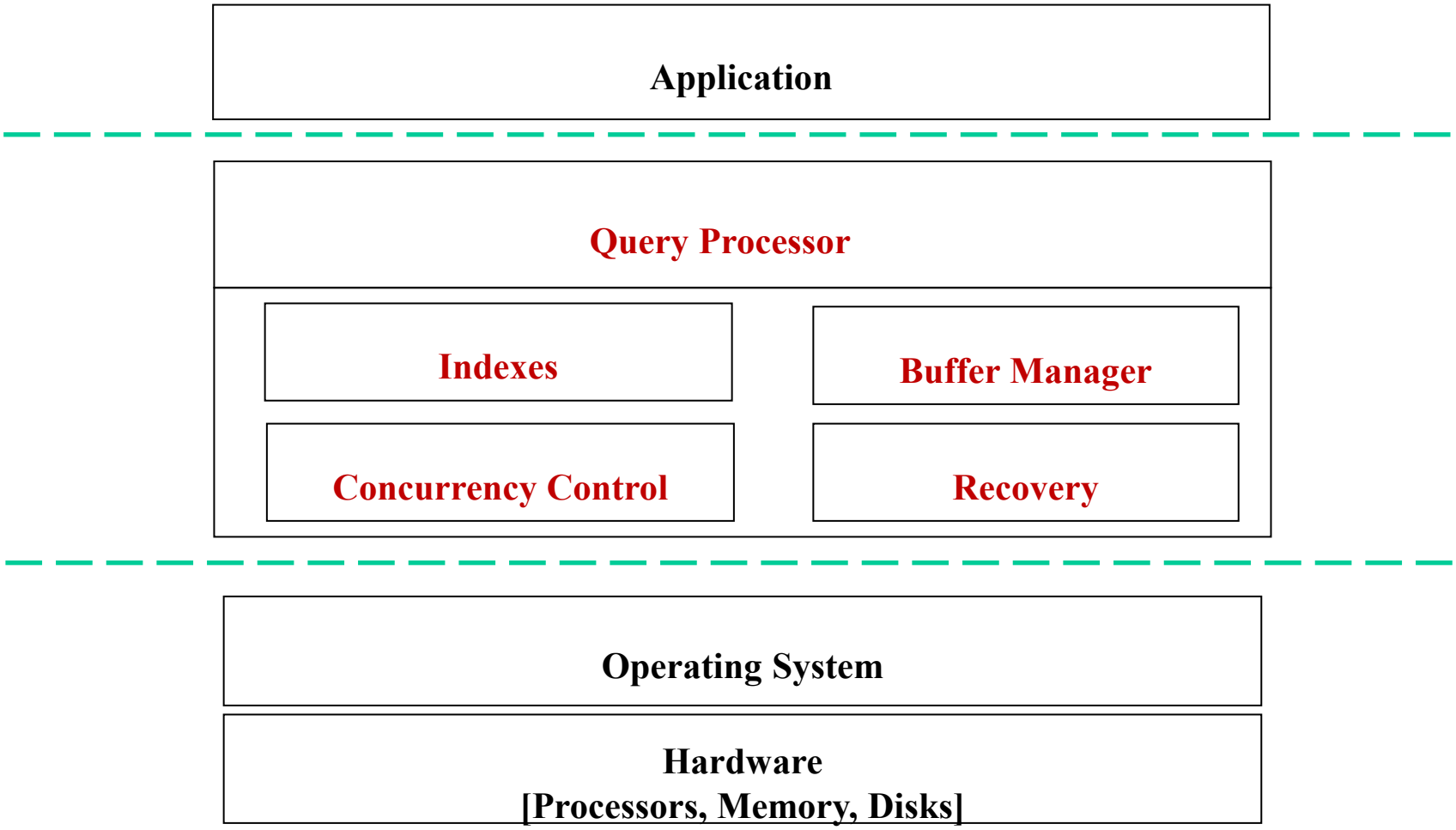
QUERY OPTIMIZATION

E0 261

Jayant Haritsa
Computer Science and Automation
Indian Institute of Science

Typical RDBMS Engine



Design of RDBMS Engines

- Transaction Processing (ACID)
  - WAL/ARIES for Atomicity/Recovery
  - 2PL for Concurrency Control
- Data Access Methods
  - B-trees/Hashing for Large Ordered Domains
  - Bitmaps for Small Categorical Domains
  - R-trees for Geometric Domains
- Memory Management
  - LRU-k (k=2 balances history and responsiveness)
- Query Processing (SQL)
  - "Black Art"

Declarative Queries

- SQL, the standard database query interface, is a declarative language
  - Specifies only what is wanted, but not how the query should be evaluated (i.e. ends, not means)
  - Example: List the names of students with their registered courses

```
select   StudentName, CourseName
from     STUDENT,  COURSE,  REGISTER
where    STUDENT.RollNo = REGISTER.RollNo   and
         REGISTER.CourseNo = COURSE.CourseNo
```

