External sorting

Exercise 1: Answer the following questions for each of these scenarios, assuming that our most general external sorting algorithm is used:

1. (a) A file with 10,000 pages and three available buffer pages.
2. (b) A file with 20,000 pages and five available buffer pages.
3. (c) A file with 2,000,000 pages and 17 available buffer pages.

1. How many runs will you produce in the first pass?
2. How many passes will it take to sort the file completely?
3. What is the total I/O cost of sorting the file?
4. How many buffer pages do you need to sort the file completely in just two passes?

Exercise 1 Solution: The answer to each question is given below.
1. In the first pass (Pass 0), $\lceil N/B \rceil$ runs of B pages each are produced, where N is the number of file pages and B is the number of available buffer pages:
(a) $\lceil 10000/3 \rceil = 3334$ sorted runs.
(b) $\lceil 20000/5 \rceil = 4000$ sorted runs.
(c) $\lceil 2000000/17 \rceil = 117648$ sorted runs.

2. The number of passes required to sort the file completely, including the initial sorting pass, is $\lceil \log_{B-1} N1 \rceil + 1$, where $N1 = \lceil N/B \rceil$ is the number of runs produced by Pass 0:

(a) $\lceil \log_{2} 3334 \rceil + 1 = 13$ passes. (b) $\lceil \log_{4} 4000 \rceil + 1 = 7$ passes. (c) $\lceil \log_{16} 117648 \rceil + 1 = 6$ passes.

3. Since each page is read and written once per pass, the total number of page I/Os for sorting the file is $2 * N * (\#passes)$:

(a) 2*10000*13 = 260000.
(b) 2*20000*7 = 280000.
(c) 2*2000000*6 = 24000000.

4. In Pass 0, $\lceil N/B \rceil$ runs are produced. In Pass 1, we must be able to merge this many runs; i.e., $B - 1 \geq \lceil N/B \rceil$. This implies that B must at least be large enough to satisfy $B * (B - 1) \geq N$; this can be used to guess at B, and the guess must be validated by checking the first inequality. Thus:

(a) With 10000 pages in the file, B = 101 satisfies both inequalities, B = 100 does not, so we need 101 buffer pages.
(b) With 20000 pages in the file, B = 142 satisfies both inequalities, B = 141 does not, so we need 142 buffer pages.

(c) With 2000000 pages in the file, B = 1415 satisfies both inequalities, B = 1414 does not, so we need 1415 buffer pages.

Query processing

Exercise 2: Consider processing the following SQL projection query:

SELECT DISTINCT E.title, E.ename FROM Executives E You are given the following information:

Executives has attributes ename, title, dname, and address; all are string fields of the same length.
The ename attribute is a candidate key.
The relation contains 10,000 pages.

There are 10 tuples per page. There are 10 buffer pages.

Consider the optimized version of the sorting-based projection algorithm: The initial sorting pass reads the input relation and creates sorted runs of tuples containing only attributes ename and title. Subsequent merging passes eliminate duplicates while merging the initial runs to obtain a single sorted result (as opposed to doing a separate pass to eliminate duplicates from a sorted result containing duplicates). The cost metric is the number of page I/Os unless otherwise noted, and the cost of writing out the result should be uniformly ignored.

1. How many sorted runs are produced in the first pass? What is the average length of these runs? What is the I/O cost of this sorting pass?
2. How many additional merge passes are required to compute the final result of the projection query? What is the I/O cost of these additional passes?
3. (a) Suppose that a clustered B+ tree index on *title* is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered?
   (b) Suppose that a clustered B+ tree index on ename is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered?

   c) Suppose that a clustered B+ tree index on <ename, title> is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered?

4. Suppose that the query is as follows:
   SELECT E.title, E.ename FROM Executives E

   That is, you are not required to do duplicate elimination. How would your answers to the previous questions change?

Exercise 2 solution:

1. The first pass will produce N1 = éN/Bù runs of B pages each, where B = 10 and N =

   5000 (we project the half of the relation, 10000/2). Therefore, the first pass produces 5000/10=500 sorted runs of 10 pages each, costing 15000 I/Os (i.e., reading 10000 and writing 5000 pages).

2. The number of additional merge passes is élogB-1N1ù, where B = 10 and N1 = 500. Therefore, élog9500ù = 3 more passes are required to merge the 500 runs. The first two merge passes need to read&write the number of pages, costing 2*2*5000=20000 I/Os. The third merge pass needs to read the number of pages, costing 5000 I/Os (we omit writing in the last pass, as it produces the final result).
3. Since ename is a candidate key, it is not necessary to perform duplicate checking for <title, ename> pairs.

