6$

$$2^3 = 2^{\infty} \quad \log_2 10^6 = \log_2 2^{\infty} \cdot 2^{10} = 20$$

20번의 접근 횟수

OLT에서 10만개

OLT에서 10만개

File organization + Access Method

Hash h(key)



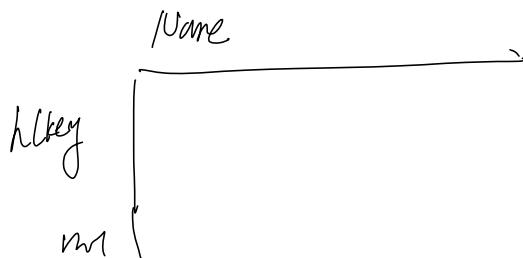
# Hashing

hash function을 사용하여 해시 값에 따라 record 위치

$h(k) \rightarrow i$

randomize

- 해싱을 기반으로 조직된 파일을 해시 파일 또는 직접 파일(direct file)이라고 한다.
- 해시 함수 또는 난수화 함수라고 하는 함수  $h$ 를 레코드의 해시 필드값에 적용하여 레코드를 저장하고 있는 디스크 블록의 주소를 산출한다.
- 대부분의 레코드에 대해 한 번의 블록 접근으로 원하는 레코드를 검색하는 매우 효율적인 방식이다.
- 파일의 키 필드로 해시 필드를 사용할 경우 이를 해시키라고 한다.
- 프로그램에서 어떤 필드값을 사용하여 레코드들의 그룹을 배타적으로 접근하고자 할 때 내부 탐색 구조로 해싱을 사용한다.
- 내부 파일에서 해싱은 일반적으로 레코드들의 배열을 이용하여 해시 테이블로 구현한다.
- 해시 필드 공간이 주소 공간보다 매우 크기 때문에 대부분의 해시 함수는 서로 다른 해시 필드값을 항상 서로 다른 주소로 사상하지는 못하는 문제점이 있다.  $\Rightarrow$  Collision
- 삽입하려는 새로운 레코드의 해시 필드 값이 다른 레코드가 이미 점유하고 있는 주소로 해싱될 때 충돌(collision)이 발생. 이 경우 새로운 레코드를 삽입할 다른 위치를 찾아야하는데 이를 충돌 해결이라 함.
- 해시 테이블을 70%에서 90% 정도 사용하도록 유지할 때 충돌 횟수도 줄이고, 공간 낭비도 줄일 수 있다.
- 충돌(Collision) 해결 기법
  - 개방 주소 지정(Open addressing): 해시 주소가 가리키고 있는 위치부터 시작하여 빈 위치를 찾을 때까지 순차적으로 그 이후의 위치들을 조사한다.
  - 체인(Chaining): 오버플로 영역에 새로운 레코드를 저장하고 다른 레코드가 이미 점유하고 있는 해시 주소 영역의 포인터에 오버플로 영역의 주소를 지정하여 충돌을 해결한다.
  - 다중 해싱(Multiple hashing): 충돌이 발생하지 않을 때까지 여러 개의 해시 함수를 적용한다. 충돌 시 필요하면 개방 주소 지정 방법을 사용한다.



(a)

Name Ssn Job Salary

Figure 16.8

## Oracle Hash Cluster Syntax

```
CREATE CLUSTER [schema.]clusternname
  (columnname datatype [, columnname datatype ...] -- this is the cluster key
   [cluster_clause { cluster_clause ...}]);
```

The cluster\_clauses that specify cluster characteristics are chosen from the following:

```
[PCTFREE n] [PCTUSED n]
[STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS n]
          [PCTINCREASE n])]

[SIZE n [K|M]]                                     -- defaults to one disk block
[TABLESPACE tblspacename]
[INDEX | [SINGLE TABLE] HASHKEYS n [HASH is expr]]
[other clauses not covered];
```

이것은 디스크 파일에 대한  
슬롯의 개수 또는 크기를 정하는  
문장입니다.

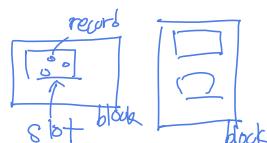
To delete an existing cluster, the DROP CLUSTER statement is used:

```
DROP CLUSTER [schema.]clustername [INCLUDING TABLES [CASCADE CONSTRAINTS]];
```

### ORACLE Create Cluster Statement Syntax

```
CREATE TABLE [schema.]tablename
  (column definitions as Basic SQL, see Figure 7.1 or Figure 8.5)
  CLUSTER clusternname (columnname [, columnname...]) -- table columns map to cluster key
  [AS subquery];
```

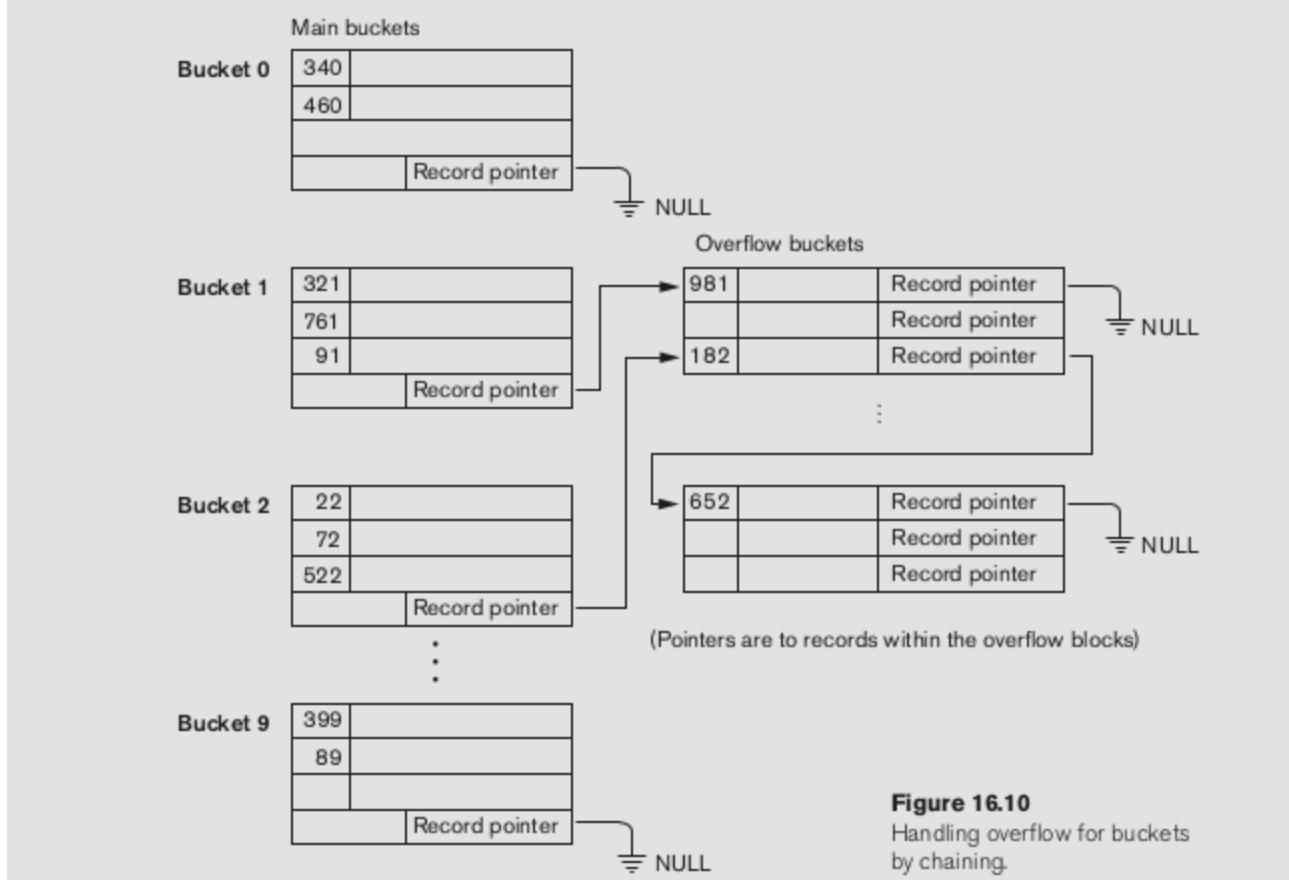
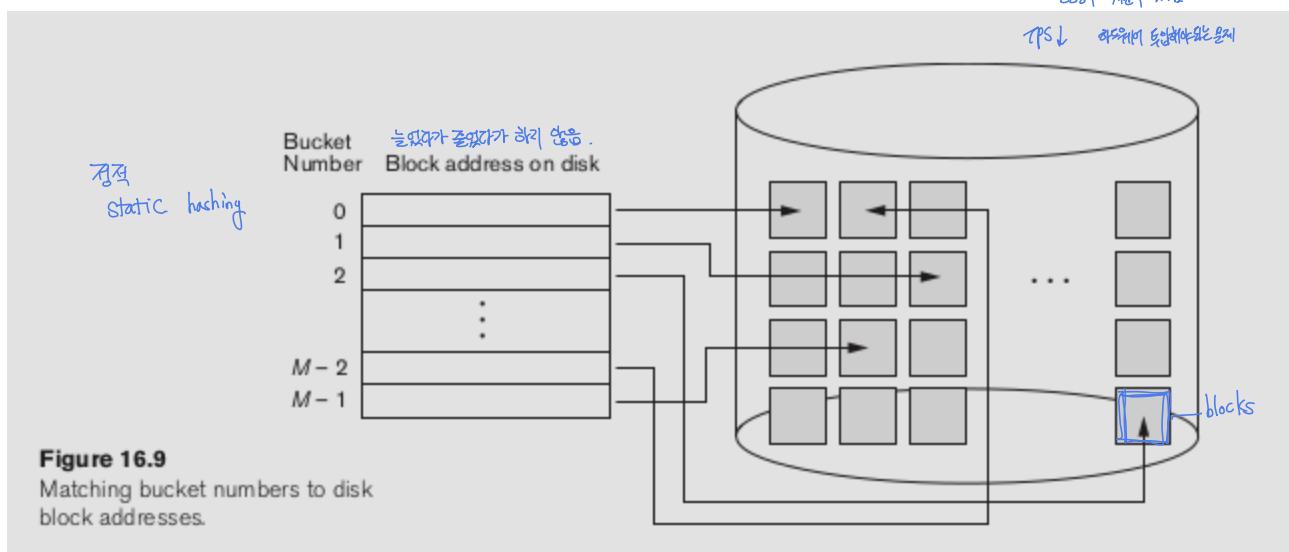
### Create Table Statement for Table to Be Contained in a Cluster



# Hashing File

- 목표 주소 공간을 같은 크기의 버켓들로 구성하고, 각 버켓에는 다수의 레코드들을 유지함.
- 버켓은 디스크 블록 한 개나 연속적인 블록인 클러스터 한 개의 크기를 갖는다.
- 해시 키가  $K$  인 레코드는 해시 함수  $i = h(K)$ 에 의해 결정된 버켓  $i$ 에 저장
- 가득 찬 버켓으로 새로운 레코드가 해시 될 경우 충돌이 발생되며 충돌된 레코드는 오버플로우(Overflow) 버켓에 저장, 각 버켓에서 오버플로우된 레코드들은 연결 리스트로 구성
- 단점
  - 버켓의 수가 고정되어 있기 때문에 파일의 레코드수가 버켓 공간에 비해 너무 적거나(공간 낭비 초래) 너무 많으면 (빈번한 충돌 발생) 문제가 발생한다.
  - 해시 키에 따른 순차적 접근은 매우 비효율적이며 레코드의 정렬이 요구된다.

디스크 I/O가  
extra 밸류, 충돌이  
일어나므로  
TPS ↓ 해도 유리한 방법이 있는지



## Types of Query

### 1. Point Query

```
SELECT balance  
FROM accounts  
WHERE number = 1023;
```

### 2. Multipoint Query

```
SELECT balance  
FROM accounts  
WHERE branchnum = 100;
```

### 3. Range Query

```
SELECT number  
FROM accounts  
WHERE balance > 10000;
```

### 4. Join Query

```
SELECT *  
FROM accounts, branch  
WHERE a.branchnum = b.branchnum;
```

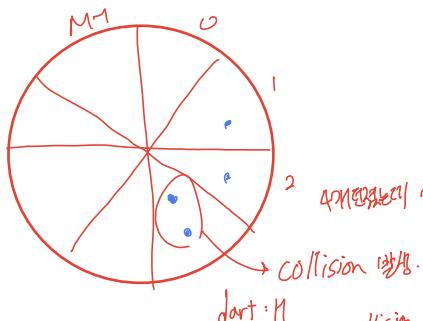
equijoin accounts  $\bowtie$  branch

Hashing File은 어떤 질의에 효과적일까? Why?

Mysql의 selfjoin은 Hash join  
hash join 한정

## Throwing Darts at Random slots

- number of darts : N
- number of random slots : M
- How many slots on the board have a dart in them?



$$\text{collision rate} = \frac{N-1}{N}$$

충돌은 예상한 상태로 충돌

예상과 차이점은 충돌을 빼면 → 실제 충돌수

365-24\*365 = 329 static hashing 개수  
충돌

$(1 - \frac{1}{M})^N$  N개를 충돌을 예상한 확률

충돌 확률은 M-1개의 충돌을 예상한 확률

$$M \cdot (1 - \frac{1}{M})^N : \text{기대값}$$

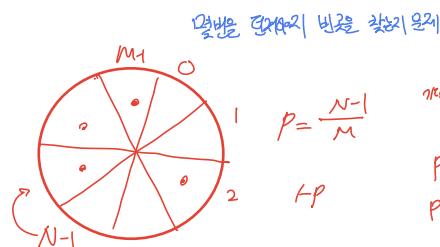
$$M - M \cdot (1 - \frac{1}{M})^N : M-1개의 충돌$$

$$= M \left( 1 - \left( 1 - \frac{1}{M} \right)^N \right) = \frac{N-M(1-e^{-1})}{N} = 1 - P^1(1-e^{-1})$$

$M \rightarrow \infty \Rightarrow e^{-1}$  1-21% 확률

## Slot Occupancy of ONE : number of Retries (Rehash chain)

- One row in any single slot
- Rehash the key value to a sequence of slots until an empty slot is located
- Dart version:
  - N darts in M slots
  - Slot occupancy 1
  - Retries until all darts in board
  - What is the expected number of retries?



