**- Chapter 12: Query Processing**

**Basic Steps in Query Processing**

1. Parsing and translation

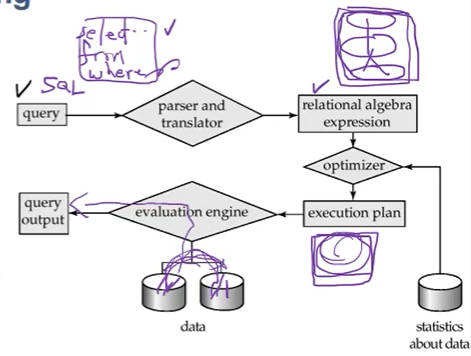
Query를 internal form으로 변환. (ex: relational algebra)

1. Optimization

Query를 계산하는 방법은 하나 이상 있기 때문에 그 중 선택. 데이터의 다양한 statistics를 이용.

1. Evaluation

Query evaluation plan을 받아서, plan을 실행하고 결과를 반환



**Query Plan**

- Evaluation Primitive

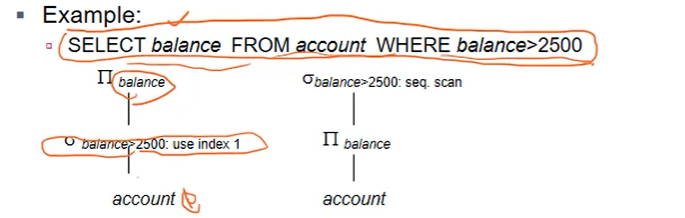
relational algebra 하나하나가 어떻게 evaluate될 것인지와 같이 묶여져 있는 것.



- Query evaluation plan

Query를 evaluate하는 데 쓸 수 있는 Primitive operation들의 연속.

Example

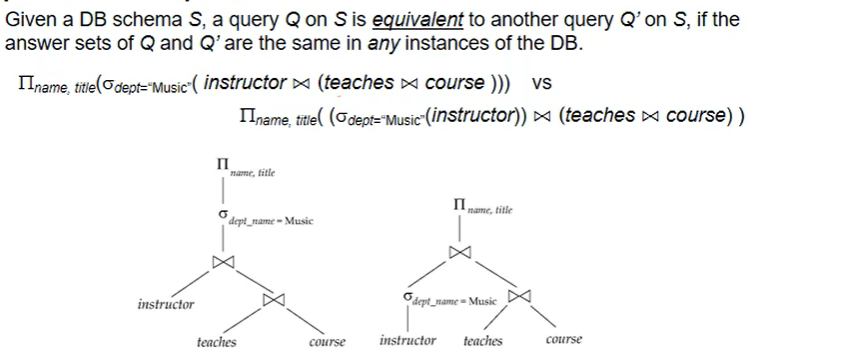


이렇게 순서를 다르게 하는 두 가지 plan이 있을 수 있음.

**Query Optimization**

효율적인 query plan을 고르는 것. Query를 evaluate하는 방법은 하나 이상이 있음.

- Equivalence of Expressions



두 expression이 모든 instance에서 항상 같은 값을 보장한다면 둘은 equivalent함.

Query Optimization은 주어진 query의 evaluation plan 중 가장 효율적인 것을 고르는 과정임.

- Generation of Evaluation Plan

1. equivalent한 expression들을 생성함.

생성하기 위해 equivalence rule(13장에 있는데 우린 안 배움)을 이용.

2. query plan을 얻기 위해 결과 expression을 annotating 함.

3. 예측된 cost에 따라 가장 싼 plan을 고름.

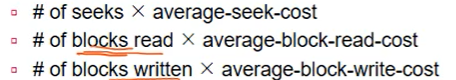
전체 process는 **cost based optimization**이라고 함.

**Measures of Query Cost**

Cost는 주로 query를 **응답하는 데 소요된 전체 시간**을 측정함.

