CSE 444 Practice Problems

Query Optimization

1. **Query Optimization**

   Given the following SQL query:

```
Student (sid, name, age, address)
Book(bid, title, author)
Checkout(sid, bid, date)

SELECT S.name
FROM Student S, Book B, Checkout C
WHERE S.sid = C.sid
AND B.bid = C.bid
AND B.author = 'Olden Fames'
AND S.age > 12
AND S.age < 20
```

   And assuming:

   - There are $10,000$ `Student` records stored on $1,000$ pages.
   - There are $50,000$ `Book` records stored on $5,000$ pages.
   - There are $300,000$ `Checkout` records stored on $15,000$ pages.
   - There are 500 different authors.
   - Student ages range from 7 to 24.

(a) Show a physical query plan for this query, assuming there are no indexes and data is not sorted on any attribute.

Solution:
Note: many solutions are possible.

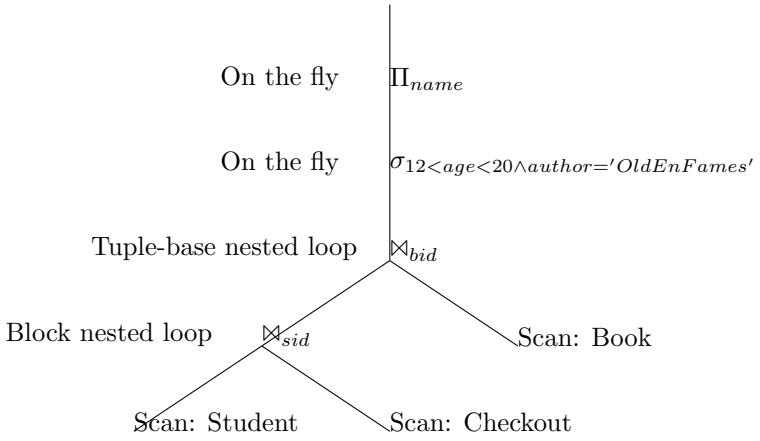


Figure 1: One possible query plan (all joins are nested-loop joins)

(b) Compute the cost of this query plan and the cardinality of the result.

Solution:

	Cost	Cardinality	Remarks
$S \bowtie C$	B(S) + B(S) * B(C) = 1000 + 1000 * 15000 = 15001000	300000 (foreign-key join)	(1)
$(S \bowtie C) \bowtie B$	T($S \bowtie C$) * B(S) = T(C) * B(S) = 300000 * 5000 = 1500000000	300000 (foreign-key join)	(2)
σ and Π	On the fly	300000 * σ_{author} * σ_{age} = 300000 * $\frac{1}{500}$ * $\frac{7}{18}$ $\approx$ 234	(3)
Total	1515001000	234	

(1) We are doing page at a time nested loop join. Also, the output is pipelined to next join.
(2) The output relation is pipelined from below. Thus, we don't need the scanning term for outer relation.
(3) We assume uniform value distributions for age and author. We assume independence among participating columns.

(c) Suggest two indexes and an alternate query plan for this query.

Solution:

Note: many solutions are possible. For purposes of illustation, we assume an unclustered B+-tree index on `Book.author` and a clustered B+tree index on `Checkout.bid`:

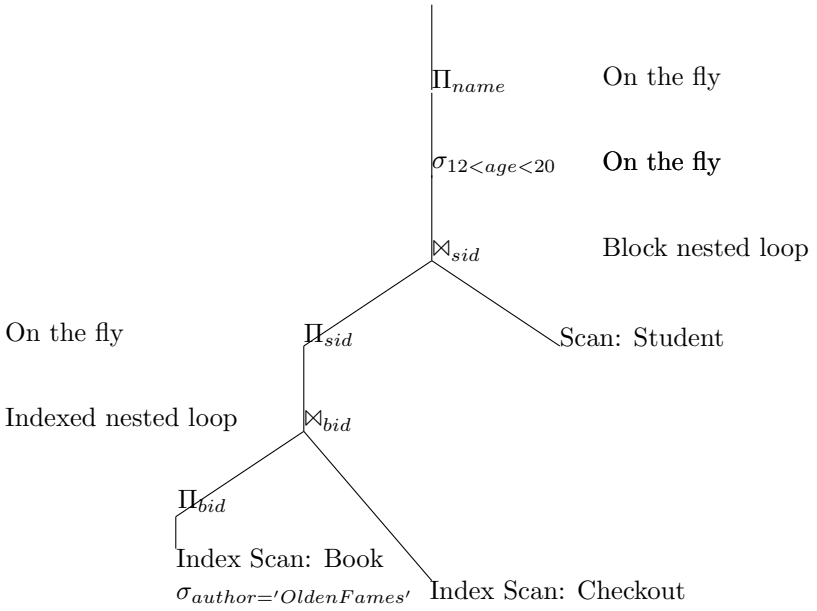


Figure 2: One possible query plan that uses the two indexes

(d) Compute the cost of your new plan.

