CSE462/562: Database Systems (Fall 24)

Lecture 19: Query Optimization

11/19/2024 & 11/21/2024

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Last updated: 10/9/2024 3:00 PM

Query processing overview

ODBC/JDBC/ command line frontend

**

SQL Query

```
SELECT S.name,E.grade
FROM student S, enrollment E
WHERE S.sid = E.sid
  AND S.adm_year = 2021
  AND E.cno = 562;
```

*SQL Parser**

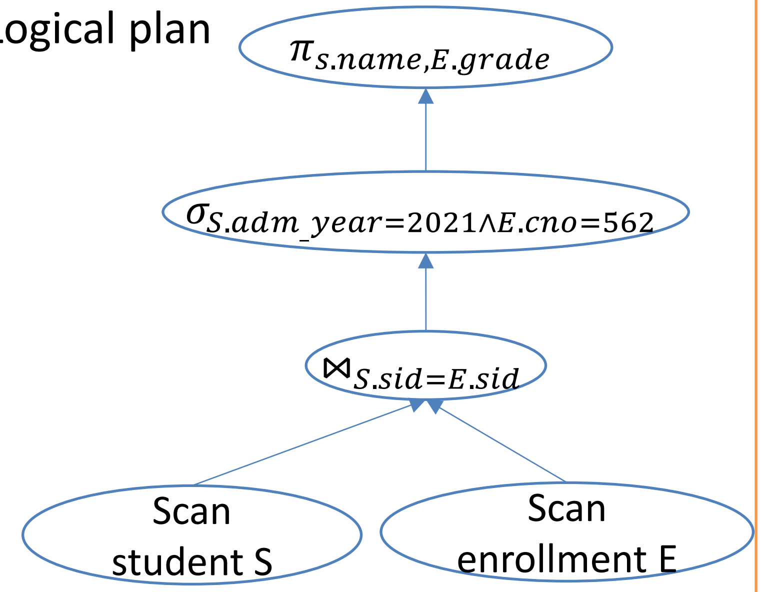
** include multiple intermediate steps (e.g., parsing tree/analysis/rewriting)*

(Extended) Relational Algebra

$\pi_{S.name,E.grade}\sigma_{S.adm\_year=2021 \wedge E.cno=562} S \bowtie_{S.sid=E.sid} E$

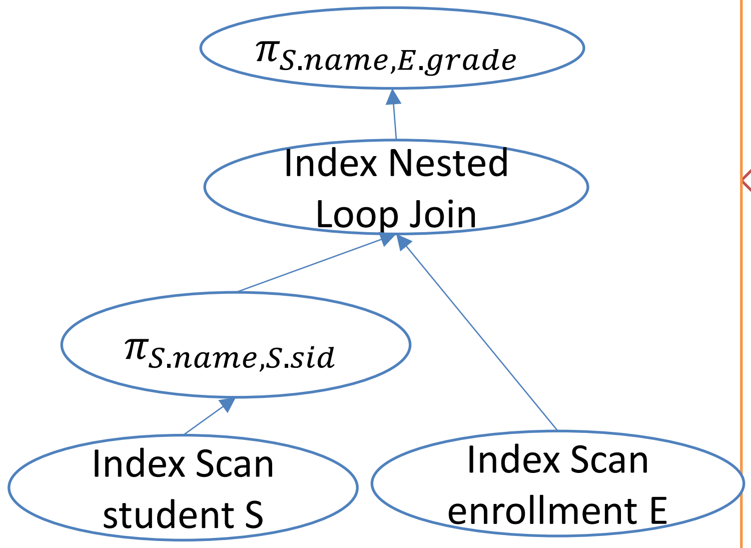
Internally represented as

Logical plan



Query Optimizer

Physical plan



Query Execution

Query result

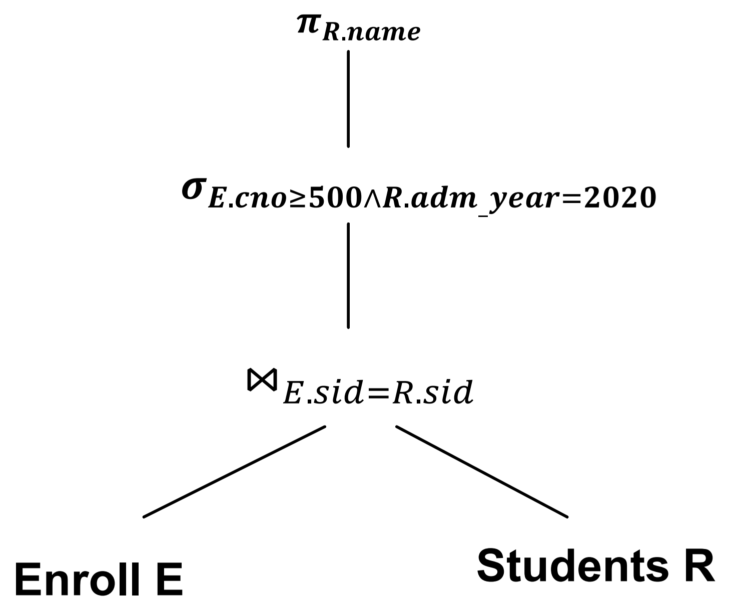
```
S.name  | E.grade
Alice   | 4.0
Charlie| 2.3
(2 rows)
```

Query optimization overview

- Query can be converted to relational algebra
- Relational Algebra converted to tree, joins as branches
- Each operator has implementation choices
- Operators can also be applied in different order!

```
SELECT  R.name
FROM  Enroll E, Students R
WHERE  E.sid=R.sid AND
   E.cno>=500 AND R.adm_year = 2020
```

$$\pi_{R.name}\sigma_{E.cno=562 \wedge E.grade \geq 3.0} E \bowtie_{E.sid=R.sid} R$$

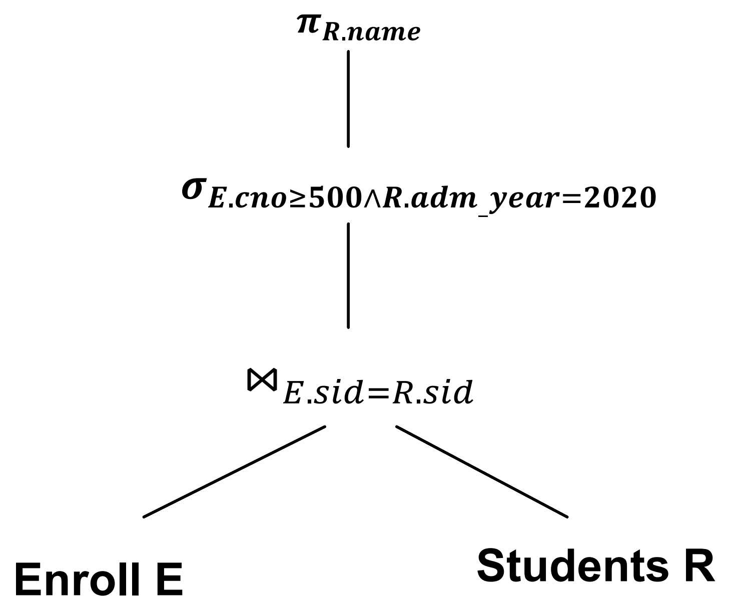


Query optimization overview

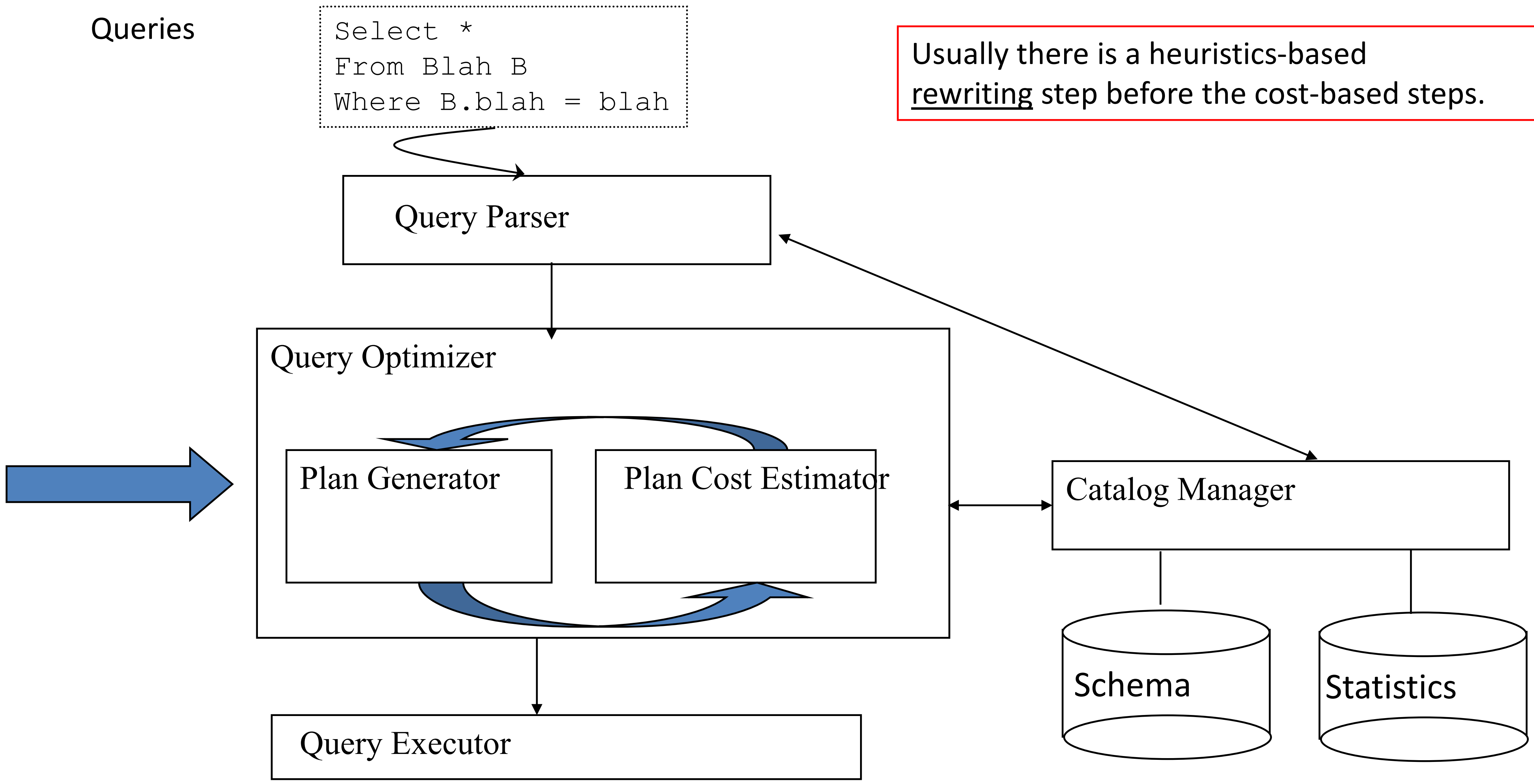
- ***Plan:*** *Tree of R.A. ops (and some others) with choice of algorithm for each op.*
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.
- Two main issues:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find best plan.
- Reality: Avoid worst plans!