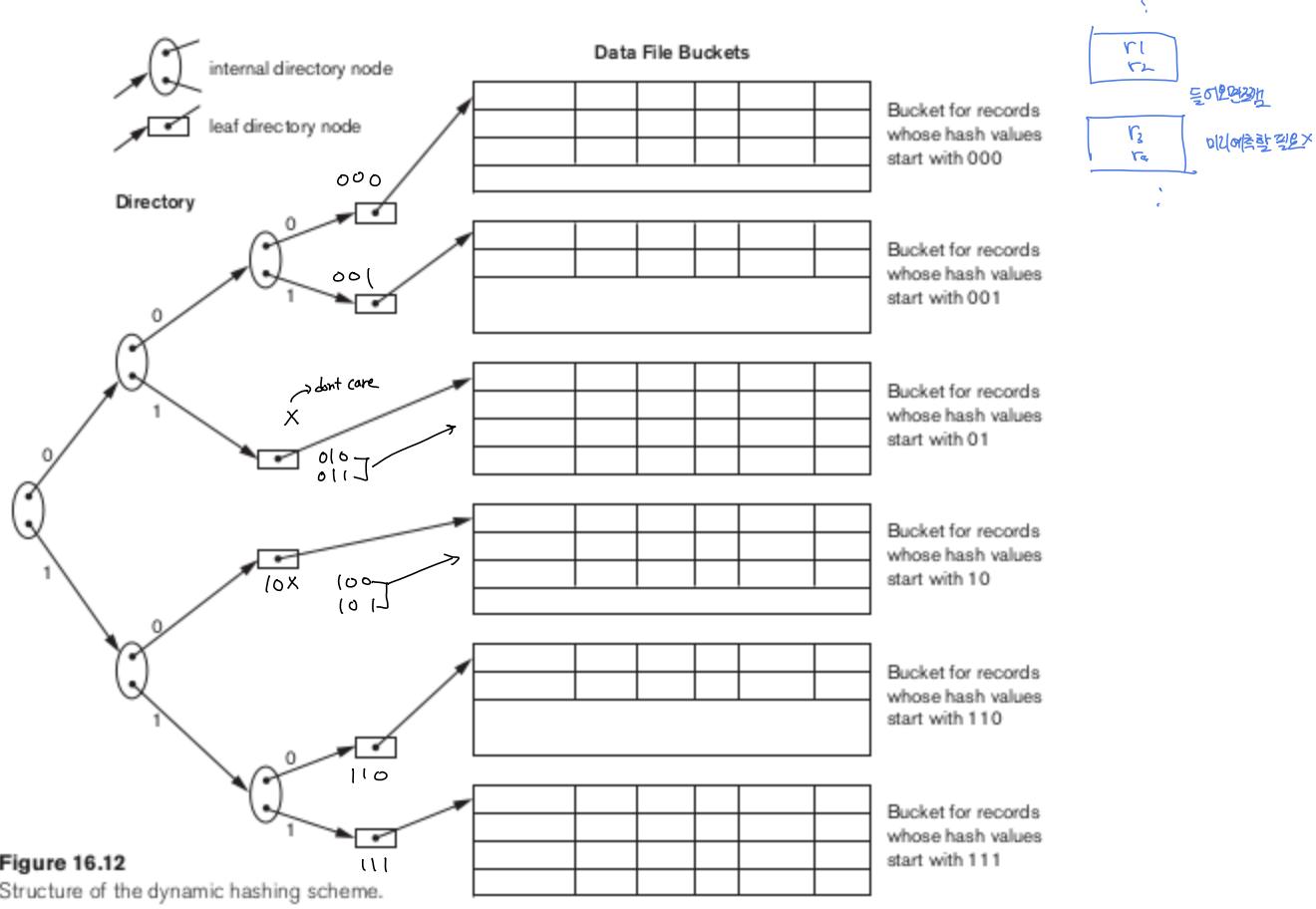
$$\begin{aligned}
 f(p) &= 1+2p+3p^2+\dots \\
 -pf(p) &= p+2p^2+\dots \\
 (1-p)f(p) &= 1+p+p^2+\dots \\
 = \frac{f(p)}{1-p} &= \frac{1}{1-p} = \boxed{\frac{1}{1-p}} \\
 P(1) &= 1-p \\
 \text{Expected length} &= 1 \cdot P(1) + 2 \cdot P(2) \\
 P(2) &= p \cdot (1-p) \\
 &\quad + 3 \cdot P(3) + \dots \\
 P(3) &= p \cdot p \cdot (1-p) \\
 &\quad = (1-p)(1+2p+3p^2+\dots) \\
 \vdots \\
 P(n) &= p^{n-1} \cdot (1-p)
 \end{aligned}$$

## 동적(dynamic) 및 확장가능(extendable) 해싱 화일

- 정적 해싱은 해시 주소 공간이 고정되어 화일을 동적으로 확장하고 축소하는 것이 어렵다. 이 문제를 해결하기 위한 기법으로는 확장가능 해싱, 선형 해싱이 있다.
- 이런 해싱 기법들은 해시 함수 결과 (아래에서 K로 표시함)를 이진수로 표현할 수 있고, 이진수 표현을 기반으로 접근 구조를 구성한다. 이런 이진수 표현을 레코드에 대한 해시값이라 한다. 이진수 표현은 비트열로 구성된다. 레코드 해시값에서 선행하는 비트들의 값을 기반으로 레코드들을 버켓들에 분배한다.

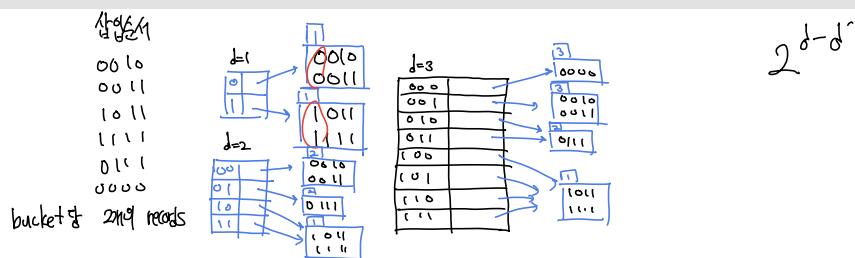
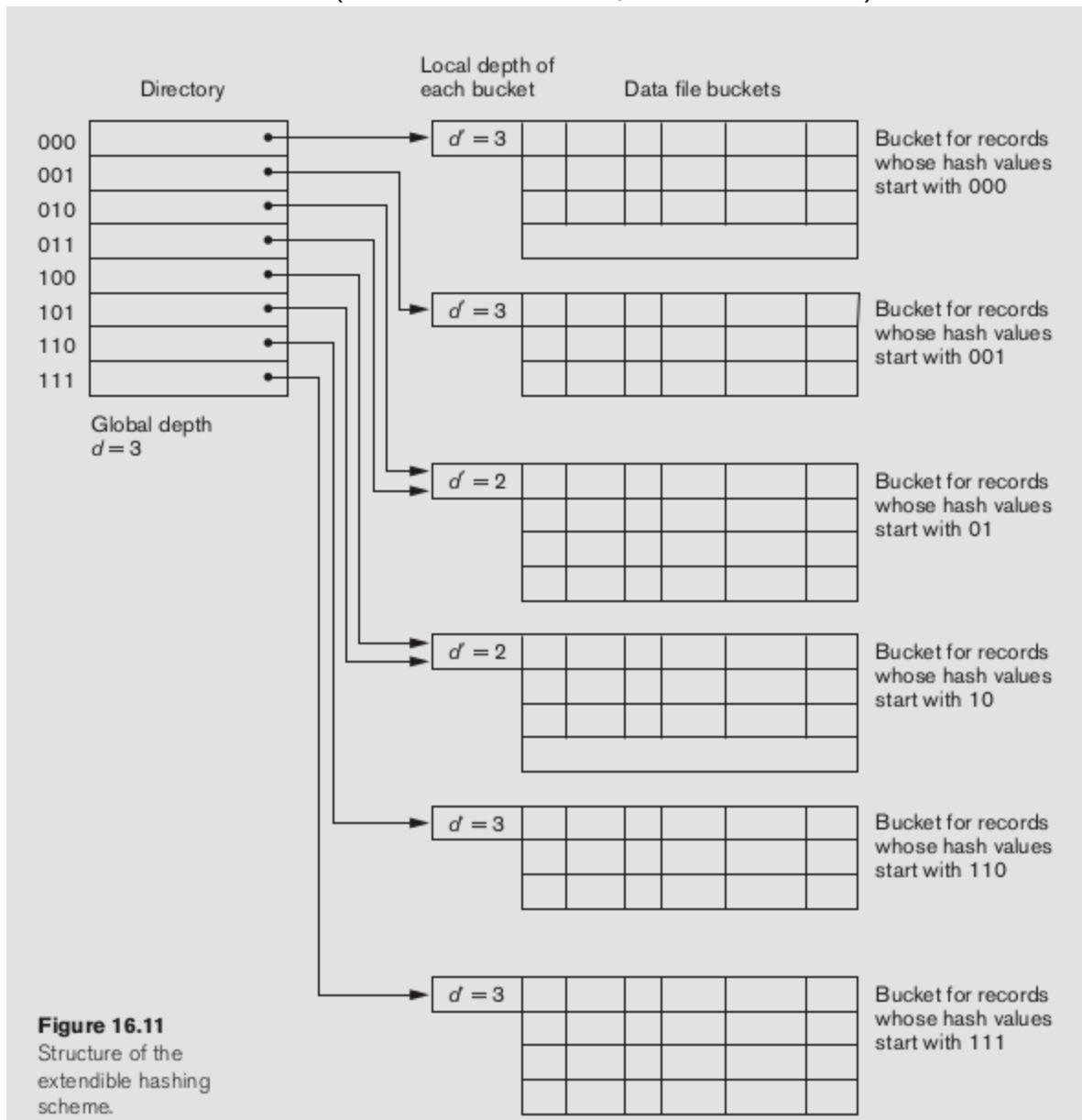
## 동적 해싱(Dynamic Hashing)

- 확장가능해싱과의 차이점은 디렉토리를 구성하는 방식이다.
- 두 개의 포인터를 갖는 내부 노드는 (해시된 주소에서) 0비트에 해당하는 왼쪽 포인터와 1비트에 해당하는 오른쪽 포인터를 가진다.
- 디스크 기반 자료구조가 아니다.
- 이를 변형하여 디스크 기반 자료구조로 만든 것이 확장가능해싱이다.
- 트리는 "거의" 균형트리다. Why?



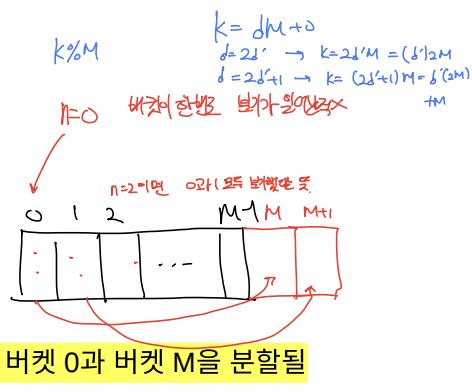
## 확장가능 해싱(Extendable Hashing)

- 디렉토리는  $2^d$  개의 버켓 주소를 갖는 배열이다. 이때  $d$ 는 전역 깊이(global depth)라 하며, 해시 값의 처음  $d$ 개 비트를 디렉토리 배열의 인덱스 값을 결정하는데 사용한다.
- $d$  개 디렉토리 엔트리들은 서로 다른 버켓 주소를 유지할 필요는 없다.
- 처음  $d'$ 개의 비트 값이 갖는 레코드가 하나의 버켓에 저장될 수 있으면 여러 디렉토리 엔트리가 같은 버켓 주소를 유지한다. 각 버켓내의 레코드가 기반으로 하는 비트 수  $d'$ 을 지역 깊이(local depth)라고 한다.
- 지역 깊이  $d'$ 과 전역 깊이  $d$ 가 같은 버켓에서 오버플로가 발생할 경우 디렉토리 배열 내의 엔트리 수는 2배가 된다.
- 어떤 레코드를 삭제한 후, 모든 버켓에 대해  $d > d'$ 인 경우
  - 디렉토리 배열내의 엔트리 수는 절반이 된다.
- 대부분의 레코드 검색은 두 개의 블록 접근(디렉토리에 대한 블록접근, 버켓에 대한 블록 접근)을 필요로 한다.



