TUTORIAL ON QUERY OPTIMIZATION
DB Logical Architecture

- **Query Execution engine**
  - Access Plan Executor
  - Parser
  - Operator Evaluator
  - Optimizer

- **Concurrency control**
  - Transaction Manager
  - Lock Manager

- **Access Methods**
- **Buffer Manager**
- **Disk Manager**
- **Recovery Manager**
Relational Operators

Selection

Projection

Join

Union

Set-difference

Table A

Table B

Table X

Table Y

Table S

Table T
Measures of Query Cost

- **Cost** is generally measured as **total elapsed time for answering query**
  - Many **factors** contribute to time cost: disk accesses, CPU, or even network communication

- Typically, **disk access is the predominant cost**, and is also relatively easy to estimate
  - Measured by taking into account
    - Number of blocks read  * average-block-read-cost
    - Number of blocks written  * average-block-write-cost
  - **Cost to write a block is greater than cost to read a block**
    - Data is read back after being written to ensure that the write was successful
Measures of Query Cost

- For simplicity we just use number of block transfers from disk as the cost measure
  - We ignore the difference in cost between sequential and random I/O for simplicity
  - We also ignore CPU costs for simplicity
  - We do not include cost to writing output to disk in our cost formulae

- Costs depends on the size of the buffer in main memory
  - Having more memory reduces need for disk access
  - Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine ahead of actual execution
  - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

- Real systems take CPU cost into account, differentiate between sequential and random I/O, and take buffer size into account
Nested-Loop Join

- **Read in outer relation R block by block**
  - Then, for each tuples in R, we scan the entire inner relation S (scan means read in S block by block)

- $n_R$: no. of record for R

- $b_R$: no. of block for R

- **Worst Cost**: $b_R + n_R \times b_S$

- **Best Cost**: $b_R + b_S$ (if smaller relation can fit in memory)

- Use small relation as outer relation

- **Buffer**: 3 pages (1 for R, 1 for S, 1 for output)
Nested Loops Join
Exercise

- Relations: R1(A,B,C) and R2(C,D,E)
- R1 has 20,000 tuples
- R2 has 45,000 tuples
- 25 tuples of R1 fit on one block (blocking factor)
- 30 tuples of R2 fit on one block
- R1 JOIN R2
- R1 need 800 blocks (20000/25)
- R2 need 1500 blocks (45000/30)
- Assume M pages in memory
- If M > 800, cost = \( b_R + b_S = 1500 + 800 = 2300 \) I/Os

- Consider only M <=800,
- Using R1 as outer relation
  - Cost: \( 20000 \times 1500 + 800 = 30000800 \) I/Os
- If R2 as outer relation
  - Cost: \( 45000 \times 800 + 1500 = 36001500 \) I/Os
Block Nested Loop Join

- Cost: $b_R + b_R \times b_S$

- If $M$ buffer pages available
  - Cost: $b_R + \lceil b_R / (M-2) \rceil \times b_S$
  - $M$ buffer pages (1 for inner S, 1 for output and all remaining $M-2$ pages to hold “block” of outer $R$

- If $R1$ is outer
  - Cost = $\lceil 800 / (M-2) \rceil \times 1500 + 800$ I/Os

- If $R2$ is outer
  - Cost = $\lceil 1500 / (M-2) \rceil \times 800 + 1500$ I/Os
Index Nested-Loop Join

Table R

DB Server

Table S

key
Index Nested-Loop Join

- Primary B+tree index on the join attribute of R2:
  \[ b_{R1} + n_{R1} \cdot (x_{R2} + 1) \]
  where:
  - \( n_{R1} (n_{R2}) \) is the number of \( R_1 \) (\( R_2 \)) tuples
  - \( x_{R2} \) is the height of the B+-tree index on the join attribute
  - \( n_{R1} \cdot (x_{R2} + 1) \) is the cost of using B+-tree index to find matching tuple in \( R_2 \)

- Secondary B+tree index on the join attribute of R2:
  \[ b_{R1} + n_{R2} \cdot (x_{R2} + 1) \]
  where \( n_{R2} \cdot (x_{R2} + 1) \) is the cost of using B+-tree index to find matching tuple in \( R_2 \)
Index Nested-Loop Join

- Hash index on the join attribute of R2:
  - $b_{R1} + n_{R1} \times H$
  - Where $H$ is the average number of page accesses necessary to retrieve a tuple from R2 with a given key

- We use:
  - $H = 1.2$ for a primary hash index and
  - $H = 2.2$ for a secondary hash index
External Sorting

- File has $b_R$ pages

- Buffer $M$ : number of main memory page buffers

- No. of runs in the first pass $R = b_R / M$

- No. of passes to sort file completely
  \[ P = \lceil \log_{M-1} \left( \frac{b_R}{M} \right) \rceil + 1 \]
  \[ P = \lceil \log_{M-1} R \rceil + 1 \]

- Total cost for sorting
  \[ = b_R \times (2 \times \lceil \log_{M-1} R \rceil + 1) \]
  \[ = b_R \times 2 \times \lceil \log_{M-1} R \rceil + b_R \]
Merge Join

- Assuming R1 and R2 are not initially sorted on the join key

- Cost = \( \text{Sorting} + b_R + b_s \)

