

# Expressiveness and Performance of Full-Text Search Languages

Chavdar Botev<sup>1</sup>, Sihem Amer-Yahia<sup>2</sup>, and Jayavel Shanmugasundaram<sup>1</sup>

<sup>1</sup> Cornell University  
Ithaca, NY 14853, USA  
cbotev@cs.cornell.edu  
jai@cs.cornell.edu  
<sup>2</sup> AT&T Labs–Research  
Florham Park, NJ 07932, USA  
sihem@research.att.com

**Abstract.** We study the expressiveness and performance of full-text search languages. Our motivation is to provide a formal basis for comparing full-text search languages and to develop a model for full-text search that can be tightly integrated with structured search. We design a model based on the positions of tokens (words) in the input text, and develop a full-text calculus (FTC) and a full-text algebra (FTA) with equivalent expressive power; this suggests a notion of completeness for full-text search languages. We show that existing full-text languages are incomplete and identify a practical subset of the FTC and FTA that is more powerful than existing languages, but which can still be evaluated efficiently.

## 1 Introduction

Full-text search is an important aspect of many information systems that deal with large document collections with unknown or ill-defined structure. The common full-text search method is to use simple keyword search queries, which are usually interpreted as a disjunction or conjunction of query keywords. Such queries are supported by traditional full-text search systems over “flat” text documents [1], over relational data [2, 16], and more recently over XML documents [9, 13, 25]. Many new and emerging applications, however, require full-text search capabilities that are more powerful than simple keyword search. For instance, legal information systems (e.g., LexisNexis®)<sup>3</sup> and large digital libraries (e.g., such as the Library of Congress (LoC))<sup>4</sup> allow users to specify a variety of full-text conditions such as the ordering between keywords and keywords distance. For example, a user can issue a query to find LoC documents that contain the keywords “assignment”, “district”, and “judge” in that order, where the keywords “district” and “judge” occur right next to each other (i.e., within a distance of 0 intervening words), and the keyword “judge” appears within 5 words of the keyword “assignment”. In a recent panel at SIGMOD 2005,<sup>5</sup> a librarian at the LoC mentioned

<sup>3</sup> <http://www.lexisnexis.com/>

<sup>4</sup> <http://thomas.loc.gov/>

<sup>5</sup> <http://cimic.rutgers.edu/sigmod05/>

that support for such “structured” full-text queries is one of the most important requirements for effectively querying LoC documents.

Given structured full-text queries, one of the practical problems that arises is being able to model, optimize and efficiently evaluate such queries. This problem has been studied for simple structured full-text queries in the information retrieval community [5], and more recently, for more complex structured full-text queries using text region algebras (TRAs) [10]. TRAs explicitly model keyword positions and pre-defined regions such as sentences and paragraphs in a document, and develop efficient evaluation algorithms for set operations between regions such as region inclusion and region ordering. While TRAs are an excellent first step, they have a fundamental limitation: they are not expressive enough to write certain natural structured full-text queries that combine inclusion and ordering of multiple regions. Further, since TRAs are based on a different algebraic model than the relational model, it is difficult to tightly integrate structured full-text search with structured search (which is usually relational).

To address the above issues, we propose a new model for structured full-text search. Specifically, we develop a Full-Text Calculus (FTC) based on first-order logic and an equivalent Full-Text Algebra (FTA) based on the relational algebra, and show how scoring can be incorporated into these models. Based on the FTC and FTA, we define a notion of completeness and show that existing query languages, including those based on TRAs, are incomplete with respect to this definition. The key difference that results in more expressive power for the FTA when compared to TRAs, is that the FTA deals with *tuples of one or more positions*, while TRAs only keep track of the start and end positions of a region during query evaluation, and lose information about individual keyword positions within regions. Further, since the FTA is based on the relational algebra, it can be tightly integrated with structured query processing.

Our next focus in the paper is on efficiency: since the FTA (or equivalently, the FTC) is based on the relational algebra, not all queries can be efficiently evaluated in time that is linear in the size of the input database. To address this issue, we identify PPRED, a practical subset of the FTC, which strictly subsumes TRAs. We also propose an algorithm that can efficiently evaluate PPRED queries in a *single pass* over inverted lists, which are a common data structure used in information retrieval. We also experimentally evaluate the performance of the algorithm.

In summary, the main contributions of this paper are:

- We introduce a new formal model for structured full-text search and scoring based on first-order logic (FTC) and the relational algebra (FTA), and define a notion of completeness for full-text languages (Section 2).
- We show that existing languages are incomplete with respect to the above definition of completeness (Section 3).
- We define a practical subset of the FTC and FTA called PPRED, which subsumes TRAs and can be evaluated in a single pass over inverted lists (Section 4).
- We experimentally study the performance of the PPRED algorithm (Section 5).

## 2 Related Work

There has been extensive research in the information retrieval community on the efficient evaluation of full-text queries [1, 23, 27], including structured full-text queries [5].

However, the work on structured full-text queries only develops algorithms for specific full-text predicates (such as window) in isolation. Specifically, existing proposals do not develop a fully composable language for many full-text predicates, and also do not study the expressiveness and complexity of the language. This observation also applies to XML full-text search languages such as XQuery/IR [4], XSEarch [9], XIRQL [13], XXL [25] and Niagara [29]. Our proposed formalism is expressive enough to capture the full-text search aspects of these existing languages, and is in fact, more powerful (see Section 4.1).

More recently, there has been some work on using text region algebras (TRAs) to model structured full-text search [7, 10, 17, 20, 22, 28]. A text region is a sequence of consecutive words in a document and is often used to represent a structural part of a document (e.g., a chapter). It is identified by the positions of the first and the last words in the region. TRAs operate on sets of text regions which may contain overlapping regions ([7]) or strict hierarchies ([20]). Common operators are the set-theoretic operators, inclusion between regions and ordering of regions [28] as defined below:

- A region  $s$  is represented as the ordered pair  $(s.l, s.r)$ , where  $s.l$  is the left end-point of the region, and  $s.r$  is its right end-point.
- A query operator has the form  $\{s \in S \mid \exists d \in D \text{ Pred}(s, d)\}$ , where  $S$  and  $D$  are sets of regions and  $\text{Pred}$  is a Boolean expression with the logical operators  $\vee$  and  $\wedge$  and containing clauses of the form  $(x \odot y)$ , where  $\odot \in \{=, <, >, \leq, \geq\}$ ,  $x \in \{s.l, s.r, s.l + \text{const}, s.l - \text{const}, s.r + \text{const}, s.r - \text{const}\}$ ,  $y \in \{d.l, d.r\}$ , and  $\text{const}$  is a constant.

Efficient algorithms have been devised to evaluate TRA queries. However, while TRAs are useful in a number of scenarios (e.g. search over semi-structured SGML and XML documents), they have limited expressive power. Consens and Milo [10] showed that TRAs cannot represent simultaneously inclusion and ordering constraints. For example, the query: find a region that contains a region  $s$  from a set  $S$  and a region  $t$  from a set  $T$  such that  $s$  comes before  $t$ , cannot be represented in TRAs. As we shall show in Section 4.2, similar queries arise in structured full-text search, for instance, when trying to find two windows nested inside another window.