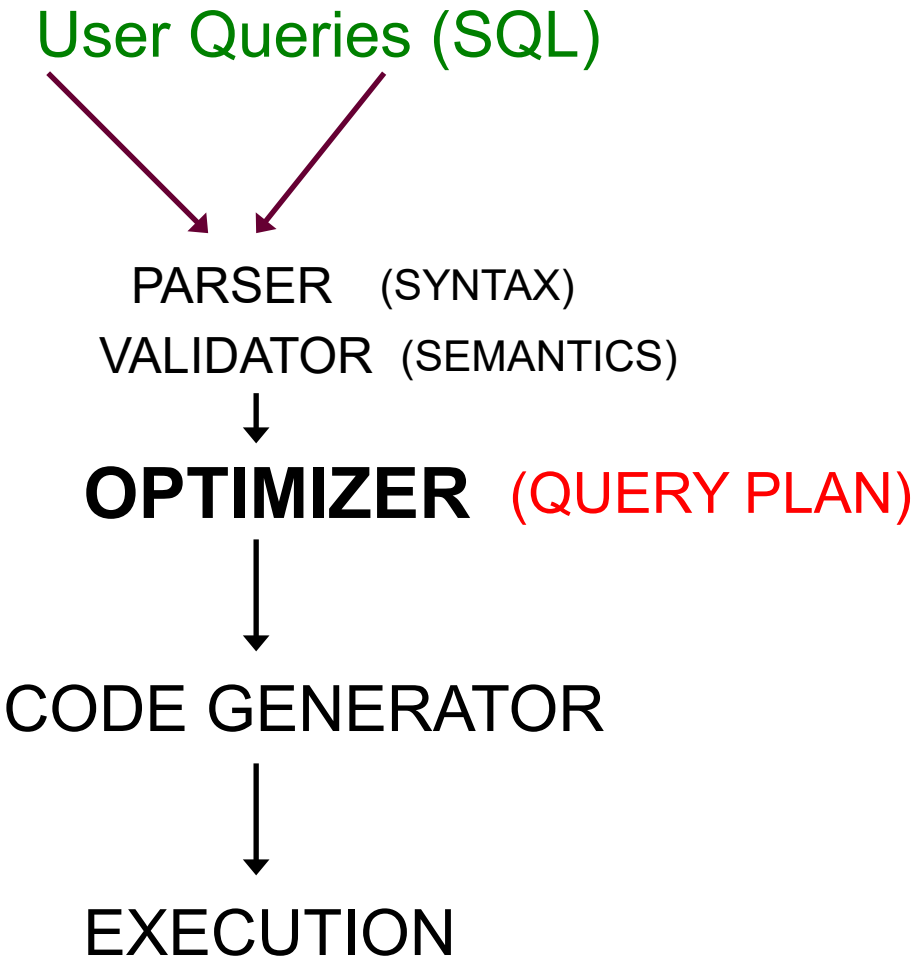
Unspecified:

join order [((S ⋈ R) ⋈ C) **or** ((R ⋈ C) ⋈ S) ?]

join techniques [Nested-Loops **or** Sort-Merge **or** Hash ?]

- DBMS query optimizer identifies efficient execution strategy:
  "query execution plan"

Query Processing Architecture



Overview of Query Optimization

- Algebra Tree: Tree of relational algebra operators (σ, π, □, …) that implement the user's query
- Query Plan: Tree of relational algebra operators, with choice of algorithm for each operator
- Design Goal: Find best plan

Motivating Example

Student (*sid*: integer, *sname*: string, *age*: integer, *gpa*: real)

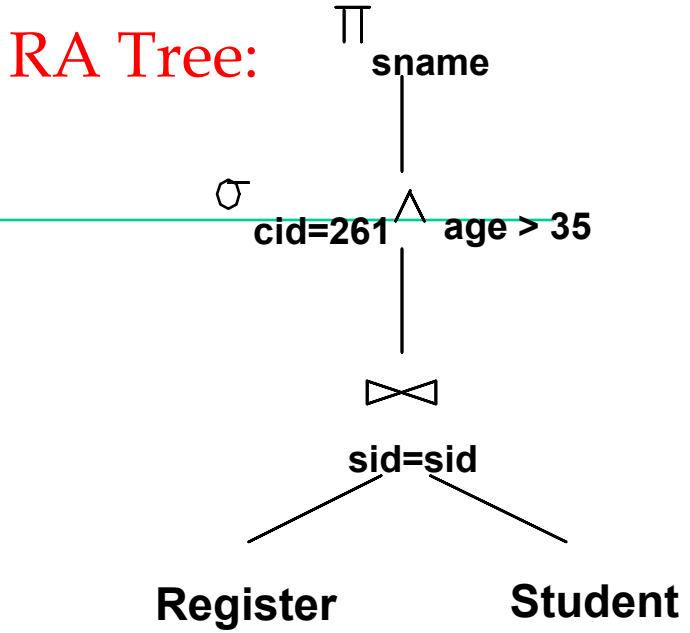
Register (*sid*: integer, *cid*: integer, *section*: integer, desc: string)