- \( \text{Sorting} = 1500 \times (2 \times \lceil \log_{M-1} (1500/M) \rceil + 1) + 800 \times (2 \times \lceil \log_{M-1} (800/M) \rceil + 1) \)

- Assuming all tuples with same values for join attributes fit in memory (each block needs to be read only once)
Merge Join

- Assuming that there is a secondary B+tree on Rx

- Cost = CR1 + CR2

- Where \( CR_x = \left( \frac{n_{Rx} \cdot ps}{0.69 \cdot bs} \right) + b_{Rx} \) for the R which has the index on the join attribute
  - \( ps \): the size of the tuple reference (tuple identifier, rid)
  - \( bs \): the size of the block
Hash join

- Hash both relations on the join attribute using the same hash function

- Since R1 is smaller, we use it as the build relation and R2 as probe relation

- Assume no overflow occurs

- If \( M \geq \frac{800}{M} \), no need for recursive partitioning, cost = \( 3(1500 + 800) = 6900 \) disk access = \( 3(b_R + b_s) \)

- Else, cost = \( 2(1500 + 800) \left[ \log_{M-1} (800) - 1 \right] + 1500 + 800 \) disk access = \( 2(b_R + b_s) \left[ \log_{M-1} (b_s) - 1 \right] + b_R + b_s \)
Optimizer Architecture

Rewriter

Algebraic Space

Cost Model

Method-Structure Space

Planner

Size-Distribution Estimator
Optimizer Architecture

- **Rewriter**: Finds equivalent queries that, perhaps can be computed more efficiently; all such queries are passed on to the Planner
  - Examples of Equivalent queries: Join orderings

- **Planner**: Examines all possible execution plans and chooses the cheapest one, i.e., fastest one
  - Uses other modules to find best plan

- **Algebraic Space**: Determines which types of queries will be examined
  - Example: Try to avoid Cartesian Products

- **Method-Structure Space**: Determines what types of indexes are available and what types of algorithms for algebraic operations can be used
  - Example: Which types of join algorithms can be used

- **Cost Model**: Estimates the cost of execution plans
  - Uses Size-Distribution Estimator for this

- **Size-Distribution Estimator**: Estimates size of tables, intermediate results, frequency distribution of attributes and size of indexes
Algebraic Space

- We consider queries that consist of `select`, `project` and `join` (Cartesian product is a special case of join)
- Such queries can be represented by a tree.
- Example:
  - `emp(name, age, sal, dno)`
  - `dept(dno, dname, floor, mgr, ano)`
  - `act(ano, type, balance, bno)`
  - `bank(bno, bname, address)`
  
  ```
  select name, floor
  from emp, dept
  where emp.dno=dept.dno and sal > 100K
  ```
Restriction 1 of Algebraic Space

- Algebraic space may contain many equivalent queries

- Important to restrict space

- **Restriction (heuristic) 1:** Only allow queries for which selection and projection:
  - are processed as early as possible
  - are processed on the fly
Performing Selection and Projection "On the Fly"

- Selection and projection are performed as part of other actions.

- Projection and selection that appear one after another are performed one immediately after another.
  - Projection and Selection do not require writing to the disk.

- Selection is performed while reading relations for the first time.

- Projection is performed while computing answers from previous action.
Processing Selection/Projection as Early as Possible

- The three trees differ in the way that selection and projection are performed.

- In T3, there is "maximal pushing of selection and projection"
  - Rewriter finds such expressions.
Restriction 2 of Algebraic Space

- Since the order of selection and projection is determined, we can write trees only with joins
- Restriction (heuristic) 2: Cross products are never formed, unless the query asks for them
- Example:

```sql
select name, floor, balance
from emp, dept, acnt
where emp.dno=dept.dno and
department.ano = acnt.ano
```
3 Trees

Which trees have cross products?
Restriction 3 of Algebraic Space

- The left relation is called the outer relation in a join and the right relation is the inner relation (as in terminology of nested loops algorithms)

- **Restriction (heuristic) 3:** The inner operand of each join is a database relation, not an intermediate result (left-deep plans)

- Example:

```sql
select name, floor, balance
from emp, dept, acnt, bank
where emp.dno=dept.dno and dept.ano=acnt.ano
     and acnt.bno = bank.bno
```
Which trees follow restriction 3?
Pipelining Joins

- Consider computing: \((\text{Emp} \bowtie \text{Dept}) \bowtie \text{Acnt}\). In principle, we should
  - compute \((\text{Emp} \bowtie \text{Dept})\), write the result to the disk
  - then read it from the disk to join it with \text{Acnt}

- When using block and index nested loops join, we can avoid the step of writing to the disk

- We allow plans that
  - Perform selection and projection early and on the fly
  - Do not create cross products
  - Use database relations as inner relations (also called left–deep trees)
Pipelining Joins - Example

1. Read block from Emp
2. Find matching Dept tuples using index
3. Find matching Acnt tuples using index
4. Write final output

Buffer
Dynamic programming algorithm to find best plan for performing join of \( N \) relations

**Intuition:**
- Find all ways to access a single relation
  - Estimate costs and choose best access plan(s)
- For each pair of relations, consider all ways to compute joins using all access plans from previous step
  - Choose best plan(s)...
- For each \( i-1 \) relations joined, find best option to extend to \( i \) relations being joined...
- Given all plans to compute join of \( n \) relations, output the best
Example

