Selinger Query Optimization

Overview

Consider the relations $R(a, b)$, $S(b, c)$, and $T(c, d)$ with clustered B+ alternative 1 tree indexes on $R.a$, $S.b$, and $T.c$ and with unclustered alternative 2 B+ tree indexes on $S.c$ and $T.d$. All columns have type `real`, and all indexes have index keys in the range $[1, 100]$. We want to optimize the following query:

```
SELECT *
 FROM  R, S, T
WHERE  R.b = S.b AND S.c = T.c
  AND  R.a <= 50
  AND  (T.c <= 50 AND T.d <= 20)
```

Pass 1

In the first pass of the Selinger query optimization algorithm, we find the minimum (estimated) cost query plan for every (relation, interesting order) pair. To do so, we estimate the cost of every possible single-relation query plan. We can draw each query plans as a tree where FTS represents a full table scan, IS_x represents a full index scan on a column x, and $\text{IS}_{x \le k}$ represents an index scan on column x which also applies the filter $x \le k$. We also make sure to push down as many selections as possible. Here are all eight of the single-relation query plans (which we've also given shorter names q_R, q_{Ra}, etc.):

q_R	q_{Ra}	q_S	q_{Sb}	q_{Sc}	q_T	q_{Tc}	q_{Td}
$\sigma_{R.a \le 50}$ | FTS | R	$\text{IS}_{a \le 50}$ | R	FTS | S	IS_b | S	IS_c | S	$\sigma_{T.d \le 20}$ | $\sigma_{T.c \le 50}$ | FTS | R	$\sigma_{R.d \le 20}$ | $\text{IS}_{c \le 50}$ | R	$\sigma_{R.c \le 50}$ | $\text{IS}_{d \le 20}$ | R

Now, let's compute the interesting order, (estimated) I/O cost, and output size of each of these query plans. Let $[R]$, $[S]$, and $[T]$ be the number of *pages* in R, S, and T. Similarly, let $|R|$, $|S|$, and $|T|$ be the number of *tuples* in R, S, and T. Assume the clustered alternative 1 B+ trees on R, S, and T have $[R]$, $[S]$, and $[T]$ leaves respectively. That is, every leaf node stores as many tuples as a page in the corresponding file. Let $[S.c]$ and $[T.d]$ be the number of leaf pages in the unclustered alternative 2 indexes on $S.c$ and $T.d$. Also assume it takes 2 I/Os to traverse a B+ tree index from root to leaf.

Query Plan	Interesting Order	I/O Cost	Output Size
q_R	None	$[R]$	$0.5[R]$
q_{Ra}	None	$2 + 0.5[R]$	$0.5[R]$
q_S	None	$[S]$	$[S]$
q_{Sb}	(b)	$2 + [S]$	$[S]$
q_{Sc}	(c)	$2 + [S.c] + \|S\|$	$[S]$
q_T	None	$[T]$	$0.1[T]$
q_{Tc}	(c)	$2 + 0.5[T]$	$0.1[T]$
q_{Td}	None	$2 + 0.2[T.d] + 0.2\|T\|$	$0.1[T]$

Some notes on interesting order:

- q_R, q_S, and q_T have no interesting order because their output is not sorted.

- q_{Ra} produces tuples sorted by a, so you might think that the interesting order of q_{Ra} is (a). However, the query never joins on $R.a$, there is no `GROUP BY` on $R.a$, and there is no `ORDER BY` on $R.a$, so we do not consider this sort order interesting. Similarly, q_{Td} has no interesting order.
- q_{Sb}, q_{Sc}, and q_{Tc} have interesting orders (b), (c), and (c) because their output is sorted on a column which is later joined.