Relational operators have a uniform *iterator* interface:

open( ), get_next( ), close( )



Cost-based query optimizer



```
Select *
From Blah B
Where B.blah = blah
```

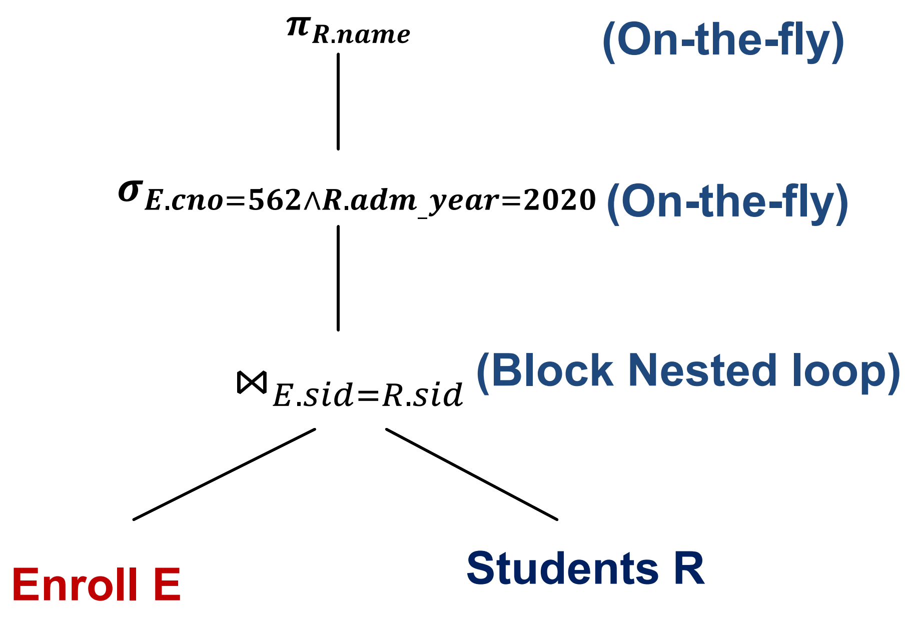
Usually there is a heuristics-based rewriting step before the cost-based steps.

Running example

- Notations: for relation R
  - T_R: number of records, N_R: number of pages in its heap file, B_R: (average) number of tuples per page
  - h_I: height of a B-tree index I over the file
  - M: private workspace size in pages
- Running example
  - Student: R(sid: int, name: varchar(19), login: varchar(19), major: char(2), adm_year: int)
    - 50 bytes/tuple, $B_R = 80$, $T_R = 40{,}000$, $N_R = 500$
    - Assume the student records in the table span 10 years (between 2012 and 2022)
  - Enrollment: E(<u>sid: int, semester: char(4), cno: int</u>, grade: double)
    - 20 bytes/tuple, $B_E = 200$, $T_E = 200{,}000$, $N_E = 1000$
    - Assume 50% of the enrollment records belong to the graduate level (>=500) courses
- Consider a simplified cost model: *cost = #page_transfers* *(i.e., ignoring the random seeks)*
  - Often good enough for approximating the trend of the cost relative to data size
  - *Correct size estimation* is key to a correct comparison of costs
- Assume we have 5 pages in the buffer

Motivating example

```
SELECT  R.name
FROM  Enroll E, Students R
WHERE  E.sid=R.sid AND
   E.cno=562 AND R.adm_year = 2020
```



Cost = 1000 + 1000 * 500 = 501,000 I/Os

- By no means the worst plan!
- Misses several opportunities: selections could have been `pushed' earlier, no use is made of any available indexes, etc.
- *Goal of optimization:* To find more efficient plans that compute the same answer.

Relational algebra equivalence

- Rules that allow the optimizer to transform a logical plan into an equivalent plan with the *same* output over any database instance
- *Selections:*
  - Cascade: $\sigma_{\theta_1 \wedge \theta_2} E \equiv \sigma_{\theta_1} \sigma_{\theta_2} E$
  - Commutative: $\sigma_{\theta_1} \sigma_{\theta_2} E \equiv \sigma_{\theta_2} \sigma_{\theta_1} E$
- *Projections:*
  - Cascade: $\pi_{A_1} \pi_{A_2} \ldots \pi_{A_n} E \equiv \pi_{A_1}(E)$ where $A_1 \subseteq A_2 \subseteq \cdots \subseteq A_n$
    - Only need to perform the final projection in a sequence of projections
- *(Inner) Joins or Cartesian product:*
  - Commutative: $E_1 \bowtie_\theta E_2 \equiv E_2 \bowtie_\theta E_1$ (allows switching the inner and outer)
  - Associative
    - Special case natural join: $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$
    - General theta join: $\left(E_1 \bowtie_{\theta_1} E_2\right) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} \left(E_2 \bowtie_{\theta_2} E_3\right)$
  - Implication: inner joins can be done in any order!
    - Join reordering: an important optimization step in DBMS

Assuming θ_2 only involves fields in E_2 and E_3

Relational algebra equivalence

- Rules for more than one operator
  - *Selection can be combined with inner join/cartesian product*

    $$\sigma_{\theta_1}\left(E_1 \bowtie_{\theta_2} E_2\right) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

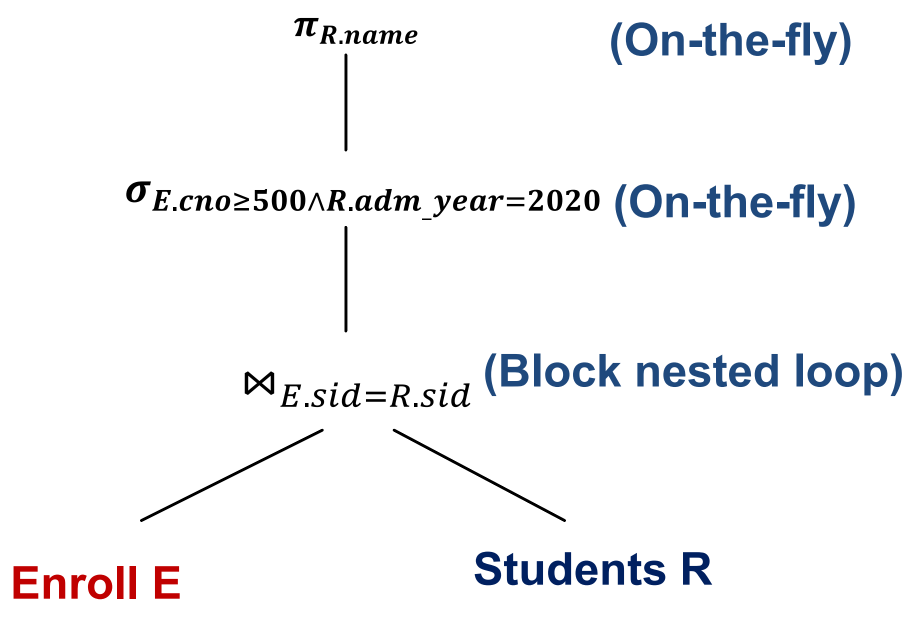
  - Projection push-down: select/join and projection commutes (provided that the predicate only involves the projected fields)
    $\pi_A \sigma_\theta E \equiv \sigma_\theta \pi_A E$ when $Var(\theta) \subseteq A$
    $\pi_{A_1 \cup A_2}(E_1 \bowtie_\theta E_2) \equiv \pi_{A_1} E_1 \bowtie_\theta \pi_{A_2} E_2$ when $Var(\theta) \subseteq A_1 \cup A_2$ and A_1, A_2 only involve fields from E_1, E_2, resp.

  - Selection push-down: join and select commutes (provided that the selection predicate only involves attributes from one side)
    $\sigma_{\theta_1}(E_1 \bowtie_\theta E_2) \equiv \left(\sigma_{\theta_1} E_1\right) \bowtie_\theta E_2$ when $Var(\theta_1) \subseteq A(E_1)$ (set of fields in E_1)
- More rules about other operators, e.g., aggregation, set operations, sort, …

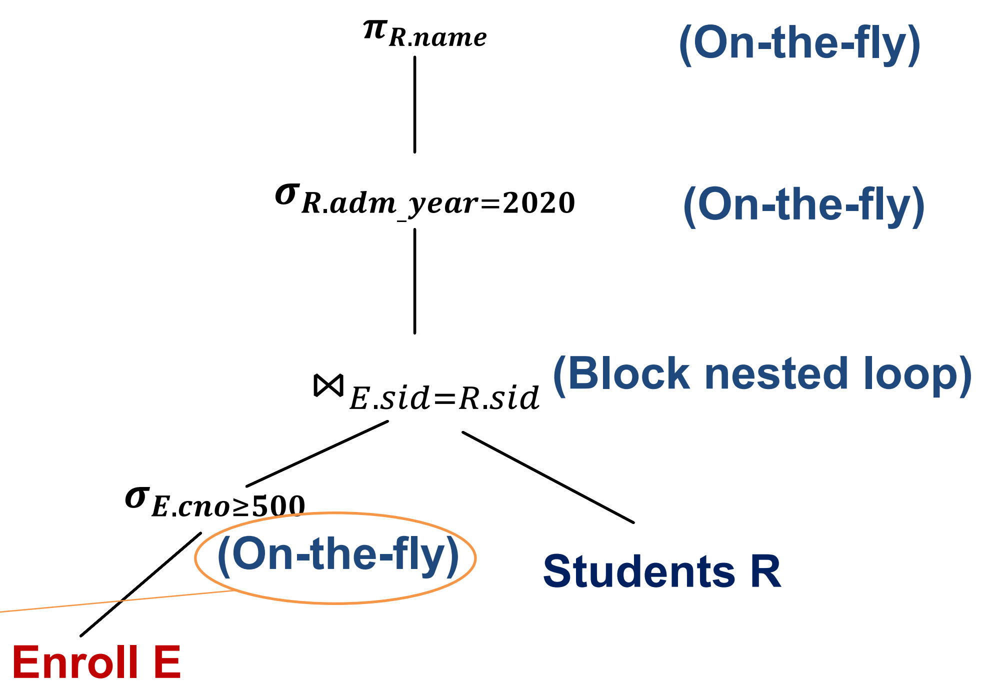
- Note: rules involving outer joins may be different
  - Exercise: Can we always push selection through outer joins? What about projections?

