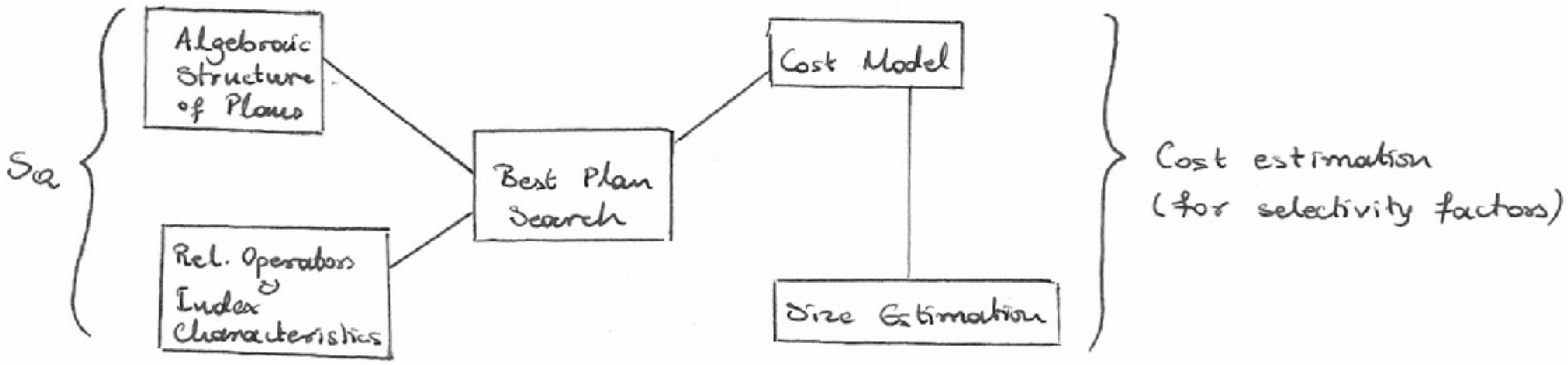
Query Optimization

- $\forall$ query Q, $\exists$ a set of equivalent plans that the DBMS can follow to answer the query. Let this set be S_Q.
- Optimizer chooses plan p_0, s.t. $cost(p_0) = min_{S_Q}\{cost(p) \mid p \in S_Q\}$

OPTIMIZER'S ARCHITECTURE



- Algebraic Structure of plans : Translation of a query into Relational Algebra. The system considers all the equivalent expressions, based on the following equivalence rules (here, only a basic subset is presented).
  - $\sigma_{\theta_1} \sigma_{\theta_2}(S) \equiv \sigma_{\theta_2} \sigma_{\theta_1}(S)$ (Selections are commutative)
  - $R \underset{A=B}{\bowtie} S \equiv S \underset{B=A}{\bowtie} R$ (Although the final result the same, the cost might be different).
  - $(R \underset{A=B}{\bowtie} S) \underset{C=D}{\bowtie} T \equiv R \underset{A=S.B}{\bowtie} (S \underset{C=D}{\bowtie} T)$ (Associativity of Joins)
  - $\sigma_{R.A=a}(R \bowtie S) = (\sigma_{A=a} R) \underset{A=B}{\bowtie} S$ (Possibly cheaper)
  - $\Pi_{R.C,\,S.D}(R \underset{A=B}{\bowtie} S) = \Pi_{C,D}((\Pi_{A,C} R) \underset{A=B}{\bowtie} (\Pi_{B,D} S))$

Based on these equivalence rules (and others that can be found in your books on page 488), the system tries to compute a sufficiently large S_Q, but not the full S_Q (huge)

- Assumptions that shrink S_Q:

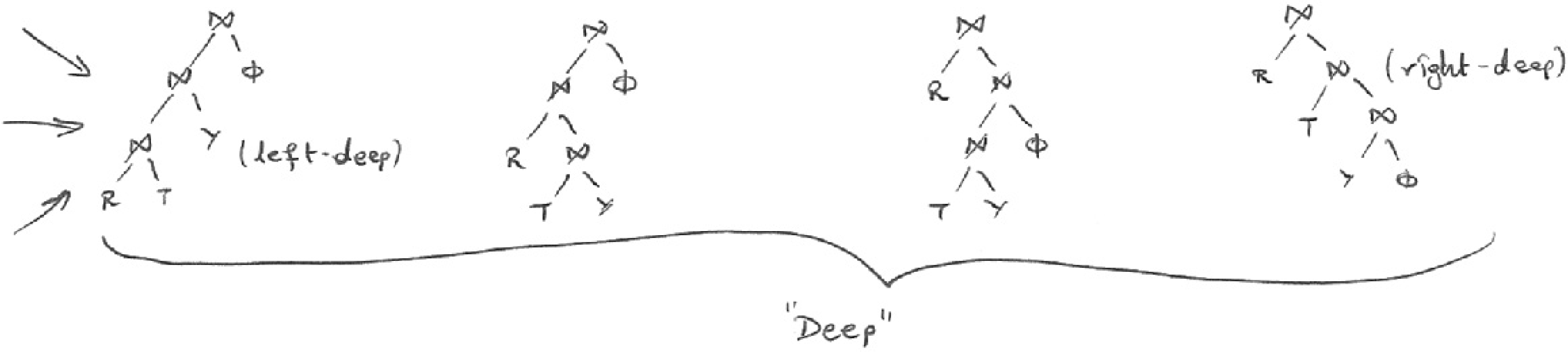
1. Plans with cartesian products are never examined, unless they are required by the query.