We want to compute the query:

```sql
select name, mgr
from emp, dept
where emp.dno=dept.dno and sal>30K and floor = 2
```

Available Indexes: B+tree index on emp.sal, B+tree index on emp.dno, hashing index on dept.floor

Join Methods: Block nested loops, index nested loops and sort-merge

In the example, all cost estimations are fictional
## Step 1 – Accessing Single Relations

<table>
<thead>
<tr>
<th>Relation</th>
<th>Interesting Order</th>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp</td>
<td>emp.dno</td>
<td>Access through B+tree on emp.dno</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access through B+tree on emp.sal</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequential scan</td>
<td>600</td>
</tr>
<tr>
<td>dept</td>
<td></td>
<td>Access through hashing on dept.floor</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequential scan</td>
<td>200</td>
</tr>
</tbody>
</table>
### Step 2 – Joining 2 Relations

<table>
<thead>
<tr>
<th>Join Method</th>
<th>Outer/Inner</th>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested loops</td>
<td>empt/dept</td>
<td>● For each emp tuple obtained through B+Tree on emp.sal, scan dept through hashing index on dept.floor to find tuples matching on dno</td>
<td>1800</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● For each emp tuple obtained through B+Tree on emp.dno and satisfying selection, scan dept through hashing index on dept.floor to find tuples matching on dno</td>
<td>3000</td>
</tr>
</tbody>
</table>
**Step 2 – Joining 2 Relations**

<table>
<thead>
<tr>
<th>Join Method</th>
<th>Outer/Inner</th>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested loops</td>
<td>dept/emp</td>
<td>● For each dept tuple obtained through hashing index on dept.floor, scan emp through B+Tree on emp.sal to find tuples matching on dno</td>
<td>2500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● For each dept tuple obtained through hashing index on dept.floor, scan emp through B+Tree on emp.dno to find tuples satisfying the selection on emp.sal</td>
<td>1500</td>
</tr>
</tbody>
</table>
## Step 2 – Joining 2 Relations

<table>
<thead>
<tr>
<th>Join Method</th>
<th>Outer/Inner</th>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
</table>
| sort merge  |             | ●Sort the emp tuples resulting from accessing the B+Tree on emp.sal into L1  
  ●Sort the dept tuples resulting from accessing the hashing index on dept.floor into L2  
  ●Merge L1 and L2 | 2300 |
|             |             | ●Sort the dept tuples resulting from accessing the hashing index on dept.floor into L2  
  ●Merge L2 and the emp tuples resulting from accessing the B+Tree on emp.dno and satisfying the selection on emp.sal | 2000 |
Suppose we want to find the natural join of: Reserves, Sailors, Boats

The 2 options that appear the best are (ignoring the order within a single join):

(Sailors▷◁ Reserves) ▷◁ Boats
Sailors▷◁(Reserves ▷◁ Boats)

We would like intermediate results to be as small as possible
Analyzing Result Sizes

- In order to answer the question in the previous slide, we must be able to estimate the size of (Sailors $\bowtie$ Reserves) and (Reserves $\bowtie$ Boats)

- The DBMS stores statistics about the relations and indexes
  - **Cardinality**: Num of tuples $N_{Tuples}(R)$ in each relation $R$
  - **Size**: Num of pages $N_{Pages}(R)$ in each relation $R$
  - **Index Cardinality**: Num of distinct key values $N_{Keys}(I)$ for each index $I$
  - **Index Size**: Num of pages $I_{Pages}(I)$ in each index $I$
  - **Index Height**: Num of non-leaf levels $I_{Height}(I)$ in each B+ Tree index $I$
  - **Index Range**: The minimum $I_{Low}(I)$ and maximum value $I_{High}(I)$ for each index $I$

- They are updated periodically (*not* every time the underlying relations are modified)
Example

SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid and S.rating > 3
    and R.agent = 'Joe'

- Cardinality(R) = 1,000 * 100 = 100,000
- Cardinality(S) = 500 * 80 = 40,000
- NKeys(Index on S.sid) = 40,000
- NKeys(Index on R.agent) = 100
- High(Index on Rating) = 10, Low = 0
- Maximum cardinality: 100,000 * 40,000
- Reduction factor of R.sid = S.sid: 1/40,000
- Reduction factor of S.rating > 3: (10–3)/(10-0) = 7/10
- Reduction factor of R.agent = 'Joe': 1/100
- Total Estimated size: (Maximum cardinality) * (Reduction factor of R.sid) * 
  (Reduction factor of S.rating) * (Reduction factor of R.agent = S.sid) = 100,000 * 
  40,000 * (1/40,000) * (7/10) * (1/100) = 700
Consider the join of the four relations named R, S, T, U:

- R(a,b), 800 total tuples
- S(b,c), 10,000 total tuples
- T(c,d), 4,000 total tuples
- U(a,d), 2,500 total tuples

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>S(b,c)</th>
<th>T(c,d)</th>
<th>U(a,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V(R,a) = 500</td>
<td>V(S,b) = 2,000</td>
<td>V(T,c) = 1,000</td>
<td>V(U,a) = 10</td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,b) = 2,000</td>
<td>V(T,c) = 1,000</td>
<td>V(U,b) = 50</td>
</tr>
<tr>
<td>V(S,c) = 1,000</td>
<td>V(T,c) = 1,000</td>
<td>V(T,d) = 1,000</td>
<td>V(U,d) = 500</td>
</tr>
</tbody>
</table>