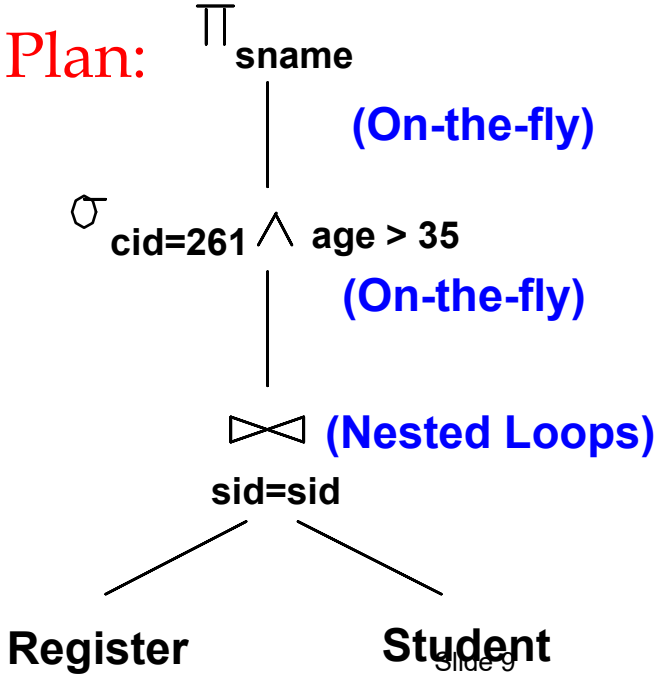
```
SELECT  S.sname
FROM  Student S,  Register R
WHERE  S.sid = R.sid AND
    R.cid = 261 AND S.age > 35
```

(Equivalent English Query: Senior registrants for DBMS course)

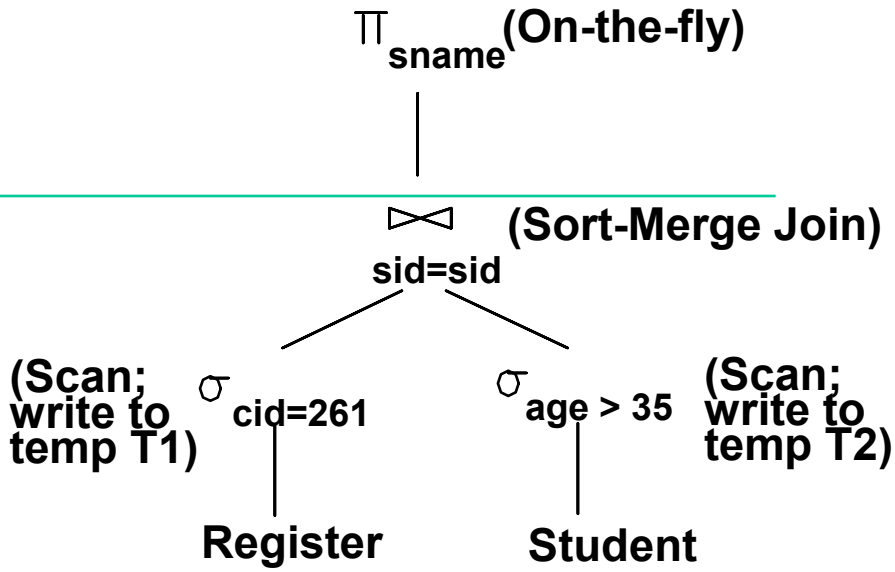
Example (contd)

- Student:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages. (40000 students)
- Register:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages. (100000 registrations)
- Cost (5 buffers)
  - 500+167*1000 = 167500 disk accesses
- Misses several opportunities
  - selections could be "pushed" earlier
  - no use is made of indexes, etc.
- Goal of optimization: To find efficient plans that compute the same answer.





Alternative Plan



- **Main difference:** push selects.
- With 5 buffers, cost of plan:
  - Scan Register (1000) + write temp T1 (10 pages, if we have 100 different courses, uniform distribution).
  - Scan Student (500) + write temp T2 (250 pages, if we have 10 different ages in range 31-40, uniform distribution).
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge-join (10+250)
  - Total: 4060 page I/Os.
- If we used BNL join of T1 and T2, join cost = 10 + 4*250, total cost = 2770.
- If we "push" projections, T1 has only sid, T2 only sid and sname:
  - T1 fits in 3 pages, cost of BNL drops to under 250 pages, total < 2000.

Design Framework

- Issues
  - For a given query, what plans are considered (i.e. plan space)?
  - How is the cost of a plan estimated (i.e. cost model, statistics)?
  - How to efficiently search through plan space for “cheapest” plan?

Design Framework (contd)

- Ideally: Want to find best plan
  Practically: Avoid worst plans!

- System R approach
  - 1979 ACM Sigmod conference − paper considered "Bible" of query optimization
  - Most widely used currently; works well for < 10 joins.

P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price,
``Access Path Selection in a Relational Database System",
Proc. of ACM SIGMOD Intl. Conf. on Management of Data, June 1979.

