CompSci 516
Data Intensive Computing Systems

Lecture 11
Query Optimization

Instructor: Sudeepa Roy

Announcements

- HW2 has been posted on sakai
  - Due on Oct 18, 11 pm
  - Start early!
  - Your AWS account should have $100 credit
  - ALWAYS Remember to stop instance!
- Midterm next week
  - Wednesday, Oct 11, in class
  - Closed book, closed electronic devices
  - Everything until and including Lecture 12

Reading Material

- [RG]
  - Query optimization: Chapter 15 (overview only)
- [GUW]
  - Chapter 16.2-16.7

- Original paper by Selinger et al. :
  - P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. *Access Path Selection in a Relational Database Management System*

    Proceedings of ACM SIGMOD, 1979. Pages 22-34
  - No need to understand the whole paper, but take a look at the example (link on the course webpage)

Acknowledgement:

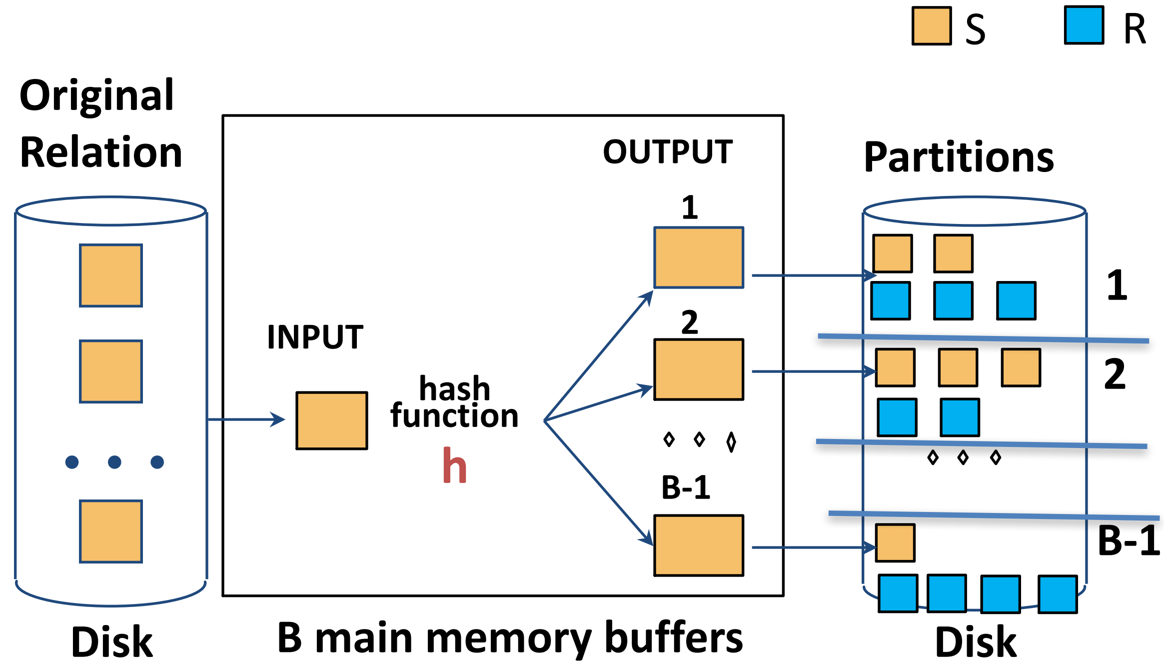
- The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
- Some of the following slides have been created by adapting slides by Profs. Shivnath Babu and Magda Balazinska

Query Evaluation and Operator Algorithm

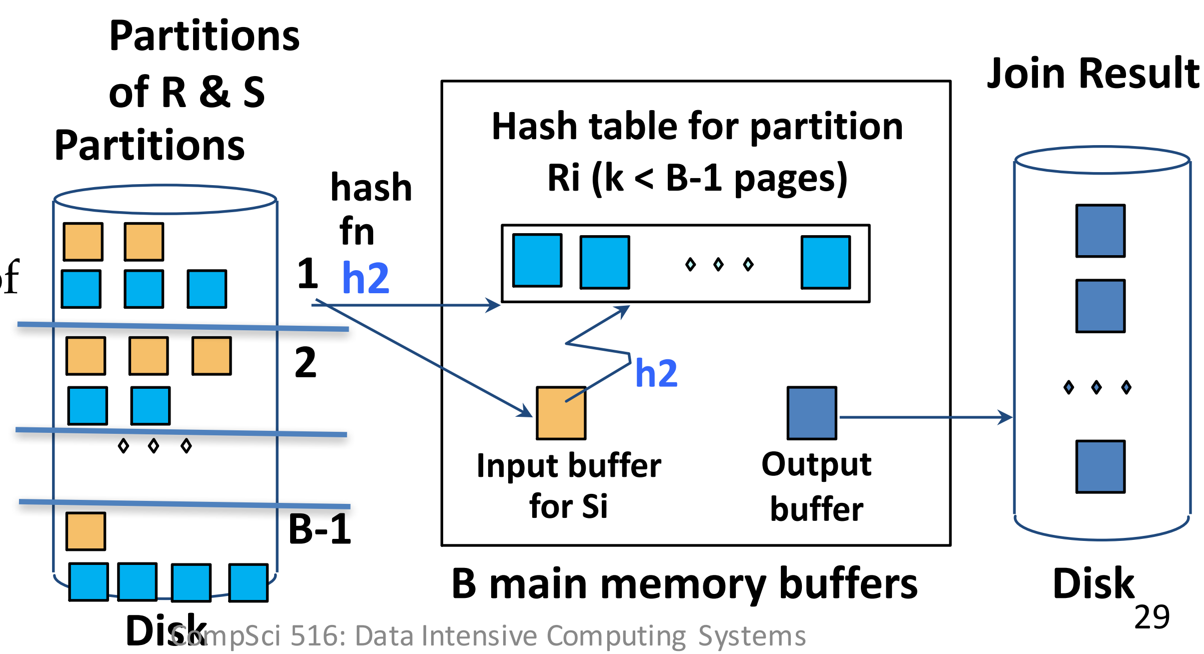
Continued from Lecture 10

Hash-Join

- Partition both relations using hash function h
- R tuples in partition i will only match S tuples in partition i



- Read in a partition of R, hash it using h2 (≠ h).
- Scan matching partition of S, search for matches.



Cost of Hash-Join

M = 1000 pages in R
p_R = 100 tuples per page

N = 500 pages in S
p_S = 80 tuples per page

- In partitioning phase
  - read+write both relns; 2(M+N)
  - In matching phase, read both relns; M+N I/Os
  - remember – we are not counting final write
- In our running example, this is a total of 4500 I/Os
  - 3 * (1000 + 500)
  - Compare with the previous joins

Sort-Merge Join vs. Hash Join

- Both can have a cost of 3(M+N) I/Os
  - if sort-merge gets enough buffer (see 14.4.2)
- Hash join holds smaller relation in buffer- better if limited buffer
- Hash Join shown to be highly parallelizable
- Sort-Merge less sensitive to data skew
  - also result is sorted

General Join Conditions

- Equalities over several attributes
  - e.g., R.sid=S.sid AND R.rname=S.sname
  - For Index Nested Loop, build index on <sid, sname> (if S is inner); or use existing indexes on sid or sname
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

- Inequality conditions
  - e.g., R.rname < S.sname
  - For Index NL, need (clustered) B+ tree index.
  - Hash Join, Sort Merge Join not applicable

Review: Join Algorithms

- Nested loop join:
  - for all tuples in R.. for all tuples in S….
  - variations: block-nested, index-nested
- Sort-merge join
  - like external merge sort
- Hash join

- Make sure you understand how the I/O varies
- No one join algorithm is uniformly superior to others
  - depends on relation size, buffer pool size, access methods, skew

Algorithms for Selection

```
SELECT  *
FROM    Reserves R
WHERE   R.rname = 'Joe'
```

- No index, unsorted data
  - Scan entire relation
  - May be expensive if not many `Joe's
- No index, sorted data (on ‘rname’)
  - locate the first tuple, scan all matching tuples
  - first binary search, then scan depends on matches
- B+-tree index, Hash index
  - Discussed earlier
  - Cost of accessing data entries + matching data records
  - Depends on clustered/unclustered
- More complex condition like day<8/9/94 AND bid=5 AND sid=3
  - Either use one index, then filter
  - Or use two indexes, then take intersection, then apply third condition
  - etc.

Algorithms for Projection

```
SELECT   DISTINCT
         R.sid, R.bid
FROM     Reserves R
```

- Two parts
  - Remove fields: easy
  - Remove duplicates (if distinct is specified): expensive
- Sorting-based
  - Sort, then scan adjacent tuples to remove duplicates
  - Can eliminate unwanted attributes in the first pass of merge sort
- Hash-based
  - Exactly like hash join
  - Partition only one relation in the first pass
  - Remove duplicates in the second pass
- Sort vs Hash
  - Sorting handles skew better, returns results sorted
  - Hash table may not fit in memory – sorting is more standard
- Index-only scan may work too
  - If all required attributes are part of index

Algorithms for Set Operations

- Intersection, cross product are special cases of joins
- Union, Except
  – Sort-based
  – Hash-based
  – Very similar to joins and projection

Algorithms for Aggregate Operations

- SUM, AVG, MIN etc.
  - again similar to previous approaches

- Without grouping:
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan

- With grouping:
  - Sort on group-by attributes
  - or, hash on group-by attributes
  - can combine sort/hash and aggregate
  - can do index-only scan here as well

Query Optimization

Query Blocks: Units of Optimization

- Query Block
  - No nesting
  - One SELECT, one FROM
  - At most one WHERE, GROUP BY, HAVING

- SQL query
- => parsed into a collection of query blocks
- => the blocks are optimized one block at a time