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

Disk access가 주로 가장 지배적인 cost임 (일반적인 centralized system에서). 또한 상대적으로 측정하기 쉽다.



상용 시스템에선 simplicity를 위해서 이런 것들만 사용함.

- number of block transfers from disk



- number of seeks



따라서 b개의 block transfer와 s번의 seek가 있었다면 비용은



우리는 CPU cost는 무시함.

결과를 disk에 쓰는 cost도 무시. -> operation의 결과는 disk에 쓰지 않고 parent operation으로부터 받을 수도 있음.

**Selection Operation**

- File Scan

Record를 하나하나 읽어서 condition을 만족하는 것을 찾음.

Full file scan: 파일의 모든 record를 탐색. (하나의relation이 하나의 file에 저장되어 있으면 file == relation. 여러 개가 있어도 비슷)

- A1: linear search

Physical address의 순서로 file block을 스캔해서 첫 record부터 마지막 record까지 봄.

예측 cost (number of disk blocks scanned)는 . Seek가 1인 이유는 인접한 record들은 같은 곳에 있다고 가정.



Key에서 찾으면 평균 cost는 반으로 줄어듦. 

Best면 cost는 1 \* t\_T + 1 \* t\_s

Linear search는 selection condition, file 안에서의 record의 순서, index의 이용 가능성 등에 상관없이 사용할 수 있음.

**Selections Using Indices**

- Index scan

Index를 이용한 search algorithm.

Selection condition이 search-key index여야 함.

- A2 (primary index, equality on key)

주어진 equality condition을 만족하는 하나의 record를 찾음.

Cost = . block들이 모두 random access라서 seek time 발생.

Tree의 높이 + data가 들어있는 block access 해서 h + 1

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

여러 개의 record를 찾음.

Record는 연속된 block에 있음.

B를 matching되는 record를 담고 있는 block의 수라고 하면,

Cost는 

- A4 (secondary index, equality)

Search-key가 candidate key이면 하나의 record를 찾음.

이 경우 Cost는 

Search-key가 non-candidate key이면 여러 개의 record를 찾음.

N 개의 matching record가 서로 다른 block에 있을 수도 있음!

Cost는 . 트리의 높이 + n개의 각각의 block.

\* n이 커지면 매우 비싸질 수도 있음! N이 전체 record의 10% 정도 이상이 되면 full file scan이 더 효율적일 수도 있음.

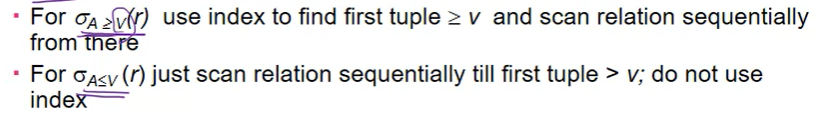
**Selections Involving Comparisons**

아래 리스트는  요런 form에 대한 selection을 수행할 수 있음:

Linear file scan

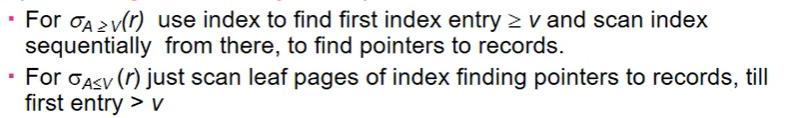
아니면 다음 방식으로 index를 사용

- A5 (primary index, comparison). (Relation은 A에 대해 sort되어 있음)



A가 V보다 작은 경우에는 index를 쓰지 않는다는 점!

- A6 (secondary index, comparison)



Secondary index이기 때문에 leaf node는 정렬되어 있지만 record들은 뒤죽박죽임.

따라서 위의 두 가지 경우 모두 다, record를 탐색하는 것은

1. 각각의record가 I/O를 필요로 하는지
2. Linear file scan이 더 싼지

에 중점이 맞춰짐.

