Implementation of Relational Operations

**Iterator Model**

- Relational operators are all subclasses of the class `iterator`:
  ```cpp
  class iterator {
    void init();
    tuple next();
    void close();
    iterator &inputs[];
    // additional state goes here
  }
  ```
- Note:
  - Edges in the graph are specified by inputs (max 2, usually)
  - Any iterator can be input to any other!

**Simple Selections**

\[ \sigma_{R.\text{attr op value}} (R) \]

- Size of result approximated as
  \((\text{size of } R) \times \text{selectivity}\)

- If no appropriate index exists:
  - Must scan the whole relation
  \[ \text{cost} = |R| \]
  - For "reserves" = 1000 I/Os.
  - Or...

**Use an index!**

- With index on selection attribute:
  1. Use index to find qualifying data entries
  2. Retrieve corresponding data records

  Total cost = cost of step 1 + cost of step 2
  - For "reserves", if selectivity = 10\% (100 pages, 10000 tuples):
    - If *clustered* index, cost is a little over 100 I/Os;
    - If *unclustered*, could be up to 10000 I/Os!
  ... unless...
Refinement for unclustered indexes

1. Find qualifying data entries.
2. Sort the rid’s of the data records to be retrieved.
3. Fetch rids in order.
   Each data page is looked at just once (though # of such pages likely to be higher than with clustering).

2 Approaches to General Selections

Approach I:
1. Find the cheapest access path
2. retrieve tuples using it
3. Apply any remaining terms that don’t match the index

Approach II: use 2 or more matching indexes.
1. From each index, get set of rids
2. Compute intersection of rid sets
3. Retrieve records for rids in intersection
4. Apply any remaining terms

Projection

• Issue is removing duplicates.
• Use sorting!!
  1. Scan R, extract only the needed attributes
  2. Sort the resulting set
  3. Remove adjacent duplicates

Cost:
Reserves with size ratio 0.25 = 250 pages.
With 20 buffer pages can sort in 2 passes, so:
\[ 1000 + 250 + 2 \times 250 + 250 = 2500 \text{ I/Os} \]

Projection Tricks

Modify the external sort algorithm:
- Modify Pass 0 to eliminate unwanted fields.
- Modify Passes 1+ to eliminate duplicates.

If an index search key contains all wanted attrs:
- Do index-only scan
  - Apply projection techniques to data entries (much smaller!)

If a B+Tree index search key prefix has all wanted attrs:
- Do in-order index-only scan
  - Compare adjacent tuples on the fly (no sorting required!)
Simple Nested Loops Join

R \bowtie S: foreach tuple r in R do
  foreach tuple s in S do
    if r_i == s_j then add <r, s> to result

Cost = \( (p_R | R |) | S | + | R | \)

Page-Oriented Nested Loops Join

R \bowtie S: foreach page b_R in R do
  foreach page b_S in S do
    foreach tuple r in b_R do
      foreach tuple s in b_S do
        if r_i == s_j then add <r, s> to result

Cost = |R| * |S| + |R|

Block Nested Loops Join

- Page-oriented NL doesn’t exploit extra buffers:
- Idea to use memory efficiently:

Cost: Scan outer + (#outer blocks * scan inner)
#outer blocks = \[ \text{# of pages of outer / blocksize} \]

Index Nested Loops Join

R \bowtie S: foreach tuple r in R do
  foreach tuple s in S where r_i == s_j do
    add <r, s> to result

Cost = |R| + (|R| * p_s) + cost to find matching S tuples

- If index uses Alt. 1, cost = cost to traverse tree from root to leaf.
  For Alt. 2 or 3:
  1. Cost to lookup RID(s); typically 2-4 I/O’s for B-Tree.
  2. Cost to retrieve records from RID(s); depends on clustering.
    - Clustered index: 1 I/O per page of matching S tuples.
    - Unclustered: up to 1 I/O per matching S tuple.
Sort-Merge Join

1. Sort R on join attr(s)
2. Sort S on join attr(s)
3. Scan sorted-R and sorted-S in tandem, to find matches

- Cost: \( \text{Sort } R + \text{Sort } S + (|R| + |S|) \)
  - But in worst case, last term could be \(|R| \times |S|\)

You could do the join during the final merging pass of sort!
1. Read R and write out sorted runs
2. Read S and write out sorted runs
3. Merge R-runs and S-runs, and find R S matches

\[ \text{Cost} = 3^*|R| + 3^*|S| \]

