Tối ưu hoá dựa trên chi phí

NGUYỄN Ngọc Hoá

Bộ môn Hệ thống thông tin

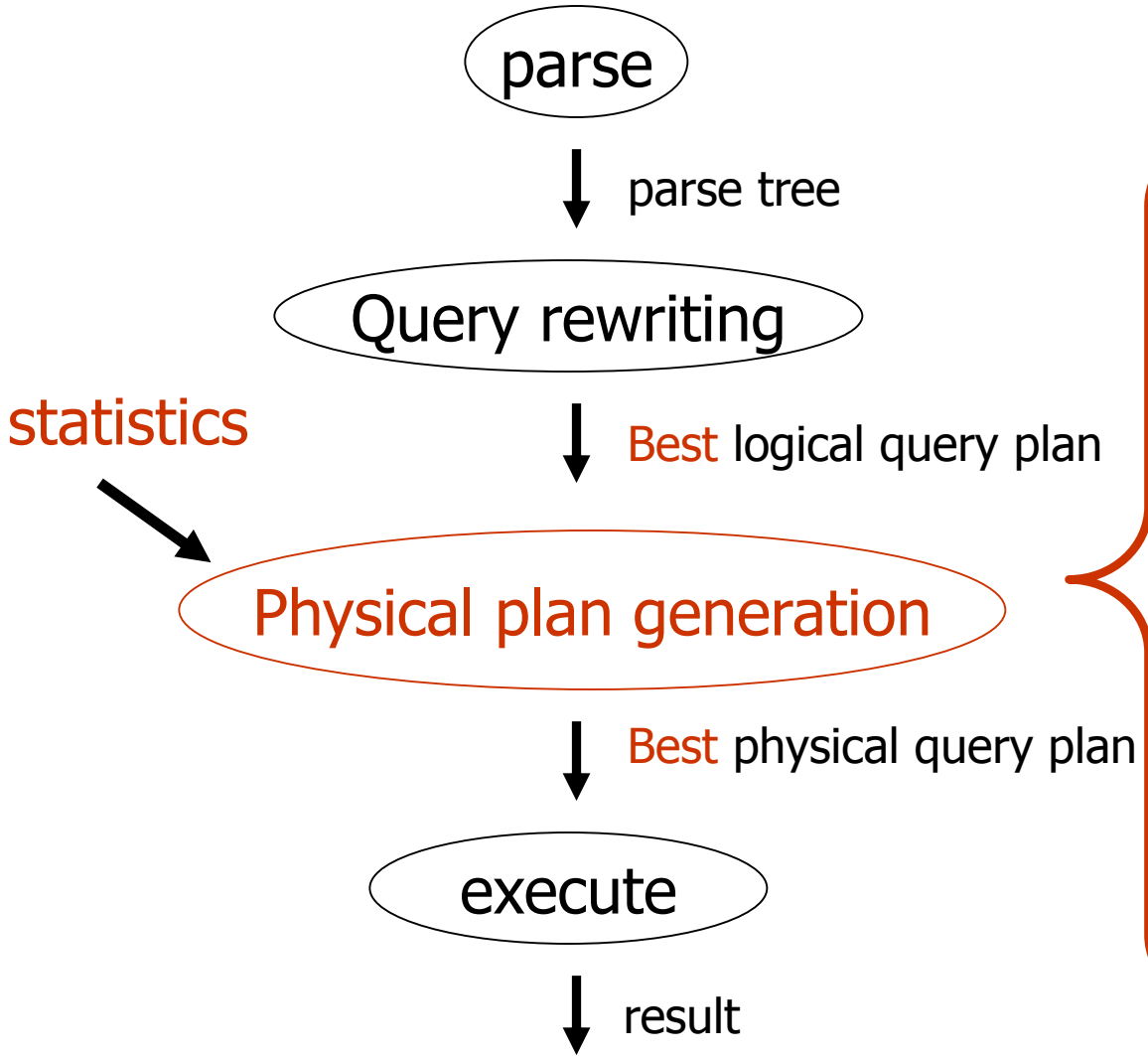
Khoa CNTT, trường Đại học Công nghệ

Đại học Quốc gia Hanoi

Hoa.Nguyen@vnu.edu.vn

Nội dung

- Mô hình chi phí
  - Nguyên lý tối ưu
  - Giải thuật Selinger
- Triển khai các phép toán trong truy vấn
  - Materialization
  - Pipelining
- Cài đặt các phép toán
  - Scan-based
  - Sort-based
  - Using existing indexes
  - Hash-based
- Ước lượng chi phí

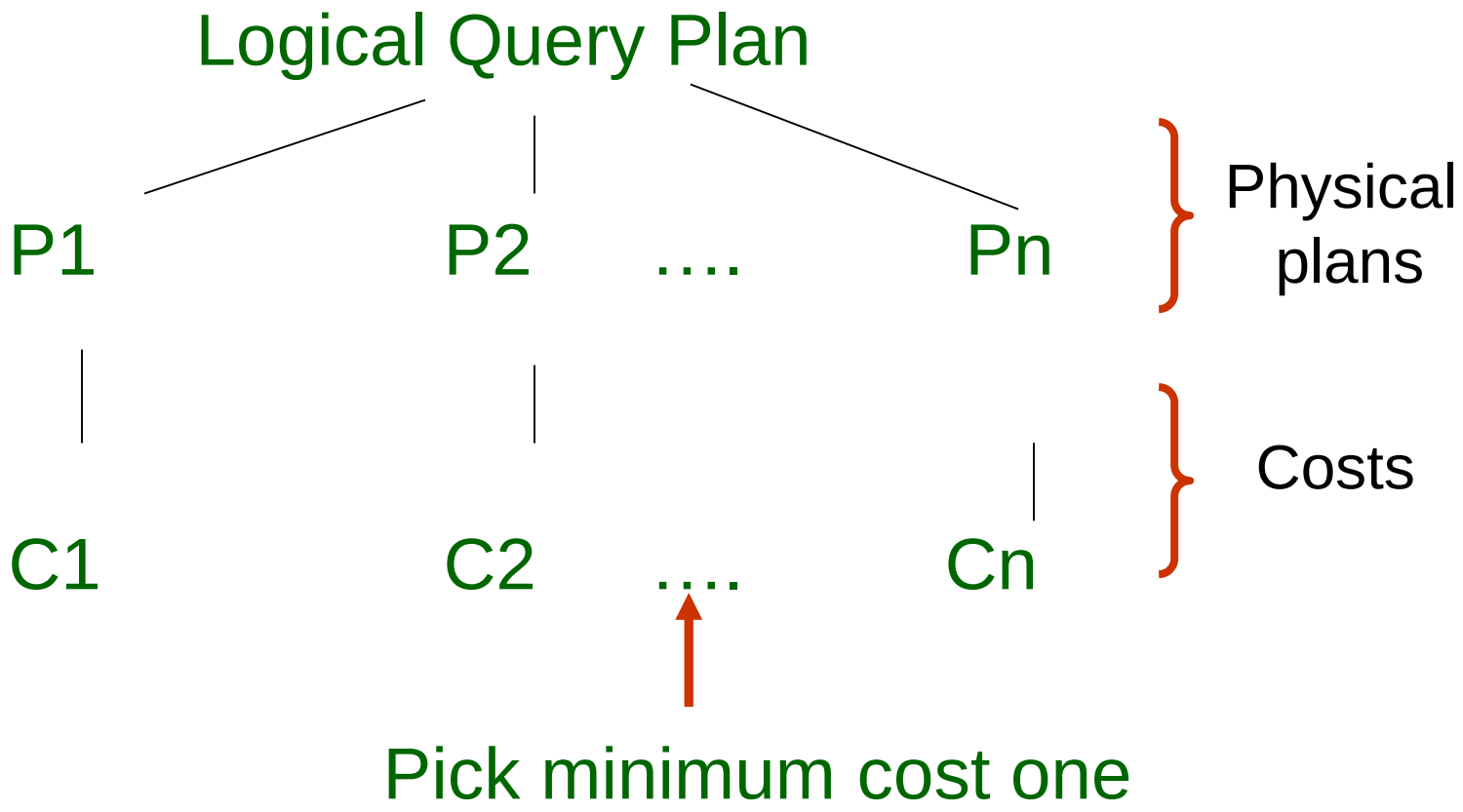


Liệt kê các physical plans có thể

Ước lượng chi phí của mỗi plan

Chọn plan có chi phí thấp nhất

Physical Plan Generation

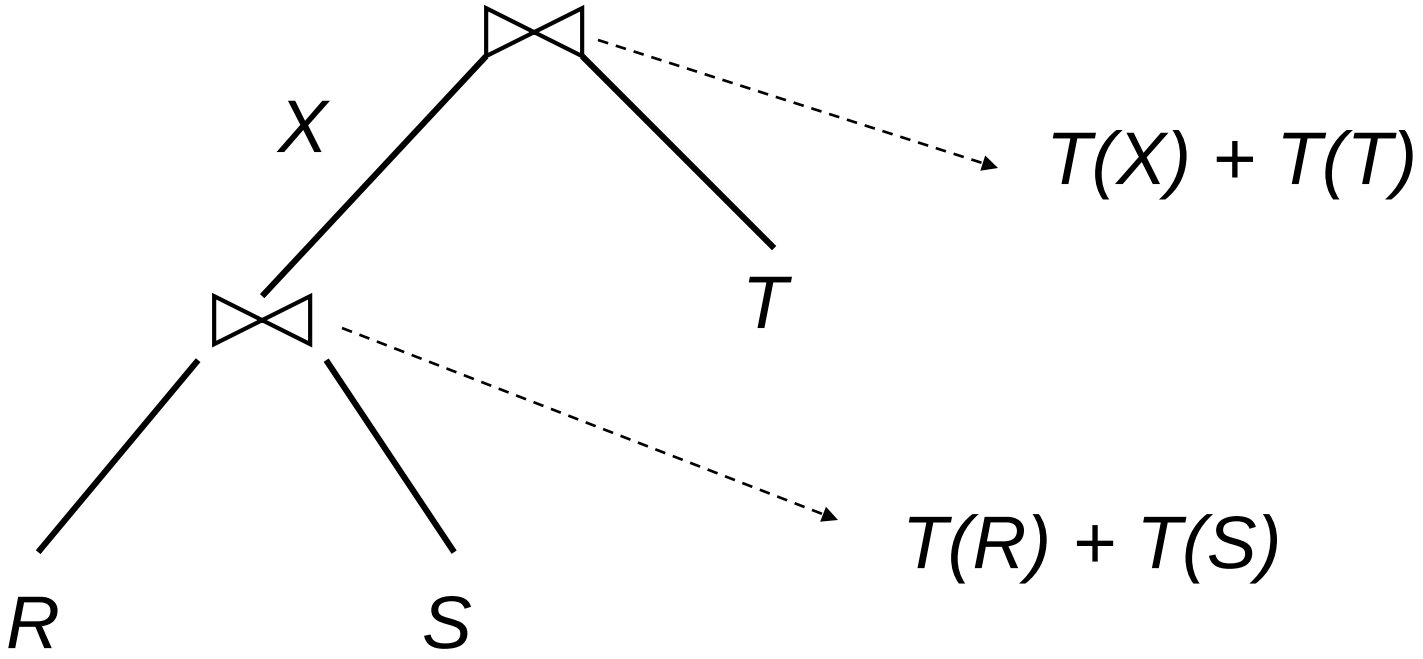


Mô hình chi phí đơn giản

$$Cost(R \bowtie S) = T(R) + T(S)$$

Các phép toán trong CPU chi phi = 0

Ví dụ



Total Cost: $T(R) + T(S) + T(T) + T(X)$

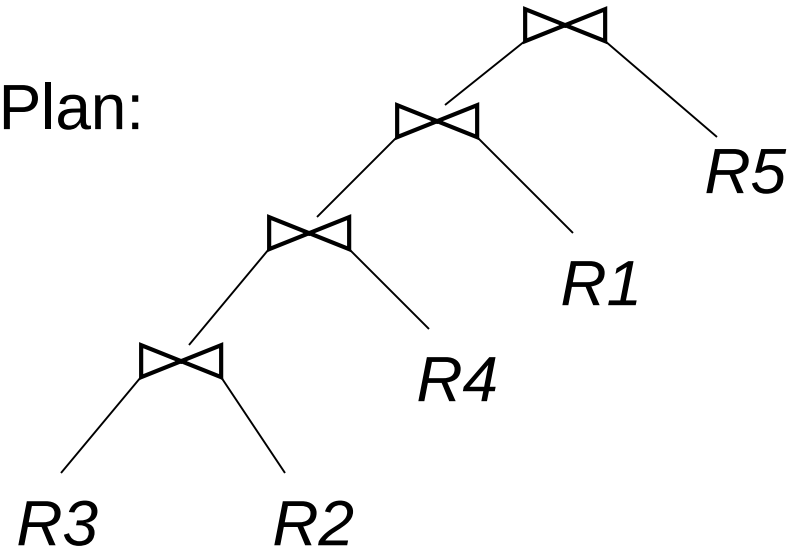
Nguyên lý tối ưu

Optimal for “whole” made up from optimal for “parts”

Nguyên lý tối ưu...