**External Sort-Merge**

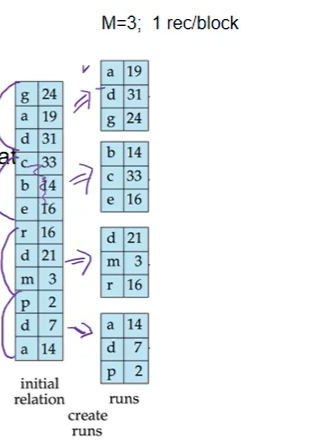
Sorting은 DB에서 중요함: order by, group by, join …

하나의 옵션: index를 만든다. 그리고 그 인덱스를 사용해서 정렬된 순서로 relation을 읽는다. 잘못되면 하나의 tuple 당 하나의 disk block access가 필요할 수도 있기 때문에 비효율적.

External Sort-Merge - Disk에서 sort하는 좋은 방법.

M을 memory size라고 하자. (in pages)

1. sorted된 **run**들을 만듦.

 Relation의 끝까지 요걸 반복함

1. M개의 block을 memory로 읽어옴.
2. In-memory block들을 정렬함.
3. Run R\_i에 정렬된 data를 쓰고 i를 늘림.

i의 최종 값을 N이라고 하자.

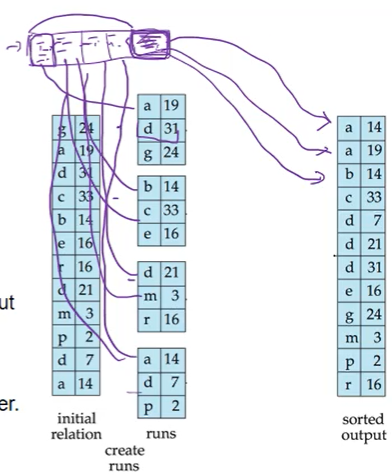
2. run들을 Merge함. (N-way merge). (if N < M)

N개의 buffer block을 input run을 위해 사용, output에 1 buffer block. 총 N+1개.

각 run의 첫 번째 block을buffer page로 읽음.

그리고 다음을 모든 input buffer page가 empty가 될 때까지 반복함

1. 모든 buffer page의 첫 번째 record (정렬된 순서)를 고름.
2. Record를 output buffer에 씀. Output buffer가 가득 차면 그걸 disk에 씀.
3. Record를 input buffer page에서 삭제. 만약 buffer page가 empty이면 next block이 있을 경우 buffer로 읽어들임.



2. run들을 merge함 (N >= M인 경우)

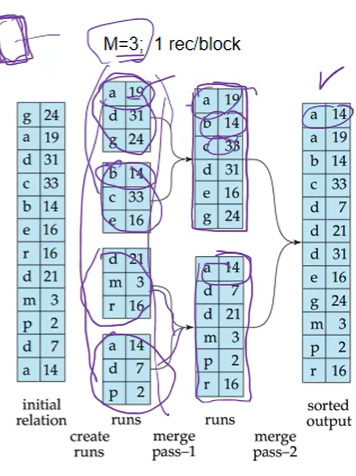
만약 N >= M인 경우, 여러 번의 merge pass들이 필요함.

각각의 pass에서, 연속된 M-1개의 run들의 group이 merge됨.

Pass는 run들의 개수를 M-1의 factor로 줄임. 그리고 같은 factor에 의해 더 긴 run들을 만듦.

예를 들어 M = 11이고 90개의 run이 있으면, 하나의 pass는 run의 개수를 9로 감소시킴. 그리고 각각은 초기의 run보다 10배의 사이즈가 더 큼.

모든 run이 하나로 합쳐질 때까지 반복된 pass가 수행됨.



**External Sort-Merge – Cost Analysis**

Merge pass의 전체 개수: . 원래 block이 b\_r개 있고 M개씩 나눴으니 leaf node의 개수가 b\_r/M이고, 이게 M-1만큼 계속 줄어듦.