Some notes on I/O cost:

- The I/O cost of q_R, q_S, and q_T is $[R]$, $[S]$, and $[T]$ because a full table scan requires us to read every page in a relation.
- The I/O cost of q_{Ra} is $2 + 0.5[R]$. It takes 2 I/Os to traverse the B+ tree index on $R.a$ from the root to its leftmost leaf. Then, we have to sequentially read (starting from the leftmost leaf and moving rightward) every leaf of the alternative 1 B+ tree that contains tuples satisfying $R.a \leq 50$. Because the values of a fall in the range $[0, 100]$, the selectivity of $R.a \leq 50$ is 0.5. Thus, we estimate that only half of the tuples in R satisfy $R.a \leq 50$. This means that we only have to read half the leaves, which requires $0.5[R]$ I/Os. A similar line of reasoning can be used to compute the I/O cost of q_{Sb} and q_{Tc}.
- The I/O cost of q_{Td} is $2 + 0.2[T.d] + 0.2|T|$. Again, it takes 2 I/Os to traverse the unclustered alternative 2 B+ tree index on $T.d$ from the root to the leftmost leaf. Then, we sequentially read (starting from the leftmost leaf and moving rightward) every leaf which points to tuples satisfying $T.d \leq 20$. Because the values of d fall in the range $[0, 100]$, the selectivity of $T.d \leq 20$ is 0.2. Thus, we estimate that only a fifth of the tuples in T satisfy $T.d \leq 20$. This means that we only need to read a fifth of the leaves, which requires $0.2[T.d]$ I/Os. For every record id read from a leaf, we have to retrieve the corresponding tuple. Because the B+ tree is unclustered, this requires 1 I/O for every tuple. Thus, it takes $0.2|T|$ I/Os. A similar line of reasoning can be used to compute the I/O cost of q_{Sc}.

Some notes on output size:

- Note that the estimated output size of q_R and q_{Ra} is the same; the estimated output size of q_S, q_{Sb}, and q_{Sc} is the same; and the estimated output size of q_T, q_{Tc}, and q_{Td} is the same. This is no coincidence. The estimated output size of a relational algebra expression will be the same no matter which query plan we choose to implement it.
- The estimated output size of q_R and q_{Ra} is $0.5[R]$ because the selectivity of $R.a \leq 50$ is 0.5.
- The estimated output size of q_S, q_{Sb}, and q_{Sc} is $[S]$ because we read every single tuple of S.
- The estimated output size of q_T, q_{Tc}, and q_{Td} is $0.2[T]$ because the selectivity of $T.c \leq 50 \wedge T.d \leq 20$ is $0.5 \times 0.2 = 0.1$.

Now that we've computed the I/O cost for every single-relation query plan, we retain the lowest cost plan for each (relation, interesting order pair). We'll use this information in Pass 2:

Relation	Interesting Order	Lowest Cost Query Plan
R	None	q_{Ra}
S	None	q_S
S	(b)	q_{Sb}
S	(c)	q_{Sc}
T	None	q_T
T	(c)	q_{Tc}

Notice that we do not retain q_R because q_{Ra} has lower cost and both have no interesting order. Similarly, we do not retain q_{Td} because q_T has lower cost and both have no interesting order. Also note that even though q_{Sc} has a very high cost[1], there is no other single-relation query plan over S with the interesting order (c), so we retain q_{Sc}.

[1]The cost of q_{Sc} involves $|S|$, the number of *tuples* in T. Typically, this value will be much *much* larger than $[S]$, the number of *pages* in T.

Pass 2

In the second pass of the Selinger query optimization algorithm, we find the minimum (estimated) cost query plan for (almost) every (set of 2 relations, interesting) order pair. To do so, we estimate the cost of (almost) every possible two-relation query plan in which the left and right subqueries are two of the minimum cost single-relation query plans we computed during Pass 1. Considering only block nested loop joins (BNLJ) and sort-merge joins (SMJ), we can list all the two-relation query plans we will consider:

BNLJ q_{Ra} q_S	SMJ q_{Ra} q_S	BNLJ q_{Ra} q_{Sb}	SMJ q_{Ra} q_{Sb}	BNLJ q_{Ra} q_{Sc}	SMJ q_{Ra} q_{Sc}
BNLJ q_S q_R	SMJ q_S q_R	BNLJ q_S q_T	SMJ q_S q_T	BNLJ q_S q_{Tc}	SMJ q_S q_{Tc}
BNLJ q_{Sb} q_R	SMJ q_{Sb} q_R	BNLJ q_{Sb} q_T	SMJ q_{Sb} q_T	BNLJ q_{Sb} q_{Tc}	SMJ q_{Sb} q_{Tc}
BNLJ q_{Sc} q_R	SMJ q_{Sc} q_R	BNLJ q_{Sc} q_T	SMJ q_{Sc} q_T	BNLJ q_{Sc} q_{Tc}	SMJ q_{Sc} q_{Tc}
BNLJ q_T q_S	SMJ q_T q_S	BNLJ q_T q_{Sb}	SMJ q_T q_{Sb}	BNLJ q_T q_{Sc}	SMJ q_T q_{Sc}
BNLJ q_{Tc} q_S	SMJ q_{Tc} q_S	BNLJ q_{Tc} q_{Sb}	SMJ q_{Tc} q_{Sb}	BNLJ q_{Tc} q_{Sc}	SMJ q_{Tc} q_{Sc}

Notice that this list doesn't include *every* pair of single-relation query plans from Part 1. Notice that BNLJ(q_{Ra}, q_T) is missing for example. This is because R does not join with T in our query, so we ignore it. Also notice that BNLJ(q_{Ra}, q_S) is considered but BNLJ(q_R, q_S) is not. This is because we q_{Ra} has a lower cost than q_R and both have the same interesting order.

As with Pass 1, we now compute the interesting order, I/O cost, and output size of these query plans. There's a lot of query plans, so let's only look at a handful. Let B be the number of buffer pages, let $SC(x) = 2x\lceil 1 + \log_{B-1}(\lceil \frac{x}{B} \rceil) \rceil$ be the cost of sorting x pages, and assume the merge phase of a sort merge join on two relations of size n and m can be done in $n + m$ I/Os.

Query Plan	Interesting Order	I/O Cost	Output Size
BNLJ(q_{Ra}, q_S)	None	$\approx 0.5[R] + \left\lceil \frac{0.5[R]}{B-2} \right\rceil [S]$	$\frac{0.5[R][S]}{100}$
SMJ(q_{Ra}, q_S)	None	$\approx SC(0.5[R]) + SC([S]) + 0.5[R] + [S]$	$\frac{0.5[R][S]}{100}$
BNLJ(q_{Ra}, q_{Sb})	None	$\approx 0.5[R] + \left\lceil \frac{0.5[R]}{B-2} \right\rceil [S]$	$\frac{0.5[R][S]}{100}$
SMJ(q_{Ra}, q_{Sb})	None	$\approx SC(0.5[R]) + 0.5[R] + [S]$	$\frac{0.5[R][S]}{100}$
BNLJ(q_{Sb}, q_{Tc})	None	$\approx 0.5[T] + 0.1[T] + [S] + \left\lceil \frac{[S]}{B-2} \right\rceil 0.1[T]$	$\frac{0.1[S][T]}{100}$
SMJ(q_{Sb}, q_{Tc})	None	$\approx 0.5[T] + 0.1[T] + SC([S]) + [S] + 0.1[T]$	$\frac{0.1[S][T]}{100}$

Some notes on interesting order:

- BNLJ(q_{Ra}, q_S), BNLJ(q_{Ra}, q_{Sb}), and BNLJ(q_{Sb}, q_{Tc}) have no interesting order because a BNLJ does not output tuples in any particular order, even if one or both of its inputs are sorted.
- SMJ(q_{Ra}, q_S) and SMJ(q_{Ra}, q_{Sb}) output tuples sorted on (b), but the (b) column is no longer interesting since it will no longer be used as part of a join, and it is not part of a `GROUP BY` or `ORDER BY`. Similarly, SMJ(q_{Sb}, q_{Tc}) outputs tuples sorted on (c), but the c column is not interesting.

Some notes on I/O cost:

- We can estimate the I/O cost of BNLJ(q_{Ra}, q_S) and BNLJ(q_{Ra}, q_{Sb}) using the BNLJ cost formula where q_{Ra} has $0.5[R]$ pages and q_{Sb} has $[S]$ pages (computed in Pass 1). The actual cost may include some additional $+2$ terms for reading indexes, but we omit them to keep things simple. We'll do this throughout our I/O cost computation in Pass 2. Also note that the right hand side of BNLJ(q_{Ra}, q_{Sb}) is a clustered alternative 1 index scan, not a full table scan. Because of this we might need to materialize q_{Sb} to disk before using it as part of a join; it depends on how smart our BNLJ implementation is. Here, we assume our BNLJ implementation is smart enough to avoid this materialization.