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4 \bowtie R5$

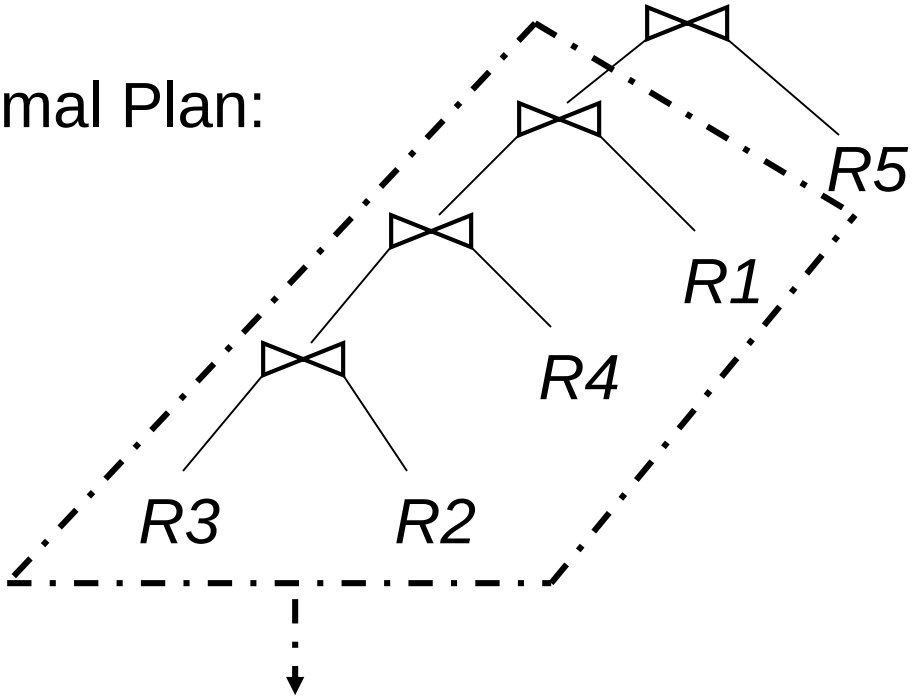
Optimal Plan:



Nguyên lý tối ưu...

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4 \bowtie R5$

Optimal Plan:

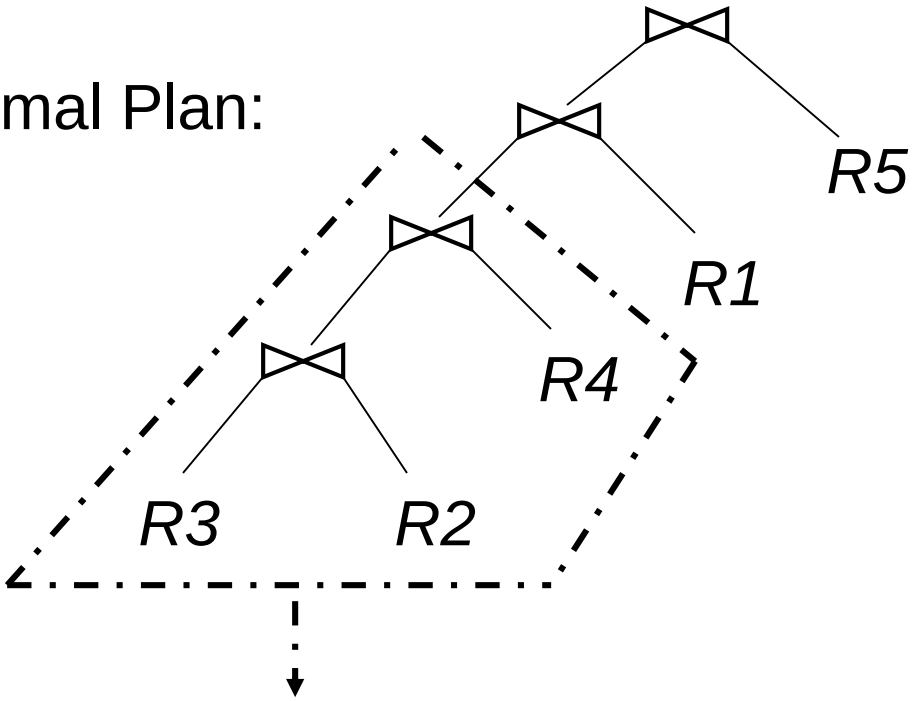


Optimal plan for joining *R3, R2, R4, R1*

Nguyên lý tối ưu...

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4 \bowtie R5$

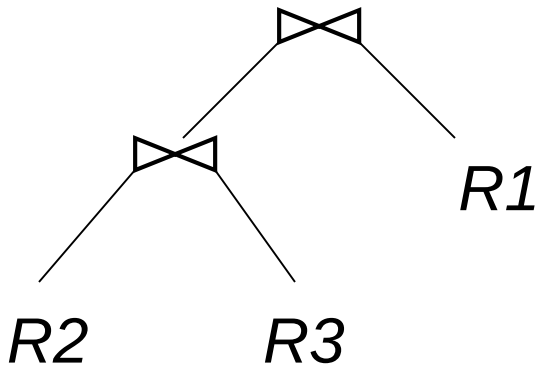
Optimal Plan:



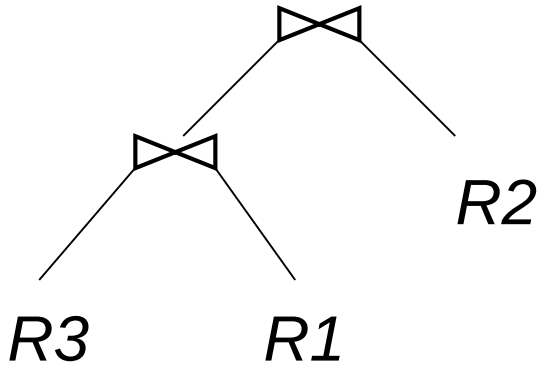
Optimal plan for joining *R3, R2, R4*

Chú ý

Query: $R1 \bowtie R2 \bowtie \ldots \bowtie Rn$

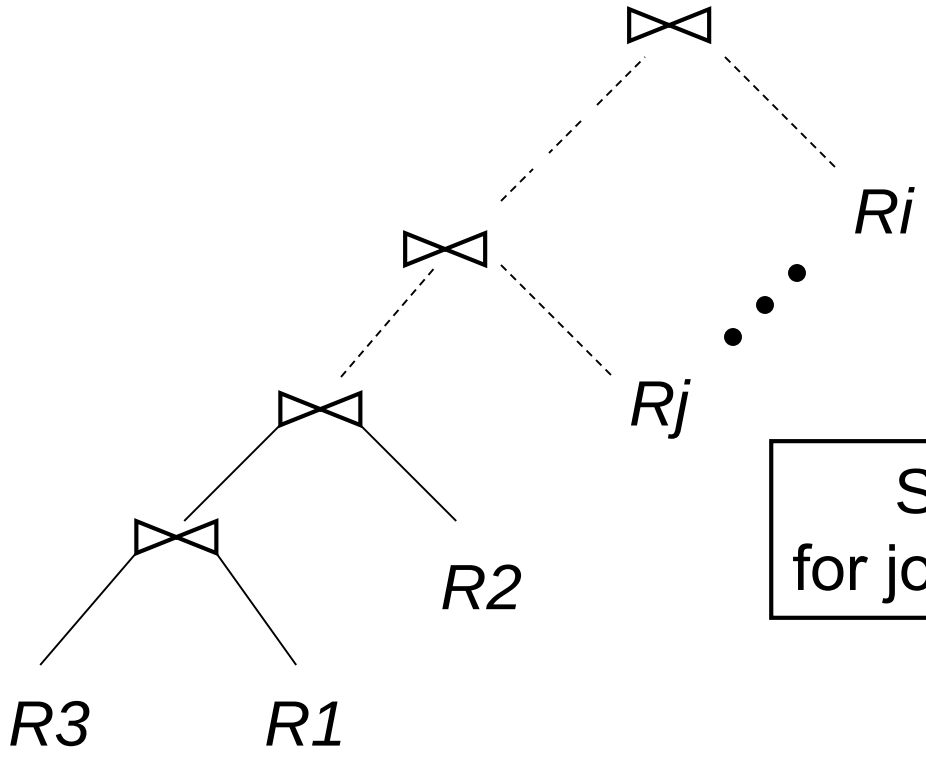


Optimal
for joining *R1, R2, R3*



Sub-Optimal
for joining *R1, R2, R3*

Chú ý…

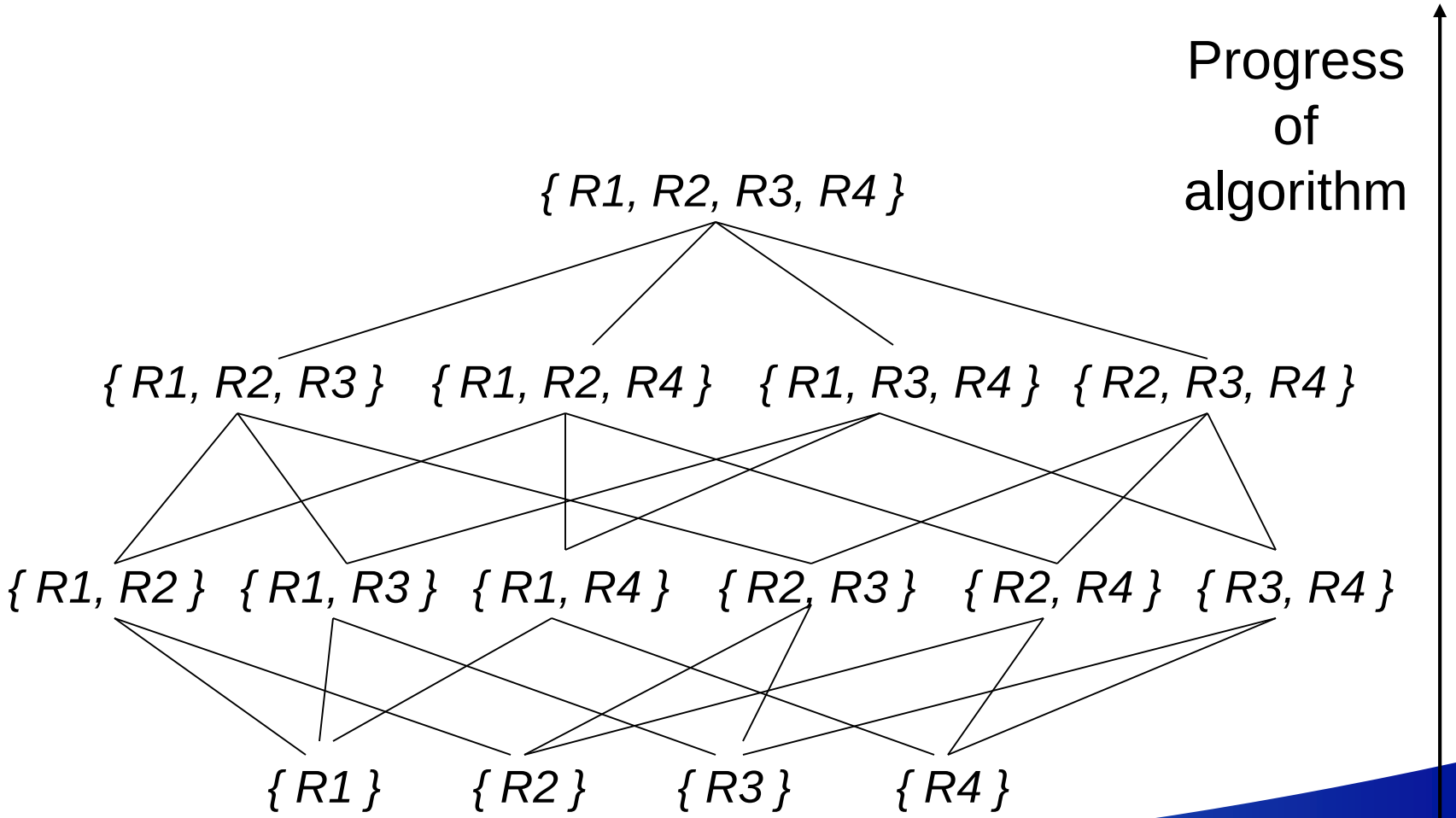


Sub-Optimal
for joining R1,…,Rn

Sub-optimal của sub-plan không thể tạo được optimal plan

Giải thuật Selinger

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Ký hiệu

OPT ({ *R1, R2, R3* }):

Chi phí của plan tối ưu với phép nối *R1,R2,R3*

T ({ *R1, R2, R3* }):