Selection push-down (no index)

- Heuristics 1: perform selections as early as possible
  - Selection is often very cheap or "free" (in I/O only cost model)
  - reduces intermediate size



Cost = 501,000 I/Os

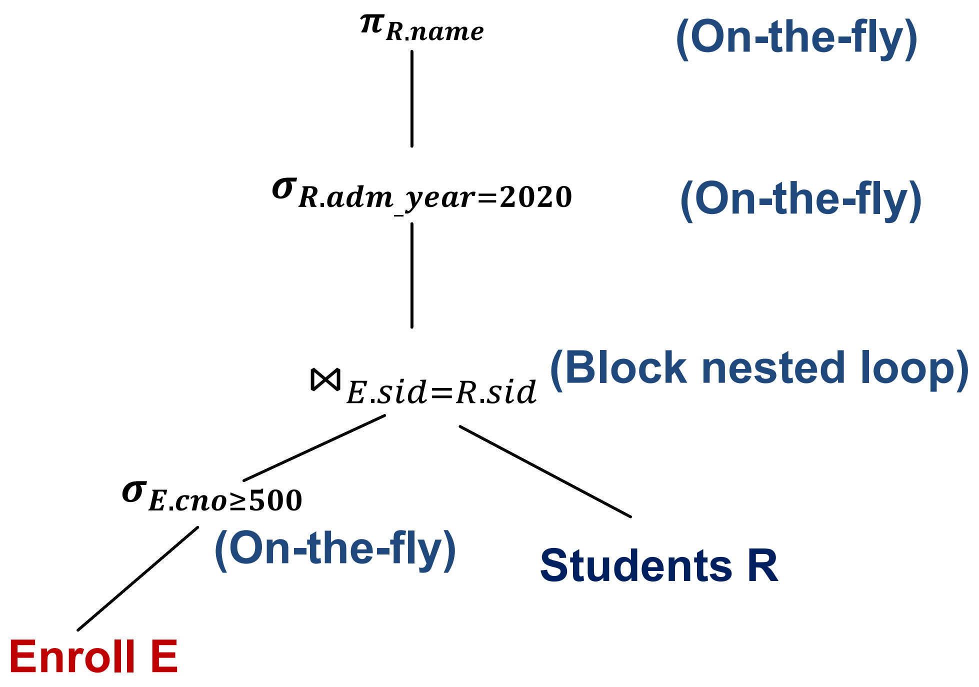


Collect one page from the outer plan, rather than the underlying scan.

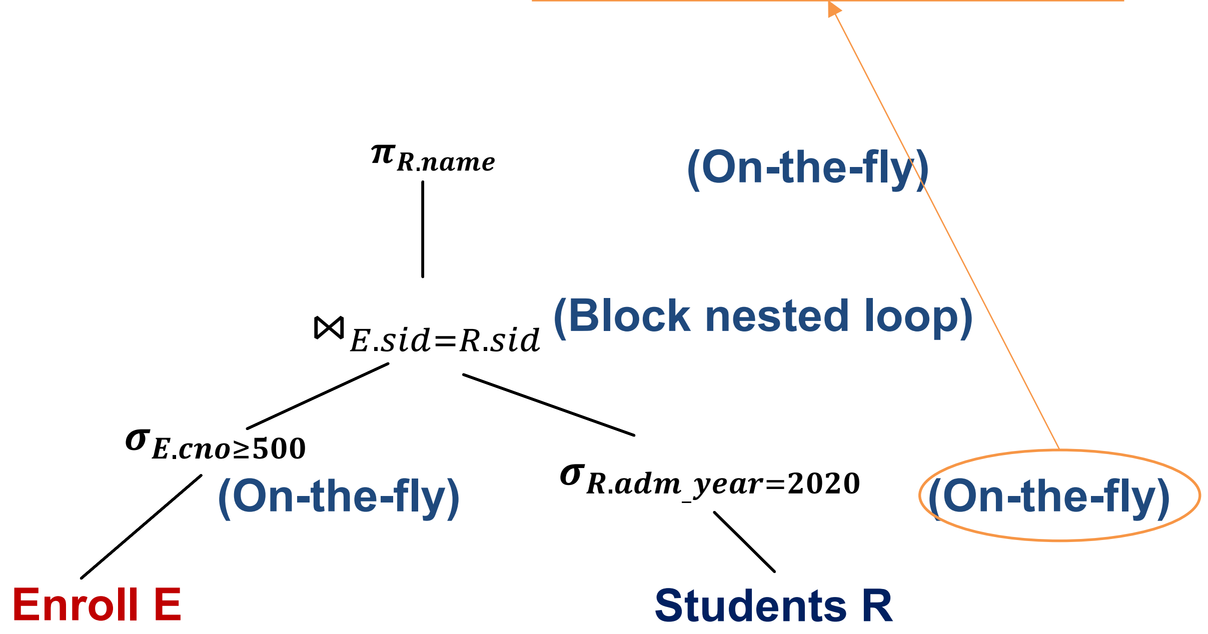
Cost = $1000 + [1000 * 0.5] \times 500 = 251{,}000$ I/Os

Selection push-down (no index)

- Can also push-down on the other side



Cost = 251,000 I/Os

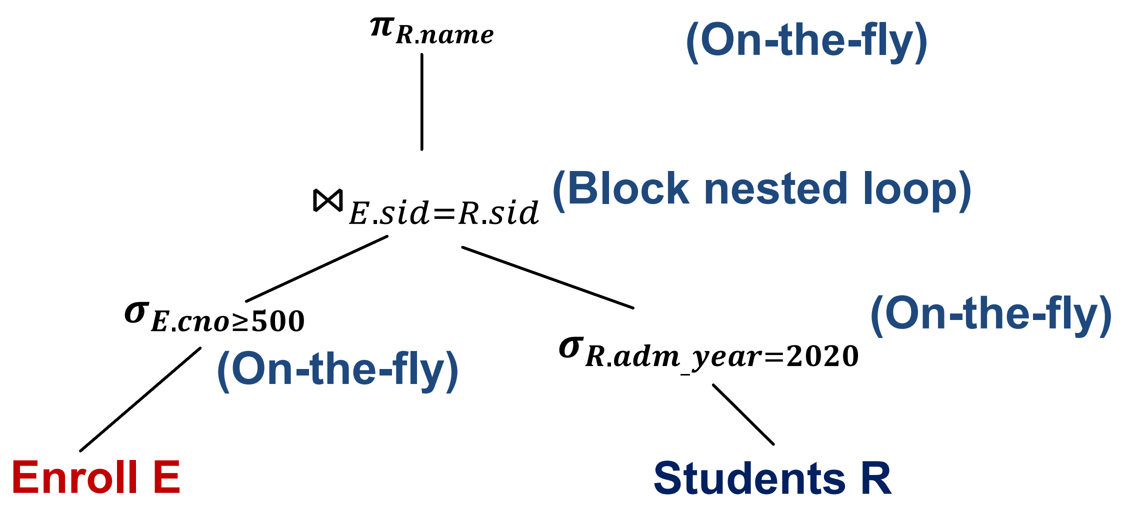


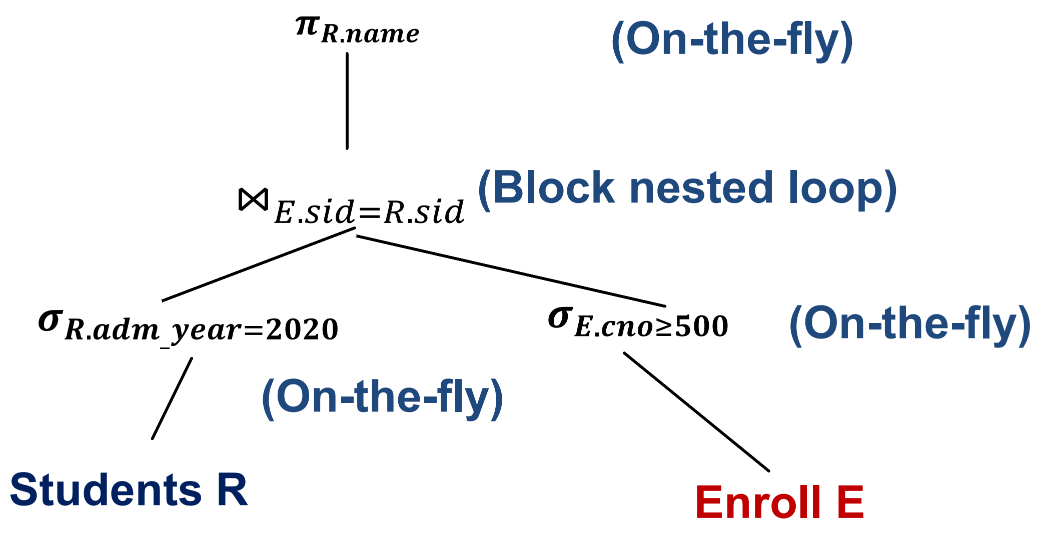
No impact on I/O because BNL scans the inner plan once for every outer block.

Cost = 251,000 I/Os

Join reordering

- Different join ordering may result in different cost
  - even if we use the same join algorithm
  - *Generally, the outer plan should have a smaller output in BNL*
    - what about hash join/sort merge join?



