Expressiveness and Performance of Full-Text Search Languages

Chavdar Botev
Cornell University
cbotev@cs.cornell.edu

Sihem Amer-Yahia
AT&T Labs–Research
sihem@research.att.com

Jayavel Shanmugasundaram
Cornell University
jai@cs.cornell.edu

Abstract

We study the expressiveness and performance of full-text search languages. Our main motivation is to provide a formal basis for comparing such languages and to develop a model for full-text search that can be tightly integrated with structured search. We develop a formal model for full-text search 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 and can be used as a basis for a study of their expressiveness. We show that existing full-text languages are incomplete and develop `COMP`, a complete full-text search language. We also identify practical subsets of `COMP` that are more powerful than existing languages, develop efficient query evaluation algorithms for these subsets, and study experimentally their performance.

1 Introduction

There has been a lot of interest in full-text search over flat files [3, 31], relational databases [4, 22], and more recently, XML documents [2, 6, 15, 18, 21, 33]. The expressiveness of such languages ranges from simple Boolean search to phrase matching to general proximity search with distance predicates. Unfortunately, there is no existing work that systematically compares these different full-text search languages, either from an expressiveness or a performance point of view; we believe that this void is mainly due to the lack of a powerful formal model for full-text search. This also makes it difficult to seamlessly integrate with structured search, which is usually based on the formal underpinnings of the relational model. In this paper, we attempt to lay down the formal foundations for full-text search languages, and to compare the expressiveness and performance of different languages.

Our first contribution (Section 2) is the development of a formal model for full-text search languages. At an abstract level, such languages require the ability to manipulate individual *tokens* (or words) and their *positions* in the input text and return the nodes (e.g., documents, tuples, or XML elements) that satisfy the full-text search condition. We thus define a formal model based on the positions of tokens in the input text. Based on this model, we define a notion of completeness and develop an associated full-text calculus (FTC) based on first-order logic and a full-text algebra (FTA) based on the relational algebra. We also show how the FTC and FTA can be extended to capture the notion of scores, such as scores computed using TF-IDF [3, 28].

Our second contribution (Section 4) is to show that existing languages are incomplete. We thus propose `COMP`, a new complete full-text search language based on the FTC. `COMP` naturally generalizes existing Boolean full-text search languages and is able to express primitives such as order specifications between words, paragraph scoping and word distance.

Our third distribution is the design of a scoring framework that can be used within our full-text search model. The scoring framework does not mandate a fixed scoring method but allows the use of large class of existing scoring methods. In Section 3, we describe the framework and show how it can be used with two of the most popular scoring methods: TF-IDF [28] and probabilistic scoring [19, 38]

Our fourth contribution (Sections 5 and 6) is the identification of practical subsets of `COMP` that are significantly more powerful than existing full-text search languages, but which can still be evaluated in a linear scan over inverted list data structures, which are commonly used in full-text search. We also experimentally evaluate our proposed algorithms using real and synthetic data sets.

2 Full-Text Model, Full-Text Calculus and Algebra

At its core, a full-text search specification has two components: (1) the *search context*, which specifies the set of *context nodes* (e.g., document corpus in an IR system, tuples in a relational database or elements in XML documents) over which the full-text search is to be performed and, (2) the *full-text condition*, which specifies the condition that should be evaluated on each context node. Only the context nodes that satisfy the full-text condition qualify as answers. As an illustration, consider the following example from the XQuery Full-Text Use Cases Document [35].

Example 1 (Use Case 10.4): *Given an XML document that contains book and article elements, find the book elements containing the "efficient" and the phrase "task completion" in that order with at most 10 intervening tokens.*

In this example, the context nodes are book elements (and not articles), and the full-text search condition is: *contains the keyword "efficient" and the phrase "task completion" in that order with at most 10 intervening tokens*. Note that the search condition includes multiple primitives: Boolean AND, phrase matching, order specifications, and distance predicates.

Thus, in order to specify a full-text search query, we need (1) a language to specify the context nodes and, (2) a language to specify the search condition (the full-text search language). For (1), we can use SQL in the case of relational data, XQuery in the case of XML documents, or simply the entire document collection in the case of traditional IR systems. Since SQL and XQuery have well-defined formal semantics, we focus on the full-text search language. We first present our model, before discussing the FTC, the FTA, and scoring issues.

2.1 Full-Text Model

We first introduce two aspects of our model: a model for context nodes, and our requirement for completeness.