Besides TRAs, there has also been a significant amount of work on using relational databases to store inverted lists, and in translating keyword queries to SQL [6, 12, 16, 18, 21, 29]; however, they do not study the completeness of languages and do not develop specialized one-pass query evaluation algorithms for structured full-text predicates.

### 3 The FTC and the FTA

Unlike SQL for querying relational data, there is no well-accepted language for expressing complex full-text search queries. In fact, many search systems use their own syntax for expressing the subset of complex queries that they support.<sup>6</sup> Instead of using one specific syntax, we adopt a more general approach and model full-text search

<sup>6</sup> <http://www.lexisnexis.com/>, <http://www.google.com>, <http://thomas.loc.gov>,  
<http://www.verity.com>

```

<html> <head> ... </head>
<body>
  <p>HR-212-IH (104)</p>
  <p><center>109th(105) Congress (106)</center></p>
  <h3><center>January(107) 4(108), 2005(109)
    </center></h3>
  ...
  <h3>SEC (404). 7(405). ASSIGNMENT(406) OF(407)
    CIRCUIT(408) JUDGES(409).</h3>
  <p><it> Each (410) circuit (411) judge (412) of (413)
    the (414) former (415) ninth (416) circuit (417)
    who (418) is (419) in (420) regular (421) active (422)
    service (423) ...</it></p>
</body>

```

Fig. 1. Positions Example

queries using calculus and algebra operations. Specifically, we use a Full-Text Calculus (FTC) based on first order logic and an equivalent Full-Text Algebra (FTA) based on the relational algebra.

The FTC and the FTA provide additional expressive power when compared to previous work on TRAS. The increased expressive power stems from the fact that the FTC and FTA deal with *tuples of positions*, instead of just start and end positions as in TRAS. Further, since the FTA is based on the relational algebra, it can be tightly integrated with structured relational queries.

### 3.1 Full-Text Search Model

We assume that full-text search queries are specified over a collection of *nodes* (which could be text documents, HTML documents, XML elements, relational tuples, etc.). Since our goal is to support structured full-text predicates such as distance and order, which depend on the position of a token (word) in a node, we explicitly model the notion of a *position* that uniquely identifies a token in a node. In Figure 1, we have used a simple numeric position for each token, which is sufficient to answer predicates such as distance and order. More expressive positions may enable more sophisticated predicates on positions such as sentence- and paragraph-distance predicates.

More formally, let  $\mathcal{N}$  be the set of nodes,  $\mathcal{P}$  be the set of positions, and  $\mathcal{T}$  be the set of tokens. The function  $Positions : \mathcal{N} \rightarrow 2^{\mathcal{P}}$  maps a node to the set of positions in the node. The function  $Token : \mathcal{P} \rightarrow \mathcal{T}$  maps each position to the token at that position. In the example in Figure 1, if the node **it** is denoted by  $n$ , then  $Positions(n) = \{410, \dots, 423, \dots\}$ ,  $Token(412) = \text{"judge"}$ ,  $Token(423) = \text{"service"}$ , and so on.

We also have the following requirement for completeness: *The full-text search language should be at least as expressive as first-order logic formulas specified over the positions of tokens in a context node.* The above requirement identifies tokens and their positions as the fundamental units in a full-text search language, and essentially describes a notion of completeness similar to that of relational completeness [8] based on first-order logic. Other notions of completeness can certainly be defined based on

higher-order logics, but as we shall soon see, defining completeness in terms of first-order logic allows for both efficient evaluation and tight integration with the relational model. We also note that each context node is considered separately, i.e., a full-text search condition does not span multiple context nodes. This is in keeping with the semantics of existing full-text languages.

### 3.2 Full-Text Calculus (FTC)

The FTC defines the following predicates to model basic full-text primitives.

- $SearchContext(node)$  is true iff  $node \in \mathcal{N}$
- $hasPos(node, pos)$  is true iff  $pos \in Positions(node)$ .
- $hasAsToken(pos, tok)$  is true iff  $tok = Token(pos)$ .

A full-text language may also wish to specify additional position-based predicates,  $Preds$ . The FTC is general enough to support arbitrary position-based predicates. Specifically, given a set  $VarPos$  of position variables, and a set  $Consts$  of constants, it can support any predicate of the form:  $pred(p_1, \dots, p_m, c_1, \dots, c_r)$ , where  $p_1, \dots, p_m \in VarPos$  and  $c_1, \dots, c_r \in Consts$ . For example, we could define  $Preds = \{distance(pos_1, pos_2, dist), ordered(pos_1, pos_2), samepara(pos_1, pos_2)\}$ . Here,  $distance(pos_1, pos_2, dist)$  returns true iff there are at most  $dist$  intervening tokens between  $pos_1$  and  $pos_2$  (irrespective of the order of the positions);  $ordered(pos_1, pos_2)$  is true iff  $pos_1$  occurs before  $pos_2$ ;  $samepara(pos_1, pos_2)$  is true iff  $pos_1$  is in the same paragraph as  $pos_2$ .

An FTC query is of the form:  $\{node | SearchContext(node) \wedge QueryExpr(node)\}$ . Intuitively, the query returns  $nodes$  that are in the search context, and that satisfy  $QueryExpr(node)$ .  $QueryExpr(node)$ , hereafter called the *query expression*, is a first-order logic expression that specifies the full-text search condition. The query expression can contain position predicates in addition to logical operators. The only free variable in the query expression is  $node$ .

As an illustration, the query below returns the context nodes that contain the keywords “district”, “judge”, and “assignment”:

$$\{node | SearchContext(node) \wedge \exists pos_1, pos_2, pos_3 \\ (hasPos(node, pos_1) \wedge hasAsToken(pos_1, 'district') \wedge \\ hasPos(node, pos_2) \wedge hasAsToken(pos_2, 'judge') \wedge \\ hasPos(node, pos_3) \wedge hasAsToken(pos_3, 'assignment'))\}$$

In subsequent examples, we only show the full-text condition since the rest of the query is the same. The following query represents the query in the introduction (find context nodes that contain the keywords “assignment”, “district”, and “judge” in that order, where the keywords “district” and “judge” occur right next to each other, and the keyword “judge” appears within 5 words of the keyword “assignment”):

$$\exists pos_1, pos_2, pos_3 (hasPos(node, pos_1) \wedge hasAsToken(pos_1, 'assignment') \wedge \\ hasPos(node, pos_2) \wedge hasAsToken(pos_2, 'district') \wedge \\ hasPos(node, pos_3) \wedge hasAsToken(pos_3, 'judge') \wedge \\ ordered(pos_1, pos_2) \wedge ordered(pos_2, pos_3) \wedge \\ distance(pos_2, pos_3, 0) \wedge distance(pos_1, pos_3, 5))$$

### 3.3 Full-Text Algebra (FTA)

The FTA is defined based on an underlying data model called a *full-text relation*, which is of the form  $R[node, att_1, \dots, att_m]$ ,  $m \geq 0$ , where the domain of *node* is  $\mathcal{N}$  (nodes) and the domain of  $att_i$  is  $\mathcal{P}$  (positions). Each tuple of a full-text relation is of the form  $(n, p_1, \dots, p_m)$ , where each  $p_i \in Positions(n)$ . Intuitively, each tuple represents a list of positions  $p_1, \dots, p_m$  that satisfy the full-text condition for node  $n$ . Since positions are modeled explicitly, they can be queried and manipulated.