(a) Using a clustered B+ tree index on title would reduce the cost to single scan or 10,000 I/Os. Using an unclustered index would cost up to 2500+100,000 (2500 is the I/O cost of scanning the leaf level of the B+ tree, where 2500 = 10 000/4 as the relation has 4 attributes; and 10000 * tuples per page). Thus, an unclustered index would not be cheaper.

(b) Using the clustered B+ tree on ename would be cheaper than sorting, in that the cost of using the B+ tree would be 10,000 I/Os. An unclustered index would require up to: 2500 (scan of index) + 10000 * tuples per page I/Os and thus be more expensive than sorting.

(c) Using a clustered B+ tree index on <ename, title> would also be more cost- effective than sorting. An unclustered B+ tree over the same attributes would allow an index-only scan, and would thus be just as economical as the clustered index. This method (both by clustered and unclustered) would cost around 5000 I/O's.

4. Knowing that duplicate elimination is not required, we can simply scan the relation and discard unwanted fields for each tuple. This is the best strategy except in the case that an index (clustered or unclustered) on <ename, title> is available; in this case, we can do an index-only scan. (Note that even with DISTINCT specified, no duplicates are actually present in the answer because ename is a candidate key. However, in the general case, a query optimizer may fail to recognize this pattern and omit the duplicate elimination step.)

Query optimization

Exercise 3: Consider a relation with this schema:

Employees (*eid:* **integer**, *ename:* **string**, *sal:* **integer**, *title:* **string**, *age:* **integer**)

Suppose that the following indexes, all using Alternative (2) for data entries, exist: a hash index on *eid*, a B+ tree index on *sal*, a hash index on *age*, and a clustered B+ tree index on <*age, sal*>. Each Employees record is 100 bytes long, and you can assume that each index data entry is 20 bytes long. The Employees relation contains 10,000 pages, and each data page contains 20 employee tuples. There are 11 buffer pages.

1. Consider each of the following selection conditions and, assuming that the reduction factor (RF) for each term that matches an index is 0.1, compute the cost of the most selective access path for retrieving all Employees tuples that satisfy the condition:
   (a) *sal* > 100
   (b) *age* = 25
   (c) *age* > 20
   (d) *eid* = 1000
   (e) *sal* > 200 ∧ *age* > 30
   (f) *sal* > 200 ∧ *age* = 20
   (g) *sal* > 200 ∧ *title* ='CFO'
   (h) *sal* > 200 ∧ *age* > 30 ∧ *title* ='CFO'
2. Suppose that, for each of the preceding selection conditions, you want to retrieve the average salary of qualifying tuples. For each selection condition, describe the least expensive evaluation method and state its cost.
3. Suppose that, for each of the preceding selection conditions, you want to compute the average salary for each *age* group. For each selection condition, describe the least expensive evaluation method and state its cost.
4. Suppose that, for each of the preceding selection conditions, you want to compute the average age for each *sal* level (i.e., group by *sal*). For each selection condition, describe the least expensive evaluation method and state its cost.
5. For each of the following selection conditions, describe the best evaluation method:
   (a) *sal* > 200 ∨ *age* = 20
   (b) *sal* > 200 ∨ *title* ='CFO'
   (c) *title* ='CFO' ∧ *ename* ='Joe'

Exercise 3 solution: The answer to each question is given below. Note that the unclustered index (on <sal>) has a height of $\log_{100}2000 = 2$ (as the index fanout - the number of index entries per page - is 100 and the number of leaf pages is 2000) and clustered (<*age, sal*>) has a height of $\log_{50}4000 = 3$ (as the index fanout is 50 and the number of leaf pages is 4000).

1.
(a) *sal* > 100 For this condition, a filescan is the best option, since a clustered index does not exist on *sal*. Using the unclustered index would accrue a cost of 2 $(lookup)$ + 10,000 $pages * \frac{20bytes}{100bytes} *$ 0.1 for the B+ index scan plus 10,000 $pages *$ 20 $tuples\ per\ page *$ 0.1 for the lookup = 20202, and would be inferior to the filescan cost of 10000.

(b) *age* = 25 The clustered B+ tree index would be the best option here, with a cost of $3\ (lookup) + 10000\ pages * 0.1\ (selectivity) = 1003$. Although the hash index has a lesser lookup time (roughly 1.2), the potential number of record lookups ($10000\ pages * 0.1 * 20\ tuples\ per\ page = 20000$) renders the clustered index more efficient.