Số bộ trong $R1 \bowtie R2 \bowtie R3$

Giải thuật Selinger…

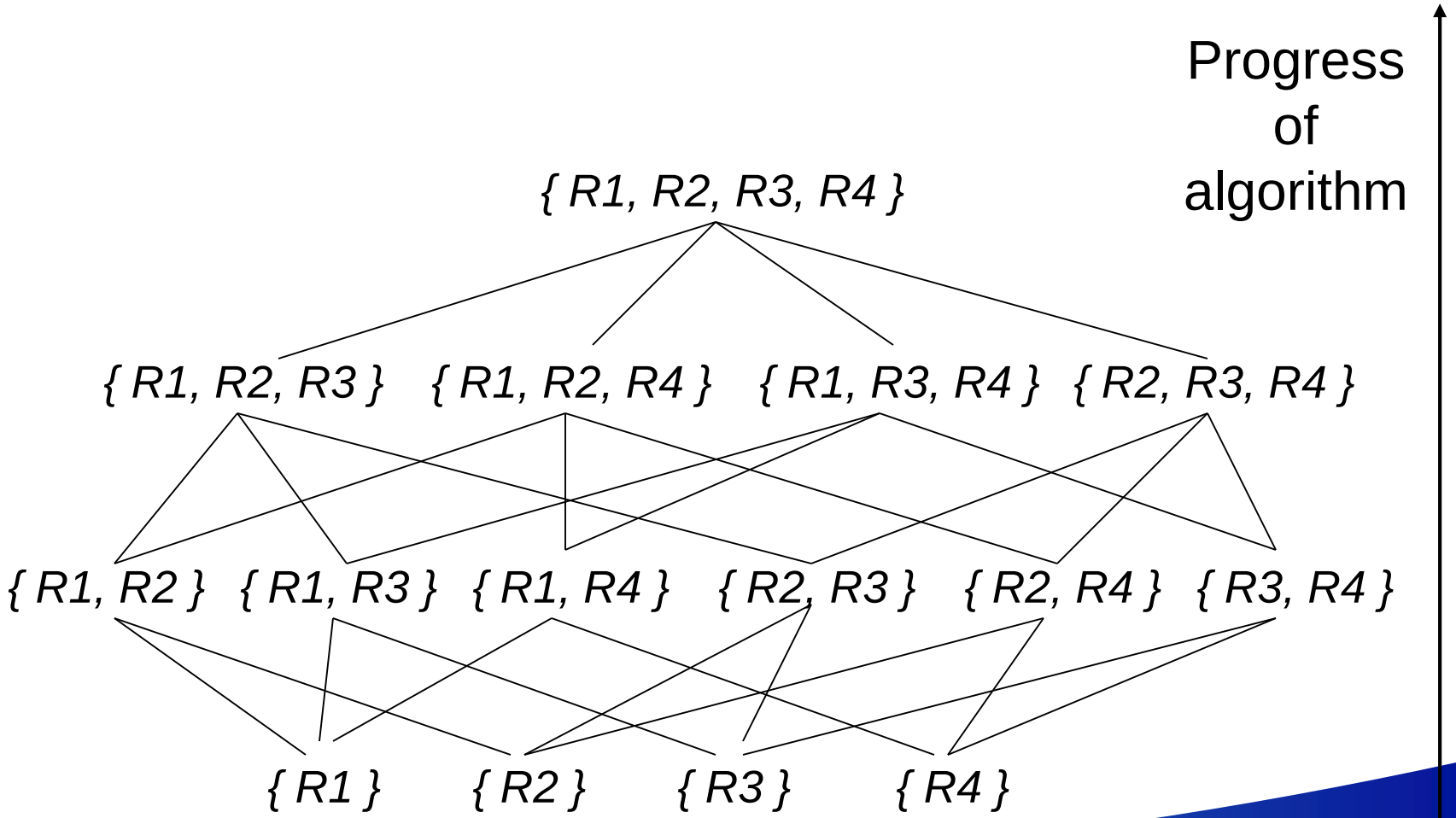
OPT ({ R1, R2, R3 }):

Min
- OPT ({ R1, R2 }) + T ({ R1, R2 }) + T(R3)
- OPT ({ R2, R3 }) + T ({ R2, R3 }) + T(R1)
- OPT ({ R1, R3 }) + T ({ R1, R3 }) + T(R2)

Note: chỉ xét với mô hình chi phí đơn giản

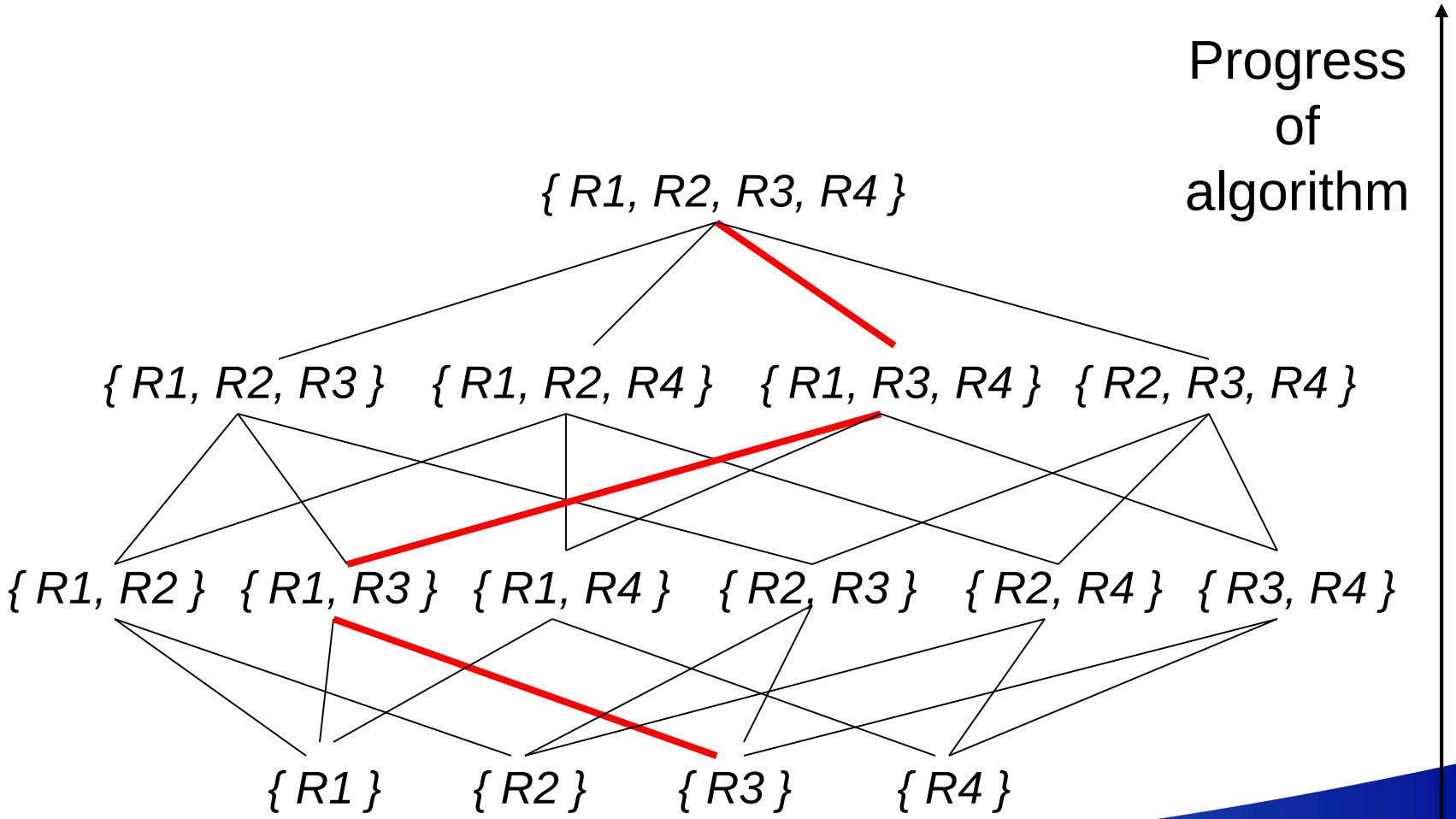
Giải thuật Selinger...

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Giải thuật Selinger...

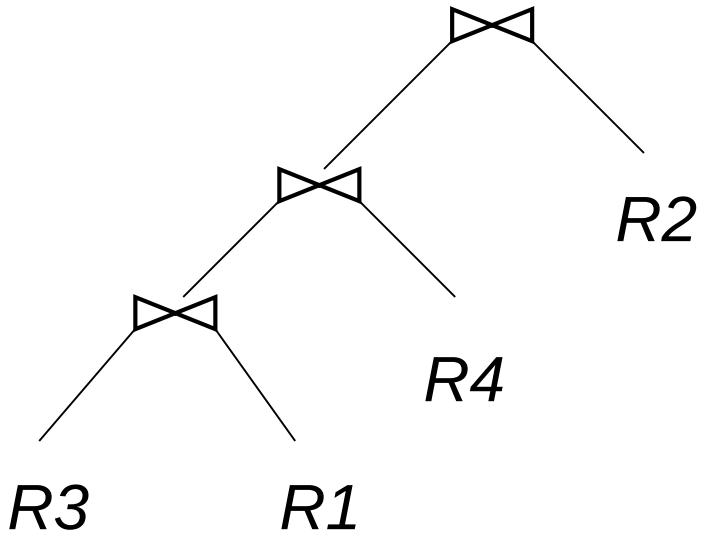
Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Giải thuật Selinger…

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$

Optimal plan:



Mô hình chi phí phức tạp

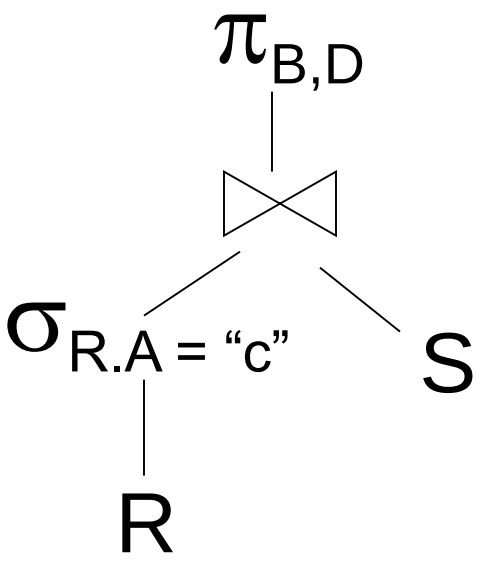
- Cách tiếp cận chính: chi phí được tính dựa trên số lần truy cập thiết bị lưu trữ I/O's
- Plan cost: Có thể tính dựa trên những tham số
  - Time to completion
  - Number of I/Os
  - Number of getNext() calls
  - Number of tuples (for each relation)
  - Number of blocks (for each relation)
  - Number of levels for an index
  - Number of first-level index blocks
  - Number of distinct values per attribute
- Tradeoff: Simplicity of estimation Vs. Accurate estimation of performance as seen by user

Nội dung

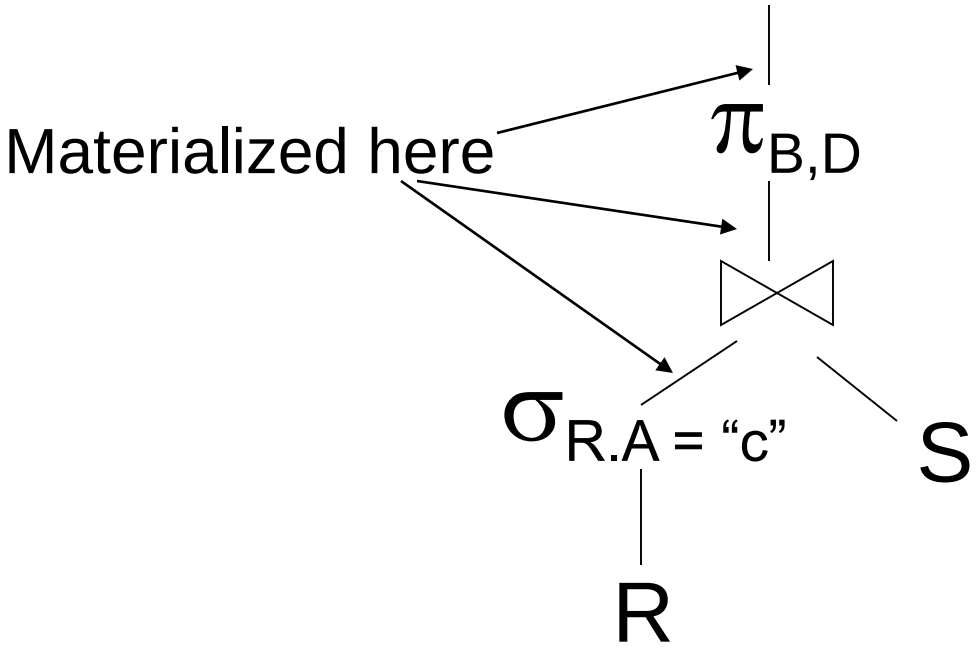
- Mô hình chi phí
  - Nguyên lý tối ưu
  - Giải thuật Selinger
- Triển khai các phép toán trong truy vấn
  - Materialization
  - Pipelining
- Cài đặt các phép toán
  - Scan-based
  - Sort-based
  - Using existing indexes
  - Hash-based
- Ước lượng chi phí

Triển khai các phép toán trong truy vấn

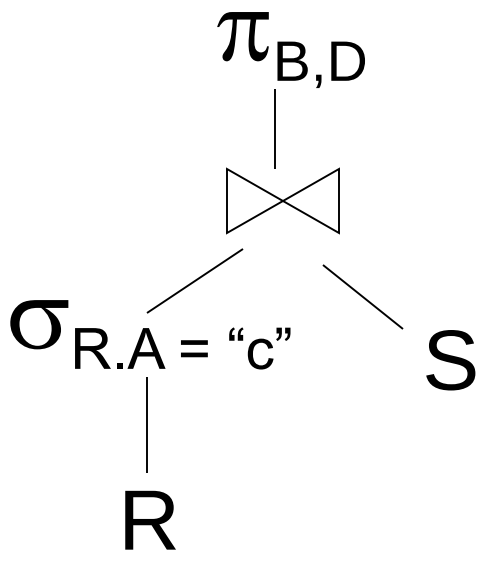
- Materialization: kết quả một phép toán sẽ được lưu trên thiết bị lưu trữ, phép toán kế tiếp sẽ đọc từ thiết bị lưu trữ đó
- Pipelining: kết quả một phép toán sẽ được sử dụng trực tiếp cho phép toán kế tiếp



Materialization



Iterators: Pipelining



➔ Mỗi phép toán sẽ hỗ trợ:

- Open()
- GetNext()
- Close()

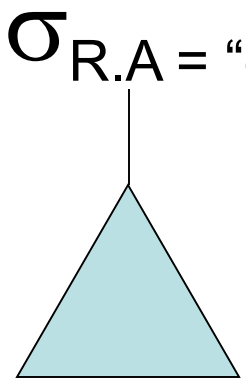
Iterator for Table Scan (R)

```
Open() {
  /** initialize variables */
  b = first block of R;
  t = first tuple in block b;
  Found = TRUE;
}

Close() {
  /** nothing to be done */
}
```

```
GetNext() {
  IF (t is past last tuple in block b) {
      set b to next block;
      IF (there is no next block)
          /** no more tuples */
           Found = FALSE;
           RETURN EOT;
      ELSE t = first tuple in b;
  }
  /** return current tuple */
  oldt = t;
  set t to next tuple in block b;
  RETURN oldt;
}
```