- Block transfers

각 merge pass마다 2\*b\_r만큼의 block transfer. 왜냐면 모든block이 한 번 읽고 한 번 씀.

따라서 external sorting의 총 block transfer 개수는: . 마지막 write cost는 세지 않음. Merge pass의 개수 \* 2에 전체 block의 수를 곱한 뒤, 마지막 꺼를 안 세서 -b\_r인데 처음에 run 만들때 2\*b\_r이 들기 때문에 더해서 +1\*b\_r이 됨.

- Seeks

Run generation: read, write에 각각 1번의 seek. 

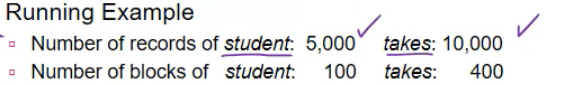
Merge phase에서는 각 run 당 buffer size가 , 각각의 merge pass당 . 여기도 마지막은 포함하지 않음.

Seek의 전체 개수는: 

**Join Operation**

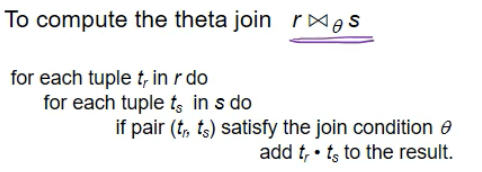
Join을 구현하기 위한 방법은 다양함. 

Cost estimate에 따라 선택함.

우리가 사용할 예시: 

**Nested-Loop Join**

Theta join을 계산하기 위해.



R은 outer relation, s는 inner relation.

Index에 상관 없이 모든 join condition에 수행할 수 있음.

두 관계 사이의 모든 tuple을 살펴보기 때문에 비쌈.