- V(R,a) : # of distinct values for attribute
Notes

- $V(R, a)$: number of distinct values for attribute

- Cost $\{R, S\} = \frac{\text{size of } R \times \text{size of } S}{\max (V(R, _), V(S, _))}$, where $_$ is the join attribute

- Cost $\{R, S, U\} = \frac{\text{size of } R \times \text{size of } S \times \text{size of } U}{\text{2 greater nums from } (V(R, _), V(S, _), V(U, _))}$, where $_$ is the join attribute
For the singleton sets, the costs and best plans are given in the table below:

<table>
<thead>
<tr>
<th></th>
<th>{R}</th>
<th>{S}</th>
<th>{T}</th>
<th>{U}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>800</td>
<td>10.000</td>
<td>4.000</td>
<td>2.500</td>
</tr>
<tr>
<td>Cost</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Best plan</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>

As the costs for all relations are the same, the dynamic programming algorithm will consider them all. The greedy algorithm however, must choose one, or also consider them all. Let’s assume it takes the one with least cost, and if they are more present of these, take the one with smallest length. So, the plan called “R” is chosen.
Now, we consider the pairs of relations

Again, the cost is 0 for each, as we do not have intermediate results

<table>
<thead>
<tr>
<th></th>
<th>{R,S}</th>
<th>{R,T}</th>
<th>{R,U}</th>
<th>{S,T}</th>
<th>{S,U}</th>
<th>{T,U}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>4.000</td>
<td>3.200.000</td>
<td>4.000</td>
<td>40.000</td>
<td>12.500</td>
<td>10.000</td>
</tr>
<tr>
<td>Cost</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Best plan</td>
<td>RxS</td>
<td>RxT</td>
<td>RxU</td>
<td>SxT</td>
<td>SxU</td>
<td>TxU</td>
</tr>
</tbody>
</table>

The dynamic programming algorithm again keeps them all for the next run, as the costs are 0. The greedy algorithm had chosen R in the previous run, so it must choose a plan based on this choice. Since there are two best plans here, it has to make a choice of them. Let’s assume it makes the wrong choice, and takes “RxU”.
Now, we consider the join of three out of these four relations:

As you can see, the best plan is clearly \((RxS)xU\), with the least cost and size. However, the greedy algorithm will not consider this plan, as it chose \(RxU\) as the best plan in previous run. Instead, it is forced to take \((RxU)xT\) as best plan now.
Finally, we consider the join of all relations. We come to these four final results (for dynamic programming):

<table>
<thead>
<tr>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>((RxS)xT)xU</td>
<td>20.000</td>
</tr>
<tr>
<td>((RxS)xU)xT</td>
<td>9.000</td>
</tr>
<tr>
<td>((RxU)xT)xS</td>
<td>20.000</td>
</tr>
<tr>
<td>((TxU)xS)xR</td>
<td>60.000</td>
</tr>
</tbody>
</table>

The dynamic programming algorithm finds the optimal solution 
((RxS)xU)xT, while the greedy algorithm, based on results of the previous run, comes to the more expensive solution 
((RxU)xT)xS.
Selecting Algorithms for Plan Operators

- For **selection operator**, consider
  - Index-scan algorithms that use single attribute indexes, multiple indexes, or multidimensional indexes
  - Table-scan algorithm using no index

- For **join operator**, consider
  - All types of join algorithms if enough statistics is available
  - If statistics is in sufficient, follow some simple ideas
    - Try one-pass algorithm or nested-loops
    - Use sort-join if one or both arguments are already sorted
    - If index is available, use index-join
    - If sort and index are not available and multi-pass join is necessary, use a hash join
Pipelining Example

- Relations:
  - \( R(W, X), b_R = 5000 \)
  - \( S(X, Y), b_S = 10000 \)
  - \( U(Y, Z), b_U = 10000 \)

- Buffer: \( M = 101 \) blocks

- Both joins are hash join

- Size \( k \) is estimated, and used to choose join algorithms
Case 1: \( k \leq 49 \)

- Can pipeline result of 1st join into 2nd join

- Two-pass hash join for \( R \bowtie S \):
  - Both \( R \) and \( S \) are hashed into 100 partitions, where each \( R \) partition has 50 blocks
  - Join corresponding \( R \) & \( S \) partitions using 50 buffer blocks for \( R \) partition, 1 block for \( S \) partition, and store the result in 49 blocks as a hash table

- One-pass hash join for the 2nd join:
  - Use 1 buffer block for \( U \) (no need to partition \( U \)), join with the intermediate result that is already in buffer

- Cost = \( 3(5000+10000) + 10000 = 55000 \)
Case 2: $49 < k \leq 5000$

- Overlap the 1st join with the hash partitioning of the 2nd join

- Two-pass hash join for the 1st join:
  - Partition R & S into 100 partitions, so that each R partition contains 50 blocks
  - Join corresponding R & S partitions (using 51 buffer blocks)
  - During the join, hash the result into 50 partitions (using the remaining 50 buffer blocks) & write the partitions to disk

- Two-pass hash join for the 2nd join:
  - Partition U into 50 partitions
  - Join corresponding partitions of intermediate result & U, using intermediate result partition as build relation (use 1 to 100 buffer blocks)

- Cost = $3(10000+5000) + k + 2(10000) + (k+10000) = 75000 + 2k$
Case 3: $k > 5000$

- Cannot use pipelining

- Two-pass hash join for the 1st join:
  - Partition $R$ & $S$ into 51 partitions, so that each $R$ partition has <100 blocks
  - Join corresponding $R$ & $S$ partitions, write results to disk

- Two-pass hash join for the 2nd join:
  - Partition intermediate result & $U$ into more than 50 partitions
  - Join corresponding partitions of $U$ & intermediate result, using the smaller partition as the build relation

- Cost = $3(5000+10000) + k + 3(10000+k) = 75000 + 4k$
Pipelining vs. Materialization

- **Pipelining**: Apply next operator to the output of one stage, as the output is generated.

