One-Pass Evaluation of Region Algebra Expressions

Matthew Young-Lai and Frank Wm. Tompa

Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1

Abstract

A region algebra is a collection of operations that apply to lists of regions — a useful model for querying text databases. In this paper, we address the problem of evaluating a region algebra expression. A naive method is to evaluate the operations one at a time, storing intermediate results and reading them back in as needed. We describe an alternative method that merges the argument region lists and performs the evaluation using a single pass over this merged list. This is often more efficient than evaluating the operations one at a time, since it visits every input region exactly once, and thus avoids writing intermediate results and then re-reading them as inputs for subsequent operations.

Key words: text database, region algebra, expression evaluation, query optimization

1 Introduction

Consider a string $x_1 x_2 \ldots x_n$ where each x_i is a character from an alphabet Σ. A region a is a pair $(a.l, a.r)$ of natural numbers representing the substring $x_{a.l} x_{a.l+1} \ldots x_{a.r-1} x_{a.r}$ ($a.l$ must be less than or equal to $a.r$). A *region algebra* is a set of functions, each of which takes one or more lists of regions as arguments, and returns a list of regions as a result. Typically, the region lists on which an algebra operates are restricted in some way so that their size is at most $O(n)$. For example, regions in a list may not be allowed to nest or overlap. However, we do not initially assume any such restrictions.

There are many examples of region algebras described in the literature (Salminen and Tompa, 1992; Burkowski, 1994; Kilpeläinen and Mannila, 1993; Clarke et al., 1995; Navarro, 1995). Such algebras are a very good indexing and query model for structured documents, which are becoming increasingly prevalent because of the World Wide Web. Since traditional database models are not well suited to structured text, region algebras have emerged as the favored alternative.

The usual way of evaluating a region algebra expression is to evaluate the function calls one at a time in some sequential order. Query "optimization" strategies can be used to try to minimize the size of intermediate results by choosing the order intelligently, or by transforming the expressions prior to evaluation (Navarro, 1995; Consens, 1998). In this paper we introduce a method that avoids the need to store and retrieve intermediate results. Rather, it uses a single pass that visits each input region exactly once.

2 Structure Selection Queries and Expressions

We define a *structure selection query* $Q(S, D)$ to be a function with two list arguments S and D that selects a subset of the regions from S according to a dependency on the regions in D. Structure selection queries are expressed in the form $\{s \in S \mid C\}$, where C is a boolean expression that consists of one or more clauses joined by the logical operators $\wedge$ and $\vee$, using brackets to specify precedence. Each clause is of the form $(x \odot y)$ where $x \in \{s.l, s.r\}$, $\odot \in \{=, <, >, \neq, \leq, \geq\}$, and $y \in \{d.l, d.r\}$, or more generally y is an arithmetic expression involving $d.l$ or $d.r$. The overall boolean expression C is qualified by either $\exists d \in D$ or $\nexists d \in D$.

An example of a structure query is CONTAINED-IN(S,D) which returns $\{s \in S \mid \exists d \in D \quad (s.l \geq d.l) \wedge (s.r \leq d.r)\}$. This selects regions in S that are inside a region from D. Similarly, NOT-CONTAINING(S,D) returns $\{s \in S \mid \nexists d \in D \quad (s.l \leq d.l) \wedge (s.r \geq d.r)\}$, which selects regions in S that do not contain a region from D. An example of a function that uses a clause with an arithmetic expression involving $d.l$ is STARTS-SOON-AFTER(S,D), which returns $\{s \in S \mid \exists d \in D \quad (s.l > d.r) \wedge (s.l < d.r + 100)\}$. This selects s regions that have a left end less than 100 characters after the right end of a d.

From this point on, we use the convention that, unless otherwise specified, the arguments to a query are always named S and D. Furthermore, we assume that s is a region in S and d is a region in D. This allows us to specify query definitions unambiguously by writing just the constraint C. We also assume the presence of $\exists d$ if no existential qualifier is specified. For example, CONTAINED-IN() is written $(s.l \geq d.l) \wedge (s.r \leq d.r)$. When we talk about more than one query at a time, or specific calls to queries, we introduce a subscript, and refer to separate queries as Q_1, Q_2 etc. In this case, the arguments are $S_1, S_2, \ldots$, and $D_1, D_2, \ldots$, and regions from the arguments are $s_1, s_2, \ldots$ and

$d_1, d_2, \ldots$.