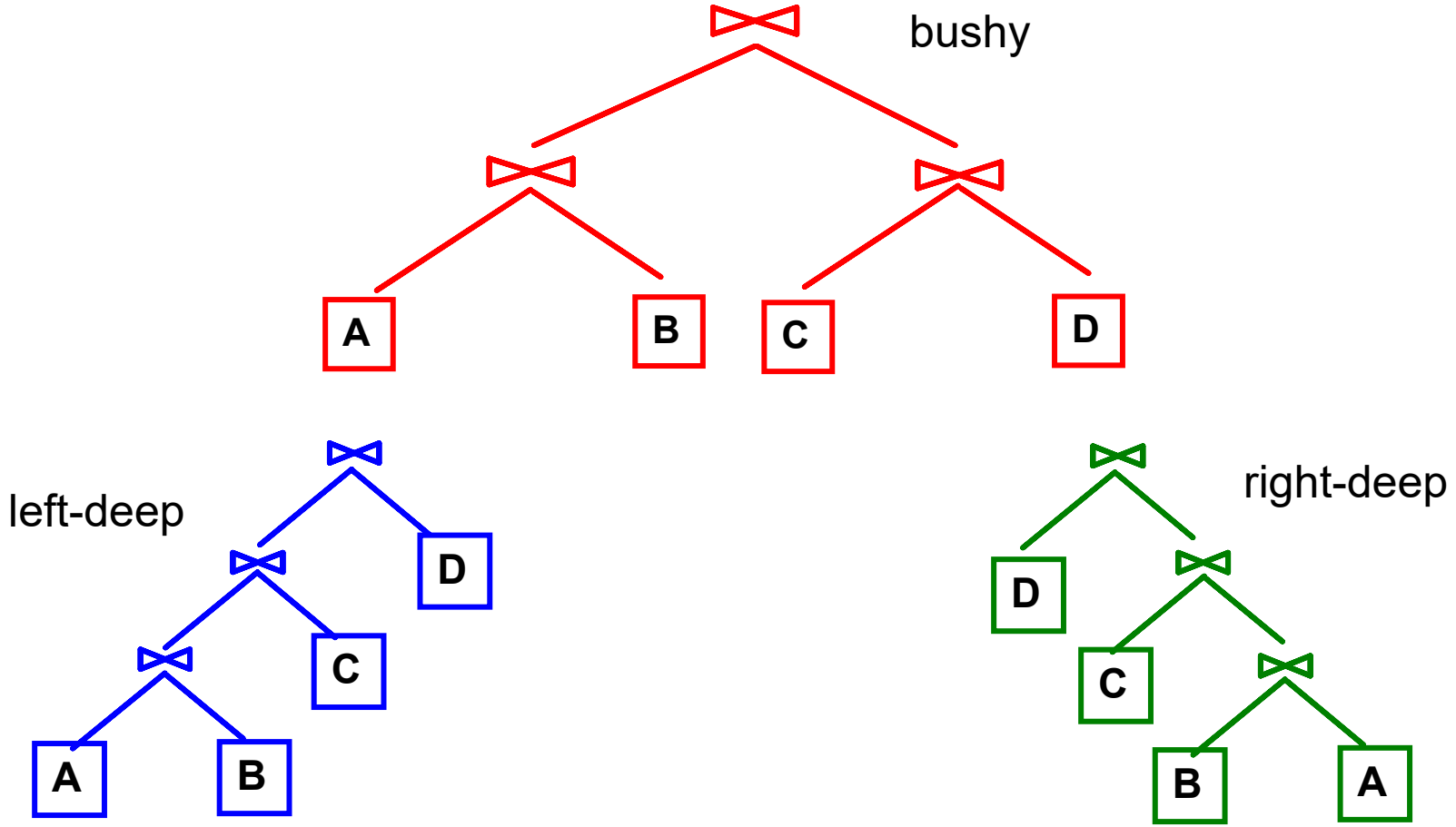
Overview of System R Optimizer

Issue 1: Plan Space

- Operator Algorithms:
  - Access: Sequential, Index (clustered / unclustered)
  - Join: NestedLoop, Sort-merge

- Plan Structure:
  - Only left-deep plans are considered
    - Left-deep plans allow output of each operator to be pipelined into the next operator without storing it in a temporary relation.
  - Cartesian products are avoided