   Why? '$\times$' tends to produce huge relations $\rightsquigarrow$ huge temporary results

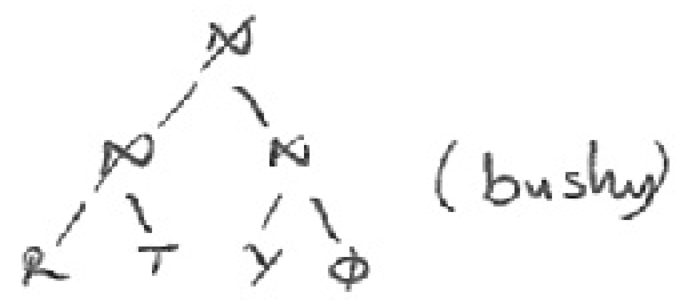
2. The right relation of a join is a base relation and not an intermediate result

e.g. $R(A,B)$, $T(B,C)$, $Y(C,D)$, $\Phi(D,E)$

$$(((R \underset{B}{\bowtie} T) \underset{C}{\bowtie} Y) \underset{D}{\bowtie} \Phi) \equiv (R \underset{B}{\bowtie} (T \underset{C}{\bowtie} Y)) \underset{D}{\bowtie} \Phi \equiv R \underset{B}{\bowtie} ((T \underset{C}{\bowtie} Y) \underset{D}{\bowtie} \Phi) \equiv R \underset{B}{\bowtie} (T \underset{C}{\bowtie} (Y \underset{D}{\bowtie} \Phi)) \equiv$$



$$\equiv ((R \underset{B}{\bowtie} T) \underset{C}{\bowtie} (Y \underset{D}{\bowtie} \Phi))$$



- Why?:
  a) Intermediate results are often big, with the consequence that the cost of that join is penalized.
  b) Nested Loops tend to use indexes for the inner relations, to reduce the overall cost.
  c) Nested Loop joins can be pipelined, if we have enough memory, so as to avoid the cost of reading and writing intermediate results ("on the fly" execution).

3. Selections and projections are <u>always</u> pipelined with joins, to reduce the sizes of intermediate results.

   eg. $\pi_{T.C}(\sigma_{S.A=a}(S \underset{B=B}{\bowtie} T)) \equiv \pi_{T.C}((\sigma_{S.A=a} S) \underset{B=B}{\bowtie} T)$

   Why don't we push $\pi_{T.C}$ inside?

• Plan Enumeration

- Algorithm based on Dynamic Programming
- Every alternative is an ordering of the joins, the join algorithm to be used and the access method (index or file scan) of each relation. The first component corresponds to the algebraic element, the rest on the physical element of the DBMS (i.e. the box termed: "Rel. Operators and Index Characteristics").

- <u>Interesting Orders</u>: An operation <u>may</u> produce an (intermediate) relation ordered on an attribute (eg. selection ~~b~~ using a B+ index).
  If an intermediate result is sorted according to an attribute, A, this order is considered <u>interesting</u> if:
  a) A participates in a join in the query (so as to avoid sorting the relation if we use the Sort-Merge join algorithm).
  b) The final result has been asked explicitly to be ordered wrt A (group by/order by clause).

- <u>System-R algorithm</u>:

  N passes, for queries involving N relations

  <u>Pass 1</u>: • ∀ relation, we consider all indexes that we can use to access it, plus using the heap-scan method.
  - These alternatives are being clustered in "equivalence classes", where each alternative in a class produces an (intermediate) relation sorted wrt <u>the same</u> interesting order. We include one class for which there is no interesting order.
  - From each class we keep for the next pass the cheapest solution. Especially for the "no interesting order" class, we keep its cheapest alternative, only if it is globally the cheapest one.

  <u>Pass 2</u>: • ∀ pair of relations that is joined in the query, we consider all the alternative ways of joining them, based on:
  - the join algorithms supported by the DBMS
  - the access methods that "survived" from pass 1.
  - any other assumptions the DBMS makes (eg only left-deep trees, etc).
  - We form equivalence classes as before, 1 ∀ interesting order + 1 for the unordered case
  - We select the best alternative for each class as before.