2.1.1 Modeling Context Nodes

Existing models for context nodes are insufficient for expressive full-text search. For instance, the XQuery data model for the book element in Figure 1 treats all the text under an element as a single text node (ignore the numbers in parentheses for now). This model is enough to identify sub-strings in the text and evaluate queries such as *find author nodes containing 'Elina'*. However, it is insufficient to answer queries such as *find books that contain the tokens 'usability' and 'testing' with 3 intervening tokens.* Most IR models solve part of this problem by tokenizing the input text, and representing each token separately. Thus, in our example, the text in the context node would be modeled as the "bag of words" $\{book, id, 1000, author, Elina, Rose, \ldots\}$. However, this model still cannot capture the distance between

```
<book(1) id(2)="usability(3)">
  <author(4)>Elina(5) Rose(6)</author(7)>
  <content(8)> Usability(9) Definition(10)
    <p(11)> Usability(12) of(13) a(14)
           software(15) measures(16)
   how(17) well(18) the(19)
   software(20) supports(21)
   achieving(22) an(23)
           efficient(24) software(25).
    </p(26)>
    </p(27)> A(28) software(29) is ...
             More(37) on(38) usability(39)
             of(40) a(41) software(42) ...
  </content(284)>
</book(285)>
```

Figure 1: Positions Example

tokens (some IR languages, however, do support limited forms of distance predicates; see Section 4.2 for more details).

In this paper, we explicitly model the *position* of a token in a context node. We argue that this model, although simple, is powerful enough to capture the semantics of existing full-text search languages. Further, it serves as the formal basis for defining position-based predicates such as proximity distance and order predicates. In Figure 1, we have used a simple numeric position (within parenthesis) for each token. Our proposed model, however, does not dictate any specific implementation of positions and is extensible with respect to the set of predicates. More expressive positions that capture the notions of lines, sentences and paragraphs can be used, and this will enable more sophisticated predicates on positions.

We now define our formal model. $\mathcal{N}$ is the set of context nodes, $\mathcal{P}$ is the set of positions, and $\mathcal{T}$ is the set of tokens. The function $Positions : \mathcal{N} \rightarrow 2^{\mathcal{P}}$ maps a context node to the set of positions in the context node. The function $Token : \mathcal{P} \rightarrow \mathcal{T}$ maps each position to the token stored at that position. In the example in Figure 1, if the context node is denoted by n, then $Positions(n) = \{1, 2, ..., 28\}$, $Token(1) = book$, $Token(2) = id$, and so on.

2.1.2 Requirement for Completeness

The full-text search language should be at least as expressive as first-order logic formulae 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 [13] based on first-order logic. We note that 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. One other issue to note in the above requirement is that each context node is considered separately, i.e., a full-text search condition does not span multiple context nodes or documents. This is in keeping with the semantics of existing full-text languages, and while other extensions are certainly possible, we do not consider them here.

2.2 Full-Text Calculus

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

- $SearchContext(node)$ is true iff $node \in \mathcal{N}$ (recall that $\mathcal{N}$ is the set of context nodes).
- $hasPos(node, pos)$ is true iff $pos \in Positions(node)$. This predicate explicitly captures the notion of positions in an XML node.
- $hasToken(pos, tok)$ is true iff $tok = Token(pos)$. This predicate captures the relationship between tokens and the positions in which they occur.

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

2.2.1 Full-Text Calculus Queries

A full-text calculus query is of the form: $\{node | SearchContext(node) \wedge QueryExpr(node)\}$. Intuitively, the query returns $node$s 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. $node$ is the only free variable in the query expression. The structure of the query expression is recursively defined as follows.

- $hasPos(node, pos_i)$ is a query expression where $node$ is the free variable and $pos_i \in VarPos$.
- $hasToken(pos_i, tok)$ is a query expression, where $pos_i \in VarPos$ and $tok \in Consts$.
- $pred(pos_1, ..., pos_m, c_1, ..., c_r)$ is a query expression, where $pred \in Preds$, $pos_i \in VarPos$ and $c_j \in Consts$.
- If qe_1 and qe_2 are query expressions, $\neg qe_1$, $qe_1 \wedge qe_2$, and $qe_1 \vee qe_2$ are query expressions.
- If qe is a query expression, then $\exists pos_i(hasPos(node, pos_i) \wedge qe)$, and $\forall pos_i(hasPos(node, pos_i) \Rightarrow qe)$ are query expressions, where $pos_i \in VarPos$.

A full-text calculus query has the conventional semantics of first-order logic. The form of the quantification in the last bullet guarantees that the query expression in the full-text calculus can be evaluated using only the positions and tokens in the context node, without having to look at other positions. This notion is similar to the notion of safety for the relational calculus.

We now provide some examples of full-text calculus queries. The following query returns the context nodes that contain the tokens 'test' and 'usability'.