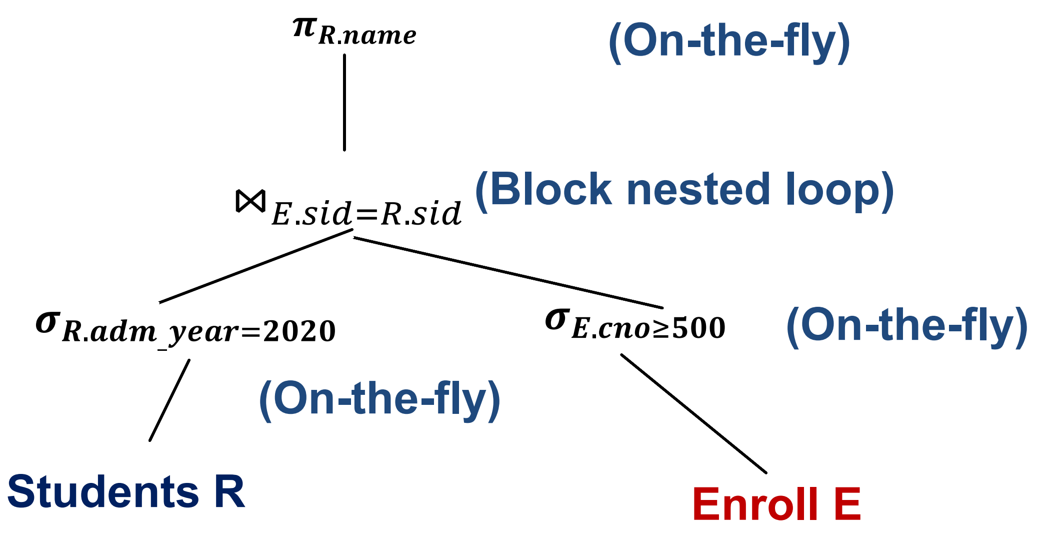


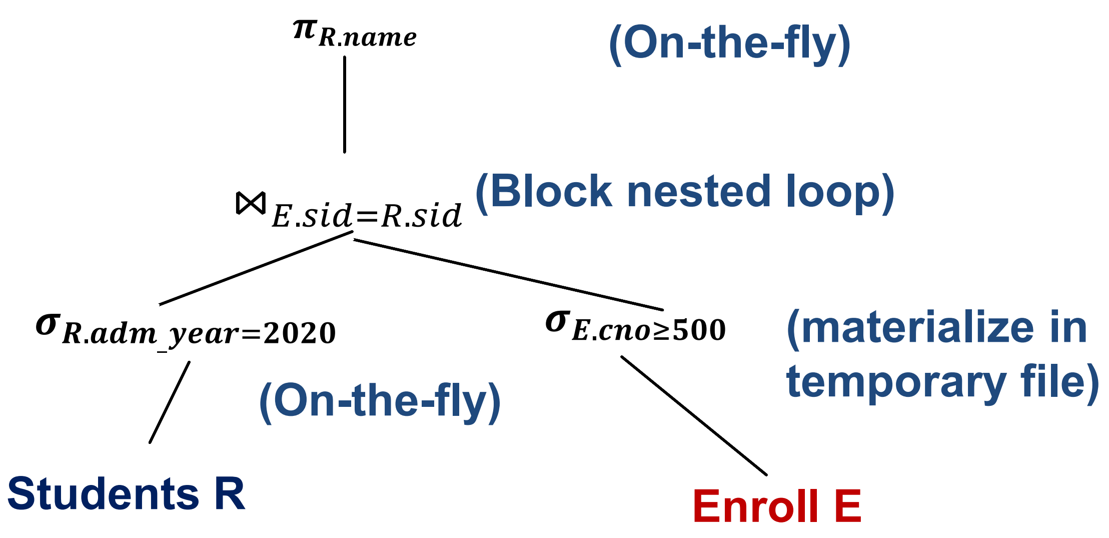
Cost = $251{,}000$ I/Os

Cost = $500 + [500 \times 0.1] \times 1000 = 50{,}500$ I/Os

Materialization of inner plan

- We can also choose to materialize the inner plan for BNL to save repeated scan on the original relation



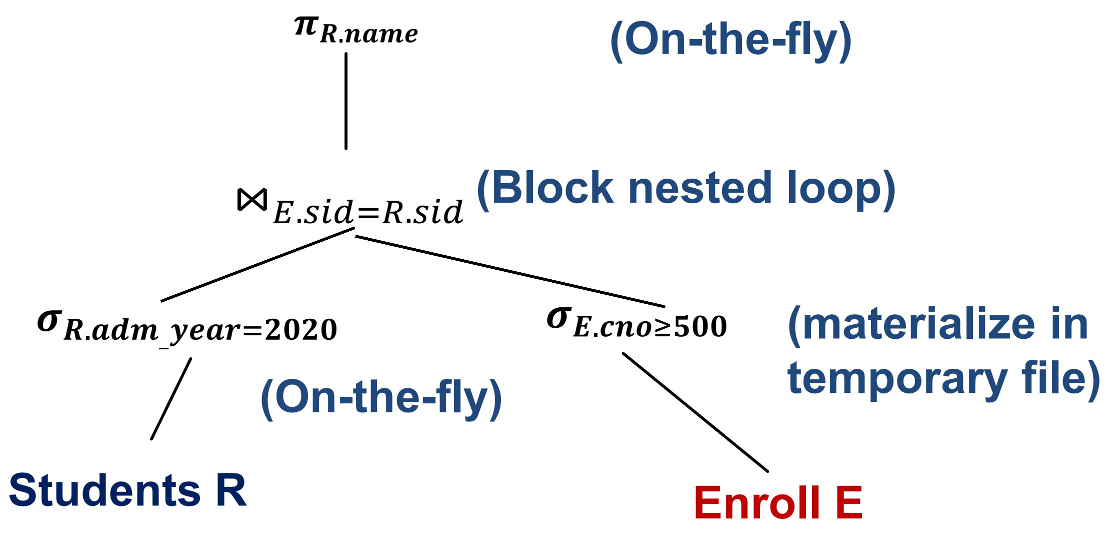


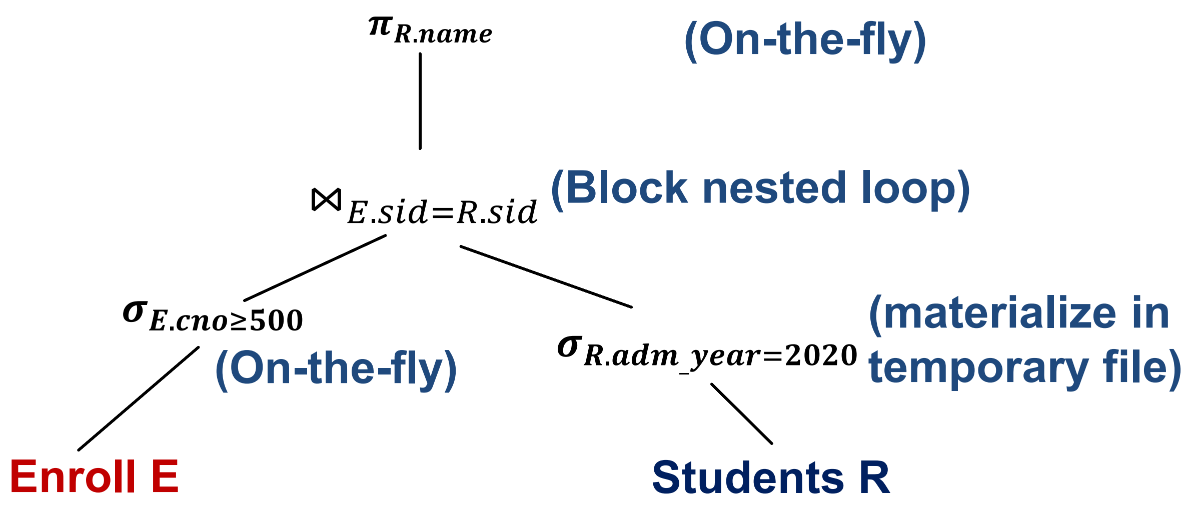
Cost = $50{,}500$ I/Os

Cost = $\underbrace{1000 + [1000 * 0.5]}_{\text{materializing inner plan}} + \underbrace{500}_{\text{BNL outer scan}} + \underbrace{[500 * 0.1] * [1000 * 0.5]}_{\text{BNL inner scan}} = 27{,}000$ I/Os

Materialization of inner plan

- Sometimes with materialization, it might be cheaper to use the larger plan as the outer





Cost = $1000 + [1000 * 0.5] + 500 + [500 * 0.1] * [1000 * 0.5]$
$= 27{,}000$ I/Os

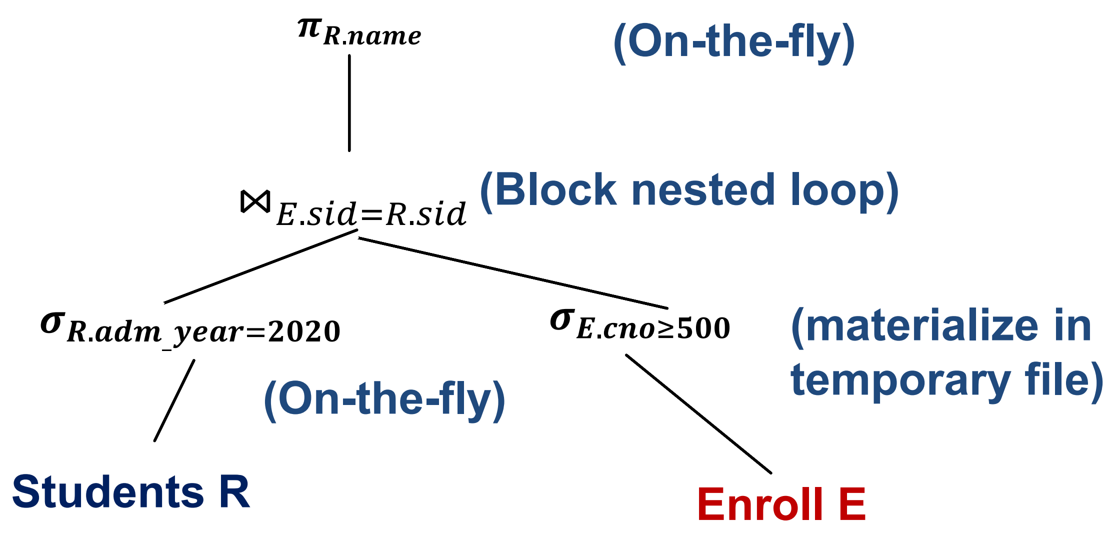
Cost = $500 + [500 * 0.1] + 1000 + [1000 * 0.5] * [500 * 0.1]$
$= 26{,}550$ I/Os

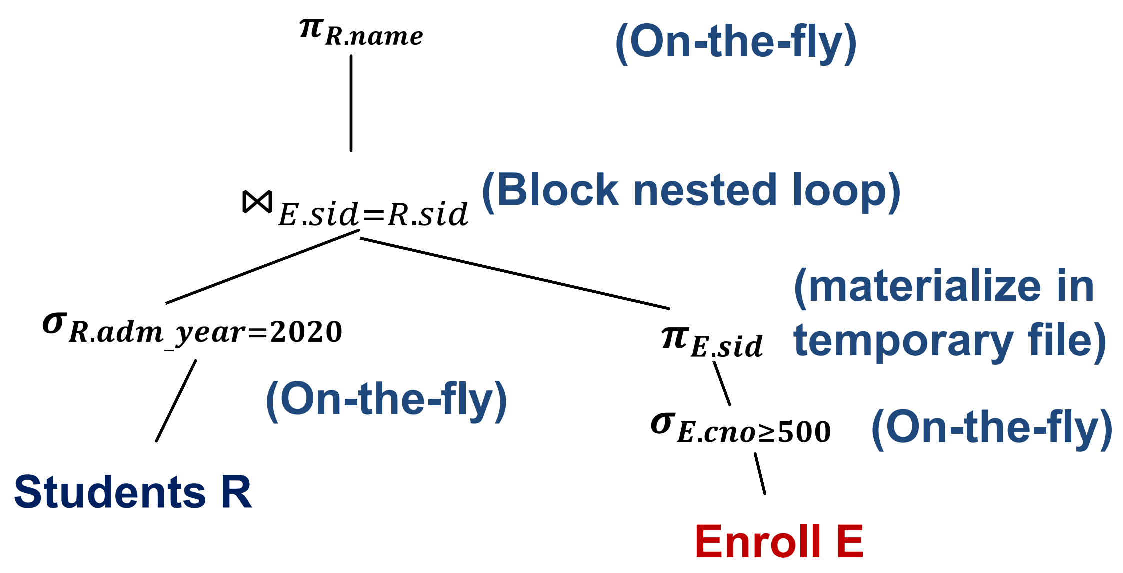
Projection push-down

- Heuristics 2: apply projection as early as possible
  - helps if materializing plan output