Solution:

$N(B)$ = # of tuples per page for `Book` = T(B)/B(B) = 10

$N(C)$ = # of tuples per page for `Checkout` = T(C)/B(C) = 20

	Cost	Cardinality	Remarks
Index Scan on Book with σ_{author}	T(B) * 1/V(B) $= 50000 * \frac{1}{500}$ $= 100$	100	(1)
$\Pi_{sid}(B \bowtie C)$	$100 * \lceil (T(C)/V(bid))/N(C) \rceil$ $= 100 * \lceil (300000/50000)/20 \rceil$ $= 100$	100*T(C)/ Max(100, V(C,bid)) $= 600$	(2)
$\bowtie_{sid}$	B(S) = 1000	≈ 234	(3)
Total	1200		

(1) We assume all intermediate index pages are in memory. Note: because `bid` is the search-key for the index, the `bid` values are in the leaf pages of the index: they are thus in memory as per the first assumption. Because we project on `bid` right after the selection, we only need these values, so we do not really need to perform any disk I/Os.

(2) One index lookup per outer tuple. Assuming uniform distribution, there will be 6 checkouts per book. Assuming all intermediate index pages are in memory, the 6 records can be fetched with only one or two disk accesses since we have a clustered index on `Checkount.bid`. The above computation is optimistic but it only incurs 100 more I/Os in the worst case.

(3) Again, the output of the previous operation is projected on *sid*. Because there are only 600 tuples, it is reasonable to assume all results can hold in memory. Since the outer relation is already in-memory, we only need to scan the inner relation `Student` one time.

(e) Explain the steps that the Selinger query optimizer would take to optimize this query.

Solution:

A query optimizer explores the space of possible query plans to find the most promising one. The Selinger query optimizer performs the search as follows:

- Only considering left-deep query plans.
  Instead of enumerating all possible plans and evaluating their costs, the optimizer keeps the efficient pipelined execution model in mind. Thus, it only looks for left-deep query plans and enumerates different join orders. It considers cartesian products as late as possible to reduce I/O costs. It considers only nested-loop and sort-merge joins.
- In bottom-up fashion.
  The optimizer starts by finding the best plan for one relation. It then expands the plan by adding one relation at a time as an inner relation. For each level, it keeps track of the cheapest plan per interesting output order, which will be explained shortly, as well as the cheapest plan overall. When computing the cost of a plan, the Selinger considers both I/O cost and CPU cost.
- Considering interesting orders.
  If the query has an ORDER BY or a GROUP BY clause, having results ordered by the columns that appear in those clauses can reduce the cost of the query plan because it can save extra I/Os needed by sort or aggregation. Similarly, attributes that appear in join conditions are considered interesting orders because they reduce the cost of sort-merge joins. When the Selinger optimizer evaluates a plan, at each stage, it keeps track of the cheapest plan per interesting order in addition to the cheapest plan overall.

2. **Query Optimization**

Consider the following SQL query that finds all applicants who want to major in CSE, live in Seattle, and go to a school ranked better than 10 (i.e., rank < 10).

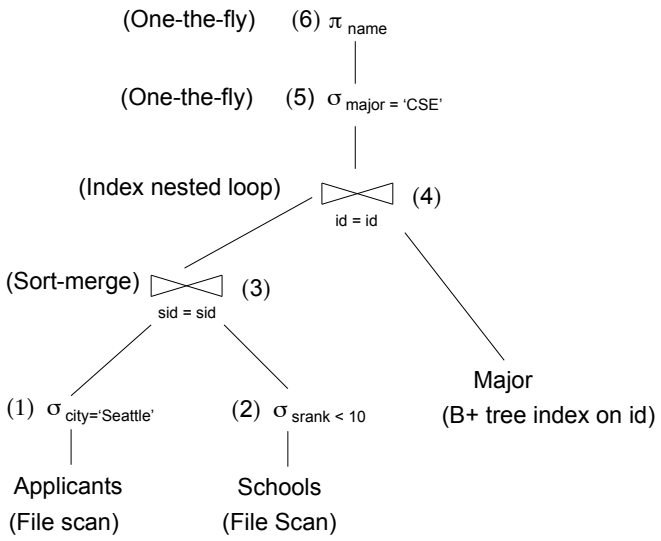
Relation	Cardinality	Number of pages	Primary key
`Applicants (id, name, city, sid)`	2,000	100	`id`
`Schools (sid, sname, srank)`	100	10	`sid`
`Major (id, major)`	3,000	200	`(id,major)`

```
SELECT A.name
FROM   Applicants A, Schools S, Major M
WHERE  A.sid = S.sid AND A.id = M.id
AND    A.city = 'Seattle' AND S.rank < 10 AND M.major = 'CSE'
```

And assuming:

- Each school has a *unique* rank number (`srank` value) between 1 and 100.
- There are 20 different cities.
- `Applicants.sid` is a foreign key that references `Schools.sid`.
- `Major.id` is a foreign key that references `Applicants.id`.
- There is an unclustered, secondary B+ tree index on `Major.id` and all index pages are in memory.