We assume that any function included in a region algebra depends on all its arguments. For structure selection queries, this means that C must not be a tautology that always selects s, nor a contradiction that always fails to select it. We can efficiently recognize such a badly defined structure selection query:

Lemma 1 *Given a structure selection query defined by a boolean expression C with m clauses, we can determine whether C is a contradiction or tautology in $O(m^2)$ time.*

PROOF. All clauses refer to points on the same domain (character positions in the string), and compare either $s.l$ or $s.r$ to some expression involving $d.l$ or $d.r$. To be a valid query, there must be at least one assignment of $s.l$ and $s.r$ making the expression true, and at least one making it false. If there are m different clauses, then there are at most $2m+1$ relevant assignment classes of $s.l$ or $s.r$ to test. These are the m points that $s.l$ and $s.r$ are compared to, the $m-1$ regions between them, the region from the start of the string to the first point, and the region from the last point to the end of the string. Thus there are exactly $\binom{2m+1}{2} + 2m + 1$ assignments of interest. If all of these are true or false then the expression is a tautology or contradiction.

The composition of a query Q_1 with another query Q_2 is found by passing the result of one call as an argument to the other. So, for example, the statement $Q_2(S_2, Q_1(S_1, D_1))$ means perform query Q_1 on S_1 and D_1, and then perform Q_2 on S_2 and the result.

Recall that the set returned by a query Q_2 is specified in the form $\{s_2 \in S_2 \mid C_2\}$ where C_2 is a boolean expression involving s_2 and d_2 with an existential qualifier on d_2. When we compose two functions Q_2 and Q_1 by passing the result of Q_1 as an argument to Q_2, the returned set is $\{s_2 \in S_2 \mid C_2 \wedge C_1\}$ with all instances of s_2 and S_2 replaced by s_1 and S_1 if the result of Q_1 is passed as the first argument to Q_2, or all instances of d_2 and D_2 replaced by s_1 and S_1 if it is passed as the second argument. For example, the set returned by the composition NOT-CONTAINING(S_2, CONTAINED-IN(S_1, D_1)) is

$$\{s_2 \in S_2 \mid (\nexists d_2 \in D_2 \quad (s_2.l \leq d_2.l) \wedge (s_2.r \geq d_2.r)) \quad \wedge$$

$$(\exists d_1 \in D_1 \quad (s_1.l \geq d_1.l) \wedge (s_1.r \leq d_1.r))\}$$

where D_2 is the result of the CONTAINED-IN() call and therefore a subset of S_1. Thus we replace instances of d_2 and D_2 by s_1 and S_1 to give

$$\{s_2 \in S_2 \mid (\nexists s_1 \in S_1 \quad (s_2.l \leq s_1.l) \wedge (s_2.r \geq s_1.r)) \quad \wedge$$

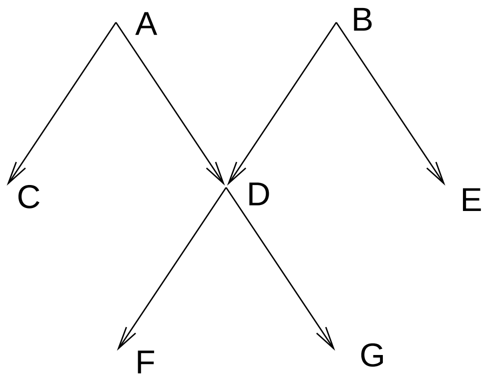

Fig. 1. An expression graph that is a DAG.

$$(\exists d_1 \in D_1 \quad (s_1.l \geq d_1.l) \wedge (s_1.r \leq d_1.r))\}$$

For a query Q_2 where the first argument is the result of a query Q_1 and the second is the result of a query Q_0, the set specification is of the form $\{s_2 \in S_2 \mid C_2 \wedge C_1 \wedge C_0\}$ with instances of s_2 and S_2 replaced by s_1 and S_1, and instances of d_2 and D_2 replaced by s_0 and S_0. Applying these rules recursively, we can write out the set specification for the result of any query in an expression.

