CSCE-608 Database Systems

Spring 2025

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: 845-4259
Email: chen@cse.tamu.edu
Office Hours: MW 1:30 pm–3:00 pm

TA: Shahin John John Stella
Office: PETR 445
Phone: (979) 575-5523
Email: ivin-98@tamu.edu
Office Hours: TR 11:00 am–12:00 pm

Assignment #3 Solutions

1. Below are the statistics for four relations:

W(a, b): T(W) = 100, V (W, a) = 20, V (W, b) = 60;
X(b, c): T(X) = 200, V (X, b) = 50, V (X, c) = 100;
Y(c, d): T(Y) = 300, V (Y, c) = 50, V (Y, d) = 50;
Z(d, e): T(Z) = 400, V (Z, d) = 40, V (Z, e) = 100.

Give the dynamic programming table entries that evaluates all join orders allowing:

a) All trees.

Answer. Below is the dynamic programming table entries for all trees:

Relation Subset	Size	Cost	V_a	V_b	V_c	V_d	V_e	Best Plan
{W,X}	334	0	20	50	100			W⋈X
{W,Y}	30,000	0	20	60	50	50		W⋈Y
{W,Z}	40,000	0	20	60		40	100	W⋈Z
{X,Y}	600	0		50	50	50		X⋈Y
{X,Z}	80,000	0		50	100	40	100	X⋈Z
{Y,Z}	2,400	0			50	40	100	Y⋈Z
{W,X,Y}	1,000	334	20	50	50	50		(W⋈X)⋈Y
{W,X, Z}	133,600	334	20	50	100	40	100	(W⋈X)⋈Z
{W,Y,Z}	240,000	2,400	20	60	50	40	100	W⋈(Y⋈Z)
{X,Y,Z}	4,800	600		50	50	40	100	(X⋈Y)⋈Z
{W,X,Y,Z}	8,000	1,334						((W⋈X)⋈Y)⋈ Z

b) left-deep trees only.

Answer. Below is the dynamic programming table entries for left-deep trees:

Relation Subset	Size	Cost	V_a	V_b	V_c	V_d	V_e	Best Plan
{W,X}	334	0	20	50	100			W⋈X
{W,Y}	30,000	0	20	60	50	50		W⋈Y
{W,Z}	40,000	0	20	60		40	100	W⋈Z
{X,Y}	600	0		50	50	50		X⋈Y
{X,Z}	80,000	0		50	100	40	100	X⋈Z
{Y,Z}	2,400	0			50	40	100	Y⋈Z
{W,X,Y}	1,000	334	20	50	50	50		(W⋈X)⋈Y
{W,X, Z}	133,600	334	20	50	100	40	100	(W⋈X)⋈Z
{W,Y,Z}	240,000	2,400	20	60	50	40	100	(Y⋈Z)⋈W
{X,Y,Z}	4,800	600		50	50	40	100	(X⋈Y)⋈Z
{W,X,Y,Z}	8,000	1,334						((W⋈X)⋈Y)⋈ Z

Discussion. In this particular example, the dynamic programming table entries for the all-tree model and for the left-deep-tree model happen to be the same, except:

1. For the left-deep-tree model, the best plan must be a left-deep tree while the best plan for the all-tree model may not be a left-deep tree. This difference can be seen for the subset {W,Y,Z} in the above two tables.

2. For the subset {W,X,Y,Z}, the all-tree model needs to consider all 7 non-trivial partitions of the subset, i.e., {(W,X), (Y,Z)}, {(W,Y), (X,Z)}, {(W,Z), (X,Y)}, {(W,Y,Z), (X)}, {(W,X,Z), (Y)}, {(W,X,Y), (Z)}, and {(W), (X,Y,Z)}, while the left-deep-tree model only needs to consider the four partitions that can make left-deep trees, i.e., {(W,Y,Z), (X)}, {(W,X,Z), (Y)}, {(W,X,Y), (Z)}, and {(X,Y,Z), (W)}, although for this particular example, as shown in the above tables, the left-deep tree ((W⋈X)⋈Y)⋈Z happens to give the best plan not only for all left-deep trees but also for all trees.

