Query Processing: Selection Operation

Smarterator <- Selection(Smarterator, predicate tree)

The predicate tree is structured in a way such that the leafs are predicates, and the nodes pointing to leafs (or other nodes) are conjunctive (AND) or disjunctive (OR) operations to be performed on these predicates. If the height of the tree is 1, we know we just need to evaluate one predicate, and thus simple selection will be used. If the height of the tree is greater than 1, we will use complex selection. Below, we describe what goes on under the hood of this function set, and give some insight onto how the relation is processed by the set of selection functions.

Simple selection is defined here as the process of subsetting a relation through the comparison of two values via a binary operation contained in the set \{<, <=, =, !=, >=, >\}.

Depending on the predicate, the selection function decides which algorithm to use. We can ask information about the attribute from the metadata. For query of type \(\sigma_{A=b}(R)\), if attribute A is not indexed, we use Linear Search (A1). In this case, we ask the BufferManager to return an iterator over the pages that contain the relation, and then iterate through every tuple within every page, checking if a tuple satisfies the condition (as long as A is not a key). If the attribute is a key, we search through tuples until we find the single tuple that satisfies the condition \(A=b\).

If there is a primary index on A and the attribute is a key, we can use A2. In this case, we ask the IndexManager to return a smarterator. This returns a smarterator with one tuple.

If there is a primary index on A and the attribute is a nonkey, we use A3. A3 is the same as A2, but can return multiple records.

If there is a secondary index on A and the attribute is a key, we use A4. In this case, we still ask the IndexManager to return a smarterator that contains a single tuple. When it comes to nonkey attributes, we use A1.

On comparisons with a secondary index, we will use A1. We will iterate either from the beginning or end until the value we are comparing (A <, >, <=, >= value).
On comparisons with a primary index, we will use A5 which asks the IndexManager for an iterator from either the beginning or end until the value in comparison (A <, >, <=, >= value).

*Complex selection* is defined here as the process of subsetting a relation through multiple comparisons of two values via binary operations, which relate to each other through either conjunctive (AND) or disjunctive (OR) operations. The selection function determines which algorithm to choose depending on the type of operation (conjunctive vs disjunctive).

On conjunctive operations, we first look at whether an access path is available for all attributes with predicates passed to our function. If one exists, we use algorithm A9, which uses the access paths to compute each predicate, optimizing our algorithms A2 through A6 to determine which we use for each predicate. Once we have all the pointers to the data, we sort each predicate’s pointers and take the intersection of them, and return the smarterator contained within that intersection.

If not all attributes have access paths, we check to see if at least one does. If we find an attribute with an access path, we use algorithm A7, which involves computing the predicate on that attribute (utilizing one of A2 through A6), load it into memory, and compute the remaining predicates on the result in memory. If none of the attributes have access paths, we must perform linear search.

If we are performing disjunctive selection, we implement algorithm A10, which is identical to A9 except we perform the union over the sets of pointers instead of the intersection.

We implement lazy fetching as our proxy design pattern for the situations in which our memory allocation for our BufferManager is not large enough to hold the entire segment of data we are reading in.