$\{node | SearchContext(node) \wedge \exists pos_1(hasPos(node, pos_1) \wedge hasToken(pos_1, 'test') \wedge$

$\exists pos_2(hasPos(node, pos_2) \wedge hasToken(pos_2,' usability')))\}$

In the subsequent examples, we only show the query expression since the rest of the query is the same. The following query returns the context nodes that contain the token 'test' and the token 'usability' with at most 5 intervening tokens.

$\exists pos_1(hasPos(node, pos_1) \wedge hasToken(pos_1,' test') \wedge \exists pos_2(hasPos(node, pos_2) \wedge$
$hasToken(pos_2,' usability') \wedge distance(pos_1, pos_2, 5)))$

The following query returns the context nodes that contain two occurrences of the token 'test' and do not contain the token 'usability'.

$\exists pos_1(hasPos(node, pos_1) \wedge hasToken(pos_1,' test') \wedge \exists pos_2(hasPos(node, pos_2) \wedge hasToken(pos_2,' test')$
$\wedge diffpos(pos_1, pos_2) \wedge \forall pos_3(hasPos(node, pos_3) \Rightarrow \neg hasToken(pos_3,' usability'))))$

2.3 Full-Text Algebra

We now define our full-text relations and algebra operators. The underlying data model for our algebra is a *full-text relation* of the form $\mathtt{R}[\mathtt{CNode}, \mathtt{att_1}, ..., \mathtt{att_m}]$ where the domain of `CNode` is $\mathcal{N}$ (context nodes), and the domain of $\mathtt{att_i}$ is $\mathcal{P}$ (positions). `R` satisfies the following properties.

- `R` has always at least the attribute `CNode`. This captures the context node for full-text search. The remaining attributes in `R` capture the essence of full-text search, which is to manipulate positions.
- Each tuple `t` in a full-text relation should satisfy the condition that for all the positions pos in `t`, $pos \in Positions(t.CNode)$. The intuition is that a full-text search query can only manipulate positions within a single context node.

A full-text algebra expression is based on the following full-text relations that characterize the search context nodes, their positions, and the tokens at these positions.

- $\mathtt{SearchContext}(\mathtt{CNode})$: This relation contains a tuple $(node)$ for each $node \in \mathcal{N}$.
- $\mathtt{HasPos}(\mathtt{CNode}, \mathtt{att_1})$: This relation contains a tuple for each ($node$,pos) pair that satisfies: $node \in \mathcal{N} \wedge pos \in Positions(node)$. Intuitively, this relation relates context nodes to their positions.
- $\mathtt{R_{token}}(\mathtt{CNode}, \mathtt{att_1})$: This is a family of relations, one for each $token \in \mathcal{T}$. R_{token} contains a tuple for each ($node$,pos) pair that satisfies: $node \in \mathcal{N} \wedge pos \in Positions(node) \wedge token = Token(pos)$. Intuitively, R_{token} contains positions that contain $token$, and is similar to an inverted list in IR.

We note that while each $\mathtt{R_{token}}$ relation is finite, there number of such relations will be infinite if $\mathcal{T}$ is infinite. However, this does not lead to a problem in defining the algebra because each algebra expression is finite, and can only refer to a finite set of such relations. Also, physically instantiating the potentially infinite set of $\mathtt{R_{token}}$ relations is not a problem because only a finite sub-set of these relations will be non-empty (because the search context is finite), so only this finite set of relations will have to be explicitly stored. This is in fact what happens in current implementations of inverted lists.

In addition, as in the calculus, we have a set of position-based predicates $Preds$.

2.3.1 Full-Text Algebra Operators and Queries

The full-text algebra operators are similar to the relational operators, but with two important differences. First, full-text algebra operators only operate on full-text relations (as defined above), and not on arbitrary relations, due to the nature of full-text search. Second, full-text algebra operators implicitly enforce that each operation only manipulates positions within a single node, and not across nodes. These two properties ensure that the full-text algebra is equivalent to the full-text calculus in characterizing full-text search. A full-text algebra expression is defined recursively as follows.

