```
FOR each chunk of M-1 blocks of S DO BEGIN
    read these blocks into main-memory buffers;
    organize their tuples into a search structure whose
        search key is the common attributes of R and S;
    FOR each block b of R DO BEGIN
        read b into main memory;
        FOR each tuple t of b DO BEGIN
            find the tuples of S in main memory that
                join with t;
            output the join of t with each of these tuples;
        END;
    END;
END;
```

Figure 8: The nested-loop join algorithm

The program of Fig. 8 appears to have three nested loops. However, there really are only two loops if we look at the code at the right level of abstraction. The first, or outer loop, runs through the tuples of S. The other two loops run through the tuples of R. However, we expressed the process as two loops to emphasize that the order in which we visit the tuples of R is not arbitrary. Rather, we need to look at these tuples a block at a time (the role of the second loop), and within one block, we look at all the tuples of that block before moving on to the next block (the role of the third loop).

Example 4: Let $B(R) = 1000$, $B(S) = 500$, and $M = 101$. We shall use 100 blocks of memory to buffer S in 100-block chunks, so the outer loop of Fig. 8 iterates five times. At each iteration, we do 100 disk I/O's to read the chunk of S, and we must read R entirely in the second loop, using 1000 disk I/O's. Thus, the total number of disk I/O's is 5500.

Notice that if we reversed the roles of R and S, the algorithm would use slightly more disk I/O's. We would iterate 10 times through the outer loop and do 600 disk I/O's at each iteration, for a total of 6000. In general, there is a slight advantage to using the smaller relation in the outer loop. □

3.4 Analysis of Nested-Loop Join

The analysis of Example 4 can be repeated for any $B(R)$, $B(S)$, and M. Assuming S is the smaller relation, the number of chunks, or iterations of the outer loop is $B(S)/(M-1)$. At each iteration, we read $M-1$ blocks of S and $B(R)$ blocks of R. The number of disk I/O's is thus $B(S)\big(M-1+B(R)\big)/(M-1)$, or $B(S)+\big(B(S)B(R)\big)/(M-1)$.

Assuming all of M, $B(S)$, and $B(R)$ are large, but M is the smallest of these, an approximation to the above formula is $B(S)B(R)/M$. That is, the cost is proportional to the product of the sizes of the two relations, divided by the amount of available main memory. We can do much better than a nested-loop join when both relations are large. But for reasonably small examples such as Example 4, the cost of the nested-loop join is not much greater than the cost of a one-pass join, which is 1500 disk I/O's for this example. In fact, if $B(S) \leq M - 1$, the nested-loop join becomes identical to the one-pass join algorithm of Section 2.3.

Although nested-loop join is generally not the most efficient join algorithm possible, we should note that in some early relational DBMS's, it was the only method available. Even today, it is needed as a subroutine in more efficient join algorithms in certain situations, such as when large numbers of tuples from each relation share a common value for the join attribute(s). For an example where nested-loop join is essential, see Section 4.6.

3.5 Summary of Algorithms so Far

The main-memory and disk I/O requirements for the algorithms we have discussed in Sections 2 and 3 are shown in Fig. 9. The memory requirements for γ and δ are actually more complex than shown, and $M = B$ is only a loose approximation. For γ, M depends on the number of groups, and for δ, M depends on the number of distinct tuples.

Operators	Approximate M required	Disk I/O	Section
σ, π	1	B	2.1
γ, δ	B	B	2.2
$\cup$, $\cap$, $-$, $\times$, $\bowtie$	$\min(B(R), B(S))$	$B(R) + B(S)$	2.3
$\bowtie$	any $M \geq 2$	$B(R)B(S)/M$	3.3

Figure 9: Main memory and disk I/O requirements for one-pass and nested-loop algorithms

3.6 Exercises for Section 3

Exercise 3.1: Give the three iterator methods for the block-based version of nested-loop join.

Exercise 3.2: Suppose $B(R) = B(S) = 10{,}000$, and $M = 1000$. Calculate the disk I/O cost of a nested-loop join.

Exercise 3.3: For the relations of Exercise 3.2, what value of M would we need to compute $R \bowtie S$ using the nested-loop algorithm with no more than (a) 100,000 ! (b) 25,000 ! (c) 15,000 disk I/O's?

! **Exercise 3.4:** If R and S are both unclustered, it seems that nested-loop join would require about $T(R)T(S)/M$ disk I/O's.

a) How can you do significantly better than this cost?

b) If only one of R and S is unclustered, how would you perform a nested-loop join? Consider both the cases that the larger is unclustered and that the smaller is unclustered.

! **Exercise 3.5:** The iterator of Fig. 7 will not work properly if either R or S is empty. Rewrite the methods so they will work, even if one or both relations are empty.

4 Two-Pass Algorithms Based on Sorting

We shall now begin the study of multipass algorithms for performing relational-algebra operations on relations that are larger than what the one-pass algorithms of Section 2 can handle. We concentrate on *two-pass algorithms*, where data from the operand relations is read into main memory, processed in some way, written out to disk again, and then reread from disk to complete the operation. We can naturally extend this idea to any number of passes, where the data is read many times into main memory. However, we concentrate on two-pass algorithms because:

a) Two passes are usually enough, even for very large relations.

b) Generalizing to more than two passes is not hard; we discuss these extensions in Section 4.1 and more generally in Section 8.

We begin with an implementation of the sorting operator τ that illustrates the general approach: divide a relation R for which $B(R) > M$ into chucks of size M, sort them, and then process the sorted sublists in some fashion that requires only one block of each sorted sublist in main memory at any one time.

4.1 Two-Phase, Multiway Merge-Sort

It is possible to sort very large relations in two passes using an algorithm called *Two-Phase, Multiway Merge-Sort* (TPMMS), Suppose we have M main-memory buffers to use for the sort. TPMMS sorts a relation R as follows:

- *Phase 1*: Repeatedly fill the M buffers with new tuples from R and sort them, using any main-memory sorting algorithm. Write out each *sorted sublist* to secondary storage.

- *Phase 2*: Merge the sorted sublists. For this phase to work, there can be at most $M - 1$ sorted sublists, which limits the size of R. We allocate one input block to each sorted sublist and one block to the output. The