- **Materialization**: Create a temporary relation as the output of a stage, pass to next stage.
Pipelining vs. Materialization

- **Advantages** of 64 bit processors
  - More main memory possible (more pipelining operations possible)
  - Complex in-memory processing does not require intermediate results being temporarily written to disk

- **Disadvantages** of 64 bit processors
  - Application must be fully supporting 64 bit to make full use of the speed advantages

- DBMS's implementing 64 bit are e.g. Oracle
Ordering of Physical Operations

- Pre-order traversal
- Post-order traversal
Points to Remember

● Step 1: Choose a logical plan
  ◦ Involves choosing a query tree, which indicates the order in which algebraic operations are applied
  ◦ Heuristic: Pushed trees are good, but sometimes “nearly fully pushed” trees are better due to indexing (as we saw in the example)
  ◦ So: Take the initial “master plan” tree and produce a fully pushed tree plus several nearly fully pushed trees

● Step 2: Reduce search space
  ◦ Deal with associativity of binary operators (join, union, …)
  ◦ Choose a particular shape of a tree (left-deep trees)
    • Equals the number of ways to parenthesize N-way join – grows very rapidly
  ◦ Choose a particular permutation of the leaves

● Step 3: Use a heuristic search to further reduce complexity
  ◦ The choice of left-deep trees still leaves open too many options
  ◦ A heuristic algorithm is used to get a ‘good’ plan
Examples
**Query Execution**

A relational database system holds three relations; $C$ (corporations), $E$ (executives) and $S$ (stock sales) with the following characteristics:

- **Relation $C$ (corporations):**
  - Tuples are stored as fixed length, fixed format records, length 6000 bytes.
  - There are 20,000 $C$ tuples.
  - Tuples contain key attribute $C.N$ (corporation number), length 20 bytes; other fields and record header make up rest.
  - There is an index on attribute $C.N$.

- **Relation $E$ (executives):**
  - Tuples are stored as fixed length, fixed format records, length 2000 bytes.
  - There are 40,000 $E$ tuples.
  - Tuples contain attribute $E.N$ (the corporation that the executive works for), length 20 bytes; other fields and record header make up rest.
  - Tuples also contain attribute $E.I$ (executive identifier), length 20 bytes.
  - There is an index on attribute $E.N$.

- **Relation $S$ (stock sales):**
  - Tuples are stored as fixed length, fixed format records, length 1000 bytes.
  - There are 120,000 $S$ tuples.
  - Tuples contain attribute $S.I$ (the identifier of the executive involved), length 20 bytes; other fields and record header make up rest.
  - There is an index on attribute $S.I$.

While the number of executives associated with each corporation varies, for evaluation purposes we may assume that each corporation has 2 executives, and each executive has 3 stock sales records associated with him or her.

The records are to be placed in a collection of 16 Kilobytes (16384 bytes) disk blocks that have been reserved to exclusively hold $C$, $E$, or $S$ records, or combinations of those records. (That is, there are no other types of records in the blocks we are discussing in this problem.) Each block uses 50 bytes for its header; records are not spanned.)

Three disk organization strategies are being considered:

1. **Sequential:** All the corporation ($C$) records are placed sequentially (ordered by corporation number) in one subset of blocks. Executive ($E$) records are separate in another set of blocks, executives are not ordered by corporation number. Finally, stock sales ($S$) records are in a third set of blocks, not ordered by executive identifier.

2. **Clustered:** For each corporation ($C$) record, the 2 executives for that corporation ($C.N = E.N$) reside in the same block. Similarly, the 6 stock sale records for those corporations are in the same block. The corporation records are not sequenced in any way.

3. **Random:** The records are placed as follows, without regard to $C.N$, $E.N$, $E.I$, $S.I$ values, as follows: Each block contains one random $C$ record, 2 random $E$ records, and 6 random $S$ records.

We are also told there are four types of queries that constitute the vast majority of the workload:

1. **Probe:** Given a corporation number, get the corporation record.
2. **Ordered scan:** For all corporations, in increasing corporation number order, get each corporation
3. Plain scan: For all corporations, in any order, get all corporations records.
4. Join: For a given corporation number \( C.N \), get the corporation record followed by all its stock sales records. (That is, get all stock sales records with \( S.I = E.I \), for any account with \( E.N = C.N \))

A) For each of the storage strategies, compute how many total disk blocks are needed for holding relations \( C \), \( E \) and \( S \). Briefly explain your answers.

