8 장

First Normal Form

Domain 이 atomic (요소들이 더 이상 나눌 수 없는 unit)이어야 함. 도메인이란? Attribute 가 가질 수 있는 값의 집합

이 때 모든 domain 이 1NF 일 때 Relational schema 가 1NF 라고 하고, DB schema 의 모든 relational schema 가 1NF 여야 DB 가 1NF 라고 함.

모든 relational 이 1NF 라고 가정.

FD

$\alpha \subseteq R$ *and* $\beta \subseteq R$

The *functional dependency* $\alpha \rightarrow \beta$ holds on R if and only if

- for any *legal* relations *r*(R),
- whenever any two tuples t_1 and t_2 of *r* agree on the attributes α,
- they also agree on the attributes β.
- That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

Closure of a Set of FDs

3 단논법이 적용됨.

F 에 의해 논리적으로 적용되는 모든 FD 의 set 은 F 의 closure 라고 함. (F^+로 나타냄)

We can find all of F^+ by applying *Armstrong's Axioms*:

- if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (***reflexivity***) ✓
- if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$ (***augmentation***) ✓
- if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (***transitivity***) ✓

이러한 rule 들은 sound (올바른 FD 만 generate) and complete (모든 FD 를 generate).

Boyce-Codd Normal Form - formally

R 이 "good" form 인지 판단하는 걸 원함.

Definition: A relation schema R is in *BCNF* (with respect to a set F of FDs) if for each FD $\alpha \rightarrow \beta$ in F^+ ($\alpha \subseteq R$ and $\beta \subseteq R$), at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for R

즉 a->b 가 자명하거나, 아닌 경우는 a 가 R 의 superkey 여야 함.

Definition: Let R be a relation scheme

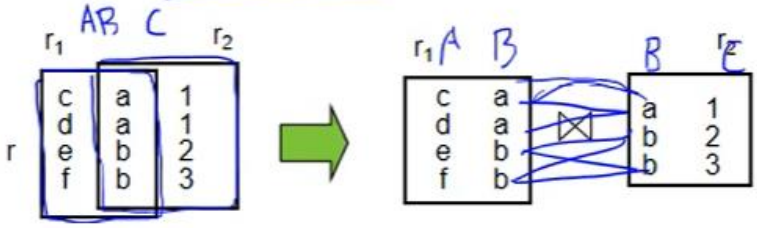
$\{R_1, ..., R_n\}$ is a *decomposition* of R

if $R = R_1 \cup ... \cup R_n$ (i.e., all of R's attributes are represented)

Lossless-join Decomposition

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

The information content of the original relation r is always the basis



Lemma: {R1, R2}는 $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$ 이면 lossless join decomposition 이다.

Dependency Preservation

Definition

Let F set of FD on R. $\{R_1, ..., R_n\}$ decomposition of R.

The ***restriction*** of F to R_i, denoted F_i, is the set of all FDs in F^+ that include only attributes of R_i

Definition

Let $F = F_1 \cup ... \cup F_n$.

The decomposition is ***dependency-preserving*** if $F^+ = F'^+$

Definition 은 F 의 closure 가 같아야 한다는 점에 유의. F = F'이면 closure 는 당연히 같음.

Third Normal Form

A relation schema R is in ***third normal form (3NF)*** if for all $\alpha \rightarrow \beta$ in F^+ at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
- α is a superkey for R
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.
  (*NOTE*: each attribute may be in a different candidate key)

Use of Attribute Closure

1. super key 체크: a+를 계산해서 a+가 R 을 포함하는지 확인.
2. FD 체크: a->b 가 F+에 포함되는지 확인. b 가 a+에 속하는지 확인하면 됨.
3. F 의 closure 의 계산: 스키마 R 의 모든 subset 에 대해 closure 를 구하면 F+가 됨.

Testing for BCNF

a->b 가 BCNF 를 위반하는지 체크함.

1. a+를 계산.
2. a+가 R 의 모든 attribute 를 포함하는지 확인 (그러면 R 의 superkey)

R 이 BCNF 인지 체크함.

1. F+를 구하지 않고 F 의 FD 만 BCNF 를 위반하지 않으면 됨. 그러면 F+도 BCNF 를 위반하지 않음이 증명되어 있음.

하지만 R 의 decomposition 을 체크하려면 F+를 써야함. (이 경우 F 는 부정확)

Denormalization for Performance

Redundancy 를 피하고 싶음. 하지만 performance 를 위해 가끔씩은 비정규화를 사용할 수도 있음.

대체 1: denormalized 된 relation 을 사용함.

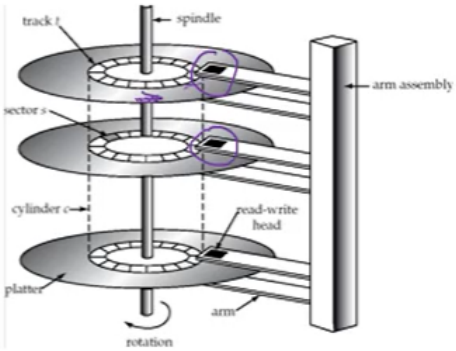
- lookup 이 빠름, update 에 더 많은 공간과 실행 시간이 필요함. 프로그래머들이 더 coding 을 해야하고 에러의 확률이 있음. 얘는 r1 과 r2 를 보존하지 않고 r 만 가지고 있음.

대체 2: materialized view 를 사용함.

- DBMs 에서 가상의 view 를 만드는데, 단점은 1 번과 모두 같고 대신 DBMS 가 해주기 때문에 프로그래머가 할 일은 없고 에러의 확률을 피해줌.

10 장

Magnetic Disks (하드디스크)



Read-write head 가 있음. 각 platter 는 track 으로 나누어져 있음. 각각의 track 은 sector 로 나누어져 있음. 주로 섹터의 크기는 512byte. 보통 트랙당 안 쪽은 500, 바깥 쪽은 1000 개의 섹터가 있음.

Performance Measures of Disks

- Access Time

Read 나 write 가 요청된 시간부터 데이터 전송이 시작된 시간 사이의 길이

Seek time - arm 이 알맞은 트랙까지 가는 시간. 보통 4~10ms.