We define the *dependency expression* for a set specification to be the boolean expression component with existential qualifiers removed. For a query Q, we denote this $E(Q)$. For example, E(CONTAINED-IN) is $(s.l \geq d.l) \wedge (s.r \leq d.r)$. We denote the dependency expression for a query Q in an expression graph G, $E_G(Q)$. For example, E_G(NOT-CONTAINING) for the graph defined by NOT-CONTAINING(S_2,CONTAINED-IN(S_1, D_1)) is

$$((s_2.l \leq s_1.l) \wedge (s_2.r \geq s_1.r)) \wedge ((s_1.l \geq d_1.l) \wedge (s_1.r \leq d_1.r))$$

A dependency expression can be seen as a general description of the possible positions of a d region that may affect whether a given s region satisfies a query. For example, letting $\Rightarrow$ denote logical implication, we can see that E(CONTAINED-IN) $\Rightarrow$ $(d.l \leq s.r)$ since $s.l \leq s.r$ for any region. In other words, any d match relevant to the decision whether to keep a given s match must start before the s ends.

An *algebra expression* is a composition of two or more queries. Define the *expression graph* representation of an algebra expression as follows: each query is represented by a node, and if one query Q_1 takes the result of another query Q_2 as an argument, then there is an edge from the node for Q_1 to the node for Q_2. The *height* of a node in an expression graph is the number of nodes on the longest path from that node to a leaf.

An expression graph must be acyclic. That is, a function call cannot use its own result as an argument, nor can its result be used by any function which contributed to its evaluation. An expression graph need not be connected, nor does it have to be a tree, however. A multi-rooted graph represents a set of expressions rather than just one. The directed acyclic graph (DAG) in Figure 1, for example, represents two expressions with a common subexpression rooted at D.

3 Bottom-Up Evaluation

The input to an expression graph is the set of arguments to the queries at the leaf nodes. The result of evaluating an expression graph is the set of results of the queries at the root nodes. The results of queries that are not roots are the intermediate results of subexpressions.

In bottom-up evaluation, the queries at the leaf nodes are evaluated one by one by reading in the two region lists corresponding to the arguments and writing out the region list corresponding to the result. Queries at non-leaf nodes can be evaluated any time after all their arguments have been evaluated.

Using such an evaluation method, a region in the input "travels" through the expression graph, being read as an input to a query, and possibly written as a result, at each step. A region in the final output of the expression graph travels all the way from a leaf to a root in this way. Other regions may travel only the first part of such a path before being discarded. Overall, however, a region is read and written once for every step [1]. As mentioned in the introduction, techniques can be used to reduce the number of repeat visits by transforming the graph prior to evaluation. However, we propose a different approach that eliminates repeat visits completely.

4 One-Pass Evaluation

We now describe an evaluation strategy that differs significantly from the bottom-up approach. Recall that an expression graph corresponds to a function from one set of region lists to another. An alternative way of viewing a set of region lists is as a single, integrated list of regions, each tagged by a type that indicates the list in which the region is located. We refer to such a

[1] We can avoid outputting intermediate results and then inputting them again in some cases by keeping them in memory. However, we are interested here in cases where memory requirements are independent of database size, so that the algebra can be used for very large text collections.

typed list as a *region inventory.* Thus, the inputs and outputs of a function corresponding to an expression graph are both region inventories. Note that for expressions formed by composing structure selection queries, the output region inventory is a subset of the input.

4.1 Forward Passes

Suppose we have an input region inventory of M regions, and that these are stored sorted primarily by left-end or by right-end (left-sorted or right-sorted). We do not specify a secondary sort order for reasons explained later. Consider the following loop which iterates the region inventory in its sort order:

for i **from** 1 **to** M
  process region i

Processing a region involves reading it from the input, and then immediately outputting it if it is part of the expression graph's output. Depending on the sort order, we refer to the loop as either a *forward deterministic pass with left-sorted input*, or a *forward deterministic pass with right-sorted input.* Here we are using the term *deterministic* as an analogy with deterministic finite transducers: in the same way that a deterministic finite transducer never needs to look ahead to a later character to choose which transition to take, a deterministic pass never needs to look ahead to a later region to decide whether to output the current region. Also, it cannot buffer alternative outputs, or choose one and then backtrack later.