- Express single-block it as a relational algebra (RA) expression

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
   (SELECT  MAX (S2.age)
    FROM  Sailors S2
    GROUP BY  S2.rating)
```

Outer block   *Nested block*

Cost Estimation

- For each plan considered, must estimate cost:

- Must estimate cost of each operation in plan tree.
  - Depends on input cardinalities
  - We've discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)

- Must also estimate size of result for each operation in tree
  - gives input cardinality of next operators

- Also consider
  - whether the output is sorted
  - intermediate results written to disk

Relational Algebra Equivalences

- Allow us to choose different join orders and to `push' selections and projections ahead of joins.
- *Selections*:

$$\sigma_{c1\wedge\ldots\wedge cn}(R) \equiv \sigma_{c1}(\ \ldots\ \sigma_{cn}(R)) \quad (Cascade)$$

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R)) \quad (Commute)$$

❖ *Projections:*

$$\pi_{a1}(R) \equiv \pi_{a1}(\ \ldots(\pi_{an}(R))) \quad (Cascade)$$

❖ *Joins:*

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \quad (Associative)$$

$$(R \bowtie S) \equiv (S \bowtie R) \quad (Commute)$$

There are many more intuitive equivalences, see 15.3.4 for details

Notation

- T(R) : Number of tuples in R
- B(R) : Number of blocks (pages) in R
- V(R, A) : Number of distinct values of attribute A in R

Query Optimization Problem

Pick the best plan from the space of physical plans

Cost-based Query Optimization

Pick the plan with least cost

Challenge:

- Do not want to execute more than one plans
- Need to estimate the cost without executing the plan

"heuristic-based" optimizer (e.g. push selections down) have limited power and not used much

Cost-based Query Optimization

Pick the plan with least cost

Tasks:

1. Estimate the cost of individual operators

   done in Lecture 9-11

2. Estimate the size of output of individual operators

   today

3. Combine costs of different operators in a plan

   today

4. Efficiently search the space of plans today

Task 1 and 2
Estimating cost and size of different operators

- Size = #tuples, NOT #pages
- Cost = #page I/O
  - but, need to consider whether the intermediate relation fits in memory, is written back to/read from disk (or on-the-fly goes to the next operator), etc.

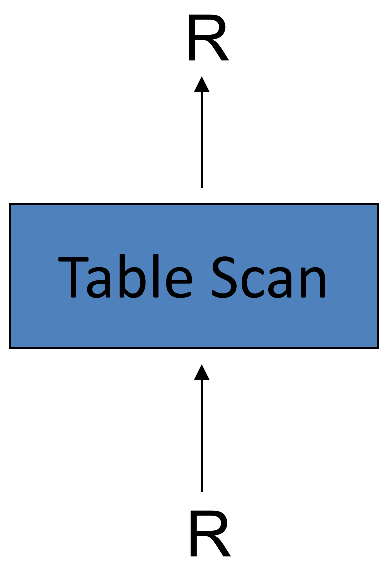
Desired Properties of Estimating Sizes of Intermediate Relations

Ideally,

- should give accurate estimates (as much as possible)
- should be easy to compute
- should be logically consistent
  – size estimate should be independent of how the relation is computed (e.g. which join algorithm/join order is used)

- But, no "universally agreed upon" ways to meet these goals

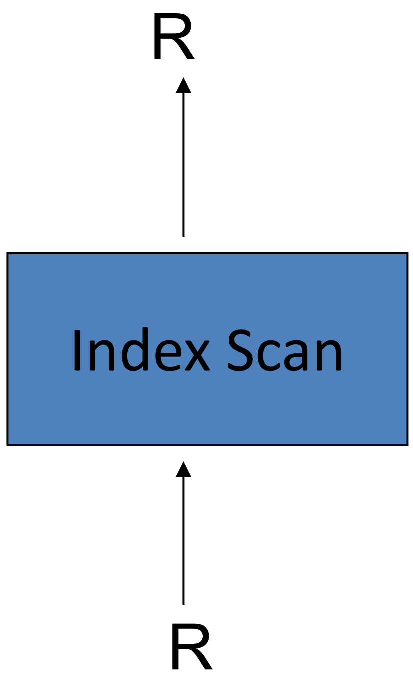
Cost of Table Scan



Cost: B(R)
Size: T(R)

T (R) : Number of tuples in R
B (R) : Number of blocks in R

Cost of Index Scan



Cost: B(R) – if clustered
T(R) – if unclustered

Size: T(R)

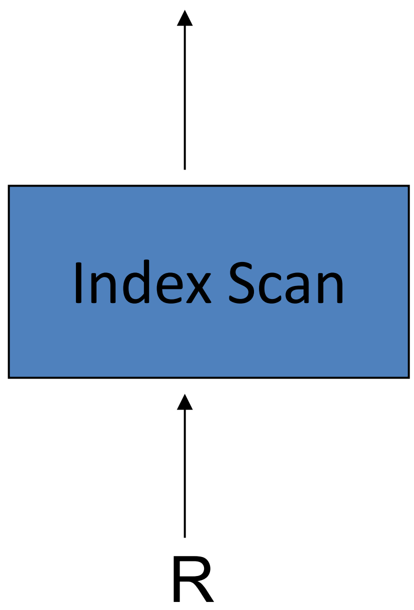
T (R) : Number of tuples in R
B (R) : Number of blocks in R

Note:

1. size is independent of the implementation of the scan/index
2. Index scan is bad if unclustered

Cost of Index Scan with Selection

$X = \sigma_{R.A > 50} R$



Cost: $B(R) * f$ – if clustered

$T(R) * f$ – if unclustered

Size: $T(R) * f$

T (R) : Number of tuples in R

B (R) : Number of blocks in R

Reduction factor

$f = (Max(R.A) - 50) / (Max(R.A) - Min(R.A))$

assumes uniform distribution

Cost of Index Scan with Selection (and multiple conditions)

$X = \sigma_{R.A > 50 \text{ and } R.B = C} R$

assume index on (A, B)

What is f1 if the first condition is 100 > R.1 > 50?

Index Scan

R

Cost: B(R) * f – if clustered
T(R) * f – if unclustered

Size: T(R) * f

Reduction factors

range selection

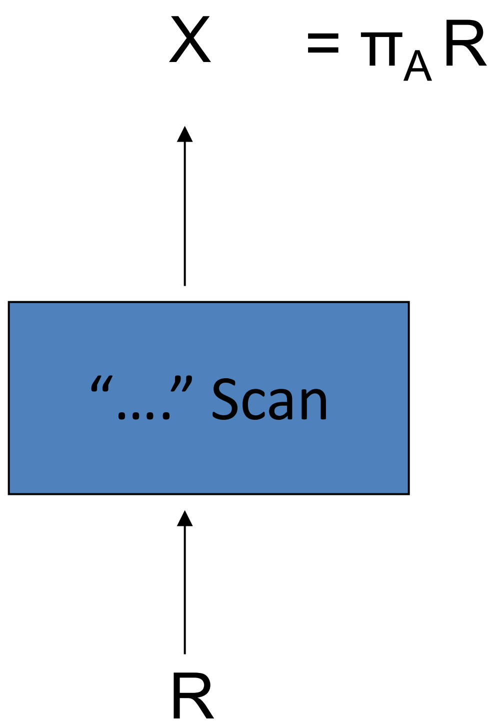
f1 = (Max(R.A) - 50) / (Max(R.A) - Min(R.A))

f2 = 1/ V(R, B) value selection

T (R) : Number of tuples in R
B (R) : Number of blocks in R
V(R, A) : Number of distinct values of attribute A in R

f = f1 * f2 (assumes independence and uniform distribution)

Cost of Projection



Cost: depends on the method of scanning R

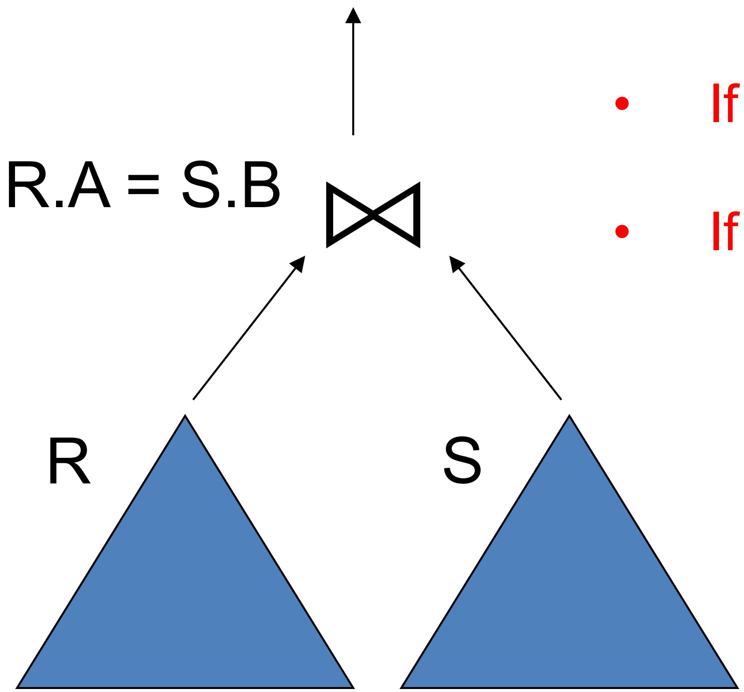
B(R) for table scan or clustered index scan

Size: T(R)

But tuples are smaller

If you have more information on the size of the smaller tuples, can estimate #I/O better

Size of Join



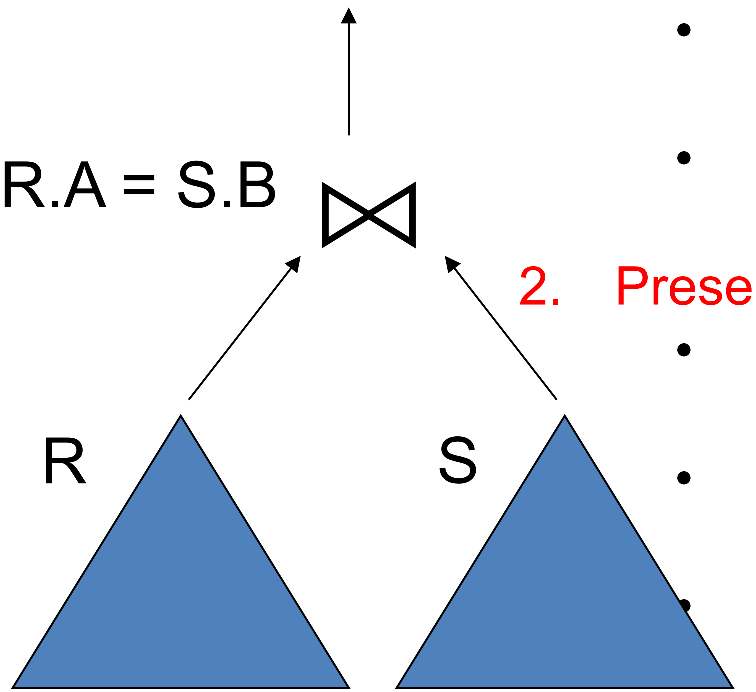
Quite tricky

- If disjoint A and B values
  - then 0
- If A is key of R and B is foreign key of S
  - then T(S)
- If all tuples have the same value of R.A= S.B = x
  - then T(R) * T(S)

T (R) : Number of tuples in R
B (R) : Number of blocks in R
V(R, A) : Number of distinct values of attribute A in R

T (R) : Number of tuples in R
B (R) : Number of blocks in R
V(R, A) : Number of distinct values of attribute A in R

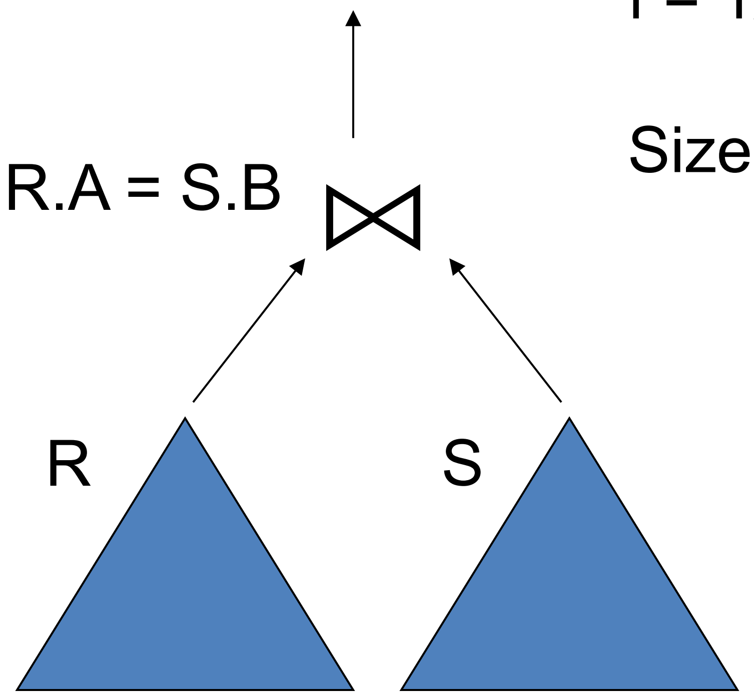
Size of Join



Two standard assumptions

1. Containment of value sets:
   - if V(R, A) <= V(S, B), then all A-values of R are included in B-values of S
   - e.g. satisfied when A is foreign key, B is key
2. Preservation of value sets:
   - For all "non-joining" attributes, the set of distinct values is preserved in join
   - $V(R \bowtie S, C) = V(R, C)$, where $C \neq A$ is an attribute in R
   - $V(R \bowtie S, D) = V(S, D)$, where $D \neq B$ is an attribute in S
   - Helps estimate distinct set size in $R \bowtie S \bowtie T$

Size of Join



Reduction factor

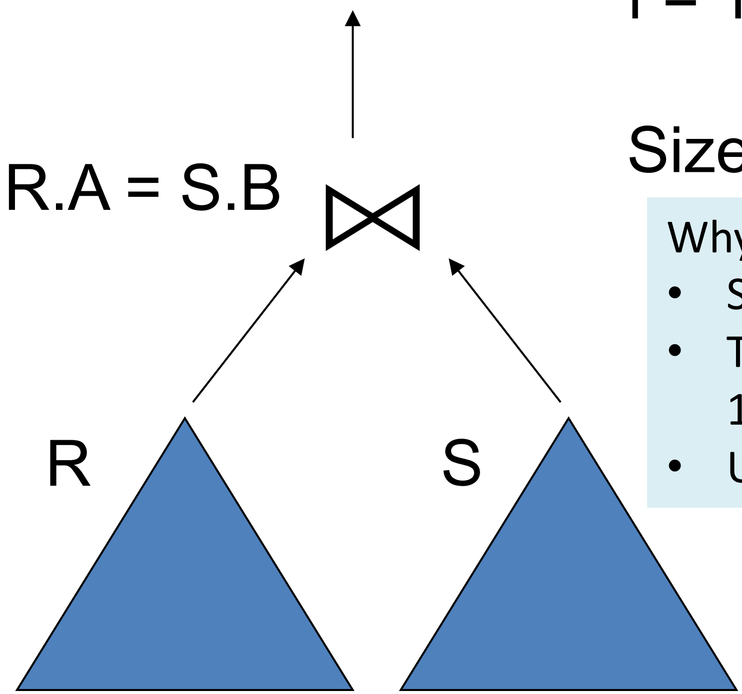
f = 1/max(V(R, A), V(S, B))

Size = T(R) * T(S) * f

T (R) : Number of tuples in R
B (R) : Number of blocks in R
V(R, A) : Number of distinct values of attribute A in R

Size of Join

Assumes index on both A and B
if one index: 1/V(..., ...)
if no index: say 1/10



Reduction factor

f = 1/max(V(R, A), V(S, B))

Size = T(R) * T(S) * f

Why max?
- Suppose V(R, A) <= V(S, B)
- The probability of a A-value joining with a B-value is 1/V(S.B) = reduction factor
- Under the two assumptions stated earlier + uniformity

T (R) : Number of tuples in R
B (R) : Number of blocks in R
V(R, A) : Number of distinct values of attribute A in R

Task 3: Combine cost of different operators in a plan

With Examples
“Given” the physical plan

- Size = #tuples, NOT #pages
- Cost = #page I/O
  - but, need to consider whether the intermediate relation fits in memory, is written back to disk (or on-the-fly goes to the next operator) etc.

Example Query

Student (<u>sid</u>, name, age, address)
Book(<u>bid</u>, title, author)
Checkout(<u>sid, bid</u>, date)

Query:

```
SELECT S.name
FROM Student S, Book B, Checkout C
WHERE S.sid = C.sid
AND B.bid = C.bid
AND B.author = 'Olden Fames'
AND S.age > 12
AND S.age < 20
```

S(<u>sid</u>,name,age,addr)
B(<u>bid</u>,title,author)
C(<u>sid,bid</u>,date)

Assumptions

- Student: S, Book: B, Checkout: C

- Sid, bid foreign key in C referencing S and B resp.
- There are 10,000 Student records stored on 1,000 pages.
- There are 50,000 Book records stored on 5,000 pages.
- There are 300,000 Checkout records stored on 15,000 pages.
- There are 500 different authors.
- Student ages range from 7 to 24.

Warning: a few dense slides next ☺

S(<u>sid</u>,name,age,addr)
B(<u>bid</u>,title,author)
C(<u>sid,bid</u>,date)

T(S)=10,000
T(B)=50,000
T(C)=300,000

B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

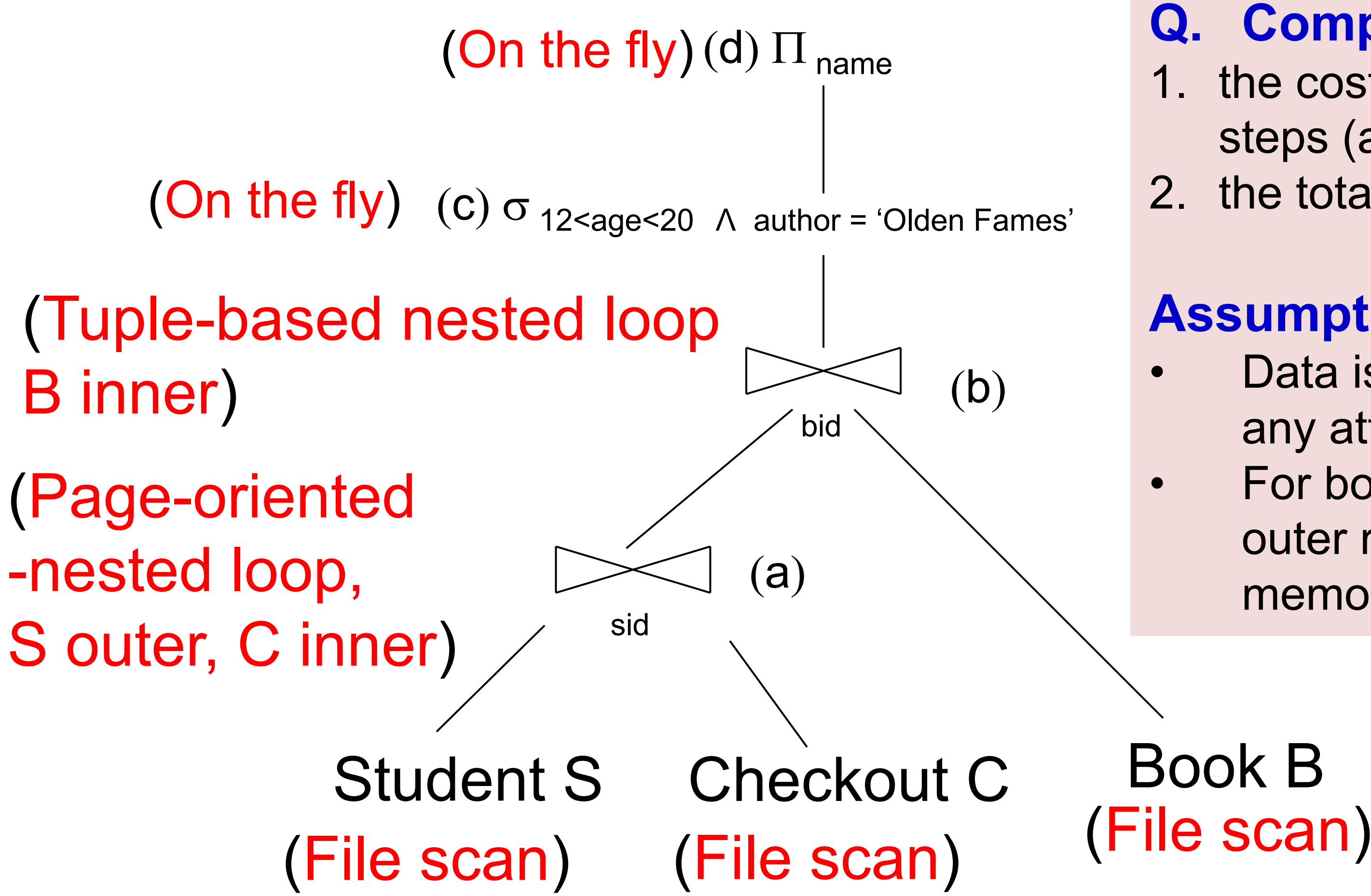
Physical Query Plan – 1

Q. Compute

1. the cost and cardinality in steps (a) to (d)
2. the total cost

Assumptions (given):

- Data is not sorted on any attributes
- For both in (a) and (b), outer relations fit in memory



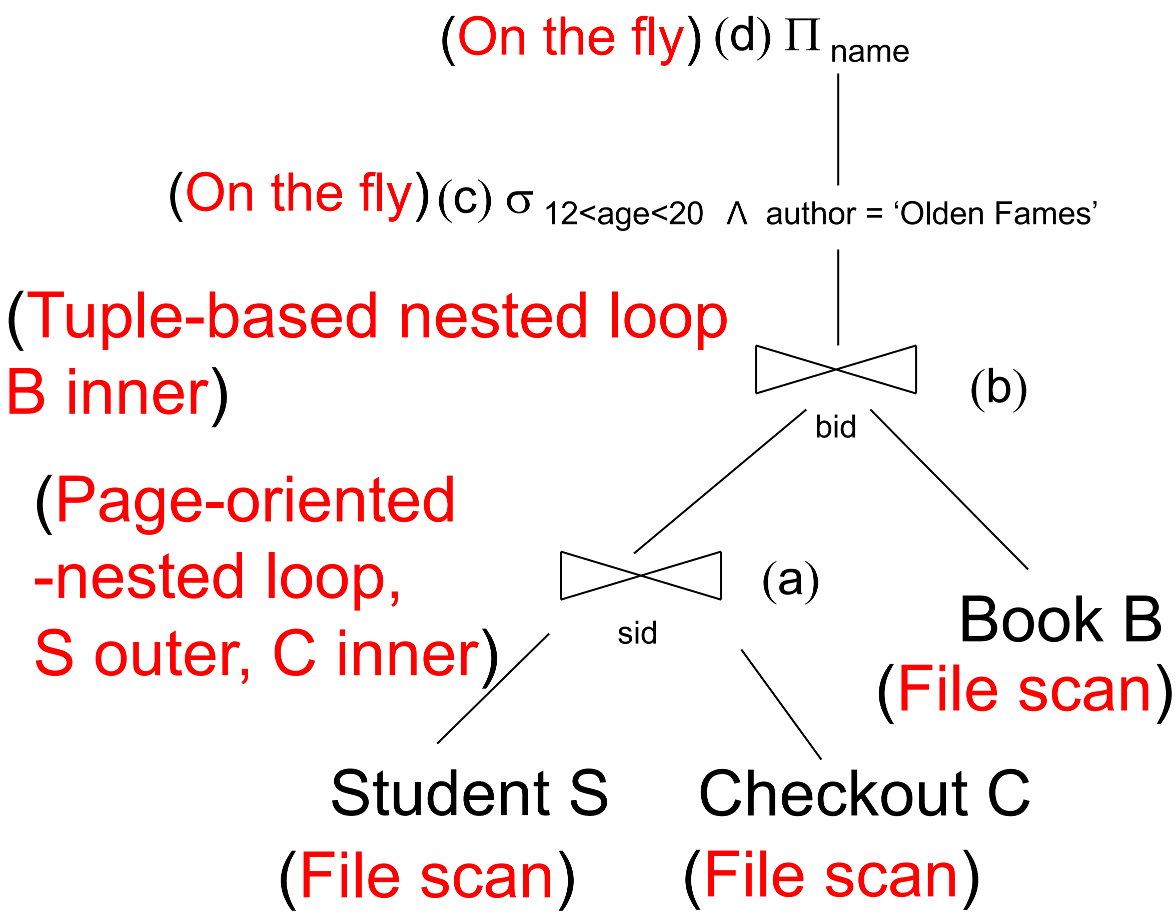
S(<u>sid</u>,name,age,addr)
B(<u>bid</u>,title,author)
C(<u>sid,bid</u>,date)

T(S)=10,000
T(B)=50,000
T(C)=300,000

B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(a)



Cost =
B(S) + B(S) * B(C)
= 1000 + 1000 * 15000
= 15,001,000

Cardinality =
T(C) = 300,000
- foreign key join, output pipelined to next join
- Can apply the formula as well

T(S) * T(C)/max (V(S, sid), V(C, sid))
= T(C)
since V(S, sid) > = V(C, sid) and
T(S) = V(S, sid)

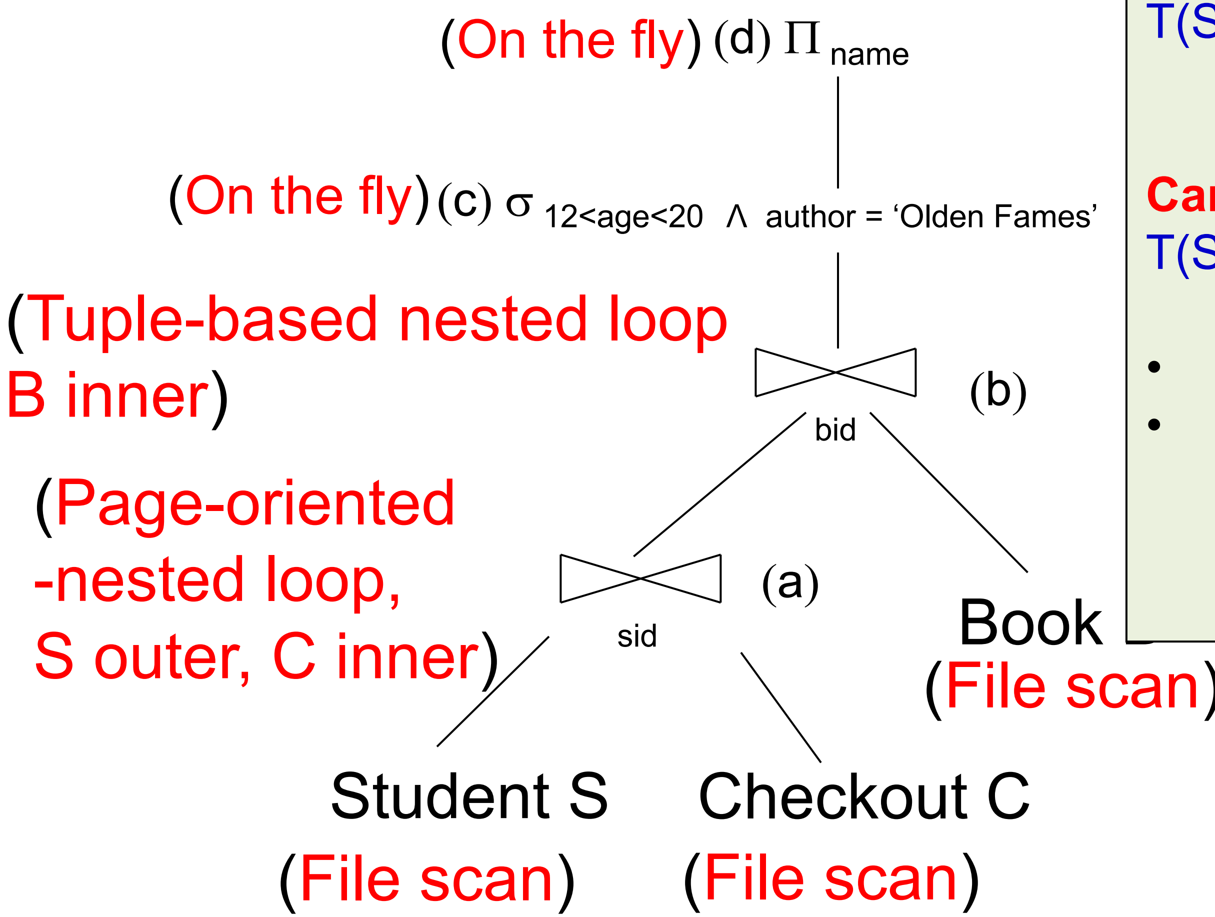
S(<u>sid</u>,name,age,addr)
B(<u>bid</u>,title,author)
C(<u>sid,bid</u>,date)

T(S)=10,000
T(B)=50,000
T(C)=300,000

B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(b)



Cost =
$T(S \bowtie C) * B(B)$
$= 300,000 * 5,000 \quad = 15 * 10^8$

Cardinality =
$T(S \bowtie C) = 300,000$

- foreign key join
- don't need scanning for outer relation