(c) *age* > 20 Again the clustered B+ tree index is the best of the options presented; the cost of this is $3\ (lookup) + 10000\ pages * 0.1\ (selectivity) = 1003$.

(d) *eid* = 1000 Since *eid* is a candidate key, one can assume that only one record will be in each bucket. Thus, the total cost is roughly 1.2 (lookup) + 1 (record access) which is 2 or 3.

(e) *sal* > 200 ∧ *age* > 30. This case is similar to the age > 20 one when we first evaluate the age > 20 clause and the cost is: $3\ (lookup) + 10000\ pages * 0.1\ (selectivity) = 1003$.

(f) *sal* > 200 ∧ *age* = 20. Similar to the previous part, except now we do not need to scan all matching index pages for age. We lookup for the first data page with age = 20, and start scanning all data pages for sal > 200 until we reach age 21. Total cost: $3\ (lookup) + 10000\ pages * 0.1\ (selectivity) * 0.1\ (selectivity) = 103$.

(g) *sal* > 200 ∧ *title* = '*CFO*' In this case, the filescan is the best available method to use, with a cost of 10000.

(h) *sal* > 200 ∧ *age* > 30 ∧ *title* = '*CFO*' Here an age condition is present, so the clustered B+ tree index on < *age,sal* > can be used. Here, the cost is $3\ (lookup) + 10000\ pages * 0.1\ (selectivity) = 1003$.

2.
(a) *sal* > 100 Since the desired result is only the average salary, an index-only scan can be performed using the unclustered B+ tree on *sal* for a cost of $2\ (lookup) + 10000 * 0.1 * 0.2\ (\ due\ to\ smaller\ index\ tuples\ ) = 202$.

(b) *age* = 25 For this case, the best option is to use the clustered index on < *age,sal* > and thus perform index-only scan. The cost of this operation is $3\ (lookup) + (10000 * 0.4) * 0.1 = 403$.

(c) *age* > 20 Similar to the *age* = 25 case, this will cost 403 using the clustered index.

(d) *eid* = 1000 Being a candidate key, only one relation matching this should exist. Thus, using the hash index again is the best option, for a cost of 1.2 (hash lookup) + 1 (relation retrieval) = 2.2.

(e) *sal* > 200 ∧ *age* > 30 Using the clustered B+ tree again as above is the best option, with a cost of 403.

(f) *sal* > 200 ∧ *age* = 20 Similarly to the *sal* > 200 ∧ *age* = 20 case in the previous problem, this selection should use the clustered B+ index for an index only scan, costing $3\ (lookup) + 4000 * 0.1\ (selectivity\ for\ age) * 0.1\ (selectivity\ for\ sal) = 43$.

(g) *sal* > 200∧*title* = '*CFO*' In this case, an index-only scan may not be used, and individual relations must be retrieved from the data pages. The cheapest method available is a simple filescan, with a cost of 10000 I/Os.

(h) *sal* > 200 ∧ *age* > 30 ∧ *title* = '*CFO*' Since this query includes an age restriction, the clustered B+ index over < *age,sal* > can be used; however, the inclusion of the title field precludes an index-only query. Thus, the cost will be: $3\ (lookup) + 10000\ pages * 0.1\ (selectivity) = 1003$.

3.
(a) *sal* > 100 The best method in terms of I/O cost requires usage of the clustered B+ index over < *age,sal* > in an index-only scan. Also, this assumes the ability to keep a running average for each age category. The total cost of this plan is: $10000 * 0.4\ (index - only\ scan) = 4000$. Note that although *sal* is part of the key, since it is not a *prefix* of the key, the entire list of pages must be scanned.

(b) *age* = 25 Again, the best method is to use the clustered B+ index in an index-only scan. For this selection condition, this will cost $3\ (age\ lookup\ ) + 4000\ pages * 0.1\ (selectivity\ on\ age) = 403$.

(c) *age* > 20 This selection uses the same method as the previous condition, the clustered B+ tree index over < *age,sal* > in an index-only scan, for a total cost of 403.

(d) *eid* = 1000 As in previous questions, *eid* is a candidate field, and as such should have only one match for each equality condition. Thus, the hash index over *eid* should be the most cost effective method for selecting over this condition, costing 1.2 (hash lookup) + 1 (tuple retrieval) = 2.2.

(e) *sal* > 200 ∧ *age* > 30 This can be done with the clustered B+ index and an index-only scan over the < *age,sal* > fields. The total estimated cost is $3\ (lookup) + 4000\ pages * 0.1\ (selectivity\ on\ age) = 403$.