## 선형 해싱(Linear Hashing)

- 디렉토리를 사용하지 않고, 해시 파일의 버켓수를 동적으로 늘리거나 줄인다.
- 초기에는 M 버켓 사용: 0, 1, ..., M - 1
- 초기 해시 함수  $h_i(K) = K \bmod M$
- 계속되는 충돌로 인해 오버플로 레코드가 발생하면, 버켓 0, 1, 2, ..., n 순서로 분할한다.
- 버켓 0은 버켓 0과 버켓 M으로 분할한다. 버켓 1은 버켓 1과 버켓 M+1으로 분할한다. ...
  - 중요한 특성:**  $h_i$ 를 기반으로 하여 버켓 0에 해시된 모든 레코드는  $h_{i+1}$ 을 기반으로 하여 버켓 0과 버켓 M을 분할될 수 있다. why?
- 해시 함수는 다음과 같다:  $h_{i+1}(K) = K \bmod 2M$
- $h_i(K) < n$ 이면 해시 함수는  $h_{i+1}$ 이고, 그렇지 않으면 해시 함수는  $h_i$ 이다.
- j번 분할하면 해시 함수는 다음과 같다:  $h_{i+j}(K) = K \bmod (2^j M)$
- 특정 버켓의 오버플로우가 발생할 때마다 분할하지 않고, 다음과 같이 분할할 수도 있다.
  - 적재인수  $r/(bfr * N)$  임계값(예, 0.9)을 넘어가면 분할한다.
  - 적재인수가 임계값(예, 0.9)을 넘어가면 합병한다)



**Algorithm 16.3.** The Search Procedure for Linear Hashing

```

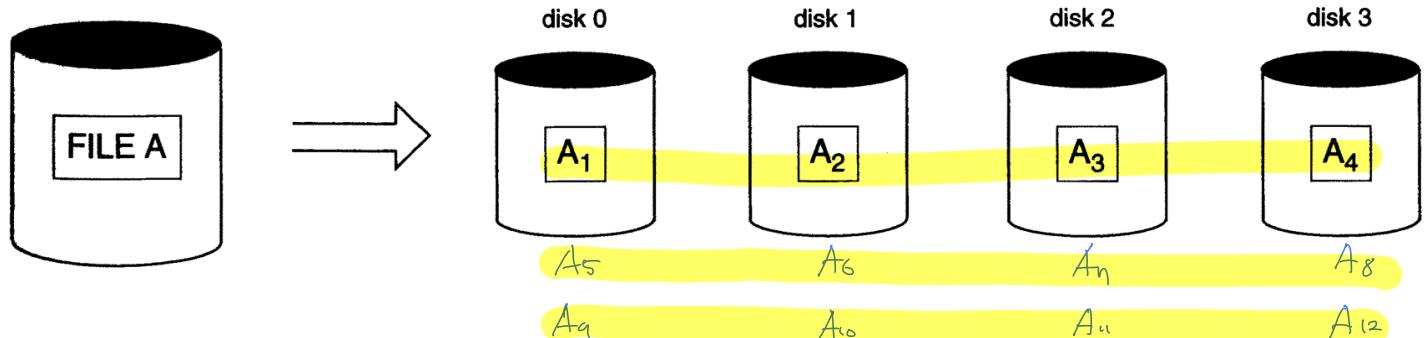
if  $n = 0$ 
  then  $m \leftarrow h_j(K)$  (* $m$  is the hash value of record with hash key  $K$ *)
else begin
   $m \leftarrow h_j(K)$ ;
  if  $m < n$  then  $m \leftarrow h_{j+1}(K)$ 
end;

```

search the bucket whose hash value is  $m$  (and its overflow, if any);

# RAID 기술을 이용한 병렬 디스크 접근

- 보조 기억 장치 기술의 성능과 신뢰도를 프로세서 기술의 수준으로 올려야 한다.
- 보조 기억 장치 기술의 주요한 발전은 RAID(Redundant Arrays of Inexpensive Disks)의 개발로 대표된다.
- RAID의 주요 목표는 메모리와 마이크로프로세서의 성능 향상과 균형을 맞출 수 있도록 디스크의 성능을 획기적인 비율로 향상시키는 데 있다.
- 자연스러운 해결책은 여러 개의 작고 독립적인 디스크를 배열로 구성하여 하나의 고성능 디스크처럼 동작하도록 하는 것이다. 여기에는 디스크의 성능 향상을 위해 병렬성을 사용하는데, 이 개념을 데이터 스트라이핑(data striping)이라 한다.
- 데이터 스트라이핑은 여러 개의 디스크가 하나의 크고 빠른 디스크처럼 보이도록 데이터를 다중 디스크로 투명하게 분산시킨다.
- Data striping: 파일 A가 네 개의 디스크에 스트라이핑



## Reliability

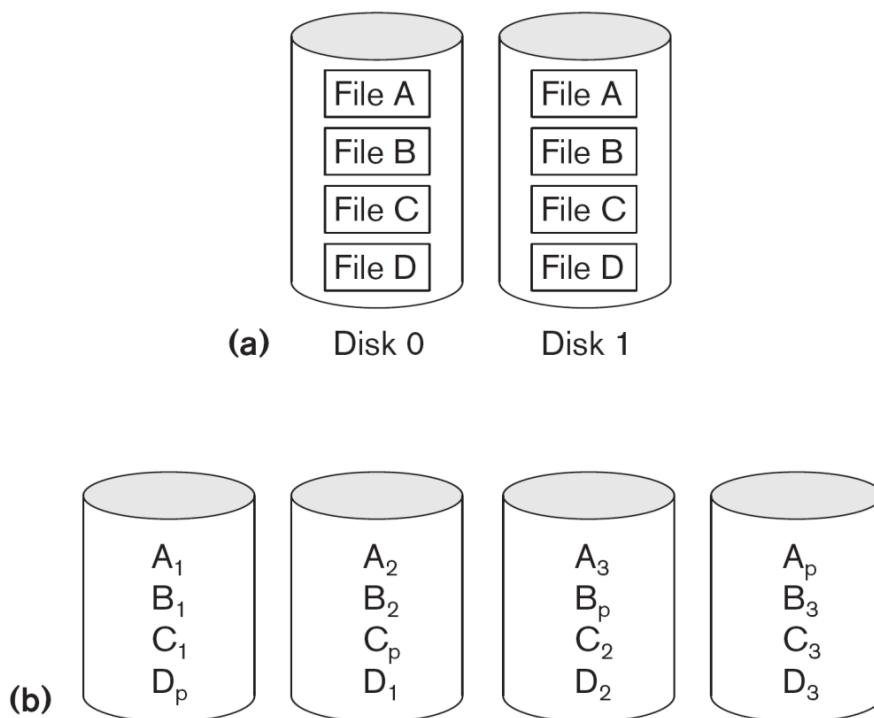
- 신뢰성을 향상시키는 첫 번째 방법은 데이터를 두 개의 동일한 물리적 디스크에 중복해서 기록하는 반사(mirroring) 또는 그림자(shadowing) 기법을 사용하는 것이다.
- 두 번째 방법은 디스크 오류 발생시 손실된 정보를 재구축하는 데 사용할 부가 정보를 저장하는 것이다.
  - 여분의 정보를 계산하기 위해 패리티 비트나 해밍 코드 같은 특별한 코드를 포함한 에러 검출 코드를 사용한다.
  - 중복된 정보를 디스크 배열상에 분산시키기 위해 몇 개의 디스크에 여분의 정보를 저장하는 방식과, 모든 디스크에 균등하게 여분의 정보를 저장하는 방식이 있다. 두 번째 방식이 더 좋은 부하 균등을 제공한다.

## Data striping granularity

- Bit-level data striping : 데이터의 각 바이트를 비트들로 분할하여 비트들이 서로 다른 디스크에 분산함으로 더 작은 단위를 데이터 전송에 사용하는 방식이다.
- Block-level data striping : 데이터를 분산하는 단위로 파일의 블록을 사용하는 방식이다. 하나의 블록에 다수의 요청을 각 디스크에 의해 병렬로 처리할 수 있어서, I/O 요청 대기 시간이 줄어든다.

## RAID Level

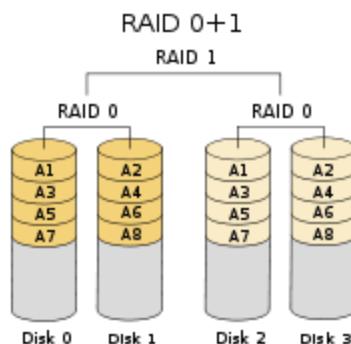
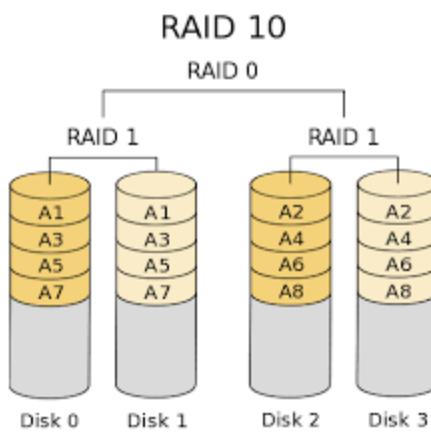
- RAID의 구조는 데이터 분산(스트라이핑)의 단위와 여분의 정보를 계산하는 데 사용되는 패턴의 두 가지 요인의 조합 방법에 따라 서로 구별된다.
  - RAID 레벨 0은 여분의 데이터를 갖지 않아서 데이터의 간신히 중복되지 않으므로 가장 좋은 쓰기 성능을 가진다.
  - RAID 레벨 1은 디스크들을 복사한다.
  - RAID 레벨 5는 블록 레벨 데이터 스트라이핑을 이용하며, 데이터와 패리티 정보를 모든 디스크에 분산시킨다.
- (a) RAID Level 1 (b) RAID Level 5

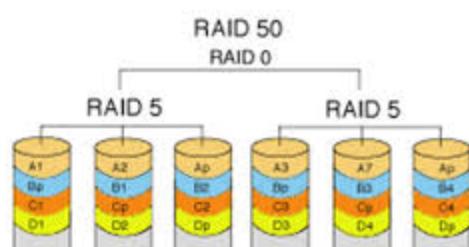


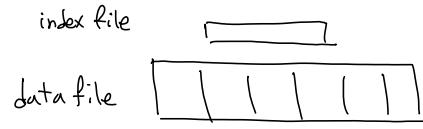
**Figure 16.14**

Some popular levels of RAID.  
 (a) RAID level 1: Mirroring of data on two disks. (b) RAID level 5: Striping of data with distributed parity across four disks.

- RAID 레벨 1에서 디스크 오류를 복구하는 것이 가장 쉽다.
  - **레벨 1은 로그 저장 같은 중요한 응용을 위해 사용한다.**
- RAID 레벨 5는 대용량의 데이터 저장을 위해 주로 사용되며, 높은 전송률을 제공한다.
- 현재 가장 널리 사용되고 있는 RAID 기술은 스트라이핑을 지원하는 레벨 0, 반사 기능을 가진 레벨 1, 패리티를 위한 추가적인 디스크를 가진 레벨 5이다.
- 주어진 여러 응용을 위한 RAID 설정을 설계하기 위해서는 RAID의 레벨, 디스크의 수, 패리티 기법의 선택, 블록 레벨 스트라이핑을 위한 디스크 그룹핑 방법 등 여러 가지 사항을 고려해야 한다.







## 데이터베이스 파일 인덱싱 기법, ~~B-Tree~~ 및 **B<sup>+</sup>-Tree**

- 단일-단계 순서 인덱스들의 유형
  - 기본 인덱스
  - 클러스터링 인덱스
  - 보조 인덱스
- 다단계 인덱스
- 해싱(Hashing)
- B-Tree와 B<sup>+</sup>-Tree를 사용하여 구현하는 동적 다단계 인덱스

### 인덱스의 생성 및 삭제

```
create index citiesx on customers (city);
```

1. without index – table scan (data file)이 필요
2. with index?

```
select * from customers where city = 'Boston' and discnt between 12 and 14;
```

- Dropping index:

```
drop index citiesx;
```

- Unique index :

```
create unique index cidx on customers (cid);
```

- cid가 unique함을 index가 보장함
- 일반적으로 unique나 PK 제약조건에 대해서 DBMS가 자동으로 unique index를 생성함

## Index - Access Path

- 인덱스 - 접근 경로
- 단일 단계 인덱스는 데이터 파일내의 레코드를 효과적으로 찾도록 도와주는 보조 파일임
- 인덱스는 보통 파일내의 한 필드에 대해 정의된다 (여러 필드에 대해 정의될 수도 있음)
- 인덱스는 <필드값, 레코드에 대한 주소>로 구성된 엔트리들을 저장한 파일이다.
- 인덱스 파일은 필드값에 따라 정렬되어 있다.
- 인덱스는 파일에 대한 접근 경로(access path)라고 불린다.
- 인덱스 엔트리는 실제 레코드 크기보다 훨씬 작기 때문에, 인덱스 파일은 데이터 파일보다 훨씬 적은 디스크 블록을 차지한다.
- 인덱스에 대한 이진 탐색으로 데이터 파일의 해당 레코드에 대한 주소를 얻을 수 있다.
- 인덱스는 밀집 또는 희소 인덱스가 될 수 있다.  
인덱스는 정렬되어 있는 순서 파일이다.  $\log_2$  경로 탐색으로 가장 원래는 트리를 따라 내려온다.
  - 밀집 (dense)인덱스는 데이터 파일내의 모든 탐색 키 값(즉, 모든 레코드)에 대한 인덱스 엔트리를 갖는다.
  - 희소 (sparse 또는 비밀집 nondense) 인덱스는 탐색 값의 일부에 대해서만 인덱스 엔트리를 갖는다.

