CS W186 Introduction to Database Systems
Fall 2019 Josh Hug **DIS 7**

1 Selectivity Estimation

Consider a relation R(a, b, c) with 1000 tuples. We have an index on a with 50 unique values in the range [1, 50] and an index on b with 100 unique values in the range [1, 100]. We do not have an index on c.

Use selectivity estimation to estimate the number of tuples produced by the following queries.

1. SELECT * FROM R

   1000 (no predicates, so select all tuples)

2. SELECT * FROM R WHERE a = 42

   Sel = 1 / unique values of a = 1/50
   1000 * (1/50) = 20

3. SELECT * FROM R WHERE b = 42

   Sel = 1 / unique values of b = 1/100.
   1000 * (1/100) = 10

4. SELECT * FROM R WHERE c = 42

   Sel = 1/10 because we do not have any information on the relation we use 1/10 as default.
   1000 * (1/10) = 100

5. SELECT * FROM R WHERE a <= 25

   Sel = (v – low) / (high – low + 1) + 1 / distinct values = (25 – 1) / (50 – 1 + 1) + 1/ 50 = 1/2
   1000 * (1/2) = 500

6. SELECT * FROM R WHERE b <= 25

   Same formula as part 5, or just look and see that the first 25 values are 1/4 of the values between 1 and 100: Sel = 1/4
   1000 * (1/4) = 250

7. SELECT * FROM R WHERE c <= 25

   Sel = 1/10
   1000 * (1/10) = 100

8. SELECT * FROM R WHERE a <= 25 AND b <= 25

   Sel = Sel(a <= 25) * Sel(b <= 25) = 1/2 * 1/4 = 1/8
   1000 * (1/8) = 125

9. SELECT * FROM R WHERE a <= 25 AND c <= 25

   Sel = Sel(a <= 25) * Sel(c <= 25) = 1/2 * 1/10 = 1/20
   1000 * (1/20) = 50

10. SELECT * FROM R WHERE a <= 25 OR b <= 25

   Sel = Sel(a <= 25) + Sel(b <= 25) – Sel(a <= 25) * Sel(b <= 25) = 1/2 + 1/4 – 1/2 * 1/4 = 5/8
   1000 * (5/8) = 625

11. SELECT * FROM R WHERE a = b

   Sel = 1/max(distinct values of a, distinct values of b) = 1/max(50, 100) = 1/100
   1000 * (1/100) = 10

12. SELECT * FROM R WHERE a = c

   We don’t have information on c, so we just use the number of distinct values in a: Sel = 1/50
   1000 * (1/50) = 20

2 Single Cost Plans

Consider the relations R(a, b), S(b, c), and T(c, d) with an alt 1 index on R.a, alt 2 clustered indexes on S.b, and T.c, and alt 2 unclustered indexes on S.c and T.d. Assume it takes 2 IOs to reach a leaf node and no index or data pages are ever cached. 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)

In the first pass of the Sellinger query optimization algorithm, we compute the minimum cost access method for every (relation, interesting order) pair. Complete the following table which computes this. Let [R], [S], and [T] and |R|, |S|, and |T| be the number of pages and tuples in R, S, and T, respectively. Let [S.b], [T.c], [S.c], and [T.d] be the number of leaf pages in each index corresponding index.

Relation	Access Method	Interesting Order	I/O Cost	Output Size	Retained
R	Full Table Scan	None	[R]	0.5[R]	No
	Index Scan on a	None	2 + 0.5[R]		Yes
S	Full Table Scan	None	[S]	[S]	Yes
	Index Scan on b	(b)	2 + [S.b] + [S]		Yes
	Index Scan on c	(c)	2 + [S.c] + |S|		Yes
T	Full Table Scan	None	[T]	0.1[T]	No
	Index Scan on c	(c)	2 +0.5[T.c] + 0.5[T]		Yes
	Index Scan on d	None	2 + 0.2[T.d] + 0.2|T|		No

Interesting Order Explanation: For there to be an interesting order the table must be sorted. This immediately rules out all full table scans. The table must also be sorted on a column that is useful later or (either used in downstream joins, group by, or order by). Only columns (b) and (c)are used later in the query, so sorting on (a) and (d) is not interesting.

IO Cost Explanation: Full table scans always require scanning all the pages.