Enrollment: E(sid: int, semester: char(3), cno: int, grade: double)

20 bytes/tuple => $\pi_{E.sid}$: $\frac{4}{20} = 20\%$ in size after projection





Cost = $1000 + [1000 * 0.5] + 500 + [500 * 0.1] * [1000 * 0.5]$
$= 27{,}000$ I/Os

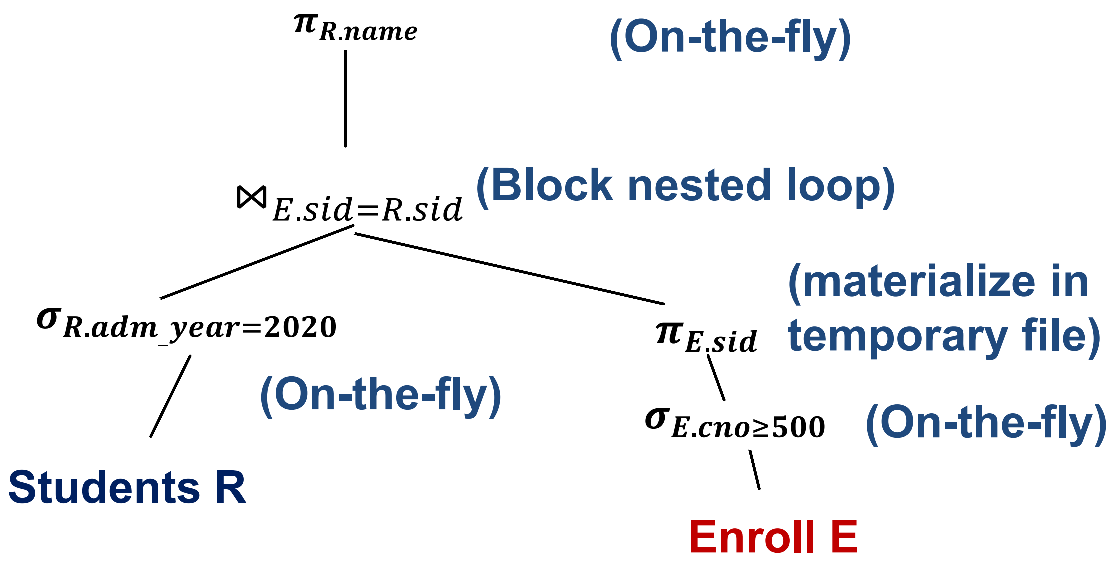
Cost = $1000 + [1000 * 0.5 * 0.2]$
$+ 500 + [500 * 0.1] * [1000 * 0.5 * 0.2]$
$= 6{,}600$ I/Os

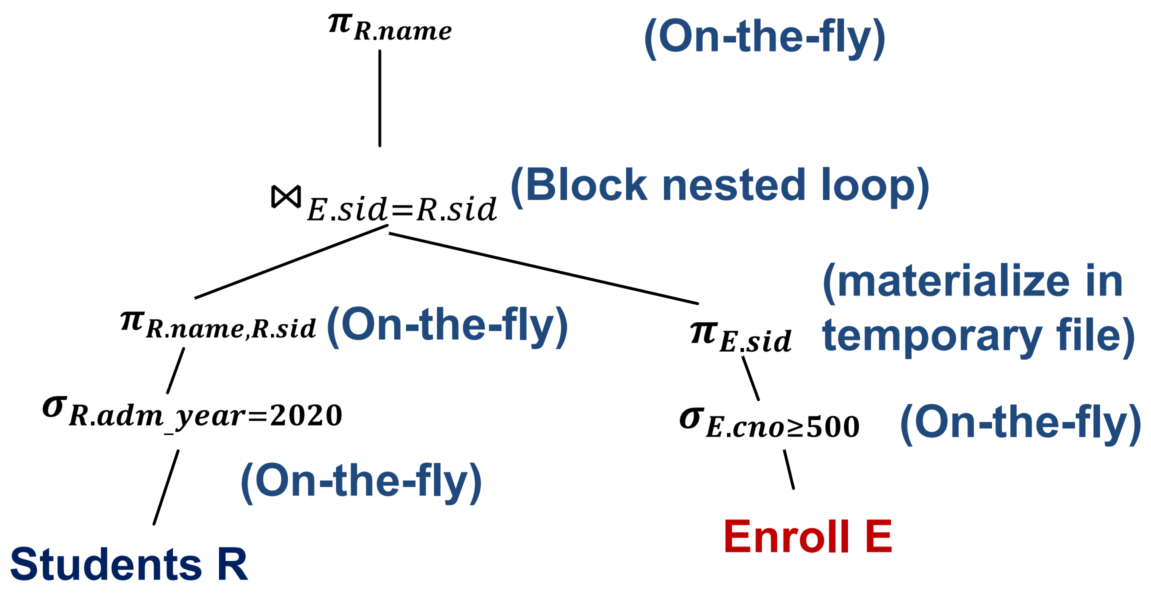
Projection push-down

- More projection push-down on the other side

R(sid: int, name: varchar(19), login: varchar(19), major: char(2), adm_year: int)

50 bytes/tuple => $\pi_{R.name,R.sid}$: $\frac{4+19+1}{50} = 48\%$ -- assuming VARCHAR uses '\0' at the end



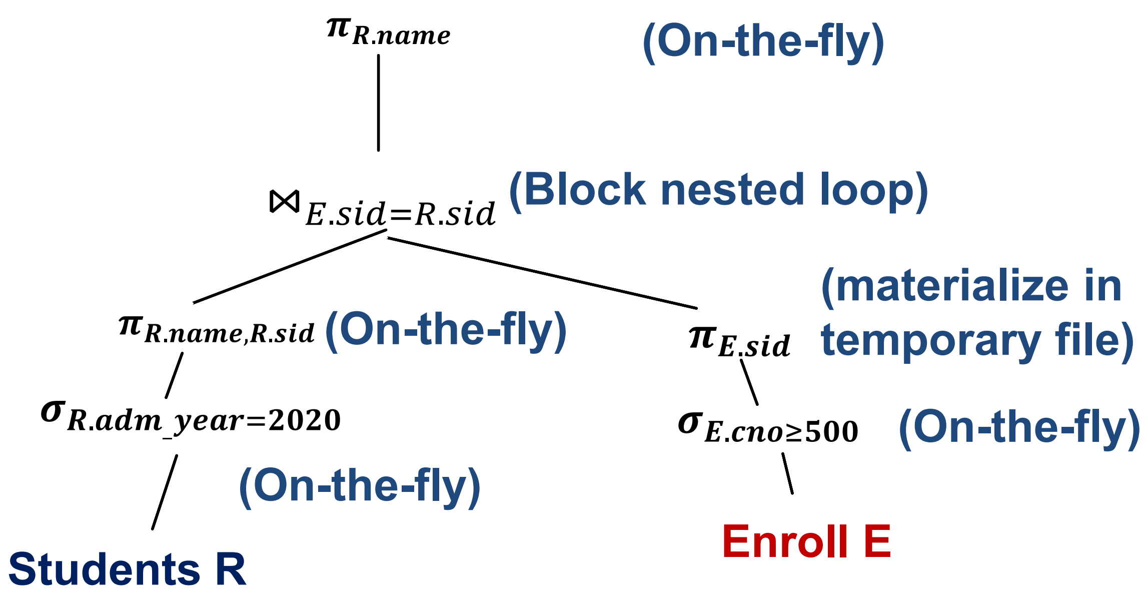


Cost = $1000 + [1000 * 0.5 * 0.2]$
$+500 + [500 * 0.1] * [1000 * 0.5 * 0.2]$
$= 6{,}600$ I/Os

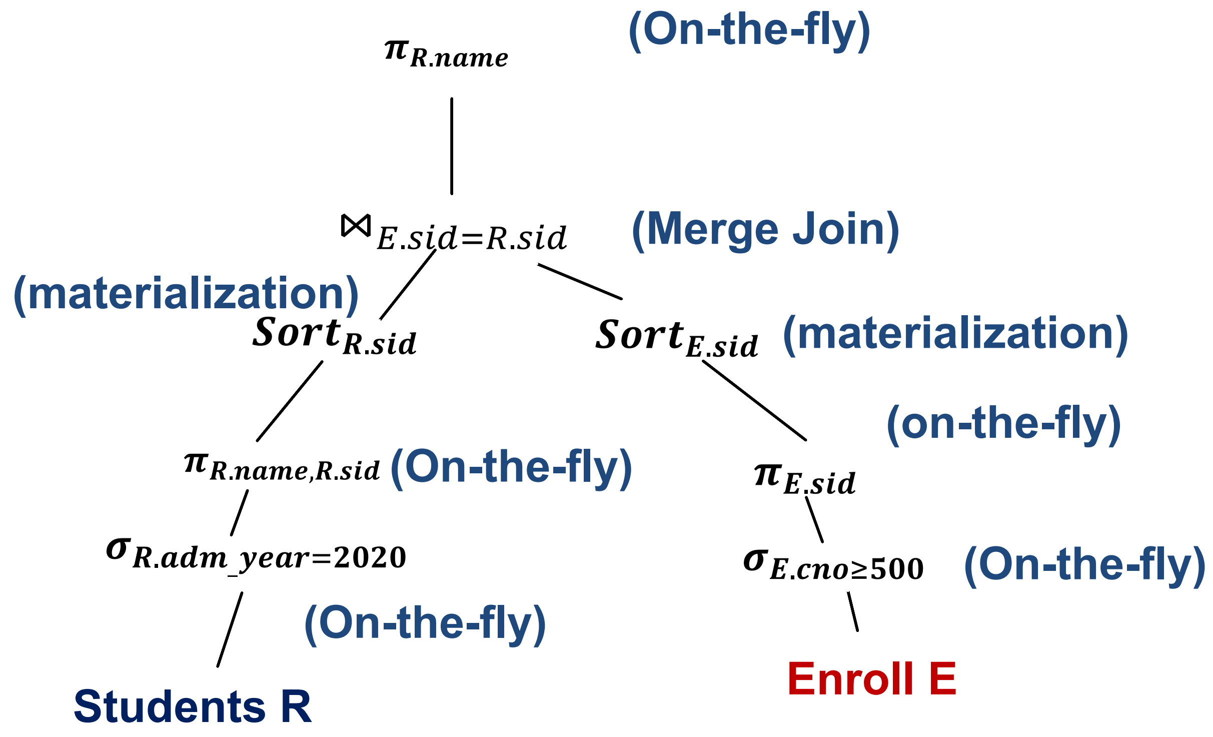
Cost = $1000 + [1000 * 0.5 * 0.2]$
$+500 + [500 * 0.1 * 0.48] * [1000 * 0.5 * 0.2]$
$= 4{,}000$ I/Os

