

Advanced Database Technology

IT University of Copenhagen

June 12, 2003

This examination assignment consists of 5 problems with a total of 13 subproblems. The weight of each problem is stated. You have 4 hours to answer all 13 subproblems. You may choose to write your answer in Danish or English. Remember to write the page number, your name, and your CPR-number on each page of your written answer. The complete assignment consists of 5 numbered pages (including this page).

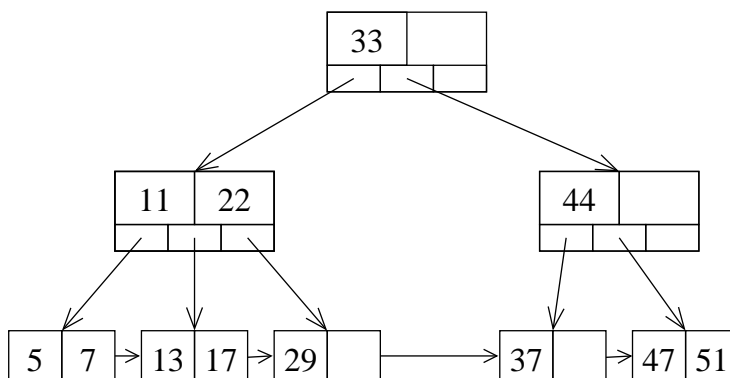
GUW refers to *Database Systems – The Complete Book* by Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom, 2002

In problems that ask for efficient algorithms, the asymptotic complexity of your algorithm will be taken into account when grading. When answering, remember to always **explain** (in reasonable detail) how you arrived at your answer.

All written aids are allowed / Alle skriftlige hjælpemidler er tilladt.

1 Index structures (25%)

Consider the below B-tree. Internal nodes hold up to 2 keys (and three pointers), and each leaf holds up to 2 keys.



a) Suppose that the keys 41 and 19 are inserted in the above B-tree. Draw the B-tree that results when performing the insertions as described in G UW (it is not necessary to draw intermediate steps).

In the following we consider *persistent hash tables*, which store a set that changes over time, and allow queries of the form “was key x in the set at time t ?” The implementation we consider is similar to ordinary chained hashing, except that each chain (i.e., the set of keys with a specific hash function value) is stored using a persistent B-tree. Since we will perform only searches for a specific key (and not e.g. range searches) it does not matter what ordering of the keys is chosen for the persistent B-tree.

b) In the index described above, consider a search for key x at time t . Describe how many I/Os are used for the search?

Hint: What is important for the time to search for x ? Define a variable for this quantity and use it to express the number of I/Os.

Ordinary B-trees allow an index that can be used to efficiently implement the selection operator $\sigma_{A \leq a}$: For a B-tree index on attribute A , this corresponds to a range query in the interval $(-\infty; a]$. In the following we consider how this can be extended to the selection operator $\sigma_{A_1 \leq a_1 \wedge A_2 \leq a_2}$ in the case where the relation is *static*, i.e., where the index does *not* have to be updated according to insertions or deletions of tuples. As in G UW we let $T(R)$ denote the number of tuples in relation R .

c) Show how to construct an index for a relation $R(A_1, A_2)$, where A_1 and A_2 are integer attributes, such that for any integers a_1 and a_2 the relation $R' = \sigma_{A_1 \leq a_1 \wedge A_2 \leq a_2}(R)$ can be computed in $O(\log_B(T(R)) + T(R')/B)$ I/Os.

Hint: Use a persistent B-tree.

2 Query evaluation (20%)

Suppose we have relations $R_1(A_1, A_2, A_3)$ and $R_2(A_2, A_4)$, where A_2 is the primary key of R_2 . All attributes have fixed length. The length of A_1 is 10 bytes, the length of A_2 is 20 bytes, the length of A_3 is 30 bytes, and the length of A_4 is 40 bytes. The relations reside on a disk with block size $B=3000$ bytes. We want to perform a sort join of R_1 and R_2 , using 101 blocks (i.e., 303000 bytes) of internal memory, of which one is reserved to be used as an output buffer.

In the following we consider a sort join of R_1 and R_2 , as described in G UW 15.4.7. Again, we let $T(R)$ denote the number of tuples in relation R .

a) For what values of $T(R_1)$ and $T(R_2)$ does the sort join use two passes? (Write a condition involving $T(R_1)$ and $T(R_2)$.)

b) Assume that, in the notation of G UW, $T(R_1) = 10,000$ and $T(R_2) = 30,000$. How many I/Os are used for the sort join in the worst case? Count also I/Os to write the output.

It is pointed out in G UW section 15.4.7 that the sort join algorithm presented there does not take into account the “atypical” case where the set of tuples with a particular value on the join attribute(s) does not fit into internal memory. The aim of the following is to arrive at an algorithm that handles the general case efficiently.

c) Modify the two-pass sort join described in G UW p. 746–747 to get a sort join that, in the worst case, uses $3(B(R) + B(S)) + O(B(R \bowtie S))$ I/Os to compute the sort join of two relations R and S . You may assume, as in G UW, that internal memory is bigger than approximately $\sqrt{B(R) + B(S)}$. Argue why your algorithm has the desired complexity.

