Access Path Selection in a Relational DBMS

Original Slides by
Presentation: Stephen Ingram
Modified by: Rachel Pottinger, Sarah Elhammadi

Why bother to optimize?

- Queries must be executed and execution takes time
- There are multiple execution plans for most queries
- Some plans cost less than others

Simple Example

- SELECT * FROM A,B,C WHERE A.n = B.n AND B.m = C.m
- A = 100 tuples
- B = 50 tuples
- C = 2 tuples
- Which plan is cheaper?
  - Join(C, Join(A, B))
  - Join(A, Join(B, C))

How did we find the right one?

1. Measure the <u>cost</u> of each query
2. Enumerate possibilities
3. Pick the least expensive one

- Is that all?

But the search space is too big

- Just for this simple join example, we have a factorial search space (n!)
- Just to remind you,
  - 20! = 2,432,902,008,176,640,000
- So now what do we do?

Use Statistics

- For each relation keep track of
  - Cardinality of tuples
  - Cardinality of pages
  - Etc.
- For each index keep track of
  - number of distinct keys in index I.
  - the number of pages in index I.
- Use these statistics in conjunction with
  - Predicates
  - Interesting Orders

Discussion

This all seems like a bit of a black art. And yet it largely works. Does this surprise you? Why or why not?

Predicates

- Predicates like =, >, NOT, etc. reduce the number of tuples

- THUS: Evaluate predicates as early as possible

Interesting Orders

- GROUP BY and ORDER BY or Join column order are interesting orders
- To find the cheapest plan we examine the the cheapest access path that produces tuples in interesting order or cheapest unordered plan.
- Sorted orders by merge scan can significantly reduce the cost of subsequent joining even though nested loop join could cheaper for the current join.

But…

- Statistics alone cannot save us
  - Expensive to compute
  - Can’t keep track of all joint statistics

- Compromise on statistics
  - Periodically update stats for each relation
- Compromise on search
  - Dynamic programming approach

Dynamic programming (Wikipedia)

- *Optimal substructure* means that optimal solutions of subproblems can be used to find the optimal solutions of the overall problem.

1. Break the problem into smaller subproblems.
2. Solve these problems optimally using this three-step process recursively.
3. Use these optimal solutions to construct an optimal solution for the original problem.

Optimal Substructure in Joins

- *Exploits principle of optimality of cost model.*
- An N-Join is really just a sequence of 2-Joins
  - 2-join becomes a single composite relation

- Find the cheapest join of a subset of the N tables and store (memoization)
- This costs 2^n , which is << n!

From the Top

- Enumerate access paths to each relation
  - Sequential scans
  - Interesting orders
- Enumerate access paths to join a second relation to these results (if there is a predicate to do so)
  - Nested loop (unordered)
  - Merge (interesting order)
- Compare with equivalent solutions found so far but only keep the cheapest