Join Orders



Issue 2: Cost Model

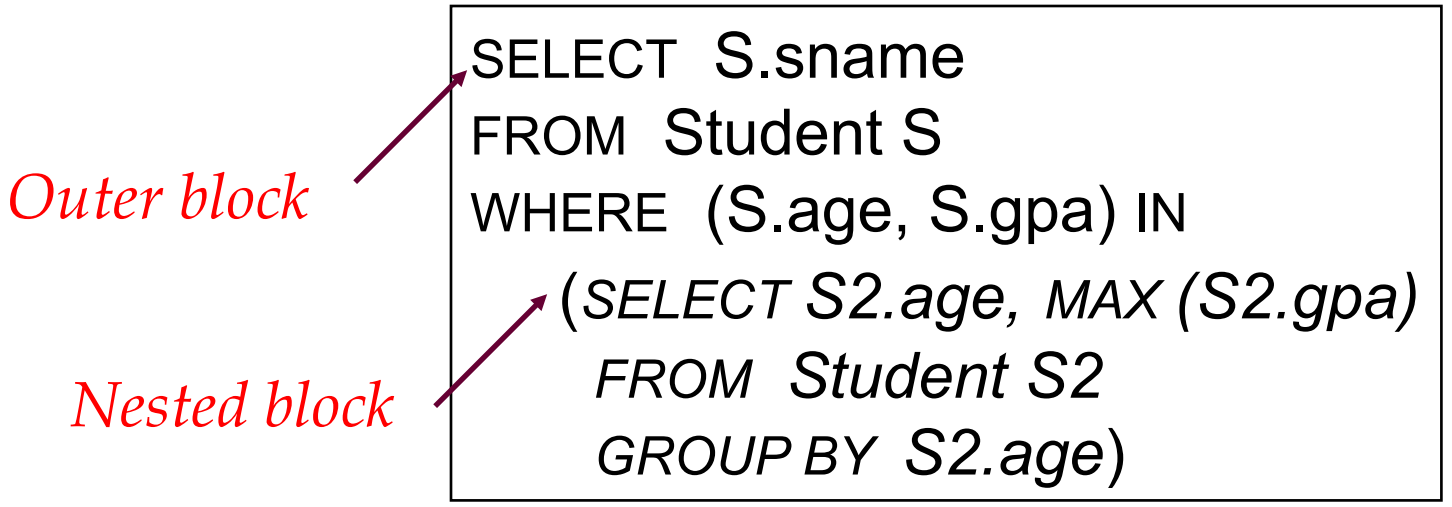
- Weighted combination of CPU and I/O costs
- Simple statistics of relations and indexes maintained in system catalogs
- Assume independence across predicates and uniform data distribution across domain
- “Wet finger” technique when stats not available!
- Statistics are only periodically updated
  - ensures that metadata locking does not become a bottleneck!

Issue 3: Search Algorithm

- Given an n-relation join, there are *n!* plans possible (not factoring in choice of join method)
- Plan prefixes are "memoryless" about order, therefore use Dynamic Programming (DP)
- DP works for $n \leq 10$

Details of System R Optimizer

Query Blocks: Units 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.

Estimations

- For each plan considered:
  - Must estimate cost of each operation in plan tree.
  - Must estimate size of result (cardinality) for each operation in tree.

Cost Model and Statistics

- Cost = I/O + w * CPU
  = Page Fetches +
  w * # of tuples returned from RSS
- Catalogs contain:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and # pages (NPages) for each index.
  - Index height (H(I)), low/high key values (Low/High) for each tree index.

Cardinality Estimation

- Consider a query block:

SELECT attribute list
FROM relation list
WHERE $term_1$ AND ... AND $term_k$

- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.
- Reduction factor (RF) associated with each term reflects the impact of the term in reducing result size.
- Result cardinality = Max # tuples * product of all RF's.
  - Implicit assumption that terms are independent !
  - Term col=value has RF 1/NKeys(I), given index I on col, o.w. 1/10
  - Term col1=col2 has RF 1/MAX(NKeys(I1), NKeys(I2)), o.w. 1/10
  - Term col>value has RF (High(I)-value)/(High(I)-Low(I)) o.w. 1/3
  - Term min < col < max has RF (max - min)/(High(I)-Low(I)) o.w. 1/4

Costing Query Blocks

- Various cases:
  - Single-relation query block
  - Multiple-relation query block
  - Nested-query blocks

Single-relation Block

- 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).

Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
  - Cost is Height(I)+1 for a B^+ tree
- Clustered index I matching one or more selects:
  - (NPages(I) + NPages(R)) * product of RF's of matching selects .
- Non-clustered index I matching one or more selects:
  - (NPages(I) + NTuples(R)) * product of RF's of matching selects .
- Sequential scan of file:
  - NPages(R) .

Note: Typically, no duplicate elimination on projections! (Exception: Done on answers if user says DISTINCT.)

Multi-relation Query Block

- Join of Relations R_1, R_2, R_3, ..., R_n .
- Equivalently, $r_1 \bowtie r_2 \bowtie r_3 \ldots \bowtie r_n$
- Decide mapping of r_i to R_j
- Decide j_i (join method)
- Decide $a(r_i)$ (access method for R_j)

Search Complexity

- Consider finding the best join-tree for $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$.
- There are $(2(n-1))! / (n-1)!$ different join-trees:
  - $n = 5$, number is 1680; $n = 7$, number is 665280;
    $n = 10$, the number is greater than 176 billion!
- No need to generate all join-trees. Using DP, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.
- This reduces time complexity to around $O(3^n)$
  - $n = 10$, number is 59000.
- With restriction to left-deep plans, the time complexity reduces to $O(n2^n)$.