- `SearchContext` is an algebra expression that returns the tuples in the full-text relation `SearchContext`.
- `HasPos` is an algebra expression that returns the tuples in the full-text relation `HasPos`.
- $\mathtt{R}_{\mathtt{token}}$ is an algebra expression that returns the tuples in the $\mathtt{R}_{\mathtt{token}}$ relation, where $token \in \mathcal{T}$.
- If $Expr_1$ is an algebra expression, $\pi_{\mathtt{CNode},\mathtt{att}_1,\ldots,\mathtt{att}_\mathtt{i}}(Expr_1)$ is an algebra expression. If $Expr_1$ evaluates to the full-text relation $\mathtt{R}_1$, the full-text relation corresponding to the new expression is: $\pi_{\mathtt{CNode},\mathtt{att}_1,\ldots,\mathtt{att}_\mathtt{i}}(\mathtt{R}_1)$, where π is the traditional relational projection operator. The attribute names of the result full-text relation are renamed to have consecutive $\mathtt{att}_\mathtt{i}$'s. Note that π *always* has to include `CNode` in the full-text algebra - this enforces the property that full-text search is always scoped within a single context node.
- If $Expr_1$ and $Expr_2$ are algebra expressions, then $(Expr_1 \bowtie Expr_2)$ is an algebra expression, If $Expr_1$ and $Expr_2$ evaluate to R_1 and R_2 repectively, then the full-text relation corresponding to the new expression is: $\mathtt{R}_1 \bowtie_{\mathtt{R}_1.\mathtt{CNode}=\mathtt{R}_2.\mathtt{CNode}} \mathtt{R}_2$, where $\bowtie_{\mathtt{R}_1.\mathtt{CNode}=\mathtt{R}_2.\mathtt{CNode}}$ is the traditional relational equi-join operation on the `CNode` attribute. The duplicate `CNode` attribute is projected out in the result full-text relation, and the position attributes are renamed to be consecutive $\mathtt{att}_\mathtt{i}$'s. Note again how the full-text algebra does not allow operations across nodes because the only predicate that is permitted in the join is equality between the attributes `CNode` of the two relations.
- If $Expr_1$ is an algebra expression, then $\sigma_{pred(\mathtt{att}_1,\ldots,\mathtt{att}_\mathtt{m},\mathtt{c}_1,\ldots,\mathtt{c}_\mathtt{q})}(Expr_1)$ is an algebra expression, where $pred \in Preds$. If $Expr_1$ evaluates to R_1, the full-text relation corresponding to the new expression is: $\sigma_{pred(\mathtt{att}_1,\ldots,\mathtt{att}_\mathtt{m},\mathtt{c}_1,\ldots,\mathtt{c}_\mathtt{q})}(\mathtt{R}_1)$, where σ is the traditional relational selection operator.
- If $Expr_1$ and $Expr_2$ are algebra expression, then $(Expr_1 - Expr_2)$, $Expr_1 \cup Expr_2$, and $Expr_1 \cap Expr_2$ are algebra expressions. These $-$, $\cup$ and $\cap$ operators have the same semantics as in traditional relational algebra.

A full-text algebra query is a full-text algebra expression that produces a full-text relation with a single attribute (this attribute has to be `CNode` by definition). The set of nodes in the result full-text relation defines the result of a full-text algebra query.

We now provide some examples of full-text algebra queries that correspond to the calculus example in Section 2.2.1. The following query returns the context nodes that contain the token 'test' and 'usability': $\pi_{\mathtt{CNode}}(\mathtt{R}_{\mathtt{test}} \bowtie \mathtt{R}_{\mathtt{usability}})$

The following query returns the context nodes that contain the token 'test' and the token 'usability' within a distance of 5: $\pi_{\mathtt{CNode}}(\sigma_{distance(p_1,p_2,5)}(\mathtt{R}_{\mathtt{test}} \bowtie \mathtt{R}_{\mathtt{usability}}))$

The following query returns the context nodes that contain two occurrences of the token 'test' and do not contain the token 'usability': $\pi_{\mathtt{CNode}}((\sigma_{diffpos(att_1,att_2)}(\mathtt{R}_{\mathtt{test}} \bowtie \mathtt{R}_{\mathtt{test}})) \bowtie (\mathtt{SearchContext} - \pi_{\mathtt{CNode}}(\mathtt{R}_{\mathtt{usability}})))$

2.4 Equivalence of Calculus and Algebra and Its Applications

Theorem 1 *Given a set of position-based predicates $Preds$, the full-text calculus and the full-text algebra are equivalent in expressive power.*

The proof is in Appendix A, and is similar to the equivalence proof for the relational calculus and algebra.

The equivalence of the full-text calculus and algebra suggests a notion of completeness for full-text search languages. This provides a formal basis for comparing the expressive power of different query languages, as we shall do in the next section. To the best of our knowledge, this is the first attempt to formalize the expressive power of full-text search languages, either for flat documents or for XML documents. Developing a full-text algebra in terms of relations also provides a foundation for tightly integrating, optimizing and evaluating structured (relational or XML) queries with full-text search.

