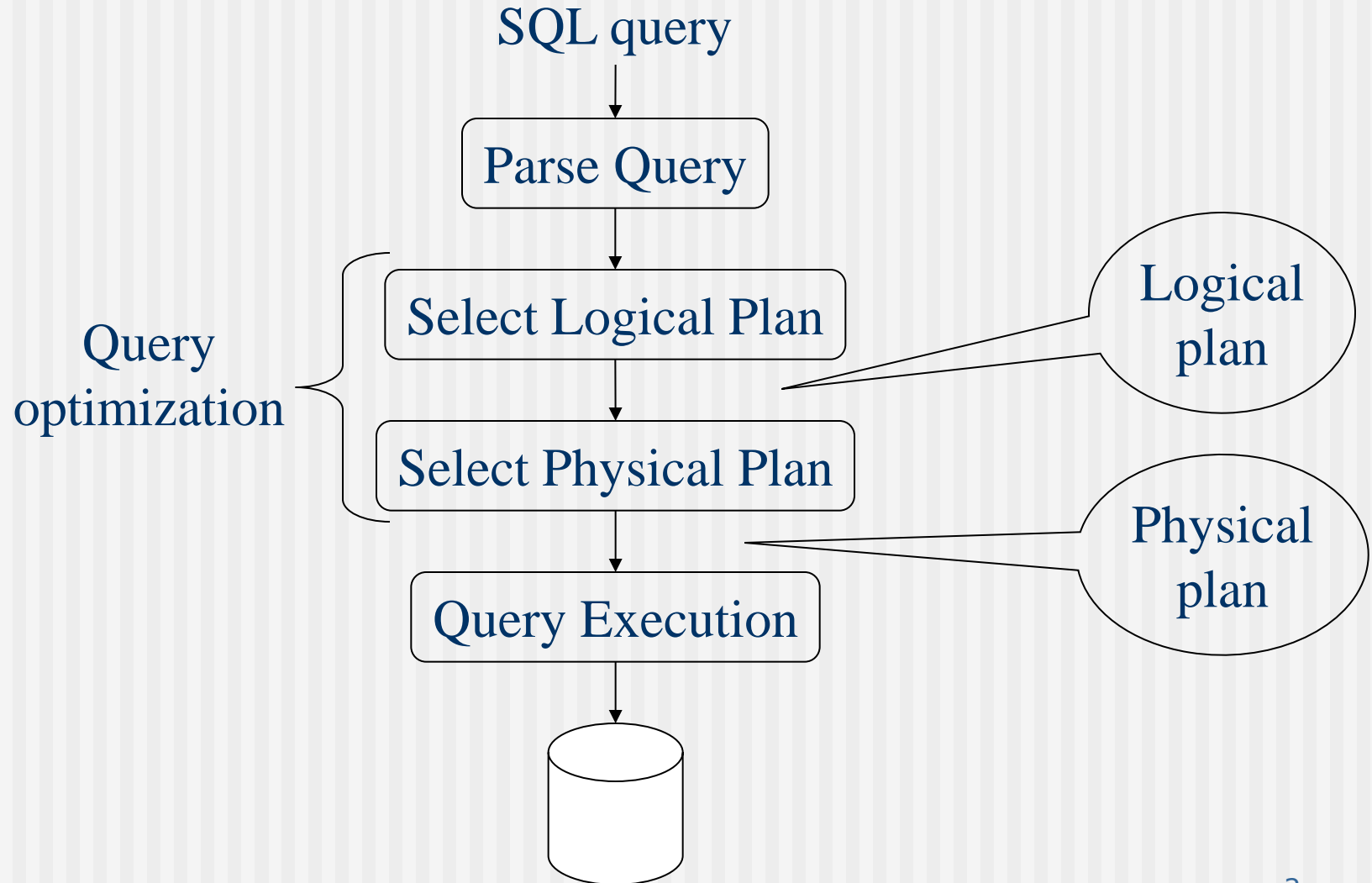