An FTA expression is defined recursively as follows:

- $R_{token}(node, att_1)$ , for each  $token \in \mathcal{T}$ , is an expression.  $R_{token}$  contains a tuple for each  $(node, pos)$  pair that satisfies:  $node \in \mathcal{D} \wedge pos \in Positions(node) \wedge token = Token(pos)$ . Intuitively,  $R_{token}$  is similar to an inverted list, and has entries for nodes that contain *token* along with its positions.
- If  $Expr_1$  is an expression,  $\pi_{node, att_{i_1}, \dots, att_{i_j}}(Expr_1)$  is an expression. If  $Expr_1$  evaluates to the full-text relation  $R_1$ , the full-text relation corresponding to the new expression is:  $\pi_{node, att_{i_1}, \dots, att_{i_j}}(R_1)$ , where  $\pi$  is the traditional relational projection operator. Note that  $\pi$  *always* has to include *node* because we have to keep track of the node being queried.
- If  $Expr_1$  and  $Expr_2$  are expressions, then  $(Expr_1 \bowtie Expr_2)$  is an expression. If  $Expr_1$  and  $Expr_2$  evaluate to  $R_1$  and  $R_2$  respectively, then the full-text relation corresponding to the new expression is:  $R_1 \bowtie_{R_1.node=R_2.node} R_2$ , where  $\bowtie_{R_1.node=R_2.node}$  is the traditional relational equi-join operation on the *node* attribute. The join condition ensures that positions in the same tuple are in the same node, and hence can be processed using full-text predicates.
- If  $Expr_1$  and  $Expr_2$  are expressions, then  $\sigma_{pred(att_1, \dots, att_n, c_1, \dots, c_q)}(Expr_1)$ ,  $(Expr_1 - Expr_2)$ ,  $(Expr_1 \cup Expr_2)$  are algebra expressions that have the same semantics as in traditional relational algebra.

An FTA query is an FTA expression that produces a full-text relation with a single attribute which, by definition, has to be *node*. The set of nodes in the result full-text relation defines the result of the FTA query.

We now show how two FTC queries in Section 3.2 can be written in the FTA:

$$\pi_{node}(R_{district} \bowtie R_{judge} \bowtie R_{assignment})$$

$$\pi_{node}(\sigma_{distance(att_2, att_3, 5)}(\sigma_{ordered(att_3, att_1)}(\sigma_{ordered(att_1, att_2)}(\sigma_{distance(att_1, att_2, 0)}(R_{district} \bowtie R_{judge}) \bowtie R_{assignment}))))$$

### 3.4 Equivalence of FTC and FTA and Completeness

**Theorem 1** *Given a set of position-based predicates  $Preds$ , the FTC and FTA are equivalent in terms of expressive power.*

The proof of equivalence is similar to that of the relational algebra and calculus and is thus omitted (see [3]). We now formally define the notion of full-text completeness.

**Definition (Full-Text Completeness):** A full-text language  $\mathcal{L}$  is said to be *full-text complete* with respect to a set of position-based predicates  $Preds$  iff all queries that can be expressed in the FTC (or the FTA) using  $Preds$  can also be expressed in  $\mathcal{L}$ .

The above definition of completeness provides a formal basis for comparing the expressiveness of full-text search languages, as we shall do in Section 4. To the best of our knowledge, this is the first attempt to formalize the expressive power of such languages for flat documents, relational databases, or XML documents.

### 3.5 Scoring

Scoring is an important aspect of full-text search. However, there is no standard agreed-upon method for scoring full-text search results. In fact, developing and evaluating different scoring methods is still an active area of research [13–15, 19, 25, 30]. Thus, rather than hard-code a specific scoring method into our framework, we describe a general scoring framework based on the FTC and the FTA, and show how some of the existing scoring methods can be incorporated into this framework. Specifically, we now show how TF-IDF [24] scoring can be incorporated, and refer the reader to [3] for how probability-based scoring [14, 30] can be incorporated. We only describe how scoring can be done in the context of the FTA; the extension to the FTC is similar.

Our scoring framework is based on two extensions to the FTA: (1) per-tuple scoring information and (2) scoring transformations. Per-tuple scoring information associates a score with each tuple in a full-text relation, similar to [14]. However, unlike [14], the scoring information need not be only a real number (or probability); it can be any arbitrary type associated with a tuple. Scoring transformations extend the semantics of FTA operators to transform the scores of the input full-text relations.

We now show how TF-IDF scoring can be captured using our scoring framework. We use the following widely-accepted TF and IDF formulas for a node  $n$  and a token  $t$ :  $tf(n, t) = occurs/unique\_tokens$  and  $idf(t) = \ln(1 + db\_size/df)$ , where  $occurs$  is the number of occurrences of  $t$  in  $n$ ,  $unique\_tokens$  is the number of unique tokens in  $n$ ,  $db\_size$  is the number of nodes in the database, and  $df$  is the number of nodes containing the token  $t$ . The TF-IDF scores are aggregated using the cosine similarity:  $score(n) = \sum_{t \in q} w(t) * tf(n, t) * idf(t) / (\|n\|_2 * \|q\|_2)$ , where  $q$  denotes query search tokens,  $w(t)$ , the weight of the search token  $t$  and  $\|\cdot\|_2$ , the  $L_2$  measure.

To model TF-IDF, we associate a numeric score with each tuple. Intuitively, the score contains the TF-IDF score for all the positions in the tuple. Initially,  $R_t$  relations contain static scores: the  $idf(t)$  for the token  $t$  at that position divided by the product of the normalization factors  $unique\_tokens * \|n\|_2$ . This is the  $L_2$  normalized TF-IDF score for each position containing the token  $t$ . Thus, if we sum all the scores in  $R_t$ , we get exactly the  $L_2$ -normalized TF-IDF score of  $t$  with regards to  $n$ .

We now describe the scoring transformations for some of the FTA operators. For traditional TF-IDF, the interesting operators that change the scores are the join and the projection. First, consider the relation  $R$  that is the result of the FTA expression  $(Expr_1 \bowtie Expr_2)$ , where the scored full-text relations produced by  $Expr_1$  and  $Expr_2$  are  $R_1$  and  $R_2$ , respectively. Then, for each tuple  $t \in R$ , formed by the tuples  $t_1 \in R_1$  and  $t_2 \in R_2$ ,  $t.score = t_1.score/|R_2| + t_2.score/|R_1|$ , where  $|R|$  denotes the cardinality of  $R$ . We need to scale down  $t_1.score$  and  $t_2.score$  because their relevance

decreases due to the increased number of tuples (solutions) in the resulting relation. For projections, the new relation should have the same total score as the original one. More formally, let the relation  $R$  be the result of the expression  $\pi_{\text{CNode}, \text{att}_1, \dots, \text{att}_n}(Expr_1)$  and let  $Expr_1$  produce the relation  $R_1$ . Then, for any tuple  $t \in R$  which is the result of the aggregation of the tuples  $t_1, \dots, t_n \in R_1$ ,  $t.score = \sum_{i=1, \dots, n} t_i.score$ .