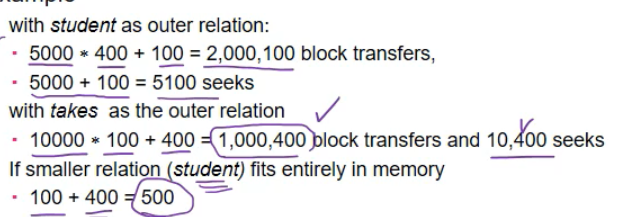
- Worst Case



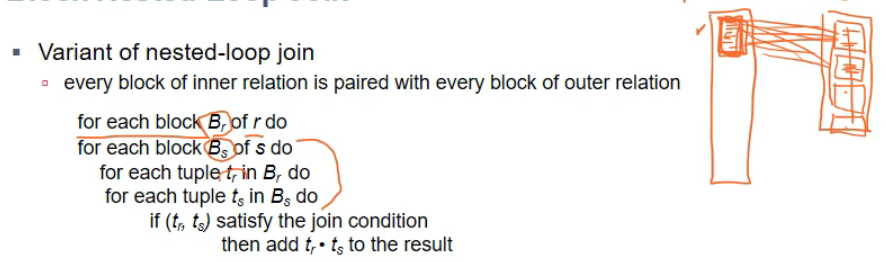
- Best Case



- Example



**Block Nested-Loop Join**



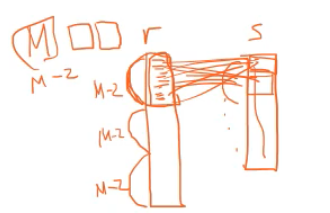
- Worst case estimate

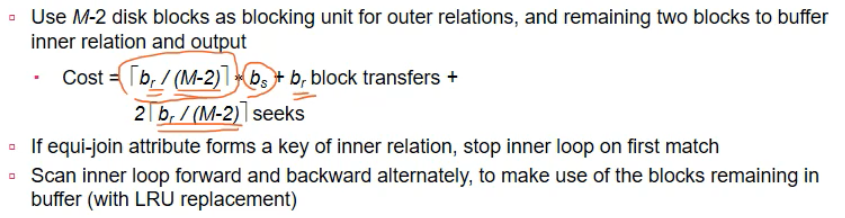


- Best Case

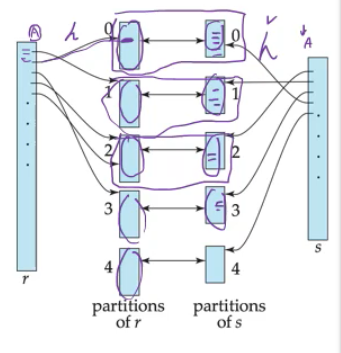


- Improvements



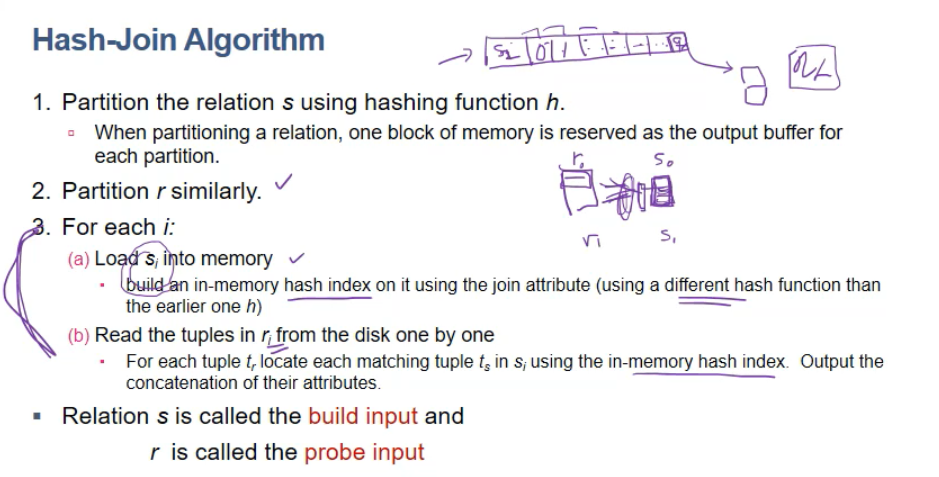


**Hash-Join**

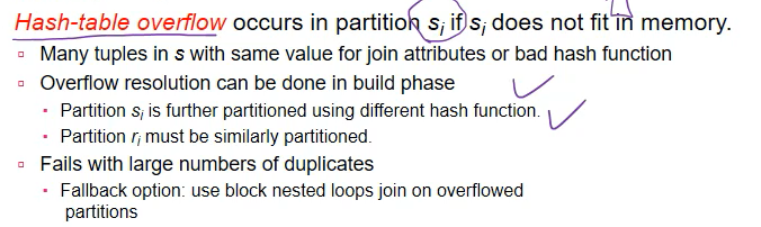
Equi-join과 natural join에 수행 가능. 

Hash function을 두 relation에 동일하게 사용.

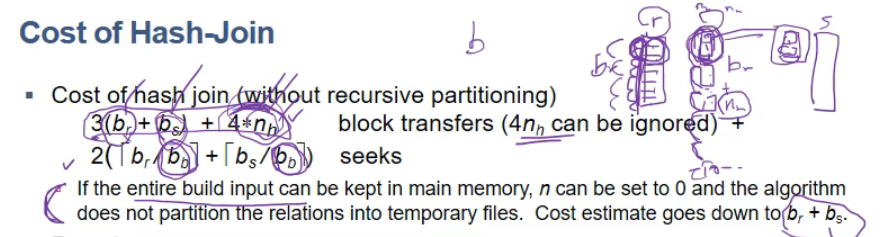
**Hash-Join Algorithm**



Value n과 hash function h은 각각의 s\_i가 memory에 속하도록 구함.