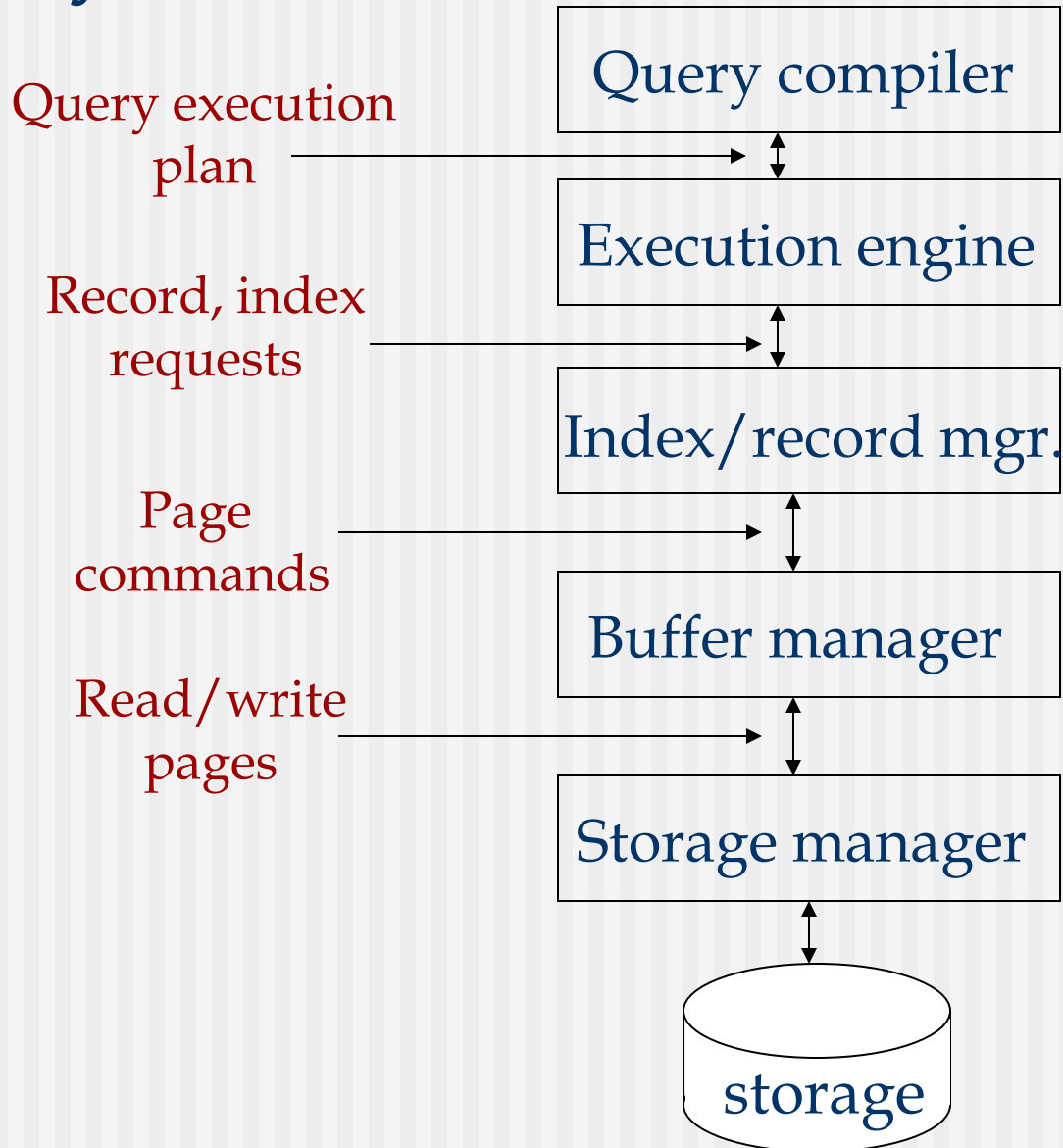