Iterator for Select

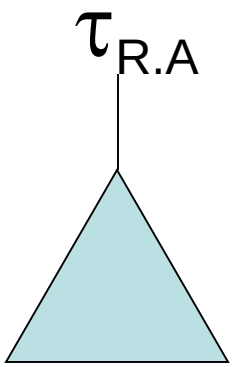



```
Open() {
  /** initialize child */
  Child.Open();
}

Close() {
  /** inform child */
  Child.Close();
}
```

```
GetNext() {
  LOOP:
      t = Child.GetNext();
      IF (t == EOT) {
        /** no more tuples */
        RETURN EOT;
      }
      ELSE IF (t.A == “c”)
        RETURN t;
  ENDLOOP:
}
```

Iterator for Sort

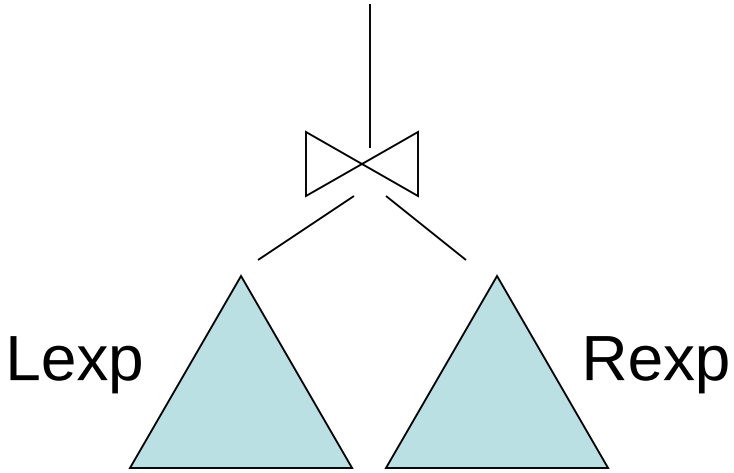



```
GetNext() {
   IF (more tuples)
      RETURN next tuple in order;
   ELSE RETURN EOT;
}
```

```
Open() {
  /** Bulk of the work is here */
  Child.Open();
  Read all tuples from Child
     and sort them
}
```

```
Close() {
  /** inform child */
  Child.Close();
}
```

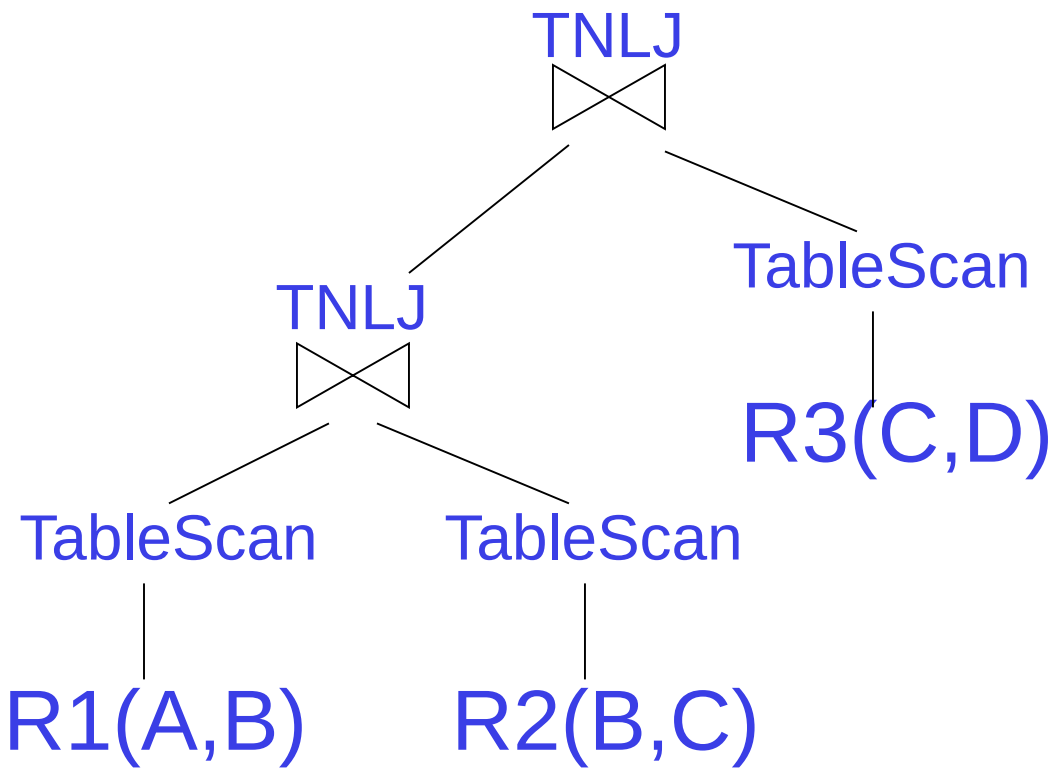
Iterator for Tuple Nested Loop Join



- TNLJ (conceptually)

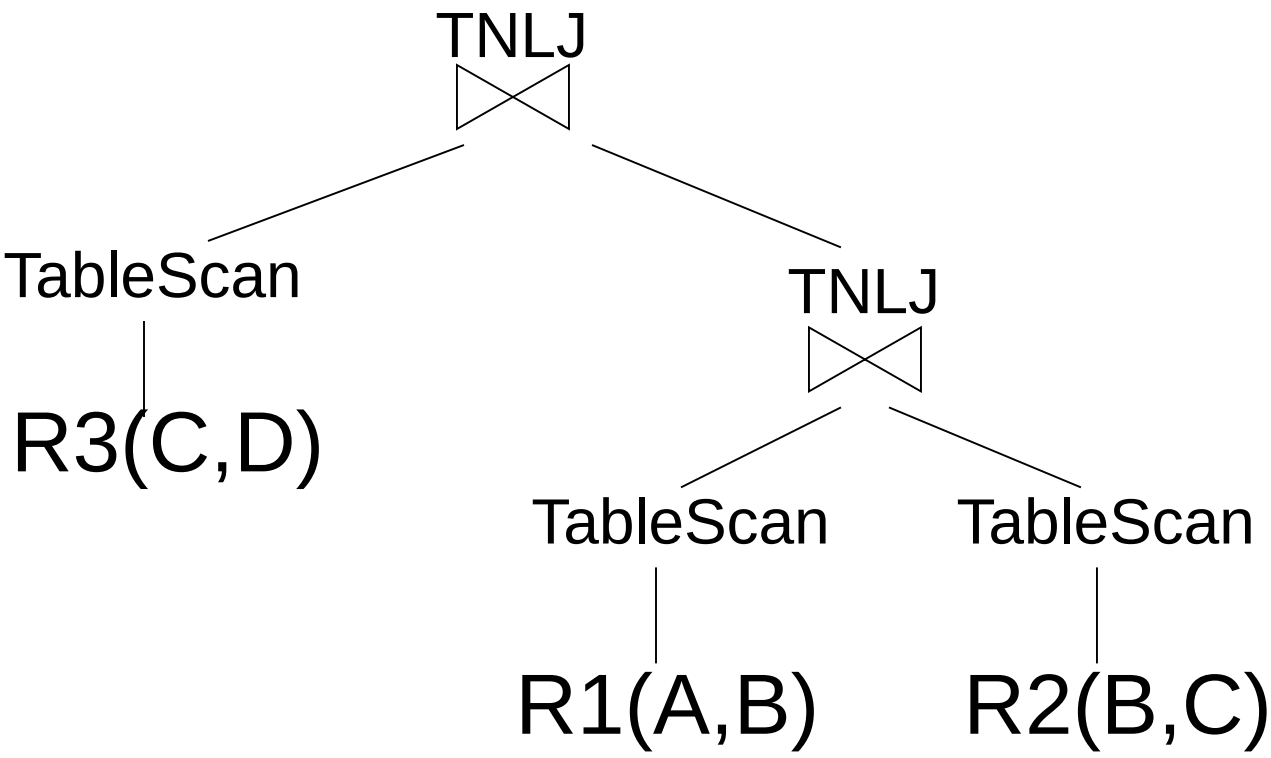
```
for each r ∈ Lexp do
    for each s ∈ Rexp do
        if Lexp.C = Rexp.C, output r,s
```

VD1: Left-Deep Plan



Question: What is the sequence of getNext() calls?

VD2: Right-Deep Plan



Question: What is the sequence of getNext() calls?

Nội dung

- Mô hình chi phí
  - Nguyên lý tối ưu
  - Giải thuật Selinger
- Triển khai các phép toán trong truy vấn
  - Materialization
  - Pipelining
- Cài đặt các phép toán
  - Scan-based
  - Sort-based
  - Using existing indexes
  - Hash-based
- Ước lượng chi phí

Nested Loop Join (NLJ)

- NLJ (conceptually)

```
for each r ∈ R1 do
    for each s ∈ R2 do
        if r.C = s.C then
            output r,s pair
```

Quan hệ được lưu trữ:

- **Theo bộ**
- **Theo từng block**

R1

B	C
a	10
a	20
b	10
d	30

⋈

C	D
10	cat
40	dog
15	bat
20	rat

R2

Cài đặt các phép toán trong truy vấn

- Duyệt toàn bộ (chẳng hạn với NLJ)
- Thi hành sau khi sắp xếp (sort-based)
- Sử dụng index đã có
- Thi hành sử dụng kỹ thuật băm (index trong quá trình thi hành)

Mô hình tính toán

- Cách tiếp cận: dựa trên việc xác định số lần đọc/ghi disk-blocks trong khi thi hành toán tử

- Đối với kế hoạch thi hành

  Cost of query plan = Sum of operator costs

- Giả thiết:
  - Bỏ qua chi phí thi hành truy vấn trong CPU
  - DBMS được cài đặt trên máy tính có 1 CPU và 1 HDD
  - Chi phí lưu kết quả được bỏ qua (độc lập với việc cài đặt toán tử)
  - Bỏ qua số lần truy xuất các khối chứa index

Tham số

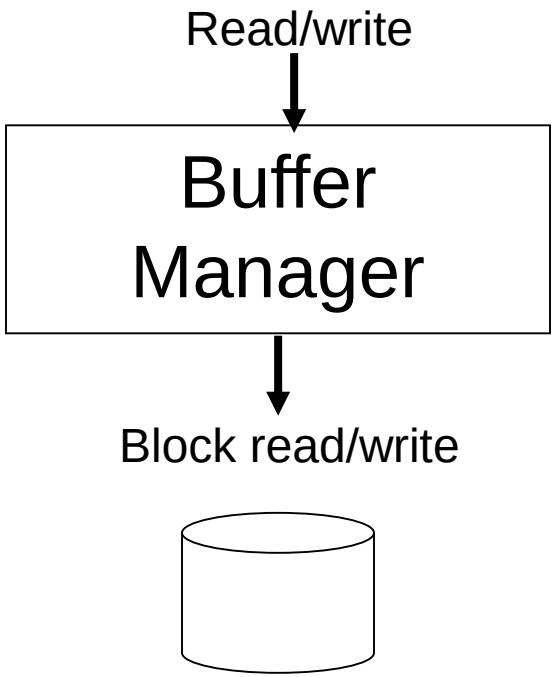
B(R) = số khối blocks mà quan hệ R được lưu trên HDD

T(R) = số bộ trong R

V(R,A) = số giá trị khác nhau của thuộc tính A trong R

M = số khối còn rảnh trong bộ nhớ buffer

Chú ý buffer trong DBMS



Sử dụng các chính sách ghi:

- Least Recently Used (LRU)
- Second-chance
- Most Recently Used (MRU)
- FIFO

- Có thể kiểm soát trực tiếp MM (chẳng hạn, không cần sử dụng VM kiểm soát với OS)

Giải thuật cài đặt

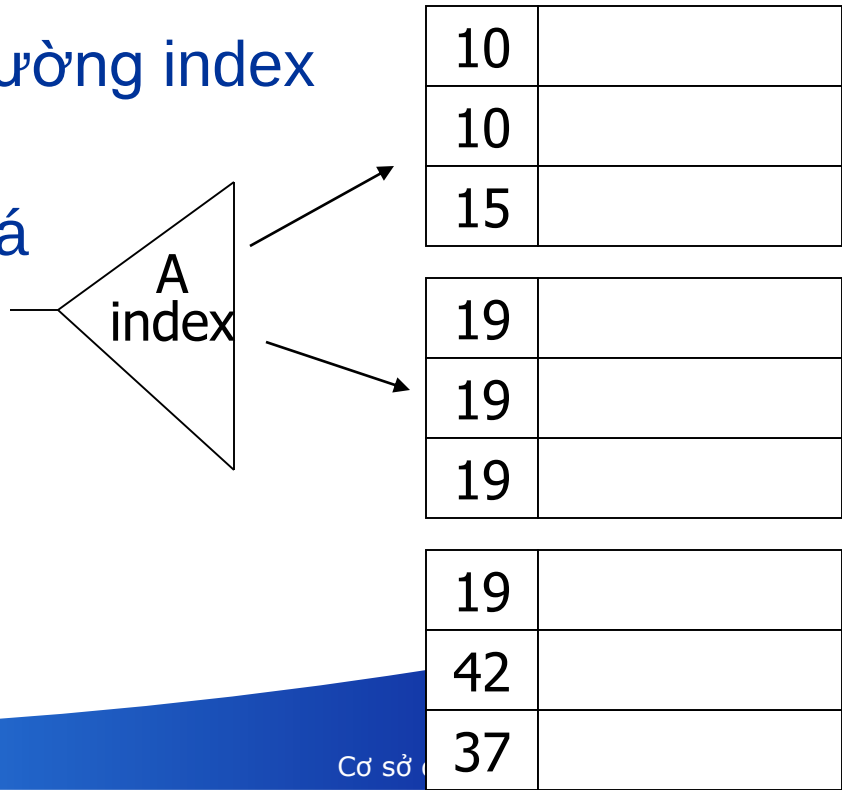
- **Chủ yếu sử dụng phương pháp**
  - Dựa trên sắp xếp các bộ
  - Sử dụng phương pháp băm – hash (index-on-fly)
  - Sử dụng chỉ mục đã xây dựng từ trước
- **Độ phức tạp của giải thuật**
  - One pass (toàn bộ dữ liệu được đưa hết vào buffer)
  - Two pass (kích thước các quan hệ lớn hơn buffer)
  - Multi pass (khi kích thước quan hệ rất lớn)
- **Phân loại toán tử**
  - Tuple-at-a-time, unary operations(σ, π)
  - Full-relation, unary operations (δ, γ)
  - Full-relation, binary operations (union, join,…)

Phân cụm dữ liệu

- Phân cụm trong mỗi quan hệ

R1 R2 R3 R4 R5 R5 R7 R8 .....

