Written Assignment 1 Reference Solutions

Assigned: Feb 17, 2022

Part I. DBMS Architecture & Storage Devices

1. Briefly explain (logical and physical) data independence in your own words. Give examples where logical and physical data independence (one example for each) is useful.

Solution. Data independence is a property that application programs are insulated from changes on how the data are structured and stored. In particular, logical data independence shield users from changes in the logical structure of the data or changes in the choice of relations to be stored. For example, user can simply ask for the GPA of a student while the student's grades for all courses is stored and GPA is only calculated upon user queries through views. On the other hand, physical data independence insulate users from changes in physical details like data layout or index structures. For instance, the DBMS can choose one particular attribute to order all the records to optimize particular queries, while the user should be able to ask any valid query in the same way on the data without knowing the underlying data layout.

2. What are main differences between embedded and client-server DBMS interfaces? Explain how could a DMBS deliver query results larger than main-memory?

Solution. Embedded DBMS interface allow application code to operate directly on physical data structures through a library within the same process. With an embedded interface, users directly make function calls to query or make changes to data, which can be more performant. However, users are usually in charge of other issues like concurrency control. In contrast, client-server DBMS interface receives user's request through a socket. The database server runs as a different process which manages its resources and all incoming requests independently. With such an interface, the users usually do not have to worry about issues like resource management and concurrency control.

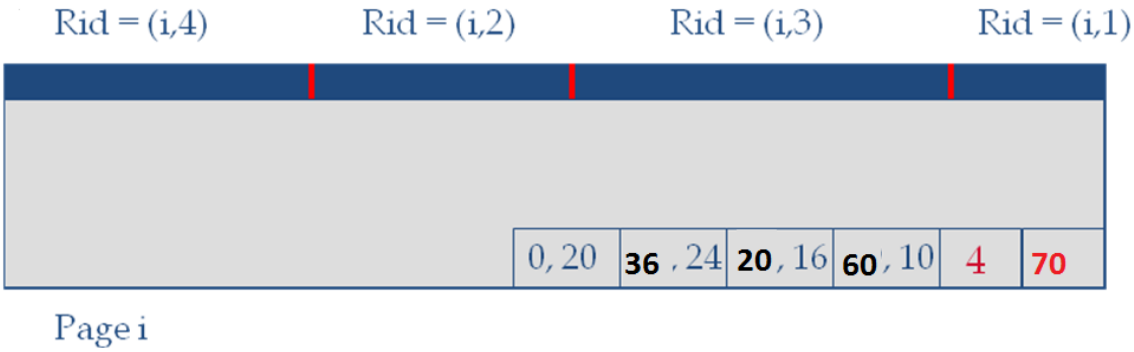
When responding to users, the DBMS will return a cursor that allows user to iterate through all results exactly once. With the help of this design, the client only need to own enough memory to hold one record, thus can handle query results larger than memory.

3. Explain why the write performance of an SSD will degrade over time.

Solution. In a flash-based SSD, each cell will be worn out after a certain amount of erase cycles. To make sure the device more durable, SSDs adopted a write-ahead log-structured write pattern to achieve wear leveling. When a page write happens, the SSD will simply append it to the end of log, and change the mapping of corresponding logical page. The Flash Translation Layer (FTL) is in charge of mapping logical pages to their physical location. Over time, SSD will run out of erased out blocks, and have to claim free space by doing garbage collection. This process will compact living pages (which still has a valid mapping) across multiple blocks into one so as to make a fresh erased block for incoming writes. As the disk is getting full, it is harder to make a fresh erased block before any writing can be done. Therefore, SSD write performance will degrade over time.

Part II. Data Organization & Buffer Manager

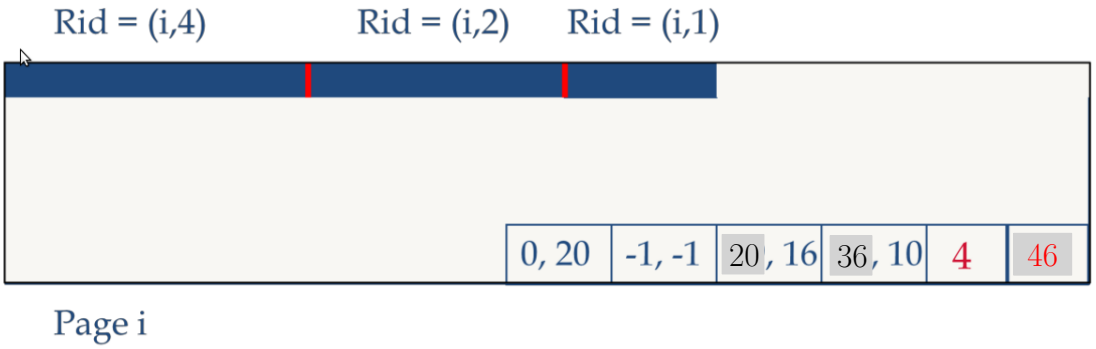
1. Suppose we store the pointer (or starting address of a record) in a page by using the offset to the beginning of the page (i.e., the first byte in a page has an offset of 0, the 2nd byte in a page has an offset of 1, and so on and so forth). Each slot entry is of the format: (record offset, record length). Consider the variable-length record page organization instance below:



(1) Show the content of the page after the deletion of Rid=(i,3)

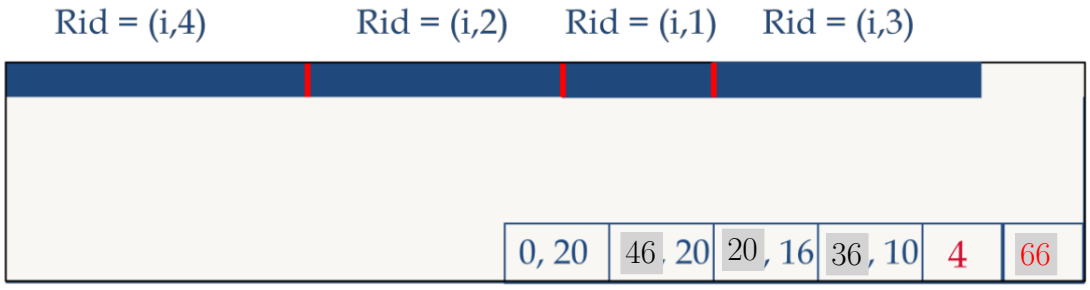
(2) Now, after step (1) has been completed, show the content of the page after the insertion of a new record to this page with 20 bytes. And what's the Rid of this new record after the insertion is done?

Solution. (1) The page layout after the deletion of Rid=(i,3) shows as below:



Note that leave the hole for record Rid = (i,3) without doing compaction is also a valid solution.

2) After the insertion of the new record (using Rid = (i, 2)), the page layout shows as below:



Note that appending the new record to the end and/or having a new slot ID (Rid = (i, 5)) are also valid.

2. In this exercise, you will compare the LRU (least recent used) and Clock eviction policy for a given workload. Here are the assumptions you must make:

- Assume there are four page slots your buffer manager must manage: P1, P2, P3, and P4.
- All four slots are empty to start. The clock hand for clock algorithm starts from P1 and go forward.
- When the buffer pool has unused slots (such as at the beginning, when all four slots are empty), it will put newly read data in the leftmost empty slot (e.g., if slots 2 and 3 are free, it will use slot 2).
- The pages to be read from disk are labeled A through G.
- For each access the page is pinned, and then immediately unpinned.

Below are two tables for describing the contents of the buffer pool at each step. A page is read at the beginning of each step and unpinned at the end. You should record, in the table, the contents of each Buffer Page after the new page has been read in.

LRU					
Step	Page Read	Buffer Frame			
		P1	P2	P3	P4
1	G	G			
2	F	G	F		
3	E	G	F	E	
4	D	G	F	E	D
5	G	G	F	E	D
6	C	G	C	E	D
7	F	G	C	F	D
8	E	G	C	F	E
9	D	D	C	F	E
10	C	D	C	F	E
11	D	D	C	F	E
12	B	D	C	B	E
13	A	D	C	B	A
14	D	D	C	B	A
15	F	D	F	B	A
Total num of miss:		11			

Clock					
Time	Page Read	Buffer Frame			
		P1	P2	P3	P4
1	G	G			
2	F	G	F		
3	E	G	F	E	
4	D	G	F	E	D
5	G	G	F	E	D
6	C	C	F	E	D
7	F	C	F	E	D
8	E	C	F	E	D
9	D	C	F	E	D
10	C	C	F	E	D
11	D	C	F	E	D
12	B	C	B	E	D
13	A	C	B	A	D
14	D	C	B	A	D
15	F	F	B	A	D
Total num of miss:		8			

Part III. Indexes

1. Suppose that the page size is 1K bytes, and the each (fixed-sized) record size is 100 bytes. You can ignore meta information like page header and bitmaps in the following calculation. Given 1 million records in a table, answering the following questions:

(1) Suppose the table is stored in a heapfile, how many pages are in this file?