It can be shown that the above propagation of scores preserves the traditional semantics of TF-IDF for conjunctive and disjunctive queries [3]. Further, this scoring method is more powerful than traditional TF-IDF because it can be generalized to arbitrary structured queries by defining appropriate scoring transformations for each operator. For instance, we can define a scoring transformation for distance selection predicates thereby extending the scope of TF-IDF scoring.

## 4 Incompleteness of Existing Full-Text Search Languages

We show the incompleteness of existing full-text languages, including TRAs.

### 4.1 Predicate-Based Languages

We first consider traditional full-text languages that have position-based predicates in addition to Boolean operators [1, 5]. A typical syntax, which we call DIST, is:

Query := Token | Query AND Query | Query OR Query | Query AND NOT Query |  
dist(Token, Token, Integer)

Token := StringLiteral | ANY

We can recursively define the semantics of DIST in terms of the FTC. If the query is a StringLiteral 'token', it is equivalent to the FTC query expression  $\exists p(\text{hasPos}(n, p) \wedge \text{hasAsToken}(p, \text{'token'}))$ . If the query is ANY, it is equivalent to the expression  $\exists p(\text{hasPos}(n, p))$ . If the query is of the form Query1 AND NOT Query2, it is equivalent to  $Expr_1 \wedge \neg Expr_2$ , where  $Expr_1$  and  $Expr_2$  are the FTC expressions for Query1 and Query2. If the query is of the form Query1 AND Query2, it is equivalent to  $Expr_1 \wedge Expr_2$ , where  $Expr_1$  and  $Expr_2$  are FTC expressions for Query1 and Query2 respectively. OR is defined similarly. The  $\text{dist}(\text{Token}, \text{Token}, \text{Integer})$  construct is the equivalent of the *distance* predicate introduced in the calculus (Section 3.2), and specifies that the number of intervening tokens should be less than the specified integer. More formally, the semantics of  $\text{dist}(\text{token1}, \text{token2}, d)$  for some tokens  $\text{token1}$  and  $\text{token2}$  and some integer  $d$  is given by the calculus expression:  $\exists p_1(\text{hasPos}(n, p_1) \wedge \text{hasAsToken}(p_1, \text{token1}) \wedge \exists p_2(\text{hasPos}(n, p_2) \wedge \text{hasAsToken}(p_2, \text{token2}) \wedge \text{distance}(p_1, p_2, d)))$ . If  $\text{token1}$  or  $\text{token2}$  is ANY instead of a string literal, then the corresponding *hasAsToken* predicate is omitted in the semantics.

As an example, the query  $\text{dist}(\text{'test'}, \text{'usability'}, 3)$  is equivalent to the FTC query expression:  $\exists p_1 \exists p_2(\text{hasPos}(n, p_1) \wedge \text{hasAsToken}(p_1, \text{'test'}) \wedge \text{hasPos}(n, p_2) \wedge \text{hasAsToken}(p_2, \text{'usability'}) \wedge \text{distance}(p_1, p_2, 3))$ .

We now show that DIST is incomplete if  $\mathcal{T}$  is not trivially small. We can also prove similar incompleteness results for other position-based predicates.

**Theorem 2** *If  $|\mathcal{T}| \geq 3$ , there exists a query that can be expressed in FTC with  $\text{Preds} = \{\text{distance}(p_1, p_2, d)\}$  that cannot be expressed by DIST.*

*Proof Sketch:* We shall show that no query in  $\text{DIST}$  can express the following FTC query:  $\exists p_1, p_2, p_3 (hasPos(n, p_1) \wedge hasAsToken(p_1, t_1) \wedge hasPos(n, p_2) \wedge hasAsToken(p_2, t_2) \wedge hasPos(n, p_3) \wedge hasAsToken(p_3, t_3) \wedge distance(p_1, p_2, 0) \wedge distance(p_2, p_3, 0))$  (find context nodes that contains a token  $t_1$  that occurs right next to a token  $t_2$  that in turn occurs right next to a token  $t_3$ ). For simplicity, we use distances with at most 0 tokens but the example can be generalized to arbitrary distances. The proof is by contradiction. Assume that there exists a query  $Q$  in  $\text{DIST}$  that can express the calculus query. We now construct two context nodes  $CN_1$  and  $CN_2$  as follows.  $CN_1$  contains the tokens  $t_1$  followed by  $t_2$  followed by  $t_3$  followed by  $t_1$ .  $CN_2$  contains the tokens  $t_1$  followed by  $t_2$  followed by  $t_2$  followed by  $t_3$  followed by  $t_3$  followed by  $t_1$ . By the construction, we can see that  $CN_1$  satisfies the calculus query, while  $CN_2$  does not. We will now show that  $Q$  either returns both  $CN_1$  or  $CN_2$  or neither of them; since this contradicts our assumption, this will prove the theorem.

Let  $C_Q$  be the calculus expression equivalent to  $Q$ . We show that by induction on the structure of  $C_Q$ , every sub-expression of  $C_Q$  (and hence  $C_Q$ ) returns the same Boolean value for  $CN_1$  and  $CN_2$ . If the sub-expression is of the form  $\exists p(hasPos(n, p) \wedge hasAsToken(p, token))$ , it returns the same Boolean value for both  $CN_1$  and  $CN_2$  since both documents have the same set of tokens. Similarly, if the sub-expression is of the form  $\exists p(hasPos(n, p))$ , it returns true for both  $CN_1$  and  $CN_2$ . If the sub-expression is of the form  $\neg Expr$ , then it returns the same Boolean value for both  $CN_1$  and  $CN_2$  because  $Expr$  returns the same Boolean value (by induction). A similar argument can also be made for the  $\wedge$  and  $\vee$  Boolean operators. If the sub-expression is of the form  $\exists p_1(hasPos(n, p_1) \wedge hasAsToken(p_1, token1) \wedge \exists p_2(hasPos(n, p_2) \wedge hasAsToken(p_2, token2) \wedge distance(p_1, p_2, d)))$ , there are two cases. In the first case,  $token1 \notin \{t_1, t_2, t_3\} \vee token2 \notin \{t_1, t_2, t_3\}$ , and it is easy to see that the sub-expression returns false for both  $CN_1$  and  $CN_2$ . In the second case,  $token1, token2 \in \{t_1, t_2, t_3\}$ . Since  $distance(token1, token2, 0)$  is true for both  $CN_1$  and  $CN_2$ , and hence  $distance(token1, token2, d)$  is true for  $d \geq 0$ , the sub-expression returns true for both  $CN_1$  and  $CN_2$ . Since we have considered all sub-expressions, this is a contradiction and proves the theorem.  $\square$

## 4.2 Text Region Algebras

We now show that TRAs are incomplete.

**Theorem 3** *There exists a query that can be expressed in FTC with  $Preds = \{ordered(p_1, p_2), samepara(p_1, p_2)\}$  that cannot be expressed in TRA (as defined in [10]).*

*Proof Sketch:* The following FTC query cannot be expressed using TRA:  $\exists pos_1, pos_2 (hasPos(node, pos_1) \wedge hasAsToken(pos_1, t_1) \wedge hasPos(node, pos_2) \wedge hasAsToken(pos_2, t_2) \wedge ordered(pos_1, pos_2) \wedge samepara(pos_1, pos_2))$  (find context nodes that contain the tokens  $t_1$  and  $t_2$  in that order within the same paragraph). The proof is very similar to the proof by Consens and Milo [10], who have shown that TRAs cannot represent simultaneously inclusion and ordering constraints. In particular, they prove that the query: *find documents with regions  $s \in S$  that contain two other regions  $t \in T$  and  $u \in U$  such that  $t$  comes before  $u$* , cannot be represented using TRA. When

we consider  $S$  to be the regions with the same start and end positions which correspond to the occurrences of the keyword  $k_1$ , and similarly for  $T$  for keyword  $k_2$ , and set  $U$  to be regions representing paragraphs, the theorem follows.  $\square$ .

