Week13_Handout

Database System
Query Processing part1-3

Week13_Handout

Database System_Query Processing part1

Query Processing

How does the query processor execute queries?

Query → Query Plan

- SQL - expression of methods to realize the query

- Overview of Query Execution
- Logical Query Plan - relational algebra optimization
- Physical Query Plan - physical operators

Example of Query Execution

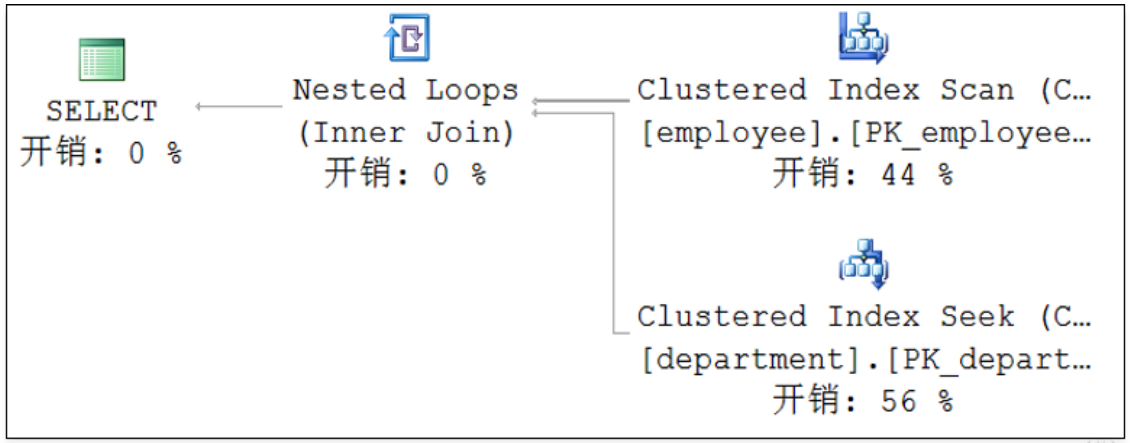
employee

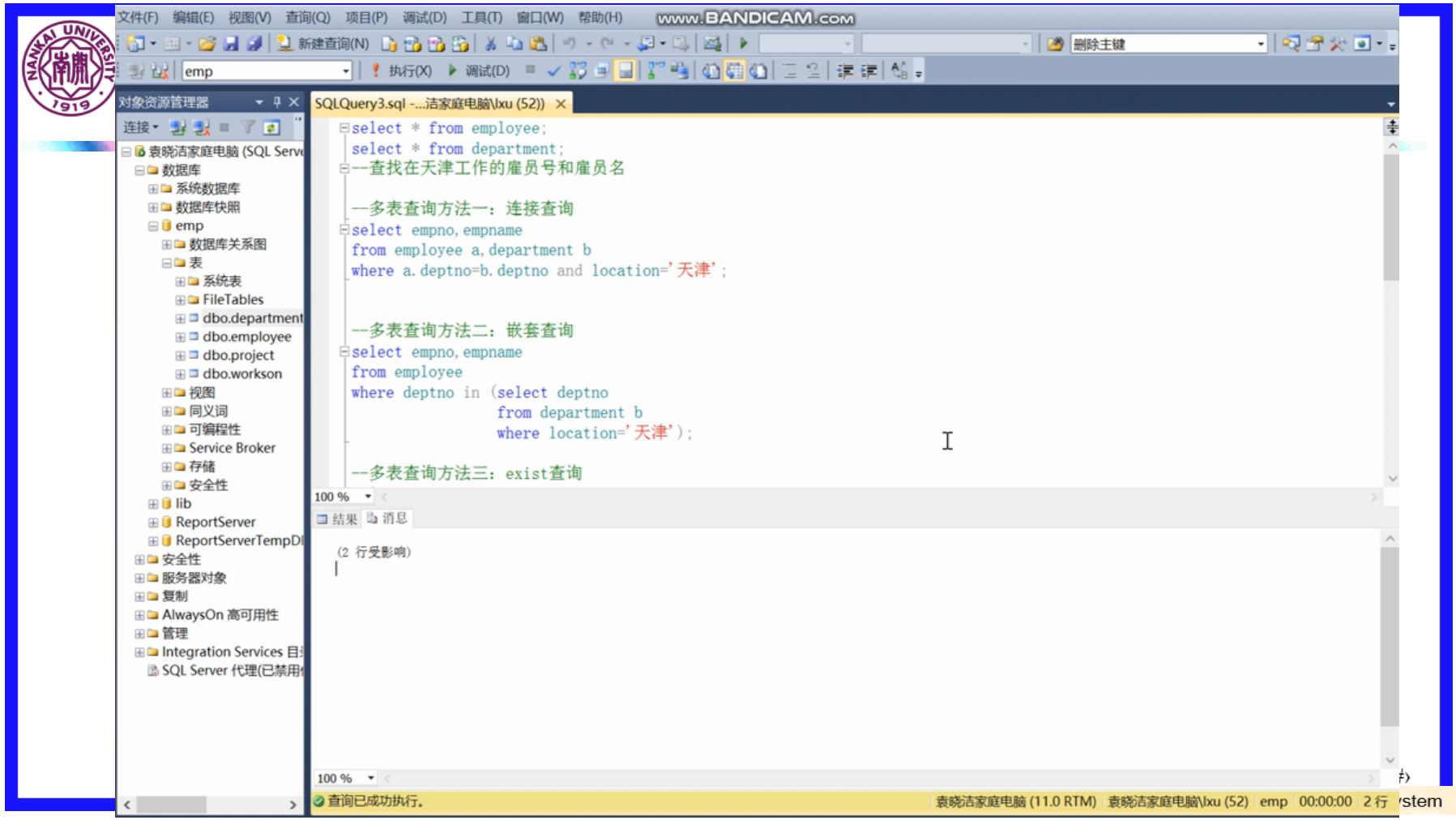
empno	empname	deptno
2581	徐唱	d2
9031	李静	d2
10102	王闻刚	d3
18316	冯新	d1
25348	张风	d3
28559	刘国风	d1
29346	赵东生	d2

department

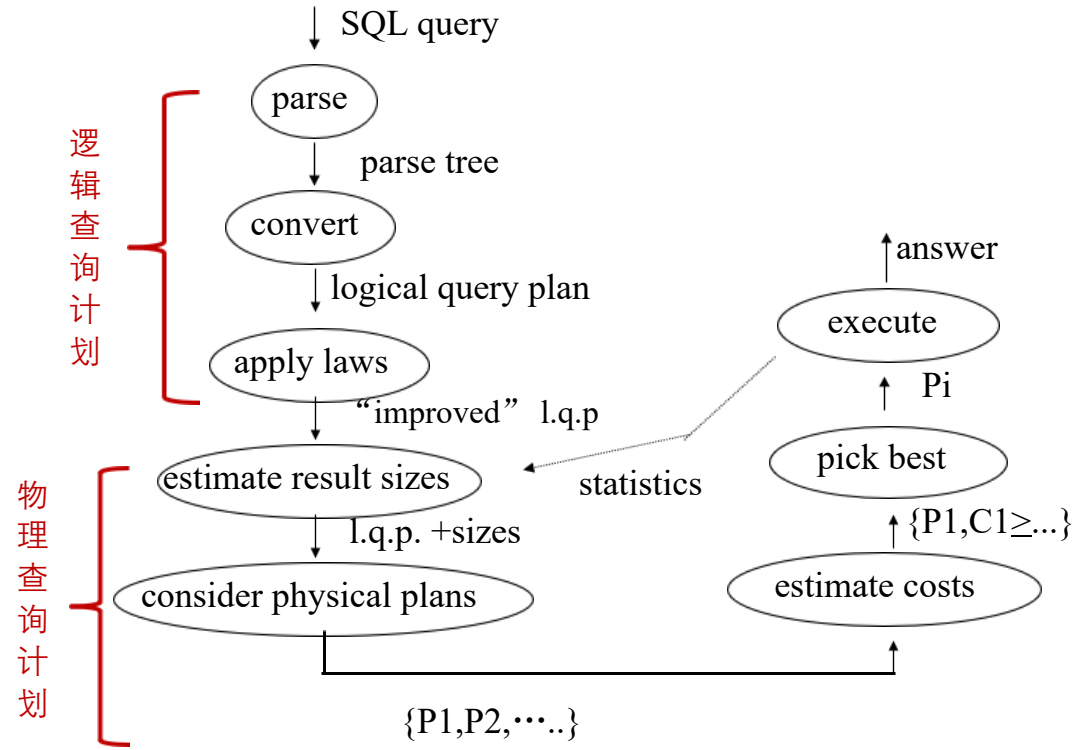
deptno	deptname	location
d1	开发部	天津
d2	财务部	北京
d3	市场部	广东

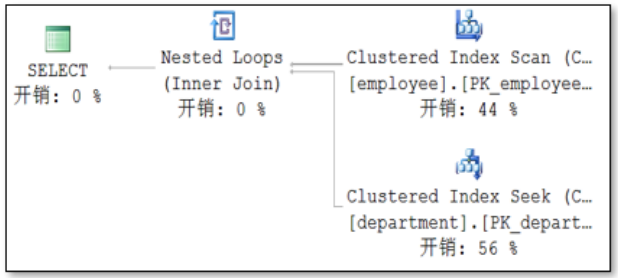
```
select empno,empname
from employee a,department b
where a.deptno=b.deptno and location='天津'
```