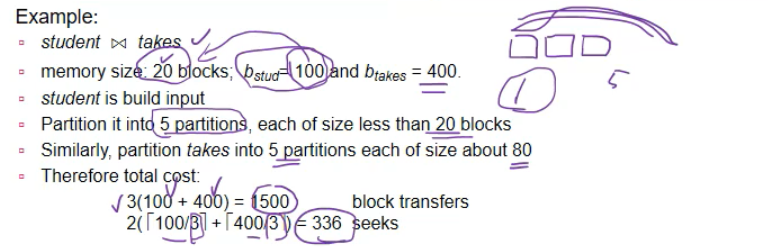


**Cost of Hash-Join**



Seek 식이 잘못됨! 위의 식에 더해서 n\_h만큼의 seek가 발생. 이게 r, s에서 발생하므로 2\*n\_h가 해당 식에 더해짐.

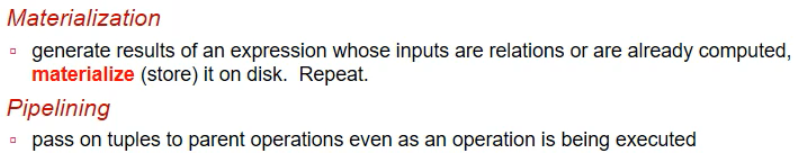
- Example



**Evaluation of Expressions**

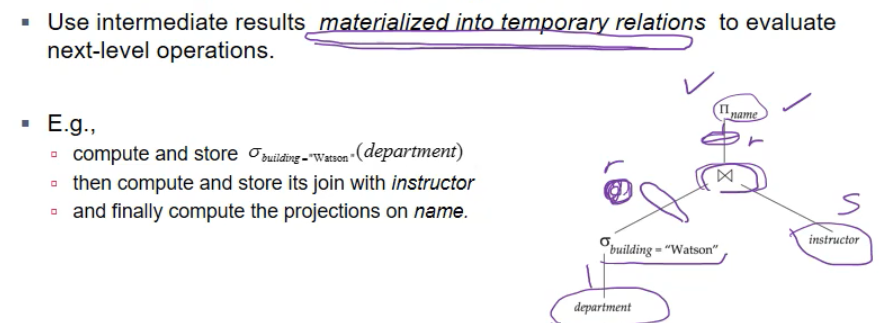
지금까지 우리는 각각의 operation을 위한 algorithm을 봄.

그러면 전체의 expression tree는 어떻게 계산할 것인가?



**Materialization**

아래에서부터 evaluate를 실행해서, 그 결과를 materialize해서 저장.



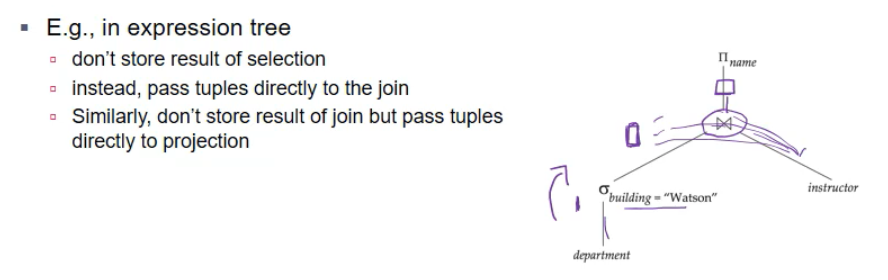
Materialized evaluation은 항상 사용할 수 있음.

Writing disk, reading back 하는 cost가 꽤 클 수 있음.



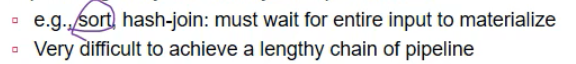
**Pipelining**

여러 개의 several operation을 동시에 계산함.