Eliminate duplicates using hashing

- Two phases:
  - Partition: use a hash function \( h \) to split tuples into partitions on disk.
    - Key property: all matches live in the same partition.
  - ReHash: for each partition on disk, build a main-memory hash table using a hash function \( h_2 \)

Two Phases

- **Partition:**

- **Rehash:**

Duplicate Elimination using Hashing

- read one bucket at a time
- for each group of identical tuples, output one
Cost of External Hashing

cost = 4|R| IO's

Memory Requirement

• How big of a table can we hash in two passes?
  – B-1 "partitions" result from Phase 0
  – Each should be no more than B pages in size
  – Answer: B(B-1).
    Said differently:
    We can hash a table of size N pages in about \( \frac{\sqrt{N}}{B} \) space
  – Note: assumes hash function distributes records evenly!

• Have a bigger table? Recursive partitioning!

Cost of Hash Join

• Partitioning phase: read+write both relations
  \( \Rightarrow 2(|R|+|S|) \) I/Os

• Matching phase: read+write both relations
  \( \Rightarrow |R|+|S| \) I/Os

• Total cost of 2-pass hash join = 3(|R|+|S|)
Hybrid Hashing

- **Idea:** keep one of the hash buckets in memory!

![](image)

**Q:** how do we choose the value of \( k \)?

**Summary: Hashing vs. Sorting**

- **Sorting pros:**
  - Good if input already sorted, or need output sorted
  - Not sensitive to data skew or bad hash functions

- **Hashing pros:**
  - Often cheaper due to hybrid hashing
  - For join: \# passes depends on size of smaller relation
  - Highly parallelizable

**Cost-based Query Sub-System**

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

- **Query Parser**
- **Query Optimizer**
  - **Plan Generator**
  - **Plan Cost Estimator**

- **Catalog Manager**
  - **Schema**
  - **Statistics**

- Usually there is a heuristics-based rewriting step before the cost-based steps.
Alternative Plans – Push Selects (No Indexes)

\[ \pi_{\text{name}} \] (On-the-fly)
\[ \sigma_{\text{bid}=100} \land \text{rating} > 5 \] (On-the-fly)
\[ E=3 \] (Page-Oriented Nested loops)
\[ \text{sid}=\text{sid} \]
\[ \text{Reserves} \]
\[ \text{Sailors} \]

500,500 IOs

\[ \pi_{\text{name}} \] (On-the-fly)
\[ \sigma_{\text{bid}=100} \land \text{rating} > 5 \] (On-the-fly)
\[ E=3 \] (Page-Oriented Nested loops)
\[ \text{sid}=\text{sid} \]
\[ \text{Reserves} \]
\[ \text{Sailors} \]

250,500 IOs

Alternative Plans – Push Selects (No Indexes)

\[ \pi_{\text{name}} \] (On-the-fly)
\[ \sigma_{\text{bid}=100} \land \text{rating} > 5 \] (On-the-fly)
\[ E=3 \] (Page-Oriented Nested loops)
\[ \text{sid}=\text{sid} \]
\[ \text{Reserves} \]
\[ \text{Sailors} \]

250,500 IOs

\[ \pi_{\text{name}} \] (On-the-fly)
\[ \sigma_{\text{bid}=100} \land \text{rating} > 5 \] (On-the-fly)
\[ E=3 \] (Page-Oriented Nested loops)
\[ \text{sid}=\text{sid} \]
\[ \text{Reserves} \]
\[ \text{Sailors} \]

250,500 IOs

Alternative Plans – Push Selects (No Indexes)

\[ \pi_{\text{name}} \] (On-the-fly)
\[ \sigma_{\text{bid}=100} \land \text{rating} > 5 \] (On-the-fly)
\[ E=3 \] (Page-Oriented Nested loops)
\[ \text{sid}=\text{sid} \]
\[ \text{Reserves} \]
\[ \text{Sailors} \]

6000 IOs

\[ \pi_{\text{name}} \] (On-the-fly)
\[ \sigma_{\text{bid}=100} \land \text{rating} > 5 \] (On-the-fly)
\[ E=3 \] (Page-Oriented Nested loops)
\[ \text{sid}=\text{sid} \]
\[ \text{Reserves} \]
\[ \text{Sailors} \]