文件(F) 编辑(E) 视图(V) 查询(Q) 项目(P) 调试(D) 工具(T) 窗口(W) 帮助(H)
www.BANDICAM.com
新建查询(N)
删除主键
emp
执行(X)
调试(D)
对象资源管理器
连接
袁晓洁家庭电脑 (SQL Serv
数据库
系统数据库
数据库快照
emp
数据库关系图
表
系统表
FileTables
dbo.department
dbo.employee
dbo.project
dbo.workson
视图
同义词
可编程性
Service Broker
存储
安全性
lib
ReportServer
ReportServerTempDI
安全性
服务器对象
复制
AlwaysOn 高可用性
管理
Integration Services 目
SQL Server 代理(已禁用
SQLQuery3.sql -...洁家庭电脑\lxu (52))
select * from employee;
select * from department;
--查找在天津工作的雇员号和雇员名
--多表查询方法一：连接查询
select empno, empname
from employee a, department b
where a.deptno=b.deptno and location='天津';
--多表查询方法二：嵌套查询
select empno, empname
from employee
where deptno in (select deptno
from department b
where location='天津');
--多表查询方法三：exist查询
100 %
结果
消息
(2 行受影响)
100 %
查询已成功执行。
袁晓洁家庭电脑 (11.0 RTM)
袁晓洁家庭电脑\lxu (52)
emp
00:00:00
2 行

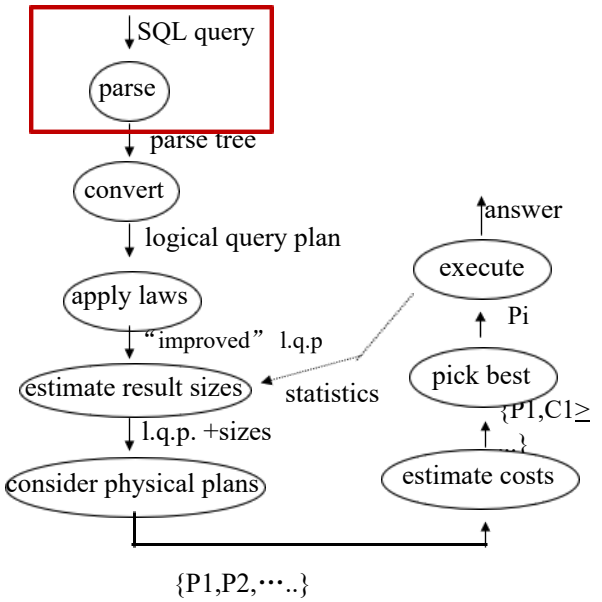
Overview of Query Execution





Step 1: Parsing

- Check syntactic correctness of the query, and transform it into a parse tree
  - Based on the formal grammar for SQL
  - Inner nodes nonterminal symbols (syntactic categories for things like <Query>, <RelName>, or <Condition>)
  - Leaves terminal symbols: names of relations or attributes, keywords (SELECT, FROM, …), operators (+, AND, OR, LIKE, …), operands (10, '%1960', …)
  - Also check semantic correctness: Relations and attributes exist, operands compatible with operators, …



Example: SQL query

Consider querying a movie database with relations

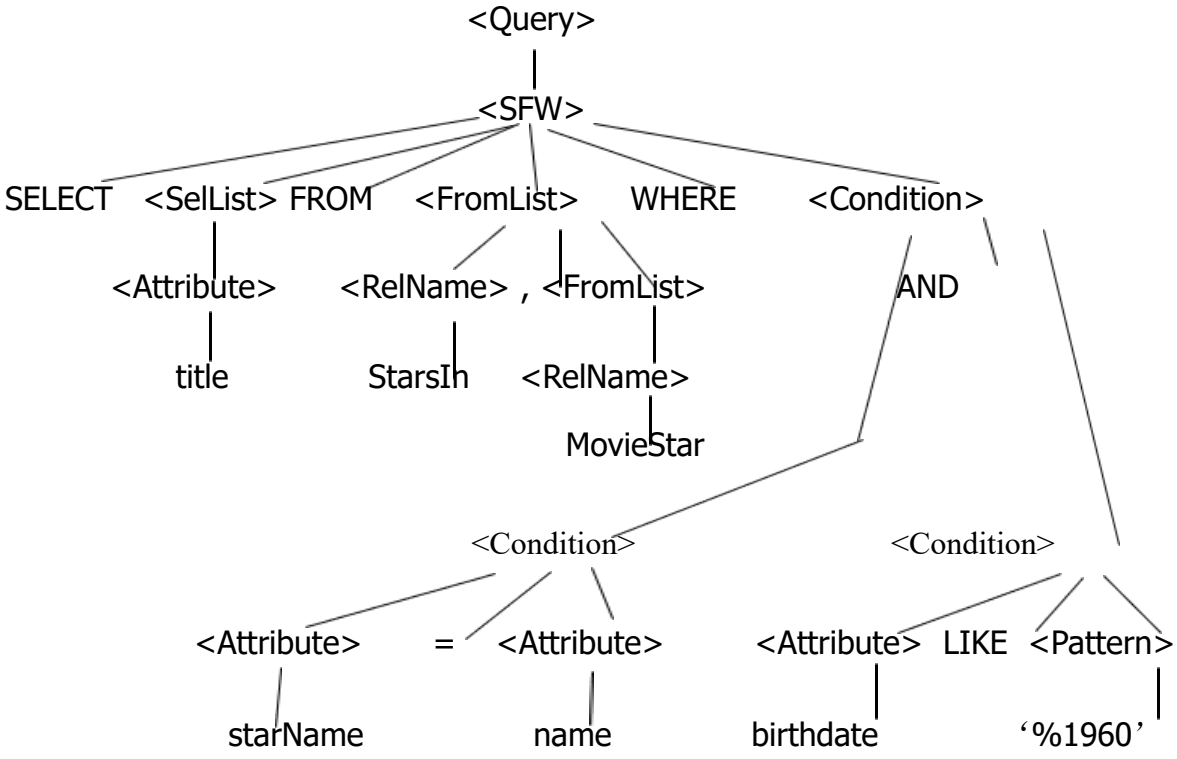
StarsIn(title, year, starName)

MovieStar(name, address, gender, birthdate)

```
SELECT title
FROM        StarsIn, MovieStar
WHERE starName = name AND birthdate LIKE '%1960' ;
```

(Find the titles for movies with stars born in 1960)

Example: Parse Tree



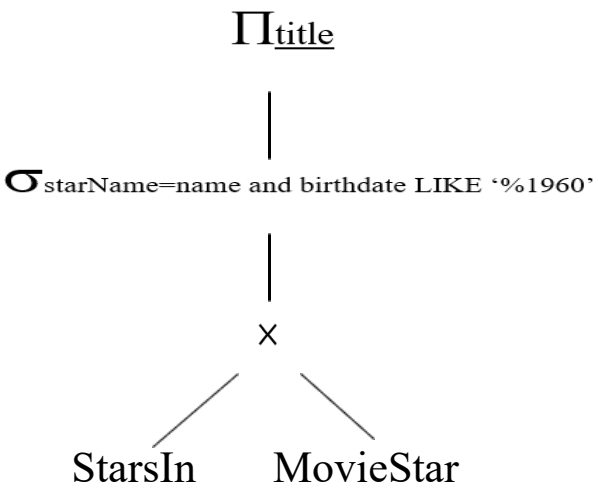
Step 2: Parse Tree –> Logical Query Plan

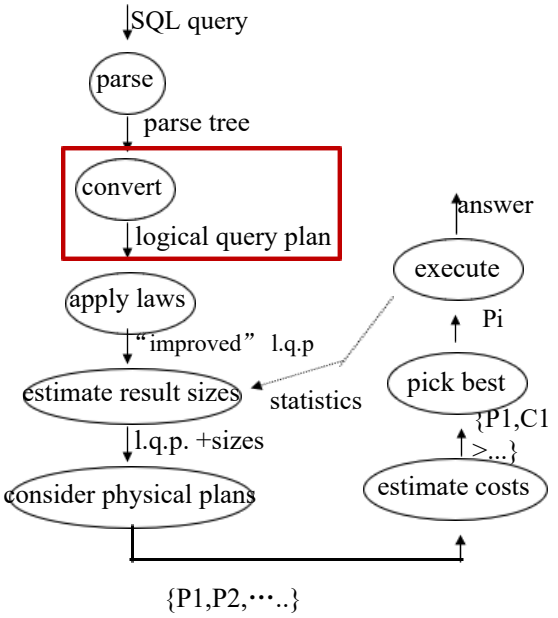
- Basic strategy:

```
SELECT    A, B, C
FROM      R1, R2
WHERE     Cond ;
```

becomes

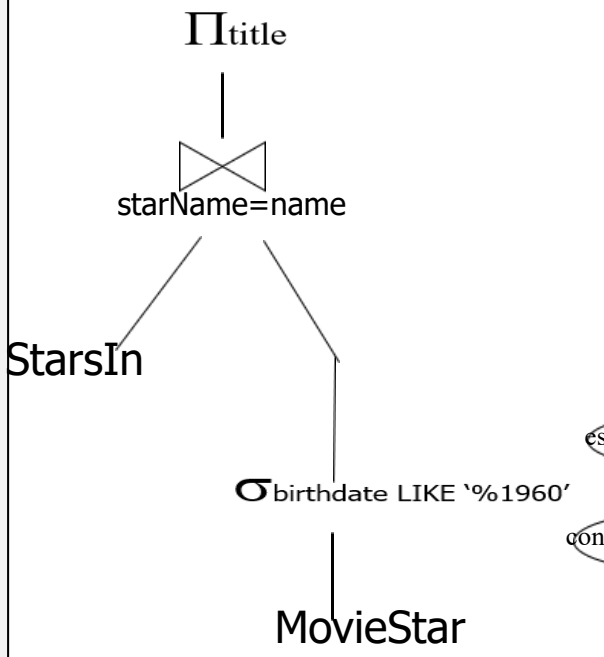
$\pi_{A,B,C}[\sigma_{Cond}(R1 \times R2)]$