Note that it is not necessary to store an input region inventory directly as a list of typed regions to use such a loop. Rather, we can store separate lists, and logically merge them at the same time as we execute the loop. Similarly, we can split the output region inventory into separate lists while executing the loop. One or both of these strategies might be appropriate depending on the form in which the input is available, and what form of output is needed. In particular, separate lists are the appropriate representation for using a region algebra interactively, or for situations where it is unknown whether a left-sorted or right-sorted input may be required.

Because it is deterministic and iterates the regions in a fixed order, the loop cannot be used to evaluate arbitrary expressions. In general, for example, it cannot be used to evaluate expressions that use both CONTAINING() and CONTAINED-IN() calls, as we will now prove.

Consider the regions shown in Figure 2. Suppose a must be contained in b, b must contain c, c must be contained in d and d must contain e. Suppose

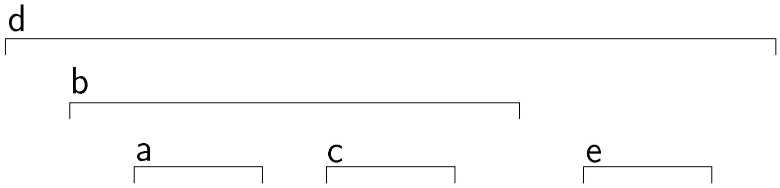

Fig. 2. A list of regions related by both CONTAINED-IN() and CONTAINING().

we iterate the list in left-sorted order: d, b, a, c, e. When we visit e we know it should be kept since it has no dependencies. However, we do not know whether to keep d without looking ahead to the point where we visit e. By extension, we do not know whether to keep b, a, and c since they indirectly depend on e. Now consider a right-sorted order: a, c, b, e, d. In this case, we read e before d and therefore know to keep both of these regions. However, we read c before d and therefore do not know whether to keep it or a and b since they depend on c.

In general, the problem is that arbitrary compositions of CONTAINED-IN() and CONTAINING() can require lookahead regardless of the sort order of the pass. We need to characterize the conditions under which an expression does not require lookahead.

We start by considering individual queries that are not part of expressions. An individual query requires no lookahead if the decision to keep or discard a region s is always based on a region d that is read before s in the loop.

Lemma 2 *A single query Q requires no lookahead with left-sorted input iff $E(Q) \Rightarrow (d.l < s.l)$.*

PROOF. If $E(Q) \Rightarrow (d.l < s.l)$ then the decision whether to keep a given s always depends on a d with an earlier left end. If the input is left-sorted, then this d is read before s. Therefore, no lookahead is needed.

If no lookahead is required, then the d needed to decide whether to keep a given s is read before that s. This means that $E(Q)$ must relate s to a d that is guaranteed to have an earlier left end. In other words, $E(Q) \Rightarrow (d.l < s.l)$.

For example, $E(Q)$ for the query $(s.l > d.l) \wedge (s.r < d.r)$ implies that $(d.l < s.l)$. However, $E(Q)$ for the query $(s.r = d.l)$ makes no such implication. Therefore, we can evaluate the first query using a forward deterministic pass with left-sorted input, but not the second.

If regions have equal left ends in a left-sorted input, then either we must make no assumptions about their relative ordering, or we have to consider a secondary sort order to decide whether d comes before s in the pass. We avoid this issue by assuming that regions with equal left-ends are read and buffered before any processing, meaning we can treat them as if they are read simultaneously. When using this strategy, the lemma is:

Lemma 3 *A single query Q requires no lookahead with left-sorted input iff $E(Q) \Rightarrow (d.l \leq s.l)$.*

The corresponding lemma for right-sorted input is as follows. Again, we assume that equal regions according to the sort order are read and buffered before any processing. The proof is similar to the last lemma.

Lemma 4 *A single query Q requires no lookahead with right-sorted input iff $E(Q) \Rightarrow (d.r \leq s.r)$.*

We now consider multiple queries. We can evaluate all queries in an expression graph G using a forward deterministic pass iff for every query Q in G, the dependency expression $E_G(Q)$ does not imply lookahead. With left-sorted input, for example, $E_G(Q)$ must imply that $(x.l \leq s.l)$ for every region x referenced in $E_G(Q)$. In other words, all regions needed to decide whether a given s satisfies a query must be visited before s. Note that this assumes that every x list referenced in $E_G(Q)$ actually does affect the result of the query, which follows directly from our assumption that functions depend on all of their arguments.

