School of Information Technologies
Dr Bryn Jeffries

COMP9120 Relational Database Systems **Semester 2, 2016**

Tutorial Week 11 Solution: Query Processing

Exercise 1. Block-Nested Loops Join

Suppose we have a schema Rel1(A, B, C) and Rel2(C, D). Each A field occupies 4 bytes, each B field occupies 12 bytes, each C field occupies 8 bytes, each D field occupies 8 bytes. Rel1 contains 100,000 records, and Rel2 contains 50,000 records. There are 100 different values for A represented in the database, 1000 different values for B, 50,000 different values for C, and 10,000 different values for D. Rel1 is stored with a sparse, clustered primary B+-tree index on the pair of columns (A,B), and Rel2 is stored with a sparse, clustered primary B+-tree index on C.

General features: assume that each page is 4K bytes, of which 250 bytes are taken for header and array of record pointers. Assume that no record is every split across several pages. Assume that data entries in any index use the format of (search key, rowid), where rowid uses 4 bytes.

Consider the following query:

```
SELECT Rel1.A, Rel1.B, Rel1.C
FROM Rel1, Rel2
WHERE Rel1.C = Rel2.C AND Rel2.D = 16;
```

and consider the query plan which calculates this as follows:

Form the equi-join of Rel1 and Rel2 by using a block nested loops join with Rel1 as the outer relation, and then filter each tuple of the join to see if the value of D is 16; if so, output the values of A, B and C from that tuple of the join.

How many pages of I/O are needed to compute this plan (assume that we have only the minimal space, say 2 pages worth, for buffering in memory)?

First we compute the storage structure information:

A page has 4096 bytes, of which 3846 are available to store data records. Since each data record of Rel1 takes 24 bytes, we can fit 160 records in a page; the remaining 6 bytes in each page are wasted. If we fill each page as much as possible, the 100,000 records thus take 100000/160 = 625 pages. If, as is a reasonable assumption in a B+-tree, we have on average 75% of the records possible, that would be 120 records per page, and so 100,000/120=833 or so pages would be used to store the data records at the leaf level of the tree.

Each entry in the primary index of Rel1 takes 20 bytes (16 for the search key (A,B) and 4 for a rowid). Each page of the index has 4096-250 = 3846 bytes available, which can be used for 3846/20=192 entries (recall that we must round the fraction down, as we can't put part of an entry into a page, and leave part for another page). If we assume that on average, pages are 75% full, the average fan-out will be 144; that is, each page in the index will point to 144 pages on the next level down. Thus the level above the leaves has 833/144, that is about 6 pages; then there is a root page which is one level above this.

For Rel2, each data record takes 16 bytes (8 for C and 8 for D), so we can fit 240 data records in a page. If each page is as full as possible, there would be 50000/240=209 pages (rounding up); if we assume each leaf page is 75% full, we would have 180 data records in each page and 50000/180=277 or 278 leaf pages. Each entry in the primary index of Rel2 takes 12 bytes (8 for the search key C and 4 for a rowid), so each page of index has room for 3846/12=320 entries; if each index page is 75% full, the fan-out on average will be 240. Thus we can expect 2 pages in the level above the leaves (to point to the 278 leaf pages), and a root page above that [if we are lucky, we might get the level above the leaves just a bit over 75% full, so there could be a single page there, and the whole tree would have only 2 levels: the root and the leaves].

Now we can think about the costs of the query processing: to do the nested block join, we read each data page of the outer relation once, and we read each data block of the inner relation as many times as there are blocks in the outer relation (because we scan the inner relation for every block of the outer one). Note that this processing does not use or examine the higher levels of the index; only the leaves which contain the data are scanned. Thus to compute the join takes 833+833*278=232407 I/Os. The filtering and output then take place in memory, incurring no extra I/O cost.