  - outer relation fits in memory

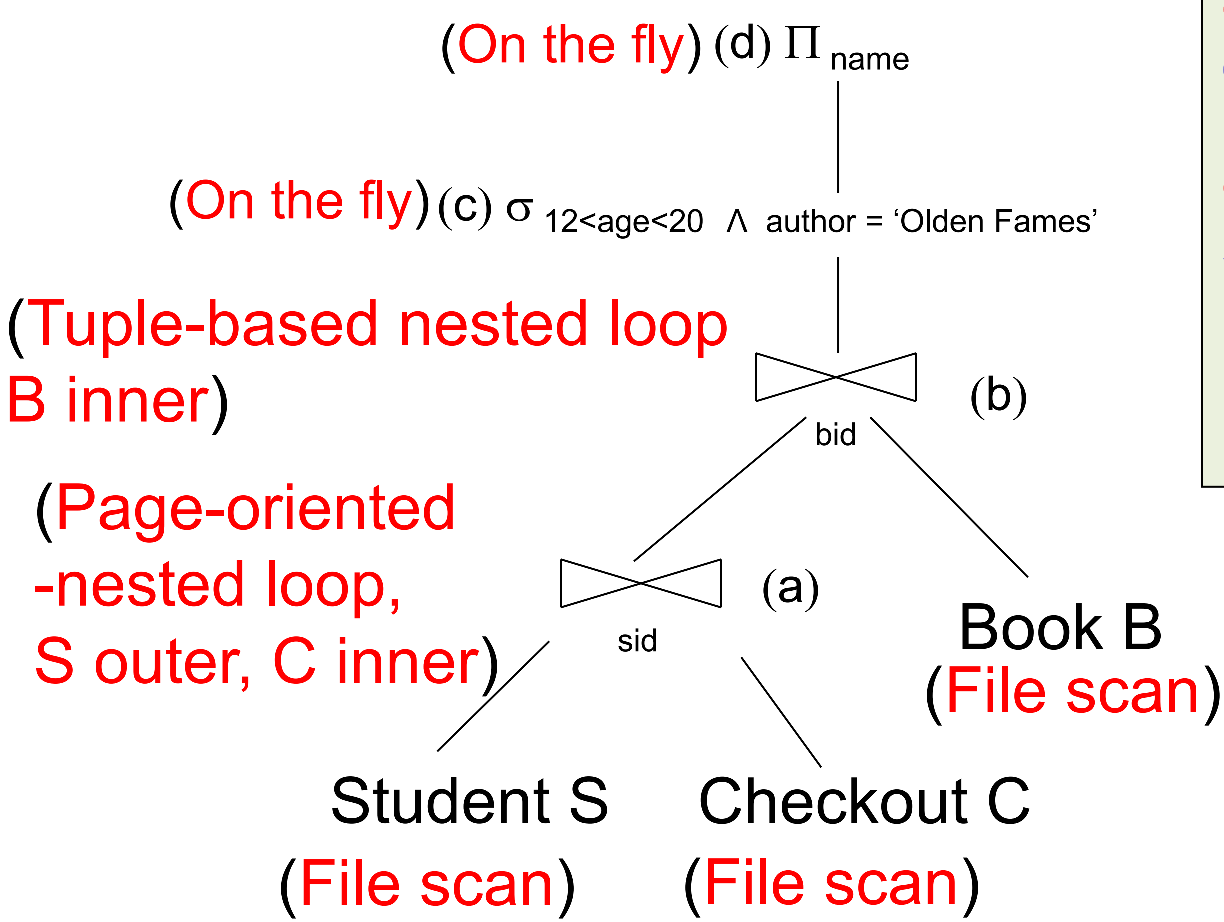
S(sid,name,age,addr)
B(bid,title,author)
C(sid,bid,date)

T(S)=10,000
T(B)=50,000
T(C)=300,000

B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(c, d)



Cost =
0 (on the fly)

Cardinality =
300,000 * 1/500 * 7/18
= 234 (approx)
(assuming uniformity and independence)

S(<u>sid</u>,name,age,addr)
B(<u>bid</u>,title,author)
C(<u>sid,bid</u>,date)

T(S)=10,000
T(B)=50,000
T(C)=300,000

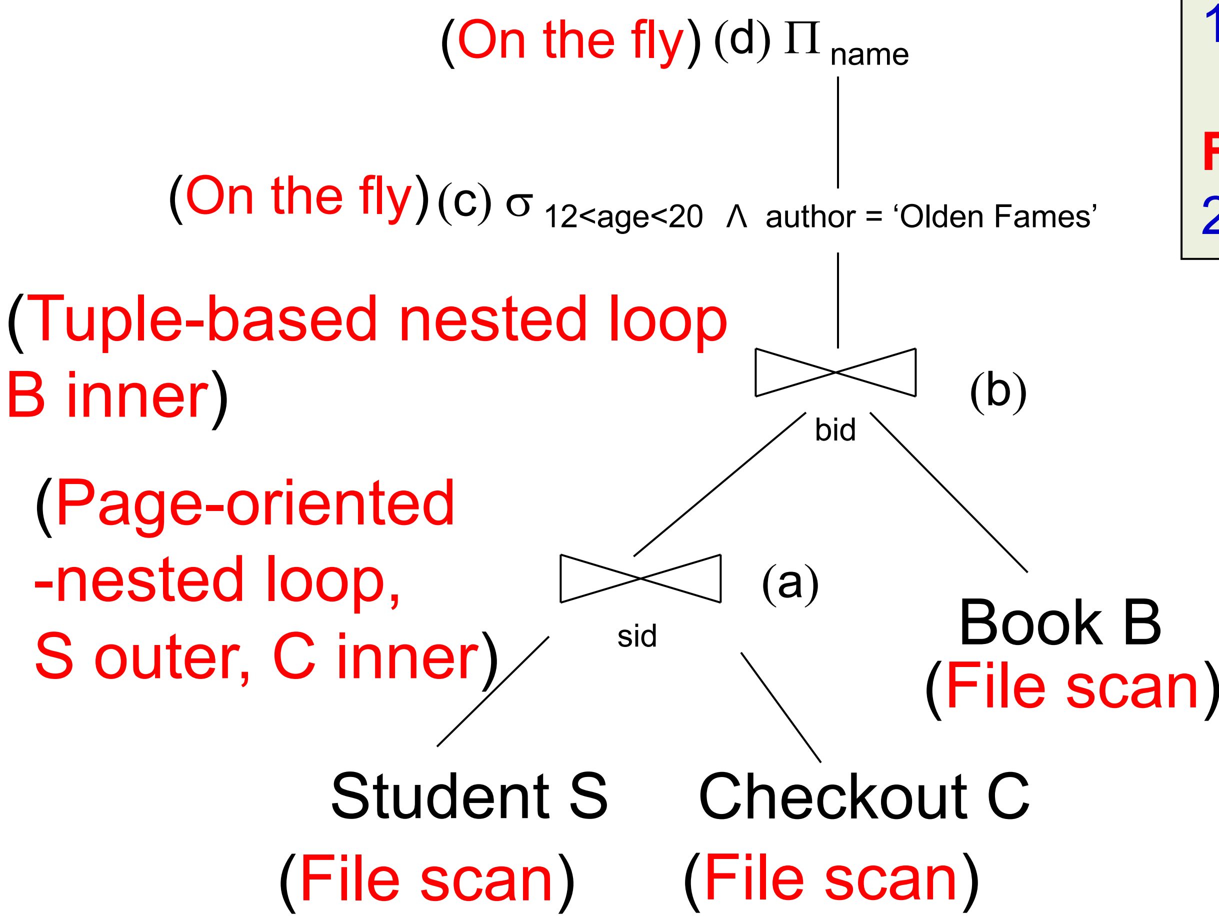
B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(Total)

Total cost =
1,515,001,000

Final cardinality =
234 (approx)



S(sid,name,age,addr)
B(bid,title,author)
C(sid,bid,date)

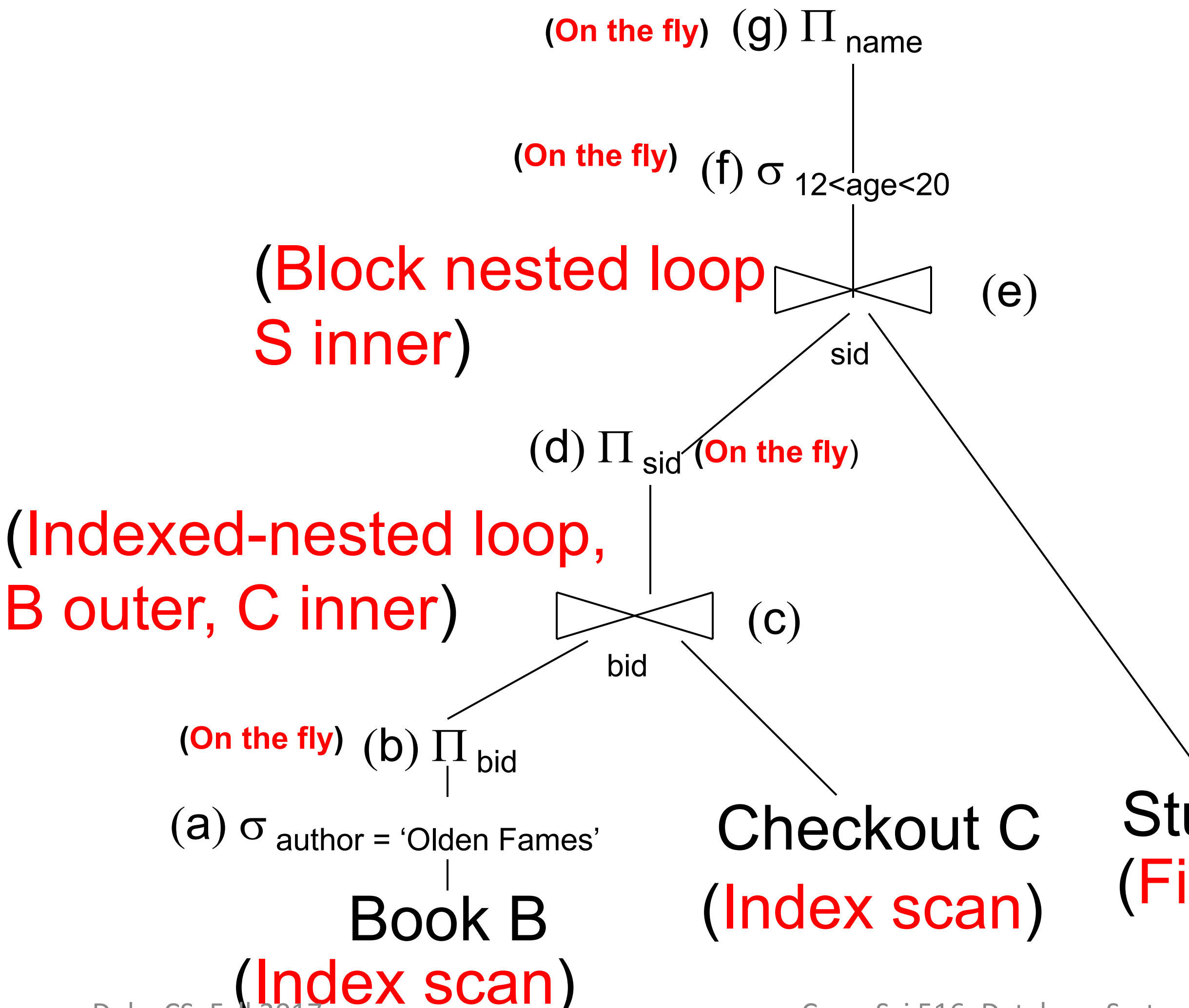
$T(S)=10{,}000$
$T(B)=50{,}000$
$T(C)=300{,}000$

$B(S)=1{,}000$
$B(B)=5{,}000$
$B(C)=15{,}000$

V(B,author) = 500
7 <= age <= 24

V(B,author) = 500
7 <= age <= 24

Physical Query Plan – 2



Q. Compute
1. the cost and cardinality in steps (a) to (g)
2. the total cost

Assumptions (given):
- Unclustered B+tree index on B.author
- Clustered B+tree index on C.bid
- All index pages are in memory
- Unlimited memory

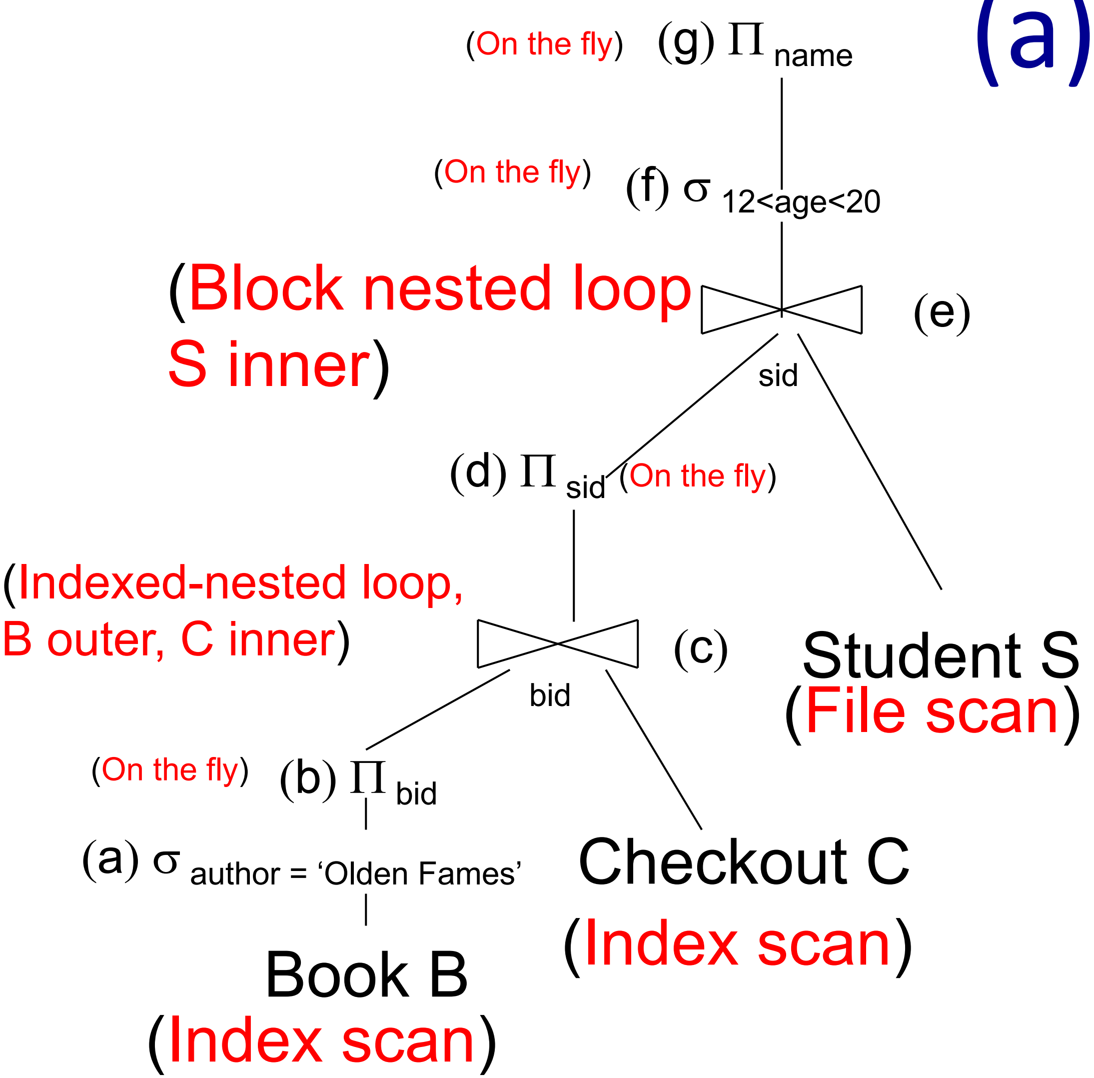
S(sid,name,age,addr)
B(bid,title,author): Un. B+ on author
C(sid,bid,date): Cl. B+ on bid

T(S)=10,000
T(B)=50,000
T(C)=300,000

B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(a)



Cost =
T(B) / V(B, author)
= 50,000/500
= 100 (unclustered)

Cardinality =
100

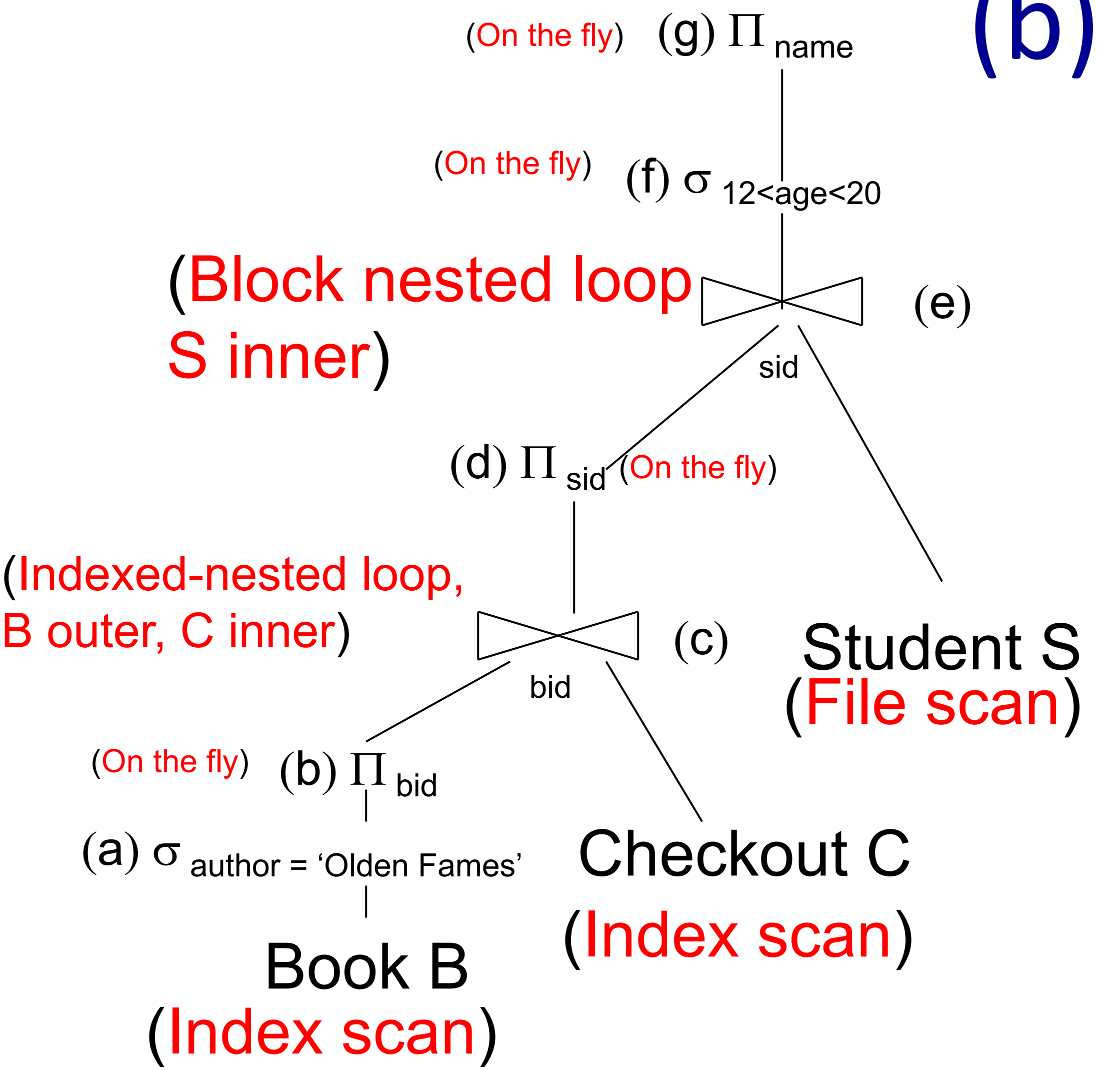
S(<u>sid</u>,name,age,addr)
B(<u>bid</u>,title,author): Un. B+ on author
C(<u>sid,bid</u>,date): Cl. B+ on bid

T(S)=10,000
T(B)=50,000
T(C)=300,000

B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(b)



Cost =
0 (on the fly)

Cardinality =
100

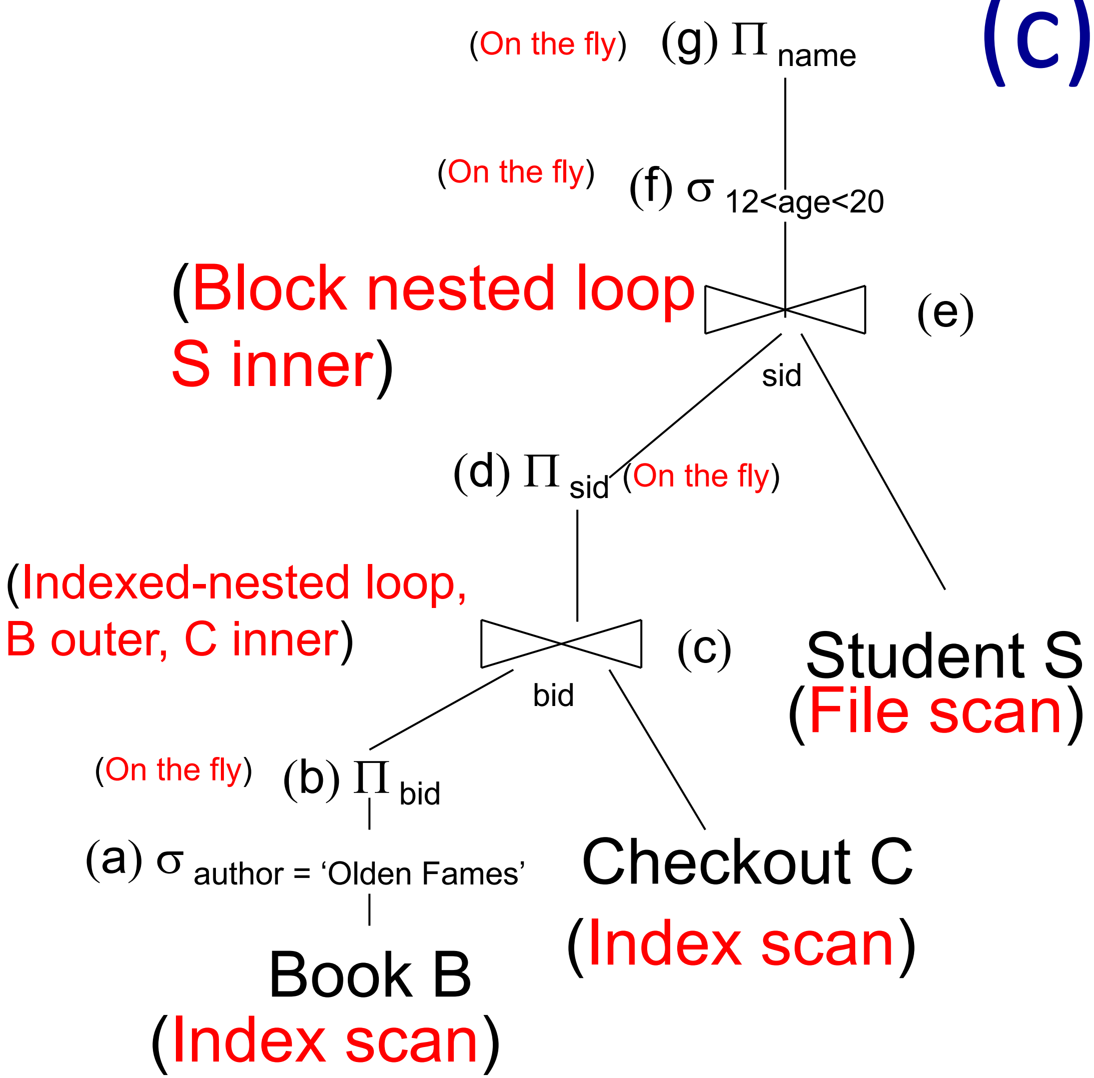
S(<u>sid</u>,name,age,addr)
B(<u>bid</u>,title,author): Un. B+ on author
C(<u>sid,bid</u>,date): Cl. B+ on bid

T(S)=10,000
T(B)=50,000
T(C)=300,000