Rotational Latancy - 섹터가 head 밑으로 오기까지 걸리는 시간. 보통 4~11ms.

- Data-transfer Time

데이터가 disk 로부터 받아오거나 저장되는 비율.

보통 25~100MB/s 임.

- Mean time to failure (MTTF) or MTBF

디스크가 failure 없이 꾸준히 동작하기로 기대되는 시간. 보통 3~5 년.

새 디스크의 경우 고장날 확률이 낮음. 나이가 들면서 수명이 줄어듦.

Reliability via Redundancy

Redundancy 란 추가적으로 정보를 저장하는 것.

- Mirroring (or shadowing)

모든 디스크를 복제함. 미러 디스크의 MTTF 는 57000 년. 수리 시간이나 failure 의 독립성 등에 의존함. 이를 full duplication 이라고 함.

- Parity

모든 n 개의 데이터 비트(혹은 블락)마다 parity bit 를 저장해둠.

Performance via Parallelism

다수의 데이터로부터 data 를 stripe 함. striping 이란 데이터를 나누어 담는 것.

- Bit-level striping

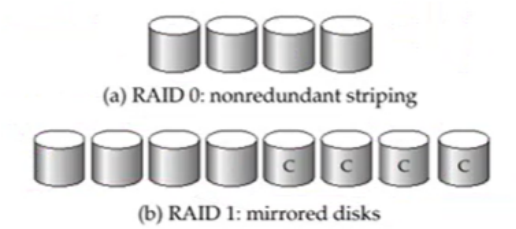
각각의 바이트의 비트들을 여러 개의 디스크에 나눠서 저장함.

만약 8 개의 디스크가 있으면 bit i 를 각각의 디스크 i 에 저장하면 됨.

- Block-level striping

n 개의 디스크에, 파일의 i 번째 block 이 디스크로 들어감.

RAID Levels



RAID Level 0: Block stripping; non-redundant. 데이터 손실이 치명적이지 않은 high performance 시스템에서 사용.

RAID Level 1: Mirrored Disk with block striping. 레벨 0 의 방법과 같은데 그걸 통째로 mirroring 하는 것. Write 퍼포먼스는 최고. DB 의 로그 파일을 저장하는 등의 용도로 많이 씀.

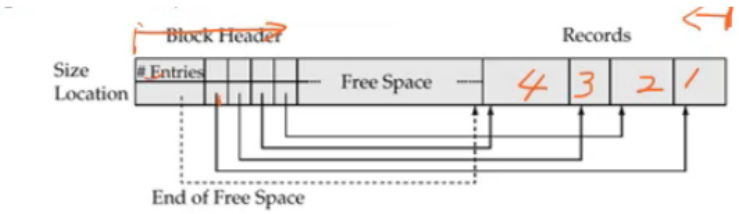
다른 책에서는 Level 1 을 striping 을 생각하지 않기도 함. 하지만 우리는 이 책을 따름.

RAID Level 5: Block-interleaved distributed parity

레벨 4 는 parity block 을 하나의 disk 에 저장하는 건데, 얘의 단점을 보완한 것. Parity 를 디스크 여러 개에 나눠서 저장.

레벨 1 과 비교하면 storage overhead 가 적고, write 오버헤드가 큼. Reliability 도 조금 떨어짐.

Slotted Page Structure



Variable-length record 는 Byte-string 으로 나타냄.

Record 가 block 보다 작다고 가정.

Page header 는 이런 것들을 담음.

- record entry 의 개수
- 해당 block 의 free space 의 끝
- 각 record 의 location 와 size.

11 장

Basic Concepts

- Search Key

파일에서 레코드들을 찾기 위한 attribute 들.

- Index file

Records 로 이루어져 있음. (index entry 라고 불린다)

Ordered Indices

- Primary Index

Search key 가 파일의 순서로 특정된 index.

Clustering index 라고도 불림.

Primary index 의 search key 는 항상은 아니지만 보통 primary key 임.

- Secondary Index

Search key 가 file 의 순서와 다르게 특정되어 있는 index

Non-clustering index 라고도 불림.

포인터가 더욱 중요한 역할을 함.

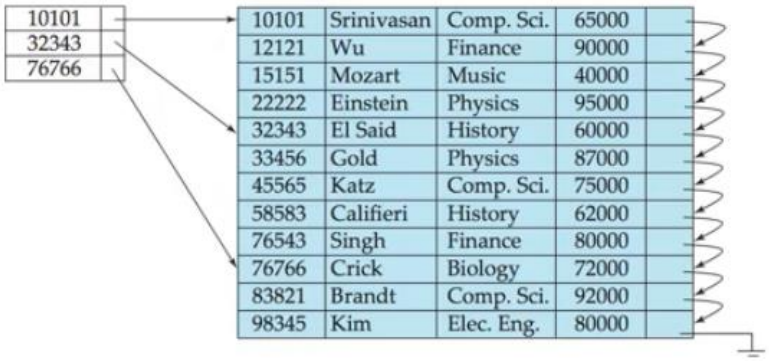
Dense Index Files

- Dense Index

모든 index 가 search-key value 에 나타남. 항상 primary index 는 아님!

- Sparse Index

일부 search-key value 들만 index record 를 담고 있음.

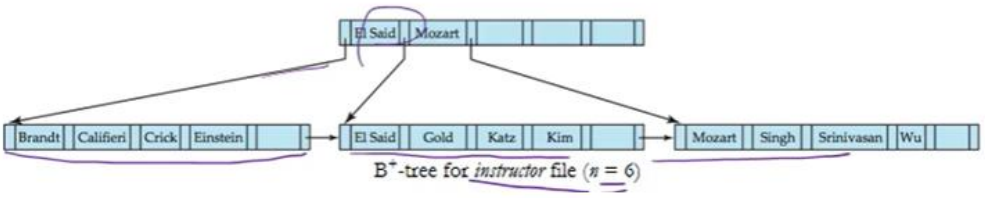


record 들이 search-key 의 순서대로 저장되어 있을 경우에 사용 가능.

Insert/delete 에서 공간과 유지보수 overhead 가 적음.

보통 record 를 찾는 데 dense index 보다 느림.