Choice of join algorithms

- If we switch to sort-merge join with 5 buffers



$\pi_{R.name}$ (On-the-fly)
$\bowtie_{E.sid=R.sid}$ (Block nested loop)
$\pi_{R.name,R.sid}$ (On-the-fly)
$\pi_{E.sid}$ (materialize in temporary file)
$\sigma_{R.adm\_year=2020}$
$\sigma_{E.cno\geq 500}$ (On-the-fly)
(On-the-fly)
Students R
Enroll E



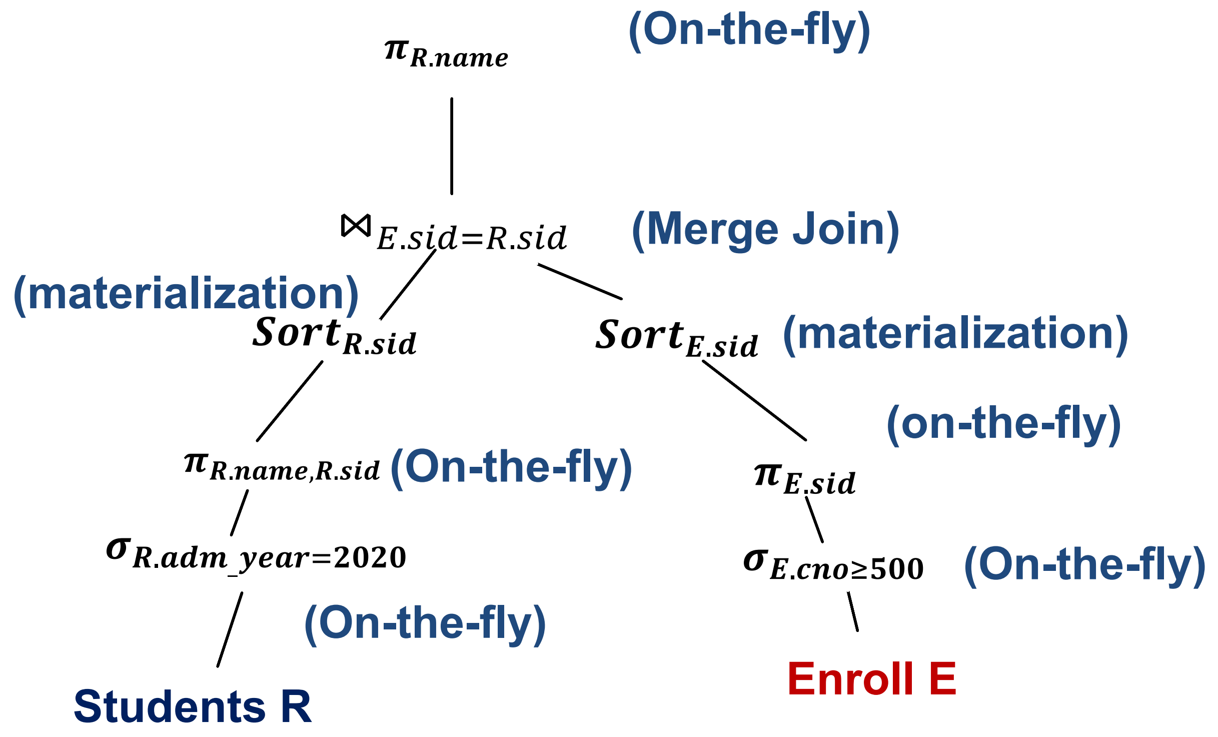
$\pi_{R.name}$ (On-the-fly)
$\bowtie_{E.sid=R.sid}$ (Merge Join)
(materialization) $Sort_{R.sid}$
$Sort_{E.sid}$ (materialization)
(on-the-fly)
$\pi_{R.name,R.sid}$ (On-the-fly)
$\pi_{E.sid}$
$\sigma_{R.adm\_year=2020}$
$\sigma_{E.cno\geq 500}$ (On-the-fly)
(On-the-fly)
Students R
Enroll E

Cost = $1000 + [1000 * 0.5 * 0.2]$
$+500 + [500 * 0.1 * 0.48] * [1000 * 0.5 * 0.2]$
$=$ 4,000 I/Os

Cost = ?

Choice of join algorithms

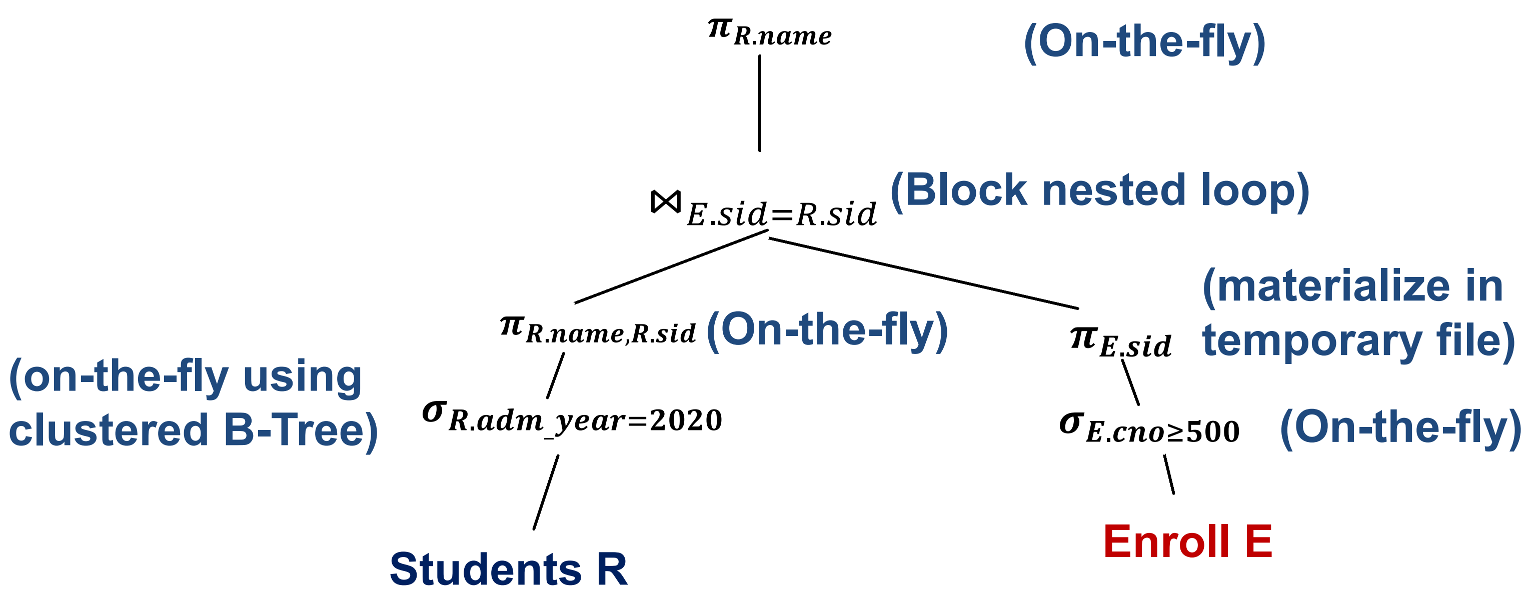
- Sort outer:
  - Size after pass 0: $\lceil 500 * 0.1 * 0.48 \rceil = 24$
    - 4 pages/run, 6 runs (need one input buffer for table scan)
  - # merge passes = $\lceil \log_4 6 \rceil = 2$
  - Total I/O: $500 + 24 + 2 \times 2 \times 24 = 620$
- Sort inner: # I/O = 1700
- Merge
  - assuming d = 5 and always fit in one page
  - 24 + 100 = 124
- Total cost = 620 + 1700 + 124 = 2,444 I/Os
  - vs BNL: 4,000 I/Os



Cost = ?

Using indexes

- If we have a clustered B-Tree index over $R(adm\_yaer)$, h = 3

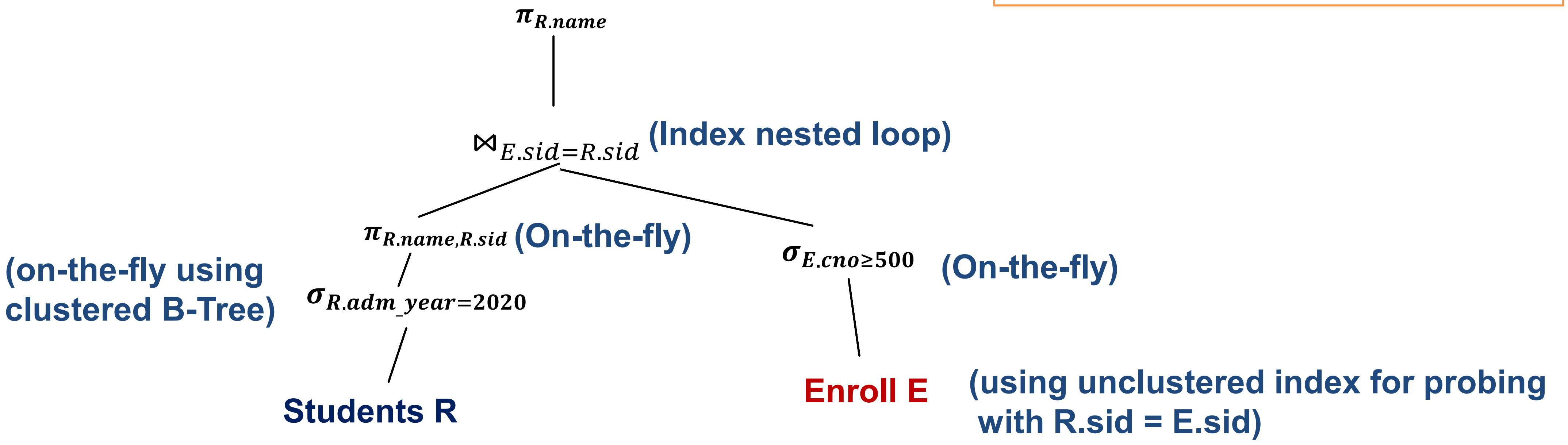


Cost = $1000 + [1000 * 0.5 * 0.2]$
$+3 + [500 * 0.1 * 0.48]$
$+[500 * 0.1 * 0.48] * [1000 * 0.5 \ * 0.2]$
$=$ 3,527 I/Os

Using indexes

- If we have an unclustered B-Tree index over $E(sid)$, h = 3
  - *Generally, index nested loop is a bad choice unless both of the following is true*
    - *outer plan output size is small*
    - *join is very selective*

assuming each student has 5 enrollment record on average



Cost = $3 + [500 * 0.1 * 0.48] + [40000 * 0.1] * (3 + 5)$
$= 32{,}027$ I/Os (vs 3,527 I/Os with BNL!)