## Example

- 주어진 데이터 파일: EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ... )
  - 데이터 파일에 대한 가정:
  - 레코드 크기  $R = 150$  바이트  $Bfr = \frac{B}{R} = \lfloor \frac{512}{150} \rfloor = 3$
  - 블록 크기  $B = 512$  바이트
  - 레코드 개수  $r = 30000$  레코드

설명나온

얻을 수 있는 값:

- 블록킹 인수(Blocking Factor)  $Bfr = \lceil B \div R \rceil = \lceil 512 \div 150 \rceil = 3$  레코드/블록
- 화일의 블록 수  $b = \lceil r \div Bfr \rceil = \lceil 30000 \div 3 \rceil = 10000$  블록
- SSN 필드에 대한 인덱스에서, 필드 크기  $V_{SSN} = 9$  바이트라고 가정하고,
- 레코드 포인터 크기  $P_R = 7$  바이트라고 가정한다. 그러면:
- 인덱스 엔트리 크기  $R_I = (V_{SSN} + P_R) = (9 + 7) = 16$  바이트
- 인덱스 블록킹 인수(Index Blocking Factor)  $Bfr_I = \lfloor B \div R_I \rfloor = \lfloor 512 \div 16 \rfloor = 32$  엔트리/블록
- 인덱스 블록 수  $b_I = \lceil r \div Bfr_I \rceil = \lceil 30000 \div 32 \rceil = 938$  블록
- 이진 탐색시 접근할 블록 수  $\lceil \log_2 b_I \rceil = \lceil \log_2 938 \rceil = 10$  블록

$$\overline{\overline{30000}}_3$$

- 이 비용은 다음의 평균 선형 탐색 비용보다 훨씬 적은 비용임: 블록 접근수  $(b/2) = 10000/2 = 5000$
- 만약 파일의 레코드들이 정렬되어 있으면, 이에 대한 이진 탐색 비용은 다음과 같다: 블록 접근수  $\lceil \log_2 b \rceil = \lceil \log_2 10000 \rceil = 13.14$

$$\log_2 2^{13} \cdot 10 = 10 + \log_2 10 = 10 + 3 \dots$$

$$10^6 = 10^3 \cdot 10^3 = 2^6 \cdot 2^{10}$$

## 단일 단계 인덱스의 유형

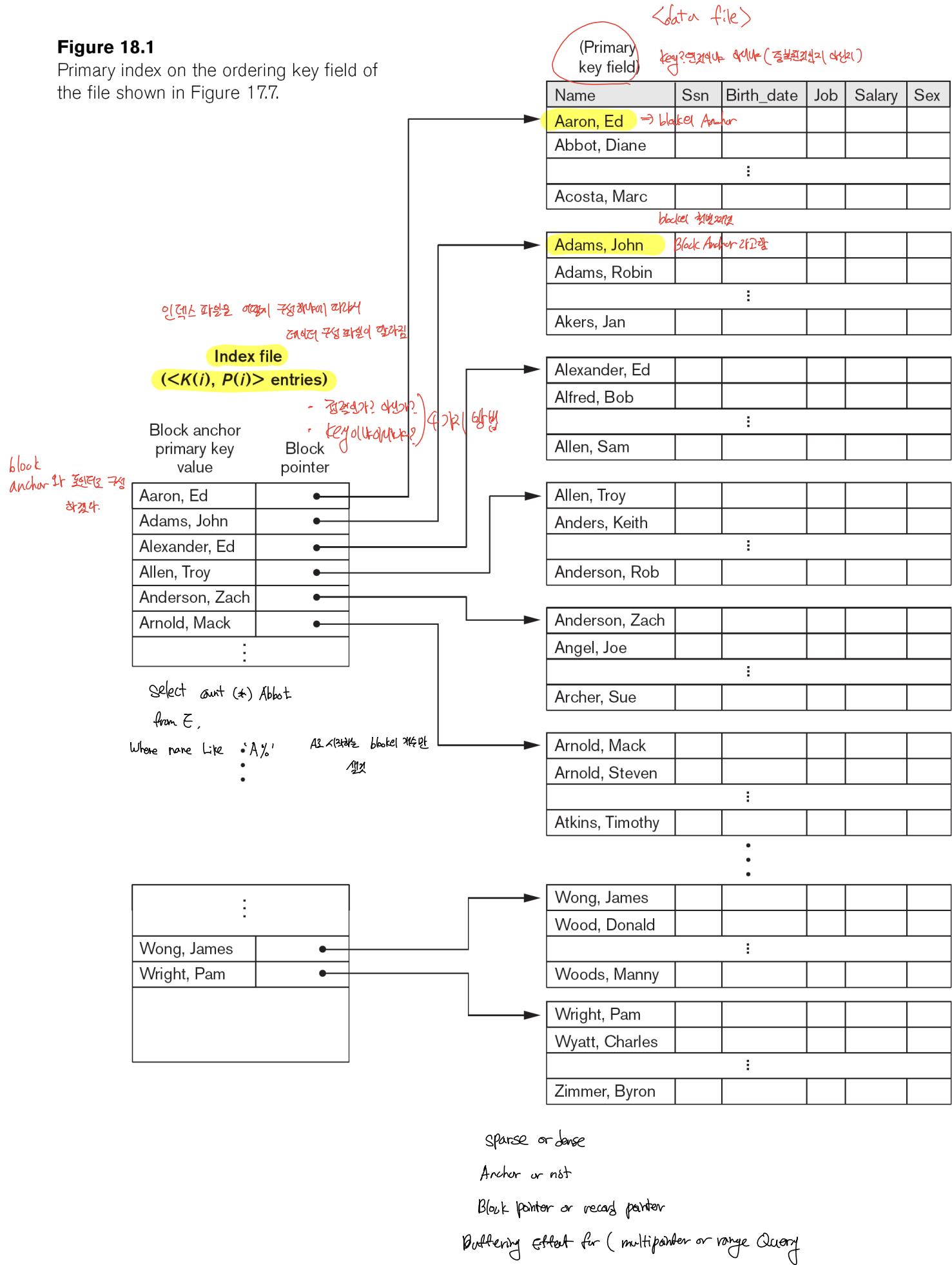
## 기본 인덱스(Primary Index)

- 순서화일(ordered data file)에 대해 정의할 수 있는 인덱스이다.
- 이 데이터 파일은 키 필드 (key field) 순으로 정렬돼 있어야 한다.
- 데이터 파일의 각 블록에 대해 하나의 엔트리를 가지며, 블록 앵커 (block anchor)라 불리는 각 블록의 첫 번째 레코드에 대한 키 필드 값을 엔트리로 갖는다.
- 각 블록의 마지막 레코드를 블록 앵커로 사용하는 방식을 이용할 수도 있다.
- 전체 탐색 값이 아니라 데이터 파일의 각 블록에 대해 하나의 엔트리 (즉, 블록의 앵커 레코드에 대한 키 값)을 가지므로, 기본 인덱스는 비밀집(희소) 인덱스이다.

sparse

**Figure 18.1**

Primary index on the ordering key field of the file shown in Figure 17.7.



## 클러스터링 인덱스(Clustering Index)

데이터 순서화 및 인덱스

nonkey field

- 순서화 파일에 대해 정의할 수 있다.
- 데이터 파일은 각 레코드에 대해 구별된 값을 갖지 않는 필드(즉, 키가 아닌 필드)에 따라 정렬된다.
- 해당 필드에 올 수 있는 각 값의 종류별 (each distinct value)로 하나의 인덱스 엔트리를 포함하며, 이 엔트리에는 그 필드 값을 가진 레코드들이 저장된 첫번째 블록에 대한 주소를 포함한다.
- 클러스터링 인덱스는 삽입과 삭제가 상대적으로 간단한 비밀집(nondense) 인덱스의 한 예이다.
- EMPLOYEE 파일의 키가 아닌 필드 DEPTNUMBER에 대한 클러스터링 인덱스

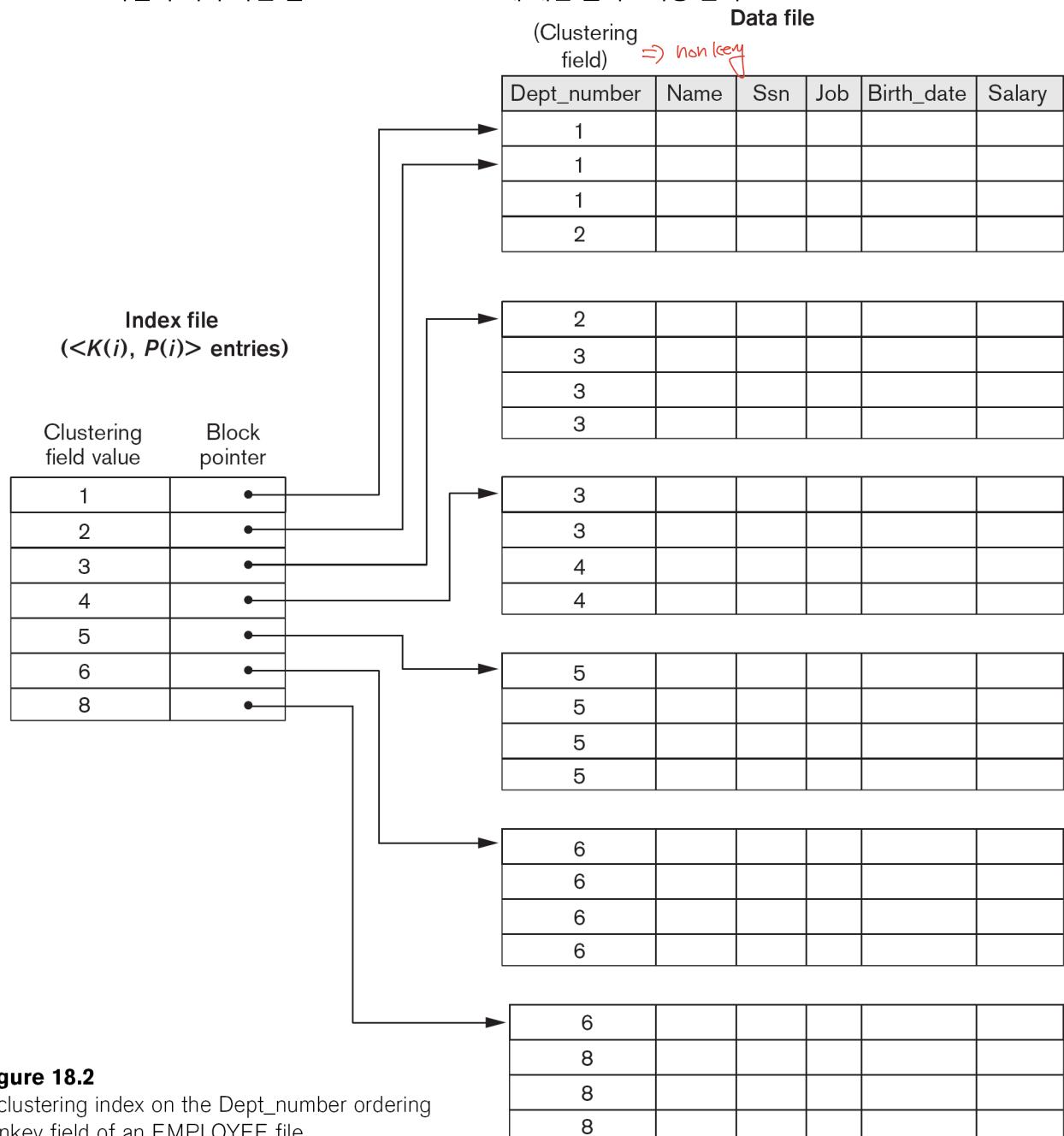


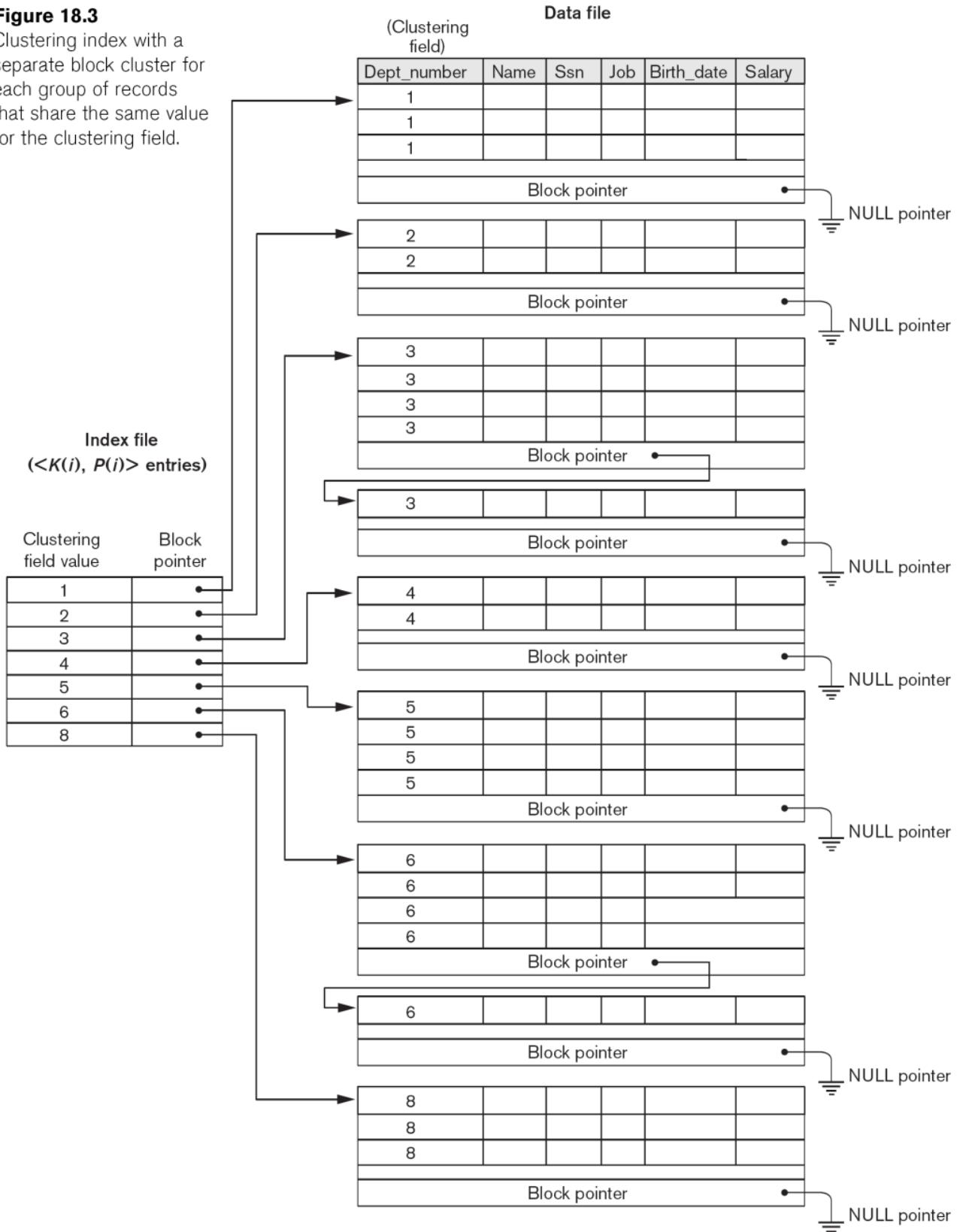
Figure 18.2

A clustering index on the Dept\_number ordering nonkey field of an EMPLOYEE file.