We now show that $E_G(Q)$ implies no lookahead for every query in G exactly when $E(Q)$ implies no lookahead for every query in G:

Theorem 5 *We can evaluate an expression graph G made up of structure selection queries using a forward, deterministic pass with left-sorted input iff for every query Q in $G, E(Q) \Rightarrow (d.l \leq s.l)$.*

PROOF. First we prove the necessity of the condition. If $\exists Q \in G$ such that $E(Q) \not\Rightarrow (d.l \leq s.l)$, then by Lemma 3, the query requires lookahead to calculate.

Next we prove the sufficiency of the condition through induction on h, the height of a query in G. For the base case of $h = 1$ (any single query Q), the dependency expression $E(Q)$ implies $(d.l \leq s.l)$ by the requirement of the theorem and therefore requires no lookahead. The induction hypothesis is that any query of height at most $h-1$ has a dependency expression $E_G(Q)$ that implies $(x.l \leq s.l)$ for every x in $E_G(Q)$.

For the induction step, we consider a query Q_0 of height h. The induction hypothesis applies to all descendants of Q_0 since they have heights less than or equal to $h-1$. Now, build the dependency expression $E_G(Q_0)$ in the usual way starting with $E(Q_0) \wedge E_G(Q_l) \wedge E_G(Q_r)$ where Q_l is the left child of Q_0, and Q_r is the right child of Q_0, and replacing instances of s_0 in $E(Q_0)$ by s_l, and instances of d_0 by s_r. Next we perform the induction step. We know that $E_G(Q_l) \Rightarrow (x.l \leq s_l.l)$ for all x in $E_G(Q_l)$ by the induction hypothesis. Also, we know that $E_G(Q_r) \Rightarrow (y.l \leq s_r.l)$ for all y in $E_G(Q_r)$ by the induction hypothesis. Finally, by the condition of the theorem, we know that $E(Q_0) \Rightarrow (d_0.l \leq s_0.l)$ which, after the substitution, is $E(Q_0) \Rightarrow (s_r.l \leq s_l.l)$. Thus $E_G(Q_0) \Rightarrow (x.l \leq s_l.l)$ for every x in $E_G(Q_0)$, from which it follows that the result of Q_0 can be calculated with no lookahead.

Theorem 6 *We can evaluate an expression graph G made up of structure selection queries using a forward, deterministic pass with right-sorted input iff $\forall Q \in G, E(Q) \Rightarrow (d.r \leq s.r)$.*

PROOF. The proof is of the same form as that of the previous theorem.

Consider an application of the theorems. The queries $(s.l \geq d.l) \wedge (s.r \leq d.r)$ and $(d.r = s.l)$ both imply $(d.l \leq s.l)$, which is the condition of Theorem 5. The theorem therefore states that, using one deterministic pass, we can evaluate *any* expression formed by composing calls to these two queries, regardless of how large.

We use the conditions of the theorems as the basis for a categorization of query functions. Henceforth, we refer to functions that satisfy $E(Q) \Rightarrow (d.l \leq s.l)$ as *condition 1 queries for a forward pass*, and those that satisfy $E(Q) \Rightarrow (d.r \leq s.r)$ as *condition 2 queries for a forward pass*. Note that a function can be in both categories. For example, the function $(d.r = s.l)$ is both condition 1 and condition 2.

Now consider a slight variation of the forward deterministic pass. Suppose the input is left-sorted, but instead of outputting every region immediately in its left-sorted order, we buffer regions temporarily so as to give a right-sorted output. For an individual query Q to require no lookahead in this case, it is necessary that all regions on which s depends be read before s is *output*. That is, unlike the normal forward deterministic pass, some of the regions s depends on may be visited after s is visited. However, in this case, they must be visited during the time that s is buffered. Since s is read in its left-sorted position, and buffered until its right-sorted position, evaluating a query Q in an expression graph G requires that $E(Q)$ implies $(d.l \leq s.r)$. We call a query that satisfies this implication a *condition 3 query for a forward pass*. To evaluate a condition 3 query Q as part of an expression graph G, $E_G(Q)$ must imply $(x.l \leq s.r)$ for all x regions in $E_G(Q)$.