6000 IOs

Alternative Plans – Push Selects (No Indexes)

\[ \pi_{\text{name}} \] (On-the-fly)
\[ \sigma_{\text{bid}=100} \land \text{rating} > 5 \] (On-the-fly)
\[ E=3 \] (Page-Oriented Nested loops)
\[ \text{sid}=\text{sid} \]
\[ \text{Reserves} \]
\[ \text{Sailors} \]

6000 IOs

\[ \pi_{\text{name}} \] (On-the-fly)
\[ \sigma_{\text{bid}=100} \land \\text{rating} > 5 \] (On-the-fly)
\[ E=3 \] (Page-Oriented Nested loops)
\[ \text{sid}=\text{sid} \]
\[ \text{Reserves} \]
\[ \text{Sailors} \]

4250 IOs

\[ \text{(Scan & Write to temp T2)} \] (On-the-fly)
Alternative Plans – Push Selects (No Indexes)

- Reserves (sid=sid)
- Sailors (bid=100)
- Rating > 5
- Sname

(Scan; write to temp T2)

4250 IOs

More Alternative Plans (No Indexes)

Main difference: Sort Merge Join

With 5 buffers, cost of plan:
- Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution) = 1010.
- Scan Sailors (500) + write temp T2 (250 pages, if have 10 ratings) = 750.
- Sort T1 (2*2*10) + sort T2 (2*4*250) + merge (10+250) = 2300
- Total: 4060 page I/Os.

If use BNL join, join = 10+4*250, total cost = 2770.

Can also `push` projections, but must be careful!
- T1 has only sid, T2 only sid, sname:
- T1 fits in 3 pgs, cost of BNL under 250 pgs, total < 2000.

More Alt Plans: Indexes

With clustered index on bid of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.

INL with outer not materialized.
- Projecting unnecessary fields from outer doesn’t help.
- Join column sid is a key for Sailors.
- At most one matching tuple, unclustered index on sid OK.
- Decision not to push rating>5 before the join is based on availability of sid index on Sailors.
- Cost: Selection of Reserves tuples (10 I/Os); then, for each, must get matching Sailors 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.

SELECT S.sname
FROM Sailors S
WHERE S.age
IN
(SELECT MAX (S2.age)
FROM Sailors S2
GROUP BY S2.rating)

For each block, the plans considered are:
- All available access methods, for each relation in FROM clause.
- All left-deep join trees (i.e., right branch always a base table, consider all join orders and join methods.)
<table>
<thead>
<tr>
<th>Relational Algebra Equivalences</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Allow us to choose different join orders and to ‘push’ selections and projections ahead of joins.</td>
</tr>
<tr>
<td>• Selections:</td>
</tr>
<tr>
<td>$\sigma_{c_1, \ldots, c_n}(R) = \sigma_{c_1}(\ldots(\sigma_{c_n}(R))\ldots)$ (cascade)</td>
</tr>
<tr>
<td>$\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_1}(\sigma_{c_1}(R))$ (commute)</td>
</tr>
<tr>
<td>• Projections:</td>
</tr>
<tr>
<td>$\pi_{a_1}(R) = \pi_{a_1}(\ldots(\pi_{a_1}, \ldots, a_n(R))\ldots)$ (cascade)</td>
</tr>
<tr>
<td>• Cartesian Product</td>
</tr>
<tr>
<td>$R \times (S \times T) \equiv (R \times S) \times T$ (associative)</td>
</tr>
<tr>
<td>$R \times S \equiv S \times R$ (commutative)</td>
</tr>
<tr>
<td>– This means we can do joins in any order.</td>
</tr>
<tr>
<td>• But...beware of cartesian product!</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>More Equivalences</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Eager projection</td>
</tr>
<tr>
<td>– Can cascade and &quot;push&quot; some projections thru selection</td>
</tr>
<tr>
<td>– Can cascade and &quot;push&quot; some projections below one side of a join</td>
</tr>
<tr>
<td>– Rule of thumb: can project anything not needed &quot;downstream&quot;</td>
</tr>
<tr>
<td>• Selection between attributes of the two arguments of a cross-product converts cross-product to a join.</td>
</tr>
<tr>
<td>• A selection on just attributes of R commutes with $R \times S$. (i.e., $\sigma(R \times S) = \sigma(R) \times S$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cost Estimation</th>
</tr>
</thead>
<tbody>
<tr>
<td>• For each plan considered, must estimate total cost:</td>
</tr>
<tr>
<td>– Must estimate cost of each operation in plan tree.</td>
</tr>
<tr>
<td>• Depends on input cardinalities.</td>
</tr>
<tr>
<td>– We’ve already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)</td>
</tr>
<tr>
<td>– Must estimate size of result for each operation in tree!</td>
</tr>
<tr>
<td>• Use information about the input relations.</td>
</tr>
<tr>
<td>• For selections and joins, assume independence of predicates.</td>
</tr>
<tr>
<td>– In System R, cost is boiled down to a single number consisting of $#I/O + \text{factor} \times #CPU$ instructions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistics and Catalogs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Need information about the relations and indexes involved. Catalogs typically contain at least:</td>
</tr>
<tr>
<td>– # tuples ($NTuples$) and # pages ($NPages$) per rel'n.</td>
</tr>
<tr>
<td>– # distinct key values ($NKeys$) for each index.</td>
</tr>
<tr>
<td>– low/high key values ($Low/High$) for each index.</td>
</tr>
<tr>
<td>– Index height ($IHeight$) for each tree index.</td>
</tr>
<tr>
<td>– # index pages ($INPages$) for each index.</td>
</tr>
<tr>
<td>• Catalogs updated periodically.</td>
</tr>
<tr>
<td>– Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.</td>
</tr>
</tbody>
</table>
### Size Estimation and Reduction Factors