Example Schema

EMP

NAME	DNO	JOB	SAL
SMITH	50	12	8500
JONES	50	5	15000
DOE	51	5	9500

DEPT

DNO	DNAME	LOC
50	MFG	DENVER
51	BILLING	BOULDER
52	SHIPPING	DENVER

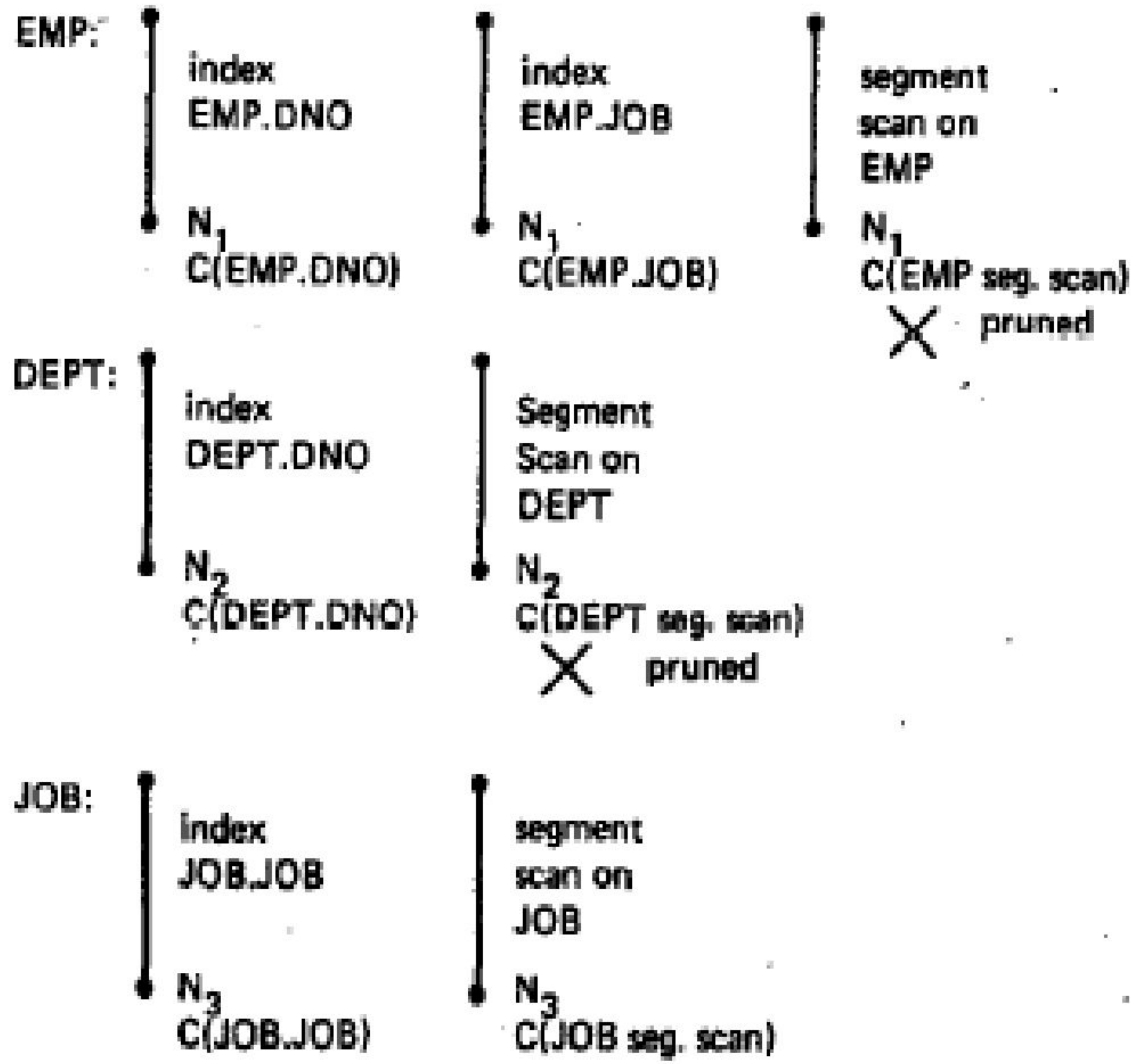
JOB

JOB	TITLE
5	CLERK
6	TYPIST