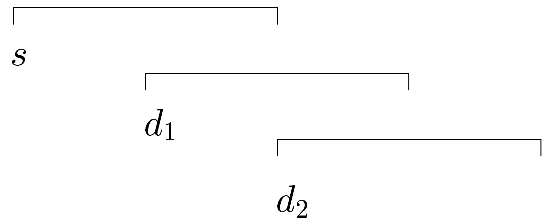

Fig. 3. Three overlapped regions.

It is *not* true that we can evaluate all queries in an expression made up of condition 3 queries using a deterministic pass with left-sorted input and right-sorted output. Consider the three regions in Figure 3. Suppose that s depends on d_1 and d_1 depends on d_2. It is true that $(d_1.l \leq s.r)$ and that $(d_2.l \leq d_1.r)$, but it is not true that $(d_2.l \leq s.r)$. Therefore, we cannot decide whether to keep s without using lookahead in a forward pass with left-sorted input and right-sorted output. The problem is that the condition 3 implication is not transitive. Therefore, not all expressions composed of condition 3 queries can be evaluated using a deterministic pass.

We can now state the conditions under which we know for sure that *all* expressions constructible using queries from a given region algebra are evaluable with a forward deterministic pass with properly sorted inputs and outputs:

Theorem 7 *For a set $\mathcal{Q}$ of query functions, we can evaluate every expression formed with queries from this set using a forward, deterministic pass with properly sorted input and output iff all functions are condition 1 or all functions are condition 2.*

PROOF. Sufficiency of the condition is established by the preceding theorems. That is, we know that we can evaluate all expressions made up entirely of condition 1 or condition 2 functions using a forward, deterministic pass.

To show the necessity of the condition, we prove that, given any set of functions that are not all condition 1 and not all condition 2 it is always possible to construct an expression that requires lookahead to compute. This is true for a pass with left-sorted input and left-sorted output by Theorem 5, and true for a pass with right-sorted input and right-sorted output by Theorem 6. It is not possible for an iteration to use right-sorted input and left-sorted output since this may require that regions be output before they are read. The only remaining option is left-sorted input and right-sorted output. For all queries in a given expression graph G to require no lookahead with this option, $E_G(Q)$ must imply $(x.l \leq s.r)$ for every Q in G.

Suppose we have a query Q_1 such that $E(Q_1) \not\Rightarrow (d.l \leq s.l)$, and another query Q_2 such that $E(Q_2) \not\Rightarrow (d.r \leq s.r)$. That is, Q_1 is not condition 1 (although it may be condition 2), and Q_2 is not condition 2 (although it may be condition

Fig. 4. A situation that requires lookahead.

1). Consider the composition $Q_2(S_2, (Q_1(S_1, D_1))$. For a region $s_1 \in S_1$, the decision whether s_1 satisfies Q_1 may require knowledge of a region $d_1 \in D_1$ with $d_1.l > s_1.l$ since $E(Q_1) \not\Rightarrow (d.l \leq s.l)$. Similarly, the decision whether a region $s_2 \in S_2$ satisfies Q_2 may require knowledge of a region $d_2 \in D_2$ with $d_2.r > s_2.r$ since $E(Q_2) \not\Rightarrow (d.r \leq s.r)$. Because D_2 is the result of Q_1, which is a subset of S_1, this means that determining whether s_2 satisfies Q_2 can depend on a region $s_1 \in S_1$ where $s_1.r > s_2.r$. Thus we can have the situation shown in Figure 4. The decision whether s_2 satisfies Q_2 is based on an s_1 where $(s_1.r > s_2.r)$, and the decision whether s_1 satisfies Q_1 is based on a d_1 where $(d_1.l > s_1.l)$. Since these conditions are consistent with $(d_1.l > s_2.r)$, we cannot calculate Q_2 without lookahead.

4.2 Reverse Passes

Now consider a reverse deterministic pass which visits regions in descending left-sorted or right-sorted order:

for i **from** M **to** 1
  process region i

All of the preceding results can be adapted to this case:

Corollary 8 *For a set $\mathcal{Q}$ of query functions, we can evaluate every expression formed with queries from this set using a* reverse*, deterministic pass with properly sorted input and output iff all functions $Q \in \mathcal{Q}$ satisfy $E(Q) \Rightarrow (d.r \geq s.r)$ or all functions satisfy $E(Q) \Rightarrow (d.l \geq s.l)$.*

PROOF. The proof is symmetrically the same as Theorem 7, with all occurrences of left and right swapped.