- Phân cụm index
  - Các bộ được sắp thêm theo trường index và phân thành bộ
  - Những bộ có chung giá trị khoá được gói trong blocks



Ví dụ

T(R) = 10,000

B(R) = 200

Nếu R được phân cụm ➔ số bộ/block=10.000/200=50

Giả sử V(R,A) = 40

➔Nếu I là chỉ mục phân cụm trên R.A, thì số lượng truy xuất IOs với phép $\sigma_{R.A = \text{“a”}}(R)$ là 250/50 = 5

➔ Nếu I là chỉ mục không phân cụm trên R.A, thì số lần truy xuất IOs đối với phép $\sigma_{R.A = \text{“a”}}(R)$ là **250** (> B(R))

Phân loại toán tử

	Bộ dữ liệu	Toàn quan hệ
Đơn nguyên	σ, π	δ, γ, τ
Nhị nguyên		union, join, difference

Với phép toán trên bộ dữ liệu-one pass

Với phép chọn $\sigma_C(R)$ và phép chiếu $\pi_L(R)$

- Chi phí: *Cost = B(R) or T(R)* nếu quan hệ không được phân cụm
- Không gian bộ nhớ: M ≥ 1 block
- Nguyên lý chung:
  - Mỗi lần đọc một block (hoặc một bộ nếu R không phân cụm)
  - Tiến hành xử lý với bộ trong khối đó

Với phép toán trên toàn quan hệ: one pass

- **Loại bỏ trùng lặp** δ: nguyên lý
  - Mỗi lần đọc một bộ, nếu xuất hiện lần đầu, copy bộ đó ra output.
  - Nếu đã từng xuất hiện, không đưa ra output.

- Để nâng cao hiệu năng, bộ nhớ cần sử dụng bảng băm.

- Requirement: $\approx$ $B(\delta(R)) \leq M$

Với phép toán trên toàn quan hệ: one pass...

Grouping: gom dữ liệu theo nhóm trong MM.

- Cần lưu một entry cho mỗi giá trị của thuộc tính nhóm, thông qua cấu trúc phục vụ tìm kiếm (hash table).
- Với mỗi kiểu nhóm, lưu giá trị gộp (hoặc những giá trị liên quan đến truy vấn).
  - MIN/MAX: lưu giá trị min/max value đã duyệt trong nhóm xét.
  - COUNT: đếm số lần một bộ thuộc nhóm xét.
  - SUM: cộng dồn giá trị nếu bộ đó thuộc nhóm xét.
  - AVG?

Các phép toán nhị nguyên: One pass

- Yêu cầu: với 2 quan hệ R, S, đảm bảo $\min(B(R),B(S)) \leq M$
- Cost = B(R) + B(S)
- Nếu R có số blocks lớn hơn S, với các phép "set union, set intersection, set difference, bag intersection, bag difference, cartesian product, natural join"
  - Đọc S vào MM, tiến hành các phép toán tương ứng với từng bộ/block của R.

Các phép toán nhị nguyên: Two pass

Xét giải thuật NLJ duyệt theo từng block:

- Giả sử B(*S*) ≤ B(*R*), và B(*S*) > M
- Đọc M-1 blocks của *S* vào MM, so sánh với tất cả khối của *R* (block by block)

```
FOR each chunk of M-1 blocks of S DO
    FOR each block b of R DO
        FOR each tuple t of b DO
            find the tuples of S in memory that join with t
            output the join of t with each of these tuples
```

- Cost = [B(*S*)/(M-1)] * (M-1 + B(*R*))
  = B(*S*) + B(*S*)B(*R*)/(M-1)
  ≈ B(S)*B(R)/M

Ví dụ

Nếu B(*R*) = 1000, B(*S*) = 500, M = 101

- Chi phí thực hiện NLJ :
  - Cần 5 lần lặp, mỗi lần đọc 100 blocks của S và join với 1000 blocks cua R.
  - Total time = 5*(100 + 1000) = 5500 I/O's
- Nếu đổi thứ tự *R* và *S*?
  - Cần lặp 10 lần, mỗi lần đọc 100 blocks R, join với 500 blocks của S, ➔ total =10*(100+500) = 6000 I/O's.
- Nếu sử dụng được one-pass (B(S) < M)
  - Cần 1500 I/O's

Two-pass algorithms based on sorting

- Là trường hợp đặc biệt của multi-pass algorithms

Ý tưởng chính: dựa trên những phép toán đơn nguyên trên *R* khi B(*R*) > M

- **First pass**:
  - Đọc M blocks của *R* vào MM
  - Sắp xếp nội dung các blocks đã đưa vào
  - Ghi kết quả sắp xếp vào M blocks trên thiết bị lưu trữ.
- **Second pass**: tạo kết quả cuối cùng từ những nhóm khối (trunk) đã được sắp xếp từ bước trên

Phép loại trùng lặp δ sử dụng sắp xếp

- Trong pha 2 (merging) , chúng ta không sắp xếp mà chỉ copy mỗi bộ một lần.
- Ví dụ: M=3, mỗi buffer block chứa được 2 bộ, R gồm 17 bộ sau: 2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3
- **First pass:** ta có được sorted sub-lists sau:

  1, 2, 2, 2, 2, 5

  2, 3, 4, 4, 4, 5

  1, 1, 2, 3, 5
- **Second pass:** sử dụng buffer cho mỗi sub-list.

Ví dụ...

Sub-list	In memory	Waiting on disk
R_1:	1, 2	2, 2, 2, 5
R_2:	2, 3	4, 4, 4, 5
R_3:	1, 1	2, 3, 5

Sub-list	In memory	Waiting on disk
R_1:	2	2, 2, 2, 5
R_2:	2, 3	4, 4, 4, 5
R_3:	2, 3	5

Ví dụ...

Sub-list	In memory	Waiting on disk
R_1:	5	
R_2:	3	4, 4, 4, 5
R_3:	3	5

Sub-list	In memory	Waiting on disk
R_1:	5	
R_2:	4, 4	4, 5
R_3:	5	

Ví dụ...

Sub-list	In memory	Waiting on disk
R_1:	5	
R_2:	5	
R_3:	5	

Sub-list	In memory	Waiting on disk
R_1:		
R_2:		
R_3:		

Analysis of δ(R)

- Chi phí I/O cho phép δ(R) : 2B(R) để tạo sorted sublists, B(R) để đọc sublists trong pha 2 ➔ **3B(R)**

- Chú ý tương quan *R* và M:
  - Với M buffers, chỉ được phép có tối đa M sublists ➔ B(R)/M ≤ M (B(R)/M is the number of sublists)

    ➔ **B(*R*) ≤ M²**

  - Để tính δ(R) cần tối thiểu **sqrt(B(*R*))** blocks MM.

Sort-based ∪, ∩, -

Ví dụ: **set union**.

Tương tự với sort based intersection & difference

- Tạo sorted sublists của *R* **và** *S*
- Sử dụng mỗi buffer cho các sorted sublists của *R* và *S*
- Xuất mỗi bộ một lần

- Chi phí: cost = **3(B(R) + B(S))** disk I/O's
- Ràng buộc: **B(R) + B(S) ≤ M²**

Join

- Vấn đề đặt ra: số lượng bộ nối từ 2 quan hệ có thể lớn hơn rất nhiều so với M

- Giải pháp?
  - Sử dụng tối đa output buffers.
  - Tối thiểu hoá lượng sorted sublists.

Simple sort-based join

- Với $R(X,Y) \bowtie S(Y,Z)$ sử dụng M buffers :
- **Sort phase:** sắp xếp cả R và S trên thuộc tính Y
- **Merge phase:** Sử dụng 2 input buffers cho *R* và *S*.
  - Đọc bộ t có Y bé nhất vào buffer của R, block đầu của S vào buffer S
  - Nếu t không nối được với bộ đầu trong buffer S, bỏ t
  - Nếu không, đọc tất cả bộ của R có cung giá trị Y như t, đưa vào M-2 buffers còn lại, nối với S
  - Lặp lại quá trình đó cho đến khi vét hết bộ trong R.

Ví dụ

Xét B(R) = 1000, B(S) = 500, M= 101

- Sắp xếp R và S, cần 4*(B(R) + B(S)) I/O's (do sau khi sắp xếp phải lưu lại trên thiết bị lưu trữ)

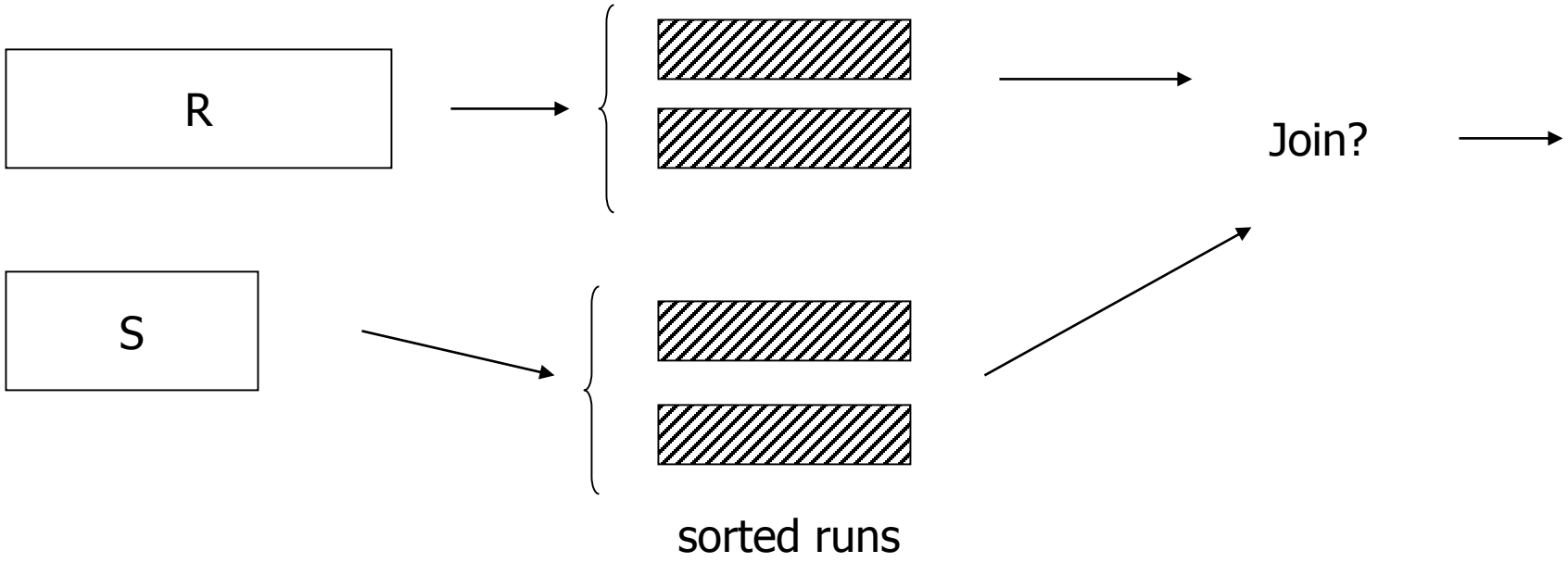
- Với pha nối, tổng số truy cập I/O's = B(R) + B(S)

➔Total I/O's = 5*(B(R) + B(S)) = 7500

- Yêu cầu về MM:
  - Để đảm bảo sắp xếp được, B(R) ≤ M^2 và B(S) ≤ M^2

Cải thiện sort-join?

- Liệu có cần sắp xếp hết quan hệ?



- Disk I/O: 3*(B(R) + B(S))
- Yêu cầu: B(R) + B(S) ≤ M²

Summary: sort-based algorithms

Operators	Approx. M required	Disk I/O
γ, δ	Sqrt(B)	3B
$\cup$, $\cap$, -	Sqrt(B(R) + B(S))	3(B(R)+B(S))
$\triangleright\triangleleft$	Sqrt(max(B(R),B(S)))	5(B(R)+B(S))
$\triangleright\triangleleft$	Sqrt(B(R)+B(S))	3(B(R)+B(S))

Two-pass algorithms based on hashing

Main idea:

- Thay vì sử dụng sorted sublists, tạo các **partitions**, dựa trên kỹ thuật băm (hashing)
  - Những bộ thoả mãn yêu cầu phép toán sẽ được nhóm trong cùng partition - bucket
- Pha hai sẽ tạo kết quả từ các partitions sử dụng one pass algorithms.