9	SALES
12	MECHANIC

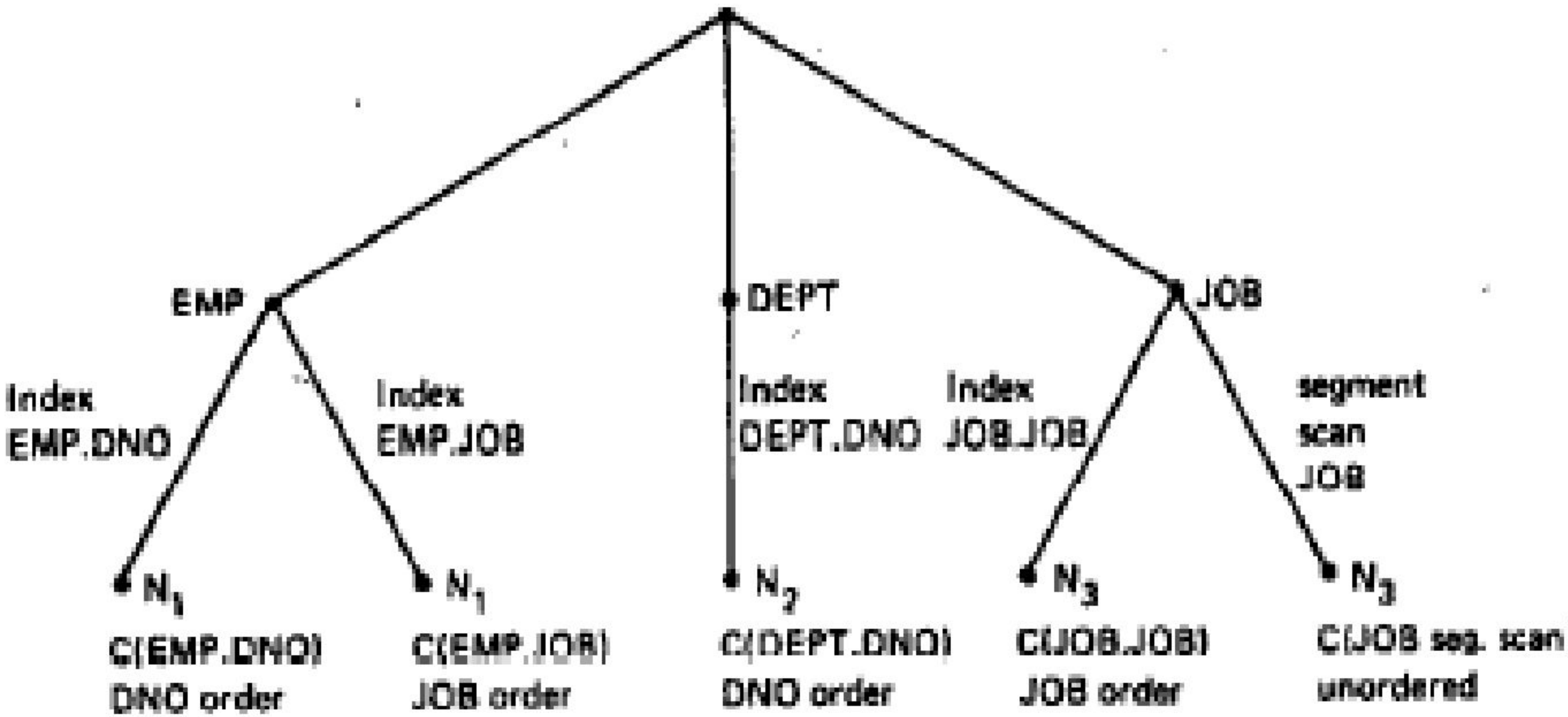
Example Query

```
SELECT   NAME, TITLE, SAL, DNAME
FROM     EMP, DEPT, JOB
WHERE    TITLE='CLERK'
AND      LOC='DENVER'
AND      EMP.DNO=DEPT.DNO
AND      EMP.JOB=JOB.JOB
```