## 5 PPRED: Language and Query Evaluation

The evaluation of FTC queries corresponds to the problem of evaluating Quantified Boolean Formulas (QBF), which is LOGSPACE-complete for data complexity (complexity in the size of the database) and PSPACE-complete for expression complexity (complexity in the size of the query) [26]. Since whether LOGSPACE is a strict subset of PTIME (polynomial time), and whether PSPACE is a strict subset of EXPTIME (exponential time) are open questions, we can only devise a query evaluation algorithm that is polynomial in the size of the data and exponential in the size of the query. This evaluation complexity is clearly unacceptable for large data sets and hence motivates the need to find efficient subsets of FTC.

In this section, we present PPRED (for Positive PREDicates), a subset of FTC which includes most common full-text predicates, such as *distance*, *ordered* and *samepara*, and is more powerful than existing languages such as DIST and TRAs. Further, PPRED queries can be evaluated in a single pass over inverted lists.

The key observation behind PPRED is that many full-text predicates are true in a contiguous region of the position space. For instance, *distance* applied to two position variables is true in the region where the position values of those variables are within the distance limit, and false outside this region. For *ordered*, a region specifies the part of the position space where the positions are in the required order. Other common full-text predicates such as *samepara*, and *window* also share this property. We call such predicates positive predicates. These predicates can be efficiently evaluated by scanning context nodes in *increasing order* of positions, which can be done in a single scan over the inverted list entries because they are typically stored in increasing order of positions.

We now formally define the PPRED language, and describe efficient query evaluation algorithms that also consider score-based pruning.

### 5.1 Positive Predicates

**Definition (Positive Predicates):** An  $n$ -ary position-based predicate  $pred$  is said to be a *positive predicate* iff there exist  $n$  functions  $f_i : \mathcal{P}^n \rightarrow \mathcal{P}$  ( $1 \leq i \leq n$ ) such that:

$$\begin{aligned} \forall p_1, \dots, p_n \in \mathcal{P} \quad (\neg pred(p_1, \dots, p_n) \Rightarrow \\ \forall i \forall p'_i \in \mathcal{P} \quad p_i \leq p'_i < f_i(p_1, \dots, p_n) \Rightarrow \\ \forall p'_1, \dots, p'_{i-1}, p'_{i+1}, \dots, p'_n \in \mathcal{P} \\ p_1 \leq p'_1, \dots, p_{i-1} \leq p'_{i-1}, \\ p_{i+1} \leq p'_{i+1}, \dots, p_n \leq p'_n \Rightarrow \neg pred(p'_1, \dots, p'_n)) \\ \wedge \\ \exists j \quad f_j(p_1, \dots, p_n) > p_j \end{aligned}$$

Intuitively, the property states that for every combination of positions that do not satisfy the predicate: (a) there exists a contiguous boundary in the position space such that all combinations of positions in this boundary do not satisfy the predicate; this

contiguous area is specified in terms of the functions  $f_i(p_1, \dots, p_n)$ , which specify the lower bound of the boundary for the dimension corresponding to position  $p_i$ , and (b) there is at least one dimension in the position space where the boundary can be advanced beyond the current boundary, i.e., at least one  $f_i(p_1, \dots, p_n)$  has value greater than  $p_i$ ; this ensures that the boundary can be pushed forward in search of a combination of positions that do satisfy the predicate.

For example, for  $distance(p_1, \dots, p_n, d)$ , we can define the  $f_i$  functions as follows  $f_i(p_1, \dots, p_n) = \max(\max(p_1, \dots, p_n) - d + 1, p_i)$ . Similarly, for  $ordered$ ,  $f_i(p_1, \dots, p_n) = \max(p_1, \dots, p_i)$ . For  $samepara$ ,  $f_i(p_1, \dots, p_n) = \min\{p \in \mathcal{P} \mid para(p) = \max(para(p_1), \dots, \max(para(p_n)))\}$  where  $para$  is a function that returns the paragraph containing a position.

**Language Description.** We now define the PPRED language, which is a strict superset of DIST. Thus, simple queries retain the same conventional syntax, while new constructs are only required for more complex queries.

Query := Token | Query AND Query | Query OR Query | Query AND NOT Query\* | SOME Var Query | Preds

Token := StringLiteral | ANY | Var HAS StringLiteral | Var HAS ANY

Preds := distance(Var, Var, Integer) | ordered(Var, Var) | ...

The main additions to DIST are the HAS construct in Token and the SOME construct in Query. The HAS construct allows us to explicitly bind position variables (Var) to positions where tokens occur. The semantics for ' $var_1$  HAS  $tok$ ' in terms of the FTC, where  $tok$  is a StringLiteral is:  $hasAsToken(var_1, tok)$ . The semantics for ' $var_1$  HAS ANY' is:  $hasPos(n, var_1)$ . While the HAS construct allows us to explicitly bind position variables to token positions, the SOME construct allows us to quantify over these positions. The semantics of 'SOME  $var_1$  Query' is  $\exists var_1 (hasPos(n, var_1) \wedge Expr)$ , where  $Expr$  is the FTC expression semantics for Query. Query\* refers to a Query with no free variables.

For example, the following PPRED query expresses the second sample query from Section 3.2 SOME p1 HAS 'assignment' SOME p2 HAS 'district' SOME p3 HAS 'judge' ordered(p1, p2) AND ordered(pos2, pos3) AND distance(p2, p3) AND distance(p1, p3, 5).

Although PPRED is not complete (e.g., it does not support universal quantification and arbitrary negation), it is still quite powerful. For instance, it can specify all of the queries used in the incompleteness proofs in Section 4 (since ordered, distance and samepara are all positive predicates). In fact, PPRED is a strict superset of DIST (since it contains all of the constructs of DIST) and of TRAs (see [3] for the proof).

## 5.2 Query Evaluation

We describe the PPRED query evaluation model and algorithm.

**Query Evaluation Model.** Each  $R_{token}$  relation is represented as an inverted list associated to  $token$ . Each inverted list contains one or more *entries*. Each entry in  $R_{token}$  is of the form:  $(node, PosList, score)$ , where  $node$  is the identifier of a node that contains  $token$ ,  $PosList$  is the list of positions of  $token$  in  $node$ , and  $score$  is the score of  $node$ . We assume that the inverted lists are sorted on node identifiers. Note that they could be sorted on scores. Figure 2(a) shows example inverted lists for the words

“district”, “judge”, and “assignment”. Inverted lists are typically accessed sequentially using a *cursor*. Advancing the cursor can be done in constant time.

