solutions_V2

Tuesday, 22 November 2022 10:27

solutions_V2

2022 Data Systems Final Exam (Solutions)

Jayitha C[1]

[1] jayitha.c@research.iiit.ac.in

November 22, 2022

Contents

Question 1

Question 1. Let $R_1, R_2, \dots, R_k$ be k relations, each having attribute A. Let A have domain values $dom(A) = (d_1, d_2, \dots, d_p)$.

Let $(n_{i1}, n_{i2}, \dots, n_{ip})$ be the number of rows, respectively, for domain values $(d_1, d_2, \dots, d_p)$ of attribute A in relation R_i. That is, n_{ij} , is the number of rows in R_i when attribute A takes domain value d_j. Let

$$T = R_1 \bowtie_{R_1.A=R_2.A} R_2 \bowtie_{R_2.A=R_3.A} R3 \dots R_{k-1} \bowtie_{R_{k-1}.A=R_k.A} R_k$$

Compute the maximum number and the minimum number of rows for T. State which strategy (if any) can be used to efficiently compute T.

Answer. The number of rows in T is given by the following equation

$$\sum_{j=1}^{p} \prod_{i=1}^{k} n_{ij} \quad (1)$$

For each value d_j in the $dom(A)$, there are $n_{1j} \times n_{2j} \dots n_{kj}$ tuples. Eq. (1) is the minimum and maximum number of tuples in T without imposing any constraints on the "histograms" of the relations. Assuming we can, then the minimum number of tuples of T is 0. This minimum can occur in the extreme case where one relation is empty. However, the requirement for this minimum case is stated below

$$\forall_{d_j \in dom(A)} \exists_{R_i} n_{ij} = 0$$

For each value d_j in the domain, there should exist at least one relation that does not contain any tuples with value d_j for A. The maximum number of tuples of T in theory is the cross product of all the relations. Let $N_1, N_2 \dots, N_k$ be the number of tuples of each relation. Let $n_{i1} = N_i$ and $n_{ij} = 0$ for all $j \neq 1$ i.e. every tuple of every relation has value d_1 for attribute A. Then the number of tuples of T is

$$\prod_{i=1}^{k} N_i \quad (2)$$

Eq. (2) can also be rewritten in terms of the histogram values as so

$$\prod_{i=1}^{k} \sum_{j=1}^{p} n_{ij}$$

This form relates to Eq. (1) using the follow property. Let $a_{11}, a_{12}, a_{21}, a_{22}$ be non-negative numbers, then

$$a_{11}a_{21} + a_{12}a_{22} \leq a_{11}a_{21} + a_{12}a_{22} + a_{11}a_{22} + a_{12}a_{21} = (a_{11} + a_{12})(a_{21} + a_{22})$$

Which strategy would I use to efficiently compute T? The objective is to reduce the number of block accesses. To do this, we can split the problem up into disjoint join operations. We first partition **ALL** the relations using the same partitioning function. The partitioning

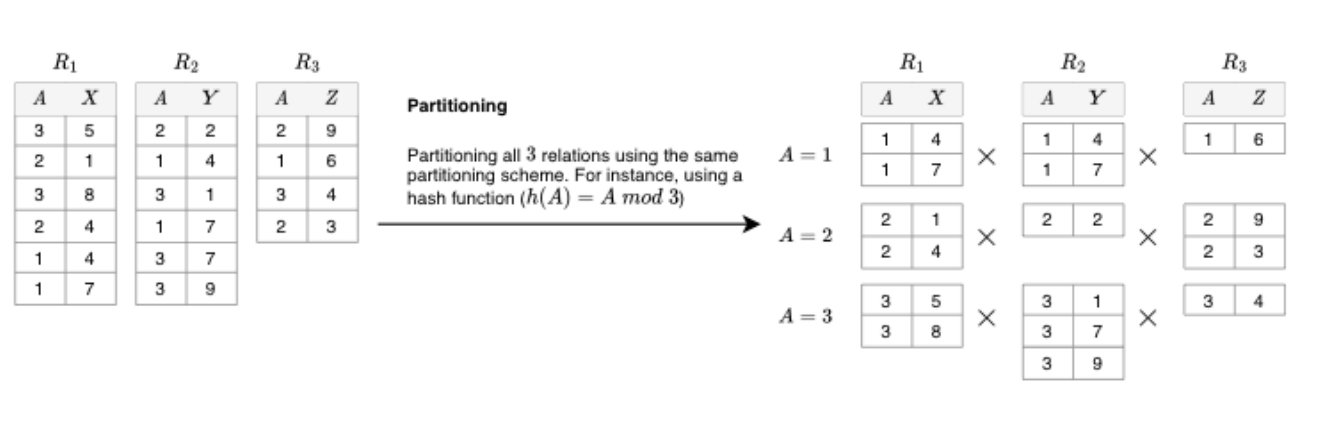


Figure 1: Partitioned multi-join example

If any partition for any Relation is empty, the result is empty.