(2) We will build an unclustered alternative 2 B+ tree on this table over an attribute A, how many index entries are contained in the leaf level?

(3) Suppose A is integer type of 4 bytes, what's the size of an index entry (assuming a rid/pid is 4 bytes)?

(4) Suppose each leaf level page is 70% filled, how many leaf level pages are needed?

(5) What is the height of the B+ tree (suppose each index level page is also 70% filled)?

(6) Given a query: find all index entries with $100<A<500$, suppose there are 300 matching data entries for this query, what's the IO cost of this query with the B+-Tree described as (5)?

(7) Given a query: find all records with $100<A<500$, suppose there are 300 matching records for this query, what's the IO cost of this query with the B+-Tree described as (5)?

(8) With this unclustered index, what's the IO cost of inserting and deleting a record?

(9) Answer (3)-(8) again suppose now we are using a clustered alternative 1 index over A.

Solution.

(1) Assume 100% utilization, it will be $10^6/\lfloor(1024/100)\rfloor = 10^5$ pages. Assume 70% utilization, it will be $\lceil 10^6/(\lfloor 1024/100\rfloor \times 0.7)\rceil = 142,858$.

(2) There will be 10^6 index entries contained in the leaf node.

(3) The size of an index entry is $4+4=8$.

(4) Each leaf can hold 89 entries

$$\lfloor 1024/8 * 0.7\rfloor = 89$$

We need 11236 leaf pages

$$\lceil 10^6/89\rceil = 11236$$

(5) Since each internal index node can hold 89 entries (since $\lfloor((\lfloor(1024-4)/8\rfloor+1)*0.7\rfloor = 89$), we need three indirections

$$\lceil log_{89}(11236)\rceil = 3$$

Therefore the height of B+-tree is $3+1=4$ (Note that 1 is for the leaf level).

(6) The IO cost is $3+\lceil 300/89\rceil = 7$. Since 3 IOs on internal nodes in B+-Tree, then $\lceil 300/89\rceil = 4$ IOs on the leaf level.

(7) The IO cost is $3+\lceil 300/89\rceil + 300 = 307$. Since 3 IOs on internal nodes in B+-Tree, 4 IOs on the leaf level, then 300 indirect IOs to retrieve the record.

(8) The cost of inserting or deleting a record will include the cost of updating the index and the cost of updating the heap file. For index updates, traversing the tree to find the index entry needs 4 IOs (reads), insert/delete the index entry on the leaf node needs 1 IO (write). Then insert/delete the record in heap file need 1 IO to read out the page and 1 IO to write it back. So in total, insert/delete a record in the table will need $4+1+1+1=7$ IOs.

(9) Since it is an alternative 1 clustered index, the index entry size (question 3) is going to be $4+100=104$. Each leaf node can hold $\lfloor 1024/104*0.7\rfloor = 6$ index entries, thus we need $\lceil 10^6/6\rceil = 166667$ leaf pages (question 4). Each internal node index node can hold $\lfloor((\lfloor(1024-4)/8\rfloor+1)*0.7\rfloor = 89$ entries. Thus, the tree height (question 5) is $\lceil log_{89}(166667)\rceil+1=4$. Since it is a clustered alternative 1 index, the index entry is also the record. Thus, the number of IO to find all index entries or records (question 6 and 7) is $3+\lceil 300/6\rceil = 53$. The cost of inserting or deleting a record (question 8) will need $4+1$ IOs since we only need to find the record and change it in place at the leaf level.

2. Suppose that we are using extensible hashing on a file that contains records with the following search-key values: (2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, 9208).

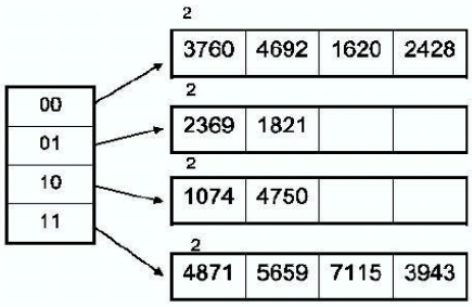
Load these records into a file in the given order using extensible hashing. Assume that every bucket can store up to four values. Show the structure of the directory every 3 insertions, and the global and local depths. Use the hash function: h(k) = k mod 128 and then apply the extensible hashing index.

Solution.