Creating partitions

- Partitions (buckets) được tạo dựa trên tất cả các thuộc tính của quan hệ, ngoại trừ các phép toán liên quan đến grouping và join thì tạo trên chính thuộc tính liên quan.

```
Initialize M-1 buckets using M-1 empty buffers;
FOR each block b of relation R DO
    read block b into the M-th buffer;
    FOR each tuple t in b DO
        IF the buffer for bucket h(t) has no room for t THEN
            copy the buffer to disk;
            initialize a new empty block in that buffer;
            copy t to the buffer for bucket h(t);
        ENDIF;
    ENDFOR;
ENDFOR;
FOR each bucket DO
    IF the buffer for this bucket is not empty THEN
    write the buffer to disk;
ENDFOR;
```

Phép loại lặp dựa trên kỹ thuật băm

- Pass 1: tạo các buckets chứa giá trị băm của tất cả các thuộc tính
- Pass 2: với mỗi bucket, sử dụng giải thuật one-pass để loại bỏ lặp.

- Chi phí: Cost = 3B(R) disk I/O's

- Yêu cầu: $B(R) \leq M*(M-1)$ hay $B(R) < M^2$

  vì kích thước bucket gần bằng (B(R)/(M-1)

Hash-based grouping and aggregation

- Pass 1: tạo partitions dựa trên băm thuộc tính **grouping**
- Pass 2: với mỗi partition, sử dụng kỹ thuật one-pass.

- Chi phí: Cost = 3B(R), yêu cầu: B(R) ≤ M^2

Hash-based set union

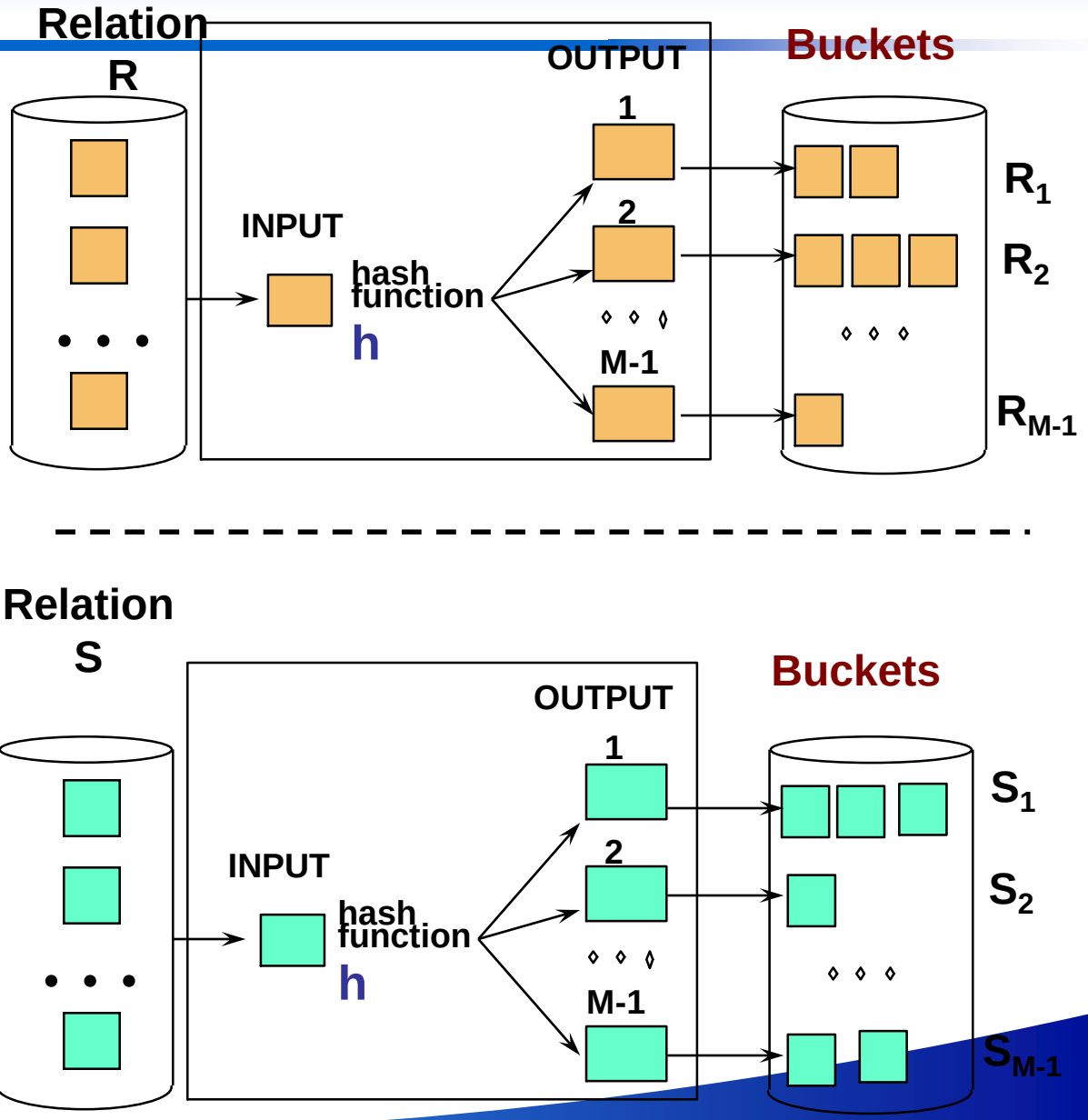
- Pass 1: tạo các partitions $R_1,\ldots,R_{M-1}$ của R, và $S_1,\ldots,S_{M-1}$ củaS (với cùng hàm hash)
- Pass 2: với mỗi cặp R_i, S_i, tính $R_i \cup S_i$ sử dụng kỹ thuật **one-pass**.
- Cost: **3(B(R) + B(S))**
- Yêu cầu: **min(B(R),B(S)) ≤ M^2**
- Tương tự với các phép intersection và difference

Partitioned hash-join

- Pass 1: tạo các partitions R_1, ..,R_{M-1} của R, và S_1, ..,S_{M-1} của S, dựa trên thuộc tính nối (sử dụng chung hàm băm)

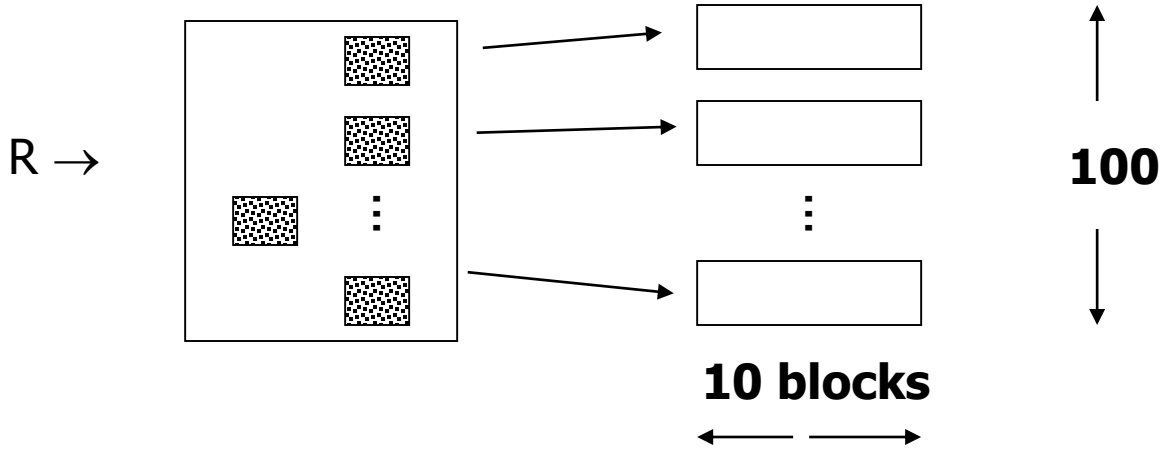
- Pass 2: với mỗi cặp R_i, S_i tính $R_i \triangleright\triangleleft S_i$ sử dụng kỹ thuật **one-pass**.

Partitioned Hash-Join



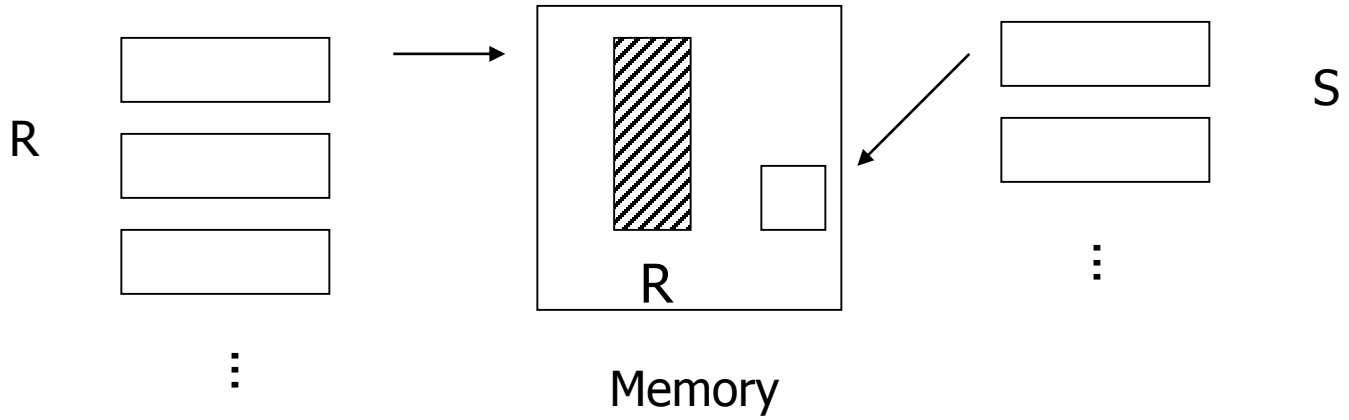
Ví dụ

- B(R) = 1000 blocks, B(S) = 500 blocks
- M = 101 blocks
- R ▷◁ S theo C
- Đầu tin, sử dụng 100 buckets
  - Read R
  - Hash
  - Write buckets



- Tương tự cho S

- Read one R bucket
- Build memory hash table
- Read corresponding S bucket block by block.



Cost

- "Bucketize:"
  - Read + write R
  - Read + write S
- Join
  - Read R
  - Read S

Total cost = 3*[1000+500] = 4500

Nhìn chung:

Chi phí Cost = **3(B(R) + B(S))**

Yêu cầu: **min(B(R),B(S)) ≤ M²**

Summary of hash-based algorithms

Operators	Approx. M required	Disk I/O
γ, δ	Sqrt(B)	3B
$\cup$, $\cap$, -	Sqrt(B(S))	3(B(R)+B(S))
$\triangleright\triangleleft$	Sqrt(B(S))	3(B(R)+B(S))

So sánh với sort-based algo.

Operators	Approx. M required	Disk I/O
γ, δ	Sqrt(B)	3B
$\cup$, $\cap$, -	Sqrt(B(R) + B(S))	3(B(R)+B(S))
$\triangleright\triangleleft$	Sqrt(max(B(R),B(S)))	5(B(R)+B(S))
$\triangleright\triangleleft$	Sqrt(B(R)+B(S))	3(B(R)+B(S))

Index-based algorithms: selection

- Thi hành $\sigma_{a=v}(R)$ khi sử dụng chỉ mục *a*

- **Chi phí Cost** = chi phí tìm trong index (thường bỏ qua) và
  - Nếu index được phân cụm: B(R)/V(R,*a*) *I/O's*
  - Nếu không phân cụm: T(R)/V(R,a) *I/O's*

Ví dụ

$\sigma_{a=v}(R)$, có B(R) = 1000, T(R) =20,000

- R được phân cụm và không có index cho a
  →1000 disk I/O's
- R có clustering index tại a, và V(R,a) = 100
  →10 I/O's
- R có không non-clustering index tại a, V(R,a) = 100
  →20,000/100 = 200 disk I/O's
- Nếu V(R,a) = 20,000 (a là khoá) → just 1 I/O

Index joins

- Giử sử với phép nối $R(X,Y) \triangleright\!\triangleleft S(Y,Z)$ và S được index theo Y

- **Algorithm**:

  Với mỗi bộ *t* của *R*, tìm các bộ trong S có giá trị *t*[*Y*] và nối với *t*.

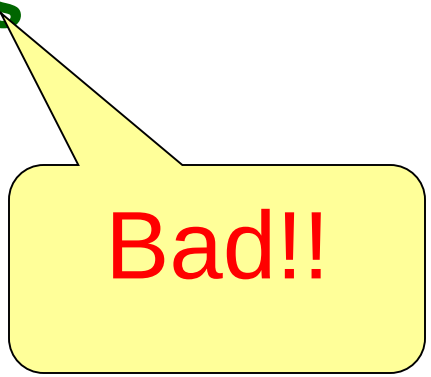
- Chi phí Cost = B(*R*) (to read *R*) và chi phí nối
  - Trung bình, mỗi bộ của *R* nối với T(*S*)/*V*(*S*,*Y*) bộ của *S*.
  - Nếu *S* có non-clustered index theo *Y* , phí nối = T(*R*)T(*S*)/*V*(*S*,*Y*)
  - Nếu *S* có clustered index theo*Y*, phí nối = T(*R*)B(*S*)/*V*(*S*,*Y*)

Ví dụ

- T(R) = 10,000, B(R) = 1000 ;
- T(S) = 5000, B(S) = 500, V(S,Y) = 100

- Phép nối R(X,Y) ⋈ S(Y,Z) có clustering index theo Y trên S có chi phí = 1000 + 10,000*(500/100) =

51,000 I/O's



Tuy nhiên, thực tế...