  <u>Pass 3</u>: ∀ 3 relations joined in the query: follow the same algorithm, by taking into account the plans that "survived" from passes 2 & 1.

  ⋮

  <u>Pass N</u>: ∀ N relations joined (that is the whole query), the same as before, selecting access plans from passes N-1 and 1. Now we have 1 equivalence class, from which we choose the overall cheapest alternative.

- By considering plans from the k-1 and 1st pass, we reject right-deep or bushy query plans.

E.g. Consider the schema:

Clerk (Name, salary, deptno)

Department (deptno, floor, manager)

- Query: Find the names of clerks with salaries > 500k that work on the second floor.

  ↳ $\Pi_{name}\, \sigma_{sal > 500k}\, \sigma_{floor = 2}\, (\text{Clerk} \bowtie_{deptno} \text{Department})$

- Physical Schema:
  - Department: hash index on floor.
  - Clerk: B+ tree on deptno, B+ tree on salary

- Interesting orders: Clerk.~~salary~~ deptno, Dept. deptno } only these can be found in a join

- Algorithm: (2 passes)

Pass 1:

- Accessing Clerk: B+ on Clerk.deptno | B+ on Clerk.sal | ~~Heap Scan~~

  Cost: 700 | 200 | ~~600~~

  700: Interesting Order - class 1

  200, ~~600~~: No interesting order - class 2

- Accessing Dept: Hashing on Dept. floor | ~~Heap Scan~~

  Cost: 50 | ~~200~~

  No interesting order - class 3

Pass 2: 1. Clerk ⋈ Dept with NLJ:

Clerk ⋈ Dept (NLJ)

B+: Cl. sal, Hash: D. floor — Cost: 1800

B+ Cl. deptno, Hash D. floor — Cost: 3000

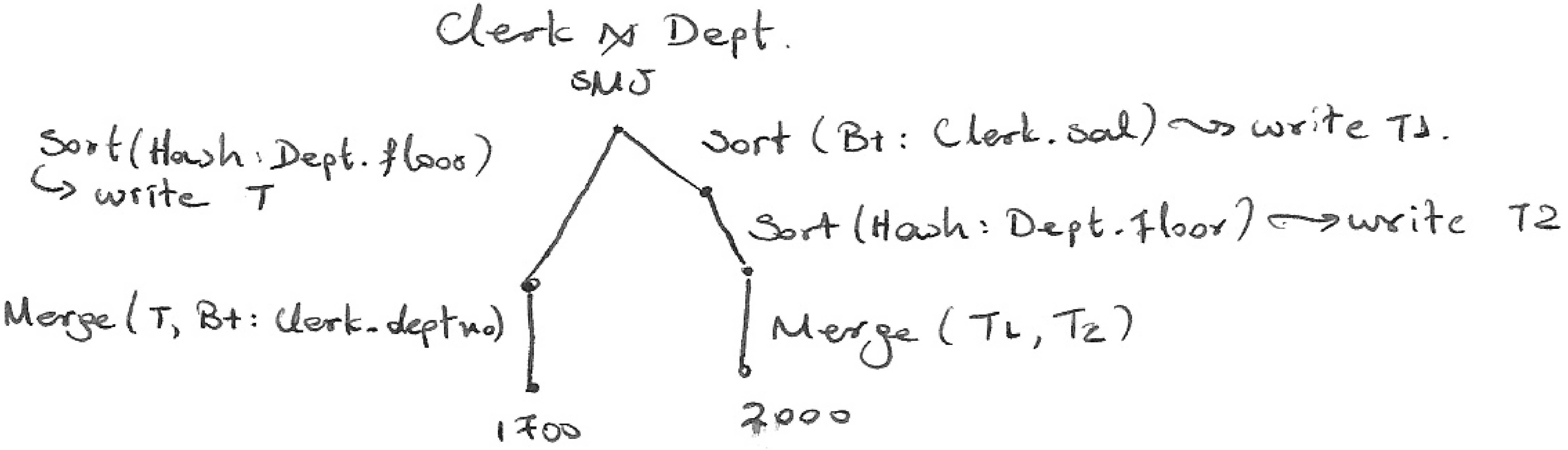
Dept ⋈ Clerk (NLJ)

Hash: Dept. floor, B+: Cl. sal — 2500

Hash: Dept. floor, B+: Cl. deptno — 1500 cheapest

No Interesting order - class 1

2. Clerk ⋈ Dept with Sort-Merge Join. (here we don't need to examine Dept. ⋈ Clerk : SMJ symmetric).



Interesting order (on deptno) - Class 2

- If we had another join, we would pick the cheapest from each class, and so on...