We refer to functions satisfying $E(Q) \Rightarrow (d.r \geq s.r)$ as *condition 1 queries for a reverse pass*, and those satisfying $E(Q) \Rightarrow (d.l \geq s.l)$ as *condition 2 queries for a reverse pass.* The classification of a function is not necessarily the same for passes in both directions. For example, $(d.l < s.l) \wedge (d.r > s.l) \wedge (d.r < s.r)$

is both condition 1 and condition 2 for a forward pass, but neither condition 1 nor condition 2 for a reverse pass.

4.3 Examples

In general, any structure selection query can be written as a set specification and then classified according to the implied order of s and d. For example, CONTAINED-IN() is condition 1 in either direction and therefore, according to the above results, we can evaluate any expression formed using CONTAINED-IN() queries with either a forward deterministic pass or a reverse deterministic pass. Similarly, CONTAINING() is condition 2 in either direction and we can evaluate arbitrary expressions with a forward or reverse deterministic pass. However, we cannot evaluate arbitrary expressions formed by composing CONTAINED-IN() and CONTAINING() with a deterministic pass since this mixes condition 1 and condition 2 queries as in the example of Figure 2.

Another example of a query is INTERSECT(S,D), which performs set intersection, returning all regions from S that are also in D. The set specification is $(s.l = d.l) \wedge (s.r = d.r)$, which is both condition 1 and condition 2 in either direction. Another set query is SUBTRACT(S,D) which returns all regions from S that are not in D. The set specification is $\nexists d(s.l = d.l) \wedge (s.r = d.r)$, which is also both condition 1 and condition 2 in either direction.

As a final example, consider AFTER(), which has the set specification $(s.l = d.r)$. This finds all regions in S that follow and abut a region in D. It is both condition 1 and condition 2 for a forward pass, but neither for a reverse pass. The opposite function BEFORE() with the set specification $(s.r = d.l)$ is condition 1 and condition 2 for a reverse pass, but neither for a forward pass.

Overall, if we have an algebra containing functions that are all of the same classification, then we can evaluate *any* expression formed with that algebra using a deterministic pass. For example, CONTAINING(), INTERSECT(), SUBTRACT(), and AFTER() are all condition 1 in a forward direction. Therefore, we can define an algebra consisting of just these functions, and evaluate any expression using a forward deterministic pass with left-sorted input. Another example is the algebra consisting of CONTAINED-IN(), INTERSECT(), SUBTRACT(), and BEFORE() which are all condition 2 in a reverse direction. We can evaluate any expression in this algebra using a reverse deterministic pass with left-sorted input.

4.4 Efficiency

The memory required to evaluate an expression with a deterministic pass is equal to the sum of the memory required to evaluate each query in the expression. All the queries we have introduced as examples use constant memory, so the memory requirement of evaluating expressions made up of such queries is linear in the size of the expression.

Additional memory is needed to satisfy our assumption that equal regions according to the sort order are buffered before processing. In general, the buffering required for this can be arbitrary. However, we can use an additional condition to eliminate this problem. Consider a left-sorted region inventory formed by merging m lists. Suppose that each of the m lists contains no regions with equal left ends, a condition that is enforced by PAT (Salminen and Tompa, 1992) and MultiText (Clarke et al., 1995), for example. Then, at most m regions at a time can share a left end in the overall region inventory. The number of lists m is linearly bounded by the number of queries in the expression graph. Therefore, the amount of buffering remains linearly bounded by the size of the expression graph. Note that the same argument holds if merging of the m lists into a region inventory is done on the fly.

Consider again using a bottom-up evaluation strategy for an expression graph. For each query, we read the arguments and output the results. Except for final results, these then have to be read again as inputs to another query. This contrasts with evaluating an expression using a deterministic pass, since that way, no intermediate results need to be written and then read again. That is, every input region is visited exactly once. The time saving of using a one-pass evaluation will vary depending on the expression and the data. Overall, the larger the size of the intermediate outputs using a bottom-up evaluation, the more saving will be realized with a one-pass evaluation.