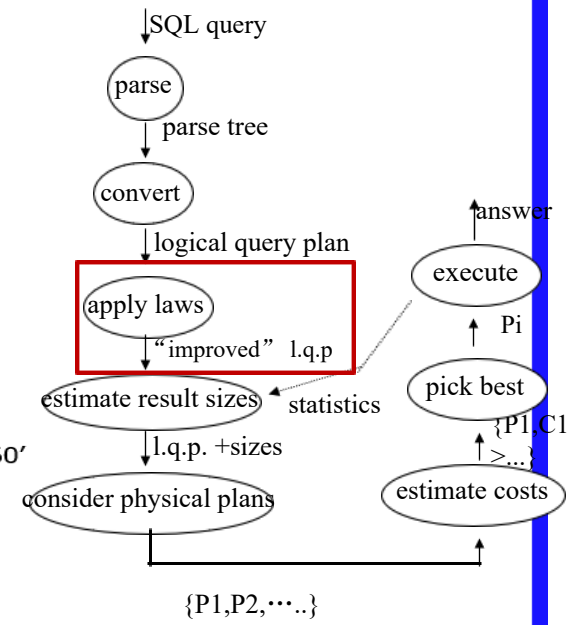




Step 3: Improving the L.Q.P

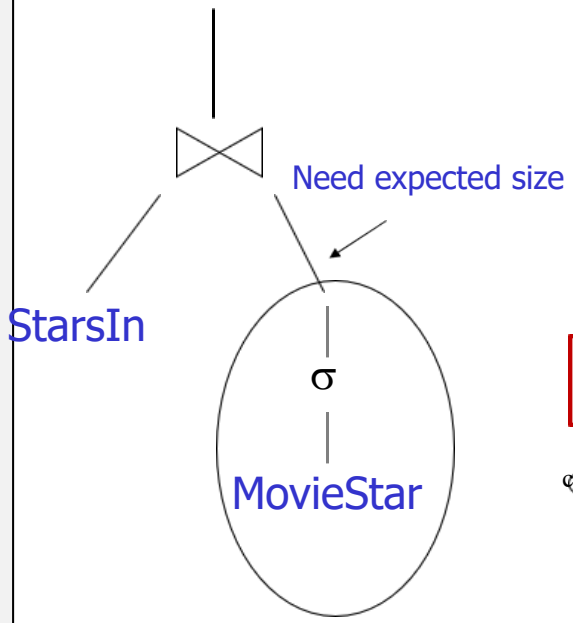
- Transform the logical query plan into an equivalent form expected to lead to better performance
  - Based on laws of relational algebra
  - Normal to produce a single optimized form, which acts as input for the generation of physical query plans

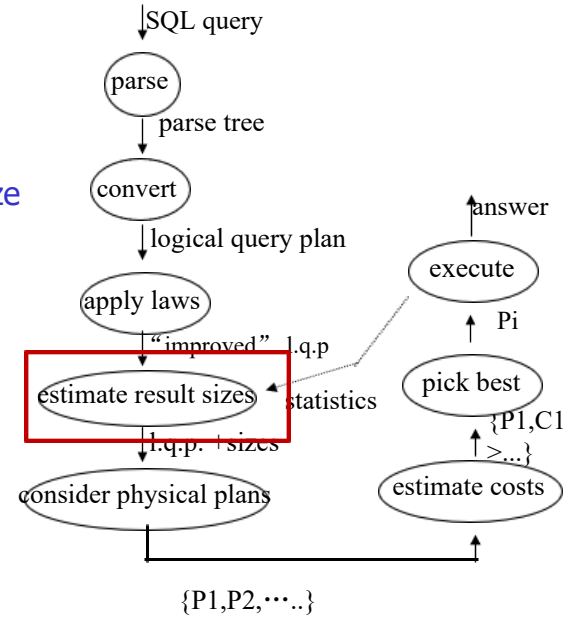




Step 4: Estimate Result Sizes

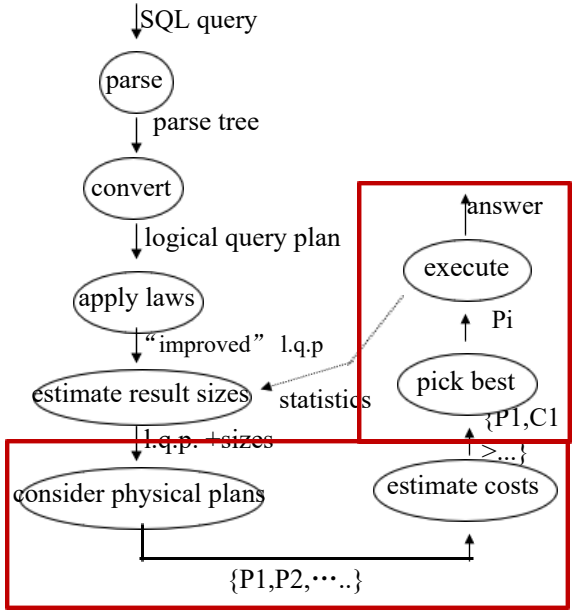
- Cost estimates of database algorithms depend on sizes of input relations
  -> Need to estimate sizes of intermediate results
  - Estimates based on statistics about relations, gathered periodically or incrementally



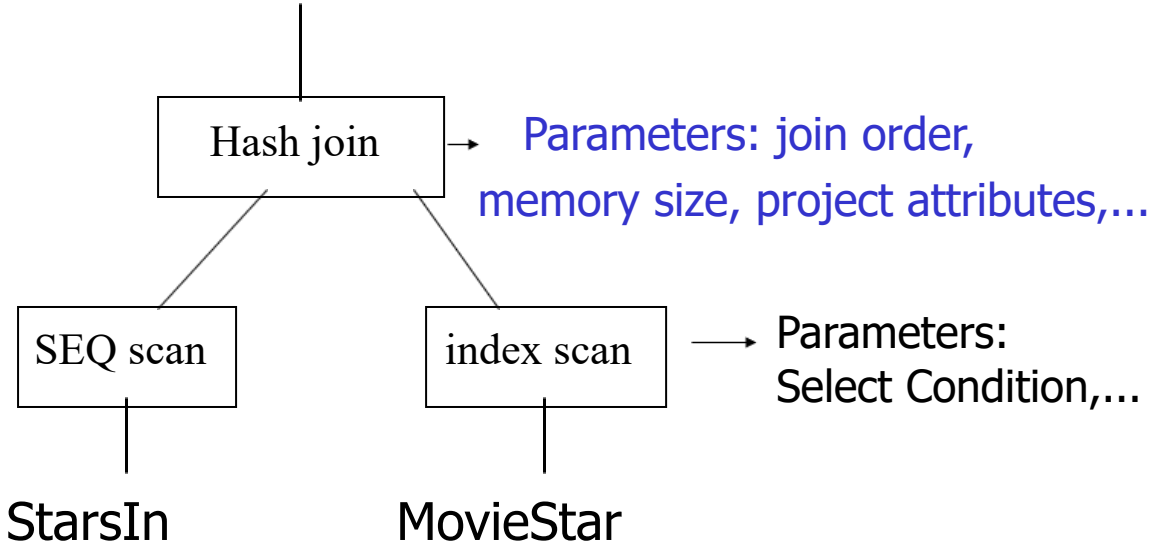


Steps 5, 6, ...

- Generate and compare query plans
  - generate alternate query plans P_1, …, P_k by selecting algorithms and execution orders for relational operations
  - Estimate the cost of each plan
  - Choose the plan P_i estimated to be "best"
- Execute plan P_i and return its result