\textit{Answer}

Sequential:
\( C \) records:
In one block we can fit
\( \leq \frac{(16384 - 50)}{6000} f = 2 \) records. The number of blocks required is
\( \frac{20000}{2} = 10000 \) blocks.
\( E \) records:
In one block we can fit
\( \leq \frac{(16384 - 50)}{2000} f = 8 \) records. The number of blocks required is
\( \frac{40000}{8} = 5000 \) blocks.
\( S \) records:
In one block we can fit
\( \leq \frac{(16384 - 50)}{1000} f = 16 \) records. The number of blocks required is
\( \frac{120000}{16} = 7500 \) blocks.

A total \( 10000 + 5000 + 7500 = 22500 \) blocks are required.

Clustered: The size of 1 \( C \), 2 \( E \) and 6 \( S \) records is 6000 +
\( 2 \times 2000 + 6 \times 1000 = 16000 \) bytes.
\( \leq \frac{(16384 - 50)}{16000} f = 1 \) set of records fit in one block.
Hence we will need 20000 blocks to store all the records in the three tables.

Random: Even though the distribution of the records among the blocks might be different, this organization also stores only 1 \( C \), 2 \( E \) and 6 \( S \) records – same as the clustered organization. Hence, the number of blocks required is once again 20000.

B) For each query type and for each storage strategy, estimate the number of disk blocks that must be transferred from disk to execute the query. Briefly explain each answer.
Assume you only have one buffer page (16 KB) in memory; thus, each time you need a block it counts as one IO (unless the request is for exactly the same block you already have in the buffer). You also have enough memory to hold a single working copy of a \( C \) record, of an \( E \) record, and of an \( S \) record.
So, for example, to do a $C$, $E$ join, you first read in a block containing a $C$ record, and copy the record to the working $C$ area. Then you read in the block containing an $E$ record into your 16 KB buffer, and join the two records. Also, ignore any IOs due to index accesses. That is, you may assume the indexes reside in memory.

Hint: Do not concern yourself with events that are very unlikely. For example, say you want to get the three executive records for a particular corporation, and that these records are randomly placed on a large number of disk blocks. There is some chance that the executive records you want happen to be in the same block (just by luck). However, for this problem you may assume this probability is negligible and it will take you exactly three IOs to get those three records.

**Answer**

1. **Probe:** In all the three organizations, only 1 I/O will be required because of an index on the search key $C.N$.
2. **Ordered Scan:** In all three organizations, the index on $C.N$ can be used to sequentially scan the entire corporations’ relation reading in only those blocks that contain one or more corporation records.  
   Sequential: 10,000 I/Os because there are 10000 blocks storing corporation records.  
   Clustered: 20,000 I/Os because there are 10000 blocks storing corporation records.  
   Random: 20,000 I/Os because there are 10000 blocks storing corporation records.  
3. **Plain Scan** As in the ordered scan, the index on $C.N$ can be used to read in the corporation records.  
   Since the index will be sorted on the key value (namely $C.N$), the corporation records will be retrieved in sorted order. Therefore the number of I/Os for this kind of query is the same as for the ordered scan.  
4. **Join:**  
   The clustered organization is most efficient in performing this operation since a corporation record and all it’s associated executives and stock sales are stored together in a single disk block. So, given a corporation number, the index $C.N$ can be used to retrieve the relevant corporation block and generate the tuples for the join. 1 I/O

In the case of both the sequential and random organizations, the steps to compute the join are as follows:  
1. Use the index on $C.N$ to retrieve the block containing the relevant corporation record. 1 I/O  
2. Move the corporation record to the working area.  
3. Use the index on $E.N$ to retrieve the block containing one of the two executive records associated with that particular corporation. 1 I/O  
4. Move the retrieved executive record into the working area.  
5. Using the executive identifier ($E.I$) in that executive record and the index on $S.I$, retrieve a block containing one of the 3 stock sales records associated with that executive. 1 I/O  
6. Join the corporation record in the working area with this stock sale record and return one of the tuples of the join.  
7. Repeat steps 5 and 6 two more times to retrieve the remaining two stock sales records associated with that executive. Generate two more tuples of the result. 2 I/O  
8. Repeat steps 3-7 once more using the second executive record to generate the remaining tuples of the result. 4 I/O  
   This gives a total of 9 I/Os.
Άσκηση - Query Optimization
Consider the following schema
Sailors(sid, sname, rating, age)
Reserves(sid, did, day)
Boats(bid, bname, size)
Reserves.sid is a foreign key to Sailors and Reserves.bid is a foreign key to Boats.bid.
We are given the following information about the database:
• Reserves contains 10,000 records with 40 records per page.
• Sailors contains 1000 records with 20 records per page.
• Boats contains 100 records with 10 records per page.
• There are 50 values for Reserves.bid.
• There are 10 values for Sailors.rating (1...10)
• There are 10 values for Boat.size
• There are 500 values for Reserves.day
Consider the following query
SELECT S.sid, S.sname, B.bname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND
B.size > 5 AND R.day = 'July 4, 2003';

(a) Assuming uniform distribution of values and column independence, estimate the number of tuples returned by this query.