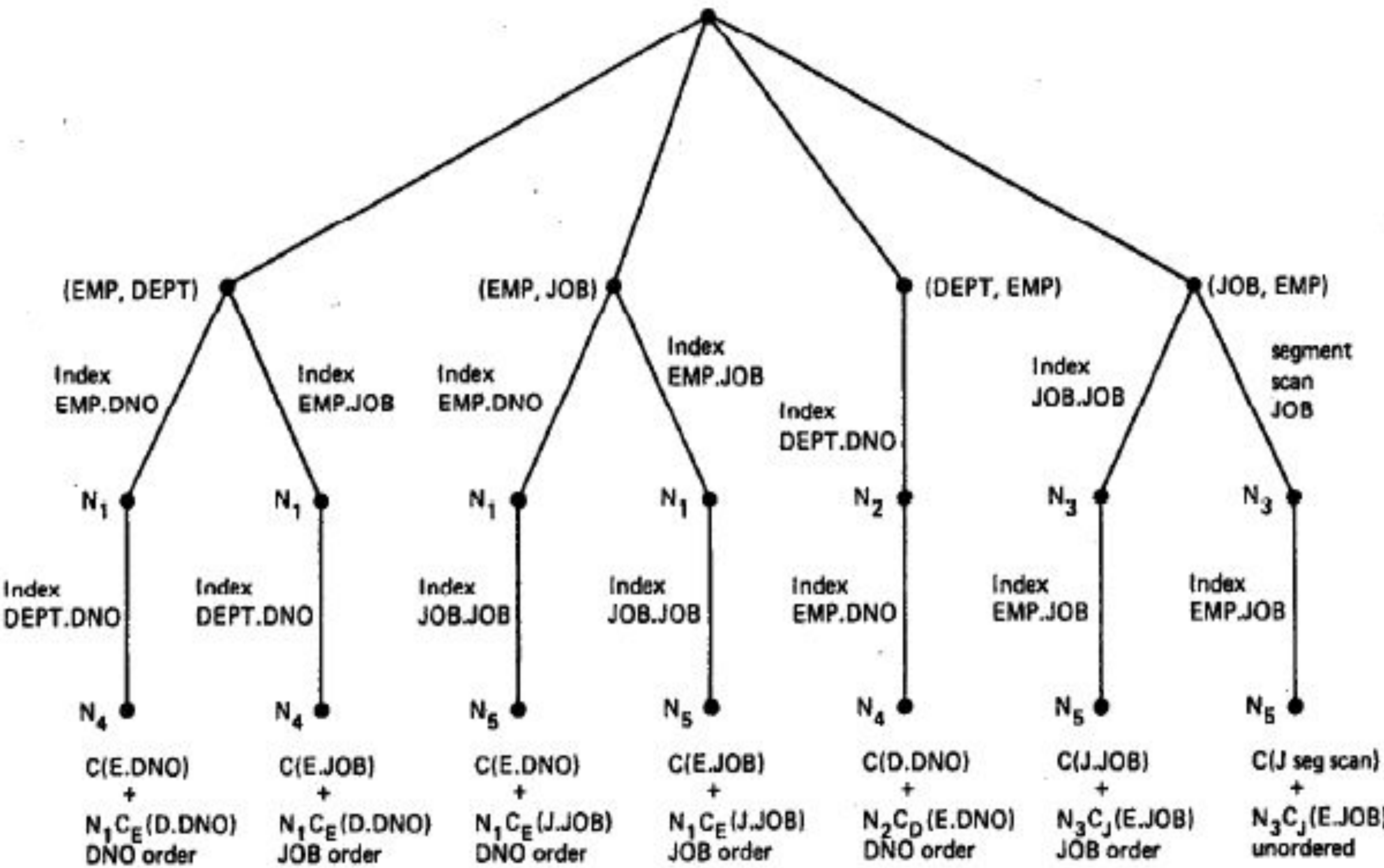
Example Initial Access Paths



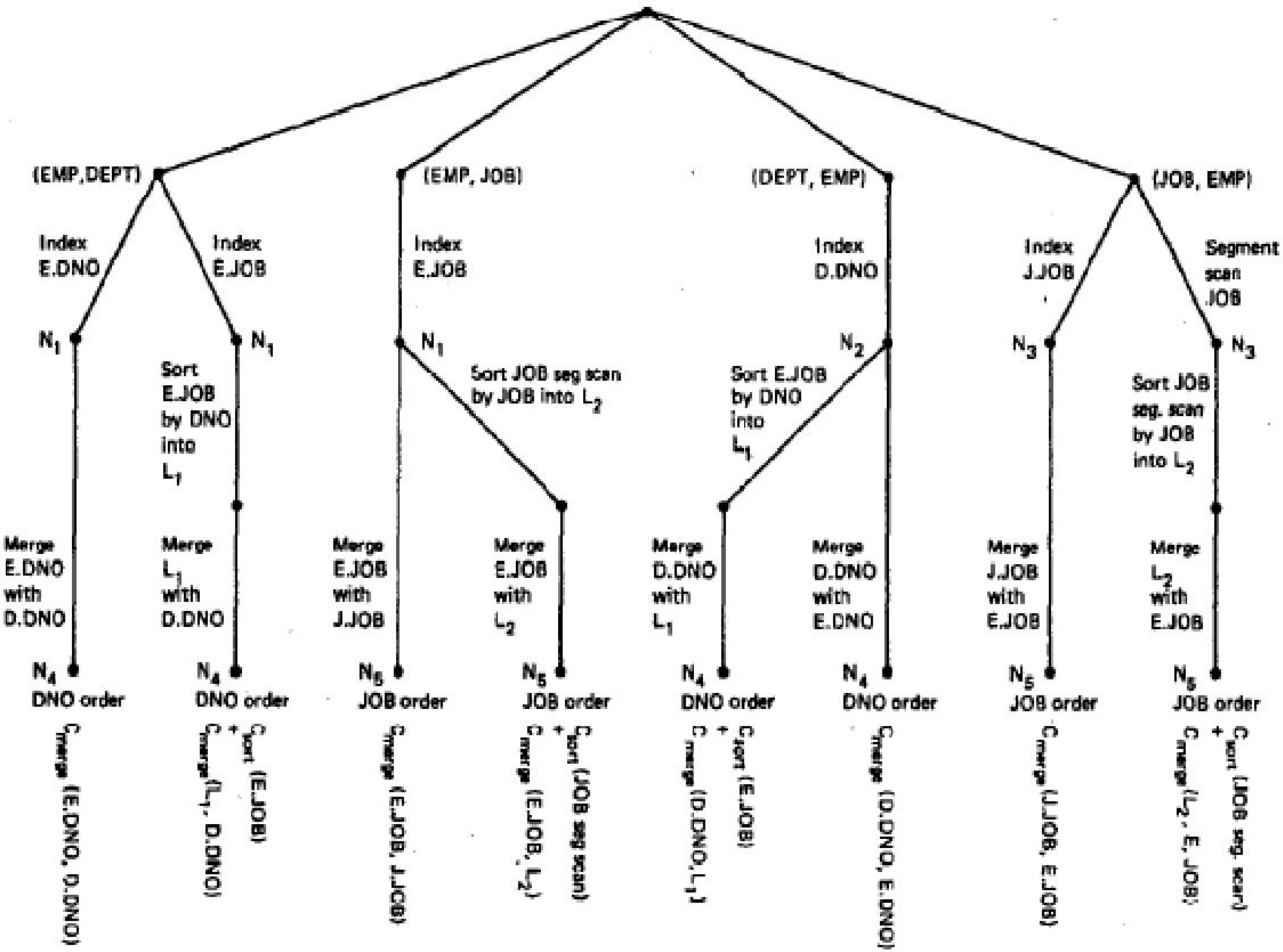
Example Search Tree



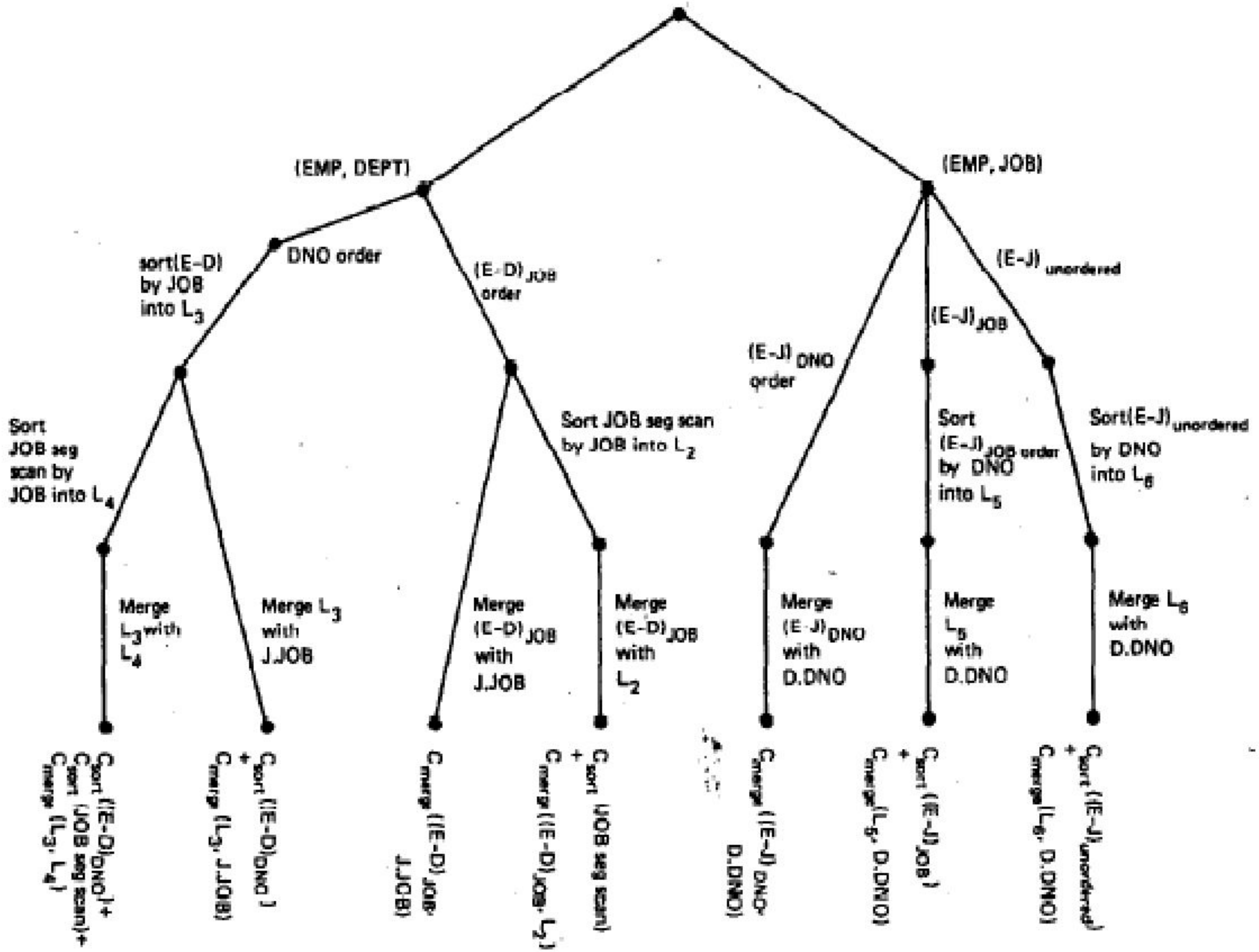
2 Relations Nested Loop



2 Relations Merge Join



Prune and 3 Relations



Major Contributions of Paper

- Cost based optimization
  - Statistics
  - CPU utilization (for sorts, etc.)
- Dynamic programming approach
- Interesting Orders

Discussion

We saw a lot of these ideas still in use 20 years later in the previous paper. Are you surprised on how much was kept, or not?