The full-text algebra also enables us to rank query results by leveraging existing work on the probabalistic relational model developed in the context of IR [19, 38]. Specifically, the probabilistic relational model includes a probability attribute for each tuple that specifies its relevance to the result relation. A tuple with a high probability is very relevant to the result relation, while a tuple with low probability is not. In addition, the model defines how these probabilities are propagated through traditional relational operators. In our context, we simply need to add a new probability attribute to our full-text relations. We can then rely on these techniques to propagate this attribute through the algebra operators, and produce ranked results.

3 Scoring

Scoring (or ranking) 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 [14, 18, 21, 20, 27, 33, 19, 38]. 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 [28] and PRA [19, 38] scoring methods can be incorporated. We only describe how scoring can be done in the context of the FTA; the extension to the FTC is similar.

3.1 TF-IDF Based Scoring

TF-IDF is one of the most common IR scoring methods [28].

Our scoring framework is based on two extensions to our model: (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 [19]. However, unlike [19], the scoring information need not be only a real number (or probability); it can be any arbitrary type that can be extended for the needs of the scoring method. Scoring transformations extend the semantics of FTA operators to transform the scores of the input full-text relations. For example, a selection operator can scale the scores based in the selection predicate (such as distance) and so on.

We now show how TF-IDF scoring can be captured using our scoring framework. We use the following widely-accepted TF and IDF formulae for a node n and a token t: $tf(n, t) = occurs(n, t)/unique\_tokens(n)$ and $idf(t) = ln(1+db\_size/df(t))$, where $occurs(n, t)$ is the number of occurrences of t in n, $unique\_tokens(n)$ is the number of unique tokens in n, db_size is the number of nodes in the database, and $df(t)$ is the number of nodes containing the token t. The TD-IDF scores are aggregated using the cosine similarity:

$score(n) = \Sigma_{t \in q} w(t) * tf(n,t) * idf(t)/(||n||_2 * ||q||_2)$, where q denotes the bag of search tokens in the query, $w(t)$ denotes the weight of the search token t and $|| \cdot ||_2$ is the L_2 measure.

To model TF-IDF, we associate a numeric score with each tuple. Intuitively, the score contains the TF-IDF measure for all the positions in the tuple. Initially, the R_t relations contain the 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 measure 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 measure of t with regards to n. It is also important to note that all of the scoring information in R_t can be precomputed.

To capture TF-IDF score of search tokens, the above tuple score can be scaled by $weight(t)/(unique\_search\_tokens * ||q||_2)$, where $unique\_search\_tokens$ is the number of unique search tokens in the query q. This scale factor is query-dependent and cannot be precomputed. Thus, we can consider that the persistent index structures contain the $idf(t)/(unique\_tokens * ||n||_2)$ score. When the R_t relation is processed, the precomputed score is multiplied by $idf(t)/unique\_search\_tokens * ||q||_2$ to obtain the final score for a tuple t:

$$t.score = idf(t)^2/(unique\_tokens * unique\_search\_tokens * ||n||_2 * ||q||_2)$$

.

We now describe the scoring transformations for each FTA operator.

- Given two expressions $Expr_1$ and $Expr_2$ that evaluate to the full-text relations $\mathtt{R}_1$ and $\mathtt{R}_2$, a tuple t_1 in $\mathtt{R}_1$, a tuple t_2 in $\mathtt{R}_2$ and t_3 in $(Expr_1 \bowtie Expr_2)$ where t_3 is the result of joining t_1 and t_2, i.e., $t_1.CNode = t_2.CNode$, the score formula associated with join is:

  $$t_3.score = t_1.score/|R_2| + t_2/|R_1|$$

  Above, $| \cdot |$ denotes the cardinality of the relation. We need to scale down the $t_1.score$ and $t_2.score$ because their relevance decreases due to the increased number of tuples (solutions) in the resulting relation. Informally, one can think of this as "the first law of thermodynamics" for conservation of energy: the join conserves the total score (energy) of the input relations because it neither adds nor removes solutions (tuples).

- Given an expression $\pi_{\mathtt{CNode},\mathtt{att}_1,\ldots,\mathtt{att}_i,\mathtt{score}}(Expr_1)$ where $Expr_1$ evaluates to R_1 and all tuples $t_1 \ldots t_n$ in R_1 that project out onto the same output tuple t_3 (i.e., they share thesame values for the projected attributes), the score formula associated with projection is:

  $$t_3.score = \Sigma_{i=1,..,n} t_i.score$$

  Projection also obeys the above score-conservation: the new relation should have the same total score as the original one.