Alt 1 scan requires reaching the first leaf/data pages and scanning until you no longer match the condition. Because the selectivity is 0.5 only 1⁄2 the leaves will actually meet the condition.

Alt 2 scans require (cost to reach leaf) + (leaf nodes read) + (data pages read). For clustered indexes we assume that data pages read is the number of data pages matching condition, but for unclustered we have to read a data page for every matching record. Note that you can only apply the selectivity for conditions that apply to the column the index is built on. If we're doing an index scan on c, the predicate relating to d does not help us avoid reading any leaf or data pages.

Output Size Explanation: This comes directly from our selectivity formulas.

Retained Explanation: We retain the access plan with the lowest IO cost for each interesting order and the lowest IO cost overall (if it's not already included with the lowest interesting orders).

3 Multi Table Plans

1. In the second pass of the Sellinger query optimization algorithm, we consider each (relation, interesting order) pair in turn. For each, we compute the cost of joining every other relation into it. In the end, we retain the minimum cost join for each (relations, interesting order) pair. Complete the following table which computes **part** of the second pass. Let B be number of buffer pages, use SC(t) for the sorting cost of table t, assume we take the ceiling of all fractions, and assume that there are not many duplicates in the join columns.

Left Relation	Left Ordering	Right Relation	Right Ordering	Join Type	Interesting Order	IO Cost of the Join	Output Size
R	(a)	S	(b)	BNLJ	None	0.5[R] + $\frac{0.5[R]}{B-2}$[S]	$\frac{0.5[R][S]}{100}$
				SMJ	None	SC(0.5[R]) + 0.5[R] + [S]	
S	None	R	None	BNLJ	None	[S] + $\frac{S}{B-2}$0.5[R]	$\frac{0.5[R][S]}{100}$
				SMJ	None	SC(0.5[R]) + SC([S]) + 0.5[R] + [S]	
S	None	T	(c)	BNLJ	None	[S] + $\frac{S}{B-2}$0.1[T]	$\frac{0.1[S][T]}{100}$
				SMJ	None	SC([S]) + [S] + 0.1[T]	
S	(b)	R	(a)	BNLJ	None	[S] + $\frac{S}{B-2}$0.5[R]	$\frac{0.5[R][S]}{100}$
				SMJ	None	SC(0.5[R]) + [S] + 0.5[R]	
S	(b)	T	(c)	BNLJ	None	[S] + $\frac{S}{B-2}$0.1[T]	$\frac{0.1[S][T]}{100}$
				SMJ	None	SC(S) + [S] + 0.1[T]	

Interesting Order Explanation: BNLJ does not make any guarantees about output order, so it can never have an interesting order. When running SMJ, the output will be ordered based on the join condition. For example, joining R and S with SMJ will produce rows ordered by the (b) column because the join condition is R.b = S.b. After evaluating this join, however, being sorted on (b) is no longer useful because (b) is not used anywhere else in the query.

The same goes for if we join S and T, after evaluating the join being ordered on (c) is no longer useful.

IO Cost of Join Explanation: Plug and chug into the formulas. Use the output size from part 1 rather than the entire table, because we are only joining the pages that made it past the scan. If we have an interesting order on the column that we are joining on, we don't have to sort that table for sort merge join because it is already sorted.

Output Size Explanation: This comes directly from our selectivity estimate for a=b.

2. Is it possible that the SMJ of the bottom row ever advances to the next pass of the query optimization algorithm?

   No. Notice we have two SMJ(S, T). Both of them have the same join IO cost and neither produce an interesting order. The bottom row's single table plan for S has an interesting order on b while the other SMJ(S, T) has no interesting order for S. Consulting the table from part 2, we can see that getting an interesting order on b from table S requires more IOs than the non-interesting order access pattern. This means that the total number of IOs for the bottom row is strictly more so we can never retain it.

3. Add a condition to the WHERE clause so that the SMJ of R and S produces an interesting order.

   S and R are joined on the b column, meaning that the result of the SMJ will be sorted on b. In order for this to be interesting, we need to use the b column in a downstream join, ORDER BY or GROUP BY. A condition we could add is: **AND S.b = T.d**

4. Why don't we consider joining R and T in our table?

   There is no join condition between R and T so this would be a cross product and we don't consider cross products in Sellinger style optimizers unless we have to.