- 같은 클러스터링 필드값을 갖는 레코드들의 각 그룹을 위해 별도의 블록 클러스터를 가지는 클러스터링 인덱스

**Figure 18.3**

Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

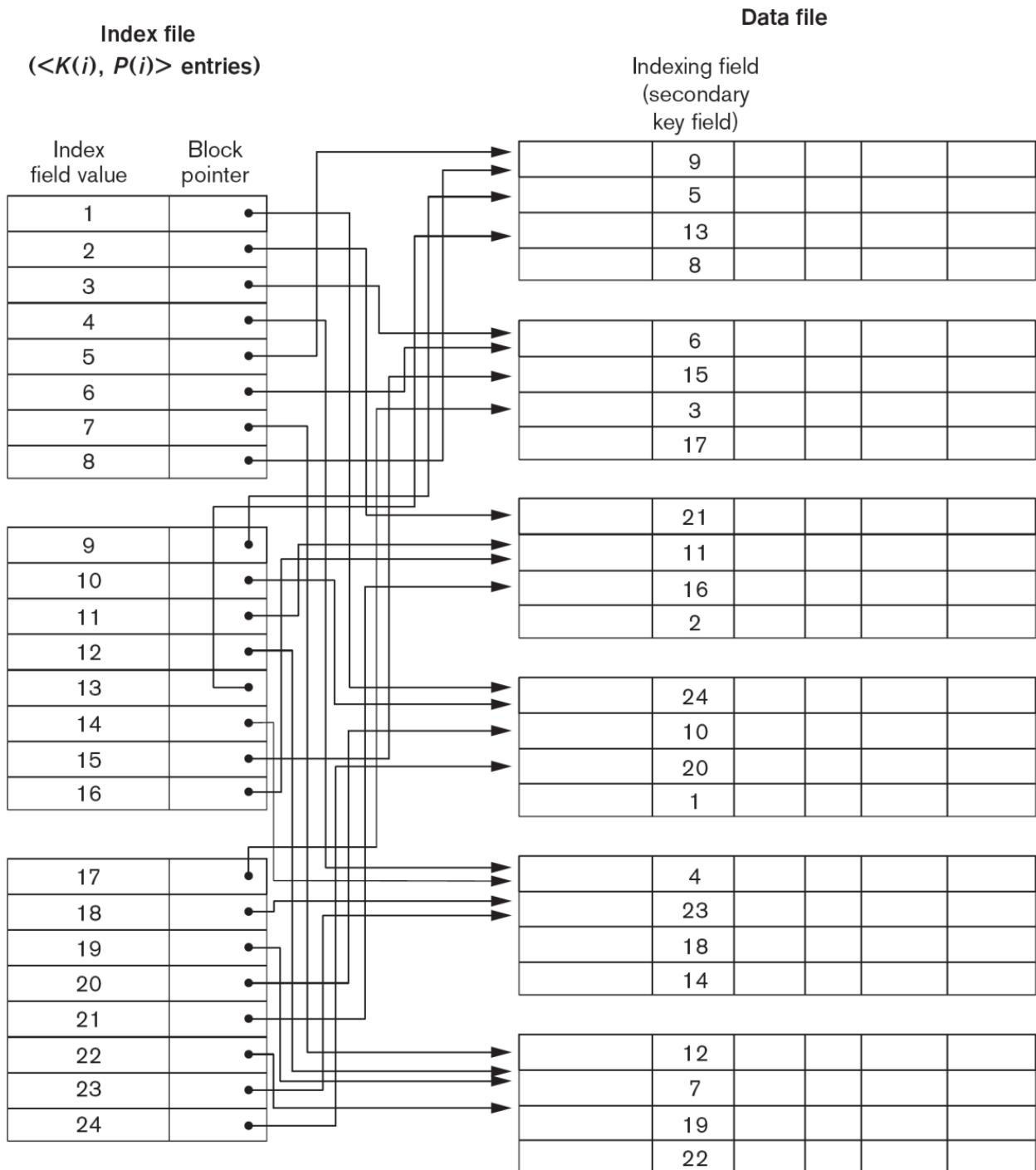


## 보조 인덱스(Secondary Index)

- 보조 인덱스는 기본 접근 방법이 이미 존재하는 파일을 접근하는 보조 수단을 제공한다.
- 보조 인덱스는 후보 키나 모든 레코드에 대해 유일한 값을 갖는 필드 또는 중복된 값을 갖는 키가 아닌 필드에 대해 만들 수 있다.
- 보조 인덱스는 두 개의 필드로 구성된 순서 파일이다.
  - 첫 번째 필드는 인덱스 필드인 데이터 파일의 비순서 필드와 같은 데이터 타입이다.
  - 두 번째 필드는 블록 포인터이거나 레코드 포인터이다.
- 같은 파일에 여러 개의 보조 인덱스가 존재할 수 있다.
- 엔트리에는 레코드에 대한 보조 키 값과 레코드가 저장되어 있는 블록 또는 레코드 자체에 대한 포인터가 있으므로, 키 필드에 대한 보조 인덱스는 밀집 인덱스이다.
- 파일의 비순서 키 필드에 대한 밀집 보조 인덱스(블록 포인터를 갖는 경우)

**Figure 18.4**

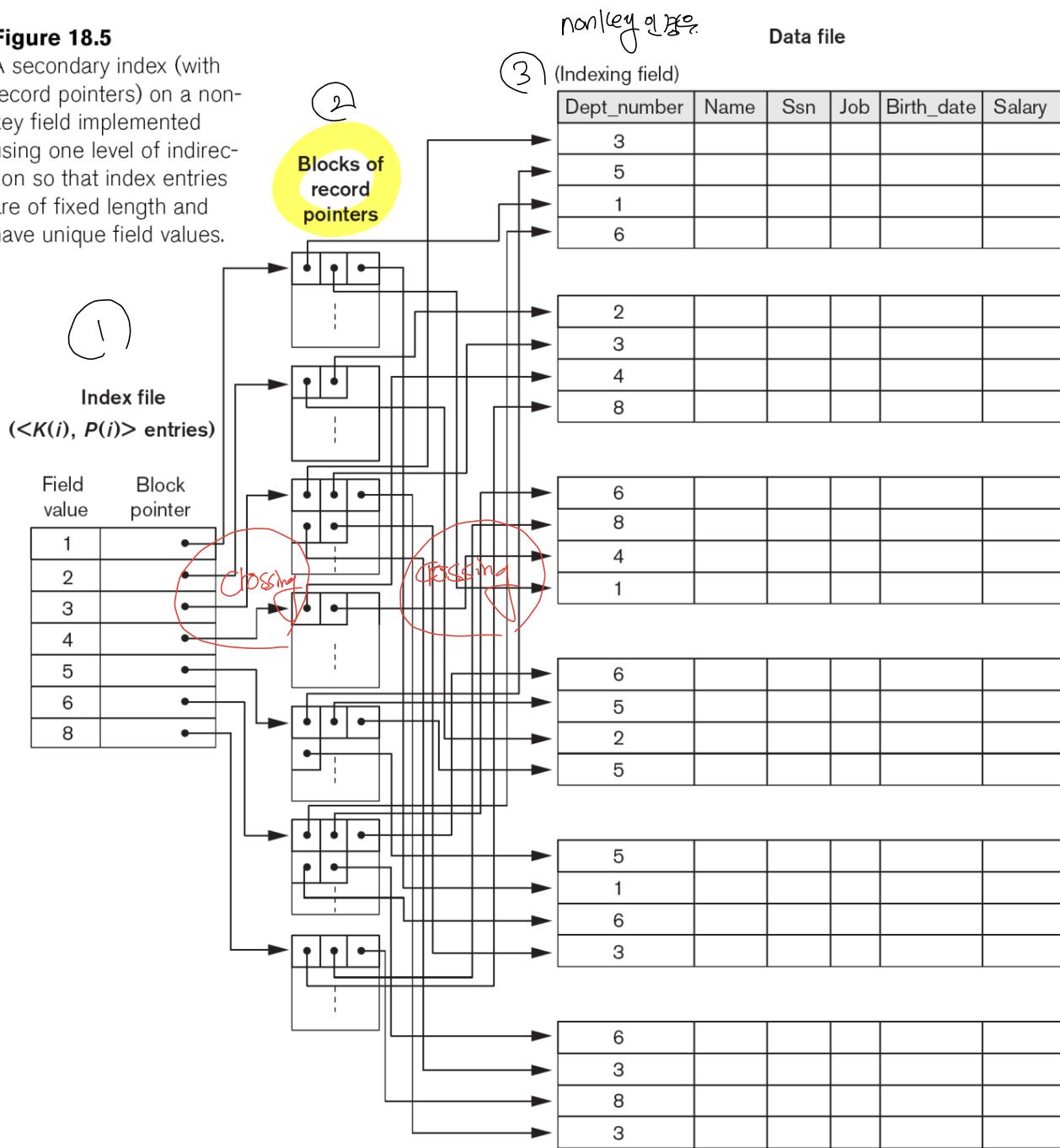
A dense secondary index (with block pointers) on a nonordering key field of a file.



- 인덱스 엔트리들이 고정 길이이고 유일한 필드값들을 갖도록 하나의 간접 단계를 이용하여 구현된, 키가 아닌 필드에 대한 보조 인덱스(레코드 포인터를 갖는 경우)

**Figure 18.5**

A secondary index (with record pointers) on a non-key field implemented using one level of indirection so that index entries are of fixed length and have unique field values.



## Summary

**Table 18.1** Types of Indexes Based on the Properties of the Indexing Field

	<b>Index Field Used for Physical Ordering of the File</b>	<b>Index Field Not Used for Physical Ordering of the File</b>
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

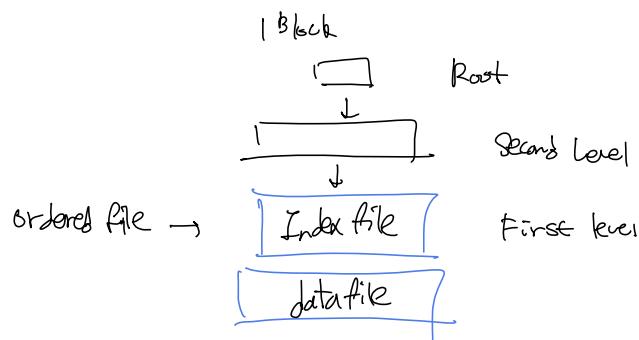
**Table 18.2** Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no <sup>a</sup>
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records <sup>b</sup> or number of distinct index field values <sup>c</sup>	Dense or Nondense	No

## 다단계(Multi-Level) 인덱스

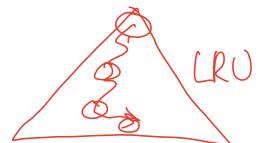
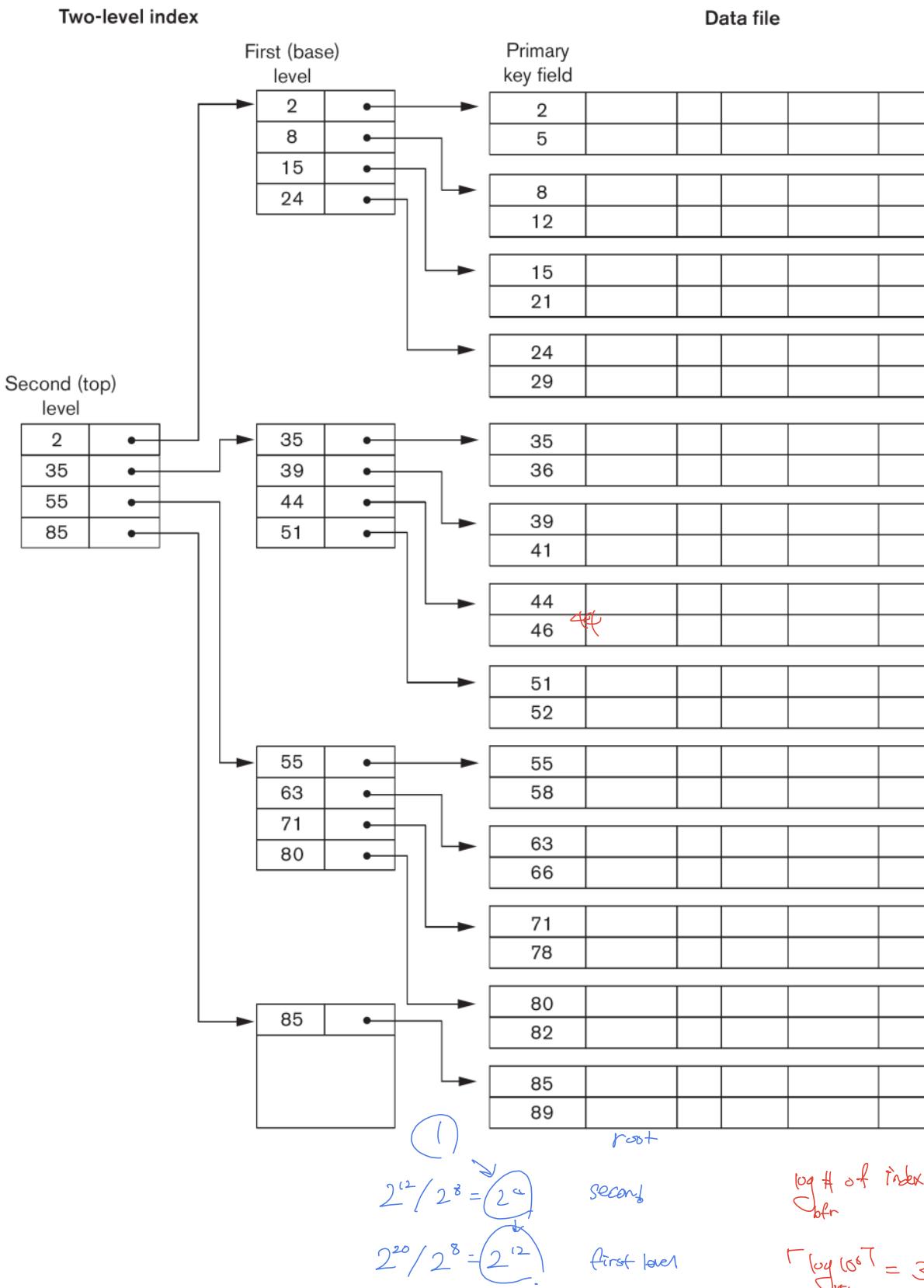
- 단일 단계 인덱스가 순서 화일이므로, 이 인덱스 자체에 대한 기본 인덱스를 만들 수 있다; 이 경우, 원래 인덱스 화일은 첫번째 단계 인덱스라 부르고 그 인덱스에 대한 인덱스는 두번째 단계 인덱스라 부른다.
- 위와 같은 과정을 반복하면 모든 엔트리를 한 블록에 저장할 수 있는 단계가 생기고, 이 단계의 블록을 최상위 단계라고 한다.
- 다단계 인덱스는 첫번째 단계 인덱스가 어떤 인덱스 유형(기본 인덱스, 클러스터링 인덱스, 보조 인덱스)이든지 사용할 수 있다.