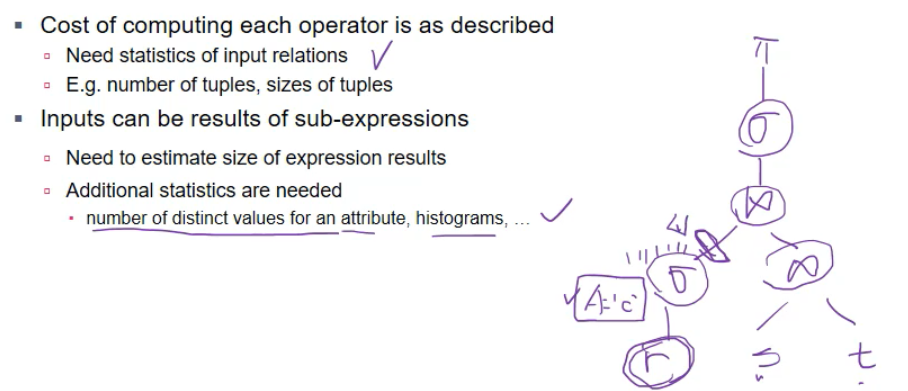
materialization보다 훨씬 쌈.

하지만 항상 가능한 것은 아님.

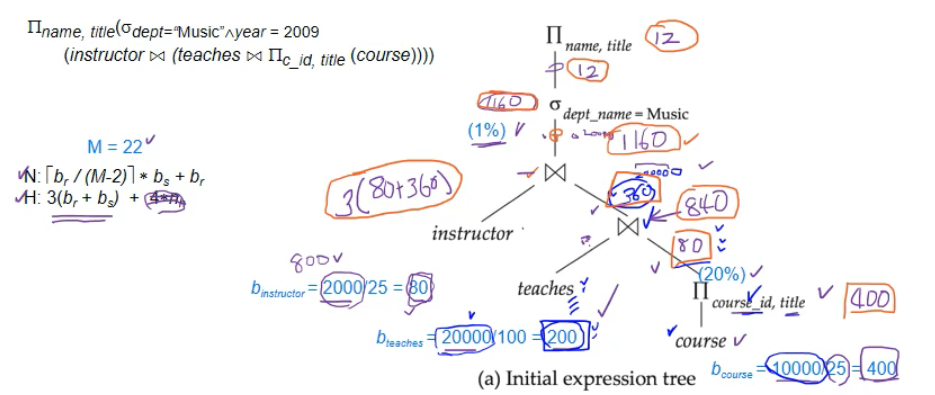


Pipelining이 효과적으로 되려면, 알고리즘, 결과 값의 크기 등을 고려해야 함.

**Cost Estimation of Expressions**

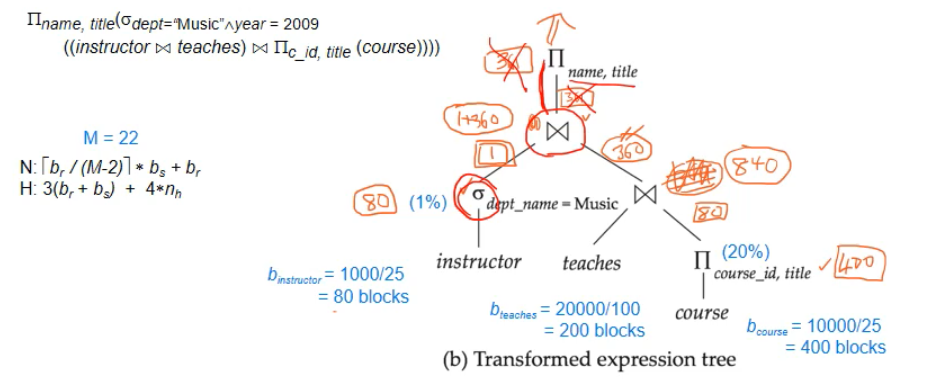


예제 – Materialization 가정.



노란 색으로 표시한 게 block transfer에 드는 비용. 다 더하면 전체 evaluation tree의 비용.

Pipelining을 통해 절약 가능. 예를 들어 join 후 바로 select, select 후 바로 project 등.



같은 식의 순서를 바꾼 것. 순서를 바꾼 것 만으로도 비용이 많이 줄어듦.