Example of B+ tree



N = 5 일 경우에는 (포인터 개수)

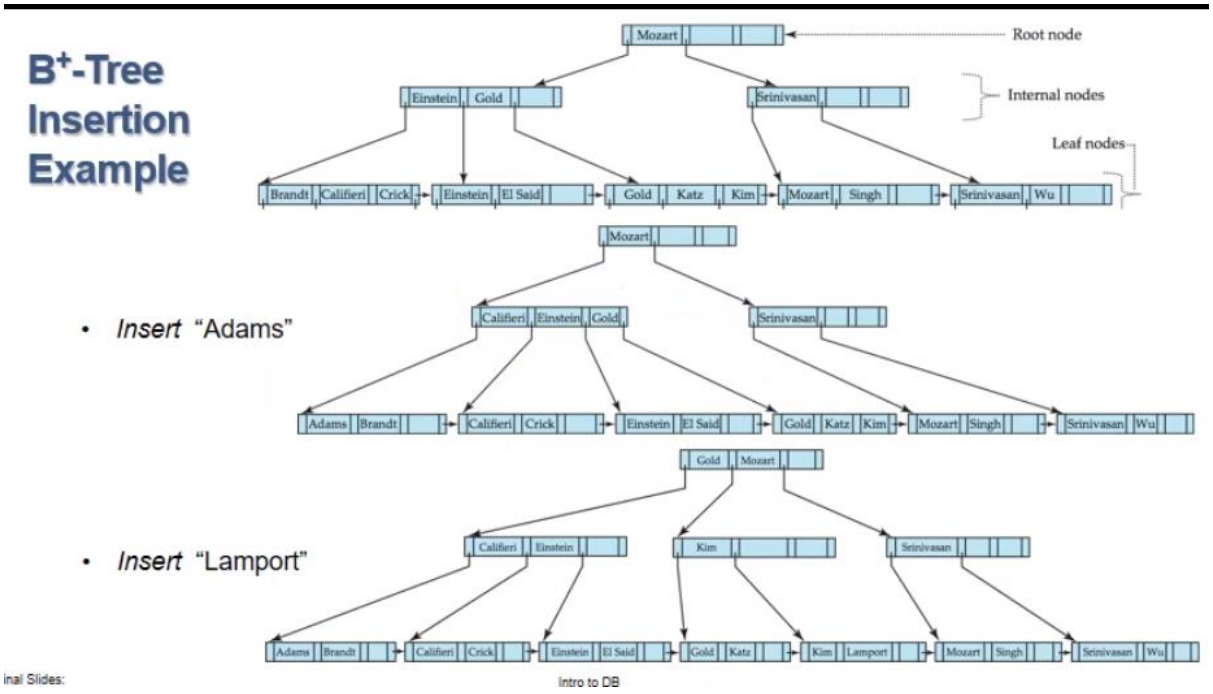
루트가 아닌 모든 node 들은

2 개 ~ 4 개의 key value 를 가져야 함. (3~5 개의 pointer)

루트가 아닌 Non-leaf node 들은

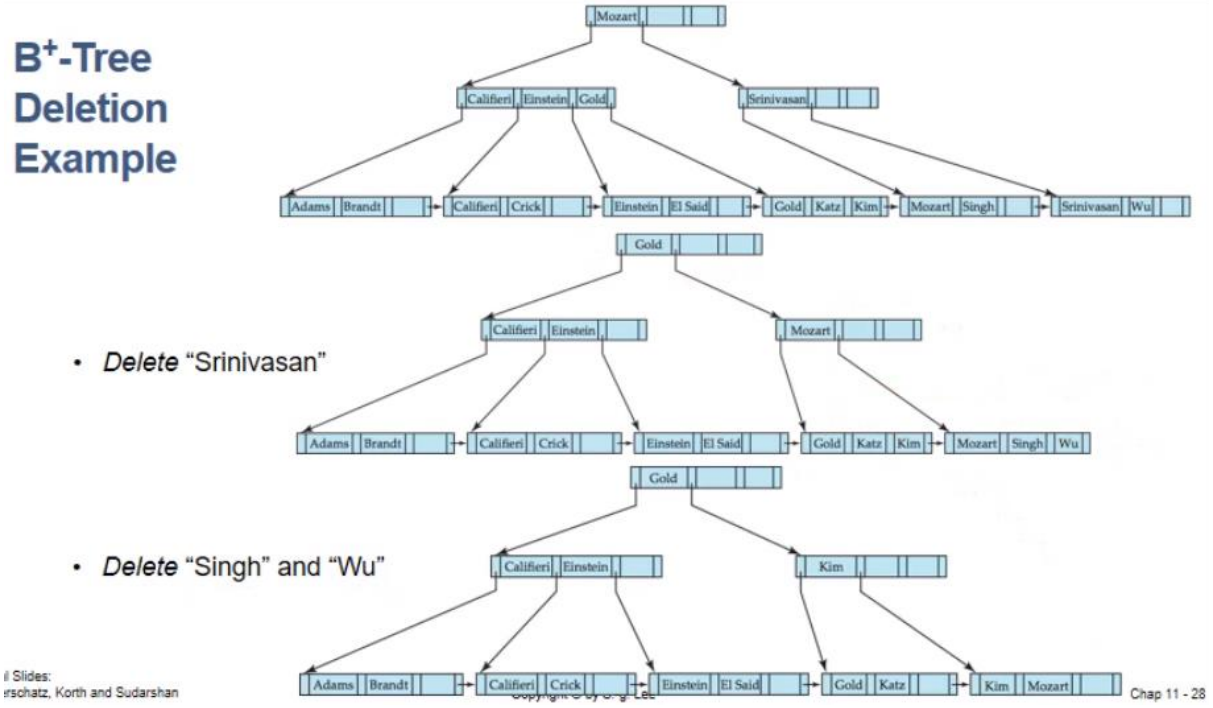
3~5 개의 children 을 가져야 함 (포인터와 같은 의미)

Root 는 최소 2 개의 children 을 가져야 함.



inal Slides:

Intro to DB



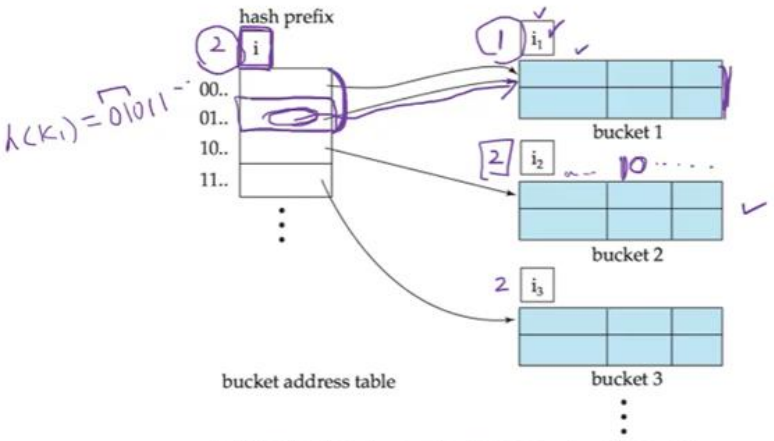
l Slides:
rschatz, Korth and Sudarshan

Chap 11 - 28

Hash Functions

Ideal hash function 은 uniform (각 bucket 에 저장되어 있는 search-key value 의 수가 같아야 함), random (file 의 search-key value 의 실제 분포와 상관 없이 각 bucket 이 같은 수의 레코드를 가짐)해야함.

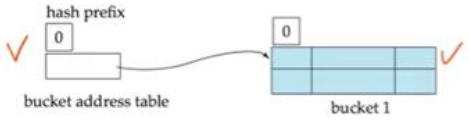
General Extendable Hash Structure



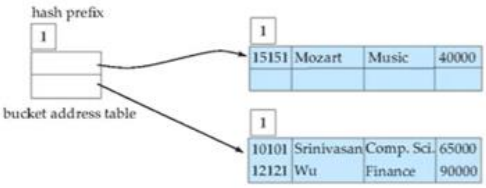
In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Extendable Hashing: Example

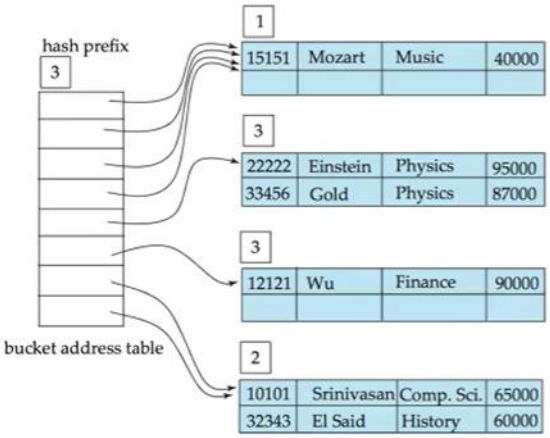
- Initial Hash structure



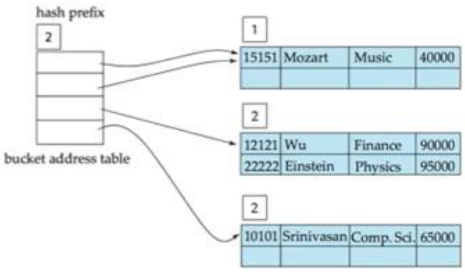
- Insert "Mozart", "Srinivasan", and "Wu"



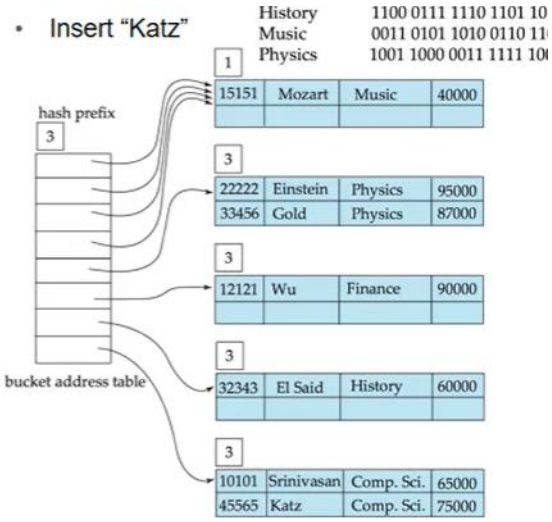
- Insert "Gold" and "El Said"



- Insert "Einstein"

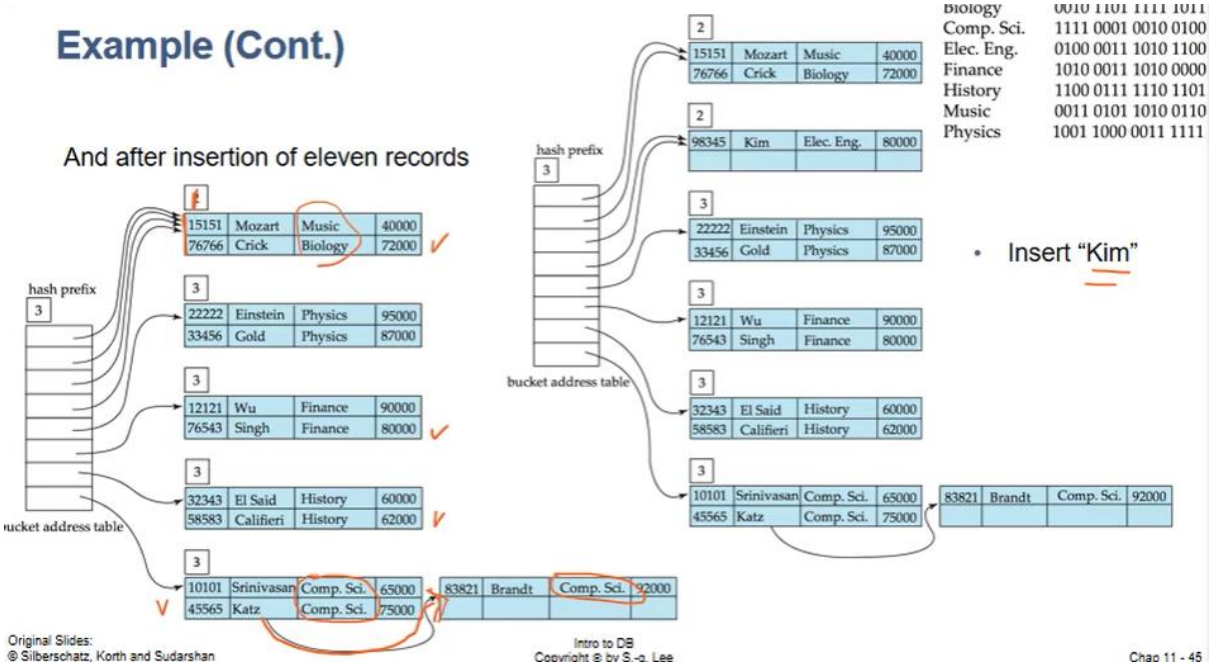


- Insert "Katz"



Example (Cont.)