- Example: ISAM(Indexed Sequential Access Method) 조직과 유사한 2-단계 기본 인덱스



**Figure 18.6**

A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.



# Search Tree

The diagram shows two binary trees. The first is a balanced binary search tree (BST) with root 5, where each node has at most one child. The second is a skewed binary tree (SKewed) with root 10^c, where every node has either left or right children.

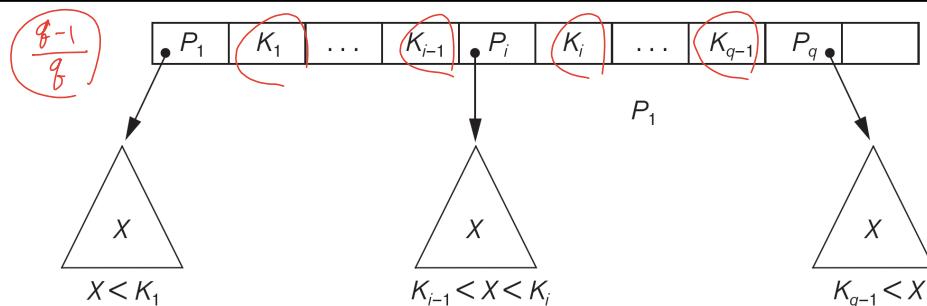
- 다단계 인덱스는 탐색 트리 (Search Tree) 형태를 갖는다. 그러나 인덱스의 모든 단계가 순서 화일이므로, 새로운 인덱스 엔트리에 대한 삽입과 삭제가 매우 복잡하다는 문제점이 있다.

## f-ary Search Tree

- 서브트리에 대한 포인터를 갖는 탐색 트리의 한 노드

**Figure 18.8**

A node in a search tree with pointers to subtrees below it.

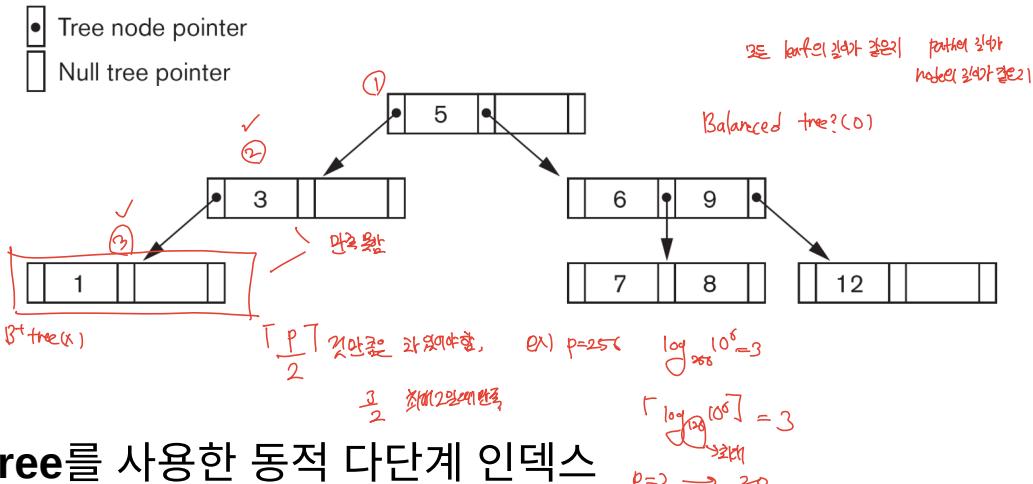


$$\log_{10} 250 \rightarrow \text{로그를 구하는 것처럼 풀어보기}$$

- 차수가  $p = 3$ 인 탐색 트리

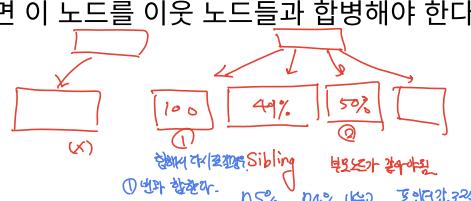
**Figure 18.9**

A search tree of order  $p = 3$ ,



**B-Tree**와 **B<sup>+</sup>-Tree**를 사용한 동적 다단계 인덱스

- 삽입과 삭제 문제 때문에, 대부분의 다단계 인덱스는 B-Tree나 B<sup>+</sup>-Tree 자료 구조를 사용한다. 이들 구조는 각 트리 노드 (즉 디스크 블록)에 새로운 인덱스 엔트리를 저장할 약간의 여유 공간을 남겨둔다.
  - 이들 자료 구조들은 새로운 탐색 키의 삽입과 삭제를 효율적으로 처리할 수 있는 탐색 트리의 변형이다.
  - B-Tree와 B<sup>+</sup>-Tree 자료 구조에서 각 노드는 하나의 디스크 블록을 할당하여 저장한다.
  - 각 노드가 최소한 절반 이상 차 있도록 보장하여 저장 효율을 높일 수 있는 방식이다. 여유 공간의 뜻
  - 완전히 차 있지 않는 노드에 삽입하는 것은 간단히 처리될 수 있다; 만약 노드가 차 있으면 삽입을 위해 두 노드로 분할한다.
  - 노드 분할은 트리의 다른 단계로 파급될 수 있다.
  - 삭제시 절반이상 차 있게 되는 노드에 대한 삭제는 간단히 처리될 수 있다.
  - 삭제시 노드가 절반이하로 차게되면 이 노드를 이후 노드들과 합병해야 한다



방식이다. 여을과 같은 뜻

드가 차 있으면 삽입을 위해 두 노드로

다.

100은 예외적인 예

절편이 가능한데는 예외처럼

Balanced tree 구현 차원.

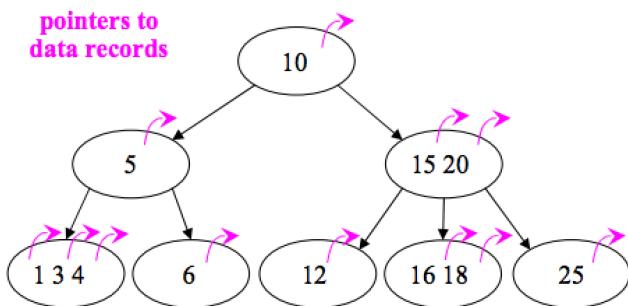
Root도 예외지만 반드시 세로로

## B-Tree와 B<sup>+</sup>-Tree의 차이점

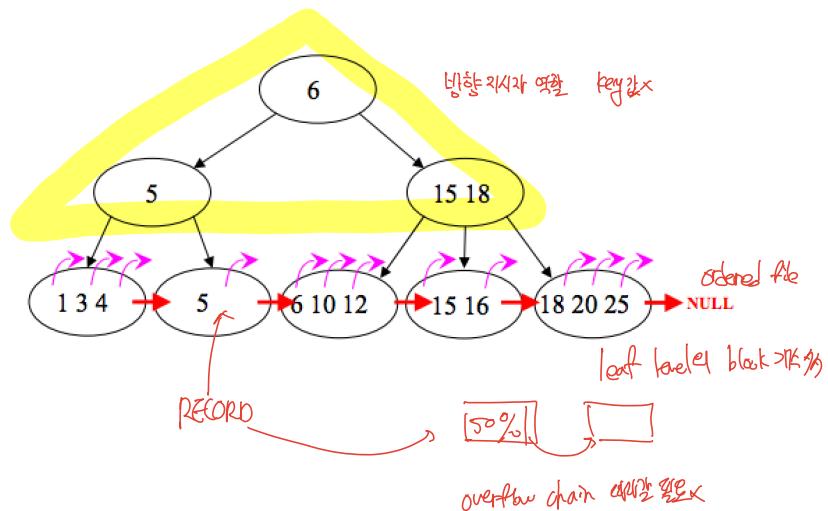
- B-Tree에서는 모든 단계의 노드들이 데이터 레코드들에 대한 포인터들을 갖는다.
- B<sup>+</sup>-Tree에서는 리프 단계의 노드들만 데이터 레코드들에 대한 포인터들을 갖는다.
- B<sup>+</sup>-Tree는 대응하는 B-Tree보다 더 적은 단계를 (더 많은 탐색 값을) 갖는다.
- 대부분의 DBMS는 B<sup>+</sup>-Tree를 사용한다. DBMS에서 B-Tree라고하면 실제로는 B<sup>+</sup>-Tree를 언급하고 있다고 보면된다. (B-Tree는 주로 정보검색과 같이 읽기가 주 작업인 경우에 사용된다.)

- B-tree vs B<sup>+</sup>-tree

B-tree of order 4



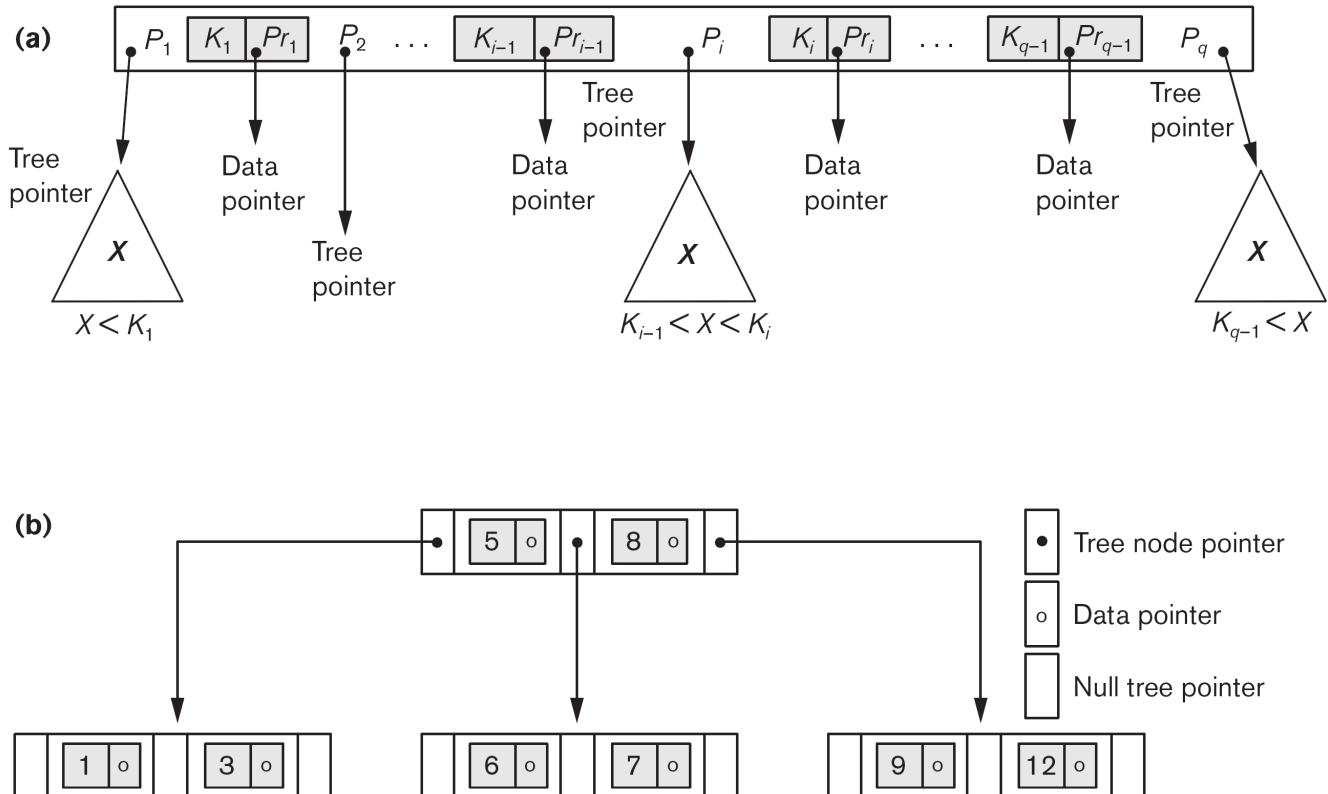
B<sup>+</sup>-tree of order 4



Oracle  
MySQL ) Primary  
Indexing

## B-Tree 구조

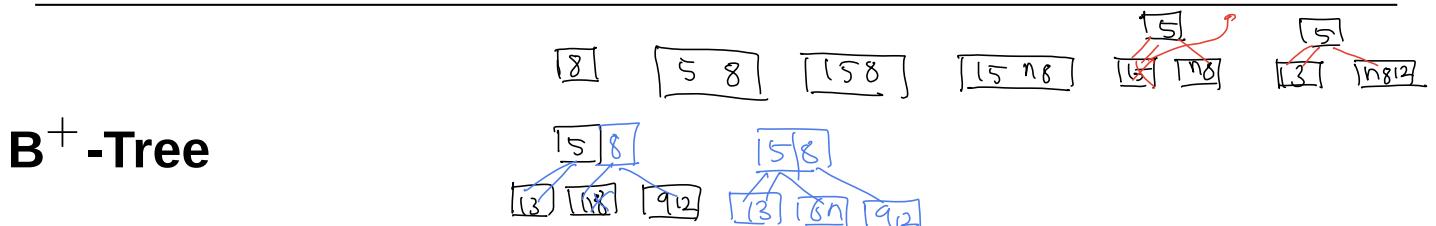
- (a)  $q - 1$ 개의 탐색값을 갖는 B-Tree의 한 노드
- (b) 차수  $p = 3$ 인 B-Tree(삽입 순서는 8, 5, 1, 7, 3, 12, 9, 6이다.)