function maps the each value in the domain of A to one partition (for instance a hash or block/range partition). Take for example the three ($k = 3$) relations shown in Fig. 1. The domain of attribute A is $\{1, 2, 3\}$. Every relation is partitioned using the same partition function ($h(A) = A \bmod 3$). Relation R_1 generates three partitions – R_1^1 of tuples that hash to 1, R_1^2 of tuples that hash to 2 and R_1^3 of tuples that hash to 0. These partitions are written back to the disk. R_2 and R_3 undergo the same partitioning. Fig. 1 shows the results of partitioning the relations. It is important to note that tuples that correspond to one partition do not join with tuples in any other partition. For instance, tuples of R_1^i only join with tuples of R_2^i and R_3^i. Therefore, the large multi-join has been split up into smaller multi-joins (or in this case, cartesian product) operations of the partitions.

$$R_1 \bowtie_A R_2 \bowtie_A R_3 = \bigcup_{i=1}^{3} R_1^i \bowtie_A R_2^i \bowtie_A R_3^i$$

Each partition join is performed one after the other. To perform the first partition join, all the tuples corresponding to partitions R_1^1, R_2^1 and R_3^1 are loaded into main memory and a cartesian product is performed. A similar procedure takes place for the other partition joins. The granularity of the partitioning function and the implementation of the operators can be altered to accomodate the main memory size.[2]

[2] For instance, if a partition is too big, we can check to see if smaller partitions can be created. If not, the the implementation of the operator needs to accomodate this case. If the partitions are too small, then too many block accesses will take place and we will not utilise the main memory efficiently. To handle this, smaller partitions can be combined using some bin-packing algorithm. The histogram will aid this optimization.

Question 2

Question 2. Consider a relation R having 10^6 ($1M$) rows with a non-null attribute B having domain values {YES, NO}. The data page size is 2000, row length for R is 100. The length of attribute B is 10, and the length of the record pointer and block pointer is 10 bytes each. Let there be a single-level secondary index on B. The amount of buffer space for storing the index is much smaller than the size of the index. What are the minimum and maximum number of data page accesses needed to execute

```
SELECT * FROM R WHERE B = "YES";
```

using the single level secondary index (NOT B/B+ tree) on B, when there are 9, 99, 980 rows with the value 'YES', when there are 40, 000 rows with the value 'YES', and when there is one row with the value 'YES'.

Answer. In theory, the secondary index can be represented as a set of $\langle K, P \rangle$ pairs where K is the key ($K \in$ {YES, NO}) and P is a record or block pointer that point to a tuple with value K. Using the implementation specified in the textbook[3] I assume indirection blocks are used to store the pointers. Each index entry points to a block which contains block or record pointers[4]. If the number of pointers are large, then the index points to a linked list of indirection blocks that hold pointers.

[3] R Elmasri and Shamkant B Navathe. *Fundamentals of Database Systems*. Springer, 2000. Accessed on 19/11/2022; Link

[4] According to the textbook, a record pointer contains a block pointer.

Since we are optimizing for the number of blocks accessed, storing record pointers over block pointers increases the size of the index and the number of block accesses. In the best case, block pointers are used and in the worst case, record pointers are used.

If record pointers are used, then the semantics associated with record pointers are followed. To elaborate, when executing the query, for each record pointer, the relevant block is read into memory. Therefore, the same block can be read into main memory multiple times if multiple record pointers point to the same block.

