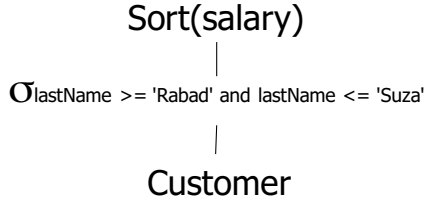
Data Management – AA 2016/17 – exam of 12/01/2017
Solutions

Solution 1
At the beginning of pass 0, we have 200 free frames in the buffer, and therefore we sort 200 pages of `R` in the buffer, and we write the corresponding first run of 200 pages. After such writing, the number of free frames is halved, and therefore we have 100 free buffer frames left. We then sort 100 pages of the remaining 175 pages of `R`, and we write the corresponding second run of 100 pages. After such writing, we have 50 free buffer frames left. We then sort 50 pages of the remaining 75 pages of `R`, and we write the corresponding third run of 50 pages. After such writing, we have 25 free buffer frames left. We then sort the last 25 pages of `R`, and we write the corresponding fourth run of 25 pages. After such writing, we have 12 free buffer frames left. Now pass 0 is completed, and, since we have 4 sorted runs, we can simply perform pass 1 of the algorithm, by using 4 of the 12 free buffer frames for merging the 4 runs and obtaining the sorted file constituting the result. The resulting algorithm has the same complexity of the two-pass algorithm, and therefore the number of page accesses required by the algorithm is $3 \times 375 = 1.125$ (as usual, we ignore the cost of writing the final result).

Solution 2
Since the most frequent query on `Customer` asks for all customers whose last name falls into a given range, a good method for storing the relation is a sorted file with sorting key `lastName`, with an associated clustering, sparse tree-based index on search key `lastName`. Indeed, it is well known that range queries are well supported by a clustering B^+-tree index.

The logical plan of query Q is as follows:



Having the clustering B^+-tree index, we can answer the query by using the index for the "selection" operator, thus finding the first page of `Customer` with the tuples satisfying the `where` condition, exploting the fact that the index is clustering, and then sorting such pages on `salary` to get the final result. Note that the result of the "selection" operator is not materialized. Rather, a pipeline approach is used to pass the result of the "selection" operator and the sorting operator.

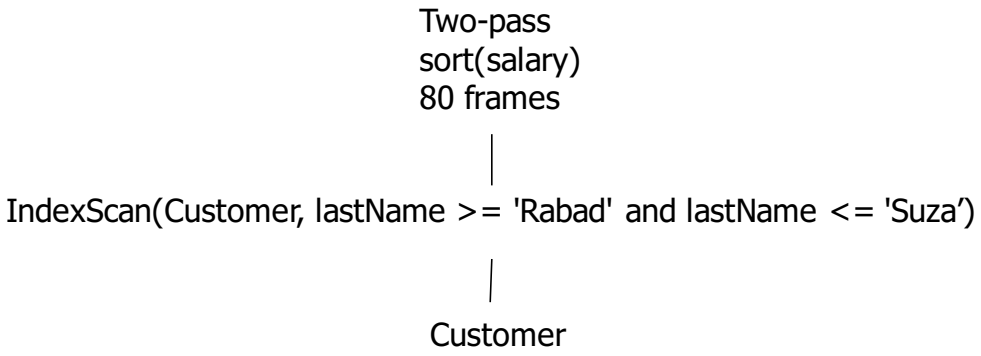
As for the cost of the algorithm, we have to compute the number of page accesses both for the selection operation, and for the sorting operation.

- **Selection**. Since each value or pointer occupies 100 Bytes, every tuple has 4 values, and since each page has space for 4000 Byte, it follows that every tuple occupies 400 Bytes, which means that each page has space for 10 tuples of `Customer`, and the relation is stored in 65.000 pages. Also, we have that each data entry requires 200 Bytes, and each page has space for 20 data entries. Taking into account the 67% occupancy rule

for the leaves of the tree-based index, we have that each leaf contains 13 data entries. Also, since each page has space for 20 index entries, we can asssume the value 15 for the fan-out. Note that, since the index is sparse, it stores one value of `lastName` for each page of `Customer`, and therefore the number of leaves are 65.000 / 13 = 5.000. It follows that reaching the right leaf requires $\log_{15}$ 5.000 = 4 page accesse.