Consider a query block:
- 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.
- RF usually called "selectivity"

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

### Result Size Estimation

- **Result cardinality** = Max # tuples * product of all RF's.
- Term col=value (given index I on col )
  - $RF = \frac{1}{NKeys(I)}$
- Term col1=col2 (This is handy for joins too…)
  - $RF = \frac{1}{\text{MAX}(NKeys(I1), NKeys(I2))}$
- Term col>value
  - $RF = \frac{\text{High}(I)-\text{value}}{\text{High}(I)-\text{Low}(I)}$

(Implicit assumptions: values are uniformly distributed and terms are independent!)
- Note, if missing indexes, assume $1/10$!

### Cost Estimates for Single-Relation Plans

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

*Recall*: Must also charge for duplicate elimination if required

### Queries Over Multiple Relations

- A heuristic 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 Nth 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.

The Dynamic Programming Table

<table>
<thead>
<tr>
<th>Subset of tables in FROM clause</th>
<th>Interesting-order columns</th>
<th>Best plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨R, S⟩</td>
<td>&lt;none&gt;</td>
<td>hashjoin(R, S)</td>
<td>1000</td>
</tr>
<tr>
<td>⟨R, S⟩</td>
<td>&lt;R.a, S.b&gt;</td>
<td>sortmerge(R, S)</td>
<td>1500</td>
</tr>
</tbody>
</table>

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-added (downstream) joins

Enumeration of 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 pruning plan space, this approach is still exponential in the # of tables.
- Recall that in practice, COST considered is #IOs + factor * CPU Inst
Physical Database Design and Tuning

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions?
  How selective are these conditions likely to be?

- For each update in the workload:
  - Which attributes are involved in selection/join conditions?
  How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Creating an ISUD Chart

Insert, Select, Update, Delete Frequencies

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Frequency</th>
<th>% Table</th>
<th>Employee Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payroll Run</td>
<td>monthly</td>
<td>100</td>
<td>S S S</td>
</tr>
<tr>
<td>Add Emps</td>
<td>daily</td>
<td>0.1</td>
<td>I I I</td>
</tr>
<tr>
<td>Delete Emps</td>
<td>daily</td>
<td>0.1</td>
<td>O O O</td>
</tr>
<tr>
<td>Give Raises</td>
<td>monthly</td>
<td>10</td>
<td>S U</td>
</tr>
</tbody>
</table>

Index Selection

- One approach:
  - Consider most important queries in turn.
  - Consider best plan using the current indexes, and see if better plan is possible with an additional index.
  - If so, create it.

- Before creating an index, must also consider the impact on updates in the workload:
  - Trade-off: indexes can make queries go faster, updates slower. Require disk space, too.
Issues to Consider in Index Selection

- Attributes mentioned in a WHERE clause are candidates for index search keys.
  - Range conditions are sensitive to clustering
  - Exact match conditions don’t require clustering
    • Or do they?!? :)”
- Try to choose indexes that benefit as many queries as possible.
- NOTE: only one index can be clustered per relation!
  - So choose it based on important queries that benefit the most from clustering!!