3 Choosing an index (20%)

Consider a space probe on its way to the equator of a remote rock planet. Upon impact it will need to determine its exact position along the equator. A previous space probe has gathered altitude data for all of the planet. The altitude data for the equator is a sequence of 40,000,000 integers in the range 0 to 10,000. The idea is to let the space probe gather altitude information in its near surroundings and search for this sequence in the data. As a small example, if the altitude data is



731, 731, 749, 748, 746, 748, 747, 749, 748, 750, 733

the sequence (749, 748) fits two locations (and thus cannot be used to locate the probe). On the other hand, the sequence (749, 748, 746) has a unique position. The space probe is equipped with hard drives that can hold all of the data, but its main memory (dated 1975) can hold only a few disk blocks. You are given the task of choosing an index for the data, such that for any sequence (a_1, a_2, \dots, a_i) of altitudes, it can efficiently be determined whether this sequence matches a unique position in the data (and if so, what this position is).

a) Suppose it is known that the sequence looked up does not “wrap around” the end of the data (as e.g. (750, 733, 731) in the above data). Suggest an efficient index that works in this setting.

b) Extend the solution such that “wrap around” sequences are also efficiently handled.

4 Query optimization (20%)

In the following we consider relational algebra on sets (not bags). It is known that the following laws hold in relational algebra:

$$R \bowtie (R' \cap R'') = (R \bowtie R') \cap (R \bowtie R'') \quad (1)$$

$$\sigma_{C_1 \wedge C_2}(R) = \sigma_{C_1}(R) \cap \sigma_{C_2}(R) \quad (2)$$

a) Use the above algebraic laws to simplify the relational algebra expression:

$$(R_1 \bowtie \sigma_{A \geq 42}(R_2)) \cap (R_1 \bowtie \sigma_{A \leq 42}(R_2)) .$$

Draw the result as an algebraic expression tree.

Suppose we have relations $R_1(A, B, C)$ and $R_2(A, B, C)$ where (in the notation of GUW):

$$T(R_1) = 900 \quad T(R_2) = 1400$$

$$V(R_1, A) = V(R_2, A) = 20 \quad V(R_1, B) = 30 \quad V(R_2, B) = 7 \quad V(R_1, C) = V(R_2, C) = 10 .$$

b) Estimate the number of tuples in

$$(\sigma_{C \geq 6}(R_1) \bowtie \sigma_{C \leq 13}(R_1)) \bowtie \pi_{A,B}(R_2)$$

using the rules in GUW chapter 16.4. State which rules you use.

Suppose we want to compute the join $R_a \bowtie R_b \bowtie R_c$, where

$$T(R_a) = 1000 \quad T(R_b) = 2000 \quad T(R_c) = 3000 .$$

According to precise estimates, we have

$$T(R_a \bowtie R_b) = 4000 \quad T(R_a \bowtie R_c) = 5000 \quad T(R_b \bowtie R_c) = 6000 .$$

The three relations have fixed length tuples. For a relation R with fixed length tuples, let $Q(R)$ denote the size of a tuple in bytes. Based on the schemas for the relations we have

$$Q(R_a) = 1024 \quad Q(R_b) = 1024 \quad Q(R_c) = 40$$

$$Q(R_a \bowtie R_b) = 2008 \quad Q(R_a \bowtie R_c) = 1024 \quad Q(R_b \bowtie R_c) = 1024 .$$

There are three possible orders in which the join can be computed:

$$(R_a \bowtie R_b) \bowtie R_c \quad (R_a \bowtie R_c) \bowtie R_b \quad (R_b \bowtie R_c) \bowtie R_a .$$

It is known that no matter what order is chosen, each join can be implemented as a two-pass sort join, as described in G UW 15.4.7.

c) Which join order uses the lowest number of I/Os? Argue why your answer is correct.

5 Error recovery (15%)

In G UW, several strategies for logging database changes (and performing error recovery) are presented. This problem considers a special kind of DBMS where, before any transaction, it is known in advance which disk blocks will be involved in the transaction. (Also, transactions take place one at a time, so problems with concurrent transactions do not need to be considered.) Furthermore, any transaction involves relatively few disk blocks, so the set of blocks involved in a transaction fits in internal memory.

We consider the following simple way of performing transactions without logging changes: Before each transaction, load all involved disk blocks into internal memory. Then perform all changes in internal memory, and finally write the blocks back to their locations on disk.

a) Point out a problem with the above strategy. Which of the properties of a transaction may be violated in case of a system crash?

b) Explain how to modify the strategy such that all properties of a transaction can be fulfilled, and such that the amount of external memory used for logging is no larger than internal memory.