Answer
The maximum cardinality this query is |R| × |S| × |B| = 109.
Reduction Factors:
1/2 for B.size > 5,
1/500 for R.day = ‘July 4, 2003’,
1/100 for R.bid = B.bid, and
1/1000 for S.sid = R/sid
So number of tuples return is (1/2)(1/500)(1/100)(1/1000)(109) = 10

(b) Consider the following query SELECT
S.sid, S.sname, B.bname FROM Sailors S, Reserves R, Boats B WHERE S.sid =

Draw all possible left-deep query plans for this query:

Answer
Note that we cannot join S and B since that would result in a cross product.

(c) For the first join in each query plan (the one at the bottom of the tree, what join algorithm would work best? Assume that you have 50 pages of memory. There are no indexes, so indexed nested loops is not an option.

*Answer*

| |R| =250, |S| = 50, |B| = 10.

R join S:
• Page-oriented nested loops join: |R| + |R| × |S| = 250 + 250 × 50 = 12570
• Block nested loops join: |R| + |R|/49 × |S| = 250 + 6 × 50 = 550
• Sort-merge join: note that 250 < (num buffers)^2 so cost is 3(|R| + |S|) = 3(250+50) = 900
• Hash join: we can use hash join since each of the R partitions from phase one fit into memory. So cost is 3(|R| + |S|) = 3(250+50) = 900.
• **Cheapest: Block nested loops**

S join R:
• Page-oriented nest loops join: |S| + |S| × |R| = 50 + 50 × 250 = 12550
• Block nested loops join: |S| + |S|/49 × |R| = 50 + 2 × 250 = 550
• Sort-merge join: 3(|R| + |S|) = 3(250+50) = 900
• Hash join: 3(|R| + |S|) = 3(250+50) = 900
• **Cheapest: Block nested loops**

B join R:
• Page-oriented nested loops: |B| + |B| × |R| = 10 + 10 × 250 = 2510
• Block nested loops join: |B| + |B|/49 × |R| = 10 + 250 = 260
• Sort-merge join: 3(10+250) = 780
• Hash join: 3(10+250) = 780
• **Cheapest: Block nested loops**

R join B:
• Page-oriented nested loops join: The thing to note is that we can fit the entire Boats relation into memory, so we only need to scan Boats once. So the cost is |R| + |B| = 260. This is the same if we use simple nested loops join.
• Block nested loops join: |R| + |R|/49 × |B| = 250 + 6 × 10 = 310
• Sort merge join: $3(10+250) = 900$
• Hash join: $3(10+250) = 900$
• Cheapest: Page-oriented nested loops, or simple nested loops

**Ασκηση - Physical Schema Design**
Name three indexes on R (including at least one hash index) that match the predicate in the following SQL query and briefly explain why each index matches.

```
SELECT * FROM R
WHERE (B = "b" OR A <> "a") AND NOT (C <= 20 OR A <> "a")
```

**Answer**
First, rewrite the predicate to CNF:

\[(B = "b" \text{ OR } A <> "a") \text{ AND } (C > 20) \text{ AND } (A = "a")\]

Now, matching indexes become obvious (check class notes). For example, B+ tree indexes that match include one on A, on C, on (A, C), on (C, A), on (A, B), on (C, B), and so on. A hash index on A also matches. At first glance, it seems that no index on B would match. Since the conjunct with B has an OR on another attribute, a hash index on B or on \{B, A\} will not match that conjunct. Similarly, any B+ tree index with B as a prefix of the index key will not match that conjunct. But looking deeper, if we rewrite the CNF further using the basic rules of logic, we find something interesting:

\[
\Leftrightarrow (B = "b" \text{ OR } A <> "a") \text{ AND } (A = "a" \text{ AND } C > 20) \\
\Leftrightarrow (B = "b" \text{ AND } A = "a" \text{ AND } C > 20) \text{ OR } (A <> "a" \text{ AND } A = "a" \text{ AND } C > 20) \\
\Leftrightarrow (B = "b" \text{ AND } C > 20 \text{ AND } A = "a") \text{ OR } \text{FALSE AND } C > 20 \\
\Leftrightarrow (B = "b" \text{ AND } C > 20 \text{ AND } A = "a") \text{ OR } \text{FALSE} \\
\Leftrightarrow B = "b" \text{ AND } C > 20 \text{ AND } A = "a"
\]

Now, we see that a hash index on B matches this new first conjunct! In fact, a hash index on \{A, B\} is even better, since it matches two conjuncts put together! Similarly, a B+ tree index on B, on \{B, C\}, on (B, A), and so on will now match at least the first conjunct. In fact, all 15 B+ tree indexes possible on R will now match this query!

**Ασκηση - Database Statistics**
Consider the following schema.

<table>
<thead>
<tr>
<th>Auctions</th>
<th>(aid, minprice, description, seller, end_date)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td>(mid, nickname, name, since)</td>
</tr>
<tr>
<td>Bids</td>
<td>(aid, buyerid, amount)</td>
</tr>
</tbody>
</table>

Assume there is an unclustered B-tree index on the key of each table. In answering questions, use the summary statistics functions that we learned about in class: \( \text{NPages()}, \text{NTuples()}, \text{Low()}, \text{High()}, \text{NKeys()}, \text{IHeight()}, \text{INPages()} \).

a. Consider the query

```
SELECT 'found it!' FROM Members WHERE mid = 98765;
```

Given the information above, write a formula for the optimizer's lowest estimated cost for this query.

**Answer**

\( \text{IHeight(Btree on members.mid)} \)
b. Consider the query
SELECT * FROM Bids, Members
WHERE bids.buyerid = members.mid AND members.since < '2001';

Write the formula the optimizer would use for the selectivity of the entire WHERE clause.