- Given $\sigma_{pred(att_1,\ldots,att_n,c_1,\ldots,c_m)}(Expr_1)$ where $Expr_1$ is an algebra expression whose corresponding full-text relation is R_1. Let R_2 is the resulting relation. If t_2 is a tuple in R_2 such that $t_1 = t_2$, then:

  $$t_2.score = t_1.score$$

  .

- Given an expression $\neg Expr_1$ where $Expr_1$ evaluates to R_1 and t is a tuple in R_1, the score formula associated with the negation is: $t.score = 1 - t.score$ (tie negation to difference in the definition of the algebra).

- Given $(Expr_1 \cup Expr_2)$ where $Expr_1$ and $Expr_2$ are algebra expressions whose corresponding full-text relations are $\mathtt{R}_1$ and R_2 and t_1 is a tuple in R_1 and t_2 is a tuple in R_2 and t_3 is the result of the union of $t1$ and t_2, the score formula associated with the union is:

  $$t_3.score = t_1.score + t_2.score$$

  .

  We assume that if $t_i.score = 0$ if $\nexists t_i \in R_i\ t_i = t_3$ for $i = 1, 2$; i.e., missing tuples are assumed to have score 0.

- Given $(Expr_1 - Expr_2)$ where $Expr_1$ and $Expr_2$ are algebra expressions whose corresponding full-text relations are R_1 and R_2. Let R_3 is the resulting relation. If t_3 is a tuple in R_3 such that $t_1 = t_3$, then:

  $$t_3.score = t_1.score$$

  .

- Similarly, given $(Expr_1 \cap Expr_2)$ where $Expr_1$ and $Expr_2$ are algebra expressions whose corresponding full-text relations are $R_1(CNode, att_1, ..., att_n)$ and $R_2(CNode, att_1, ..., att_n)$. Let R_3 is the resulting relation. Let t_1 is a tuple in R_1 and t_2 is a tuple in R_2 such that $t_1.CNode = t_2.CNode, t_1.att_1 = t_2.att_1, ..., t_1.att_n = t_2.att_n$, and $t_3 \in R_3$ be the resulting tuple, then:

  $$t_3.score = \text{Min}(t_1.score, t_2.score)$$

  .

The following theorem holds.

Theorem 2 *The TF-IDF propagation of scores preserves the traditional semantics of TF-IDF for conjunctive and disjunctive queries.*

Proof sketch. We consider restricted FTC expressions of the form $\{node \mid hasPos(node) \wedge QueryExpr(node)\}$ where $QueryExpr(node)$ can be one of the following

- $hasPos(node, p) \wedge hasToken(p, t)$ for some $p \in \mathcal{P}, t \in \mathcal{T}$

- $(QueryExpr_1(node)) \wedge (QueryExpr_2(node))$ for some restricted FTC expressions $QueryExpr_1(node)$ and $QueryExpr_1(node)$

- $(QueryExpr_1(node)) \vee (QueryExpr_2(node))$ for some restricted FTC expressions $QueryExpr_1(node)$ and $QueryExpr_1(node)$

We assume that all search tokens are distinct. This can be achieved by considering the search token position in the query to be part of the search token. Notice that this does not influence the TF-IDF score of query results. Let two search tokens t_1 and t_2 have the same TF measure tf and IDF measure idf. Let the weight of the first one be w_1 and the weight of the second one be w_2. Then their combined TF-IDF score is $(w_1 * tf * idf + w_2 * tf * idf)/(||n||_2 * ||q||_2) = (w_1 + w_2) * tf * udf/(||n||_2 * ||q||_2)$, i.e. it is the same as a token with weight $w_1 + w_2$.

We use structural induction on the restricted FTC expression E. We will prove the following invariant. Let E_1 is a subexpression of E. Let $AExpr_1$ be its corresponding FTA expression. Then, for every attribute att_i in the resulting relation R_1 of $AExpr_1$ and its corresponding search token q_i, the following holds $\forall u \in \pi_{CNode,att_i}(AExpr_1)\ u.score = score(u.CNode, q_i)$. Here, $score(n, q_i) = w(q_i) * tf(n, q_i) * idf(q_i)/(||n||_2 * ||q||_2)$ is the score of the search context node $n \in \mathbb{N}$ with respect to the token q_i.

- Let $E = hasPos(node, p) \wedge hasToken(p, t)$ for some $p \in \mathcal{P}, t \in \mathcal{T}$, i.e. we are searching for the token t. The corresponding FTA expression is $\pi_{CNode}(\mathtt{R}_t)$. The score of every $n \in \mathbb{N}$ is

$$\begin{aligned} score(n) &= \sum_{u \in \mathtt{R}_t} u.score = \sum_{u \in \mathtt{R}_t} \frac{idf(t)^2}{unique\_tokens * unique\_search\_tokens * ||n||_2 * ||q||_2} && (1) \\ &= \frac{occurs * idf(t) * idf(t)}{unique\_tokens * unique\_search\_tokens * ||n||_2 * ||q||_2} && (2) \\ &= \frac{w(t) * tf(n,t) * idf(t)}{||n||_2 * ||q||_2} && (3) \end{aligned}$$

  This is exactly the TF-IDF score with respect to the search token t.