(f) *sal* > 200 ∧ *age* = 20 This is similar to the previous selection conditions, but even cheaper. Using the same index-only scan as before (the clustered B+ index over < *age,sal* >), the cost should be $3 + 10000 * 0.4 * 0.1\ (age\ selectivity) * 0.1\ (sal\ selectivity) = 43$.

(g) *sal* > 200∧ *title* = '*CFO*' Since we have a clustered < *age,sal* > index, we can use the ordering of the data to answer this query with a simple filescan while keeping running average for each age and each qualifying tuple.

(h) *sal* > 200 ∧ *age* > 30 ∧ *title* = '*CFO*' Using the clustered B+ tree over < *age,sal* > would accrue a cost of $3 + 10000 * 0.1\ (selectivity\ of\ age) = 1003$.

4.
(a) *sal* > 100 Since we only need sal and age attributes, the best operation involves scanning the clustered index on <age, sal>, filtering on sal>100 and sorting the result. We can do initial sorting pass using 10 buffers (because 1 is needed for scanning the index), and the merging passes using 11 buffers (1 is used for output, so the base of the log in the formula will be 10). Given the selectivity of the predicate result has 400 pages that can be sorted in $\log_{10} \frac{400}{10} = 2$ passes. Finally, we do not need to write out the final result and we can do the aggregation during the last pass. The final cost is:
$4000\ (reading\ the\ index) + 400\ (first\ pass) + 400 * 2 * 2\ (sorting) - 400\ (writing\ out\ the\ result) = 5600$.

(b) *age* = 25 This case is similar to the previous, only we can probe the clustered index and then read the 400 qualifying pages.
Option 1: sort on sal the qualifying pages, where the cost is: $3\ (lookup) + 400\ (reading\ the\ index) + 400\ (first\ pass) + 400 * 2 * 2\ (sorting) - 400\ (writing\ out\ the\ result) = 2003$.
Option 2: Since the predicate states that age has value of 25, we could skip sorting, where the cost is: $3\ (lookup) + 400\ (reading\ the\ index)$.

(c) *age* > 20 Using the same approach as the previous case, the cost is 2003.

(d) *eid* = 1000 Being a candidate key, only one relation should match with a given *eid* value. Thus, the estimated cost should be 1.2 (hash lookup) + 1 (tuple retrieval).

(e) *sal* > 200∧ *age* > 30 In this case, we can also use the clustered index on <age, sal> similarly to the case b). We first probe to find first tuple satisfying age > 30, then scan the 400 qualifying pages and filter on the predicate sal > 200. The qualifying tuples can be stored in 40 pages ($4000 * 0.1\ (age\ selectivity) * 0.1\ (sal\ selectivity)$). Using other 10 buffers we can produce 4 sorted runs of 10 pages, which we can merge in one pass during which we also perform aggregation. The total cost is: $3(lookup) + 400\ (reading\ the\ index) + 40\ (first\ pass) + 40\ (merging) = 483$.

(f) *sal* > 200 ∧ *age* = 20 Similarly to 3f) we probe the clustered index on <age,sal> and read the qualifying tuples from 40 pages of the index.
Option 1: sort on sal the qualifying page, where the cost is: $3(lookup) + 400\ (reading\ the\ index) + 40\ (first\ pass) + 40\ (merging)$.
Option 2: Since the predicate states that age has value of 20, we could skip sorting, where the cost is: $3\ (lookup) + 40\ (reading\ the\ index) = 43$.

(g) *sal* > 200 ∧ *title* = '*CFO*' In this case, we need to do a filescan, perform the filters to find $10000 * 20 * 0.1\ (selectivity\ for\ sal) * 0.1\ (selectivity\ for\ title) = 2000$ and project sal and age attributes. These attributes require 40 pages. We can use 10 buffers to generate the sorter runs that we can then merge and perform aggregation in another

pass for the total cost of: $10000\ (filescan) + 40\ (first\ pass) + 40\ (merging) = 10080$. If we tried to use index on sal, we would need to scan all index pages that satisfy sal > 200 and retrieve all matching tuples for the cost of $2\ (lookup) + 2000 * 0.1(index\ scan) + 10000 * 20 * 0.1\ (selectivity\ for\ sal) = 20202$. There are no additional costs because aggregation can be done by keeping running averages while retrieving tuples.