Example: One Physical Plan



Example

```
SELECT   B, D
FROM     R, S
WHERE    R.C=S.C
  AND    R.A = "c"
  AND    S.E = 2
```

R	A	B	C
	a	1	10
	b	1	20
	c	2	10
	d	2	35
	e	3	45

S	C	D	E
	10	x	2
	20	y	2
	30	z	2
	40	x	1
	50	y	3

Answer

B	D
2	x

Relational Algebra - used to describe logical plans

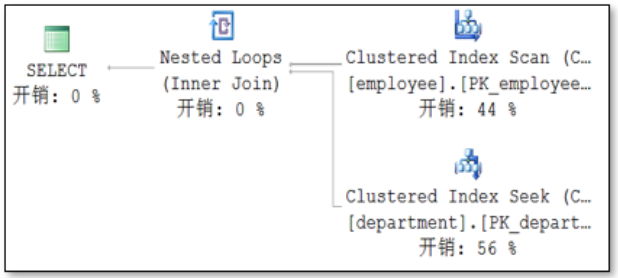
Ex: Original logical query plan

SELECT B,D -> $\Pi_{B,D}$

WHERE ... -> $\sigma_{R.A="c" \wedge S.E=2 \wedge R.C=S.C}$

FROM R,S -> X

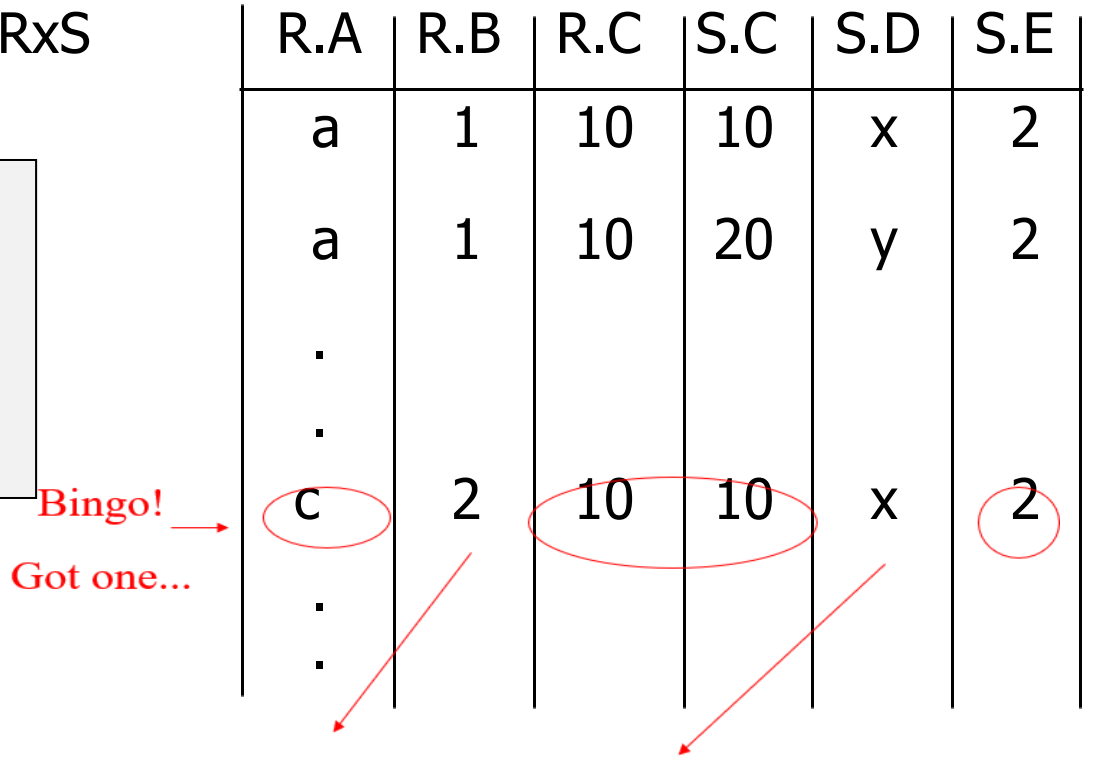
R S



OR: $\Pi_{B,D}[\sigma_{R.A="c" \wedge S.E=2 \wedge R.C=S.C}(R \times S)]$

How to execute query?

- Do Cartesian product RxS
- Select tuples
- Do projection



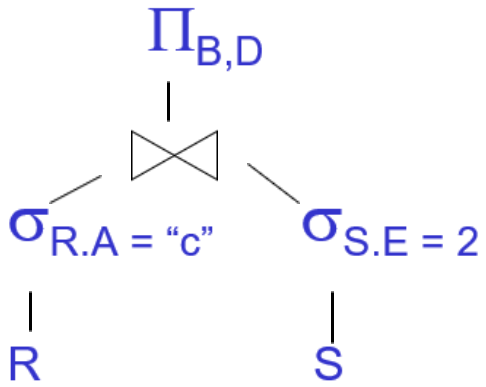
Problem

- A Cartesian product RxS may be LARGE
  - need to create and examine n x m tuples, where n = |R| and m = |S|
  - For example, n = m = 1000 => 10^6 records

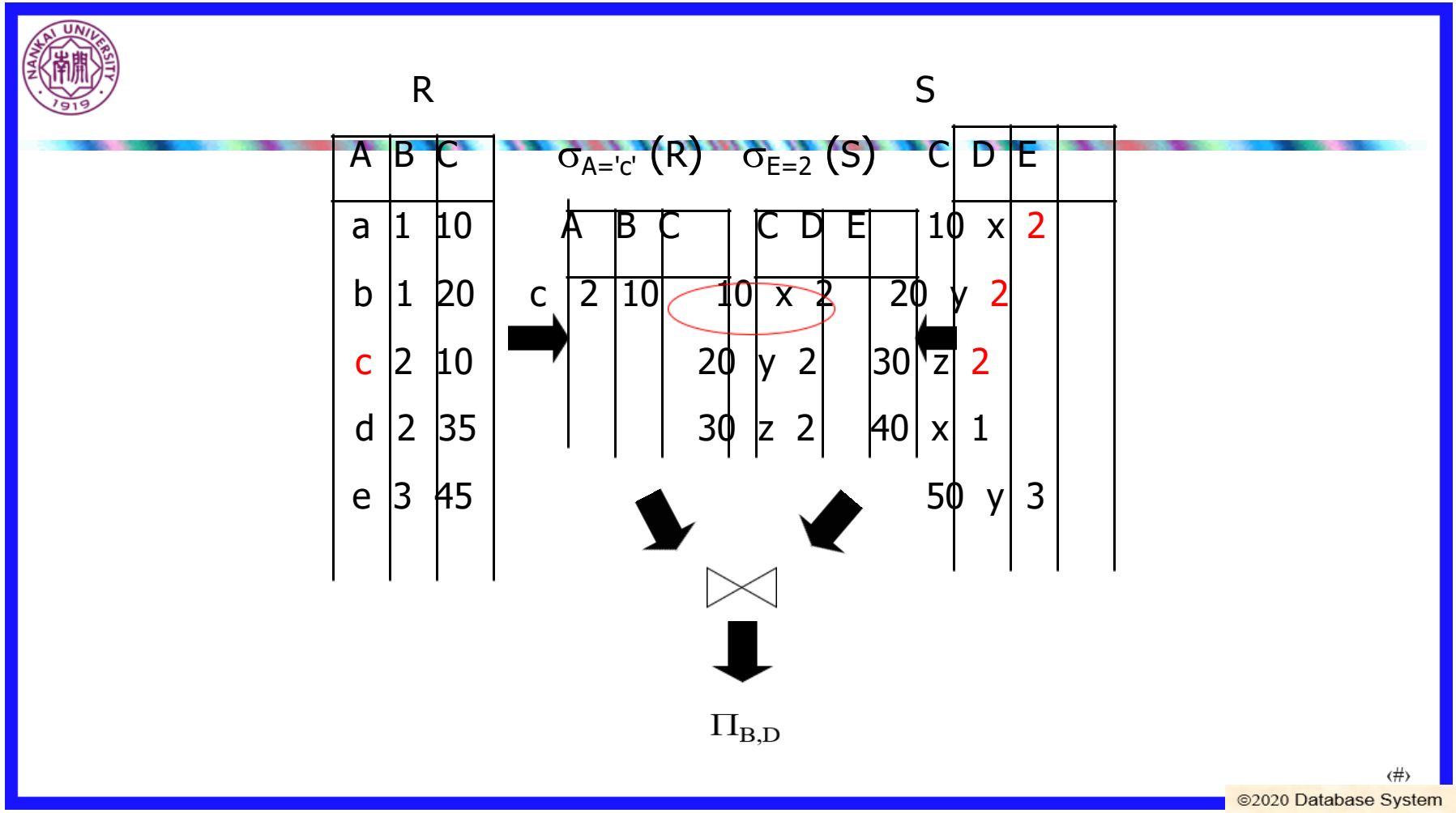
-> need more efficient evaluation methods

Improved logical query plan:

Plan II



natural join


R
S
A B C
a 1 10
b 1 20
c 2 10
d 2 35
e 3 45
σA='c' (R)
σE=2 (S)
A B C
c 2 10
C D E
10 x 2
20 y 2
30 z 2
40 x 1
C D E
10 x 2
20 y 2
30 z 2
50 y 3
ΠB,D
‹#›
©2020 Database System

Physical Query Plan:

Detailed description to execute the query:

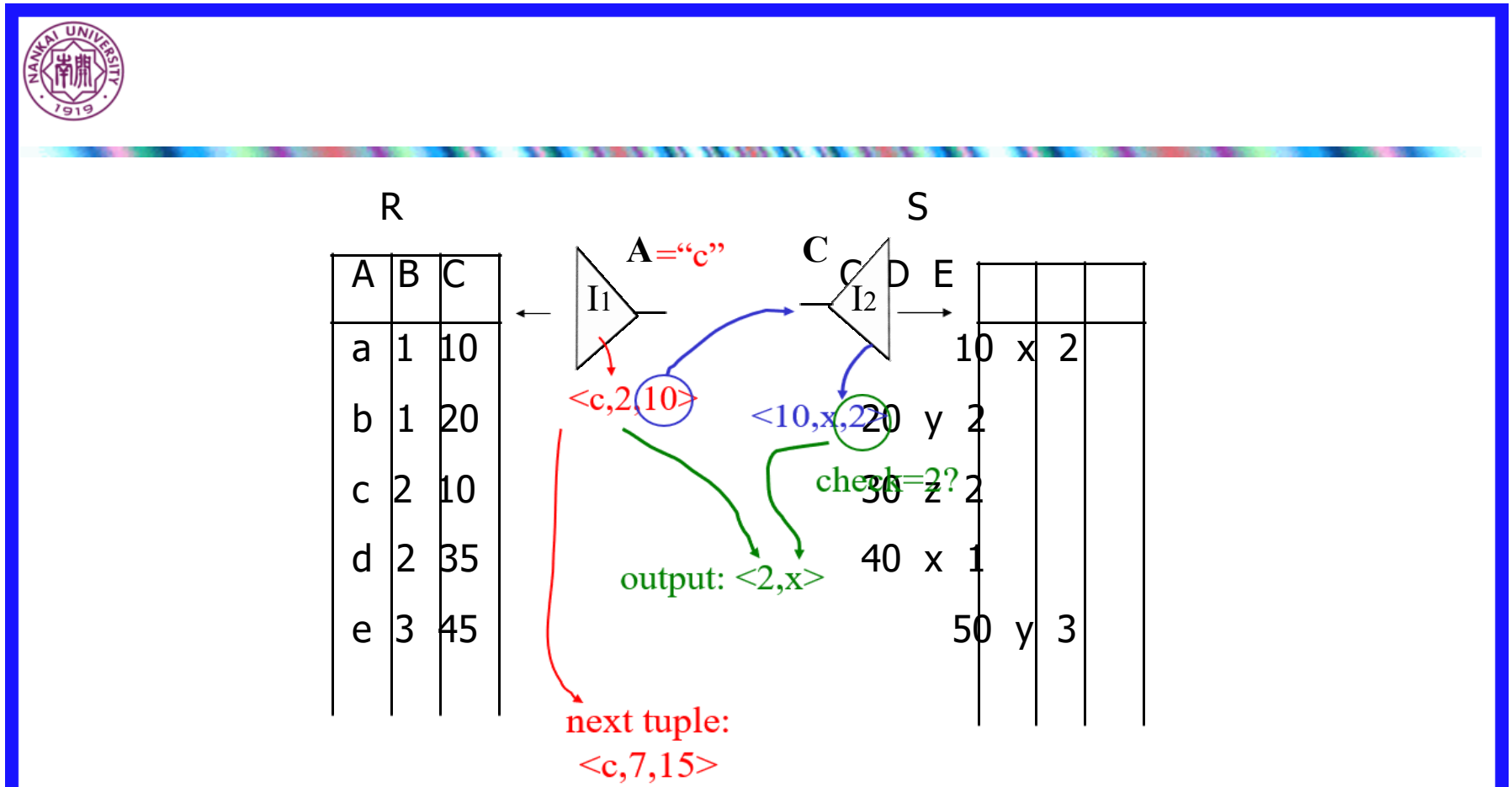
- algorithms to implement operations; order of execution steps; how relations are accessed; For example:

(1) Use R.A index to select tuples of R with R.A = "c"

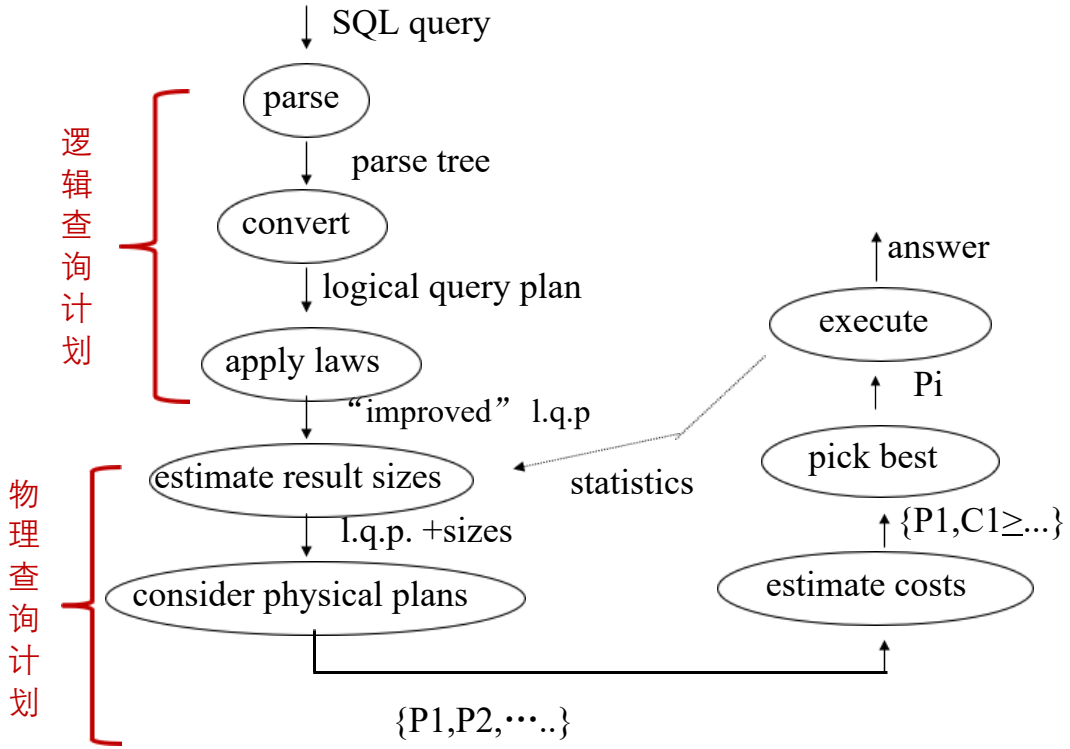
(2) For each R.C value found, use the index on S.C to find matching tuples

(3) Eliminate S tuples with S.E ≠ 2

(4) Join matching R,S tuples, project on attributes B and D, and place in result



Summary - Overview of Query Execution



多选题 1分

互动交流一-不定项选择题

在数据库的查询处理过程中，以下哪个描述是正确的?

A 先生成物理查询计划，再生成逻辑查询计划

B 先生成逻辑查询计划，再生成物理查询计划

C 生成物理查询计划之前，需要估计运算对象的大小尺寸

D 生成逻辑查询计划之前，需要估计运算对象的大小尺寸

提交

单选题 1分

互动交流二

在数据库的查询处理过程中，以下哪个描述是正确的?

A 生成多个物理查询计划，生成多个逻辑查询计划

B 生成一个物理查询计划，生成多个逻辑查询计划

C 生成多个物理查询计划，生成一个逻辑查询计划

D 生成一个物理查询计划，生成一个逻辑查询计划

提交

单选题 1分

互动交流三

在数据库的查询处理过程中，如果构建索引，会对哪个查询计划生成有影响？

A 只对逻辑查询计划有影响

B 只对物理查询计划有影响

C 对物理查询计划和逻辑查询计划都有影响

D 对物理查询计划和逻辑查询计划都没有影响

提交

主观题 10分

互动交流四

以下几个动作是数据库查询执行过程的几个基本步骤，请给出它们执行的正确顺序。

1. 解析SQL语句
2. 估计中间运算结果大小
3. 生成物理查询计划
4. 逻辑优化
5. 生成逻辑查询计划
6. 执行最佳查询计划

提交

Week13_Handout

Database System_Query Processing part2

Algebraic optimization of logical query plans

‹#›

Query Processing

How does the query processor execute queries?

Query → Query Plan

- SQL - expression of methods to realize the query

- Overview of Query Execution
- Logical Query Plan - relational algebra optimization
- Physical Query Plan - physical operators

Relational Algebra

StarsIn(title, year, starName), MovieStar(name, address, gender, birthdate)

```
SELECT title
FROM       StarsIn, MovieStar
WHERE starName = name AND birthdate LIKE '%1960' ;
```

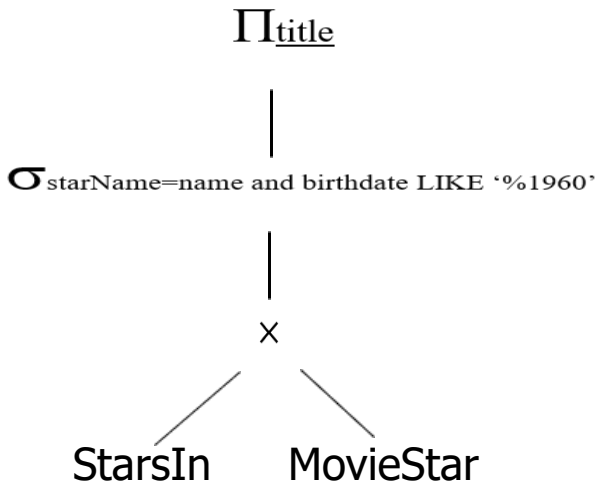
$$\Pi_{\text{title}}(\sigma_{\text{starName=name and birthdate LIKE '\%1960'}}(StarsIn \times MovieStar))$$

$$\Pi_{\text{title}}(StarsIn \underset{\text{starName=name}}{\bowtie} \sigma_{\text{birthdate LIKE '\%1960'}}(MovieStar))$$

$$\Pi_{\text{title}}(StarsIn \underset{\text{starName=name}}{\bowtie} \Pi_{\text{name}}(\sigma_{\text{birthdate LIKE '\%1960'}}(MovieStar)))$$

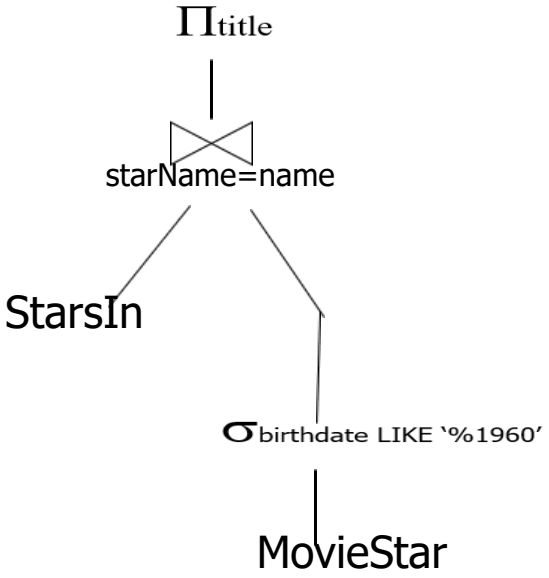
Relational Algebra Optimization

Plan I



是否等价?

Plan II



Relational algebra optimization

- Transformation rules

  (preserve equivalence)

- What are good transformations?

Rules: Natural joins

$R \bowtie S = S \bowtie R$ (*commutative*)

$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$ (*associative*)

- Carry attribute names in results, so order is not important
  -> Can evaluate in any order

Rules:

Cross products & union similarly
(both associative & commutative):

$$(R \times S) \times T = R \times (S \times T)$$

$$R \times S = S \times R$$

$$R \cup (S \cup T) = (R \cup S) \cup T$$

$$R \cup S = S \cup R$$

Rules: Selects

NB: $\wedge$ = AND; v = OR

1. $\sigma_{p1 \wedge p2}(R) = \sigma_{p1}[\sigma_{p2}(R)]$
2. $\sigma_{p1 v p2}(R) = [\sigma_{p1}(R)] \cup [\sigma_{p2}(R)]$

1. Especially useful (applied left-to-right): Allows compound select conditions to be split and moved to suitable positions

Bags vs. Sets

R = {a,a,b,b,b,c}
S = {b,b,c,c,d}
RUS = ?

