A Second Look at Relational Algebra

Class Notes for COS 480/580 and MAT 500

Sudarshan S. Chawathe

University of Maine

October 13, 2006

Logical and Physical Operators Recall the operators, such as projection and join, described in the quick introduction to relational algebra.[1] The definitions of those operators specify only the results of applying the operators, not the methods used to compute the results. In contrast to such *logical operators*, we will now define some *physical operators* by providing a procedural description of their implementations. In general, the mapping between logical and physical operators is not one-to-one. Multiple physical operators may be combined to implement a single logical operator and, similarly, a single physical operator may implement multiple logical operators.

Nested-Loop Join Recall our definition of a predicate join:

$$R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$$

By using the definitions of the selection and cross product operators, we may also define the join independently as

$$R \bowtie_{\theta} S = \{(r, s) \mid r \in R \wedge s \in S \wedge r\theta s\}$$

where we use the notation (r, s) to denote the tuple obtained by concatenating the attributes of tuple r with those of s, and where $r\theta s$ is true if the predicate θ evaluates to true when applied to r and s. This definition suggests a simple nested-loop method for evaluating the join: For each tuple of the left operand of the join, we examine each tuple of the right operand and produce the pair as output if it satisfies the join predicate. This method is summarized by Algorithm 1, where we use the notation $r\|s$ to denote the tuple obtained by appending the attributes of s to those of r.

[1] Sudarshan S. Chawathe, A Quick Introduction to Relational Algebra, Class notes. `http://cs.umaine.edu/~chaw/` October 2006.

Algorithm 1 Nested-Loop Join

```
1: procedure NLJOIN(R,S,θ)
2:     for all tuples r ∈ R do
3:         for all tuples s ∈ S do
4:             if rθs then
5:                 print r||s
6:             end if
7:         end for
8:     end for
9: end procedure
```

At first glance, the nested-loop join seems quite efficient, as it considers each pair of R-S tuples exactly once. Given relations R and S with m and n tuples, respectively, the join predicate on line 4 is evaluated mn times. This standard analysis, which is based on counting the number of CPU operations, is appropriate for data that reside in main memory (RAM). However, in a database environment, the data typically reside in secondary memory (disks). Since disks are slower than main memory by several orders of magnitude, we must revise our analysis by counting the number of accesses to secondary memory.

Data Access Model In the conventional access model used for main memory, fetching a data item (e.g., a byte) by specifying its main-memory address incurs a fixed cost, regardless of the order in which items are accessed. For example, reading 100 bytes that reside in contiguous memory locations costs the same as reading 100 bytes whose locations are spaced 1000 bytes apart in memory. Figure 1 illustrates two cases of this kind. (More careful analyses on modern architectures model the heterogeneous access costs imposed by caches, but we will not discuss them here.)

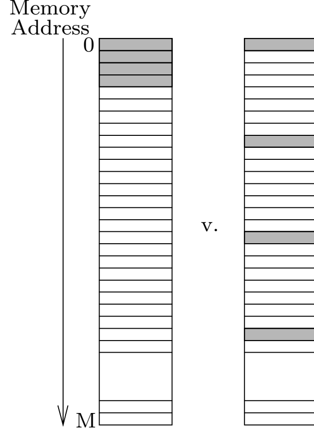

Figure 1: *Main-memory access of four contiguous items costs the same as the access of four items spaced farther apart.*

This model is inappropriate for data stored on disks, which partition data into *blocks* that form the unit of access. The number of data items per block depends on the item sizes relative to the block size (in bytes). In Figure 2, we depict a simplified example with three items per block. In general, accessing x contiguous items of size y with a block size of B incurs $\lceil xy/B \rceil$ block operations. If the items are widely separated, each item resides on a separate block so that xy block accesses are necessary.

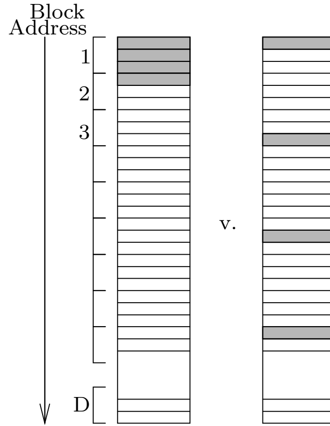

Figure 2: *Accessing the four contiguous items on the left requires two block operations while accessing the four items on the right requires four.*

When a block of data from disk is read into main memory, it is often fruitful to retain that data for some time in the hope that future requests for items in the same block may be fulfilled without incurring additional disk operations. Most database systems allocate some space, say C blocks, for this purpose. As new blocks are read from disk, they are stored in the buffer. If the buffer is full then space is created by discarding some block. The method used to determine the block to discard varies. For the following, we assume the popular *least-recently used (LRU)* policy: The block that is discarded is the one that has been unused for the longest period of time.