The total number of block accesses needed to execute the query is the sum of the number of index blocks and R data blocks accessed.

$$\# \text{ INDEX BLOCKS} + \# \text{ DATA BLOCKS}$$

Some math:

The number of data records per page $= \lfloor \frac{2000}{100} \rfloor = 20$

The number of pages needed to store $R = \lceil \frac{10^6}{20} \rceil = 5 \times 10^4$

The number of block or record pointers per page $= \lceil \frac{2000}{10} \rceil = 200$

The number of index records per page ($\langle K, P \rangle$) $= \lceil \frac{2000}{20} \rceil = 100$

There are only 2 index records, therefore, only one block is needed to store the index records (more blocks are needed to store the indirection blocks)

| YES | = 999980

In the best case, all YES records are stored contiguously in $5 \times 10^4 - 1$ blocks. The number of indirection blocks accessed is

$$\left\lceil \frac{5 \times 10^4 - 1}{200} \right\rceil = \left\lceil \frac{5 \times 10^4}{200} - \frac{1}{200} \right\rceil = \left\lceil 250 - \frac{1}{200} \right\rceil = 250$$

The number of index blocks accessed is $= 1 + 250 = 251$. The additional block access is necessary to bring the index records into memory.

Therefore, the minimum number of block accesses is $= 5 \times 10^4 - 1 + 251 = 50250$

In the worst case, all blocks are accessed and a few are accessed multiple times. Specifically, the worst case needs 999980 block accesses. Since record pointers are used, 999980 record pointers are stored in the index. The number of index blocks accessed is

$$1 + \left\lceil \frac{999980}{200} \right\rceil = 1 + \lceil 4999.9 \rceil = 5001$$

Therefore, the maximum number of block accesses is $= 999980 + 5000 = 1004981$

| YES | = 40000

Again, in the best case these records are stored contiguously. The number of data block accesses is

$$\left\lceil \frac{40000}{20} \right\rceil = 2000$$

and the index holds block pointers for each of these 2000 blocks. The number of index block accesses is

$$1 + \left\lceil \frac{2000}{200} \right\rceil = 11$$

Therefore, the minimum number of block accesses is $= 2000 + 11 = 2011$

In the worst case, we access at most 40000 blocks[5]. 4K block pointers

[5] We cannot partition 40000 rows into 50000 blocks such that each block gets at least one row

are stored in the index. The number of index blocks accessed is

$$1 + \left\lceil \frac{40000}{200} \right\rceil = 201$$

Therefore, the maximum number of block accesses is $= 40000 + 201 = 40201$

| YES | = 1

Both in the best and worst case, one data block and two index blocks are accessed.

Therefore, the maximum and minimum number of block accesses is $= 1 + 2 = 3$

# YES	Minimum	Maximum
999981	50250	1004981
40001	2011	40201
1	3	3

There is another argument to be made for the maximum number of block accesses. The textbook discusses an implementation of the secondary index that involves storing $< K, P >$ pairs explicitly where the key K is repeated for each pointer. In such a case, each block can only store 100 pointers and not 200 as assumed before. If this argument is made, I present the results below

# YES	Minimum	Maximum
999981	50500	1009981
40001	2021	40401
1	3	3

Question 3

Question 3. Consider four relations R_1, R_2, R_3, R_4 like in Question 1, with attribute A in all relations and domain values of A as $dom(A) = \{1, 2, 3, 4, 5, 6, 7\}$. Let n_{ij} values be:

$dom(A)$	R_1	R_2	R_3	R_4
1	10	10	40	50
2	20	20	80	60
3	20	40	90	90
4	100	0	100	70
5	10	20	20	80
6	10	10	30	50
7	20	30	40	30

Table 1: n_{ij} values for each R_i for each value $j \in dom(A)$

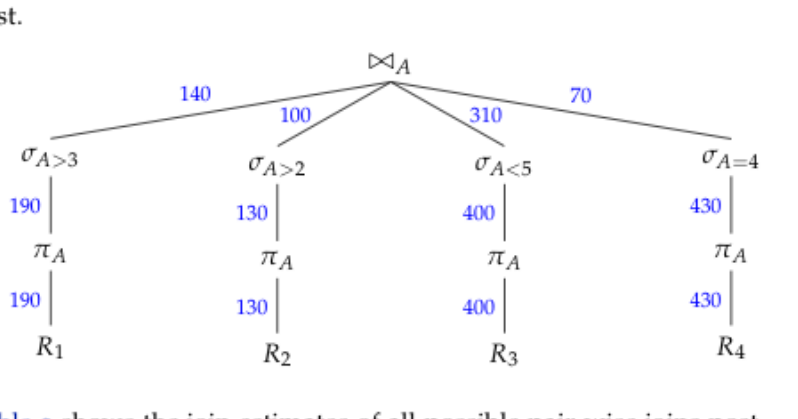
Consider the following query.

```
SELECT R1.A
FROM R1, R2, R3, R4
WHERE R1.A = R2.A AND R2.A = R3.A AND R3.A = R4.A
    AND R1.A > 3 AND R2.A > 2 AND R3.A < 5 AND R4.A = 4
```

Come up with the best possible cost-based query optimization plan with sizes of the intermediate results (in the number of rows). Give at most three sentences of justification for your query execution plan. What is the result of the query?