Derivation of Search Complexity

- Number of binary trees with n leaf nodes is $\frac{1}{n+1}\left({}^{2n}C_n\right)$, known as the Catalan number
  - standard text-book derivation (e.g. Horowitz & Sahni)
- But, each join order has to be a complete binary tree.
- Number of complete binary trees given n leaf nodes is $\frac{1}{n}\left({}^{2(n-1)}C_{(n-1)}\right)$ (Horowitz & Sahni).
- Since there are n! permutations of the leaves, the total comes to what is given in the previous slide: $\frac{2(n-1)!}{(n-1)!}$

Plan Space Search

- Once first k relations are joined, method to join the composite to $(k+1)^{st}$ relation is independent of the order of joining first k , that is, applicable predicates, join orderings, etc. are all the same

$\Rightarrow$ Markovian (future depends only on present, not past, i.e. “memoryless”)

$\Rightarrow$ Dynamic Programming

(global optimal requires local optimal)

Example Search Space Reduction

- Consider $r_1 \bowtie r_2 \bowtie r_3$. There are 12 different join orders:

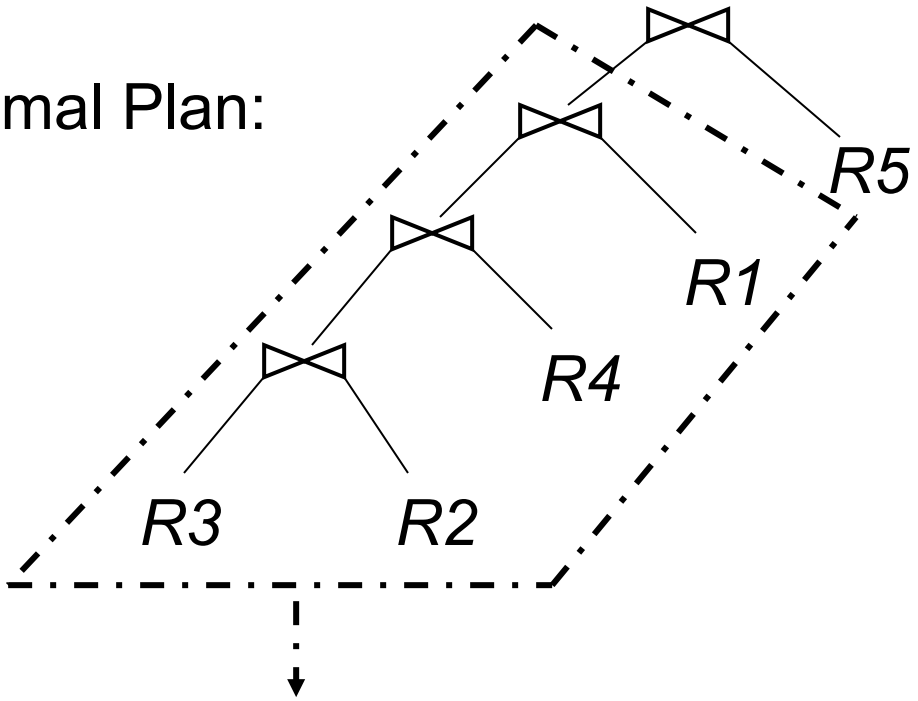
$r_1 \bowtie (r_2 \bowtie r_3)$ $r_1 \bowtie (r_3 \bowtie r_2)$ $(r_2 \bowtie r_3) \bowtie r_1$ $(r_3 \bowtie r_2) \bowtie r_1$
$r_2 \bowtie (r_1 \bowtie r_3)$ $r_2 \bowtie (r_3 \bowtie r_1)$ $(r_1 \bowtie r_3) \bowtie r_2$ $(r_3 \bowtie r_1) \bowtie r_2$
$r_3 \bowtie (r_1 \bowtie r_2)$ $r_3 \bowtie (r_2 \bowtie r_1)$ $(r_1 \bowtie r_2) \bowtie r_3$ $(r_2 \bowtie r_1) \bowtie r_3$

- To find best join order for $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$, there are 12 different join orders for computing $r_1 \bowtie r_2 \bowtie r_3$, and 12 orders for computing the join of this result with r_4 and r_5. Thus, there appear to be 144 join orders to examine. However, once we have found the best join order for the subset of relations $\{r_1, r_2, r_3\}$, we can use only that order for further joins with r_4 and r_5. Thus, instead of 144 choices to examine, we need to examine only 12 + 12 = 24 choices.

Principle of Optimality†

Query: *R1* ⋈ *R2* ⋈ *R3* ⋈ *R4* ⋈ *R5*

Optimal Plan:



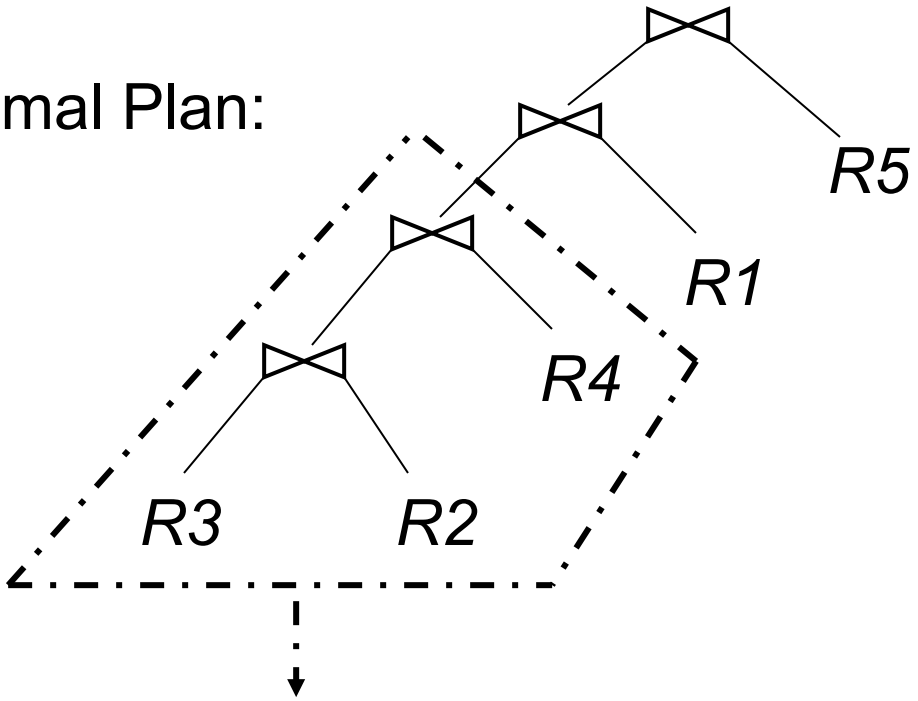
Optimal plan for joining *R3, R2, R4, R1*