B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(c)



- one index lookup per outer B tuple
- 1 book has T(C)/ T(B) = 6 checkouts (uniformity)
- # C tuples per page = T(C)/B(C) = 20
- 6 tuples fit in at most 2 consecutive pages (clustered) could assume 1 page as well

Cost <=
100 * 2= 200

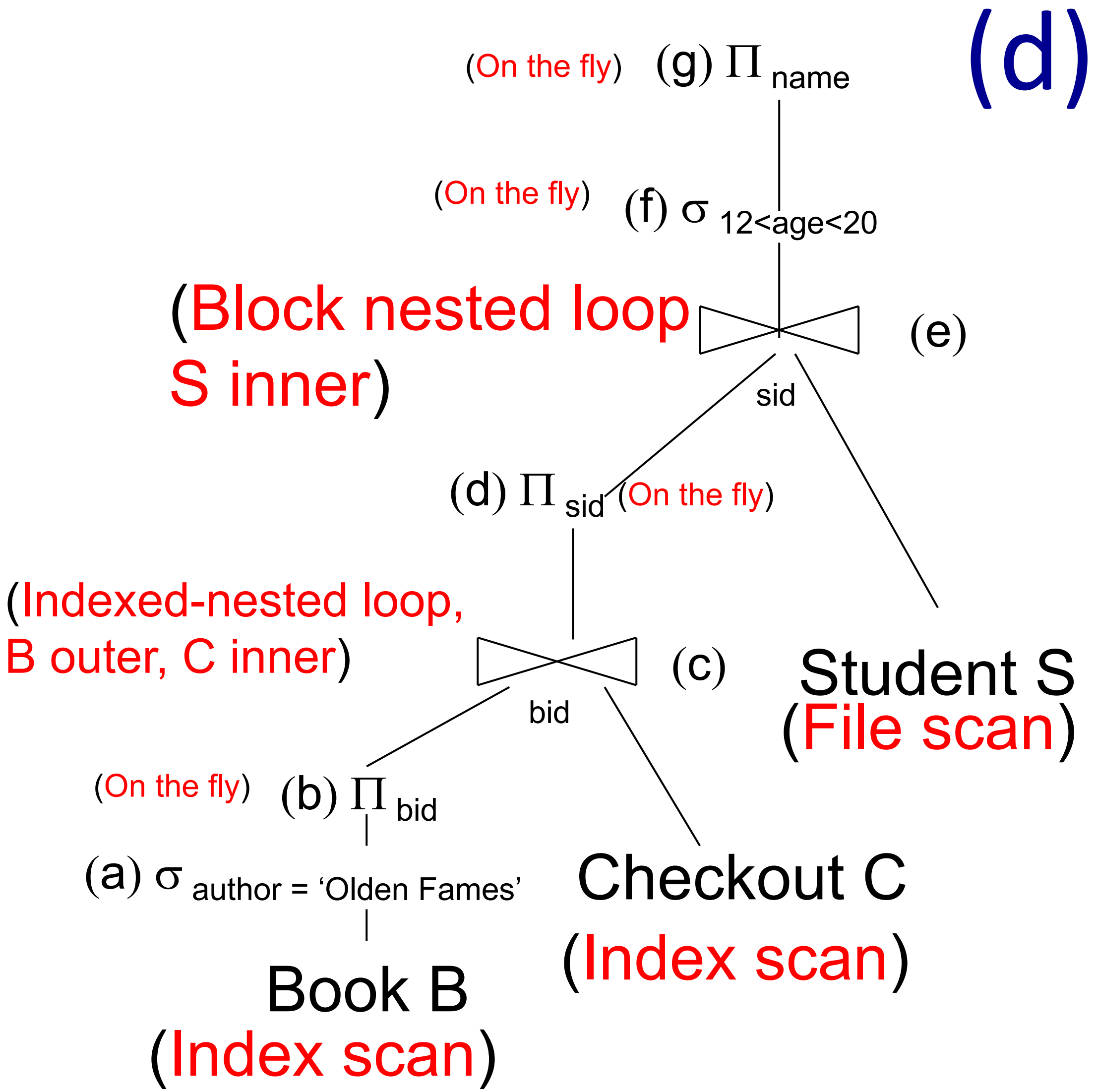
Cardinality =
100 * 6 = 600

= 100 * T(C)/ MAX(100, V(C, bid)) assuming
V(C, bid) = V(B, bid) = T(B) = 50,000

S(<u>sid</u>,name,age,addr)
B(<u>bid</u>,title,author): Un. B+ on author
C(<u>sid,bid</u>,date): Cl. B+ on bid

T(S)=10,000 B(S)=1,000
T(B)=50,000 B(B)=5,000
T(C)=300,000 B(C)=15,000

V(B,author) = 500
7 <= age <= 24



Cost =
0 (on the fly)

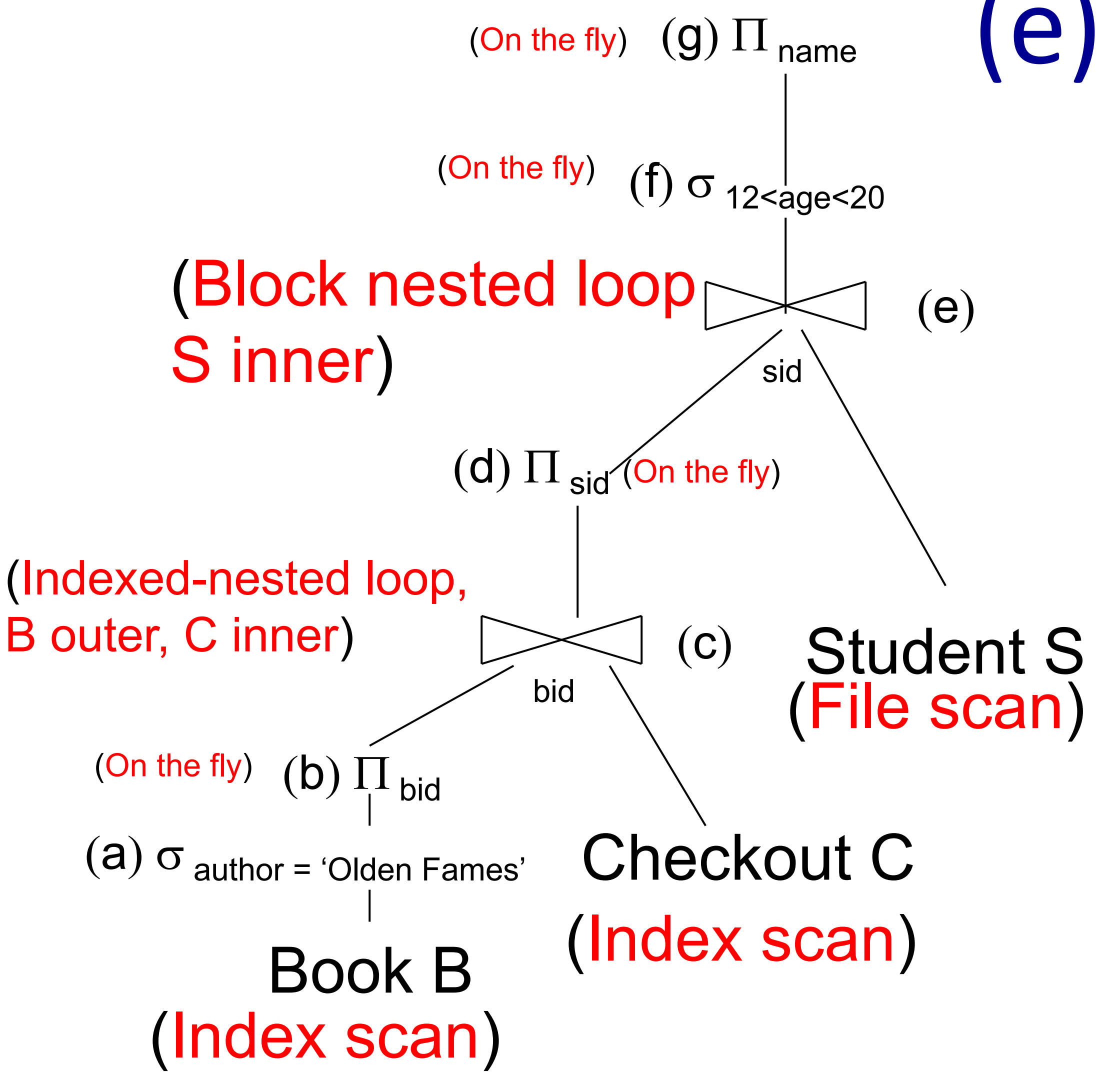
Cardinality =
600

S(sid,name,age,addr)
B(bid,title,author): Un. B+ on author
C(sid,bid,date): Cl. B+ on bid

T(S)=10,000 B(S)=1,000
T(B)=50,000 B(B)=5,000
T(C)=300,000 B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(e)



Outer relation is already in (unlimited) memory
need to scan S relation

Cost =
B(S) = 1000

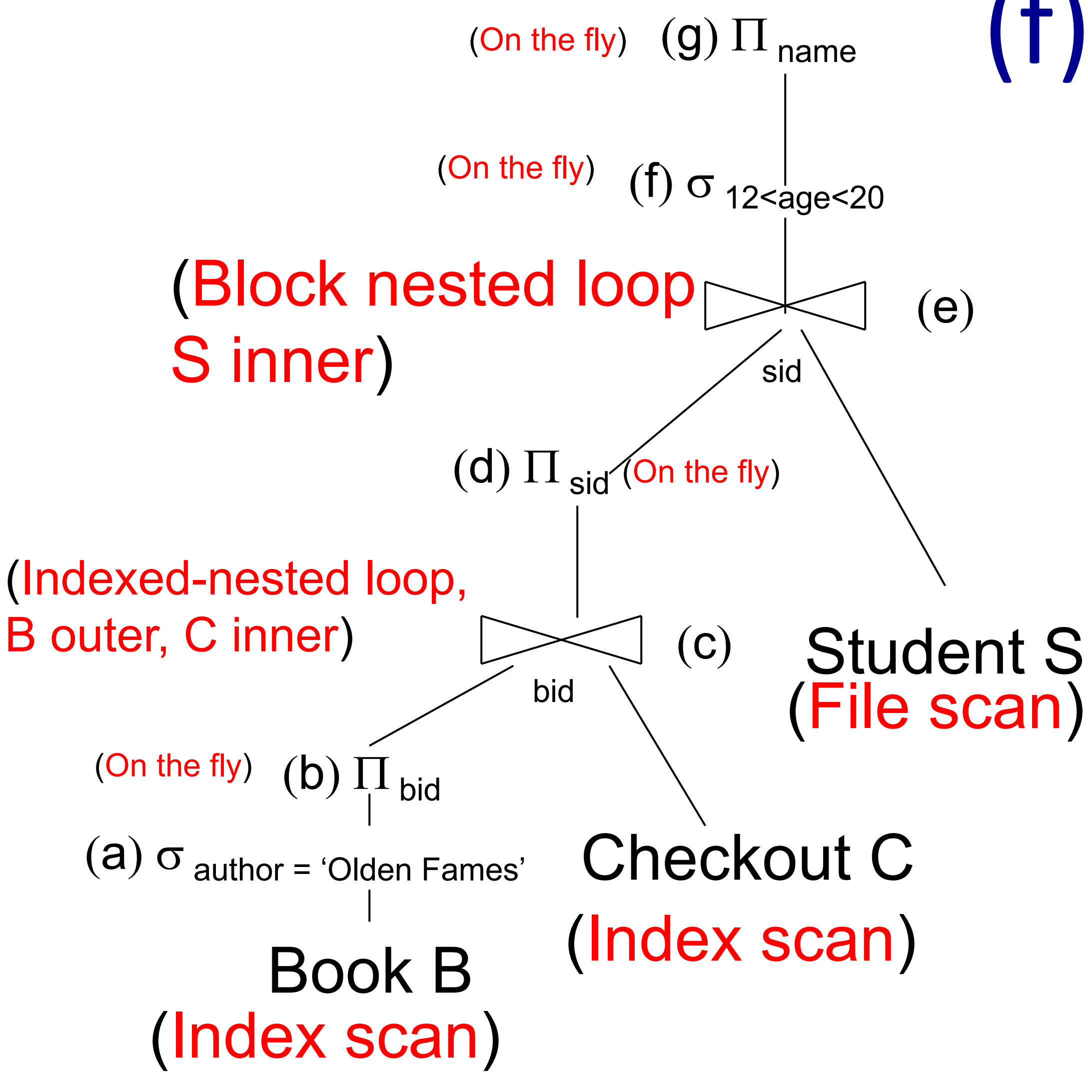
Cardinality =
600
(one student per checkout)

S(sid,name,age,addr)
B(bid,title,author): Un. B+ on author
C(sid,bid,date): Cl. B+ on bid

T(S)=10,000 B(S)=1,000
T(B)=50,000 B(B)=5,000
T(C)=300,000 B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(f)



Cost =
0 (on the fly)

Cardinality =
600 * 7/18 = 234 (approx)

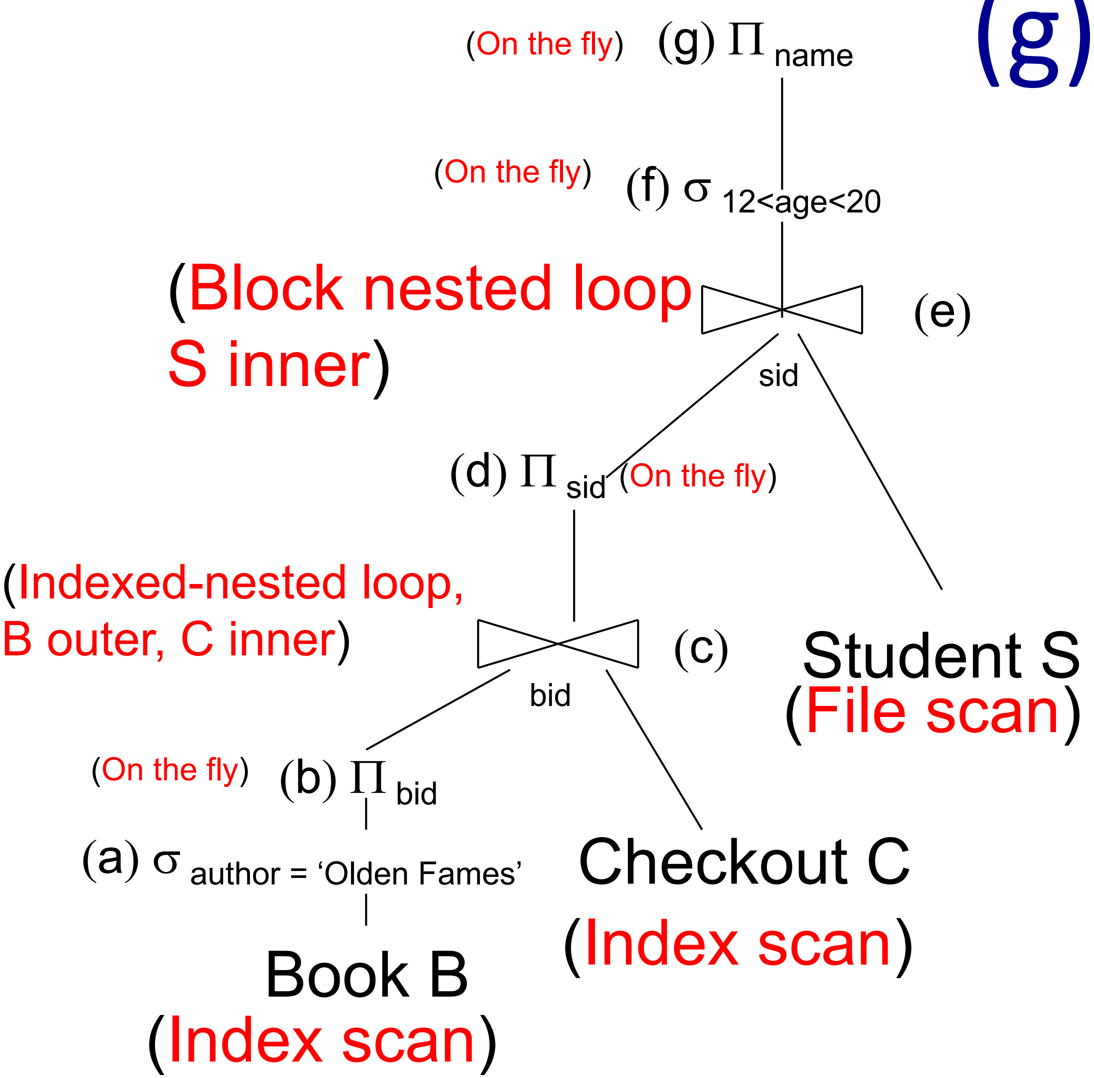
S(sid,name,age,addr)
B(bid,title,author): Un. B+ on author
C(sid,bid,date): Cl. B+ on bid

T(S)=10,000
T(B)=50,000
T(C)=300,000

B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(g)



Cost =
0 (on the fly)

Cardinality =
234

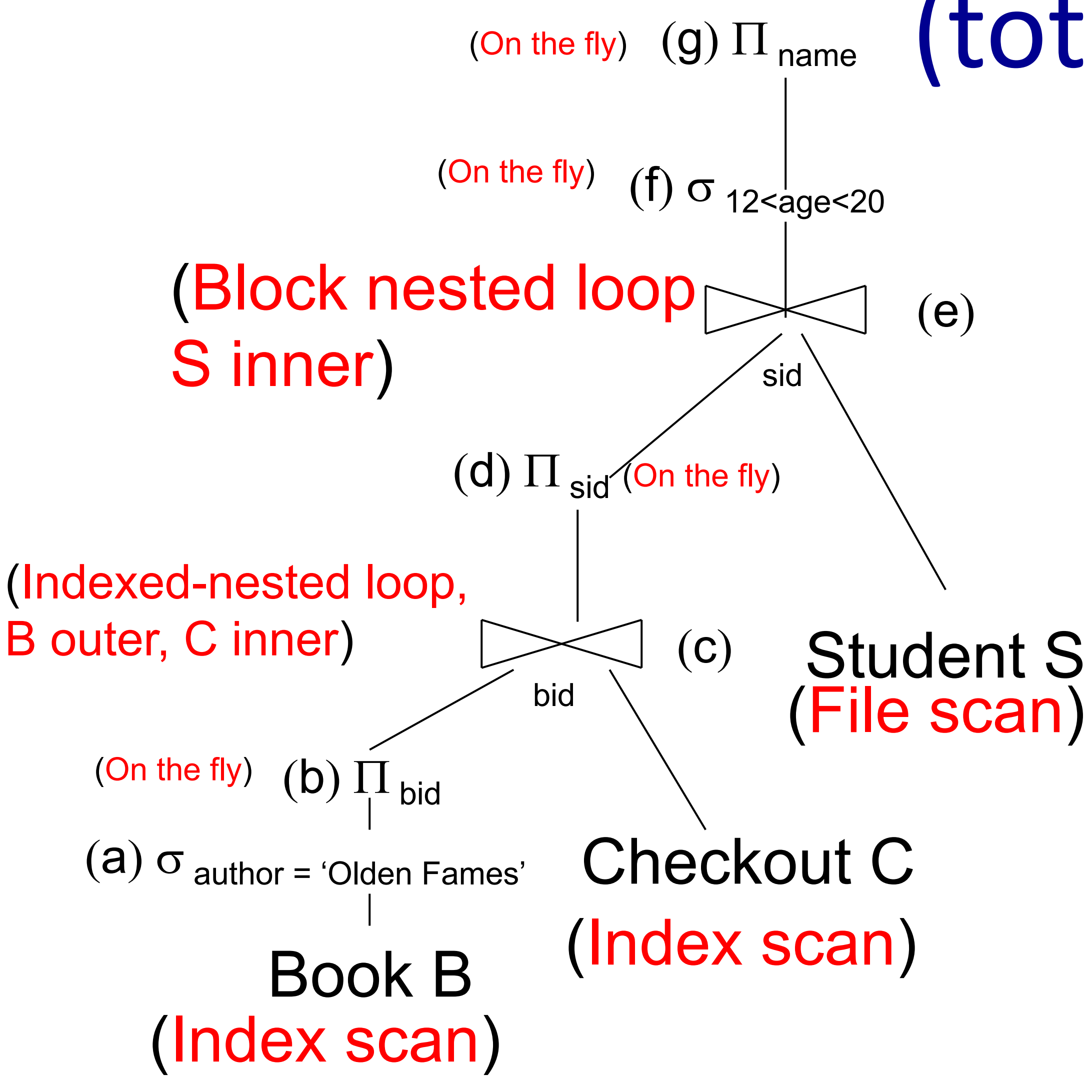
S(<u>sid</u>,name,age,addr)
B(<u>bid</u>,title,author): Un. B+ on author
C(<u>sid,bid</u>,date): Cl. B+ on bid

T(S)=10,000
T(B)=50,000
T(C)=300,000

B(S)=1,000
B(B)=5,000
B(C)=15,000

V(B,author) = 500
7 <= age <= 24

(total)



Total cost = 1300