**Query Evaluation Overview.** We now illustrate how positive predicates enable efficient query evaluation. Consider the simple query  $\pi_{node}(\sigma_{distance(att_1,att_2,1)}(R_{district} \bowtie R_{judge}))$  (return nodes that contain the words “district” and “judge” within at most 1 word of each other). The naive evaluation approach over the inverted lists shown in Figure 2(a) would compute the Cartesian product of the positions for each node and then apply the distance predicate. For the node 1, this corresponds to computing 9 pairs of positions (3 in each inverted list), and then only selecting the final pair (139,140) that satisfies the distance predicate. However, using the property that distance is a positive predicate, we can determine the answer by only scanning 6 pairs of positions (3 + 3 instead of 3 \* 3), as described below.

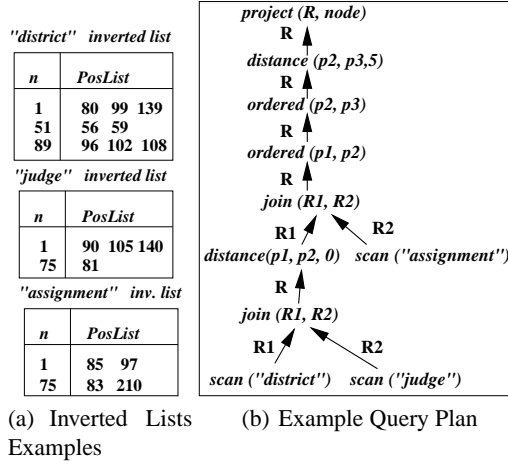
The query evaluation starts with the smallest pair of positions (80, 90) for node 1 and check whether it satisfies the distance predicate. Since it does not, we move *the smallest position* to get the pair (99, 90). Since this pair still does not satisfy the predicate, we again move the smallest position until we find a solution: (99, 105), (139, 105), (139, 140). Note that each position is scanned exactly once, so the complexity is linear in the size of the inverted lists. The reason the smallest position could be moved is because the distance predicate is true in a contiguous region, and if the predicate is false for the smallest position in the region, one can infer that it is also false for other positions without having to explicitly enumerate them.

Let us now consider a slightly more complex example using the *ordered* predicate:  $\pi_{node}(\sigma_{ordered(att_1,att_2,att_3)}(R_{district} \bowtie R_{judge} \bowtie R_{assignment}))$  (return nodes that contain the words “district”, “judge” and “assignment” in that order). For node 1, the first combination of positions (80, 90, 85) does not satisfy *ordered*. However, unlike the window predicate, we *cannot* move the cursor corresponding to the smallest position to get the combination (99, 90, 85); doing so will cause the solution (80, 90, 97) to be missed! (note that we cannot move a cursor back if we want a single scan over the positions). Rather, for *ordered*, we need to move the smallest position that *violates the order*. In our example, we should move the third cursor to get the combination (80, 90, 97).

In the above examples depending on the full-text predicate, different strategies may have to be employed to produce the correct results efficiently. This becomes harder with complex full-text queries (i.e., where different predicates are combined). Furthermore, the problem becomes even more complex when the query contains multiple predicates over possibly overlapping sets of positions. Which cursors should be moved in this case? Does the order in which cursors used by different predicates are moved matter? Is there a general strategy for evaluating arbitrary combinations of FTA queries with positive predicates? We answer these questions in the next section.

One aspect to note is that our query evaluation algorithms efficiently evaluate full-text predicates a node at a time before moving on to the next node. An important consequence of this is that our algorithms can be combined with any existing top-k evaluation technique (e.g. [11]), which prunes nodes from consideration based on their score (our algorithms will just not evaluate queries over the pruned nodes).

**Query Evaluation Algorithms.** A query is first converted to FTA operators and is rewritten to push down projections wherever possible so that spurious positions are not



**Fig. 2.** Sample Inverted Lists and Query Plan

propagated. Figure 2(b) shows a sample FTA operator plan for the query in Section 1. Since we do not want to materialize the entire output full-text relation corresponding to an operator, each operator exposes a new API for traversing its output. This API ensures that successive calls can be evaluated in a single scan over the inverted list positions. We denote the output full-text relation for an operator  $o$ ,  $R$  which has  $n$  position columns. The API, defined below, maintains the following state:  $node$ , which tracks the current node, and  $p_1, \dots, p_n$ , which track the current positions in  $node$ .

- **advanceNode()**: On the first call, it sets  $node$  to be the smallest value in  $\pi_{node}(R)$  (if one exists; else  $node$  is set to NULL). It also sets position values,  $p_1, \dots, p_n$  such that:  $(node, p_1, \dots, p_n) \in R \wedge \forall p'_1, \dots, p'_n (node, p'_1, \dots, p'_n) \in R \Rightarrow p'_1 \geq p_1 \wedge \dots \wedge p'_n \geq p_n$  (i.e., it sets positions  $p_1, \dots, p_n$  to be the smallest positions that appear in  $R$  for that  $node$ ; we will always be able to find such positions due to the property of positive predicates). On subsequent calls,  $node$  is updated to the next smallest value in  $\pi_{node}(R)$  (if one exists), and  $p_1, \dots, p_n$  are updated as before.
- **getNode()**: Returns the current value of  $node$ .
- **advancePosition(i, pos)**: It sets the values of  $p_1, \dots, p_n$  such that they satisfy:  $(node, p_1, \dots, p_n) \in R \wedge p_i > pos \wedge \forall p'_1, \dots, p'_n (node, p'_1, \dots, p'_n) \in R \wedge p'_i \geq pos \Rightarrow (p'_1 \geq p_1 \wedge \dots \wedge p'_n \geq p_n)$  (i.e., the smallest values of positions that appear in  $R$  and that satisfy the condition  $p_i > pos$ ), and returns true. If no such positions exist, then it sets  $p_i$ s to be NULL and returns false.
- **getPosition(i)**: Returns the current value of  $p_i$ .

Given the operator evaluation tree in Figure 2(b), the general evaluation scheme proceeds as follows. To find a solution **advanceNode** is called on the top project operator which simply forward this call to the distance selection operator below it. The latter tries to find a solution by continuously calling **advancePosition** on the ordered predicate below it until it finds a satisfying tuple of positions. The ordered predicates behaves in a similar manner: it advances through the result of the underlying operator

---

**Algorithm 1** Join Evaluation Algorithm

---

**Require:**  $inp1, inp2$  are the two API inputs to the join, and have  $c_1$  and  $c_2$  position columns, respectively

```
1: Node advanceNode() {
2:   node1 = inp1.advanceNode(); node2 = inp2.advanceNode();
3:   while node1 != NULL && node2 != NULL && node1 != node2 do
4:     if node1 < node2 then node1 = inp1.advanceNode();
5:     else node2 = inp2.advanceNode(); endif
6:   end while
7:   if node1 == NULL || node2 == NULL then return NULL;
8:   else [node1 == node2]
9:     set  $p_i$  ( $i < c_1$ ) to inp1.getPosition(i);
10:    set  $p_i$  ( $i \geq c_1$ ) to inp2.getPosition(i - c_1);
11:    node = node1;
12:    return node1; endif }
13:
14: boolean advancePosition(i, pos) {
15:   if ( $i < c_1$ ) then
16:     result = inp1.advancePosition(i, pos);
17:     if (result) then  $p_i = inp1.getPostion(i)$ ; endif
18:     return result;
19:   else //Similar for inp2 end if }
```

---

until it finds a tuple that satisfies it. The evaluation proceeds down the tree until the leaves (the scan operators) are reached. The latter simply advances through the entries in the inverted lists. The entire evaluation is pipelined and no intermediate relation needs to be materialized.