Answer. After using the heuristics detailed in Pg. 700 of the textbook[6], the optimization problem boils down to a join ordering problem as shown below. The intermediate sizes are shown post the selection and projection operators. Since the histogram is provided, I assume the query optimizer uses the histogram values to estimate cost.

[6] Optimizations include – (1) Pushing selections and projections down the query tree and (2) changing cartesian product followed by selection operators to join operators. The projection operator is pushed down to all relations because only attribute A participates in the query (See point 5 on Pg 700). Note that the projection here is SQL **SELECT**, therefore, the result is not a relation, it is a table i.e. duplicates are not eliminated



-	R_1	R_2	R_3	R_4
R_1		900	10K	7K
R_2			3.6K	0
R_3				7K
R_4				

Table 2: Join size estimates of pairwise joins

Table 2 shows the join estimates of all possible pair-wise joins post the selection and projection operations. Either the join between relation R_2 and R_4 or the multijoin between relations R_1, R_3 and R_2

needs to be performed to ascertain that the resulting relation is empty. The join estimates in Table 2 indicate that performing the join between R_2 and R_4 at the earliest is optimal. Therefore, the optimal join tree is a left-deep tree with the first join operation being that between R_2 and R_4. All in all, the query plan first loads R_4 perfoming the selecion and projection operators, then similar operations are performed on R_2 while simultaneously probing tuples of $R_4.A$. The results of this join operator (0 rows) stay in the main memory as the build for further join processing, which does not take place. Therefore, the final optimized query plan is shown below (Fig. 2)

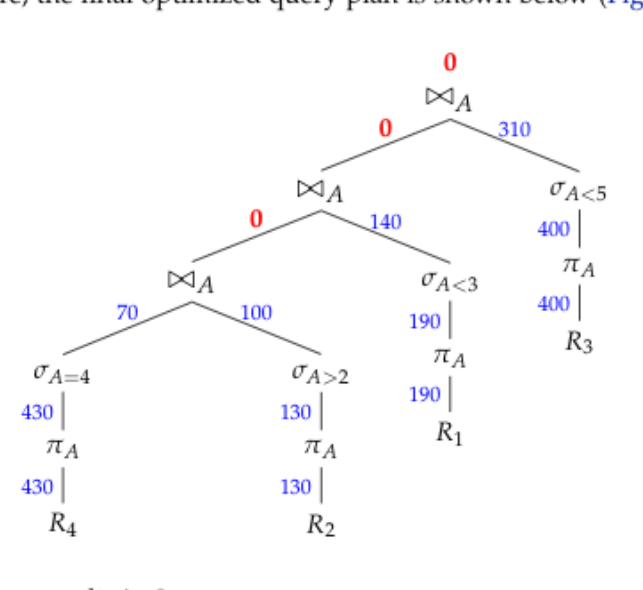


Figure 2: Cost-Optimized Query Plan

The query results in 0 rows.

Question 4

Question 4. Answer the following

a. A non-serializable schedule can still maintain database consistency. Give an example with two transactions having both read and write operations to illustrate this. Show the precedence graph.

b. Assume the database relation R is just created and loaded with rows. Take two transactions with read and write operations to show:

i. Time stamp ordering and locking concurrency algorithms generate the same schedule.

ii. Time stamp ordering and locking concurrency control algorithms generate different schedules.

c. Consider R with attribute A having domain values $\{1, 2, \dots, 9\}$ and let the number of rows with attribute A having values $i = 1, 2, \dots 9$, as $\{n_1, n_2, \dots, n_9\}$, respectively, all n_i's are greater than bfr of relation R. Consider the update –

```
BEGIN TRANSACTION
    UPDATE R SET A = A + 1;
END TRANSACTION
```

If the system crashes right after the update of A from 7 to 8, state what the recovery algorithm does at the restart, assuming a no-fix and no-flush strategy to bring R to the consistent state. State two ways through which successful updates need not be redone when the system recovers.

