Join Algorithms Reference Sheet

Suppose that we want to join tables R and S on the attribute a: $R \bowtie_{R.a=S.a} S$

Tuple-based Nested Loop Join

```
For each tuple t1 in R:
      For each tuple t2 in S:
            If t1.a == t2.a:
                  Join(t1, t2)
```

Page IO: **B(R) + T(R)B(S)**
Memory Usage: **2 pages**

Tuple-based nested loop join is the most basic type of join. For each tuple in R, the inner loop scans over S once and joins every tuple that matches. This, however, is very inefficient for large tables. Because S is scanned once for every tuple in R, the total number of page reads from S is T(R)B(S). Furthermore, the outer loop reads over R one time, incurring B(R) page reads. Therefore, the total cost is B(R) + T(R)B(S).

An advantage of the tuple and page-based nested loop joins is that they only require a single page of R and S to be in main memory at any given time. Therefore, the memory usage is 2 pages.

Page-based Nested Loop Join

```
For each page p1 in R:
      For each page p2 in S:
            For each tuple t1 in p1:
                  For each tuple t2 in p2:
                        If t1.a == t2.a:
                              Join(t1, t2)
```

Page IO: **B(R) + B(R)B(S)**
Memory Usage: **2 pages**

Page-based nested loop join is an optimization of tuple-based nested loop join because it scans over S once for every *page* in R, while tuple-based join scans S once for every *tuple* in R. As a result, it prevents S from being scanned as many times, thus decreasing the number of disk reads.

Similarly to the tuple-based nested loop join, all of R is read once with a total of B(R) page reads. Then, for each page in R, all of S is read once. Therefore, the total cost of reading S once for each page of R is B(R)B(S). Hence, the cost of reading both tables is B(R) + B(R)B(S).

The memory usage is still 2 pages, since one page of R and one page of S need to be in main memory at any point.

Group-of-pages Nested Loop Join

Let M be the number of pages available in main memory.

```
For each group of M-1 pages r in R:
      For each page s in S:
            For each tuple t1 in r:
                  For each tuple t2 in s:
                        If t1.a == t2.a:
                              Join(t1, t2)
```

Page IO: **B(R) + B(R)B(S)/(M-1)**
Memory Usage: **M**

The group of pages nested loop join is an optimization over the page-based nested loop join. Instead of reading a single page of R at a time before scanning S, it reads over M-1 pages of R before scanning S. This takes advantage of extra memory available, and requires there to be at least M pages in main memory. It allows for S to be scanned fewer times; precisely speaking, S needs to be scanned B(R) / (M-1) times. It may help to think of the algorithm as splitting R into chunks of size M-1 pages, and then joining each of the tuples in each chunk of R with all of the tuples in S.

The algorithm still reads all the blocks in R once, so page reads from R end up being B(R). Since S is scanned B(R) / (M-1) times, the cost of reading S is B(S)B(R)/(M-1). Therefore, the algorithm makes a total of B(R) + B(S)B(R)/(M-1) disk reads.

Index-based Join

Assume S has an index on the join attribute S.a.

1. Sequentially scan each tuple t1 in R.
2. Use the index to fetch the tuple from S with the same join attribute value t1.a.
3. Join t1 with each tuple from S with the equal join attribute value.

Page IO (clustered): **B(R) + T(R)B(S) / V(S, a)**
Page IO (unclustered): **B(R) + T(R)T(S) / V(S, a)**
Memory Usage: **2 pages**

When the index is clustered, the index-based join ends up reading an average of B(S) / V(S, a) pages from S for each tuple in R. If the index is unclustered, the index-based join reads an average of T(S) / V(s, a) pages from S for each tuple in R because there is no guarantee that the tuples in the unclustered index are contiguous in memory, which means that another block read might be necessary to fetch each tuple. The clustered index lets us take advantage of how tuples in S with the same join attribute value are generally on the same block, which prevents us from having to read a new block every time we want to read a tuple of the same join value.

Hash Join

1. Scan the smaller table, R, and build a hash table in main memory. The hash table maps each distinct value of the join attribute to a list of tuples that have that attribute value.
2. Scan S sequentially. For each tuple s in S, check the hash table to see if R has any tuples which have the same value of the join attribute.
3. Join each tuple in S with any tuples in R which are found to have the same join attribute.

Page IO: **B(R) + B(S)**
Memory Usage: **B(R)**

Step 1 of the algorithm sequentially reads in every tuple from R to build the hash table, so step 1 reads B(R) pages from disk. Step 2 sequentially reads in every tuple from S to compare it with the matching tuples from R, so it ends up reading B(S) pages. Combined, the total cost is B(R) + B(S).

The algorithm builds a hash table on R, and the hash table will have size approximately equal to that of R. Therefore, it will take around B(R) pages of memory to build the hash table. This is the main disadvantage of a hash join: it requires the entire smaller relation to be able to fit in main memory!

Sort-merge Join

1. Read each tuple sequentially from table R into a list, and sort the list on its join attribute.
2. Read each tuple sequentially from table S into a list, and sort the list on its join attribute.
3. Iterate over each of the lists from left to right and merge tuples with equal join attribute values.

Page IO: **B(R) + B(S)**
Memory Usage: **B(R) + B(S)**

The algorithm uses B(R) disk reads in step 1 to scan all of table R. It then uses B(S) disk reads in step 2 to read all of S. Although it takes time to sort the tables, we generally ignore that cost because we only consider cost of reading from and writing to disk. Therefore, total cost is B(R) + B(S).

The algorithm requires all of the tuples in R and S to be held in memory at the same time, since it has to sort the lists containing all their tuples. Therefore, memory usage is B(R) + B(S).