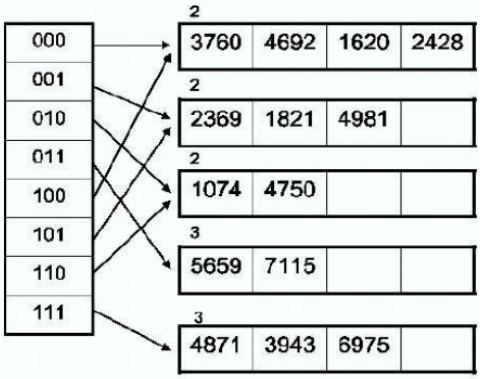
em 2

2369	1000001	7115	1001011
3760	0110000	1620	1010100
4692	1010100	2428	1111100
4871	0000111	3943	1100111
5659	0011011	4750	0001110
1821	0011101	6975	0111111
1074	0110010	4981	1110101
		9208	1111000

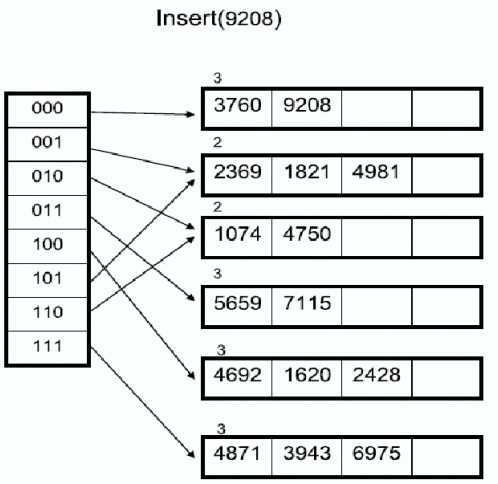
Insert(2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750)



Insert(6975, 4981)



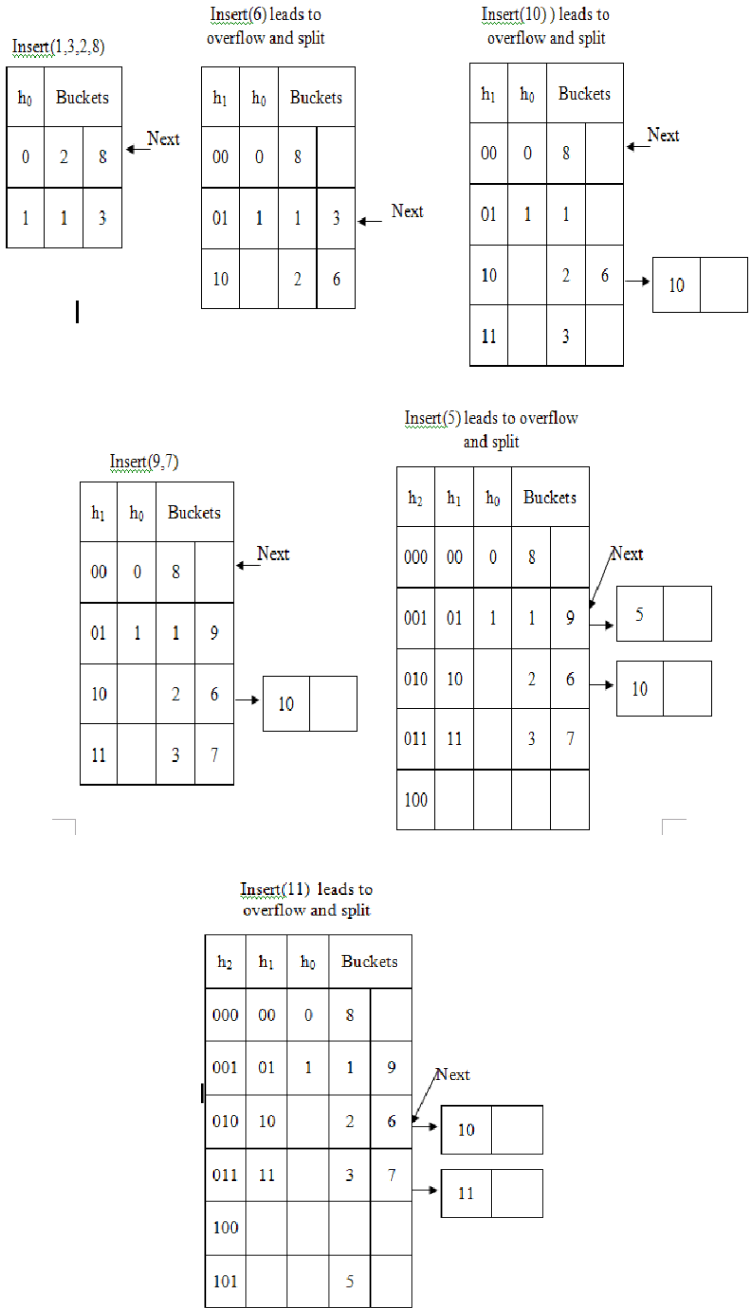
Insert(9208)



3. Suppose we are using linear hashing as discussed in class. Assume two records per bucket and the hash function $h(x) = x$. The intermediate hash functions $h_i(x)$, are given by the last i bits of $h(x)$. Assume a split is initiated whenever an overflow bucket is created. Starting with an empty hash file with the hash function $h_1(x)$ and two buckets show the significant intermediate steps (overflows and splits) when keys are inserted in the order given below:

$$(1,3,2,8,6,10,9,7,5,11)$$

Solution.



