THE UNIVERSITY OF SYDNEY

COMP9120 Relational Database Systems

Tutorial Week 12 Example Solution: Query Optimization

Exercise 1. Index Nested Loops Join

Consider the same scenario as last week. 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 again 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;
```

Last week you calculated the cost (in pages of I/O, assuming a minimal buffer pool of 2 pages) of a plan in which the join of *Rel1* and *Rel2* was performed using a block nested loops join with Rel1 as the outer relation, followed by the selection and projection steps. Assuming 75% page fill, you should have calculated this plan to cost 232,407 page I/O operations.

Now consider a different query plan, where the join is processed as an index nested loop join (using the primary index on *Rel2.C*), and then each tuple of the join is filtered to check the value of *D*. How does this affect the cost?

In this plan, we scan Rel1, reading each of its pages once. For each tuple in Rel1, we do a lookup on Rel2 for a given value of C. Each lookup costs 3 I/Os (or 2 if we were lucky and the tree had only 2 levels). Thus the cost of the join is 833+100000*3=300833 I/Os (or 200833 if we are lucky with the structure of Rel2). Again filtering and output are done in memory without further I/O cost.

Exercise 2. Alternative plans

Suggest any additional indices that might be helpful, and then propose a query plan to take advantage of those indices. What is the cost of the resulting plan? How much extra space is used for the extra indices?

The condition on Rel2.D in the where clause yields a small number of matches (estimate 5, since there are 10,000 distinct values in 50,000 records), so we might want to first select out these tuples, which would be helped by a secondary index on Rel2.D. To join the result of

this with Rel1, we might then want to do an index join with Rel1 as the inner table, in which case we would also need an index on Rel1.C.

So consider how these two indexes. Since both relations have primary indexes, we can only add secondary, dense, unclustered indexes. For the index on Rel1.C each entry must be 12 bytes (8 for C and 4 for the rowid), so a page of the index can fit 320 entries; if each index page is 75% full, the average fan-out will be 240. Because the index is dense, the leaf level must have 100,000 entries, so 417 leaf pages are needed; thus 2 pages are at the level above the leaves, and a root page above that. The total space for the secondary index on Rel1.C is 1+2+417=420 pages.

The index on Rel2.D has entries of 12 bytes (8 for the search key D and 4 for the rowid), so again the average fan-out is 240 if the index pages are 75% full. There are 50,000 entries in the leaf level (since the index is dense), so there are 208 or 209 leaf pages of secondary index; thus the level above can be the root. The space for the index on Rel2.D is 1+209=210 pages.

Given these indexes, the query plan is: use the secondary index on Rel2.D to find the records with Rel2.D=16; then for each of them, look up the matching records in Rel1 using the index on Rel1.C, and output the values of A, B and C.

The cost of the selection using Rel2.D is to look in the root, then look in the appropriate leaf page of the index, then for each of these index entries fetch the data record from Rel2. That is, the selection will cost 1+1+5=7 I/Os (note that Rel2's data records are clustered on C, so the ones with given value of D will not generally be together; thus finding 5 such data records needs 5 I/Os).

The index join costs are scanning the output of the selection, and for each tuple, doing an index-based lookup to get the matching tuples of Rel1. A single index-based lookup on Rel1.C requires a read of the index root, one page at the next level, one page at the leaf level, and then retrieving the pages containing the corresponding data records. There are on average 2 matching data records in Rel1 (as there are 100,000 records and 50,000 different values for C), so we need to do 1+1+1+2=5 I/Os to do each index-based lookup. The cost of the join would then be 1+5*5=26 pages of I/O.

If the join is pipelined with the selection, we don't need to scan the output of the selection separately, saving 1 I/O in the join cost; thus the total cost of the query is 7+25=32 pages of I/O.