- Let $E = (QueryExpr_1(node)) \wedge (QueryExpr_2(node))$. Let $QueryExpr_1(node)$ and $QueryExpr_2(node)$ have corresponding FTA expressions $AExpr_1$ and $AExpr_2$ respectively. Let R_1 and R_2 be the results of the evaluation of $AExpr_1$ and $AExpr_2$. Remember that the search tokens (i.e. postition attributes in the resulting full-text relations) are distinct. As in the proof of Theorem 1, the FTC expression E evaluates to the relation $R(CNode, att_1, ..., att_n)$ that is the result of $AExpr_1 \bowtie AExpr_2$.

  Let att_i is a position attribute of R. Without loss of generality, att_i also belongs to the relation R_1. Using the induction hypothesis, we get that $\forall u \in \pi_{CNode,att_i}(AExpr_1)\ u.score = score(u.CNode, q_i)$. We have that $\pi_{CNode,att_i}(AExpr_1 \bowtie AExpr_2) = \pi_{CNode,att_i}(AExpr_1)$ because $AExpr_1$ and $AExpr_2$ evaluate to relations that have no position attributes in common. Furthermore, for every tuple $u \in R_1$, there exist exactly $|R_2|$ tuples $v_1, ..., v_{|R_2|}$, each with score $u.score/|R_2|$, such that $u.CNode = v_j.CNode \wedge u.att_i = v_j.att_i$ for $j = 1, ..., |R_2|$. Consequently, $\sum_{j=1}^{|R_2|} v_j.score = u.score = score(u.CNode, q_i)$.

  Let $v \in \pi_{CNode,att_i}(AExpr_1 \bowtie AExpr_2) = \pi_{CNode,att_i}(AExpr_1)$. Thus, there exist tuples $v_1, ..., v_{|R_2|}$ such that $v.CNode = v_j.CNode \wedge v.att_i = v_j.att_i$ for $j = 1, ..., |R_2|$. Therefore, $v.score = \sum_{j=1}^{|R_2|} v_j.score = score(u.CNode, q_i)$.

- The case $E = (QueryExpr_1(node)) \vee (QueryExpr_2(node))$ is similar to the previous one.

QED

Further, this scoring method is more powerful than traditional TF-IDF because it can be generalized to arbitrarily complex queries (not just simple conjunctive and disjunctive queries) by defining appropriate scoring transformations for the other operators. For instance, we can define a scoring transformation for distance selection predicates thereby extending the scope of TF-IDF scoring.

3.2 Probability Based Scoring

One of the popular scoring methods employed by the IR community is using probability-based measures to indicate the relevance of a context node to a full-text search condition. The formal underpinnings for this work is specified by the probabilistic relational model [19, 38]. Specifically, this model includes a probability attribute for each tuple that specifies its score (relevance) to the result relation. A tuple with a high probability score is very relevant to the result relation, while a tuple with low probability score is not. In addition, the model defines how these probabilities are propagated through traditional relational operators.

It is easy to incorporate the above probability-based scoring in the FTA; we simply need to add a new probability attribute to the full-text relations. This new attribute will represent the probability (score) of each tuple in the full-text relation. Since all FTA operations are specified in terms of relational algebra operations, we can directly use the techniques in the probabilistic relational model to propagate the scores for arbitrarily complex FTA expressions.

The probabilistic relational algebra is the most prominent scoring method in full-text search [19]. This algebra operates on tuples with a score attribute. The score of a tuple represents the probability associated with that tuple. A score formula is associated with each operator with transforms its input tuples scores into output tuples scores. We adapt the relational probabilistic model to our algebra. Every full-text relation $\mathtt{R_{token}}$, where $token \in \mathcal{T}$, is augmented with a $score$ attribute. Conceptually, the score of a tuple in $\mathtt{R_{token}}$ represents the probability that that tuple contains $token$. Hence, the value of $score$ should be a float between 0 and 1. This value can be computed using a variety of techniques, including TF and IDF [31]. For example, if TF-IDF is used, then the score of each tuple could be defined as IDF/NF, where NF is the normalizing factor used in computing the TF-IDF score (using the formula TF*IDF/NF). We associate a score formula with each operator in our algebra. Each formula guarantees that output tuples will have a score value between 0 and 1. In the following, we assume that every full-text relation R_i has a $score$ attribute.