**Figure 18.10**

B-tree structures. (a) A node in a B-tree with  $q - 1$  search values. (b) A B-tree of order  $p = 3$ . The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

Btree는 5개의 탐색값  
B<sup>+</sup>tree는 5개의 인덱스



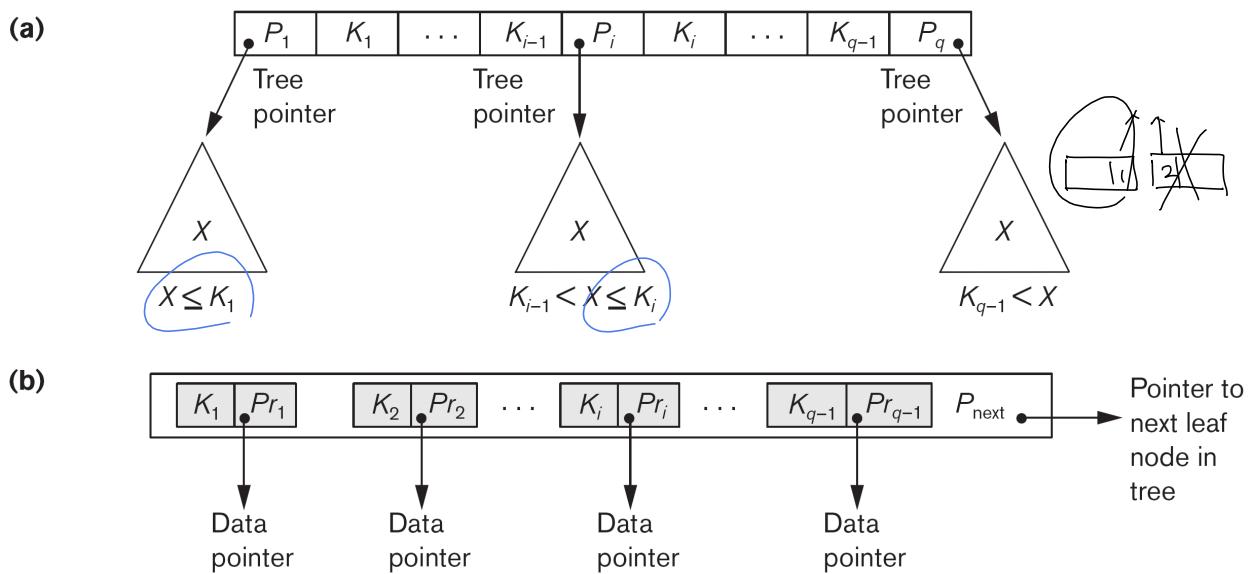
## B<sup>+</sup>-Tree of Order $p$

- (a)  $q - 1$ 개의 탐색값을 갖는 내부 노드
- (b)  $q - 1$ 의 탐색값과  $q - 1$ 의 데이터 포인터를 가지는 B<sup>+</sup>-Tree의 리프 노드

**Figure 18.11**

The nodes of a B<sup>+</sup>-tree. (a) Internal node of a B<sup>+</sup>-tree with  $q - 1$  search values.

(b) Leaf node of a B<sup>+</sup>-tree with  $q - 1$  search values and  $q - 1$  data pointers.



## Internal node structure

1. Each internal node is of the form  $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$  where  $q \leq p$  and each  $P_i$  is a tree pointer.
2. Within each internal node,  $K_1 < K_2 < \dots < K_{q-1}$ .
3. For all search field values  $X$  in the subtree pointed at by  $P_i$ , we have  $K_{i-1} < X \leq K_i$  for  $1 < i < q$ ;  $X \leq K_i$  for  $i = 1$ ; and  $K_{i-1} < X$  for  $i = q$  (see Figure 18.11(a)). 10  
 $P_{\text{leaf}}$
4. Each internal node has at most  $p$  tree pointers.
5. Each internal node, except the root, has at least  $\lceil p/2 \rceil$  tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with  $q$  pointers,  $q \leq p$ , has  $q - 1$  search field values.

## Leaf node structure

1. Each leaf node is of the form  $\langle \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_{\text{next}} \rangle$  where  $q \leq p$ , each  $Pr_i$  is a data pointer, and  $P_{\text{next}}$  points to the next leaf node of the  $B^+$ -tree.
2. Within each leaf node,  $K_1 \leq K_2 \dots, K_{q-1}, q \leq p$ .
3. Each  $Pr_i$  is a data pointer that points to the record whose search field value is  $K_i$  or to a file block containing the record (or to a block of record pointers that point to records whose search field value is  $K_i$  if the search field is not a key).
4. Each leaf node has at least  $\lceil p/2 \rceil$  values.
5. All leaf nodes are at the same level.

## Searching in $B^+$ -Tree

**Algorithm 18.2.** Searching for a Record with Search Key Field Value  $K$ , Using a  $B^+$ -tree

```
n ← block containing root node of  $B^+$ -tree;  
read block n;  
while (n is not a leaf node of the  $B^+$ -tree) do  
    begin  
        q ← number of tree pointers in node n;  
        if  $K \leq n.K_1$  (* $n.K_i$  refers to the  $i$ th search field value in node n*)  
            then  $n \leftarrow n.P_1$  (* $n.P_i$  refers to the  $i$ th tree pointer in node n*)  
        else if  $K > n.K_{q-1}$   
            then  $n \leftarrow n.P_q$   
        else begin  
            search node n for an entry  $i$  such that  $n.K_{i-1} < K \leq n.K_i$   
             $n \leftarrow n.P_i$   
        end;  
    read block n  
end;  
search block n for entry  $(K_i, Pr_i)$  with  $K = K_i$ ; (* search leaf node *)  
if found  
    then read data file block with address  $Pr_i$  and retrieve record  
    else the record with search field value  $K$  is not in the data file;
```

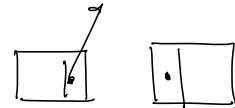
# Insertion in B<sup>+</sup>-Tree

- Full algorithm은 교재 참고

## Insertion

Perform a search to determine what bucket the new record should go into.

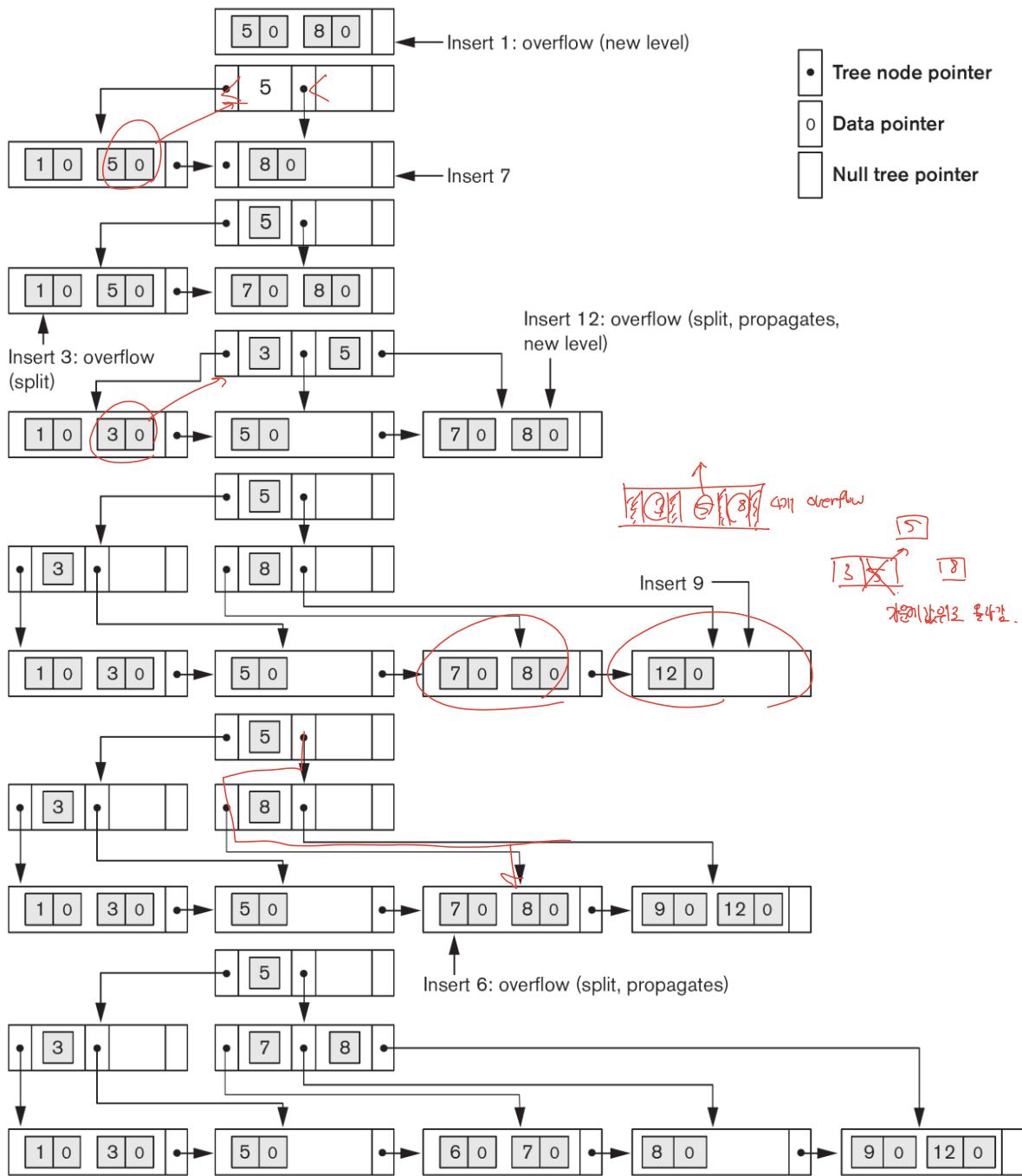
- If the bucket is not full (at most  $b - 1$  entries after the insertion), add the record.
- Otherwise, split the bucket.
  - Allocate new leaf and move half the bucket's elements to the new bucket.
  - Insert the ~~new leaf's smallest key~~ and address into the parent.
  - If the parent is full, split it too.
    - Add the middle key to the parent node.
    - Repeat until a parent is found that need not split.
- If the root splits, create a new root which has one key and two pointers.



- $p = 3$ 이고  $p_{leaf} = 2$ 인 B<sup>+</sup>-Tree에 대한 삽입의 예

$$\begin{array}{ll} p=3 & p_{leaf}=2 \\ 2,3 & 1,2 \end{array}$$

Insertion sequence: 8, 5, 1, 7, 3, 12, 9, 6



**Figure 18.12**

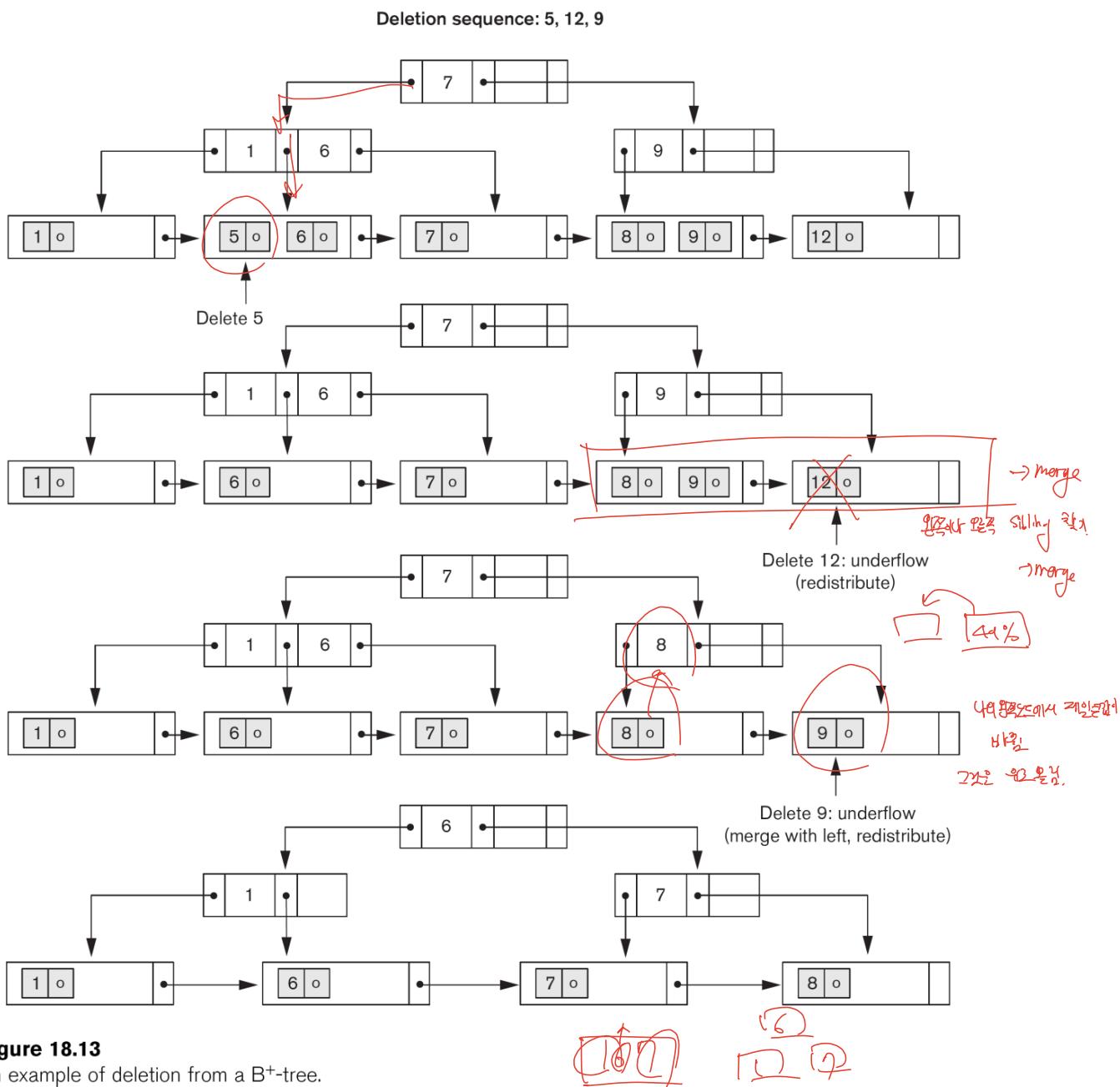
An example of insertion in a B<sup>+</sup>-tree with  $p = 3$  and  $p_{leaf} = 2$ .