2. Suppose that we have the following key values:

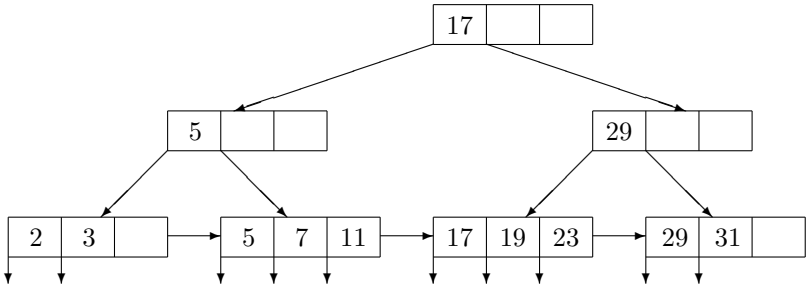
$$29, 5, 7, 17, 19, 31, 2, 23, 11, 3.$$

Construct B+-trees for these keys where the order of the B+-tree is

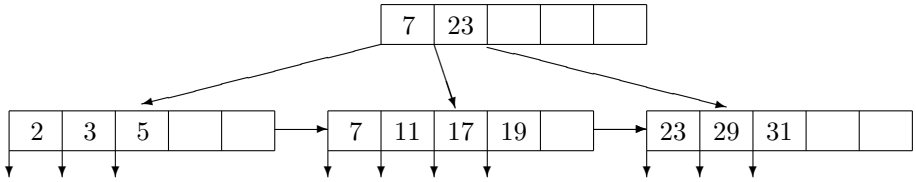
Solution.

Since the number of keys in each node can vary, there can be many B+trees that can satisfy the given conditions (you do not have to use the insertion operation). The following are possible solutions.

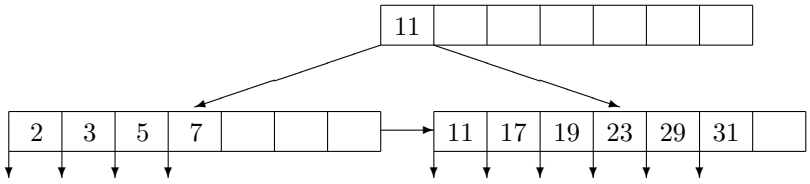
(a) Order 3



(b) Order 5



(c) Order 7



3. Discuss how to execute a deletion operation in an extensible hash table. What are the advantages and disadvantages of restructing the table if its smaller size after deletion allows for compression of certain blocks?

Solution. Below is the algorithm for deletion in extensible hash-table:

input: a tuple t with search key x
\\ h is the hash-function, H is the directory, i is the current hash-index.

1. m = the first i bits of $h(x)$
2. let the block with the address $H[m]$ be B
3. **IF** t is in B **THEN** delete t from B **ELSE** return ("t not found")
4. let j be the block index of B
5. **WHILE** B has a buddy block B' mergeble with B, **DO**
   move all tuples in B' to B; free block B'
   set the block index of B to $j = j$–1
   m_{j-1} = the first $j-1$ bits of m_j
   Let all $H[m_{j-1}{}^{**}]$ point to B
6. **IF** the largest block index j_{max} is smaller than i, **THEN**
   construct a new directory H_0 of size $2^{j_{max}}$
   let the hash index $i = j_{max}$
   FOR each string m_0 of j_{max} bits **DO** let $H_0[m_0]$ = any $H[m_0{}^{**}]$

The advantage of merging mergeble blocks and restructuring the hash table is that the process frees up space whenever possible, thus, has better memory utilization. In particular, when the hash directory is stored in the main memory, reducing the size of the hash directory can significantly improve main memory utilization.

On the other hand, this makes the deletion operation in extensible hash structure less efficient:
(1) after the deletion, we need to check if the buddy block is mergeble, which costs extra disk I/O's;
(2) restructuring the entire hash table is very expensive; and
(3) in the worst cases, repeated merging/splitting of blocks due to repeated deletions/insertions can cause repeatedly restructuring the hash directory, which can significantly hurt the performance of the system.