COURSE 9

Query Optimization

Architecture of Database Engine



Query Execution



Schema for Examples

Students (sid: integer, sname: string, age: integer)

Courses (cid: integer, name: string, location: string)

Evaluations (sid: integer, cid: integer, day: date, grade: integer)

■ Students:

- Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

■ Courses:

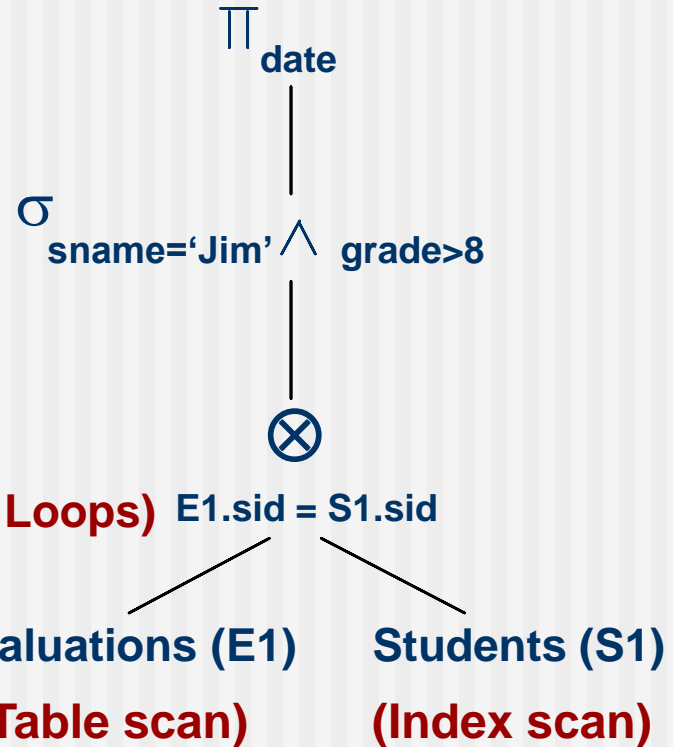
- Each tuple is 50 bytes long, 80 tuples per page, 100 pages.

■ Evaluations:

- Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.

Query Execution Plans

```
SELECT E1.date
FROM Evaluations E1, Students S1
WHERE E1.sid=S1.sid AND
      S1.sname = 'Jim' AND
      E1.grade > 8
```



Query Plan:

- logical tree
- implementation choice at every node
- scheduling of operations.

Some operators are from relational algebra, and others (e.g., scan, group) are not.

The Leaves of the Plan: Scans

- **Table scan**: iterate through the records of the relation.
- **Index scan**: go to the index, from there get the records in the file
- **Sorted scan**: produce the relation in order. Implementation depends on relation size.
- How do we combine Operations?
 - **The iterator model**. Each operation is implemented by 3 functions:
 - *Open*: sets up the data structures and performs initializations
 - *GetNext*: returns the next tuple of the result.
 - *Close*: ends the operations. Cleans up the data structures.
 - Enables pipelining!
 - Contrast with **data-driven materialize model**.
 - Sometimes it's the same (e.g., sorted scan).

Query Optimization Process

- Parse the SQL query into a logical tree:
 - identify distinct blocks (corresponding to nested sub-queries or views).
- Query rewrite phase:
 - apply **algebraic transformations** to yield a cheaper plan.
 - Merge blocks and move predicates between blocks.
- Optimize each block: **join ordering**.
- Complete the optimization: select scheduling (pipelining strategy).

Overview of Query Optimization

- Plan: *Tree of R.A. ops, with choice of alg 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.
- **Practically**: Avoid worst plans!