Answer. a. Let T_1 and T_2 be the two transactions:

$$T_1 = R_1(x)\, W_1(x = 5)\, R_1(x)\, C_1$$
$$T_2 = R_2(x)\, W_2(x = 5)\, C_2$$

where $W_i(x = 5)$ indicates that the transactions write value 5 for **x**. Let the schedule S (also shown in Table 3) be

$$S = R_1(x)\, W_1(x = 5)\, R_2(x)\, W_2(x = 5)\, R_1(x)\, C_1\, C_2$$

T_1	T_2
$R_1(x)$	
$W_1(x = 5)$	
	$R_2(x)$
	$W_2(x = 5)$
$R_1(x)$	
C_1	
	C_2

Table 3: Schedule S

This schedule is not serializable as evidenced by the precedence graph (Fig. 3). However, since both transactions write the same value for **x**, the schedule S is equivalent to both serial schedules $S' = T_1\, T_2$ and $S'' = T_2\, T_1$. Hence, the database maintains consistency.

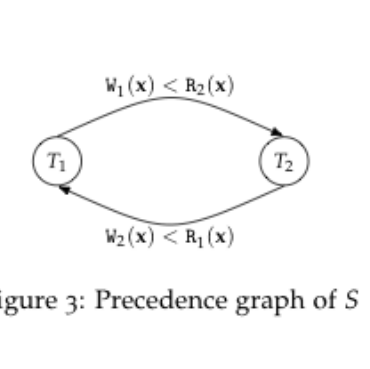


Figure 3: Precedence graph of S

b. Let the two transactions be T_1 and T_2

$$T_1 = R_1(z)\, W_1(z)\, R_1(x)\, W_1(x)\, C_1$$
$$T_2 = R_2(y)\, R_2(x)\, W_2(x)\, W_2(y)\, C_2$$

Table 4 shows a schedule generated by the timestamp ordering protocol. The table shows the read and write timestamps for data value **x** that is common to both transactions.

T_1	T_2	TS_values
$R_1(z)$		
$W_1(z)$		
	$R_2(y)$	
$R_1(x)$		$read\_TS(x) = 1$
$W_1(x)$		$write\_TS(x) = 1$
$COMMIT_1$		
	$R_2(x)$	$read\_TS(x) = 2$
	$W_2(x)$	$write\_TS(x) = 2$
	$W_2(y)$	
	$COMMIT_2$	

Table 4: A schedule generated by timestamp ordering protocol equivalent to $S = T_1\, T_2$.

Fig. 4 shows two schedules generated by the 2PL protocol. The first schedule is the same as that generated by the timestamp ordering protocol (Table 4). Whereas, the second schedule is different. The two schedules are also equivalent to different serial schedules.

c. If a transaction hasn't been committed, then all the operations of the transaction must be undone. The Write-Ahead Log (WAL) protocol includes a before image (the UNDO portion) on the log when pages have been updated in the stable storage before committing. Therefore, if there is no "end_transaction" log entry, then the no-fix/no-flush strategy performs an UNDO operation with the before-image on the log.

One way of **reducing** the number of operations that need to be redone is by increasing the granularity of the transaction. For instance, the query can be broken into the following pieces

```
UPDATE R SET A = A + 1
WHERE A = X;
```

for each X in the domain. Commits are performed after each piece. Another way is to force a commit after each page is updated. To aid the performance of both these approaches, check-

T_1	T_2
$LOCK(z)$	
$R_1(z)$	
$W_1(z)$	
	$LOCK(y)$
	$R_2(y)$
$LOCK(x)$	
$R_1(x)$	
$W_1(x)$	
$UNLOCK(x)$	
$UNLOCK(z)$	
$COMMIT_1$	
	$LOCK(x)$
	$R_2(x)$
	$W_2(x)$
	$UNLOCK(x)$
	$W_2(y)$
	$UNLOCK(y)$
	$COMMIT_2$

(a) Equivalent to $S = T_1\, T_2$

T_1	T_2
$LOCK(z)$	
$R_1(z)$	
$W_1(z)$	
	$LOCK(y)$
	$R_2(y)$
	$LOCK(x)$
	$R_2(x)$
	$W_2(x)$
	$UNLOCK(x)$
$LOCK(x)$	
$R_1(x)$	
$W_1(x)$	
$UNLOCK(x)$	
$UNLOCK(z)$	
	$W_2(y)$
	$UNLOCK(y)$
	$COMMIT_2$
$COMMIT_1$	

(b) Equivalent to $S = T_2\, T_1$

Figure 4: Schedules generated by the 2PL protocol. The difference occurs based on which transaction receives the lock on **x** first.

pointing can be implementing to reduce the size of the log accessed during recovery.

References

R Elmasri and Shamkant B Navathe. *Fundamentals of Database Systems*. Springer, 2000. Accessed on 19/11/2022; Link.