4. Discuss the necessary changes to insertion, deletion, and lookup algorithms for linear hash tables if the search keys are not unique

Solution. Following are the changes in the algorithms for linear hash table if the search keys are not unique.

1. **Insertion:** If the search keys are unique, then before we insert, we need to check if the key is already in the bucket. In case multiple copies of the say key are allowed, we do not need to check the uniqueness when we insert.

2. **Deletion:** If search keys are unique, the deletion operation can return immediately after it finds a tuple with that key and deletes the tuple. On the other hand, if multiple tuples can have the same key, the deletion operation has to search all blocks in the bucket and makes sure that all tuples with that key are deleted.

3. **Lookup:** Similar to the deletion operation, when multiple tuples are allowed to have the same search key, the lookup opeation has to search all blocks in the bucket to find all these tuples.

5. Suppose that we want to compute the set union of two relations S and R, where S and R are both sets and each takes $1,000$ blocks. Suppose that the main memory has $M = 200$ blocks.

(1) How many disk I/O's do you need if you use the two-pass sort-based algorithm discussed in class (and in the textbook)?

(2) Can you modify the two-pass sort-based algorithm to save some disk I/O's? (hint: use the idea of the hybrid algorithm as discussed in the hash-based algorithm).

Solution. (1) As given, we have $B(S) = B(R) = 1,000$, and $M = 200$. Thus, the memory condition $M = 200 \geq \sqrt{B(S) + B(R)} \approx 45$ is satisfied and we can apply the two-pass sort-based algorithm. As given in class, the number of disk I/O's needed for the two-pass sort-based set union algorithm is $3(B(S) + B(R)) = 6,000$.

(2) In order to save some disk I/O's, we can use the idea of hybrid algorithms. For this, we keep a sublist in the main memory (as we discussed in class, keeping a single sublist in the main memory maximizes the number of saved disk I/O's). The most interesting question is how we decide the size of the sublists. Suppose each sublist is of x blocks, where x is to be decided later. Then we can use the following algorithm:

1. break relations R and S into sorted sublists of size x, and write all of them, except the last sublist l_S of S, back to disk;
2. read one block from each sublist of R and S in disk into main memory;
3. repeatedly, move the smallest tuple among all the sublists (including l_S) to the output block (if there are two such smallest tuples, then just move one to the output block and disregard the other). If a block becomes empty, then read the next block from the corresponding sublist in disk into main memory.

The number of disk I/O's saved by this algorithm is $2x$ (for 1 read and 1 write for the blocks in the sublist l_S), compared to the standard two-pass sort-based algorithm as given in (1). Thus, the total number of disk I/O's spent by this algorithm is $6,000 - 2x$.

To make this algorithm work, certain conditions must be satisfied. Because the sublist size is x, each of the relations R and S has $1000/x$ sublists. Now since in the 2nd phase, we need in the main memory x blocks for the sublist l_S, and one block for each of the rest sublists of R and S, the main memory size must satisfy

$$M \geq x + 2 \cdot 1000/x - 1.$$

For example, if we let $x = 100$, then $x + 2 \cdot 1000/x - 1 = 119 < M = 200$, so we have enough main memory space for this value of x, and the total number of disk I/O's spent by this hybrid algorithm is $6,000 - 2x = 5,800$.

To maximize the saved disk I/O's, we solve the equation $M = x + 2 \cdot 1000/x - 1$, which gives $x = 190.\cdots$. Thus, $x = 190$ is the largest integer that can satisfy the condition above. For this value of x, the total number of disk I/O's spent by the hybrid algorithm is $6,000 - 2 \cdot 190 = 5,620$.

Remark 1. In the above discussion, we did not count the block used as the output block in main memory in phase 2 (note that in sort-based algorithm, we do not need an additional input block). If we also count the output block, then the condition becomes $M \geq x + 2 \cdot 1000/x$, and the best vlue of x is 189.

Remark 2. It is acceptable if your solution is not the optimal but you tried some meaningful values for x, such as $x = 100$ or $x = 150$. On the other hand, you must present an analysis to verify that you have sufficient main memory space for the value of x you chose.