(compare with 1,515,001,000 for plan 1!)

Final cardinality = 234 (approx)
(same as plan 1!)

End of Lecture 11

Start of Lecture 12

Task 4:
Efficiently searching the plan space

Use dynamic-programming based Selinger's algorithm

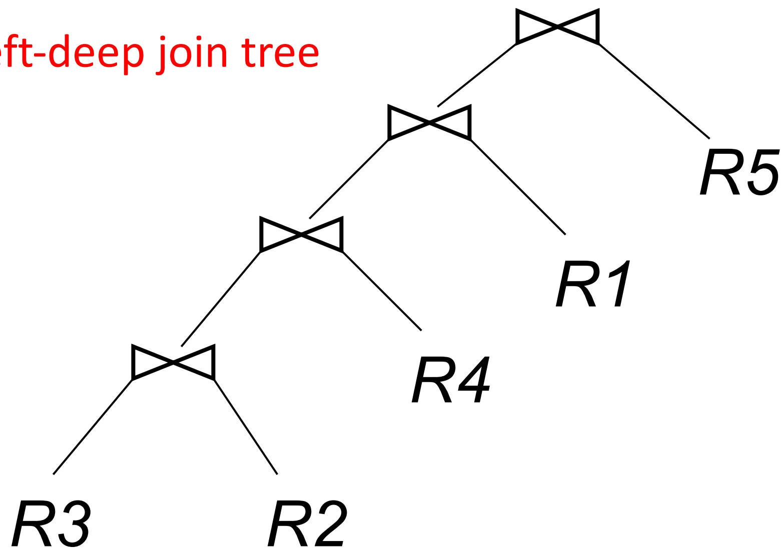
Heuristics for pruning plan space

- Apply predicates as early as possible
- Avoid plans with cross products
- Only left-deep join trees

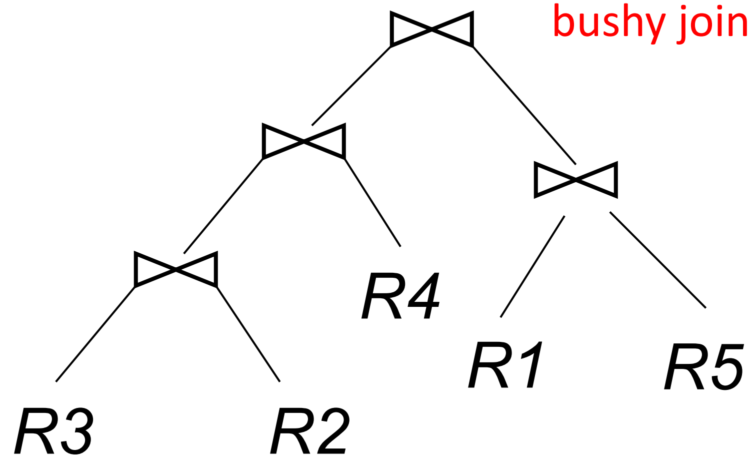
Join Trees

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

left-deep join tree



bushy join tree



(logical plan space)

- Several possible structure of the trees
- Each tree can have n! permutations of relations on leaves

(physical plan space)

- Different implementation and scanning of intermediate operators for each logical plan

Selinger Algorithm

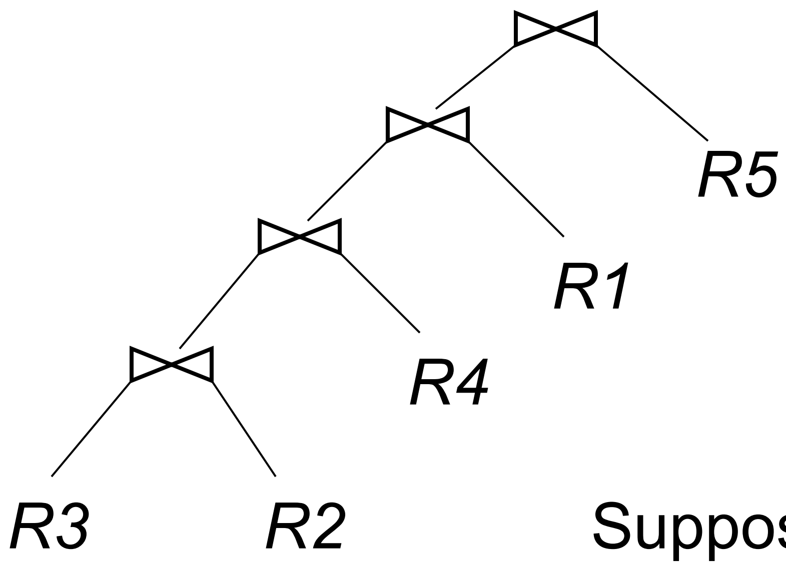
- Dynamic Programming based
- Dynamic Programming:
  - General algorithmic paradigm
  - Exploits "principle of optimality"
    - Useful reading: Chapter 16, Introduction to Algorithms, Cormen, Leiserson, Rivest
- Considers the search space of left-deep join trees
  - reduces search space (only one structure)
  - but still n! permutations
  - interacts well with join algos (esp. NLJ)
  - e.g. might not need to write tuples to disk if enough memory

Principle of Optimality

$\boxed{\text{Optimal for “whole” made up from optimal for “parts”}}$

Principle of Optimality

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

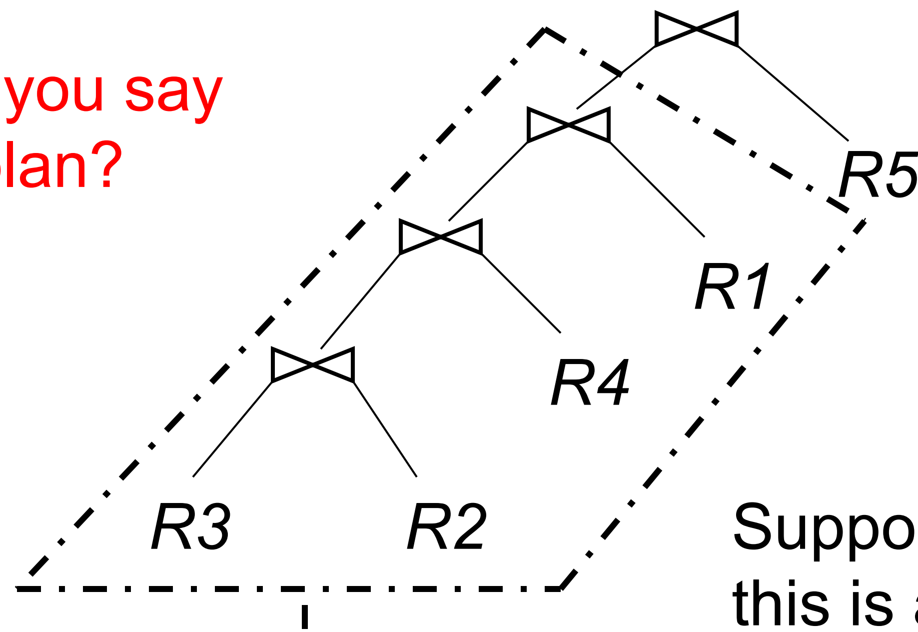


Suppose,
this is an Optimal Plan
for joining R1…R5:

Principle of Optimality

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

Then, what can you say about this sub-plan?



Suppose,
this is an Optimal Plan for joining R1…R5:

This has to be the optimal plan for joining *R3, R2, R4, R1*

Principle of Optimality

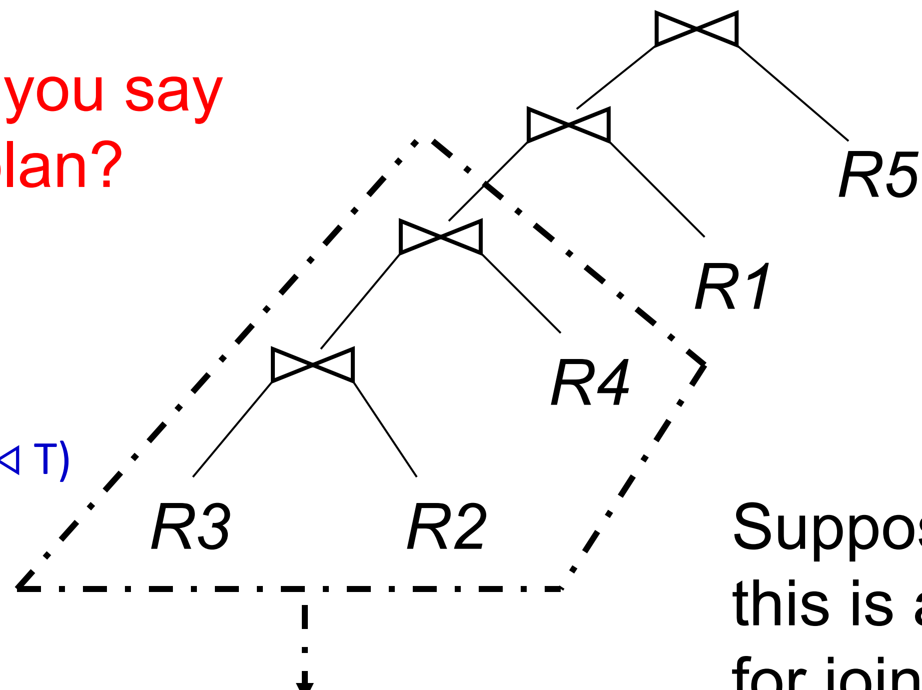
Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4 \bowtie R5$