Part IV. Relational Algebra

1. Consider a database with the following schema (when more than one attribute is underlined, the combination of them is the primary key):

Customer (cname, address)
Dealership (dname, address)
Car (model, manufacturer)
Visits (cname, dname, times_a_year)
Likes (cname, model)
Serves (dname, model, price)

Write the following queries in **relational algebra**:

(1) Find all customers who has visited "Ford Provo" ("Ford Provo is a dealership name).

(2) Find all dealerships that serve both Toyota and Ford.

(3) Find all dealerships that serve at least one of the car models John likes for no more than $10, 000.

(4) For each dealership, find all car models served at this dealership that are liked by none of the customers who visit that dealership.

(5) Find all customers who visit *only* those dealerships that serve some car models they like.

(6) Find all customers who have visited *every* dealership that serves some car models they like.

(7) Find those customers who like all cars Mike likes.

(8) Find those customers who like exactly the same set of cars as Mike does.

(9) For each car model, find the dealership(s) that serves it at the lowest price.

(10) Find the dealership name(s) visited by all customers from SLC (SLC is an address).

Solution. Note: the answers provided below are not necessarily the only ones that work.

(1) $\pi_{cname}(\sigma_{dname=\text{Ford Provo}} Visits)$;

(2) $\pi_{dname}\sigma_{model=Toyota}(Serves) \bigcap \pi_{dname}\sigma_{model=Ford}(Serves)$;

(3) $\pi_{dname}(\sigma_{price \leq 10000}(Serves \bowtie (\pi_{model}(\sigma_{cname=John} Likes))))$

(4) $\pi_{dname,model} Serves - \pi_{dname,model}((\pi_{cname,dname} Visits) \bowtie Likes)$

(5) $\pi_{cname} Visits - \pi_{cname}(\pi_{cname,dname}(Visits) - \pi_{cname,dname}(Likes \bowtie Serves))$

(6) $\pi_{cname} Visits - \pi_{cname}(\pi_{cname,dname}(Likes \bowtie Serves) - \pi_{cname,dname} Visits)$

(7) $\pi_{cname} Customer - \pi_{cname}((Customer \times \pi_{model}(\sigma_{cname=Mark} Likes)) - Likes)$

If you use division (as described in Part IV.2), it can be get from $Likes / \pi_{model}(\sigma_{cname=Mark} Likes)$

(8) $Likes/\pi_{model}(\sigma_{cname=Mark}Likes) - \pi_{cname}(Likes - \pi_{cname}Customer \times \pi_{model}(\sigma_{cname=Mark}Likes))$

You can replace the first division clause by any solution of (7).

(9) $\rho(S1, Serves)$, $\rho(S2, Serves)$,
$\pi_{dname,model}Serves - \pi_{S1.dname,S1.model}(S1 \bowtie_{S1.model=S2.model \wedge S1.price>S2.price} S2)$

(10) $\pi_{dname}Dealership - \pi_{dname}(\pi_{cname,dname}(\sigma_{address="SLC"}Customer \times Dealership) - \pi_{cname,dname}Visits)$

If you use division, it can be get from $\pi_{cname,dname}Visits/(\pi_{cname}(\sigma_{address='SLC'}Customer))$

2. There is an alternative relational algebra operator division ($/$). Consider two relation R and S whose schemas are $(A_1, \cdots, A_m, B_1, \cdots, B_n)$ and $(B_1, \cdots, B_n)$ respectively. R/S returns a relation of schema $(A_1, \cdots, A_m)$ where each record p is associated with all tuple s of S in R (i.e., you can always find (p, s) in R). Use the relational algebra operator we discuss in class to express R/S.

Solution. The following relational algebra clause (uses only the core operators) can be used to express R/S:

$$\pi_{A_1,A_2,\cdots,A_m}(R) - \pi_{A_1,A_2,\cdots,A_m}((\pi_{A_1,A_2,\cdots,A_m}(R) \times S) - R)$$