Cost-Based Query optimization

- Query can be dramatically improved by changing access methods, order of operators.
  - Volcano interface
- Relational Algebra Equivalences provide theoretical foundation for valid transformations
  - *Naïve approach:*

```
EQ={E}
repeat
  foreach plan E in EQ
    apply each equivalence rule to some subexpression of E to obtain E′
    if E′ is not in EQ
      add E′ to EQ
until no new plan can be added to EQ
```

  - The search space is intractable even with a subset of the available rules
    - Consider a constrained space for $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$ where we only apply join reordering
      - there are $\frac{(2(n-1))!}{(n-1)!}$ possible join orders $(n! = 1 \times 2 \times \cdots \times n)$
      - n = 3 => 12 join orders; n = 4 => 120 join orders; n = 5 => 1680 join orders, …

Cost-Based Query optimization

- Other issues with query optimization
  - Cost estimation
  - Choosing access path
  - Optimizing multi-relation queries (aka join queries)
- Case study on the *"System-R"-style query optimizer*
  - The first and the most classic RDBMS (from IBM)
  - Its optimizer has huge influence on query optimizers in today's DBMS

Highlights of System R Optimizer

- Impact:
  - Most widely used currently; works well for < 10 joins.
- Cost estimation:
  - Very inexact, but works fine in practice.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
  - *State-of-the-art has a lot of improvement on this topic.*
- Plan Space: Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
  - Cartesian products avoided.

Query blocks: unit of optimization

- An SQL query is parsed into a collection of *query blocks*
  - and these are optimized one block at a time.
- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (This is an over-simplification but serves for the purpose now.)

```
SELECT  S.name
FROM  student S
WHERE  EXISTS
    (SELECT  E.sid
     FROM  enroll E
     WHERE E.sid = S.sid
        AND E.grade >= 3.7)
```

Outer block

Inner (Nested) block

- For each block, the plans considered are:
  - All available access methods, for each relation in FROM clause (potentially with projection/selection push-downs).
  - All *left-deep join trees* (i.e., right branch always a base table, consider all join orders and join methods.)

Translating SQL to logical plan

- Inner block:
  - $Q(sid) \leftarrow \sigma_{E.sid=sid \wedge E.grade \geq 3.7} E$
- Outer block:
  - $S \leftarrow student$
  - ~~$\pi_{S.name} \sigma_{S.sid\ IN\ Q(sid\ )} S$~~ (?)
  - Can't express it in standard relational algebra!
- Need to extend the relational algebra
  - Choice 1: correlated evaluation (by extending the algebra)
    - Treat the inner block as a function, evaluate it for every outer tuple scanned
    - IN operator is implemented as a set membership testing
  - Choice 2: decorrelation (by introducing semi-joins and anti-joins)
    - semi-join: $A \ltimes_\theta B \triangleq \pi_{A(R)}(A \bowtie_\theta B)$ (set version)
      - for multi-set version, multiplicity of the same tuple in the result is the same as that in A
    - anti-join: $A \triangleright_\theta B \triangleq A - A \ltimes_\theta B$
    - Example: $\pi_{name} S \ltimes_{S.sid=E.sid} \sigma_{E.grade \geq 3.7} E$
      - Can we replace $\ltimes$ with $\bowtie$? (No!)

```
SELECT  S.name
FROM  student S
WHERE  EXISTS
   (SELECT  *
    FROM  enroll E
    WHERE E.sid = S.sid
       AND E.grade >= 3.7)
```

Cost estimation

- To compare different plans, we need to estimate the cost
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must estimate *size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
  - In System R, cost is boiled down to a single number: #I/O + *c* * #CPU instructions
    - Can refine #I/O as the #page_transferred * t_T + #seeks * t_S
    - c, t_T and t_S are all configurable parameters – must be set correctly
  - Q: Is "cost" the same as estimated "run time"?

Statistics and catalog

- Need information about the relations and indexes involved. *Catalogs* typically contain at least:
  - # tuples (**NTuples**) and # pages (**NPages**) per relation.
  - # distinct key values (**NKeys**) for each index.
  - low/high key values (**Low/High**) for each index.
  - Index height (**IHeight**) for each tree index.
  - # index pages (**INPages**) for each index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

Size estimation and selectivity

```
SELECT  attribute list
FROM  relation list
WHERE  term1 AND … AND termk
```

- Consider a query block:
- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.
- *Selectivity* associated with each *term* reflects the impact of the *term* in reducing result size. *Result cardinality* = Max # tuples * product of all selectivities.
  - also called *reduction factor (RF)*

Result size estimation

- *Result cardinality* = Max # tuples * product of all RF's.
  (Implicit assumption that
  values are uniformly distributed and *terms* are mutually independent!)
- Term *col=value*
  RF = *1/NKeys(col)*
- Term *col1=col2* (This is handy for joins too…)
  RF = *1/MAX(NKeys(col1), NKeys(col2))*
- Term *col>value, where col is in the range of* (L, H)
  RF = *(H – max(L,min(value,H)))/(H-L)*

Estimation errors

- Estimation errors lead to poor optimizer decisions
    - Example: suppose there's an employee table with *level* and *salary* columns
- Data skewness is one of the main reasons that may lead to bad estimations
    - Example: with the same employee table, let's say $level \in (0,10]$
        - selectivity of $level > 6$?
            - estimation: $\frac{10-6}{10-0} = 40\%$
            - common sense tells us this is significantly lower than 40% (e.g., 20%...)
                - Solution

- The assumption of *mutual independence* of the predicates may not hold! In other words,
    - We assume $\Pr(term_1 = true \wedge term_2 = true \wedge \cdots \wedge term_k = true) = \Pi_{1 \le i \le k} \Pr(term_i = true)$
    - Does not always hold in practice => estimation errors!
            - selectivity of $\text{level} > 6$: 20%
            - selectivity of $salary \geq \$400{,}000$: 30%
            - selectivity of $\text{level} \geq 6 \wedge salary \geq \$400{,}000$?
                - Unlikely to be $20\% \times 30\% = 6\%$! (common sense tells us this is likely to be 20% ...)
- Solution: histograms

Reduction Factors & Histograms

- We build histograms in the database catalog to provide better size estimation for common predicates over one or more columns
  - equi-width: *equal key ranges, store both key ranges and values*
  - equi-depth: equal number of items, *only store the quantiles and the total number of items!*

  - Example: 0.0, 0.2, 0.8, 1.3, 1.5, 1.6, 1.8, 1.85, 2.0, 2.1, 2.2, 3.0
    - 4 buckets

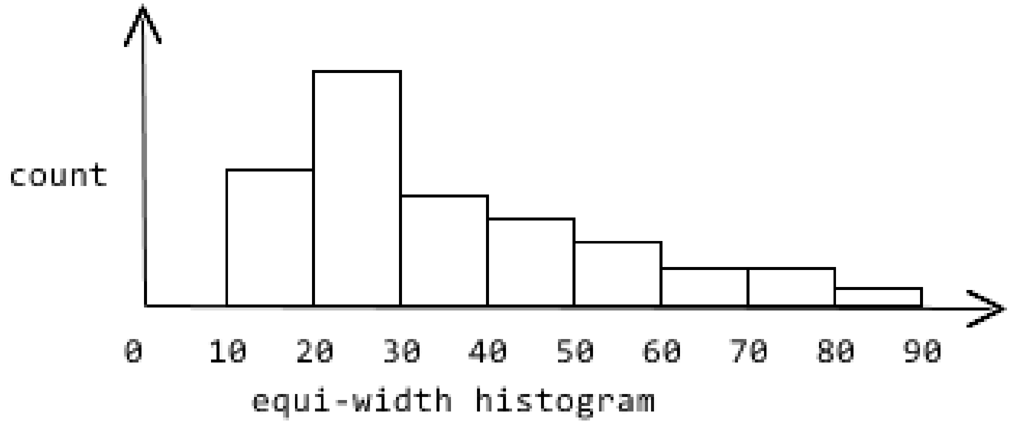
equi-width

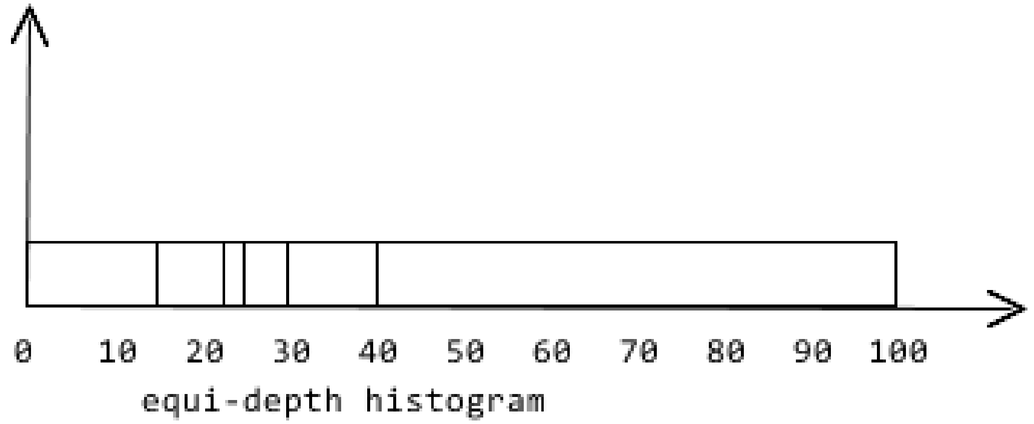
Range	[0.0, 0.75]	(0.75,1.5]	(1.5,2.25]	(2.25,3.0]
Count	2	3	6	1

equi-depth

quantile	min	25%	50%	75%	100%
key value	0.0	0.8	1.6	2.0	3.0

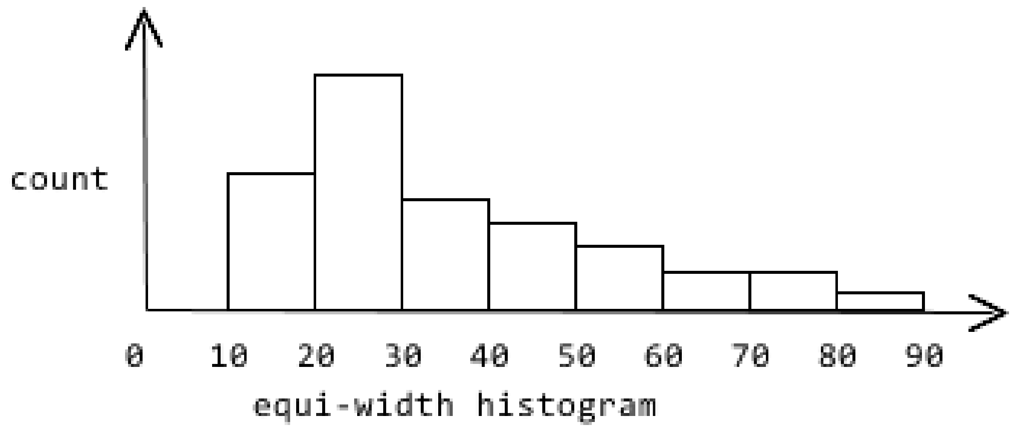
A visual comparison: equi-width vs equi-depth (1D)