Issues in Index Selection (Contd.)

- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - If range selections are involved, order of attributes should be carefully chosen to match the range ordering.
  - Such indexes can sometimes enable index-only strategies for important queries.
    • For index-only strategies, clustering is not important!
- When considering a join condition:
  - B+-tree on inner is very good for Index Nested Loops.
    • Should be clustered if join column is not key for inner, and inner tuples need to be retrieved.
  - Clustered B+ tree on join column(s) good for Sort-Merge.

Clustering and Joins

SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname='Toy' AND E.dno=D.dno

• Clustering is especially important when accessing inner tuples in INL.
  - Should make index on E.dno clustered.
• Suppose that the WHERE clause is instead:
  WHERE E.hobby='Stamps' AND E.dno=D.dno
  - If many employees collect stamps, Sort-Merge join may be worth considering. A clustered index on D.dno would help.
• Summary: Clustering is useful whenever many tuples are to be retrieved.

Multi-Attribute Index Keys

- To retrieve Emp records with age=30 AND sal=4000, an index on <age,sal> would be better than an index on age or an index on sal.
  - Such indexes also called composite or concatenated indexes.
  - Choice of index key orthogonal to clustering etc.
• If condition is: 20<age<30 AND 3000<sal<5000:
  - Clustered tree index on <age,sal> or <sal,age> is best.
• If condition is: age=30 AND 3000<sal<5000:
  - Clustered <age,sal> index much better than <sal,age> index!
• Composite indexes are larger, updated more often.
Index-Only Plans

• A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

SELECT D.mgr FROM Dept D, Emp E WHERE D.dno=E.dno
SELECT D.mgr, E.eid FROM Dept D, Emp E WHERE D.dno=E.dno
SELECT E.dno, COUNT(*) FROM Emp E GROUP BY E.dno
SELECT E.dno, MIN(E.sal) FROM Emp E GROUP BY E.dno
SELECT AVG(E.sal) FROM Emp E WHERE E.age=25 AND E.sal BETWEEN 3000 AND 5000

Horizontal Decompositions (Contd.)

• Contracts (Cid, Sid, Jid, Did, Pid, Qty, Val)
  • Suppose that contracts with value > 10000 are subject to different rules.
    – So queries on Contracts will often say WHERE val>10000.
  • One approach: clustered B+ tree index on the val field.
  • Second approach: replace contracts by two new relations, LargeContracts and SmallContracts, with the same attributes (CSJPQV).
    – Performs like index on such queries, but no index overhead.
    – Can build clustered indexes on other attributes, in addition!

Masking Conceptual Schema Changes

CREATE VIEW Contracts(cid, sid, jid, did, pid, qty, val) AS SELECT * FROM LargeContracts UNION SELECT * FROM SmallContracts

• Horizontal Decomposition from above
• Masked by a view.
  – NOTE: queries with condition val>10000 must be asked wrt LargeContracts for efficiency: so some users may have to be aware of change.
    • i.e. the users who were having performance problems
    • Arguably that’s OK – they wanted a solution!

More Guidelines for Query Tuning

• Minimize the use of DISTINCT: don’t need it if duplicates are acceptable, or if answer contains a key.
• Minimize the use of GROUP BY and HAVING:

SELECT MIN (E.age) FROM Employee E GROUP BY E.dno HAVING E.dno=102

• Consider DBMS use of index when writing arithmetic expressions: E.age=2*D.age will benefit from index on E.age, but might not benefit from index on D.age!
Guidelines for Query Tuning (Contd.)

- **Avoid using intermediate relations:**

<table>
<thead>
<tr>
<th>SQL</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT E.dno, AVG(E.sal) FROM Emp E, Dept D WHERE E.dno=D.dno AND D.mgrname='Joe' GROUP BY E.dno</td>
<td>SELECT T.dno, AVG(T.sal) FROM Temp T GROUP BY T.dno</td>
</tr>
</tbody>
</table>

  - Does not materialize the intermediate reln Temp.
  - If there is a dense B+ tree index on `<dno, sal>`, an index-only plan can be used to avoid retrieving Emp tuples in the second query! 