Then, what can you say about this sub-plan?

We are using the associativity and commutativity of joins

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

$R \bowtie S = S \bowtie R$



Suppose, this is an Optimal Plan for joining R1…R5:

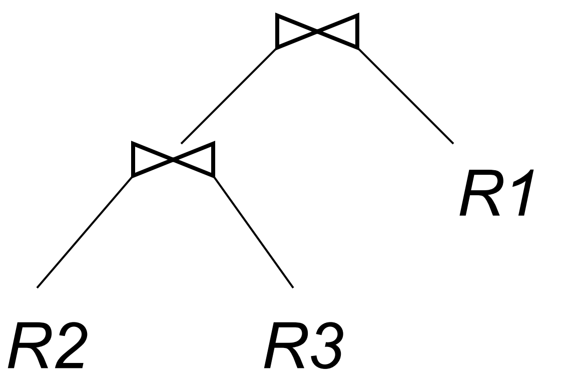
This has to be the optimal plan for joining *R3, R2, R4*

Exploiting Principle of Optimality

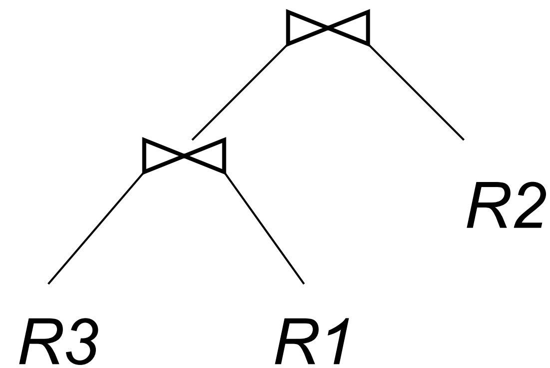
Query: $R1 \bowtie R2 \bowtie \quad \ldots \quad \bowtie Rn$

Both are giving the same result

R2 ⋈ R3 ⋈ R1 = R3 ⋈ R1 ⋈ R2



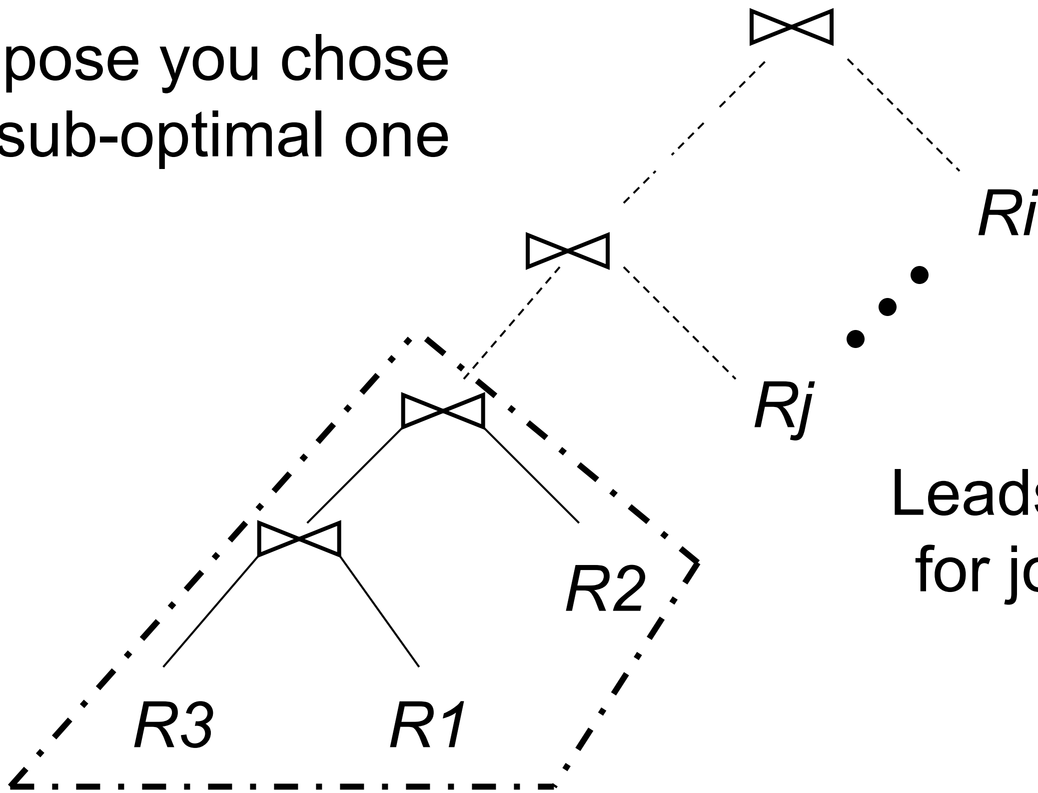
Optimal
for joining *R1, R2, R3*



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

Exploiting Principle of Optimality

Suppose you chose
the sub-optimal one



Leads to sub-Optimal
for joining R1,…,Rn

A sub-optimal sub-plan cannot lead to an optimal plan

Notation

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

Cost of optimal plan to join $R1,R2,R3$

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

Number of tuples in $R1 \bowtie R2 \bowtie R3$

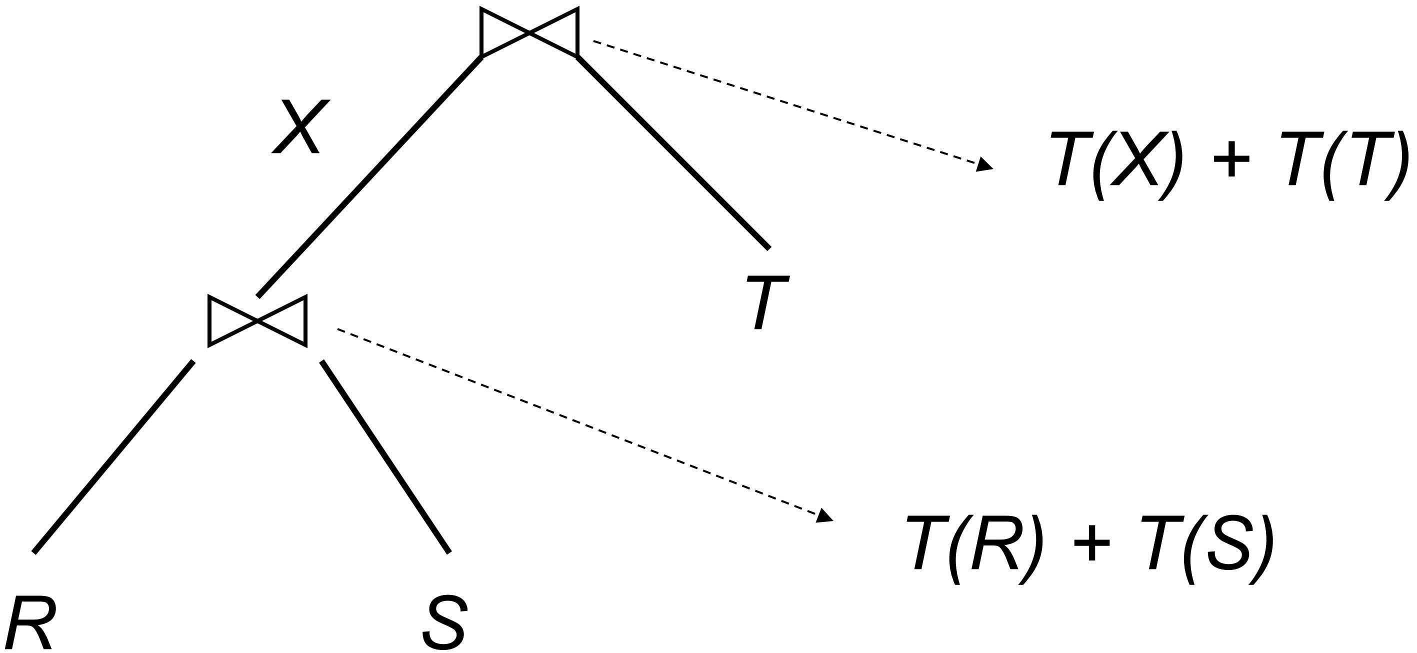
Simple Cost Model

$$\text{Cost}(R \bowtie S) = T(R) + T(S)$$

All other operators have 0 cost

Note: The simple cost model used for illustration only, it is not used in practice

Cost Model Example



$$\boxed{\text{Total Cost: } T(R) + T(S) + T(T) + T(X)}$$

Selinger Algorithm:

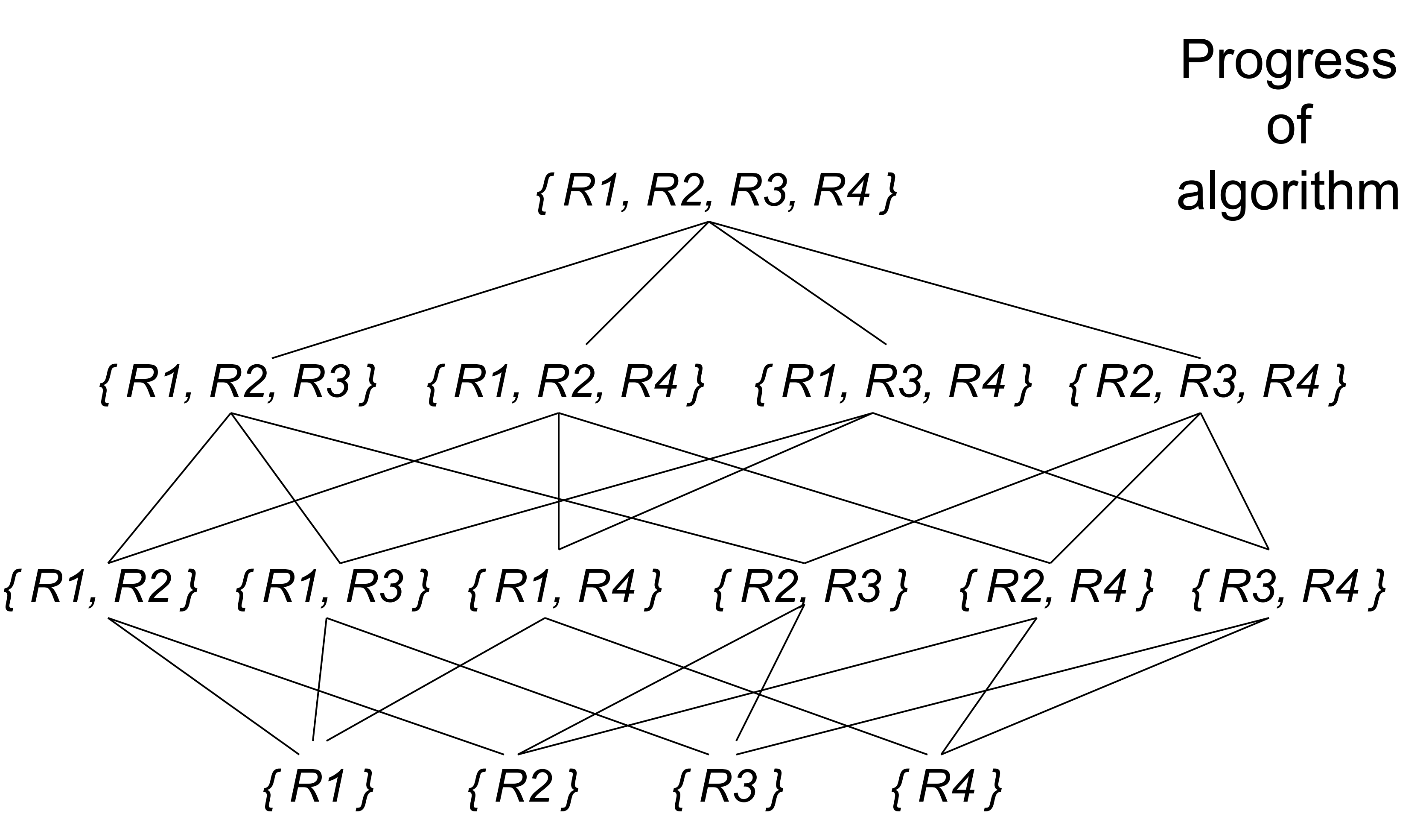
OPT ({ R1, R2, R3 }):

$$
\text{Min} \begin{cases} \text{OPT}(\{R1, R2\}) + T(\{R1, R2\}) + T(R3) \\ \text{OPT}(\{R2, R3\}) + T(\{R2, R3\}) + T(R1) \\ \text{OPT}(\{R1, R3\}) + T(\{R1, R3\}) + T(R2) \end{cases}
$$

Note: Valid only for the simple cost model

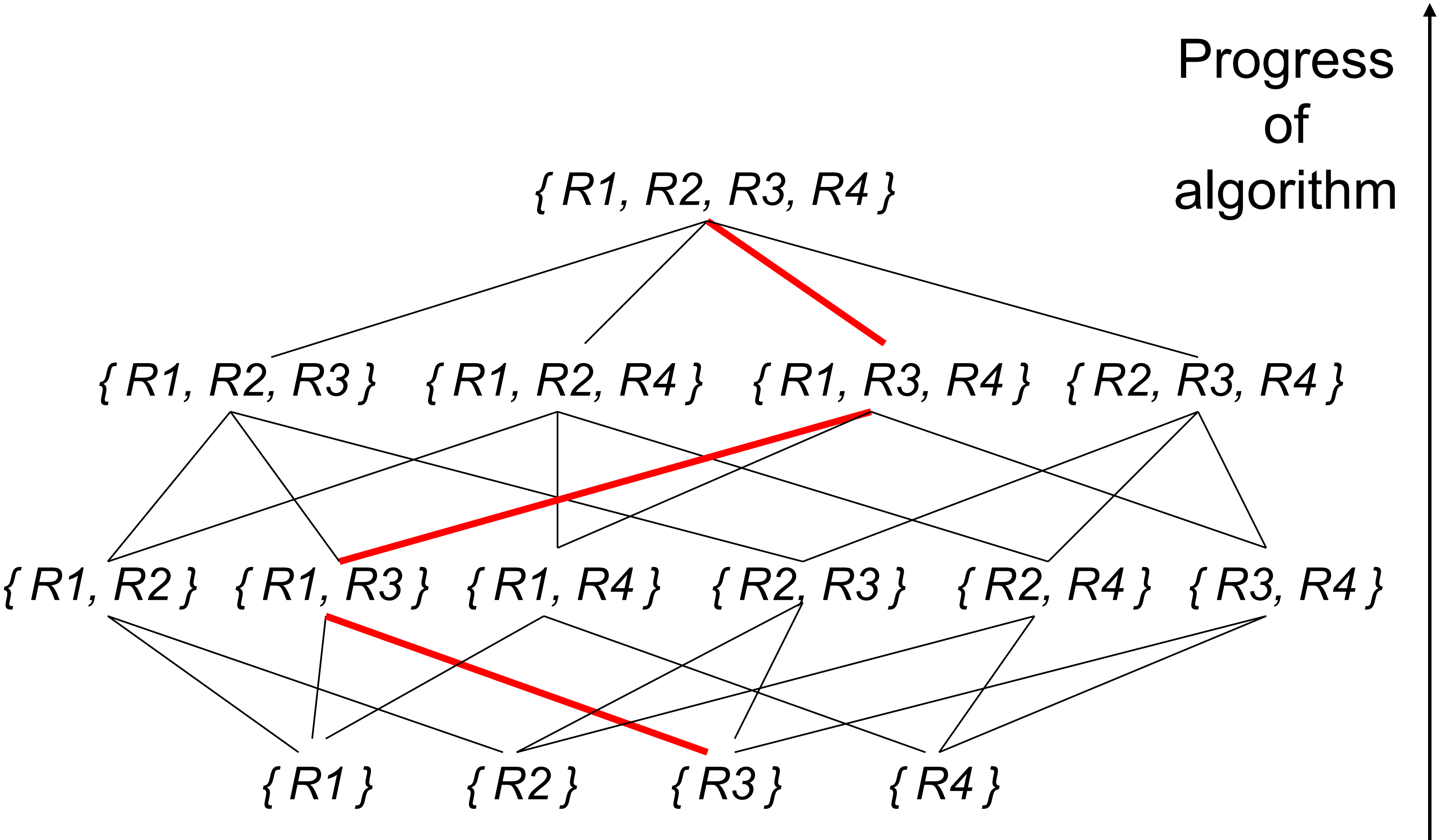
Selinger Algorithm:

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



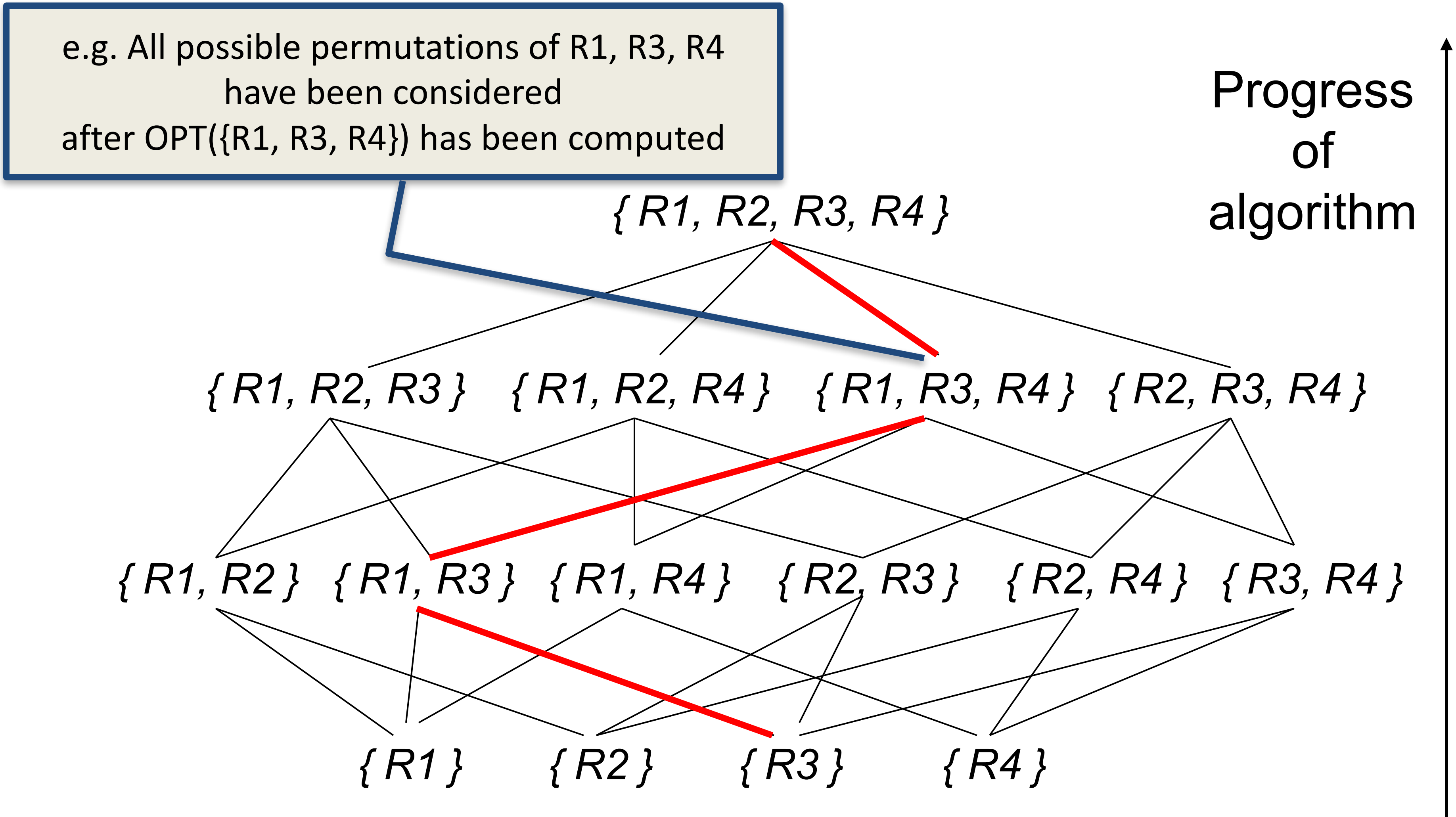
Selinger Algorithm:

Query: *R1* ⋈ *R2* ⋈ *R3* ⋈ *R4*



Selinger Algorithm:

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