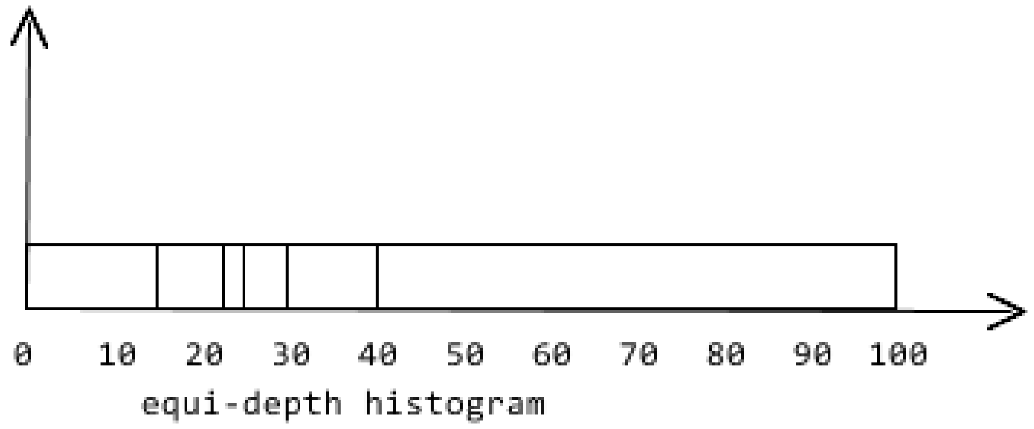
X-axis: domain of interest (e.g., age), Y-axis: frequency of values in a given range

Histogram Construction: equi-width



Equi-width: EASY
Given the number of buckets B and domain size, this is easily done with one linear pass over the data using an array of counters.
Note that B determines the size of a histogram.

Histogram Construction: equi-depth



Equi-depth: more time consuming
Sorting and finding the quantiles. Need to handle the corner cases with many duplicate keys.

Finding selectivity using histograms

- (Implicitly) map the histogram back to an approximate relation
  - then apply the predicate to the approximate relation
- Example: 1 <= x <= 1.75
  - actual selectivity = 3 / 12 = 25%

equi-width

Range	[0.0, 0.75]	(0.75,1.5]	(1.5,2.25]	(2.25,3.0]
Count	2	3	6	1

$$3 \times \frac{1.5-1}{0.75} + 6 \times \frac{1.75-1.5}{0.75} = 4$$

estimated selectivity = 4/12 = 33.3%

equi-depth

quantile	min	25%	50%	75%	100%
key value	0.0	0.8	1.6	2.0	3.0

estimated selectivity =

$$\left(\frac{1.75-1.6}{2.0-1.6} \times 0.25 + 0.5\right) - \left(\frac{1-0.8}{1.6-0.8} \times 0.25 + 0.25\right) \approx 28.1\%$$

- In general, equi-depth has better error guarantees over the estimation.
  - Does not mean it is always better than equi-width over a specific instance

Join size estimation

- Does the above make sense for joins?

- Q: Given a join of R and S, what is the range of possible result sizes (in #of tuples)?
  - If join is on a key for R (and a Foreign Key in S)?
    - $|R \bowtie S| = |S|$
    - A common case, can treat it specially
- General case: join on {A} ({A} is not key for either relation)
  - estimate each tuple r of R generates NTuples(S)/NKeys(A,S) result tuples, so…

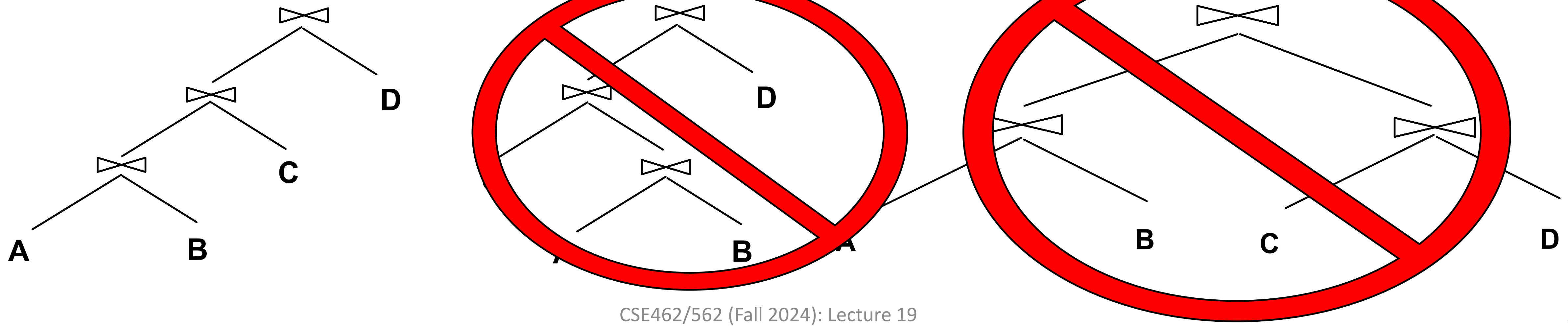
    NTuples(R) * NTuples(S)/NKeys(A,**S**)
  - but can also consider it starting with S, yielding: NTuples(S) * NTuples(R)/NKeys(A,**R**)
  - If these two estimates differ, take the lower one!
    - Q: Why?

Access path selection

- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans
- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:
  - Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
  - The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are *pipelined* into the aggregate computation).
- We've discussed the cost estimation for these operators in previous lectures.

Join reordering

- Fundamental decision in System R: *only left-deep join trees* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly
    - *we need to restrict the search space.*
    - If we only consider join orders
      - all possible join orders: $\frac{(2(n-1))!}{(n-1)!}$
      - left-deep joins: $n!$
    - In practice, plan space is slightly larger than $n!$
      - due to the need to consider alternative join algorithms, and/or interesting orders
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., sort-merge join).



Enumeration of left-deep plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method for each join.
  - *Approach: Dynamic programming*
- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation.
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. *(All 2-relation plans.)*
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. *(All N-relation plans.)*
- For each subset of relations, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each *interesting order* of the tuples.

A note on “Interesting Orders”

- An intermediate result has an “interesting order” if it is sorted by any of:
  - ORDER BY attributes
  - GROUP BY attributes
  - Join attributes of yet-to-be-planned joins

Enumeration of left-deep plans (contd.)

- An N-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE have been used up.
  - i.e., avoid Cartesian products if possible.
- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an `interestingly ordered' plan or an additional sort/hash operator.
- In spite of the pruned search space, this approach is still exponential in the # of tables.

Example of enumeration of left-deep plans

```
SELECT S.sid, COUNT(*)
FROM student S, enroll E, course C
WHERE S.sid = E.sid
       AND E.dname = C.dname AND E.cno = C.cno
       AND c.category = 'elective'
```

<u>Student:</u>
 Hash and clustered B-Tree on *sid*
<u>Enroll:</u>
 Clustered B-tree on *(dname, cno)*
 Unclustered B-tree on sid
<u>Course:</u>
 Clustered B-Tree on category

- For simplicity, let's first ignore interesting orders.
- Pass1: Best plan(s) for accessing each relation
  - Student, Enroll: Heap Scan
  - Course: B-tree on category

Pass 2 in the enumeration of left-deep plans

- For each of the plans in pass 1, generate plans joining another relation as the inner, using all join methods (and considering all possible inner access methods as well)
  - To find the best plan for $S \bowtie_{S.sid=E.sid} E$
    - If heap scan over S is outer,
      - Block nested loop with heap scan over E
      - Index nested loop with unclustered B-tree over E(sid)
      - Sort-merge join with E
        - add sorting over S, and consider sorting E or index scan with B-tree over S(sid)
      - Hash join with heap scan over E
    - If heap scan over E is outer,
      - Block nested loop with heap scan over S
      - Index nested loop with hash index over S(sid)
      - Index nested loop with B-tree index over S(sid)
      - Sort-merge join with E
        - add sorting over E, and consider sorting S or index scan with B-tree over S(sid)
      - Hash join with heap scan over E
  - To find the best plan for $E \bowtie_{E.dname=C.dname \wedge E.cno=C.cno} C$
  - What about $S \bowtie C$? No predicate available now => don't consider now
- Retain cheapest plan for each pair of relations, example:
  - $S \bowtie_{S.sid=E.sid} E$: Sort-merge join between heap scan of S and heap scan of E
  - $E \bowtie_{E.dname=C.dname \wedge E.cno=C.cno} C$: Index nested loop between (index scan of C over category) and (clustered B-Tree over $E(sid)$)

Pass 3 in the enumeration of left-deep plans

- For any triples of relations, pick one as the inner and use the best plan from pass two for the join of the other two relations to find the best possible plan
  - For $S \bowtie_{S.sid=E.sid} E \bowtie_{E.dname=C.dname \wedge E.cno=C.cno} C$, consider
    - $(S \bowtie E) \bowtie C$
      - BNL between (Sort-merge join between heap scan of S and heap scan of E) and heap scan over C
      - SMJ between (Sort-merge join between heap scan of S and heap scan of E) and heap scan over C
      - HJ between (Sort-merge join between heap scan of S and heap scan of E) and heap scan over C
      - INL between (Sort-merge join between heap scan of S and heap scan of E) and heap scan over C
    - $(E \bowtie C) \bowtie S$
      - BNL between (INL between E and C) and heap scan over S
      - SMJ between (INL between E and C) and heap scan over S
      - HJ between (INL between E and C) and heap scan over S
      - INL between (INL between E and C) and hash or B-tree over S(sid)

  - Retain the one with the best cost
    - add any unapplied predicate/aggregation/deduplication/projection/sorting on top
      - this is the final plan as the optimization result