(h) *sal* > 200 ∧ *age* > 30 ∧ *title* = '*CFO*' In this case, the number of tuples that satisfy all 3 conditions is $10000 * 20 * 0.1\ (selectivity\ for\ sal) * 0.1\ (selectivity\ for\ age) * 0.1\ (selectivity\ for\ title) = 200$ We need two attributes, sal and age, sized 20 bytes each, to compute the result and we can store 200 pairs of them on 4 buffer pages. This means that contrary to previous cases, we can do the aggregation in memory (without external sort). We can retrieve the matching values by probing the clustered index and scanning the data pages for the cost of $3\ (lookup) + 10000 * 0.1\ (selectivity\ of\ age) = 1003$

5.
(a) *sal* > 200 ∨ *age* = 20 In this case, a filescan would be the most cost effective, because the most cost effective method for satisfying *sal* > 200 alone is a filescan.

(b) *sal* > 200 ∨ *title* = '*CFO*' Again a filescan is the better alternative here, since no index at all exists for *title*.

(c) *title* = '*CFO*' ∧ *ename* = '*Joe*' Even though this condition is a conjunction, the filescan is still the best method, since no indexes exist on either *title* or *ename*.

Transactions

Exercise 4: Consider a database with objects *X* and *Y* and assume that there are two transactions *T*1 and *T*2. Transaction *T*1 reads object *X,* and then writes objects *Y and X*. Transaction *T*2 reads object *X,* then reads object *X* once more, and finally writes objects *X* and *Y* (i.e. T1: R(X), W(Y), W(X); T2: R(X), R(X), W(X), W(Y))

1. Give an example schedule with actions of transactions *T*1 and *T*2 on objects *X* and *Y* that results in a write-read conflict.
2. Give an example schedule with actions of transactions *T*1 and *T*2 on objects *X* and *Y* that results in a read-write conflict.
3. Give an example schedule with actions of transactions *T*1 and *T*2 on objects *X* and *Y* that results in a write-write conflict.
4. For each of the three schedules, show that Strict 2PL disallows the schedule.

Exercise 4 solution: The answer to each question is given below.

1. The following schedule results in a write-read conflict:
   T2:R(X), T2:R(X), T2:W(X), T1:R(X) ...
   T1:R(X) is a dirty read here.
2. The following schedule results in a read-write conflict:

T2:R(X), T1:R(X), T1:W(Y), T1:W(X), T2:R(X) ...
Now, T2 will get an unrepeatable read on Y.

3. The following schedule results in a write-write conflict:
   T1:R(X), T2:R(X), T2:R(X), T1:W(Y), T2:W(X), T1:W(X), T2:W(Y) ...
   Now, the last values of X and Y are neither completely from T1 nor completely from T2. This is write-write conflict.
4. Strict 2PL resolves these conflicts as follows:
   (a) Here, T1 could not get a shared lock on X because T2 would be holding an exclusive lock on X. Thus, T1 would have to wait until T2 was finished.
   (b) Here, T1 could not get an exclusive lock on X because T2 would already be holding a shared or exclusive lock on X.
   (c) Here, T2 could not get an exclusive lock on X since T1 would already be holding a shared lock on X.

Concurrency control

Exercise 5: Consider the following sequences of actions, listed in the order they are submitted to the DBMS:

- **Sequence S1:** T1:R(X), T2:W(X), T2:W(Y), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit
- **Sequence S2:** T1:R(X), T2:W(Y), T2:W(X), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit

For each sequence and for Strict 2PL with deadlock detection (show the waits-for graph in case of deadlock) concurrency control mechanism, describe how the concurrency control mechanism handles the sequence. Assume that the timestamp of transaction *Ti* is *i*. Use lock-based concurrency control mechanisms, and add lock and unlock requests to the previous sequence of actions. The DBMS processes actions in the order shown. If a transaction is blocked, assume that all its actions are queued until it is resumed; the DBMS continues with the next action (according to the listed sequence) of an unblocked transaction.

Exercise 5 solution: The answer to each question is given below.

1. In deadlock detection, transactions are allowed to wait; they are not aborted until a deadlock has been detected.
   Sequence S1: T1 gets a shared-lock on X;
   T2 blocks waiting for an exclusive-lock on X;
   T3 gets an exclusive-lock on Y;
   T1 blocks waiting for an exclusive-lock on Y;
   T3 finishes, commits and releases locks;
   T1 wakes up, gets an exclusive-lock on Y, finishes up and releases lock on X and Y;

T2 now gets both an exclusive-lock on X and Y, and proceeds to finish.
No deadlock.
Sequence S2: There is a deadlock. T2 waits for T1, while T1 waits for T2.
Deadlocks are resolved either using a timeout or by maintaining a waits-for graph.