And after insertion of eleven records



- Insert "Kim"

Extendable Hashing vs Other Schemes.

Extendable hashing 의 장점:

- hash performance 는 파일이 커지면서 줄어들지 않음.
- minimal space overhead.

단점:

- record 를 찾기 위해 indirection 의 레벨이 늘어남.
- bucket address table 자체는 엄청 커질 수도 있음. (메모리보다 더!)
    - 디스크에도 매우 큰 연속된 공간을 할당할 수는 없음.
- bucket address table 의 크기를 바꾸는 건 비싼 연산.
    - 따라서 너무 큰 테이블에는 적용하지 않음.

12 장

Query Plan

- Evaluation Primitive

relational algebra 하나하나가 어떻게 evaluate 될 것인지와 같이 묶여져 있는 것.

$\sigma_{balance>2500}(account)$: use index 1
$\sigma_{balance>2500}(account)$: use table scan

- Query evaluation plan

Query 를 evaluate 하는 데 쓸 수 있는 Primitive operation 들의 연속.

- Generation of Evaluation Plan

1. equivalent 한 expression 들을 생성함.
    생성하기 위해 equivalence rule(13 장에 있는데 우린 안 배움)을 이용.
2. query plan 을 얻기 위해 결과 expression 을 annotating 함.
3. 예측된 cost 에 따라 가장 싼 plan 을 고름.

전체 process 는 **cost based optimization** 이라고 함.

Measures of Query Cost

Cost 는 주로 query 를 **응답하는 데 소요된 전체 시간**을 측정함.

많은 요인들이 time cost 에 기여함: disk access, CPU, network communication (이건 분산 환경에서)

Disk access 가 주로 가장 지배적인 cost 임 (일반적인 centralized system 에서). 또한 상대적으로 측정하기 쉽다.

- number of block transfers from disk

  t_T : time to transfer one block: 10~40 ms

- number of seeks

  t_S : time for one seek: 8~20 ms (disk seek + rotational delay)

따라서 b 개의 block transfer 와 s 번의 seek 가 있었다면 비용은

$$b \times t_T + s \times t_S$$

우리는 CPU cost 는 무시함.

결과를 disk 에 쓰는 cost 도 무시. -〉 operation 의 결과는 disk 에 쓰지 않고 parent operation 으로부터 받을 수도 있음.

Selection Operations

- A1: linear search

  Physical address 의 순서로 file block 을 스캔해서 첫 record 부터 마지막 record 까지 봄.

  예측 cost (number of disk blocks scanned)는 $b_r * t_T + 1 * t_S$. Seek 가 1 인 이유는 인접한 record 들은 같은 곳에 있다고 가정.

  b_r: # of blocks containing records from relation r

  Key 에서 찾으면 평균 cost 는 반으로 줄어듦. $\text{cost} = (b_r/2) * t_T + 1 * t_S$

  Best 면 cost 는 1 * t_T + 1 * t_s

  Linear search 는 selection condition, file 안에서의 record 의 순서, index 의 이용 가능성 등에 상관없이 사용할 수 있음.

- A2 (primary index, equality on key)

주어진 equality condition 을 만족하는 하나의 record 를 찾음.

Cost = $(h_i + 1) * (t_T + t_S)$. block 들이 모두 random access 라서 seek time 발생.

Tree 의 높이 + data 가 들어있는 block access 해서 h + 1

- A3 (primary index, equality on non-key)

여러 개의 record 를 찾음.

Record 는 연속된 block 에 있음.

B 를 matching 되는 record 를 담고 있는 block 의 수라고 하면,

Cost 는 $h_i * (t_T + t_S) + t_S + t_T * b$

- A4 (secondary index, equality)

Search-key 가 candidate key 이면 하나의 record 를 찾음.

이 경우 Cost 는 $= (h_i + 1) * (t_T + t_S)$

Search-key 가 non-candidate key 이면 여러 개의 record 를 찾음.

N 개의 matching record 가 서로 다른 block 에 있을 수도 있음!

Cost 는 $(h_i + n) * (t_T + t_S)$. 트리의 높이 + n 개의 각각의 block.

* n 이 커지면 매우 비싸질 수도 있음! N 이 전체 record 의 10% 정도 이상이 되면 full file scan 이 더 효율적일 수도 있음.

- A5 (primary index, comparison). (Relation 은 A 에 대해 sort 되어 있음)

- For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
- For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple > v; do not use index

A 가 V 보다 작은 경우에는 index 를 쓰지 않는다는 점!

- A6 (secondary index, comparison)

- For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
- For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$

Secondary index 이기 때문에 leaf node 는 정렬되어 있지만 record 들은 뒤죽박죽임.

따라서 위의 두 가지 경우 모두 다, record 를 탐색하는 것은

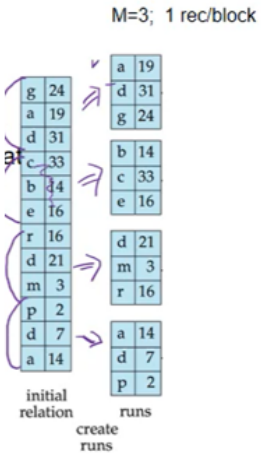
1. 각각의 record 가 I/O 를 필요로 하는지
2. Linear file scan 이 더 싼지

에 중점이 맞춰짐.

External-Sort Merge

M 을 memory size 라고 하자. (in pages)

1. sorted 된 run 들을 만듦.