- **Sorting**. After reaching the right leaf, we then have to sort all the pages with the qualifying records. Since we need to sort all the last names starting with two letters ('R' and 'S'), and we know that the last names of customers are equally distributed on the first letter over the 26 letters of the alphabet, the number of qualifying records are (650.000/26) × 2 = 50.000, stored in 50.000/10 + 1 =5.001 pages. The records of such pages must be sorted having 80 buffer frames available. Since 5.001 < 80 × 80, we can use the two-pass merge-sort algorithm, with a cost of 3 × 5001 = 15.003 page accesses.

The corresponding physical plan is therefore as follows:



and the total cost in terms of page accesses is 4 + 15.003 = 15.007 (as usual, we ignore the cost of writing the final result).

Solution 3

The cost of the block nested-loop algorithm, considering `V` as the outer relation, is $P(\texttt{V}) + P(\texttt{V}) \times P(\texttt{S})/(M-2)$, where $P(\texttt{V})$ is the number of pages of relation `V`, and $P(\texttt{S})$ is the number of pages of relation `S`.

The cost of the index-based algorithm is $P(\texttt{S}) + TP(\texttt{S}) \times 2$, where $TP(\texttt{S})$ is the number of tuples per page of relation `S`, and 2 is the cost of searching for the tuple with a given value of attribute `C` using the hash-based index. So, we have to see if there are values of M such that $P(\texttt{V}) + P(\texttt{V}) \times P(\texttt{S})/(M-2)$ is less than $P(\texttt{S}) + TP(\texttt{S}) \times 2$, i.e., such that (10.000 + 10.000 × 200.000) / $(M-2)$ is less than 200.000 + 40.000.000 × 2. It can be seen that this happens when $M > 27$.

Solution 4

The relation is stored in 500.000/50 = 10.000 pages. The availability of a good hash function on `owner` that distributes the tuples of `CAR` uniformly suggests the two-pass algorithm based on hashing. Indeed, we can execute such algorithm if the number of pages of the relation is less than or equal to $M \times (M-1)$, where M is the number of free buffer frames. In our

case $M \times (M-1) = 101 \times 100 = 10.100$, and therefore we can indeed use the two-pass algorithm based on hashing. Such algorithm uses the first pass to distribute, using the hash function on `owner`, the tuples of the relation in $M-1$ buckets, in such way that tuples with the same value of owners are in the same bucket. In the second pass, we treat each bucket in isolation. For each bucket we store in the buffer one tuple for each value of `owner`, and we accumulate the result (in this case, the count) while reading the pages of the bucket. After the processing of the bucket, we write the content of the buffer in the result. The cost is obviously $3 \times 10.000 = 30.000$ page accesses (as usual, we ignore the cost of writing the final result).

Solution 5

5.1 S is not a 2PL schedule, because transaction 1 should unlock x after the first read (as transaction 3 wants to write on x), and therefore, for S to follow the 2PL protocol, transaction 1 should acquire the exclusive lock on y (needed for the last action of S) before unlocking x. But if this happens, transaction 2 will not be able to write on y.

5.2 S is not view-serializable. Indeed, in all serial schedules where transaction 1 comes before transaction 2, $w_1(y)$ is not the final write on y, differently from S, and in all serial schedules where transaction 2 comes before transaction 1, $r_1(x)$ reads from $w_2(x)$, differently from S.

5.3 The behaviour of the timestamp-based scheduler when processing S is as follows:

$r_1(x) \rightarrow$ OK, rts(x)=1
$w_3(x) \rightarrow$ OK, wts(x)=3, cb(x)=`false`
$w_3(z) \rightarrow$ OK, wts(z)=3, cb(z)=`false`
$c_3 \rightarrow$ OK, wts-c(x)=3, cb(x)=`true`, wts-c(z)=3, cb(z)=`true`
$w_2(x) \rightarrow$ OK (Thomas rule)
$w_2(y) \rightarrow$ OK, wts(y)=2, cb(y)=`false`
$c_2 \rightarrow$ OK, wts-c(x)=2, cb(y)=`true`
$r_4(x) \rightarrow$ OK, rts(x)=4
$w_4(z) \rightarrow$ OK, wts(z)=4, cb(z)=`false`
$w_1(y) \rightarrow$ OK (Thomas rule)

5.4 S is strict, because whenever a transaction reads from another transaction, the latter has committed, and whenever a transaction writes on another transaction, the latter has committed.