We may use this model to analyze Algorithm 1. Suppose each tuple of R occupies p bytes and each tuple of S occupies q bytes. Further, suppose the buffer has space for only two blocks ($C = 2$), of which one is reserved for blocks of R and the other is reserved for blocks of S. Relation R occupies mp bytes and thus $\lceil mp/B \rceil$ blocks on disk. Similarly, S occupies $\lceil nq/B \rceil$ blocks. Each block of R is read once by the outer loop. For each such block, every block of S is read by the inner loop. Since there is space to buffer only one block per relation, each of these block reads results in a disk access. Thus, this nested-loop join incurs the cost of $\lceil mp/B \rceil \cdot (1 + \lceil nq/B \rceil)$ disk accesses.

Block Nested-Loop Join By modifying Algorithm 1 to iterate over blocks of R and S (instead of tuples), we obtain the block nested-loop join, which is summarized in Algorithm 2. The outermost two loops iterate over the blocks of R and S while the innermost two loops iterate over all pairs of tuples of R and S that are in the blocks currently in memory.

Algorithm 2 Block Nested-Loop Join

```
 1: procedure BNLJOIN(R,S,θ)
 2:     for all blocks B_R ∈ R do
 3:         read(B_R)
 4:         for all blocks B_S ∈ S do
 5:             read(B_S)
 6:             for all tuples r ∈ B_R do
 7:                 for all tuples s ∈ B_S do
 8:                     if rθs then
 9:                         print r||s
10:                     end if
11:                 end for
12:             end for
13:         end for
14:     end for
15: end procedure
```

While the block nested-loop join algorithm is a significant improvement over the earlier nested-loop al-

gorithm, there are several other methods for computing joins, such as the sort-merge join and the hybrid hash join, that are typically much more efficient. We refer the reader to Graefe's survey[2] for more information on these, as well as other topics related to query processing in databases. We note, however, that the block nested-loop join remains valuable because, unlike most other join methods, it makes no assumptions about the nature of the join predicate θ.

Queries Above, we have sampled some topics in the definition, use, and implementation of a specific query language, viz., relational algebra. We now address a fundamental question: What constitutes a query language or, more simply, a query? We present a brief answer here, referring the interested reader to a standard text[3] for much more on the topic: We define a *query* to be a mapping from database instances to database instances that is *well-typed*, *computable*, and *generic*. The requirement of well-typedness means, essentially, that the schema of the result of a query is fixed and cannot, for example, depend on the instance on which the query is evaluated. The requirement of computability refers to the standard notion of *Turing-computability* which states, essentially, that it must be possible to implement the mapping using a simple, standard model of computation that is well understood and that is equivalent to most traditional computers. (A notable exception, and one that leads to some intriguing results, is a quantum computer; however, we do not discuss it further here.) The requirement of genericity is the most interesting of the three and is discussed below.

Genericity To define genericity, we will use the idea of permutations of the domain: A *permutation* of the domain $\mathcal{D}$ is a one-to-one, onto mapping from $\mathcal{D}$ to $\mathcal{D}$. For example, if $\mathcal{D}$ is the set of integers, then the mapping $\rho_1(x) = x+1$, which maps each integer to its successor, is a permutation. However, the mapping $\rho_2(x) = |x|$, which maps an integer to its absolute value (unsigned magnitude), is not a permutation because it is not one-to-one (e.g., $\rho_2(-5) = \rho_2(5) = 5$). In addition, ρ_2 is not onto because, for instance, there is no x for which $\rho_2(x) = -5$. We say a permutation ρ is the identity on a set $C \subset \mathcal{D}$ if $\rho(x) = x$ for all x in C.

[2]Goetz Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys* 25/2 (June 1993): 73–170.

[3]Serge Abiteboul, Richard Hull, and Victor Vianu, *Foundations of Databases* (Addison-Wesley, 1995).

Intuitively, genericity requires that query results depend only on those relationships between values (domain items) that are explicated by relations in the database, and not on any other relationships resulting from implementation choices or other non-instance information. More precisely, consider two schemas, $\mathcal{S}_1$ and $\mathcal{S}_2$, and a finite set C of constants from the domain $\mathcal{D}$. A mapping q from instances of $\mathcal{S}_1$ to instances of $\mathcal{S}_2$ is called *C-generic* iff, for every instance I of $\mathcal{S}_1$ and every permutation ρ that is the identity on C, $q(\rho(I)) = \rho(q(I))$. We may think of the requirement that queries be generic as a generalization of the simpler idea of *data independence*, which states that results should not depend low-level implementation details such as, for example, whether integers are implemented using 32 or 64 bits.

Although genericity is a simple requirement, it has significant and often surprising consequences. One such consequence is the non-expressibility of the parity query using the simple relational algebra defined above: It is not possible to write a simple relational algebra query that returns a nonempty result if and only if the number of tuples in a relation is even. What is even more interesting is that the non-expressibility of the parity query persists even if we add additional constructs, such as recursion, which allow us to express seemingly more complex queries, such as transitive closure (reachability in a network). In practical query languages, this problem is typically fixed by including aggregation functions (sum, average, etc.) and other constructs.