(a) What is the cost of the query plan below? Count only the number of page I/Os.

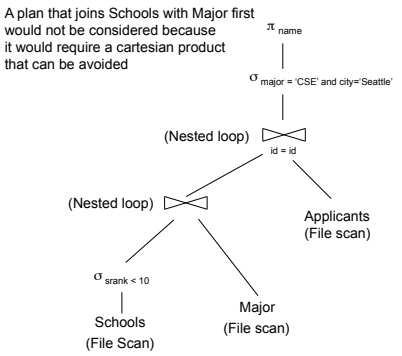


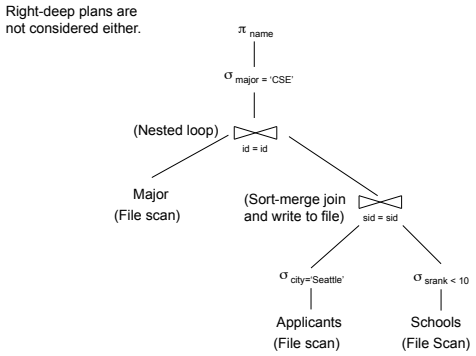
Solution:
The total cost of this query plan is 119 I/Os computed as follows:

- (1) The cost of scanning Applicants is 100 I/Os. The output of the selection operator is $\frac{100}{20} = 5$ pages or $\frac{2000}{20} = 100$ tuples.
- (2) The cost of scanning Schools is 10 I/Os. The selectivity of the predicate on rank is $\frac{10-1}{100} = 0.09$. The output is thus $0.09 * 10 \approx 1$ page or $0.09 * 100 \approx 9$ tuples.
- (3) Given that the input to this operator is only six pages, we can do an in-memory sort-merge join. The cardinality of the output will be 9 tuples. There are two ways to compute this: (a) $\frac{100*9}{(\max(100,9))} = 9$ (see book Section 15.2.1 on page 484) or (b) consider that this is a key-foreign key join and each applicant can match with at most one school but keep in mind that the predicates on city and rank were independent, hence only 0.9 of the applicants end-up with a matching school.
- (4) The index-nested loop join must perform one look-up for each input tuple in the outer relation. We assume that each student only declares a handful of majors, so all the matches fit in one page. The cost of this operator is thus 9 I/Os.
- (5) and (6) are done on-the-fly, so there are no I/Os associated with these operators.

(b) The Selinger optimizer uses a dynamic programming algorithm coupled with a set of heuristics to enumerate query plans and limit its search space. Draw two query plans for the above query that the Selinger optimizer would NOT consider. For each query plan, indicate why it would not be considered.

Solution:
Many solutions were possible including:





3. **Query Optimization**

Consider the schema R(a,b), S(b,c), T(b,d), U(b,e).

(a) For the following SQL query, give two equivalent logical plans in relational algebra such that one is likely to be more efficient than the other. Indicate which one is likely to be more efficient. Explain.

```
SELECT   R.a
FROM     R, S
WHERE    R.b = S.b
         AND S.c = 3
```

Solution:

i. $\pi_a(\sigma_{c=3}(R \bowtie_{b=b} (S)))$

ii. $\pi_a(R \bowtie_{b=b} \sigma_{c=3}(S)))$

ii. is likely to be more efficient

With the select operator applied first, fewer tuples need to be joined.

(b) Recall that a *left-deep* plan is typically favored by optimizers. Write a left-deep plan for the following SQL query. You may either draw the plan as a tree or give the relational algebra expression. If you use relational algebra, be sure to use parentheses to indicate the order that the joins should be performed.

```
SELECT   *
FROM     R, S, T, U
WHERE    R.b = S.b
         AND S.b = T.b
         AND T.b = U.b
```

Solution:

$((R \bowtie_{b=b} S) \bowtie_{b=b} T) \bowtie_{b=b} U$

(c) Physical plans. Assume that all tables are clustered on the attribute b, and there are no secondary indexes. All tables are large. Do not assume that any of the relations fit in memory.

For the left-deep plan you gave in (b), suggest an efficient physical plan.

Specify the physical join operators used (hash, nested loop, sortmerge, etc.) and the access methods used to read the tables (sequential scan, index, etc.). Explain why your plan is efficient.

For operations where it matters, be sure to include the details — for instance, for a hash join, which relation would be stored in the hash tables; for a loop join, which relation would be the inner or outer loop. You should specify how the topmost join reads the result of the lower one.

Solution:

join order doesn't matter, sortmerge for every join, seqscan for R,S,T,U. Fully pipelined.

"clustered index scan" instead of seqscan is also correct.

(d) For the physical plan you wrote for (c), give the estimated cost in terms of $B(...)$, $V(...)$, and $T(...)$. Explain each term in your expression.

Solution:

$B(R) + B(S) + B(T) + B(U)$. Just need to read each table once.