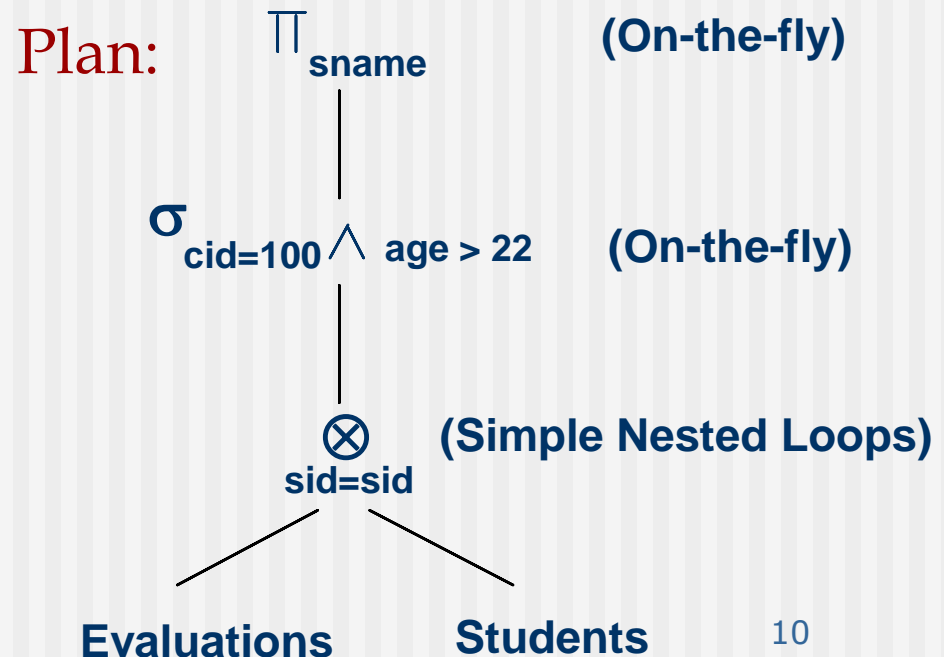
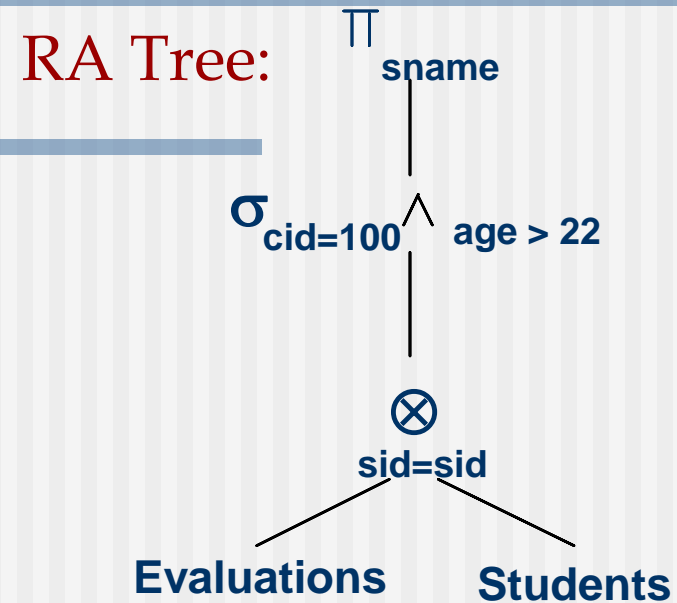
Highlights of System R Optimizer

- Impact:
 - Most widely used currently; works well for < 10 joins.
- **Cost estimation:** Approximate art at best.
 - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
 - Considers combination of CPU and I/O costs.
- **Plan Space:** Too large, must be reduced.
 - Only the space of *left-deep plans* is 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 avoided.

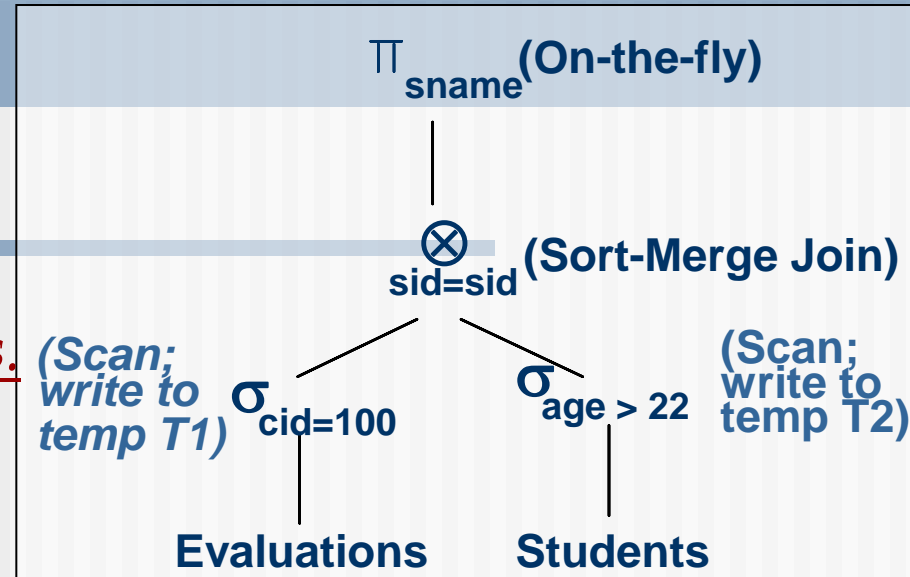
Motivating Example

```
SELECT S.sname
FROM Evaluations E, Students S
WHERE E.sid=S.sid AND
      E.cid=100 AND S.age>22
```

