Tutorial: implementing RA operators

BNLJ. Suppose B(R)=B(S)=10,000, and M=1000.
Calculate the disk I/O cost of a block nested-loop join.
For the above relations, what value of M would we need to compute R ⋈ S using the block nested-loop algorithm with no more than

a. 100,000 I/Os
b. 25,000 I/Os
c. 15,000 I/Os

$$\mathrm{B}(R) + \frac{B(R)}{M-1} B(S)$$

- 10,000+10,000*10,000/999 = ceiling(110,100.1)=110,101

From: B(R)+B(R)*B(S)/(M-1) <= limit

M >= B(R)*B(S)/(limit – B(R)) – 1

a. M >= 10,000*10,000/(100,000 – 10,000) -1=ceiling(1110.1) = 1,111

b. M >= 10,000*10,000/(25,000 – 10,000) -1=ceiling(6665.7) = 6666

c. M >= 10,000*10,000/(15,000 – 10,000) -1=19999.

However no amount of memory will make number of disk I/Os less than 10,000+10,000 – the total number of blocks in both relations. So the answer is - impossible to do the join with 15,000 disk I/Os.

SMJ_HJ. Assume that you want to join two relations R(A,B) and S(B,C). The two relations are stored as simple (unsorted) heap files. When would you prefer a hash join to a sort-merge join, and when would you prefer a sort-merge join to a hash-join?

- Hash join:
  - if either R or S is much smaller than the other, in this case we will need less memory;
  - if we want to perform the join in parallel.

- Sort-merge join:
  - if the output is expected to be sorted;
  - if the hashing keys are skewed, and thus some hash buckets can be too large to be processed in memory.

SET UNION. Discuss the use of algorithmic techniques learned in class for implementing a set union operator: i.e. the union of 2 relations with duplicates eliminated. Present a high-level pseudocode for each proposed technique.

I. One-pass block nested loop algorithm

Input: 2 relations R and S, $B(S)<B(R)$, $B(S)<=M-1$

1. Read entire S into M-1 memory buffers
2. Build a search structure where the key is the entire tuple
3. Output all tuples of S
4. Read R block-by-block and compare its tuples to the tuples in the search structure.
5. If a tuple of R is nor found, output it

SET UNION. 2/4

II. Two-pass sort-based union algorithm

1. Repeatedly bring M blocks of R into main memory, sort the tuples, and write sorted runs to disk
2. Do the same for S, to create sorted runs for S
3. Use one main-memory buffer for each run from R and S. Initialize each with the first block from the corresponding run.
4. Repeatedly find the smallest tuple t among all buffer elements. Copy t to output and remove from buffers all copies of t, advancing current buffer pointer. If a buffer has been processed, refill it from the corresponding run.

SET UNION. 3/4

III. Two-pass hash-based union algorithm.

1. As in duplicate elimination, we separately hash both R and S to M-1 buckets, using the same hash function for both relations.
2. If a tuple t appears in both R and S, we will find it in the buckets with the same hash value: suppose Ri and Si.
3. We process 1 pair of buckets at a time, and output only a single copy of t

SET UNION. 4/4

IV. Index-based algorithms

- In general, the index for the entire tuple is not very common. However, if R and S have 1 or 2 attributes each, it may happen that the index on these attributes exists both for R and for S.
- If the indexes are B-trees, we can scan the leaves of both trees in order, using the pointers from leaf to leaf. When more than one copy of a key is found in either index, we output a single copy, and skip all the rest.
- If the indexes are Hashes, we can use them only if for both R and S the same hashing was used. In this case, we will read the keys in the buckets Ri and Si with the same hash value, and output only a single copy of each key.

COMPARING:

Consider the join $R \bowtie_{R.A=S.B} S$ and the following information on R and S.

Relation R contains 10000 tuples and has 10 tuples per block.
Relation S contains 2000 tuples and has 10 tuples per block.
Attribute B of relation S is the primary key for S.
Neither R nor S is sorted on the join attribute.
Neither relation has any indexes built on it.
There are 51 main memory buffers available.

COMPARING:
Consider the join R $\bowtie_{R.A=S.B}$ S.

A. What is the cost of joining R and S using the simple nested loop join? What is the minimum number of buffer pages required for this cost to remain unchanged?

B. What is the cost of joining R and S using the block nested loop join? What is the minimum number of buffer pages required for this cost to remain unchanged?

C. What is the cost of joining R and S using a sort-merge join? What is the minimum number of buffer pages required for this cost to remain unchanged?

D. What is the cost (in disk I/O's) of joining R and S using a hash join? What is the minimum number of buffer pages required for this cost to remain unchanged?

E. What join algorithm yields the least cost if you were free to choose the number of free buffers? Briefly motivate your answer and give the exact optimal cost.

F. How many tuples does the join of R and S produce, at most, and how many blocks are required to store the result of the join back on disk?