## **Deletion in B<sup>+</sup>-Tree**

- Full algorithm은 교재 참고

### **Deletion**

- Start at root, find leaf L where entry belongs.
  - Remove the entry.
    - If L is at least half-full, done!
    - If L has fewer entries than it should,
      - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
      - If re-distribution fails, merge L and sibling.
  - If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
  - Merge could propagate to root, decreasing height.
- 
- B<sup>+</sup>-Tree에 대한 삭제의 예



**Figure 18.13**  
An example of deletion from a B<sup>+</sup>-tree.

## Examples

## The order $p$ and $p_{leaf}$ of B<sup>+</sup>-tree

## Assume

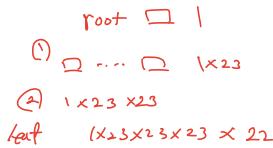
- the search key field is  $V = 9$  bytes long,
  - the block size is  $B = 512$  bytes,
  - a record pointer is  $Pr = 7$  bytes,
  - and a block pointer is  $P = 6$  bytes.

Calculate the order  $p$  and  $p_{leaf}$  of B<sup>+</sup>-tree

$$512 \geq (9+\eta)P_{\text{leaf}} \quad \left\lfloor \frac{512}{16} \right\rfloor = P_{\text{leaf}} = 32 \times 8 \cdot 64 = 2^{22}$$

$$512 \geq p*6 + (p-1)*9$$

$$P = \left\lfloor \frac{512+4}{15} \right\rfloor = 34 \times 0.69 = 23$$



## Levels of B<sup>+</sup>-ree

Assume that each node is 69 percent full. Calculate the average number of entries at root level, level 1, level 2, leaf level:

## How many disk I/O for B+-tree indexing

$\log_{10} N$

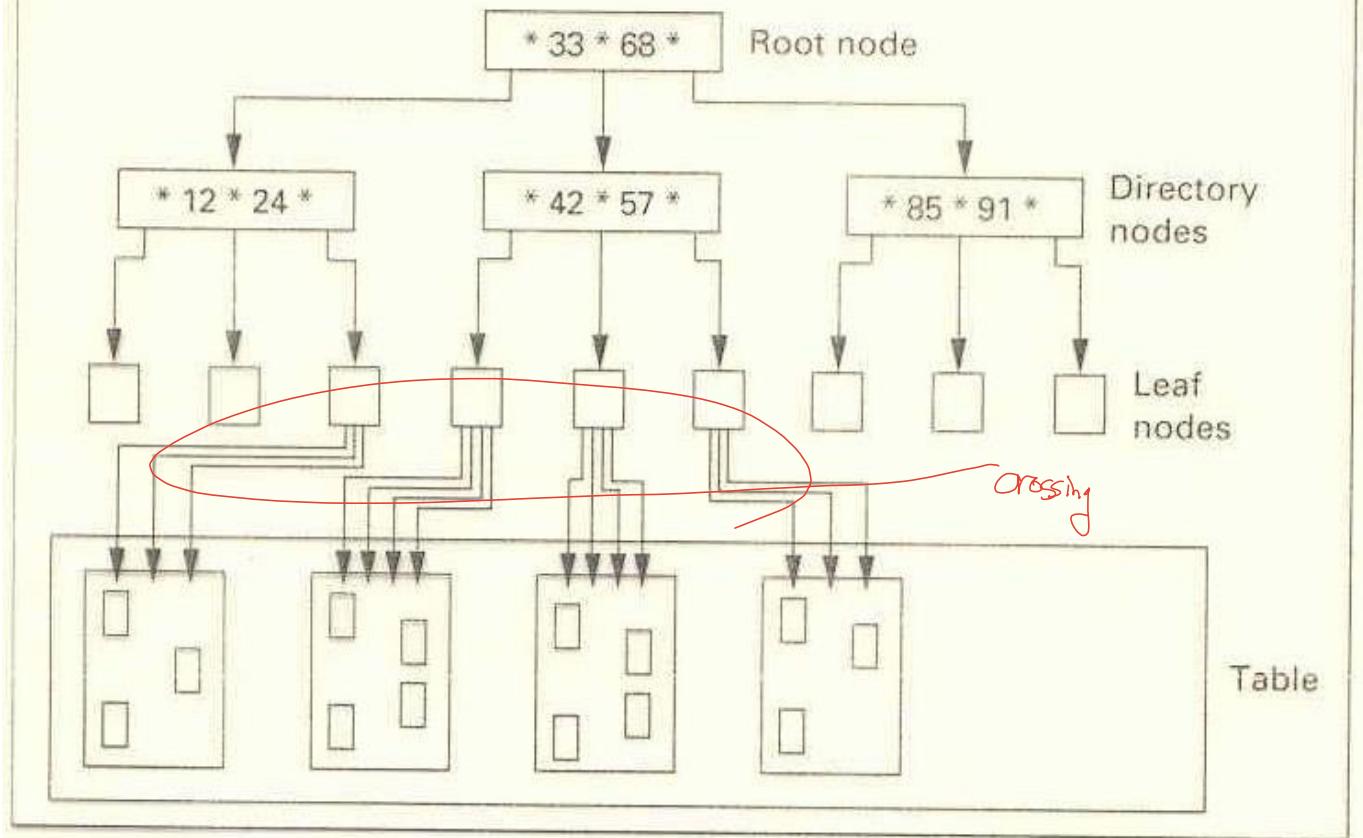
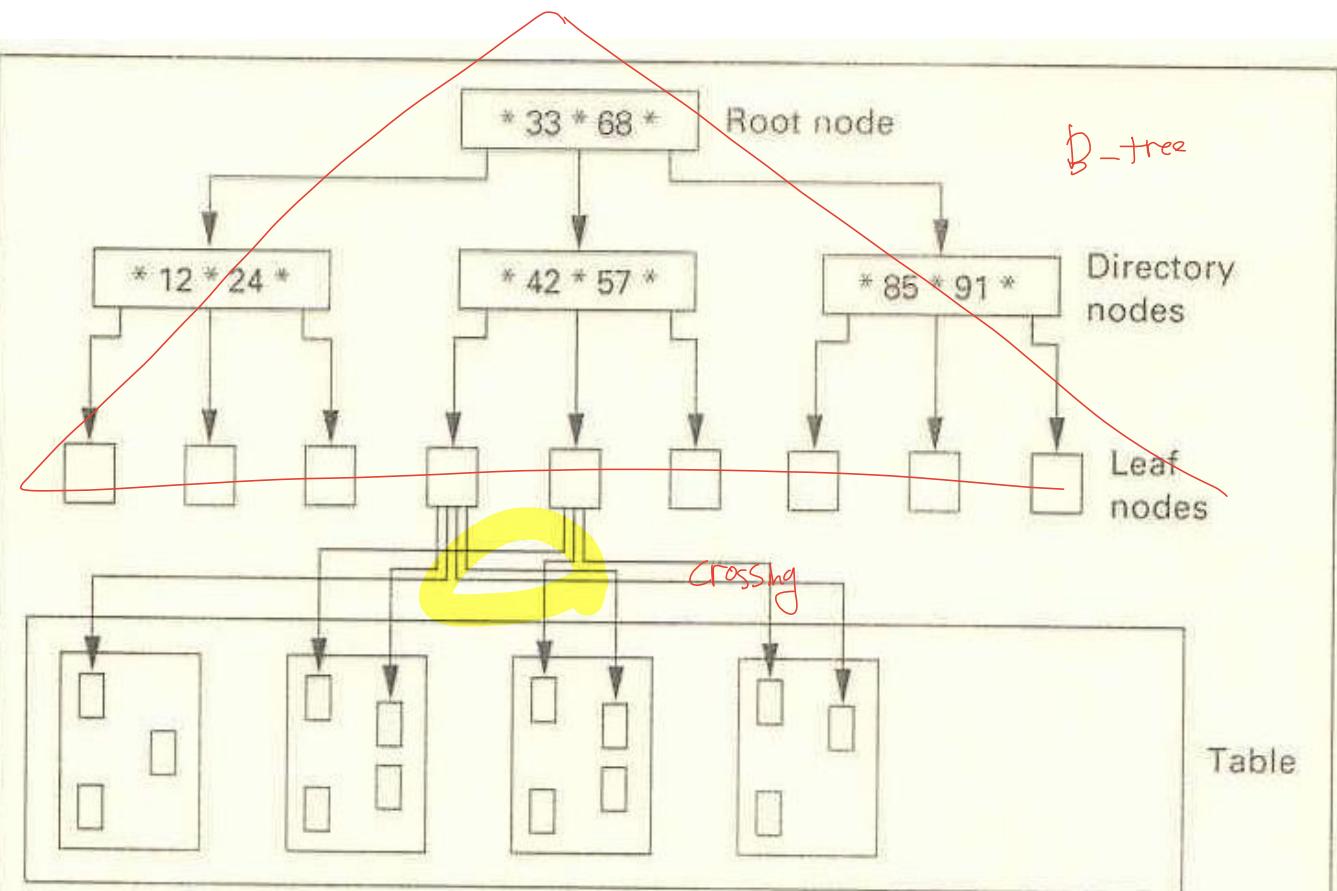
- In general,  $\lceil \log_f N \rceil$  probes are required.  $f$ : fanout,  $N$ : # of rows
  - Disk page buffering: index pages may be buffered in main memory. If top 2 levels are buffered, only 1 I/O to find a ROWID is needed.
  - Assume that # of rows =  $10^6$ , ROWID = 4 bytes, keyvalue = 4 bytes, diskpage = 2 KB
    - How many I/Os to find a ROWID?
    - How many I/Os to find a ROWID if binary search is used?

## Clustered vs Nonclustered

- Assume 10 million customers
- customers : row size 100 bytes on 2KB pages
- Attributes:
  - straddr
  - cityst
  - zipcode
  - age
  - incomeclass : 1 – 10 등급
  - hobby : 50 distinct values
  - major1dept : 고객이 가장 많이 시간을 보낸 곳
  - major2dept : 고객이 두 번째로 많이 시간을 보낸 곳
- mailing labels을 만들려고 함
- SQL Query:

```
select name, straddr, cityst, zipcode from customers where
city = 'Boston' and age between 18 and 50 and hobby in ('...');
```

- Clustering effect. Clustering의 효과는?
  - Assume clustered index on zipcode (Note that zipcode → city) :
  - Boston 지역은 전체 고객의 1/50
  - 200,000 rows
  - 10,000 disk pages
  - 10,000 disk I/O
  - 10,000/80 (초당 80 I/O) = 125 sec
- Assume non-clustered index :
  - where 절의 다른 조건을 만족하는 row가 전체의 1/5
  - 40,000 rows가 500,000 disk pages에 흩어져 있다.
  - 40,000 disk I/O / 80 (초당 80 I/O) = 500 sec



## Reorganizing Clustered pages

- Eventually, the index can lose much of its clustering property  $\Rightarrow \text{REORGANIZE data pages}$
- 언제 재조직이 가능한가?
- 24-hours up-time system인 경우는?

B<sup>+</sup> tree      Primary      Indexing

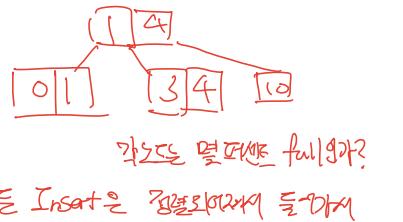
상당히 빠르게 데이터 검색하기

## Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B+-tree requires  $\geq 1$  IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (bulk loading)
- Efficient alternative 1:**
  - sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: Bottom-up B+-tree construction**
  - As before sort entries
  - And then create tree layer-by-layer, starting with leaf level
  - Implemented as part of bulk-load utility by most database systems

한번 만들었는데

여기로 가시(들)기



각 노드는 몇 페스토 full인가?

모든 Insert는 정렬된 데이터를 가짐