- **Cost:** 500+500*1000 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.



Alternative Plans 1



- Main difference: push selects.
- With 5 buffer pages,
- cost of plan:
 - Scan Evaluations (1000) + write temp T1 (10 pages, if we have 100 courses, uniform distribution). – *total 1010 I/Os*
 - Scan Students (500) + write temp T2 (250 pages, if we have 10 ages). – *total 750 I/Os*
 - Sort T1 (2*2*10), sort T2 (2*3*250), merge (10+250) – *total 1800 I/Os*
 - Total: 3560 page I/Os.
- If we used BNL join, 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 pgs, total < 2000.

Alternative Plans 2

- With clustered index on *cid* of Evaluations, we get
 $100,000/100 = 1000$ tuples on
 $1000/100 = 10$ pages.
- INL with pipelining (outer is not materialized).

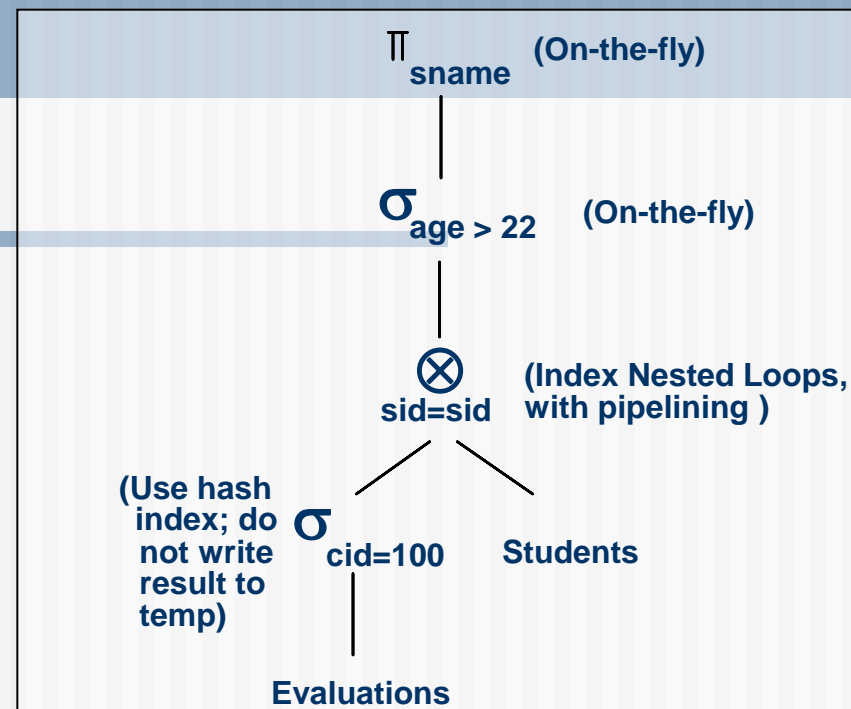
- Projecting out unnecessary fields from outer doesn't help.

- Join column *sid* is a key for Students.

- At most one matching tuple, un-clustered index on *sid* OK.

- Decision not to push *age>22* before the join is based on availability of *sid* index on Students.

Cost: Selection of Evaluations tuples (10 I/Os); for each, must get matching Students tuple (1000×1.2); total **1210 I/Os**.



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. (This is an oversimplification, but serves for now.)