We now show how different PPRED operators can implement the above API. The API implementation for the inverted list scan and project operators are straightforward since they directly operate on the inverted list and input operator API, respectively. Thus, we focus on joins and selections. The implementation for set difference and union is similar to join, and is not discussed here.

Algorithm 1 shows how the API is implemented for the join operator. We only show the implementation of the `advanceNode` and `advancePos` methods since the other methods are trivial. Intuitively, `advanceNode` performs an equi-join on the *node*. It then sets the positions  $p_i$  to the corresponding positions in the input.

`advancePosition(i, pos)` moves the position cursor on the corresponding input.

Algorithm 2 shows how the API is implemented for selections implementing predicate  $pred$  with functions  $f_i$  defined in Section 5.1. Each call of `advanceNode`, advances *node* until one that satisfies the predicate is found, or there are no *cnodes* left. The satisfying node is found using the helper method `advancePosUntilSat`, which returns true iff it is able to advance the positions of the current *node* so that they satisfy the predicate  $pred$ . `advancePosition` first advances the position on its input, and then invokes `advancePosUntilSat` until a set of positions that satisfy  $pred$  are found.

`advancePosUntilSat` first checks whether the current positions satisfy  $pred$ . If not, it uses the  $f_i$  functions to determine a position  $i$  to advance, and loops back until a set of positions satisfying  $pred$  are found, or until no more positions are available. This

---

**Algorithm 2** Predicate Evaluation Algorithm

---

**Require:**  $inp$  is API inputs to the predicate with  $c$  position columns

```
1: Cnode advanceCnode() {
2:   cnode = inp.advanceCnode();
3:   while cnode != NULL && !advancePosUntilSat() do
4:     cnode = inp.advanceCnode();
5:   end while
6:   return cnode; }
7:
8: boolean advancePosition(i,pos) {
9:   success = inp.advancePosition(i,pos);
10:  if !success then return false; endif
11:   $p_i = inp.getPos(i)$ ; return advancePosUntilSat(); }
12:
13: boolean advancePosUntilSat () {
14:  while !pred( $p_1, \dots, p_c$ ) do
15:    find some  $i$  such that  $f_i(p_1, \dots, p_c) > p_i$ 
16:    success = inp.advancePos( $i, f_i(p_1, \dots, p_c)$ );
17:    if success then return false; end if
18:     $p_i = inp.getPosition(i)$ ;
19:  end while
20:  return true; }
```

---

is the core operation in selections: scanning the input positions until a match is found. Positive predicates enable us to do this in a single pass over the input.

**Correctness and Complexity.** We now present a sketch of the proof of correctness of the above algorithm (see [3] for the full proof). First, it is not hard to see that every answer returned by the algorithm results from evaluating the corresponding  $\text{PPRED}$  query. The `advancePosUntilSat` function of the predicate operator does not return until satisfying positions are found or the end of an inverted list is reached. The join operator only returns a tuple if both input tuples correspond to the same node. The union operator only returns a tuple if at least one of its inputs produces that tuple. The set-difference operator only returns tuples that are produced by the first input only.

We prove that the algorithm does not miss any query results inductively on each operator. The scan always moves the cursor to the first position of the node for `advanceNode` or to the first position that is after  $pos$  for `advancePosition(i, pos)`. Therefore, it is trivially correct. Selection only moves the cursor  $p_i$  for which  $p_i < f_i(p_1, \dots, p_n)$ , and the definition of positive predicates guarantees that we do not miss results. Similarly, the join operator moves the cursors only while one of the predicates is violated by a higher-level operator. The correctness of project, union, and set-difference can be proved similarly.

To calculate the query evaluation complexity, we define the following parameters: **entries\_per\_token** is the maximum number of scanned entries in a token inverted list (this is either the entire inverted list in the case of regular query processing, or some subset in the case of top-k processing); **pos\_per\_entry** is the maximum number of positions in an entry in a token inverted list; **toks.Q** is the number of search keywords

in a query  $Q$ ;  $\mathbf{preds\_Q}$  is the number of predicates in a query  $Q$ ;  $\mathbf{ops\_Q}$  is the number of operations in a query  $Q$ . The complexity of a PPRED query is:

$O(\mathbf{entries\_per\_token} \times \mathbf{pos\_per\_entry} \times \mathbf{toks\_Q} \times (\mathbf{preds\_Q} + \mathbf{ops\_Q} + 1))$  Intuitively, every node and every position within a node is processed at most once. For every combination of positions, we process each operator at most once.

## 6 Experiments

The main goal of our experiments is to study the performance of the PPRED query evaluation algorithm. We also compare its performance with two other techniques:

1. **BOOL**, which is a restriction of **DIST** with an empty set of position-based predicates; i.e. it only contains the Boolean operators and keyword matching. Such evaluation has been studied and optimized extensively in the IR community [27], and serves as a baseline because it does not incur the cost of predicate evaluation.
2. **REL**, which is a direct implementation of FTA using regular relational operators, such as proposed in [6, 12, 16, 18, 21, 29]. This helps illustrate the performance benefits of the PPRED query evaluation algorithm.

### 6.1 Experimental Setup

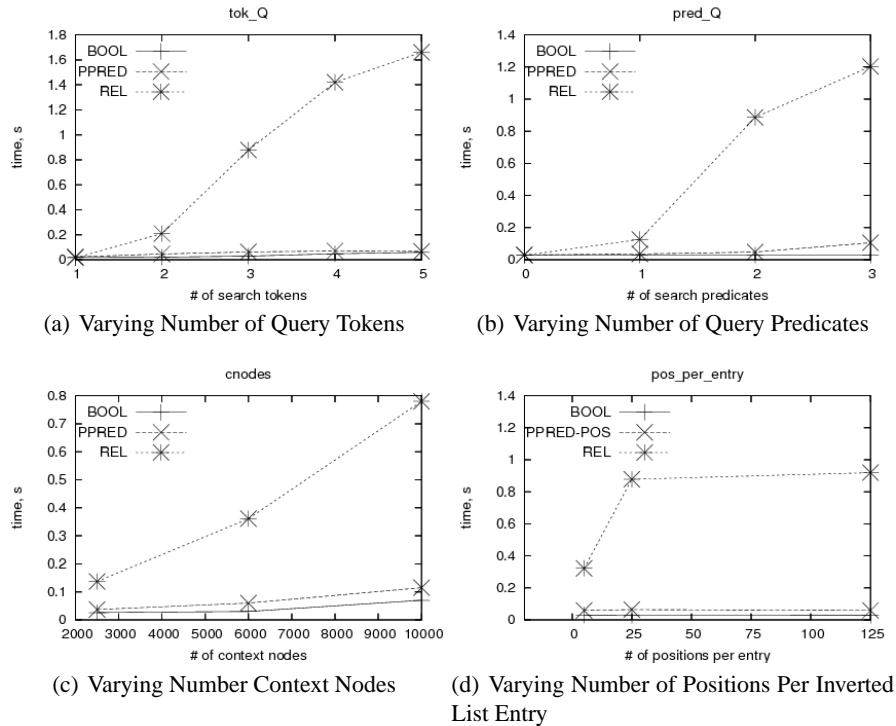
We used the 500MB INEX 2003 dataset,<sup>7</sup> which contains over 12000 IEEE papers represented as XML documents. Since we are interested in full-text search, we ignored the XML structure and indexed the collection as flat documents, i.e., each document corresponds to a context node. We also ran experiments using synthetic data; since the results were similar, we only report the results for the INEX collection.