† Optimality slides from Shivnath Babu

Principle of Optimality

Query: *R1* ⋈ *R2* ⋈ *R3* ⋈ *R4* ⋈ *R5*

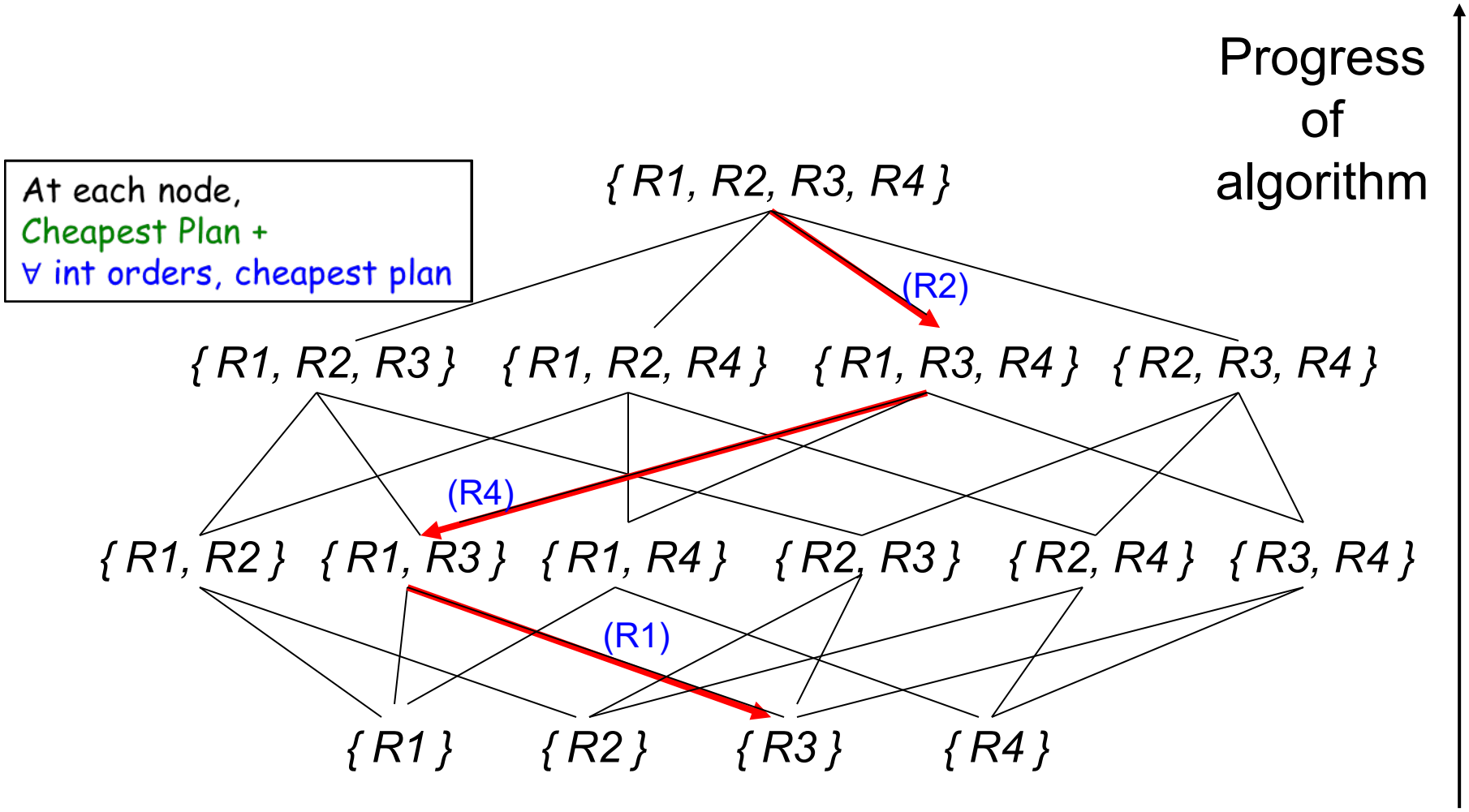
Optimal Plan:



Optimal plan for joining *R3, R2, R4*

Selinger Algorithm:

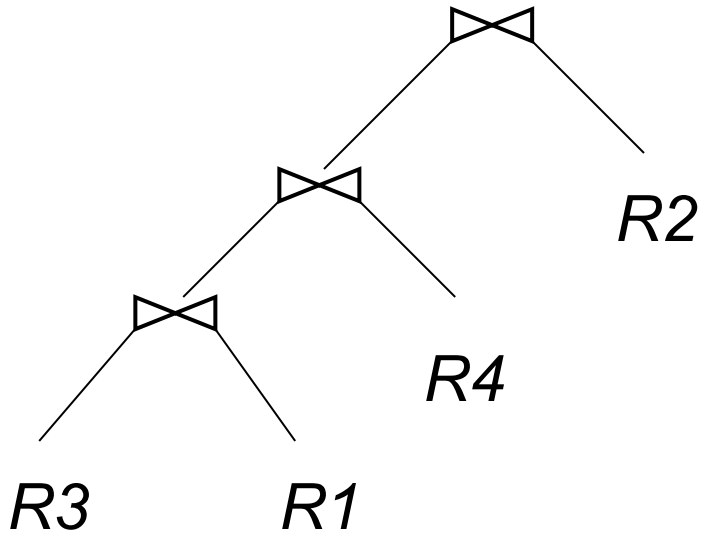
Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Selinger Algorithm:

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

Optimal plan:



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.
- 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 (GROUP BY, ORDER BY, or join column) of the tuples.

explicit

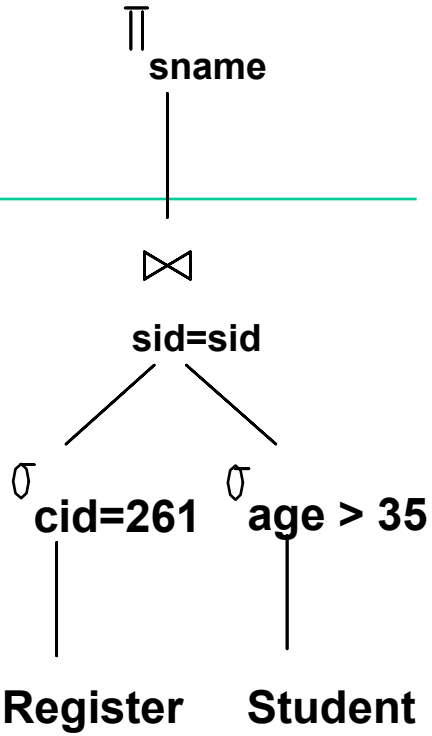
implicit

Enumeration of Plans (Contd.)

- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an "interestingly ordered" plan or an additional sorting operator.
- An N–1 way plan is not combined with an additional relation unless there is a join condition between them or all predicates in WHERE clause have been used up.
  - i.e., avoid Cartesian products if possible.
- In spite of pruning plan space, this approach is still exponential in the # of tables (hence the limitation to 10 joins)

Example

Student:
B+ tree on *age*
B+ tree on *sid*
Register:
B+ tree on *cid*



- Pass1:
  - Student: B+ tree matches age>35, and is probably cheapest. However, if this selection is expected to retrieve a lot of tuples, and index is unclustered, file scan may be cheaper.
    - Still, B+ tree plan kept (because tuples are in age order).
  - Register: B+ tree on cid matches cid=261; cheapest.

- Pass 2:
  - We consider each plan retained from Pass 1 as the outer, and consider how to join it with the (only) other relation.
    - e.g., *Register as outer*: B^+ tree index can be used to get Student tuples that satisfy sid = outer tuple's sid value.

Detailed Example

- Work through Figures 3, 4, 5, 6 in the paper

Nested Query Blocks

- Uncorrelated nested queries are basically constants to be computed once during execution
- Correlated nested queries are like function calls.

Example

```
SELECT  S.sname
FROM  Student S
WHERE  S.salary =
   (SELECT  avg(salary)
    FROM  Student)
```

← Independent

```
SELECT  S.sname
FROM  Student S
WHERE  S.salary >
   (SELECT  A.salary
    FROM  Advisor A
    WHERE A.id=S.advisor)
```

← Correlated

Handling

```
SELECT  S.sname
FROM  Student S
WHERE EXISTS
   (SELECT  *
    FROM  Register R
    WHERE  R.cid=261
    AND  R.sid=S.sid)
```

- Nested block is optimized independently, with the outer tuple considered as providing a selection condition.
- Outer block is optimized with the cost of "calling" nested block computation taken into account.
- Implicit ordering of these blocks means that some good strategies are not considered. The non-nested version of the query is typically optimized better.

Nested block to optimize:

```
SELECT  *
FROM  Register R
WHERE  R.cid=261
   AND  R.sid=outer value
```

Equivalent non-nested query:

```
SELECT  S.sname
FROM Student S, Register R
WHERE  S.sid=R.sid AND R.cid=261
```

Limitations

- Only considers left-deep plans
- Statistics make major assumptions of uniformity and independence (will address these issues in later papers on histograms and sampling)
- Nested queries are handled in a straightforward manner without un-nesting
- Instead of dynamic programming, could also consider alternatives such as randomized algorithms based on simulated annealing

END QUERY PROCESSING

E0 261