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

Outer block

Nested block

For each block, the plans considered are:

- All available access methods, for each reln in FROM clause.
- All *left-deep join trees* (i.e., all ways to join the relations one-at-a-time, with the inner reln in the FROM clause, considering all reln permutations and join methods.)

Cost Estimation

- For each plan considered, must estimate 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.
- We'll discuss the **System R** cost estimation approach.
 - Very inexact, but works ok in practice.
 - More sophisticated techniques known now.

Statistics and Catalogs

- Need information about the relations and indexes involved. *Catalogs* typically contain at least:
 - number of tuples (NTuples) and number of pages (NPages) for each relation.
 - number of distinct key values (NKeys) and NPages for each index.
 - Index height, low/high key values (Low/High) for each tree 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 Reduction Factors

- Consider a query block:

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

- Maximum number of 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 no. of 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*
 - Term *col₁=col₂* has RF: $1/MAX(NKeys(I_1), NKeys(I_2))$
 - Term *col>value* has RF: $(High(I)-value)/(High(I)-Low(I))$

Relational Algebra Equivalences

- Allow us to choose different join orders and to 'push' selections and projections ahead of joins.
- Selections: $\sigma_{c1 \wedge \dots \wedge cn} (R) \equiv \sigma_{c1}(\dots(\sigma_{cn} (R)))$ (*Cascade*)
 $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (*Commute*)
- Projections: $\pi_{a1}(R) \equiv \pi_{a1}(\dots (\pi_{an}(R)))$ (*Cascade*)
- Joins: $R \otimes (S \otimes T) \equiv (R \otimes S) \otimes T$ (*Associative*)
 $(R \otimes S) \equiv (S \otimes R)$ (*Commute*)
+ show that: $R \otimes (S \otimes T) \equiv (T \otimes R) \otimes S$

More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection.
- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.
- A selection on just attributes of R commutes with $R \bowtie S$. (i.e., $\sigma(R \bowtie S) \equiv \sigma(R) \bowtie S$)
- Similarly, if a projection follows a join $R \bowtie S$, we can 'push' it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.

Enumeration of Alternative Plans

- 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 ops 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, about 1.2 for hash index.
 - Clustered index I matching one or more selects:
 - $(NPages(I)+NPages(R)) * \text{product of RF's of matching selects.}$
 - Non-clustered index I matching one or more selects:
 - $(NPages(I)+NTuples(R)) * \text{product of RF's of matching sels.}$
 - Sequential scan of file:
 - $NPages(R).$
- + **Note**: Typically, no duplicate elimination on projections!
(Exception: Done on answers if user says DISTINCT.)

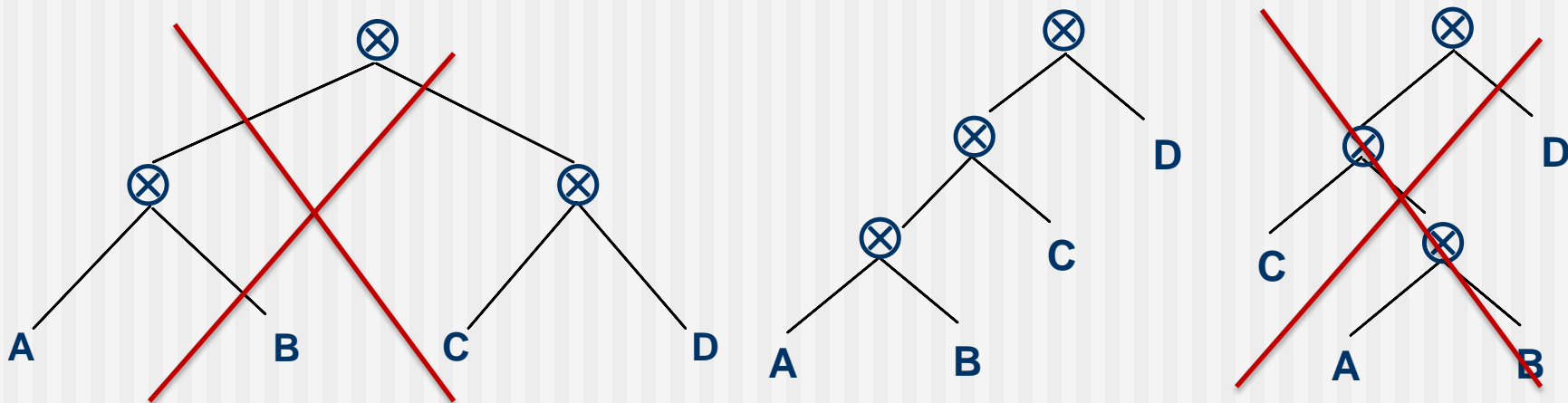
Example

```
SELECT S.sid  
FROM Students S  
WHERE S.age=20
```