- The I/O cost of BNLJ(q_{Sb}, q_{Tc}) is $0.5[T] + 0.1[T] + [S] + \left\lceil \frac{[S]}{B-2} \right\rceil 0.1[T]$. The righthand side of this BNLJ (i.e. q_{Tc}) includes a filter over an index scan. As a result, we have to materialize q_{Tc} to disk before we can use it as the righthand side of a join[2]. It takes $0.5[T]$ I/Os to read in q_{Tc} and $0.1[T]$ I/Os to write it to disk. Note that the number of I/Os to read and write q_{Tc} are different because of the selections that filter out some of the tuples we read in. After we have materialized q_{Tc}, we incur the standard BNLJ cost of $[S] + \left\lceil \frac{[S]}{B-2} \right\rceil 0.1[T]$. Again, we have dropped some $+2$ terms to keep things simple.

- The cost of SMJ(q_{Ra}, q_S) is $SC(0.5[R]) + SC([S]) + 0.5[R] + [S]$. Neither q_{Ra} nor q_S output tuples sorted by (b), so we have to sort both. Sorting q_{Ra} takes $SC(0.5[R])$ I/Os and sorting q_S takes $SC([S])$ I/Os. Merging the two takes $0.5[R] + [S]$ I/Os. Note that if we have enough buffer pages, it's possible to use the optimized variant of sort-merge join in which we join the two relations as we merge them. In this case, the I/O cost would be even lower. Yet again, we omit $+2$ terms for simplicity.

- The cost of SMJ(q_{Ra}, q_{Sb}) is $SC(0.5[R]) + 0.5[R] + [S]$. q_{Ra} is not sorted by (b), but q_{Sb} is. Thus, we have to sort q_{Ra}, which takes $SC(0.5[R])$ I/Os, but we do not have to sort q_S. Merging the two takes $0.5[R] + [S]$ I/Os. As above, we can also used the optimized variant of sort-merge join if we have enough buffer pages.

- To evaluate SMJ(q_{Sb}, q_{Tc}), we first materialize q_{Tc} (as we did with BNLJ(q_{Sb}, q_{Tc}). This incurs $0.5[T] + 0.1[T]$ I/Os. Then, we sort q_{Sb} because it is not sorted on column (c); this incurs $SC([S])$ I/Os. We do not have to sort q_{Tc} because it is already sorted on column (c). Finally, we merge q_{Sb} and q_{Tc} which incurs $[S] + 0.1[T]$ I/Os. Note that materializing q_{Tc} is not strictly necessary; we can likely use q_{Tc} as part of our sort-merge join without materializing it. We materialize it here to keep things simple.

Some notes on output size:

- As we noted in Pass 1, the output size of a relational algebra expression is the same no matter which plan we choose to implement it. Thus, BNLJ(q_{Ra}, q_S), SMJ(q_{Ra}, q_S), BNLJ(q_{Ra}, q_{Sb}), and SMJ(q_{Ra}, q_{Sb}) all have the same output size. The maximum size of $R \bowtie_{R.b=S.b} S$ is $[R][S]$. The selectivity of $R.a \leq 50 \wedge R.b = S.b$ is the selectivity of $R.a \leq 50$ multiplied by the selectivity of $R.b = S.b$. The selectivity of $R.a \leq 50$ is 0.5, and the selectivity of $R.b = S.b$ is $\frac{1}{\max(100,100)} = \frac{1}{100}$ Thus, the selectivity of $R.a \leq 50 \wedge R.a = R.b$ is $\frac{0.5}{100}$, and the output size is $\frac{0.5[R][S]}{100}$. A similar calculation is used to compute the output size of SMJ(q_{Sb}, q_{Tc}) and BNLJ(q_{Sb}, q_{Tc}).

[2] Note that we did not have to materialize q_{Sb} as the righthand side of a join because it was a clustered alternative 1 index scan. It's relatively straightforward to implement BNLJ to handle this. We *do* have to materialize q_{Tc} because it is a *selection on top of* an index scan.