- Giả sử có hai quan hệ:

  StarsIn(title, year, starName)

  MovieStar(*name*, address, gender, birthdate)

trong đó MovieStar được indexing theo *name*

- Với truy vấn SQL :

```
SELECT birthdate
FROM StarsIn, MovieStar
WHERE title = 'King Kong' AND starName = name;
```

Thực tế…

- Đầu tiên, thực hiện phép chọn trong *StarsIn* thoả mãn *title='King Kong'*. Giả sử có được 10 tuples.

- Sử dụng *name* trong các bộ tìm được để nối với *MovieStar*, nếu không tính các ngôi sao trùng tên, ta sẽ có V(*MovieStar, name*) = T(*MovieStar*)

- Cuối cùng, I/O cost = B(*StarsIn*) + T($\sigma_{name='King\ Kong'}$(*StarsIn*))

Joins using sorted indexes

Xét phép nối $R(X,Y) \triangleright\triangleleft S(Y,Z)$

- Nếu *S* có B-tree index theo *Y*,
  - Tạo sorted sublists của of *R* và
  - Nối theo kiểu sắp xếp, lấy các bộ trong S theo thứ tự đã index
- Nếu cả R và S đều có B-tree index theo *Y*, sử dụng *zigzag-join*.

Ví dụ

- T(R) = 10,000, B(R) = 1000,
- T(S) = 5000, B(S) = 500, V(S,Y) = 100, B-Tree index theo *Y*
- Cả hai quan hệ và index đều được phân cụm. M = 101 buffers

Joins using sorted indexes

- Tạo 10 sorted sublists của *R*, *c*ost = 2B(*R*)
- Sử dụng 10 buffers cho sublists của *R*, 1 buffer cho S (index)
- Nối các bộ từ các buffers trên

- Total cost = 2B(R) + B(R) + B(S) + index lookup =

  2000 + 1000 + 500 + index lookup = 3500 + index lookup

Zigzag Join

- Giả sử cả R và S đều có B-Tree indexes theo Y.
- Các bộ trong R với những giá trị Y không xuất hiện trong S sẽ bị loại bỏ, tương tự với S.

Ví dụ:

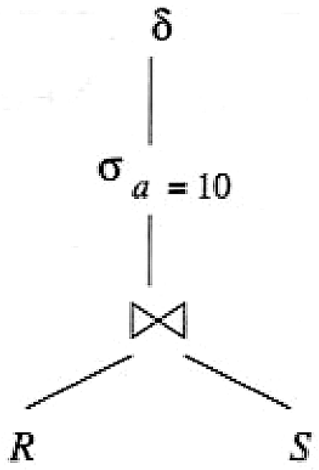
- Giả sử R có Y: 1,3,4,4,4,5,6; S có Y là: 2,2,4,4,6,7
- Bắt đầu với cặp 1 và 2.
- Vì 1<2 ➔ bỏ qua 1 trong R, xét tiếp cặp 3, 2
- Vì 2<3 ➔ bỏ qua 2 trong S.
- Tương tự, bỏ qua 3 trong R, nối các bộ 4 và 6
- …
- Chi phí: B(R) + B(S)

Nội dung

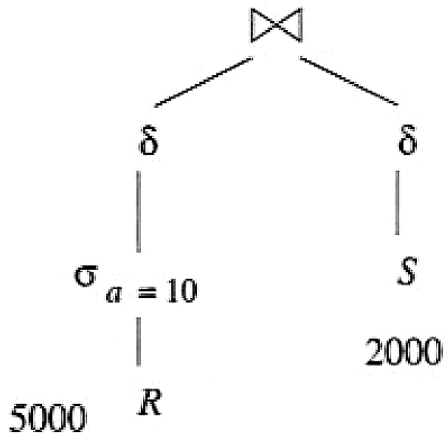
- Mô hình chi phí
  - Nguyên lý tối ưu
  - Giải thuật Selinger
- Triển khai các phép toán trong truy vấn
  - Materialization
  - Pipelining
- Cài đặt các phép toán
  - Scan-based
  - Sort-based
  - Using existing indexes
  - Hash-based
- Ước lượng chi phí

Bài toán đặt ra

$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$



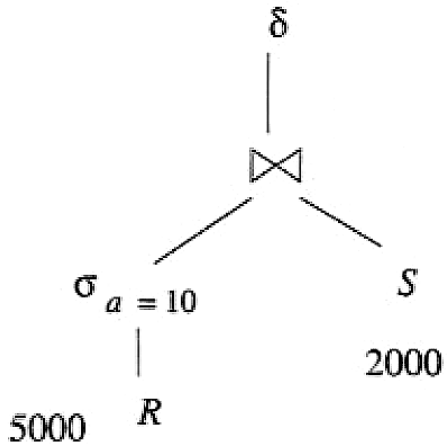
Initial logical query plan



(a)

2 ứng viên sau đều được coi như *best logical query plan*.

Plan nào sẽ được chọn?



(b)

Ước lượng chi phí thi hành

- Không thể thi hành query để đánh giá cost
- Không thể xác định chính xác ngay được những bộ dữ liệu trung gian

} Cần ước lượng chi phí thi hành mỗi truy vấn

Khi ước lượng, cần chú trọng:

1. Ước lượng được chi phí chính xác nhất có thể
2. Dễ thực hiện được việc ước lượng

Projection

- Kích thước dữ liệu kết quả của phép chiếu luôn có thể xác định chính xác.

1. Số bộ trong phép Projection π luôn được giữ nguyên.
2. Kết quả thường có kính thước bé hơn.

Selection

Với $S = \sigma_{A=c}(R)$, kết quả phép chọn có số bộ là **T(S) = T(R) / V(R,A)**

Với $S = \sigma_{A<c}(R)$, thường số bộ được tính T(S) = T(R)/2, nhưng tổng quát là **T(R)/3**

Với $S = \sigma_{A\neq c}(R)$, số bộ kết quả được ước lượng

T(S) = T(R) * [(V(R,A)-1)/V(R,A)], hay

T(S) = T(R)

Selection ...

Với $S = \sigma_{C \text{ AND } D}(R) = \sigma_C(\sigma_D(R))$, ta ước lượng trước $T(\sigma_D(R))$ và sau đó ước lượng T(S).

Ví dụ:

$S = \sigma_{a=10 \text{ AND } b<20}(R)$

T(R) = 10,000, V(R,a) = 50

T(S) = (1/50)* (1/3) * T(R) = 67

Tuy nhiên: ước lượng $\sigma_{a=10 \text{ AND } a>20}(R)$?

Selection ...

- Với $S = \sigma_{C \text{ OR } D}(R)$, khi đó kết quả có thể được ước lượng phí đơn giản là $T(S) = T(\sigma_C(R)) + T(\sigma_D(R))$.
  - **Tuy nhiên, có thể dẫn đến trường hợp** $T(S)>T(R)$!

Số bộ kết quả có thể ước lượng chi tiết hơn. Giả sử $T(R)=n$, $m_1 = T(\sigma_C(R))$, và $m_2 = T(\sigma_D(R))$. Khi đó:

$T(S) = n(1-(1-m_1/n)(1-m_2/n))$

- **Ví dụ**: $S = \sigma_{a=10 \text{ OR } b<20}(R)$. $T(R) = 10000$, $V(R,a) =50$
  - Nếu sử dụng ước lương đơn giản: $T(S) = 3533$
  - Nếu sử dụng ước lượng chi tiết: $T(S) = 3466$

Natural Join R(X,Y) ▷◁ S(Y,Z)

Việc ước lượng phép toán này tương đối phức tạp, tuỳ thuộc vào dữ liệu thực của R và S

- Có thể kết quả không có bộ nào

  $T(R \bowtie S) = 0$

- Hoặc tất cả các bộ, chẳng hạn nếu R.Y=S.Y = *a* thì

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

Giả thiết

A1: Containment of value sets

- Nếu V(R,Y) ⊆ V(S,Y), thì tất cả giá trị của Y trong R được coi như có trong **Y** của S
- Chẳng hạn Y là khoá ngoại của R, và là khoá trong S

A2: Preservation of set values

- Nếu *A* là thuộc tính của R nhưng không phải của S, thì
  $V(R \triangleright\!\triangleleft S, A) = V(R, A)$
- Chỉ đúng khi Y là khoá ngoại của R, và là khoá trong S

Natural Join…

- Giả sử Y là thuộc tính đơn, số bộ kết quả của $T(R \bowtie S)$ là

Nếu $V(R,Y) \leq V(S,Y)$, theo A1, ta suy ra $r \in R$ sẽ khớp được với vài bộ của S với xác xuất là $T(S)/V(S,Y)$

➔ $T(R \bowtie S) = T(R)*T(S)/V(S,Y)$

- Tương tự, nếu $V(S,Y) \leq V(R,Y)$, ta có thể ước lượng được $T(R \bowtie S) = T(R)*T(S)/V(R,Y)$.

- Từ đó **$T(R \bowtie S) = T(R)*T(S)/\max\{V(R,Y),V(S,Y)\}$**

Ví dụ

- **Xét**

  R(a,b), T(R)=1000, V(R,b)=20
  S(b,c), T(S)=2000, V(S,b)=50,
  V(S,c)=100
  U(c,d), T(U)=5000, V(U,c)=500

với phép nối **R ▷◁ S ▷◁ U**

- TH1:

T(R ▷◁ S) = 1000*2000 / 50 = 40,000
T((R ▷◁ S) ▷◁ U) = 40000 * 5000 / 500 = **400,000**

- TH2:

T(S ▷◁ U) = 20,000
T(R ▷◁ (S ▷◁ U)) = 1000*20000 / 50 = **400,000**

Chú ý:

1. **Giá trị ước lượng số bộ kết quả cuối cùng phải độc lập so với thứ tự nối**
2. **Kết quả trung gian có thể có số lượng bộ khác nhau**

Nối tự nhiên sử dụng nhiều thuộc tính

- Xét phép nối $R(x,\mathbf{y_1},\mathbf{y_2}) \triangleright\triangleleft S(\mathbf{y_1},\mathbf{y_2},z)$. Khi đó

$$T(R \triangleright\triangleleft S) = T(R)*T(S)/ (m_1*m_2),$$

Trong đó $m_1 = \max\{V(R,y_1),V(S,y_1)\}$

$m_2 = \max\{V(R,y_2),V(S,y_2)\}$

- Thật vậy, nếu $r \in R$, $s \in S$ thì xác xuất r và s có thể nối được với nhau là $\mathbf{1/\max\{V(R,y_1),V(S,y_1)\} = 1/m_1}$
- Vì y1 và y2 độc lập với nhau, thế nên $T(R \triangleright\triangleleft S) = T(R)*T(S)/(m_1*m_2)$

Ví dụ:

T(R)=1000, V(R,b)=20, V(R,c)=100

T(S)=2000, V(S,d)=50, V(S,e)=50

$R(a,b,c) \triangleright\triangleleft_{R.b=S.d \text{ AND } R.c=S.e} S(d,e,f)$

$T(R \triangleright\triangleleft S) = (1000*2000)/(50*100)=400$

Với các phép toán khác

Nhân chéo Cartesian : $T(R \times S) = T(R) * T(S)$

Bag Union: tổng các bộ thành phần

Set union: tổng các bộ quan hệ có số bộ lớn hơn với nửa số bộ của quan hệ còn lại.

Intersection: nửa số bộ của quan hệ có số lượng bé hơn

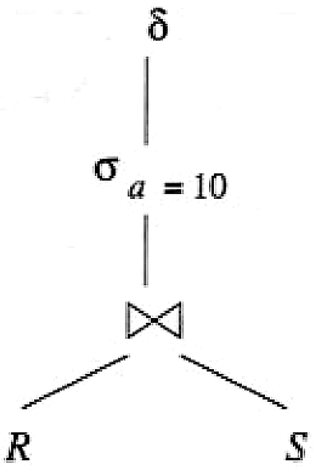
Difference: T(R-S) = T(R) - 1/2*T(S)

Duplicate elimination $\delta(a_1,...,a_n)$ trong R:

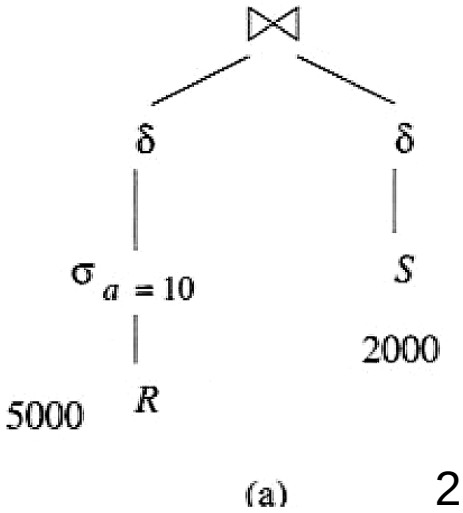
$T(\delta(R))$= V(R,[a1...an]) ~= *min[V(R,a1)*...*V(R,an), 1/2*T(R)]*

Chọn kế hoạch

$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$

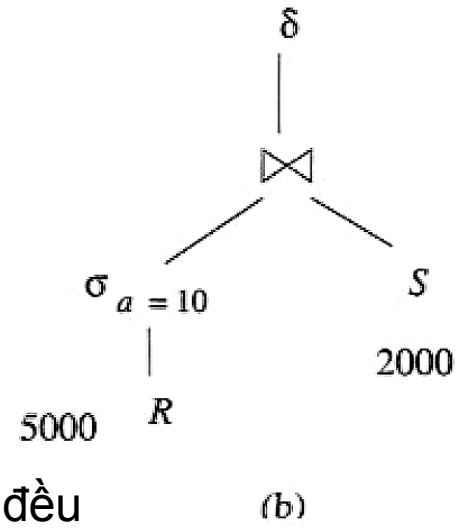


Initial logical query plan



2 ứng viên sau đều được coi như *best logical query plan*.