- If we have an **index on *age***:
 - $(1/NKeys(I)) * NTuples(R) = (1/10) * 40000$ tuples retrieved.
 - **Clustered index**: $(1/NKeys(I)) * (NPages(I) + NPages(R)) = (1/10) * (50 + 500)$ pages are retrieved. (This is the **cost**)
 - **Unclustered index**: $(1/NKeys(I)) * (NPages(I) + NTuples(R)) = (1/10) * (50 + 40000)$ pages are retrieved.
- If we have an **index on *sid***:
 - Would have to retrieve all tuples/pages. With a **clustered index**, the **cost** is **50+500**, with **un-clustered index**, **50+40000**.
- Doing a **file scan**:
 - We retrieve all file pages (**500**).

Queries Over Multiple Relations

- 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.*
 - 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., SM 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.
- 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.

Enumeration of Plans (cont.)

- 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, unless all predicates in WHERE have been used up.
 - i.e., avoid Cartesian products if possible.
- In spite of pruning plan space, this approach is still exponential in the number of tables.

Example

Pass1:

■ *Students*: B+ tree matches $age > 22$, and is probably cheapest. However, if this selection is expected to retrieve a lot of tuples, and index is un-clustered, file scan may be cheaper.

- Still, B+ tree plan kept (because tuples are in *age* order).

■ *Evaluations*: B+ tree on *cid* matches $cid=100$; 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., *Evaluations as outer*: Hash index can be used to get *Students* tuples that satisfy $sid = \text{outer tuple's } sid \text{ value.}$)

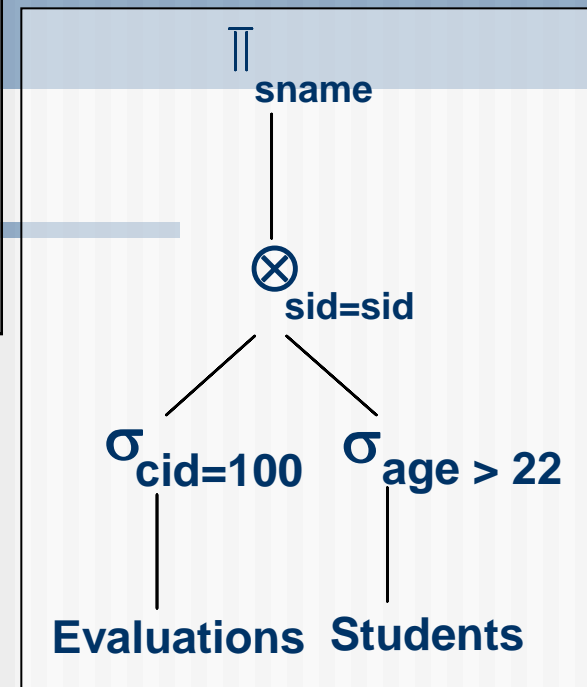
Students:

B+ tree on *age*

Hash on *sid*

Evaluations:

B+ tree on *cid*



Nested Queries

- 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.

```
SELECT S.sname
FROM Students S
WHERE EXISTS
  (SELECT *
   FROM Evaluations E
   WHERE E.cid=103
   AND E.sid=S.sid)
```

Nested block to optimize:

```
SELECT *
FROM Evaluations E
WHERE E.cid=103
   AND S.sid= outer value
```

Equivalent non-nested query:

```
SELECT S.sname
FROM Students S,
Evaluations E
WHERE S.sid=E.sid
   AND E.cid=103
```