5 General Functions

None of the preceding results depend on the functions in the expression being structure selection queries. In fact, an arbitrary function from region lists to region lists can be classified as condition 1 or condition 2 in the same way as a structure selection query: write out a set specification that describes an output region in terms of the region on which it depends, and then check their implied order in the region inventory. For general functions, there is the possibility that an output depends on more than one other region. In this case, the implication must hold between the output and every region on which it depends.

For example, consider a function MERGE-OVERLAPS(S,D) that finds pairs (s, d) such that $(s.l < d.l) \wedge (s.r > d.l) \wedge (s.r < d.r)$ and outputs a region running from $s.l$ to $d.r$ for each such pair. This description implies that the right end of both s and d is less than or equal to the right end of the output. Therefore, the function is condition 2 for a forward pass. The left ends of both s and d are also greater than or equal to the left end of the output. Therefore the function is also condition 2 for a reverse pass. Therefore, we can evaluate any expression constructed with this and other condition 2 functions using a deterministic pass.

6 Using a One-Pass Evaluation Method

The simplest way to use a one-pass evaluation method is to design an algebra consisting entirely of condition 1 functions, or entirely of condition 2 functions for the same pass direction. This ensures that any expression that can be constructed with the algebra can be evaluated using the one-pass method. Unfortunately, it limits the mix of functions that can be included. For example, only one of CONTAINED-IN() and CONTAINING() can be used.

A somewhat less restrictive option is to break the algebra into groups of functions where each group is entirely condition 1 or condition 2 for the same pass direction. Then we only allow the user to construct expressions using functions from one group at a time. In this way, functions from different groups can be used together by constructing separate expressions and using the output of one expression as the input to another.

A final, even less restrictive option is to break the algebra into groups of functions as above, but allow the user to construct expressions that arbitrarily mix functions from different groups. Then, the system can automatically break expressions into subexpressions containing functions from the same group. Each such subexpression can then be evaluated with one-pass, and the output of intermediate subexpressions used as the inputs to other sub-expressions. This strategy could be used as the basis of an optimization method together with existing methods to transform the graph before evaluation. Rather than only attempting to minimize the size of intermediate results however, we could also attempt to minimize the number of required passes. Further work is needed to develop this idea into a robust optimization method.

7 Summary and Future Work

We have described an evaluation method for region algebra expressions that uses a single deterministic pass over an input region inventory. This method can be more efficient than bottom-up evaluation of an expression since it visits each input region only once, eliminating the need to output intermediate results and then read them back in. It is defined in terms of a general way of classifying functions, rather than only considering a fixed set of functions from typical algebras.

Future work is needed to examine alternative forms of one-pass evaluation. For example, there may be different rules restricting the design of an algebra where the evaluation method allows constant lookahead in the region inventory rather than no lookahead. Also, we wish to explore evaluation methods that use input and output region inventories that are not strictly left-sorted or right-sorted. For example, suppose all regions of a given type are displaced by a constant number of positions from their left or right sorted positions, or suppose that regions of each type are sorted within a region inventory but there is no restriction on the ordering of regions of different types. Overall, an exploration of such cases might yield alternative methods with the same efficiency as the one described in this paper, but fewer restrictions on the permissible functions.

References

Burkowski, F. J., 1994. An algebra for hierarchical organized text-dominated databases. Information Processing and Management 28, 313–324.

Clarke, C., Cormack, G., Burkowski, F., 1995. An algebra for structured text search and a framework for its implementation. Computer Journal 38 (1), 43–56.

Consens, M. P., 1998. Algebras for querying text regions: Expressive power and optimization. Journal of Computer and System Sciences 57, 272–88.

Kilpeläinen, P., Mannila, H., 1993. Retrieval from hierarchical texts by partial patterns. In: Korfhage, R., Rasmussen, E., Willett, P. (Eds.), SIGIR '93 – Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. New York.

Navarro, G., 1995. A language for queries on structure and contents of textual databases. In: Proceedings of the 18th SIGIR.

Salminen, A., Tompa, F., 1992. PAT expressions: an algebra for text search. Acta Linguistica Hungarica 41 (1–4), 277–306.