Plan nào sẽ được chọn?



Ước lượng số bộ

Plan a

T($\sigma_{a=10}(R)$) =

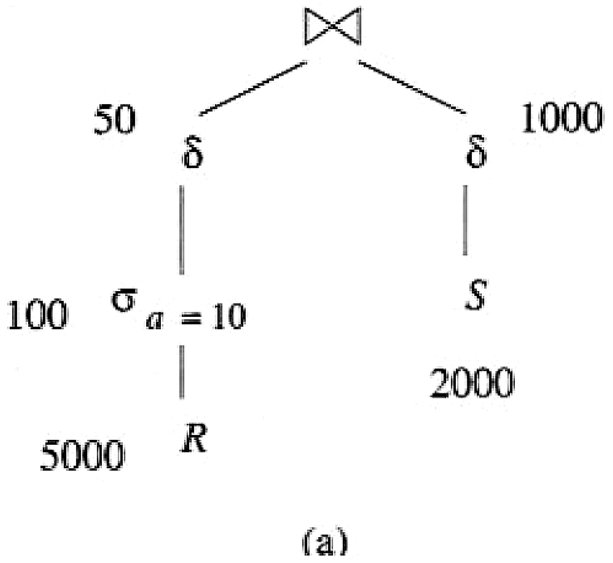
5000/50 = 100

T($\delta(\sigma_{a=10}(R))$) =

min{1*100, 100/2} = 50

T($\delta(S)$) =

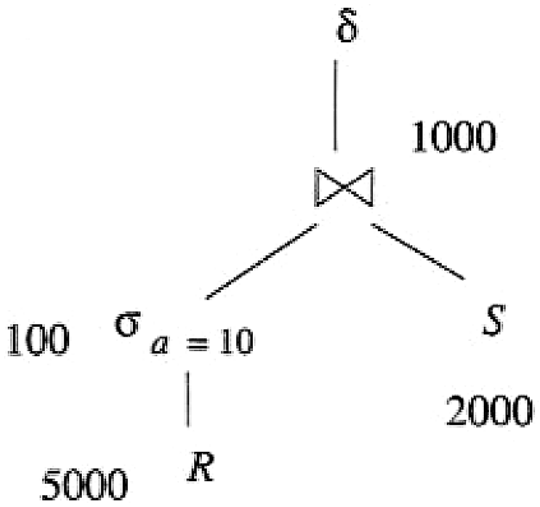
min{200*100, 2000/2} = 1000



(a)

$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$

Ước lượng số bộ...



(b)

Plan b

T($\sigma_{a=10}(R)$) =
5000/50 = 100

T($\sigma_{a=10}(R) \rhd\lhd S$) =
100*2000/200 = 1000

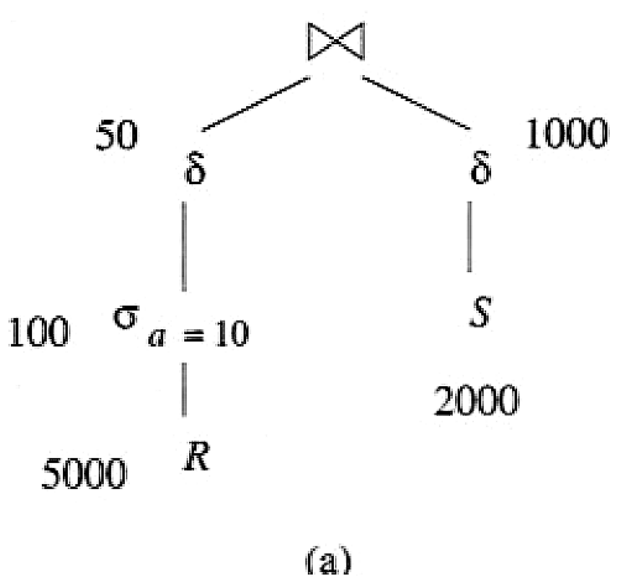
$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$

Ước lượng số bộ...

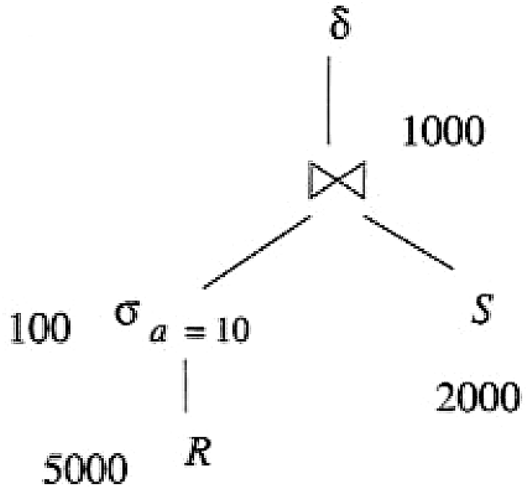
Từ đó, số lượng bộ trung gian Ta = 1150 và Tb = 1100

Số lượng bộ ước lượng được của mỗi kế hoạch là:

Tpa = 1400 và Tpa = 1600



(a)



(b)

Vậy

- plan có chi phí ước lượng tốt nhất là plan b (xét số bộ trung gian)
- Plan a có chi phí tốt hơn nếu xét tổng thể kế hoạch

Thứ tự nối

- Với tối ưu hoá dựa theo ước lượng chi phí, việc xác định thứ tự các phép nối tự nhiên rất quan trọng

- **Xét**

  R(a,b), T(R)=1000, V(R,b)=20
  S(b,c), T(S)=2000, V(S,b)=50, V(S,c)=100
  U(c,d), T(U)=5000, V(U,c)=500

với phép nối **R** ▷◁ **S** ▷◁ **U**

- TH1:

T(R ▷◁ S) = 1000*2000 / 50 = 40,000
T((R ▷◁ S) ▷◁ U) = 40000 * 5000 / 500 = **400,000**

- TH2:

T(S ▷◁ U) = 20,000
T(R ▷◁ (S ▷◁ U)) = 1000*20000 / 50 = **400,000**

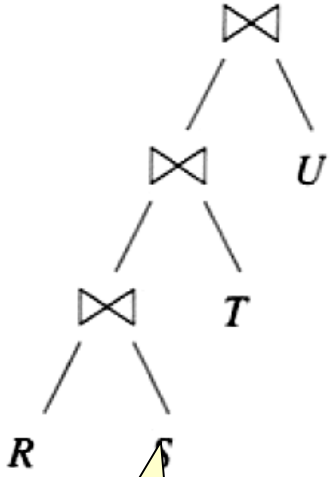
Chú ý:
1. **Giá trị ước lượng số bộ kết quả cuối cùng phải độc lập so với thứ tự nối**
2. **Kết quả trung gian có thể có số lượng bộ khác nhau**

Nối hai quan hệ

- Trong một số giải thuật nối, thứ tự phép nối hai quan hệ cũng ảnh hưởng đến chi phí thi hành phép nối.
  - Giải thuật one-pass join sẽ luôn để quan hệ có kích thước bé hơn sang trái và sẽ được đọc vào bộ nhớ trước.
    - Quan hệ trái (bé hơn) được gọi là quan hệ cơ sở (*build relation)*.
    - Quan hệ phải, được gọi là quan hệ dò (*probe relation)*, được đọc mỗi lần từng block và từng bộ sẽ được nối với build relation trong bộ nhớ.
  - Nested-Loop join luôn xem quan hệ trái được xét ở vòng lặp ngoài.
  - Index-join luôn xem quan hệ phải đã được đánh chỉ mục.

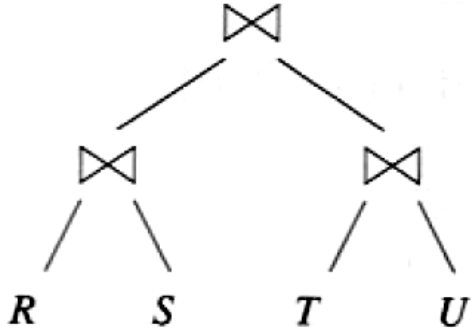
Cây nối quan hệ

- Với phép nối nhiều quan hệ, có rất nhiều cách nối khác nhau
- Các phép nối có thể được minh hoạ theo cấu trúc cây



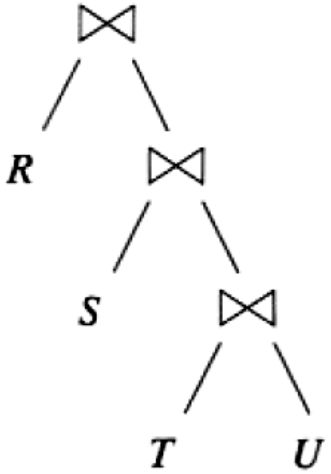
(a)

left-deep tree
All right children are leaves.



(b)

bushy tree



(c)

righ-deep tree
All left children are leaves.

Left-Deep Join Trees

Cấu trúc được sử dụng vì với phép nối n quan hệ

1. Số lượng cây LDT có thể lớn, nhưng không lớn bằng số cây có thể có.
   - Số cây LDT = n!, trong khi số cây có thể có $T(n) = \Sigma_{i=1\ldots n-1} T(i)T(n - i)$
2. LDT cho phép cài đặt hiệu quả các giải thuật nối thông dụng - nested-loop joins và one-pass joins.
   - One-pass algorithm cần MM cho 2 quan hệ trung gian tại mỗi thời điểm.
   - Quan hệ trái (build relation) nằm trong MM.
   - Để tính R⋈S, ta cần giữ R và kết quả tính trong MM ➔ cần buffer = B(R) + B(R⋈S).

Ước lượng nối nhiều quan hệ

- Giả sử cần nối n quan hệ $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. Ta cần xây dựng bảng ước lượng chi phí của n! LDT để chọn LDT tốt nhất

Ví dụ:

R(a,b) V(R,a) = 100 V(R,b) = 200	S(b,c) V(S,b) = 100 V(S,c) = 500
T(c,d) V(T,c) = 20 V(T,d) = 50	U(d,a) V(U,a) = 50 V(U,d) = 1000

Singleton Sets				
	{R}	{S}	{T}	{U}
Size	1000	1000	1000	1000
Cost	0	0	0	0
Best Plan	R	S	T	U

Pairs of relations						
	{R,S}	{R,T}	{R,U}	{S,T}	{S,U}	{T,U}
Size	5000	1M	10,000	2000	1M	1000
Cost	0	0	0	0	0	0
Best Plan	R $\bowtie$ S	R $\bowtie$ T	R $\bowtie$ U	S $\bowtie$ T	S $\bowtie$ U	T $\bowtie$ U

Chú ý: size được xem như số bộ của quan hệ; cost là số bộ trung gian được sinh ra

Nối nhiều quan hệ...

Pairs of relations						
	{R,S}	{R,T}	{R,U}	{S,T}	{S,U}	{T,U}
Size	5000	1M	10,000	2000	1M	1000
Cost	0	0	0	0	0	0
Best Plan	R ⋈ S	R ⋈ T	R ⋈ U	S ⋈ T	S ⋈ U	T ⋈ U

Triples of Relations				
	{R,S,T}	{R,S,U}	{R,T,U}	{S,T,U}
Size	10,000	50,000	10,000	2000
Cost	2000	5000	1000	1000
Best Plan	(S ⋈ T) ⋈ R	(R ⋈ S) ⋈ U	(T ⋈ U) ⋈ R	(T ⋈ U) ⋈ S

Nối nhiều quan hệ...

Grouping	Cost
((S ⋈ T) ⋈ R) ⋈ U	12,000
((R ⋈ S) ⋈ U) ⋈ T	55,000
((T ⋈ U) ⋈ R) ⋈ S	11,000
((T ⋈ U) ⋈ S) ⋈ R	**3,000**
(T ⋈ U) ⋈ (R ⋈ S)	*6,000*
(R ⋈ T) ⋈ (S ⋈ U)	*2,000,000*
(S ⋈ T) ⋈ (R ⋈ U)	*12,000*

Tổng kết

- Cần chú ý hai phương pháp triển khai phép toán trong truy vấn:
  - Materialization
  - Pipelining
- Với mỗi phép toán trong truy vấn, có thể sử dụng nhiều giải thuật để thi hành
  - Scan-based
  - Sort-based
  - Using existing indexes
  - Hash-based
- Mỗi kế hoạch thi hành truy vấn người dùng cần được ước lượng chi phí, tìm plan có chi phí tốt nhất có thể.

Bài tập

Consider a relation R with attributes A and B with the following characteristics:

- 5,000 tuples with 10 tuples per page
- A 2-level B+ tree index on attribute A with up to 100 index entries per page
- Attribute A is a candidate key of R
- The values that the attribute A takes in relation R are uniformly distributed in the range 1 to 100,000.

Answer the following questions

(a) Assuming that the aforesaid index on A is unclustered, estimate the number of page fetches needed to compute the query $\sigma_{A>1000 \text{ AND } A<6000}(R)$. Explain your reasoning.

(b) What would be the cost if the above index were clustered? Again, explain your reasoning.

Suppose we wish to compute the expression

$$\tau_b\big(R(a,b) \bowtie S(b,c) \bowtie T(c,d)\big)$$

That is, we join the three relations and produce the result sorted on attribute b. Let us make the simplifying assumptions:

i. We shall not "join" R and T first, because that is a product.

ii. Any other join can be performed with a two-pass sort-join or hash-join, but in no other way.

iii. Any relation, or the result of any expression, can be sorted by a two-phase, multiway merge-sort, but in no other way.

iv. The result of the first join will be passed as an argument to the last join one block at a time and not stored temporarily on disk.

v. Each relation occupies 1000 blocks, and the result of either join of two relations occupies 5000 blocks.

Which query plan uses the fewest disk I/Os?