We varied the data and query parameters described in Section 5.2 by appropriately varying the number of documents and the query keywords. To study the effect of each parameter on query performance, we varied only one parameter and fixed others at their default values. The range of values for each parameter are:  $\mathbf{entries\_per\_token}$  took on the values 1000, 10000, 100000 (default 10000),  $\mathbf{pos\_per\_entry}$  took on the values 25, 75, 125, 200 (default 125),  $\mathbf{toks\_Q}$  took on the values 1, 2, 3, 4, 5 (default 3) and  $\mathbf{preds\_Q}$  took on the values 0, 1, 2, 3, 4 (default 2). We used *distance* as the representative positive predicate. We only show the results for varying  $\mathbf{entries\_per\_token}$ ,  $\mathbf{pos\_per\_entry}$ ,  $\mathbf{toks\_Q}$ , and  $\mathbf{preds\_Q}$  since the other results are similar.

All the algorithms were implemented in C++ and used TF-IDF scoring. We ran our experiments on an AMD64 3000+ computer with 1GB RAM and one 200GB SATA drive, running under Linux 2.6.9.

### 6.2 Experimental Results

Figures 3(a) and 3(b) show the performance of the algorithms when varying  $\mathbf{toks\_Q}$  and  $\mathbf{preds\_Q}$ , respectively. The performance of **BOOL** and **PPRED** scales linearly, while the performance of **REL** degrades exponentially. This is explained by the fact that **REL** uses traditional relational joins, which compute the entire Cartesian product of positions within a node, while **PPRED** and **BOOL** use single-pass evaluation algorithms.



**Fig. 3.** Experiments on the INEX collection

Interestingly, the performance of PPRD is only slightly worse than BOOL, which suggests that PPRD only introduces a slight additional overhead over the baseline.

Figures 3(c) and 3(d) show the performance of the algorithms when varying **entries\_per\_token** and **pos\_per\_entry**, respectively. PPRD and BOOL scale gracefully, while REL does not scale well. Again, PPRD performs only slightly worse than BOOL, suggesting that there is little overhead to evaluating positive predicates.

## 7 Conclusion

We introduced the FTC and FTA as a new formalism for modeling structured full-text search and showed that existing languages are incomplete in this formalism. We also identified a powerful subset of the FTC and FTA that can be evaluated efficiently in a single pass over inverted lists. As part of future work, we plan to capture more aspects such as stemming and thesauri; we believe that these can be modeled as additional predicates in the FTC. Since the FTA is based on the relational algebra, we also plan to explore the joint optimization of full-text and relational queries.

<sup>7</sup> <http://www.is.informatik.uni-duisburg.de/projects/inex03/>

## References

1. R. Baeza-Yates, B. Ribiero-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
2. G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, S. Sudarshan. *Keyword Searching and Browsing in Databases using BANKS*. ICDE 2002.
3. C. Botev, S. Amer-Yahia, J. Shanmugasundaram. "On the Completeness of Full-Text Search Languages". Technical Report, Cornell University, 2005. <http://www.cs.cornell.edu/database/TeXQuery/Expressiveness.pdf>
4. J. M. Bremer, M. Gertz. *XQuery/IR: Integrating XML Document and Data Retrieval*. WebDB 2002.
5. E. W. Brown. *Fast Evaluation of Structured Queries for Information Retrieval*. SIGIR 1995.
6. T. T. Chinenyanga, N. Kushmerick. *Expressive and Efficient Ranked Querying of XML Data*. WebDB 2001.
7. C. Clarke, G. Cormack, F. Burkowski. *An Algebra for Structured Text Search and a Framework for its Implementation*. *Comput. J.* 38(1): 43-56 (1995)
8. E.F. Codd. *Relational Completeness of Database Sublanguages*. R. Rustin (ed.), *Database Systems*, 1972.
9. S. Cohen et al. *XSearch: A Semantic Search Engine for XML*. VLDB 2003.
10. M. P. Consens, T. Milo. *Algebras for Querying Text Regions: Expressive Power and Optimization*. *J. Comput. Syst. Sci.* 57(3): 272-288 (1998)
11. R. Fagin, A. Lotem, M. Naor. *Optimal Aggregation Algorithms for Middleware*. *J. of Comp. and Syst. Sciences* 66 (2003)
12. D. Florescu, D. Kossmann, I. Manolescu. *Integrating Keyword Search into XML Query Processing*. WWW 2000.
13. N. Fuhr, K. Grossjohann. *XIRQL: An Extension of XQL for Information Retrieval*. SIGIR 2000.
14. N. Fuhr, T. Rölleke. *A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems*. *ACM TOIS* 15(1), 1997.
15. Y. Hayashi, J. Tomita, G. Kikui. *Searching Text-rich XML Documents with Relevance Ranking*. SIGIR Workshop on XML and Information Retrieval, 2000.
16. V. Hristidis, L. Gravano, Y. Papakonstantinou. *Efficient IR-Style Keyword Search over Relational Databases*. VLDB 2003.
17. J. Jaakkola, P. Kilpelinen. *Nested Text-Region Algebra Report C-1999-2*, Dept. of Computer Science, University of Helsinki, January 1999
18. J. Melton, A. Eisenberg. *SQL Multimedia and Application Packages (SQL/MM)*. SIGMOD Record 30(4), 2001.
19. S.-H. Myaeng, D.-H. Jang, M.-S. Kim, Z.-C. Zhoo. *A Flexible Model for Retrieval of SGML Documents*. SIGIR 1998.
20. G. Navarro, R. Baeza-Yates. *Proximal Nodes: a Model to Query Document Databases by Content and Structure*. *ACM Trans. Inf. Syst.*, 15(4), 1997.
21. A. Salminen. *A Relational Model for Unstructured Documents*. SIGIR 1987.
22. A. Salminen, F. Tompa. *PAT Expressions: an Algebra for Text Search*. *Acta Linguistica Hungar.* 41 (1-4), 1992
23. G. Salton, M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
24. G. Salton. *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer*. Addison Wesley, 1989.
25. A. Theobald, G. Weikum. *The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking*. EDBT 2002.
26. M. Vardi. *The Complexity of Relational Query Languages*. STOC 1982.
27. I. Witten, A. Moffat, and T. Bell. "Managing Gigabytes: Compressing and Indexing Documents and Images". Morgan Kaufmann Publishing, 1999.
28. M. Young-Lai, F. Tompa. *One-pass Evaluation of Region Algebra Expressions*. *Inf. Syst.*, 28(3), 2003.
29. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. *On Supporting Containment Queries in Relational Database Management Systems*. SIGMOD 2001.
30. E. Zimanyi. *Query Evaluations in Probabilistic Relational Databases*. *Theoretical Computer Science*, 1997.