- Option 1 SUM
  RUS = {a,a,b,b,b,b,b,c,c,c,d}
- Option 2 MAX
  RUS = {a,a,b,b,b,c,c,d}

Option 2 (MAX) makes this rule work:

$\sigma_{p1vp2}(R) = \sigma_{p1}(R) \cup \sigma_{p2}(R)$

Example: R={a,a,b,b,b,c}

P1 satisfied by a,b; P2 satisfied by b,c

$\sigma_{p1vp2}(R) = \{a,a,b,b,b,c\}$

$\sigma_{p1}(R) = \{a,a,b,b,b\}$

$\sigma_{p2}(R) = \{b,b,b,c\}$

$\sigma_{p1}(R) \cup \sigma_{p2}(R) = \{a,a,b,b,b,c\}$

Rules: Products to Joins

Definition of Natural Join:

$$R \bowtie S = \pi_L[\sigma_C(R \times S)] ;$$

Condition C equates attributes common to R

and S, and π_L projects one copy of them out

Applied right-to-left
- definition of general join applied similarly

Rules: σ + $\bowtie$ combined

Let p = predicate with only R attribs

q = predicate with only S attribs

m = predicate with attribs common to R,S

$\sigma_p (R \bowtie S) = [\sigma_p (R)] \bowtie S$

$\sigma_q (R \bowtie S) = R \bowtie [\sigma_q (S)]$

More rules can be derived...

Derived rules for σ + $\bowtie$

$$\sigma_{p \wedge q}(R \bowtie S) = [\sigma_p(R)] \bowtie [\sigma_q(S)]$$

$$\sigma_{p \wedge q \wedge m}(R \bowtie S) = \sigma_m[(\sigma_p R) \bowtie (\sigma_q S)]$$

$$\sigma_{p \vee q}(R \bowtie S) = [(\sigma_p R) \bowtie S] \cup [R \bowtie (\sigma_q S)]$$

Some "good" transformations- Summary

- $\sigma_{p1 \wedge p2}(R) \rightarrow \sigma_{p1}[\sigma_{p2}(R)]$
- $\sigma_{p}(R \bowtie S) \rightarrow [\sigma_{p}(R)] \bowtie S$
- $R \bowtie S \rightarrow S \bowtie R$

- No transformation is always good
- Usually good: early selections

单选题 1分

互动交流一

已知关系模式R(A,B)和S(B,C)，有以下两个关系代数运算：

Q1: $\sigma_{A=1}(R \bowtie S)$ Q2: $(\sigma_{A=1}(R)) \bowtie$

A Q1和Q2产生的结果一样

B Q1的结果总是包含Q2的结果

C Q2的结果总是包含Q1的结果

D Q1和Q2产生不同的结果

提交

多选题 1分

互动交流二-不定项选择题

在关系代数的优化策略中，以下哪个运算符应该尽可能的早做?

A Π

B $\bowtie$

C $\times$

D σ

提交

Week13_Handout

Database System_Query Processing part

Physical Query Plans

Query Processing

How does the query processor execute queries?

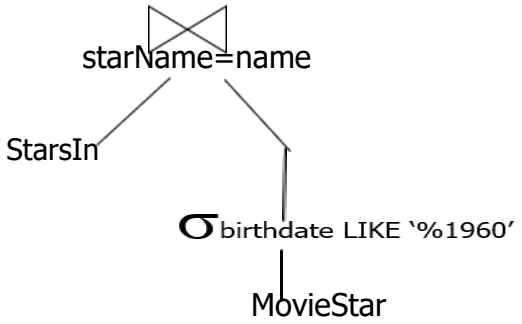
Query → Query Plan

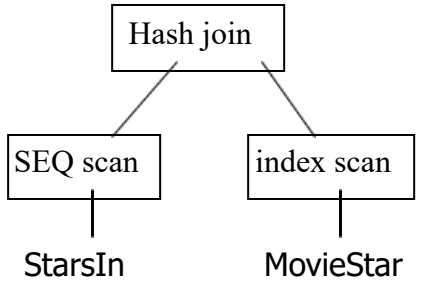
- SQL - expression of methods to realize the query

- Overview of Query Execution
- Logical Query Plan - relational algebra optimization
- Physical Query Plan - physical operators

Physical operators

- Principal methods for executing operations of relational algebra
- Building blocks of physical query plans
- Major strategies
  - scanning tables
  - sorting, indexing, hashing





Cost Estimates

- Estimate only # of disk I/O operations
  - dominating efficiency factor
    - exception: communication of data over network
- Simplifying assumptions
  - ignore the cost of writing the result
    - result blocks often passed in memory to further operations ("pipelining")
  - I/O happens one block at a time

    (e.g., ignore usage of cylinder sized blocks)

Parameters for Estimation

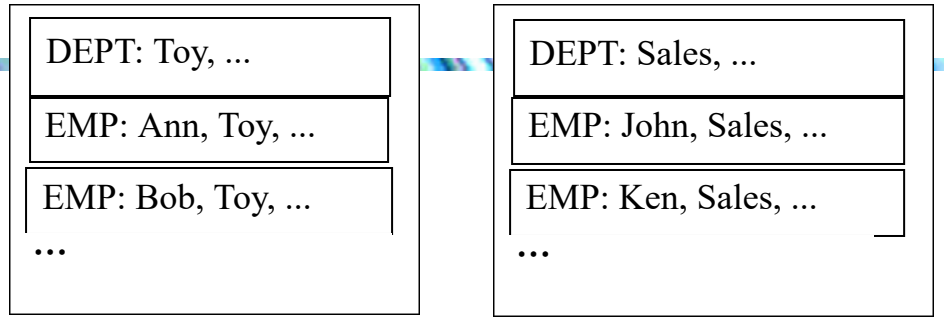
- M: # of available main memory buffers (estimate)

- Kept as statistics for each relation R:
  - T(R) : # of tuples in R
  - B(R): # of blocks to hold all tuples of R
  - V(R, A): # of distinct values for attribute R.A
    = SELECT COUNT (DISTINCT A) FROM R

Cost of Scanning a Relation

- Normally assume relation R to be clustered, that is, stored in blocks exclusively (or predominantly) used for representing R
- For example, consider a *clustered-file organization* of relations

  DEPT(Name, …) and EMP(Name, Dname, …)



- Relation EMP might be considered clustered, relation DEPT probably not
- For a clustered relation R, sufficient to read (approx.) B(R) blocks for a full scan
- If relation R not clustered, most tuples probably in different blocks => input cost approx. T(R)

Classification of Physical Operators (1)

- By method:
  - sort-based
    - process relation(s) in the sorted order
  - hash-based
    - process relation(s) partitioned in hash buckets
  - index-based
    - apply existing indexes
    - especially useful for selections

Classification of Physical Operators (2)

- By applicability and cost:
  - one-pass methods
    - if at least one argument relation fits in main memory
  - two-pass methods
    - if memory not sufficient for one-pass
    - process relations twice, storing intermediate results on disk
  - multi-pass
    - generalization of two-pass for HUGE relations

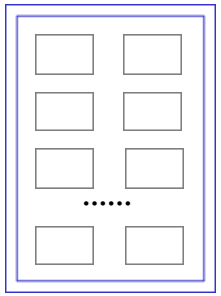
Implementing Selection

- How to evaluate $\sigma_C(R)$?
  - Sufficient to examine one tuple at a time -> Easy to evaluate in one pass:
    - Read each block of R using one input buffer
    - Output records that satisfy condition C
  - If R clustered, cost = B(R); else T(R)
- Projection $\pi_A(R)$ in a similar manner

R

AA	BB	CC
a_1	b_1	c_1
⋮	⋮	⋮

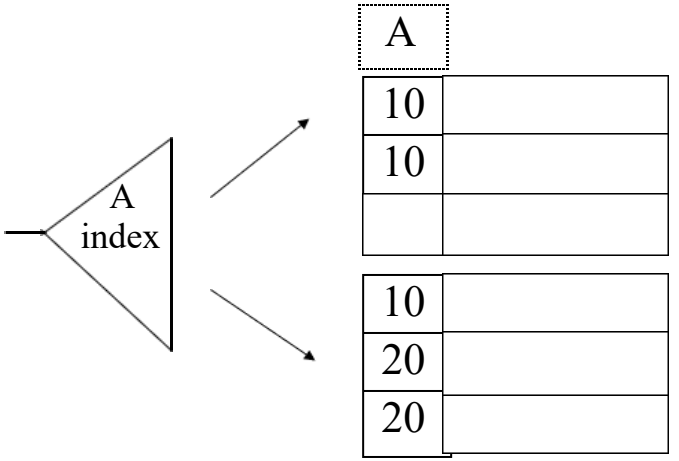
M



Index-Based Selection

- Consider selection $\sigma_{A='c'}(R)$
- If there is an index on R.A, we can locate tuples t with t.A='c' directly
- What is the cost?
  - How many tuples are selected?
    - estimate: T(R)/V(R,A) on the average
    - if A is a primary key, V(R,A) =T(R) -> 1 disk I/O

- Index is clustering, if tuples with A='c' are stored in consecutive blocks (for any 'c')



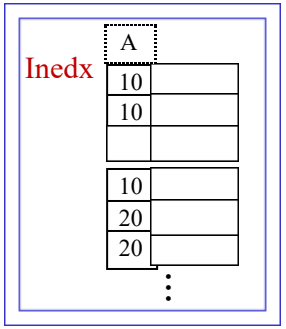
Index-Based Selection

- Consider selection $\sigma_{A='c'}(R)$
- If there is an index on R.A, we can locate tuples t with t.A='c' directly
- What is the cost?
  - How many tuples are selected?
    - estimate: T(R)/V(R,A) on the average
    - if A is a primary key, V(R,A) =T(R) -> 1 disk I/O

R

A	B	C
a_1	b_1	c_1
⋮	⋮	⋮

M



Selection using a clustering index

- We estimate a fraction T(R)/V(R,A) of all R tuples to satisfy A='c'. Apply same estimate to data blocks accessible through a clustering index => $\lceil B(R)/V(R,A) \rceil$ is an estimate for the number of block accesses

- Further simplifications: Ignore, e.g.,
  - cost of reading the (few) index blocks
  - unfilled room left intentionally in blocks
  - ...