Answer
\[
\frac{1}{\text{MAX}(\text{Nvals(bids.buyerid)}, \text{NVals(members.mid)})} \times \frac{2001 - \text{MIN(members.since)}}{\text{MAX(members.since)} - \text{MIN(members.since)}}
\]

d. Consider the following query:
SELECT R.*
FROM R, S, T
WHERE R.a = S.b
AND S.b = T.c

The following plans are generated during an intermediate pass of the System R optimizer algorithm. For each plan, write down the ordering column(s) of its output if any, and whether the plan would get pruned (P) or kept (K) at the end of the pass. If there is no clear ordering on the output, write “none”.

<table>
<thead>
<tr>
<th>PLAN</th>
<th>Cost in I/Os</th>
<th>Ordering Columns of Output</th>
<th>Prune or Keep</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexNestedLoops (FileScan(R), IndexOnlyScan(Btree on S.b))</td>
<td>2010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SortMergeJoin(FileScan(R), FileScan(S))</td>
<td>3010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlockNestedLoops(FileScan(R), FileScan(S))</td>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IndexNestedLoops(IndexOnlyScan(Btree on (R.d, R.a)), IndexScan(Btree on S.b))</td>
<td>30000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Answer

<table>
<thead>
<tr>
<th>PLAN</th>
<th>Cost in I/Os</th>
<th>Ordering Columns of Output</th>
<th>Prune or Keep</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexNestedLoops (FileScan(R), IndexOnlyScan(Btree on S.b))</td>
<td>2010</td>
<td>None</td>
<td>P</td>
</tr>
</tbody>
</table>
SortMergeJoin(
  FileScan(R),
  FileScan(S))  
  |  
  |  
  3010  |  
  (R.a, S.b)  |  
  |  K  |  

BlockNestedLoops(
  FileScan(R),
  FileScan(S))  
  |  
  |  
  1010  |  
  None  |  
  |  K  |  

IndexNestedLoops(
  IndexOnlyScan(Btree on
  (R.d, R.a)),
  IndexScan(Btree on S.b))  
  |  
  |  
  30000  |  
  (R.d, R.a)  |  
  |  P  |  

**Greek - Logical Query Optimization**

Draw the query tree structures for the following:

1. \( \pi_A((R \in \Join_{B=C} S) \Join_{D=E} T) \)

2. \( \pi_A((\pi_E(T) \Join_{E=D} \pi_{ACD}(S)) \Join_{C=B} R) \)
   - Write down the sequence of steps needed to transform Query #1 to Query #2.
   - List the attributes that each of the schemas R, S, and T **must** have and the attributes that each (or some) of these schemas must **not** have in order for the above transformation to be correct.

**Answer**

\[ \pi_A \]

\[ \Join_{D=E} \]

\[ \Join_{B=C} \]

\[ \pi_{ACD} \]

\[ \Join_{C=B} \]

\[ \Join_{E=D} \]

\[ \pi_E \]

\[ \pi_A \]

R must have: B (because of the join C=B)
S must have: ACD (because of \( \pi_{ACD} \))
T must have: E (because of \( \pi_E \))

These schemas should not have identically named attributes, because otherwise it will not be clear which of the two identically named attributes will be renamed in the joins. In particular, T should not have A and C, because \( \pi_{ACD}(S) \) clearly suggests that it is expected that S will have the attribute A that
will survive for the outermost projection $\pi_A$ to make sense, and the attribute C, which should survive in order for the join with R to make sense.

**Transformation of Query #1 to Query #2**

1. **Associativity the join**: $(R \Join S) \Join T \equiv R \Join (S \Join T)$
   
   $\pi_A((R \Join B=C S) \Join D=E T) = \pi_A(R \Join B=C (S \Join D=E T))$

2. **Commutativity for the join (B=C)**: $R \Join S \equiv S \Join R$
   
   $\pi_A(R \Join B=C (S \Join D=E T)) \equiv \pi_A((S \Join D=E T) \Join C=B R )$

3. **Using commutativity for the join (D=E)**
   
   $\pi_A((S \Join D=E T) \Join C=B R ) \equiv \pi_A((T \Join E=D S) \Join C=B R )$

4. **Pushing projection $\pi_A$ to the first operand of the join C=B.**
   
   $\pi_A((T \Join E=D S) \Join C=B R ) \equiv \pi_A(\pi_A(T \Join E=D S) \Join C=B R )$

5. **Pushing projection to the first operand in the innermost join (D=E), this is possible if AC are the attributes of S and not of T.**
   
   $\pi_A(\pi_A(T \Join E=D S) \Join C=B R ) \equiv \pi_A(\pi_A(T \Join E=D S)) \Join C=B R )$

6. **Pushing projection to the second operand in the innermost join:**
   
   $\pi_A(\pi_A(T \Join E=D S) \Join C=B R ) \equiv \pi_A(\pi_A(T \Join E=D \pi_ACD(S)) \Join C=B R )$

7. **Reverse of pushing a projection:**
   
   $\pi_A(\pi_A(T \Join E=D \pi_ACD(S)) \Join C=B R ) \equiv \pi_A(\pi_A(T \Join E=D \pi_ACD(S) \Join C=B R )$