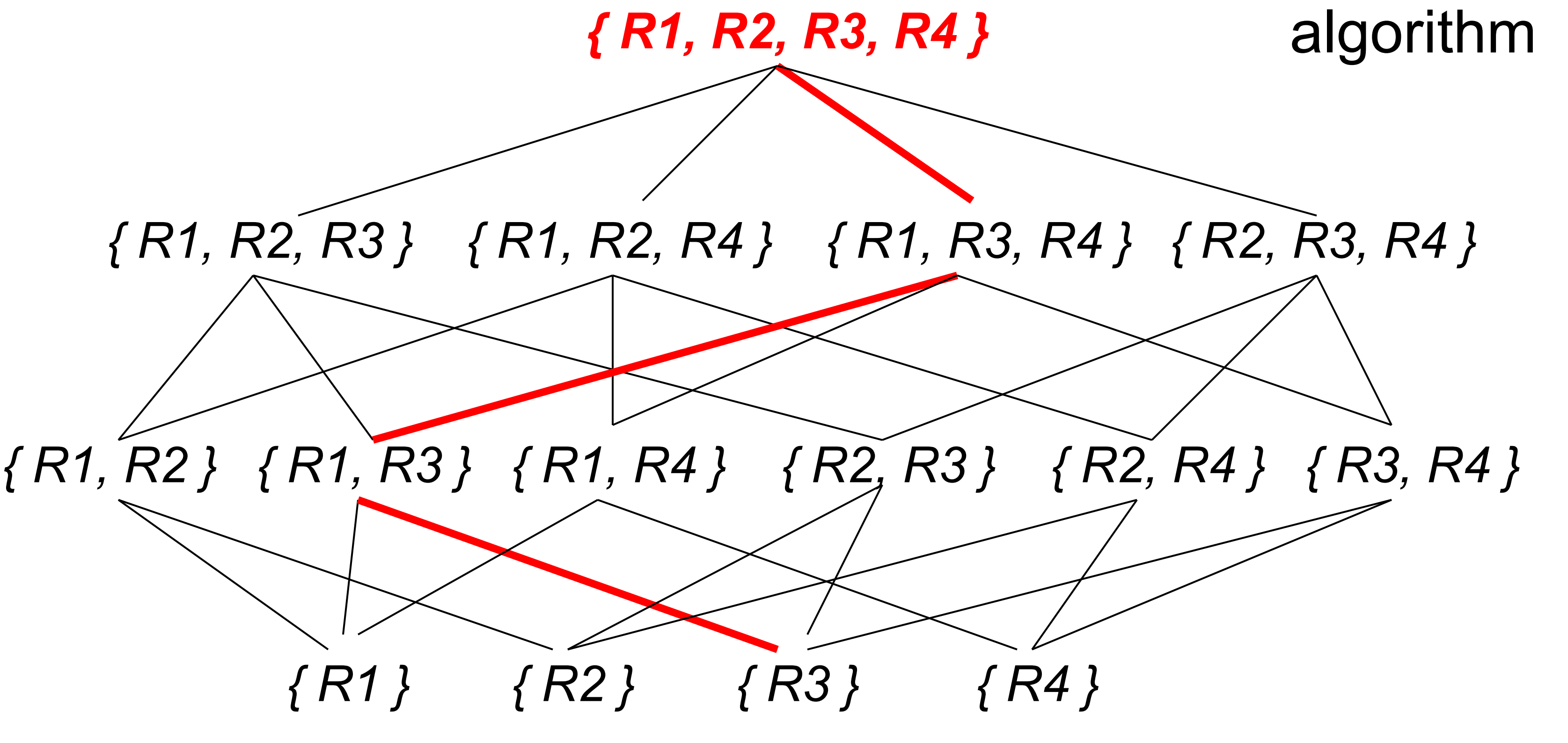


Selinger Algorithm:

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

Q. How to optimally compute join of {R1, R2, R3, R4}?

Ans: First optimally join {R1, R3, R4} then join with R2 as inner.

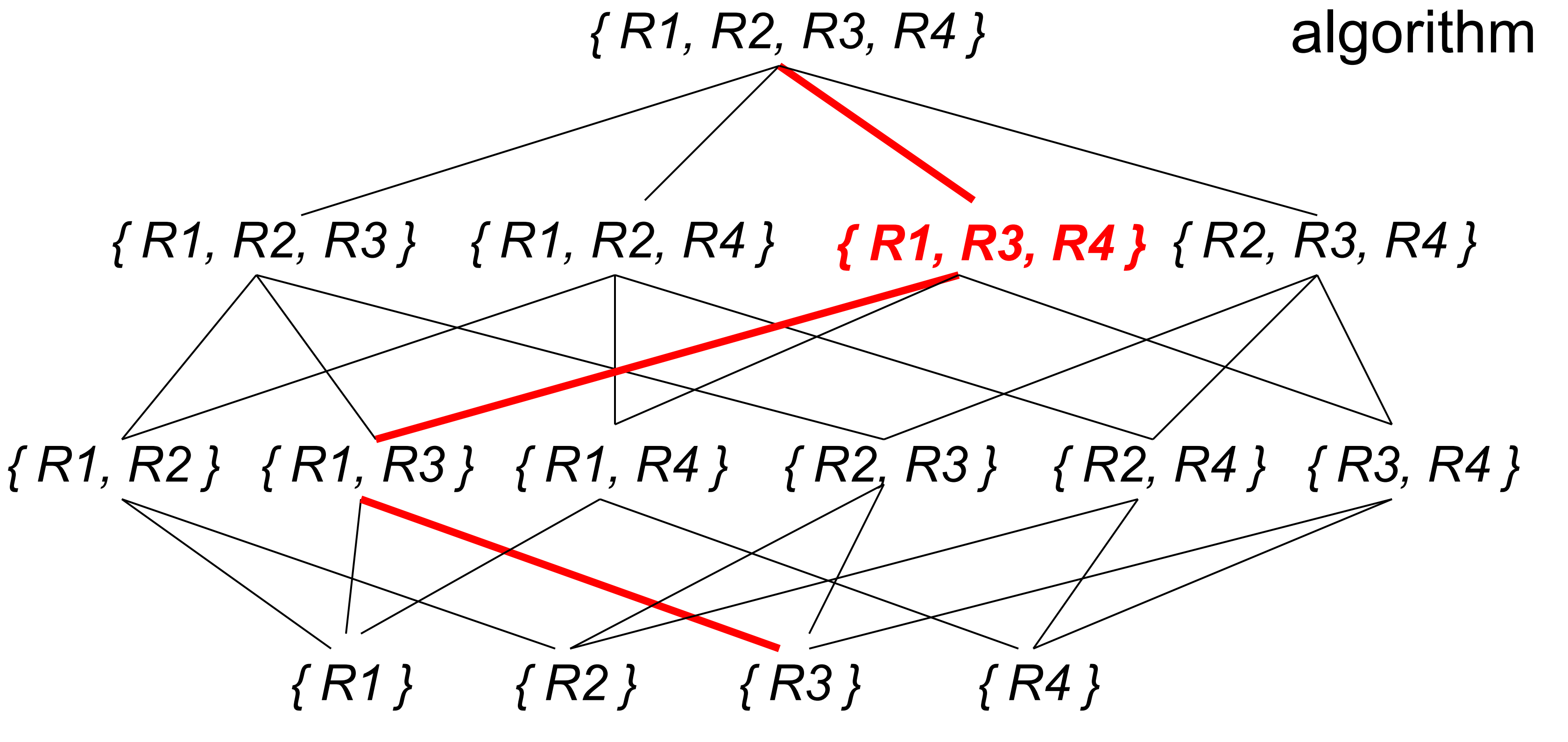


Selinger Algorithm:

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

Q. How to optimally compute join of {R1, R3, R4}?

Ans: First optimally join {R1, R3}, then join with R4 as inner.

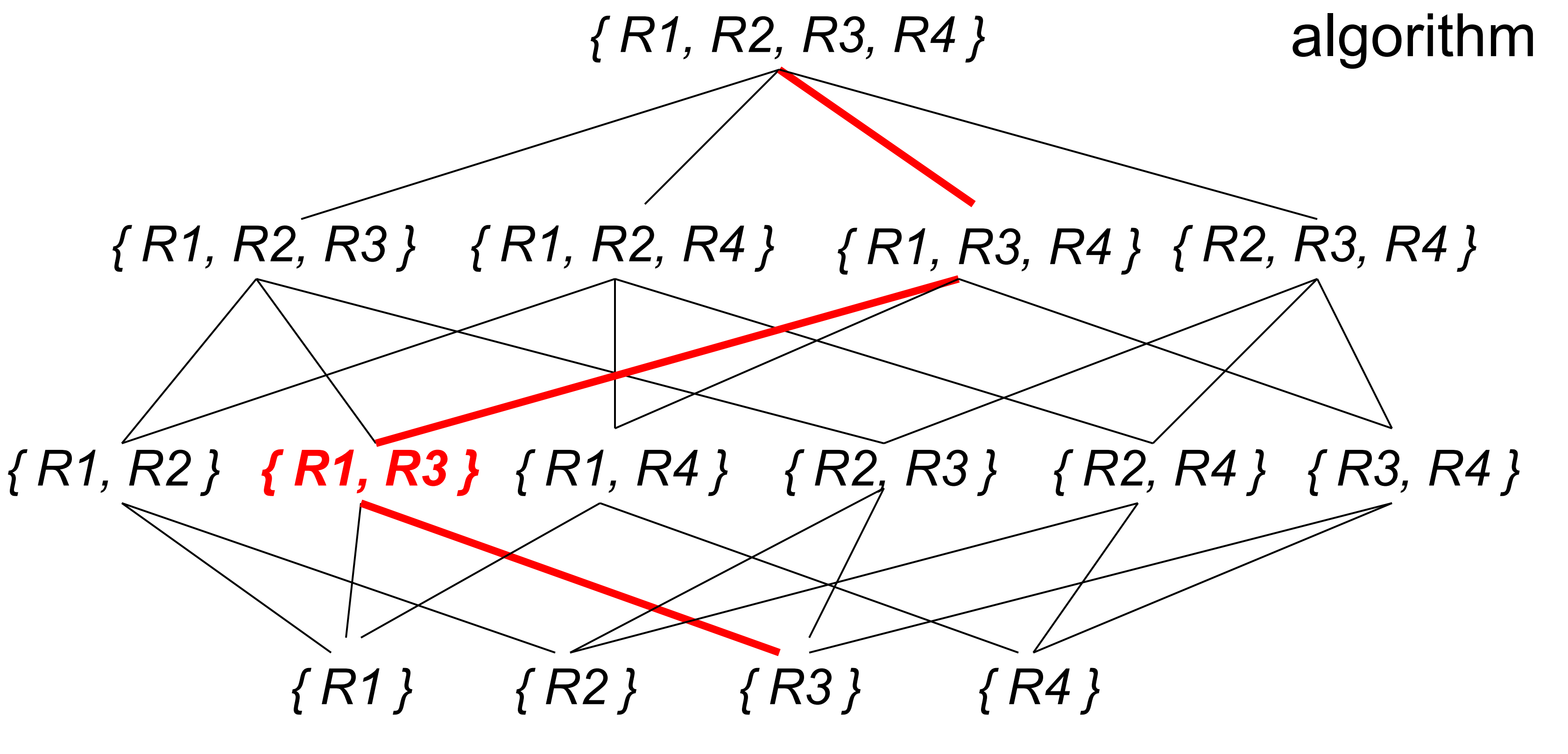


Selinger Algorithm:

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

Q. How to optimally compute join of {R1, R3}?

Ans: First optimally join {R3}, then join with R1 as inner.

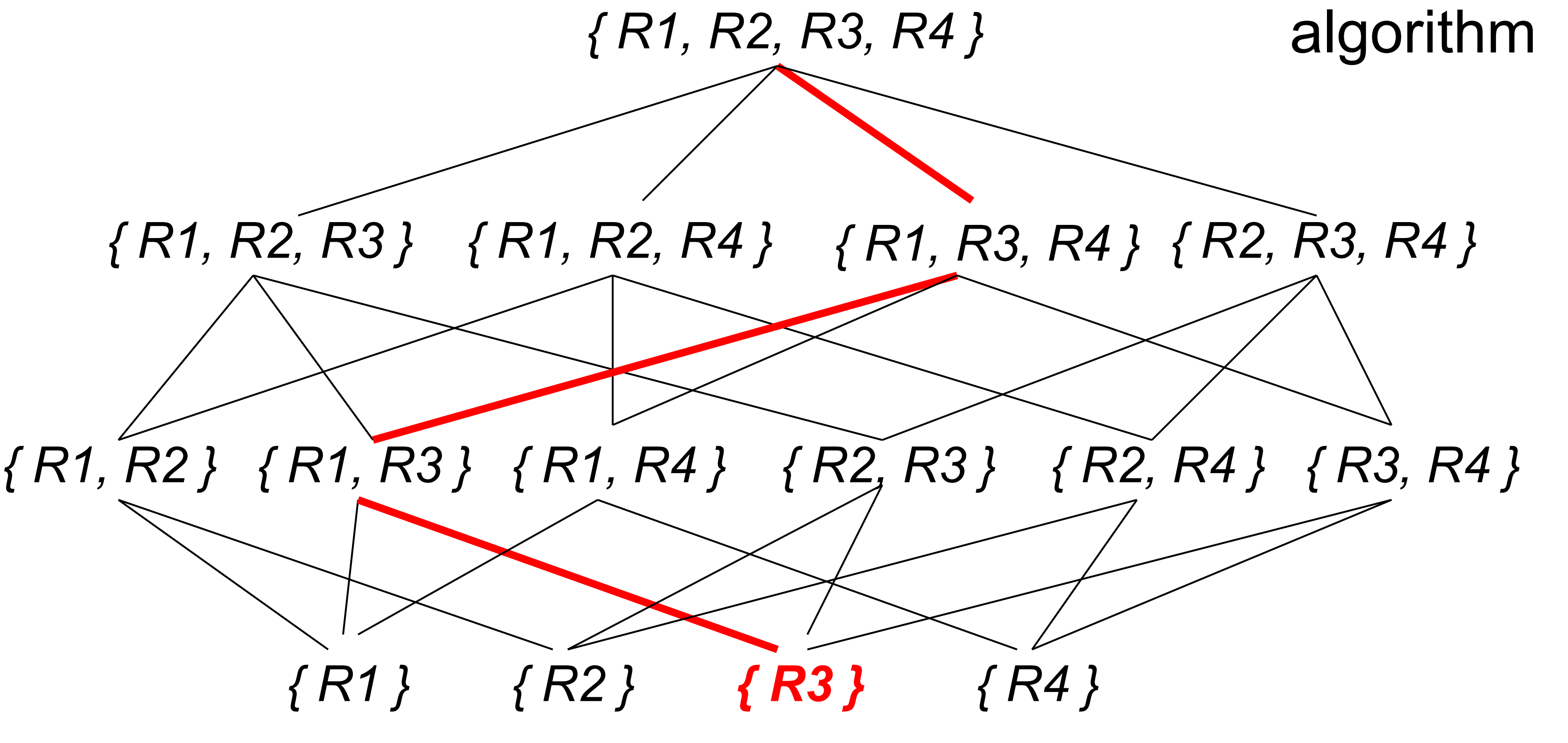


Selinger Algorithm:

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

Q. How to optimally compute join of {R3}?

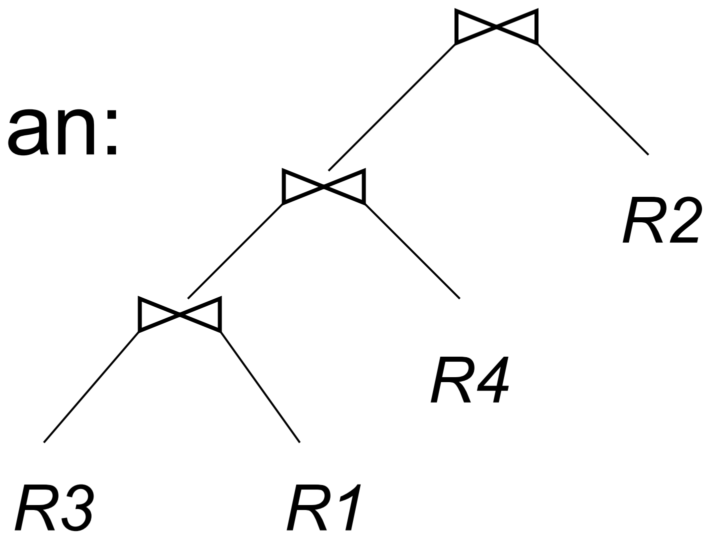
Ans: Single relation – so optimally scan R3.



Selinger Algorithm:

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

Final optimal plan:



NOTE : There is a one-one correspondence between the permutation (R3, R1, R4, R2) and the above left deep plan

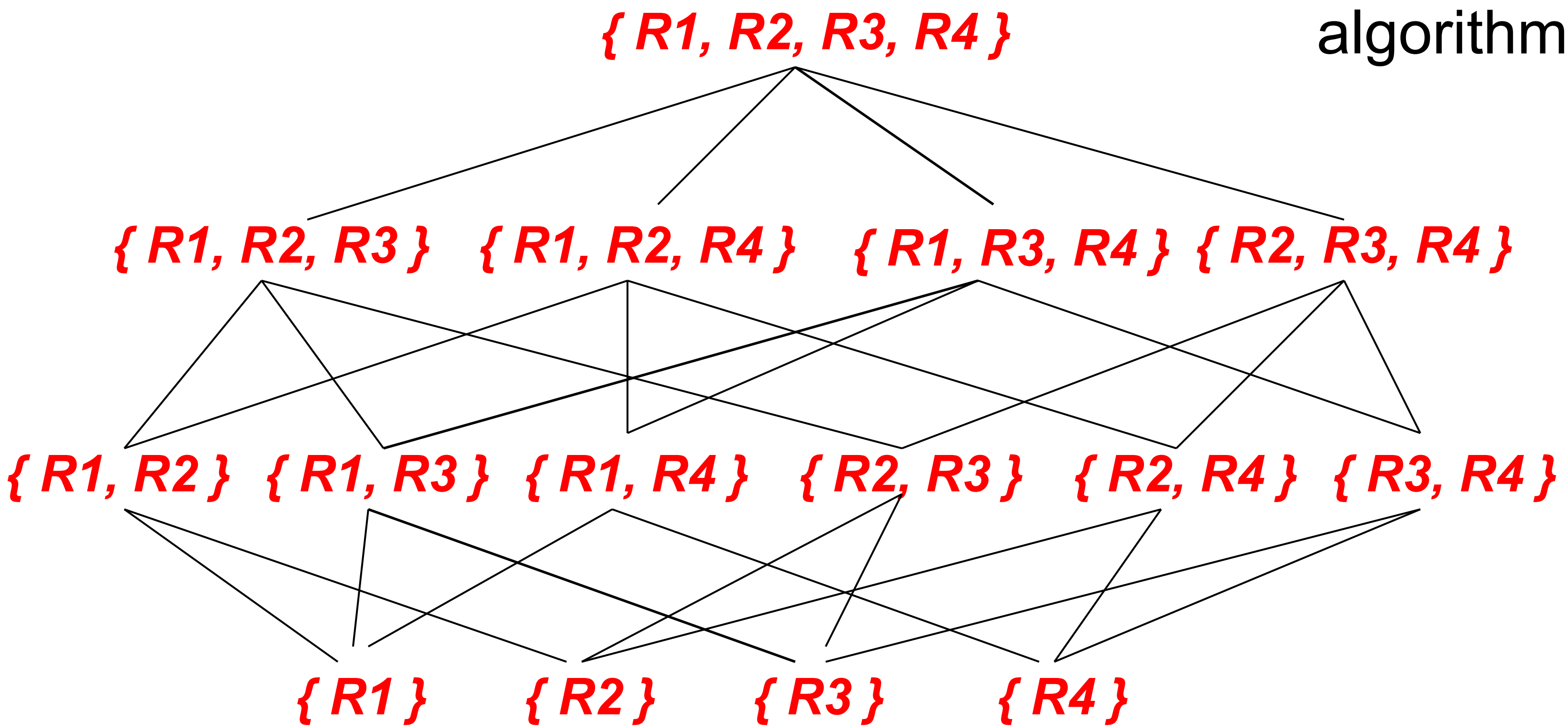
Selinger Algorithm:

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

NOTE: (*VERY IMPORTANT*)

- This is *NOT* done by top-down recursive calls.
- This is done BOTTOM-UP computing the optimal cost of *all* nodes in this lattice only once (dynamic programming).

Progress of algorithm



More on Query Optimizations

- See the survey (on course website):

"An Overview of Query Optimization in Relational Systems" by Surajit Chaudhuri

- Covers other aspects like
  - Pushing group by before joins
  - Merging views and nested queries
  - "Semi-join"-like techniques for multi-block queries
    - covered later in distributed databases
  - Statistics and optimizations
  - Starbust and Volcano/Cascade architecture, etc