Selection Example

Consider $\sigma_{A=0}(R)$ when T(R)=20,000, B(R)=1000, and there's an index on R.A

- simple scan of R
  - if R not clustered: cost = T(R) = 20,000
  - if R clustered: cost = B(R) = 1000
- if V(R,A)=100 and index is …
  - not clustering -> cost = T(R)/V(R,A) = 200
  - clustering -> cost = B(R)/V(R,A)= 10
- if V(R,A)=20,000 (i.e., A is key) -> cost = 1

Time if disk I/O
15 ms
5 min
15 sec
3 sec
0.15 sec
15 ms

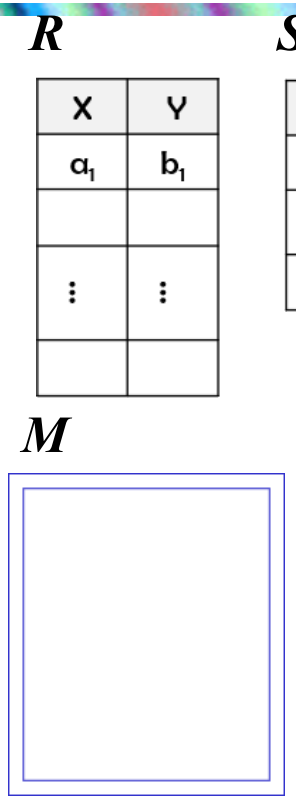
‹#›

Processing of Joins

- Consider natural join R(X,Y) ⋈ S(Y,Z)
  - general joins rather similarly, possibly with additional selections (for complex join conditions)
- Assumptions:
  - Y = join attributes common to R and S
  - S is the smaller of relations: B(S) ⩽⩽B(R)

One-Pass Join

- Requirement: B(S) < M, i.e., S fits in memory
- Read entire S in memory (using one buffer);
  Build a dictionary (balanced tree, hash table) using join attributes of tuples as search key
- Read each block of R (using one buffer);
  For each tuple t, find matching tuples from the dictionary, and output their join
- I/O cost: B(S) + B(R) ⩽



What If Memory Insufficient?

- Basic join strategy:
  - "nested-loop" join
  - ”1+n pass” operation:
    - one relation read once, the other repeateadly
  - no memory limitations
    - can be used for relations of any size

Nested-loop join

- Nested-loop join (conceptually)

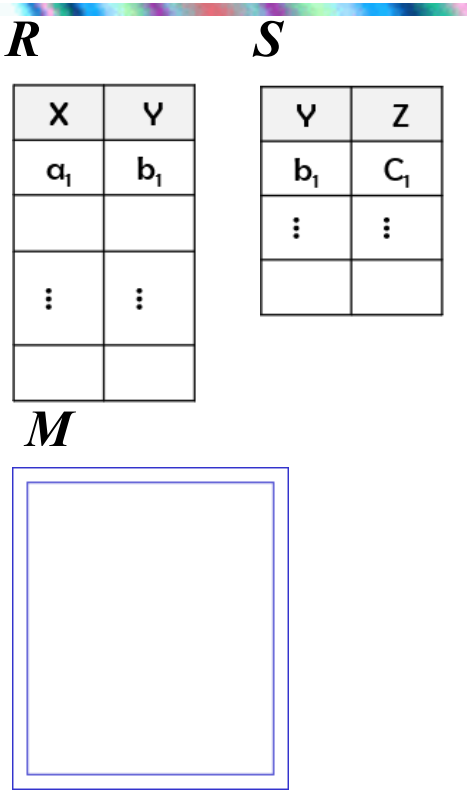
```
for each tuple s ∈ S do
    for each tuple r ∈ R do
        if r.Y = s.Y then
            output join of r and s;
```

- Cost (like for Cartesian product):
  T(S) *(1 + T(R)) = T(S) + T(S)T(R)

block-based nested-loop join

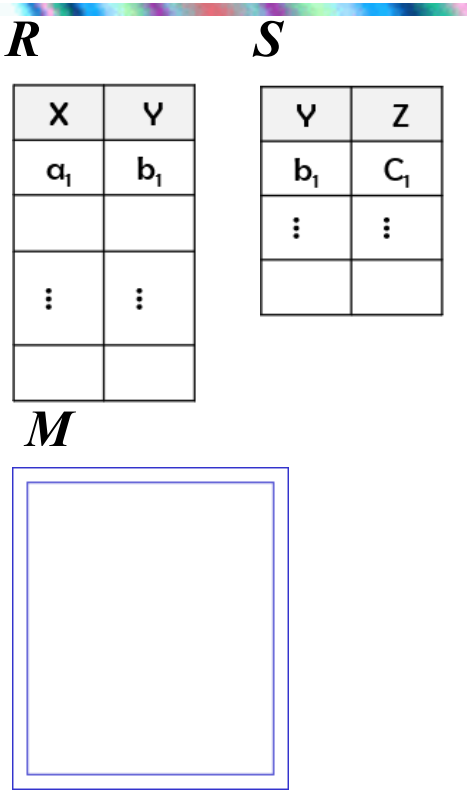
- If R and S clustered, can apply block-based nested-loop join:

```
for each chunck of M-1 blocks of S do
    Read blocks in memory;
    Insert tuples in a dictionary using the join
                    attributes;
    for each block b of R do
       Read b in memory;
       for each tuple r in b do
          Find matching tuples from  the dictionary;
        output their join with r;
```



Analysis of Nested-Loop join

- B(S)/(M-1) outer-loop iterations;
  Each reads M-1 + B(R) blocks
  -> total cost = B(S) + B(S)B(R)/(M-1),
  or approx. B(S)B(R)/M blocks
- Not the best method, but sometimes the only choice
- Next: More efficient join algorithms



Cost of Block-Based Nested-Loop Join

- Consider R(X,Y) ⋈ S(Y,Z) when B(R)=1000, B(S)=500, and M = 101
  - Use 100 buffers for loading S
    -> 500/100 = 5 chunks
  - Total I/O cost = 5 x (100 + 1000) = 5500 blocks
- R as the outer-loop relation -> I/O cost 6000
  - in general, using the smaller relation in the outer loop gives an advantage of B(R) - B(S) operations

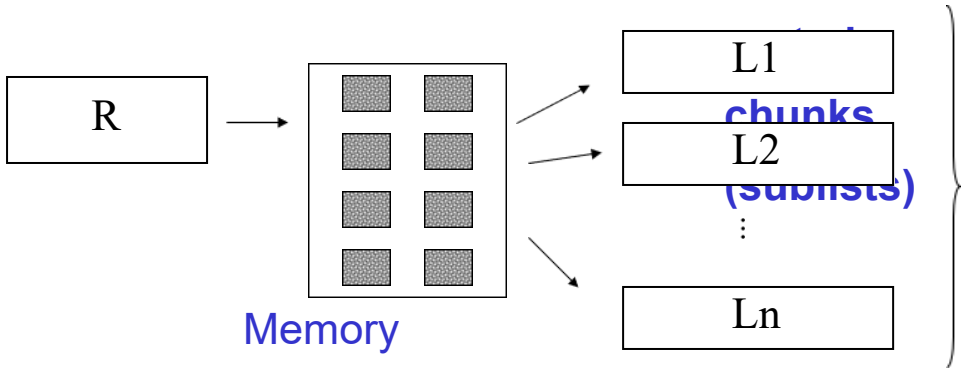
Sort-Based Two-Pass Join

- Idea: Joining relations R and S on attribute Y is rather easy, if the relations are sorted using Y
  - IF not too many tuples join for any value of the join attributes. (E.g. if $\pi_Y(R) = \pi_Y(S) = \{y\}$, all tuples match, and we may need to resort to nested-loop join)
- If relations not sorted already, they have to be sorted (with two-phase multi-way merge sort, since they do not fit in memory)

One way to sort: Merge Sort

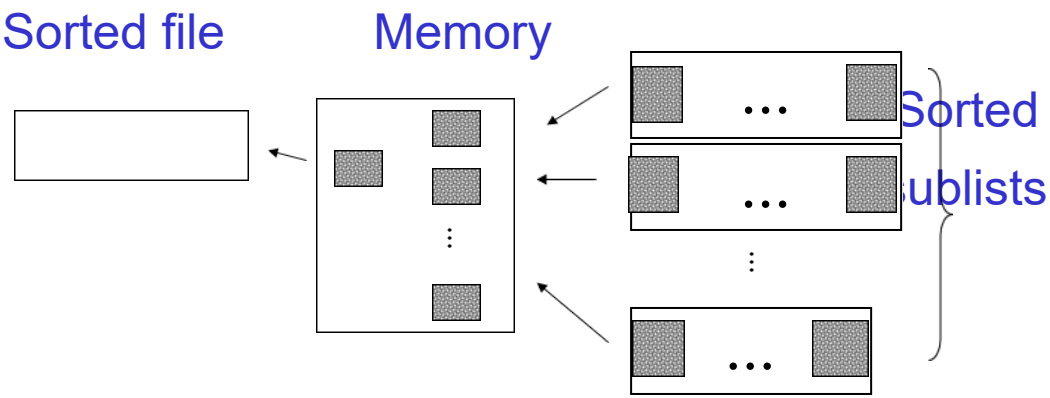
(i) For each main-memory-sized chunk of R:

- Read chunk
- Sort in memory (say, using quicksort)
- Write to disk



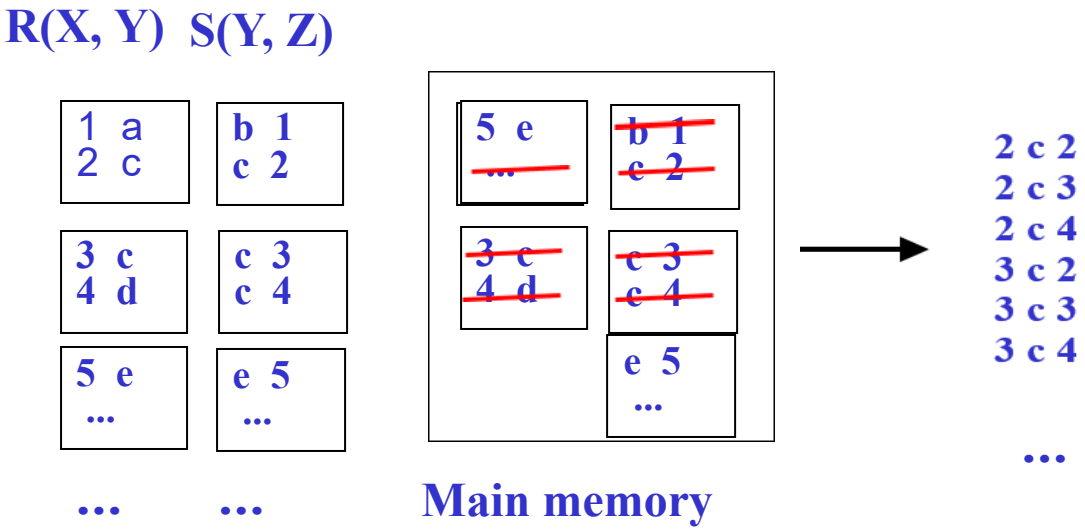
One way to sort: Merge Sort (cont.)

(ii) Read all chunks + merge + write out



Always move the smallest head element to the output buffer

Example: Join of R and S sorted on Y



Sort-Based Two-Pass Join

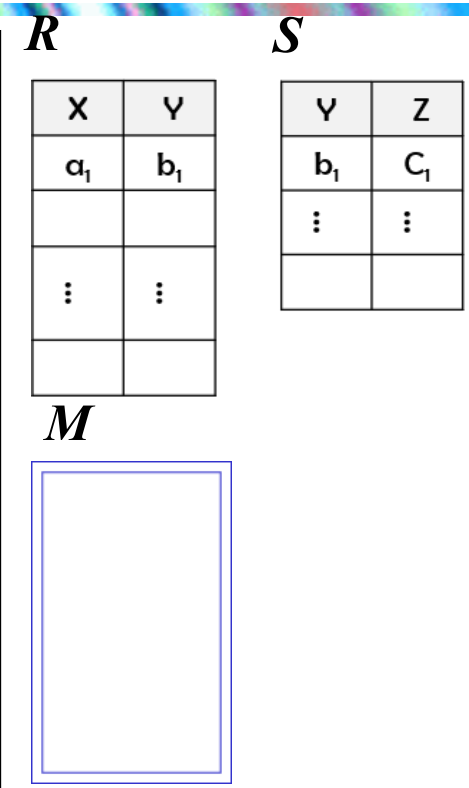
1. Sort R with join attributes Y as the sort key;
2. Do the same for relation S;
3. Merge the sorted relations, using 1 buffer for current input block of each relation:
   - skip tuples whose Y-value y not in both R and S
   - read blocks of both R and S for all tuples whose Y value is y
   - output all possible joins of the matching tuples $r \in R$ and $s \in S$

Analysis of Sort-Based Two-Phase Join

- Limitations? Sorting requires

  max(B(R), B(S)) ⩽⩽M² $M \geq \sqrt{\max\{B(R), B(S)\}}$

- Variation: Perform only phase I (building of sorted sublists) of the sorting, and merge all of them (can handle at most M) for the join
  - I/O cost = 3 x (B(R) + B(S))
  - requires the union of R and S to fit in at most M sublists, each of which at most M blocks long
  - if B(R)+ B(S) ⩽⩽M² $M \geq \sqrt{B(R)+B(S)}$



Analysis of Sort-Based Two-Phase Join

- Consider R(X,Y) ⋈ S(Y,Z) when B(R)=1000, B(S)=500, and M = 101
  - Remember two-phase multiway merge sort:
    - each block read + written + read + written once
      -> 4 x (B(R) + B(S)) = 6000 disk I/Os
  - Merge of sorted relations for the join:
    - B(R) + B(S)= 1500 disk I/Os
  - Total I/O cost = 5 x (B(R) + B(S)) = 7500
- Seems big, but for large R and S much better than B(R)B(S)/M of block-based nested loop join

Two-Phase Join with Hashing

- Idea: If relations do not fit in memory, first hash the tuples of each relation in buckets. Then join tuples in each pair of buckets.
- For a join on attributes Y, use Y as the hash key
  - Hash Phase: For each relation R and S:
    - Use 1 input buffer, and M-1 output buffers as hash buckets
    - Read each block and hash its tuples; When output buffer gets full, write it on disk as the next block of that bucket

Two-Phase Join with Hashing

- The hashing phase produces buckets (sequences of blocks)

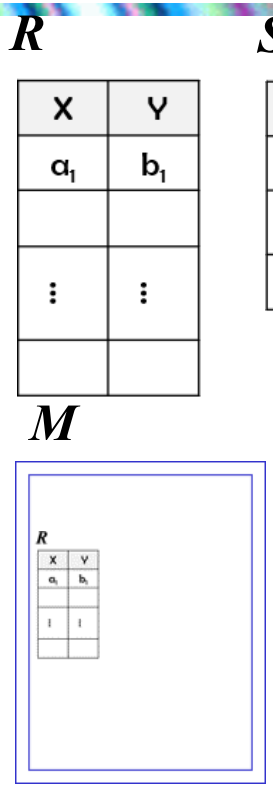
  $R_1, \ldots, R_{M-1}$ and $S_1, \ldots, S_{M-1}$

- Tuples $r \in R$ and $s \in S$ join iff $r.Y = s.Y$

  => $h(r.Y) = h(s.Y)$

  => r occurs in bucket R_i and

  s occurs in bucket S_i for the same i



Hash-Join: The Join Phase

- For each i = 1, ···, M-1, perform one-pass join between buckets R_i and S_i
  -> the smaller one has to fit in M-1 main memory buffers
- Average size for bucket R_i is approx. B(R)/M, and B(S)/M for bucket S_i

-> Approximated memory requirement min(B(R), B(S)) < M^2

$$M > \sqrt{\min\{B(R), B(S)\}}$$

Index-Based Join

- Still consider R(X,Y) ⋈ S(Y,Z)
- Assume there's an index on S.Y
- Can compute the join by
  - reading each tuple t of R
  - locating matching tuples of S by index-lookup for t.Y, and
  - outputting their join with tuple t
- Efficiency depends on many factors

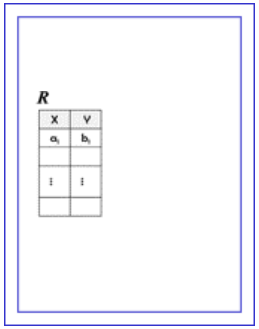
R

X	Y
a_1	b_1
⋮	⋮

S

Y	Z
b_1	C_1
⋮	⋮

M



Cost of Index-Based Join

- Cost of scanning R:
  - B(R), if clustered; T(R), if not
- On the average, T(S)/V(S,Y) matching tuples found by index lookup; Cost of loading them (total for all tuples of R):
  - T(R)T(S)/V(S,Y), if index not clustered
  - T(R)B(S)/V(S,Y), if index clustered
- Cost of loading tuples of S dominates
- if Y primary key of S:
  - B(R) + T(R), if R clustered, and
  - T(R) + T(R) = 2T(R), if R not clustered

I/O of R:

B(R) or T(R)

I/O of S:

T(R)T(S)/V(S,Y)

T(R)B(S)/V(S,Y)

‹#›

Joins Using a Sorted Index

- Still consider R(X,Y) ⋈ S(Y,Z)
- Assume there's a sorted index on both R.Y and S.Y
  - B-tree or a sorted sequential index
- Scan both indexes in the increasing order of Y
  - like merge-join, without need to sort first
  - if index dense, can skip nonmatching tuples without loading them
  - very efficient
- Details to excercises?

Summary - Query Processing

- Overall picture

  Query -> logical plan -> physical plan -> execution
- Relational algebra level
  - transformations
  - good transformations
- Physical operators
  - selections, joins
  - one-pass, two-pass, (multi-pass)
  - based on scanning, sorting, hashing, existing indexes

‹#›

课后练习（自选完成）

For the first problems, use the following schema, for an on-line bookstore :

Cust (CustID, Name, Address, State, Zip)

Book (BookID, Title, Author, Price, Category)

Order (OrderID, CustID, BookID, ShipDate)

For the join Order ⋈ Cust, state the minimum number of memory blocks you would need to perform each of the following join algorithms. Again, you may assume B(Order) = 3,000 blocks and B(Cust) = 1,000 blocks.

a. One pass join

b. Nested loop join

c. Two pass simple sort join

d. Two pass Sort join (also known as sort-merge join)

e. Two pass hash join