Relation 의 끝까지 요걸 반복함

(a) M 개의 block 을 memory 로 읽어옴.
(b) In-memory block 들을 정렬함.
(c) Run R_i 에 정렬된 data 를 쓰고 i 를 늘림.

i 의 최종 값을 N 이라고 하자.

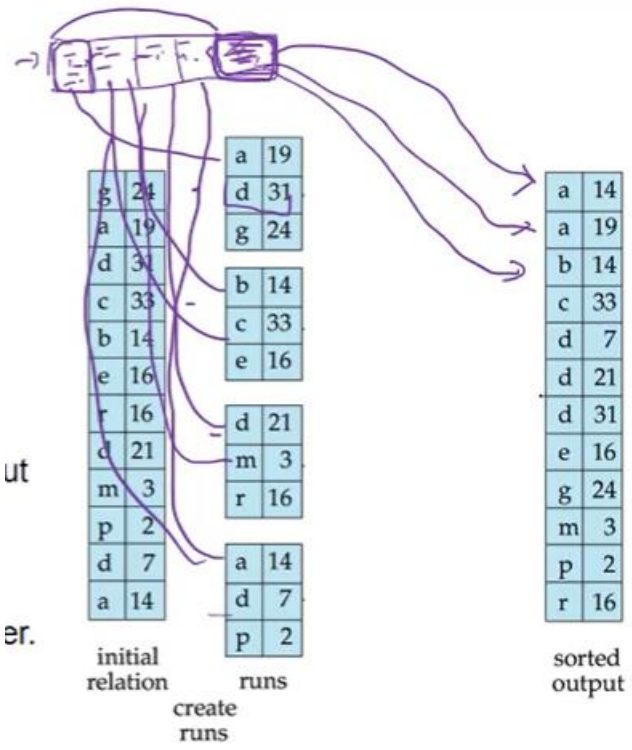
2. run 들을 Merge 함. (N-way merge). (if N < M)

N 개의 buffer block 을 input run 을 위해 사용, output 에 1 buffer block. 총 N+1 개.

각 run 의 첫 번째 block 을 buffer page 로 읽음.

그리고 다음을 모든 input buffer page 가 empty 가 될 때까지 반복함

1. 모든 buffer page 의 첫 번째 record (정렬된 순서)를 고름.
2. Record 를 output buffer 에 씀. Output buffer 가 가득 차면 그걸 disk 에 씀.
3. Record 를 input buffer page 에서 삭제. 만약 buffer page 가 empty 이면 next block 이 있을 경우 buffer 로 읽어들임.



n 〈 m 인 경우 가능. 즉 run 의 개수보다 메모리가 더 클 경우임.

2. run 들을 merge 함 (N 〉= M 인 경우)

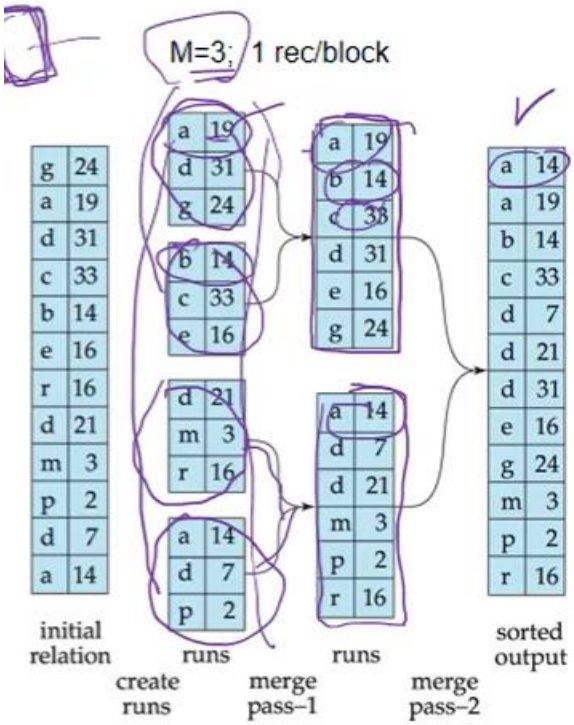
만약 N 〉= M 인 경우, 여러 번의 merge pass 들이 필요함.

각각의 pass 에서, 연속된 M-1 개의 run 들의 group 이 merge 됨.

Pass 는 run 들의 개수를 M-1 의 factor 로 줄임. 그리고 같은 factor 에 의해 더 긴 run 들을 만듦.

예를 들어 M = 11 이고 90 개의 run 이 있으면, 하나의 pass 는 run 의 개수를 9 로 감소시킴. 그리고 각각은 초기의 run 보다 10 배의 사이즈가 더 큼.

모든 run 이 하나로 합쳐질 때까지 반복된 pass 가 수행됨.



n 〉= m 인 경우. 즉 run 의 개수가 메모리보다 같거나 큼.

External Sort-Merge - Cost Analysis

Merge pass 의 전체 개수: $\lceil \log_{M-1}(b_r/M) \rceil$. 원래 block 이 b_r 개 있고 M 개씩 나눴으니 leaf node 의 개수가 b_r/M 이고, 이게 M-1 만큼 계속 줄어듦.

- Block transfers

각 merge pass 마다 2*b_r 만큼의 block transfer. 왜냐면 모든 block 이 한 번 읽고 한 번 씀.

따라서 external sorting 의 총 block transfer 개수는: $b_r (2\lceil \log_{M-1}(b_r / M) \rceil + 1)$. 마지막 write cost 는 세지 않음. Merge pass 의 개수 * 2 에 전체 block 의 수를 곱한 뒤, 마지막 꺼를 안 세서 - b_r 인데 처음에 run 만들때 2*b_r 이 들기 때문에 더해서 +1*b_r 이 됨.

- Seeks

Run generation: read, write 에 각각 1 번의 seek. $2\lceil b_r / M \rceil$

Merge phase 에서는 각 run 당 buffer size 가 b_b (read/write b_b blocks at a time) , 각각의 merge pass 당 $2\lceil b_r / b_b \rceil$. 여기도 마지막은 포함하지 않음.

Seek 의 전체 개수는: $2\lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2\lceil \log_{M-1}(b_r / M) \rceil - 1)$

Nested-Loop Join

Theta join 을 계산하기 위해.

To compute the theta join $r \bowtie_\theta s$

```
for each tuple t_r in r do
    for each tuple t_s in s do
        if pair (t_r, t_s) satisfy the join condition θ
            add t_r • t_s to the result.
```

- Worst Case

there is memory only to hold one block of each relation

$n_r * b_s + b_r$ block transfers + $n_r + b_r$ seeks

- Best Case

the smaller relation fits entirely in memory: use it as the inner relation

$b_r + b_s$ transfers + 2 seeks

- Example

with *student* as outer relation:

- 5000 * 400 + 100 = 2,000,100 block transfers,
- 5000 + 100 = 5100 seeks

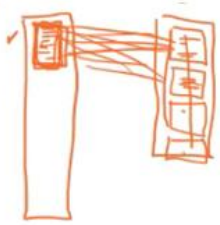
with *takes* as the outer relation

- 10000 * 100 + 400 = 1,000,400 block transfers and 10,400 seeks

If smaller relation (*student*) fits entirely in memory

- 100 + 400 = 500

Block Nested-Loop Join



- Variant of nested-loop join
  - every block of inner relation is paired with every block of outer relation

```
for each block B_r of r do
for each block B_s of s do
   for each tuple t_r in B_r do
    for each tuple t_s in B_s do
         if (t_r, t_s) satisfy the join condition
                then add t_r • t_s to the result
```

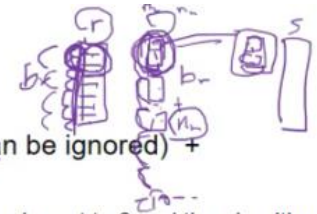
- Worst case estimate

  - Each block in the inner relation *s* is read once for each *block* in the outer relation
  - $b_r * b_s + b_r$ block transfers + $2 * b_r$ seeks

- Best Case

  - $b_r + b_s$ block transfers + 2 seeks.

Cost of Hash-Join



- Cost of hash join (without recursive partitioning)
  - $3(b_r + b_s) + 4 * n_h$ block transfers ($4n_h$ can be ignored) +
  - $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
  - If the entire build input can be kept in main memory, *n* can be set to 0 and the algorithm does not partition the relations into temporary files. Cost estimate goes down to $b_r + b_s$.

Seek 식이 잘못됨! 위의 식에 더해서 n_h 만큼의 seek 가 발생. 이게 r, s 에서 발생하므로 2*n_h 가 해당 식에 더해짐.

Example:

- *student* ⋈ *takes*
- memory size: 20 blocks; b_{stud}= 100 and b_{takes} = 400.
- *student* is build input
- Partition it into 5 partitions, each of size less than 20 blocks
- Similarly, partition *takes* into 5 partitions each of size about 80
- Therefore total cost:
  - 3(100 + 400) = 1500 block transfers
  - 2(⌈100/3⌉ + ⌈400/3⌉) = 336 seeks

Evaluation of Expressions

지금까지 우리는 각각의 operation 을 위한 algorithm 을 봄.

그러면 전체의 expression tree 는 어떻게 계산할 것인가?

Materialization

- generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.

Pipelining

- pass on tuples to parent operations even as an operation is being executed

- Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk

14 장

Properties of a Transaction

ACID properties

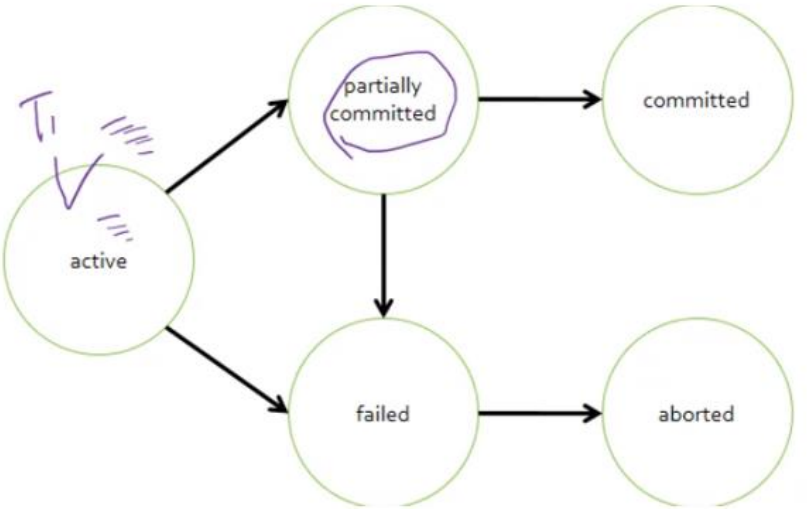
Atomicity: all or nothing

Consistency (Correctness) : consistent 상태에서 다른 consistent 상태로 전이되는 것.

Isolation: 각 트랜잭션은 interfered 되면 안 됨. Concurrency. 방해받지 않음.

Durability: 트랜잭션이 끝난다면 그 효과는 durable & public 해야 함.

Transactions States



- **Active**: the initial state, transaction stays in this state while it is executing
- **Partially committed**: the final statement has been executed
- **Failed**: normal execution can no longer proceed
- **Aborted**: transaction has been rolled back and the database restored to its state prior to the start of the transaction.
  - Two options after it has been aborted:
    - restart the transaction – only if no internal logical error
    - kill the transaction
- **Committed**: after *successful completion*.

Schedules

트랜잭션의 simplified view

트랜잭션이 local buffer 에서 read 와 write 를 하며 데이터에 대한 임의의 연산을 수행한다고 가정.

Read 와 write 를 제외한 다른 연산들은 무시. - 오직 read, write 연산으로만 simplified schedule 이 이루어짐.

Serializability

기본 가정 - 각각의 트랜잭션은 데이터베이스의 consistency 를 유지해야 함.

즉, 트랜잭션들의 serial execution 은 데이터베이스 consistency 를 유지함.

여러 개가 동시에 실행되어도 이렇게 serial 의 효과를 보장해야 한다는 뜻.

Concurrent schedule 은 serial 과 같으면, serializable 하다.

S1 은 모든 DB 의 가능한 인스턴스에서 S2 의 실행과 같은 DB 상태 결과가 되면 동치이다.

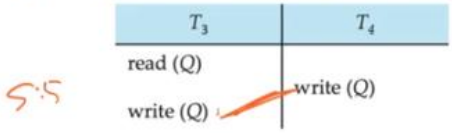
Conflict Serializability

- Instructions I_i and I_j of transactions T_i and T_j respectively, conflict if and only if
  - there exists some item Q accessed by both I_i and I_j, and
  - at least one of these instructions is a write(Q)
    1. I_i = read(Q), I_j = read(Q). not conflict.
    2. I_i = read(Q), I_j = write(Q). conflict.
    3. I_i = write(Q), I_j = read(Q). conflict
    4. I_i = write(Q), I_j = write(Q). conflict

같은 아이템 Q 에 대해서 트랜잭션 두개가 모두 접근하고, 둘 중 하나가 write(Q)를 가지고 있으면 conflict. 둘 다 read 일 경우는 conflict 아님.

직관적으로, conflict 는 둘 사이의 temporal order 를 강요함.

- A schedule S is *conflict serializable* if it is conflict equivalent to a serial schedule
- Example of a schedule that is **not** conflict serializable:

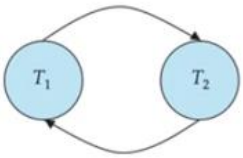


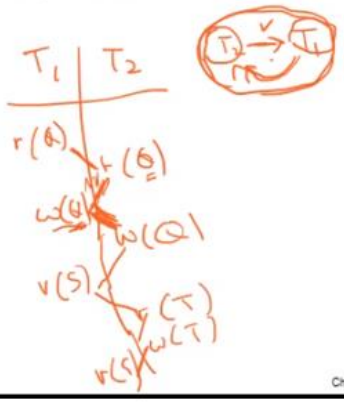
Conflict serializable 이란 non-conflict 한 스케쥴에서 순서를 바꿔서 같은 스케쥴이 된다면 conflict serializable 이라고 함.

이 때 한 트랜잭션의 순서를 바꾸는 것은 아님. 두 트랜잭션 사이의 순서를 바꾸는 것.

Testing for Serializability

- **Precedence graph** — a direct graph where the vertices are the transactions
  - draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
  - We may label the arc by the item that was accessed.
- Example





al Slides:
erschatz, Korth and Sudarshan
Intro to DB
Copyright © by S.-g. Lee
Chap 14 - 2

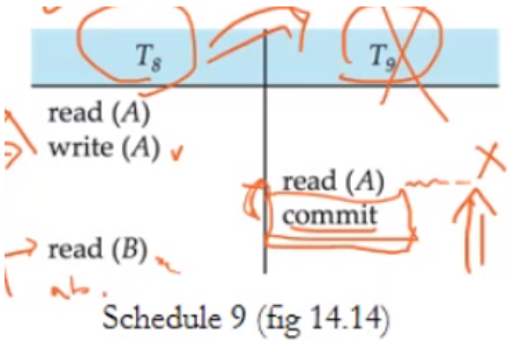
Test for Conflict Serializability

Conflict serializable 하려면 해당 스케쥴에 cycle 이 없어야 함(acyclic).

Cycle 탐지는 n^2. (혹은 n+e 가능)

만약 precedence graph 가 사이클이 없으면, topological sorting 을 통해서 serial 한 스케쥴을 얻을 수 있음.

Recoverability



Schedule 9 (fig 14.14)

스케쥴 9 는 recoverable 하지 않음. T8 에서 문제가 생기면 T9 도 롤백해야하는데 이미 커밋이 되어있기 때문.

스케쥴이 Recoverable 하려면 T_j 가 아이템을 읽었는데 T_i 가 그 전에 해당 아이템을 write 한 경우, T_i 의 커밋이 T_j 보다 앞에 있어야 함.

Cascadeless Schedules

Cascading rollback 은 다음 경우 일어날 수 없음.

- Cascading rollbacks cannot occur if
  - for each pair of transactions T_i and T_j
  - such that T_j reads a data item previously written by T_i,
  - the commit of T_i appears before the read of T_j

T_j 는 T_i 가 커밋할 때까지 기다리는 것.

Dirty read 를 하지 않는 것 (transaction 에서 commit 되지 않은 write 는 읽지 않는다.

모든 cascadeless 스케쥴은 또한 복구 가능하다.

15 장

Lock-Based Protocols

Lock 은 concurrent access 를 제어하기 위한 메커니즘.

두 가지 모드가 있음.

1. *exclusive (X) mode*: both read and write (lock-X instruction)
2. *shared (S) mode*: only read (lock-S instruction)

Lock request 는 concurrency-control manager 에 의해 만들어짐.

트랜잭션은 request 가 승인된 후에만 진행될 수 있음.

Granting of Locks

Lock-compatibility matrix

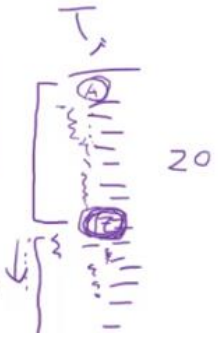
	S	X
S	true	false
X	false	false

T1 이 shared 일 때 T2 가 shared 를 요청할 때만 가능.

Two-Phase Locking Protocol (2PL)

아래의 규칙들이 모든 트랜잭션에 대해 lock 을 요청하고 풀어줄 때 사용됨.

- 2PL
  - Phase 1: Growing Phase
    - transaction may obtain locks
      - can acquire a **lock-S** or **lock-X** on item
      - can convert a **lock-S** to a **lock-X** (**upgrade**)
    - transaction may not release locks
  - Phase 2: Shrinking Phase
    - transaction may release locks
      - can release a **lock-S** or **lock-X**
      - can convert a **lock-X** to a **lock-S** (**downgrade**)
    - transaction may not obtain locks



Strict / Rigorous 2PL

Strict 2PL

Exclusive lock 은 commit/abort 할 때까지 release 하지 않고 가지고 있음.

Cascading rollback 을 피할 수 있음. (dirty read 를 방지하기 때문)

Rigorous 2PL (많이 사용)

모든 lock 을 commit/abort 까지 release 하지 않고 hold 함.

트랜잭션들은 커밋된 순서로 serialize 될 수 있음.