- Given an expression $\pi_{\mathtt{CNode,att_1,\ldots,att_i,score}}(Expr_1)$ where $Expr_1$ evaluates to $\mathtt{R_1}$ and all tuples $t_1 \ldots t_n$ in $\mathtt{R_1}$ that project out onto the same output tuple t_3 (i.e., they share thesame values for the projected attributes), the score formula associated with projection is:
  $t_3.score = 1 - (1 - t_1.score) \times (1 - t_2.score) \times \ldots \times (1 - t_m.score)$
  This formula aggregates the scores of input tuples whose value is between 0 and 1 into a single score whose value is between 0 and 1.

- Given two expressions $Expr_1$ and $Expr_2$ that evaluate to the full-text relations $\mathtt{R_1}$ and $\mathtt{R_2}$, a tuple t_1 in $\mathtt{R_1}$, a tuple t_2 in $\mathtt{R_2}$ and t_3 in $(Expr_1 \bowtie Expr_2)$ where t_3 is the result of joining t_1 and t_2, i.e., $t_1.CNode = t_2.CNode$, the score formula associated with join is:

  $t_3.score = t_1.score \times t_2.score$ Note that the join preserves the fact that the score of tuples has to be a value between 0 and 1.

- Given an expression $\sigma_{pred(\mathtt{att_1,\ldots,att_m,c_1,\ldots,c_q})}(Expr_1)$ where $Expr_1$ evaluates to $\mathtt{R_1}$, the score formula associated with a predicate depends on the predicate $pred$. Therefore, given a tuple t in $\mathtt{R_1}$, its score is defined as follows:

$t.score = t.score \times f$ where f is a function associated with the predicate and evaluates to a value between 0 and 1. For example, the function associated with the predicate $distance(p_1, p_2, dist)$ is: $f = 1 - |t.p_1 - t.p_2|/dist$.

- Given an expression $\neg Expr_1$ where $Expr_1$ evaluates to $\mathtt{R}_1$ and t is a tuple in $\mathtt{R}_1$, the score formula associated with the negation is: $t.score = 1 - t.score$ (tie negation to difference in the definition of the algebra).
- Given $(Expr_1 \cup Expr_2)$ where $Expr_1$ and $Expr_2$ are algebra expressions whose corresponding full-text relations are $\mathtt{R}_1$ and $\mathtt{R}_2$ and t_1 is a tuple in $\mathtt{R}_1$ and t_2 is a tuple in $\mathtt{R}_2$ and t_3 is the result of the union of $t1$ and t_2, the score formula associated with the union is: $t_3.score = 1 - (1 - t_1.score) \times (1 - t_2.score)$
- Given two expressions $Expr_1$ and $Expr_2$ that evaluate to the full-text relations $\mathtt{R}_1$ and $\mathtt{R}_2$, a tuple t_1 in $\mathtt{R}_1$, a tuple t_2 in $\mathtt{R}_2$ and t_3 in $(Expr_1 \cap Expr_2)$ where t_3 is the result of joining t_1 and t_2, i.e., $t_1.CNode = t_2.CNode$, the score formula associated with join is: $t_3.score = t_1.score \times t_2.score$ Intuitively, the intersection can be interpreted as a join on all attributes.
- The score for the case $Expr_1 - Expr_2$ can be derived using $Expr_1 - Expr_2 = Expr_1 \cap \neg Expr_2$.

4 Completeness of Full-text Search Languages

In this section, we show the incompleteness of existing full-text languages with respect to the algebra and calculus. We then define a complete full-text language based on the full-text calculus that naturally generalizes existing languages.

4.1 Incompleteness of Boolean Full-Text Search Languages

Boolean full-text search languages are commonly used in IR, and have also been proposed for XML full-text search [18, 33]. A typical syntax for such languages, which we shall call `BOOL`, is given below. The simplest query is a search token, which can either be a string literal (such as 'test') or the keyword `ANY`, which matches any token in a node. In addition, the query can be composed with Boolean operators.

Query := Token | `NOT` Query | Query `AND` Query | Query `OR` Query

Token := StringLiteral | `ANY`

We can recursively define the semantics of `BOOL` in terms of our calculus. If the query is a StringLiteral `'token'`, it is equivalent to the calculus query expression $\exists p(hasPos(n, p) \wedge hasToken(p, 'token'))$. If the query is `ANY`, it is equivalent to the expression $\exists p(hasPos(n, p))$. If the query is of the form `NOT Query`, it is equivalent to $\neg Expr$, where $Expr$ is the calculus expression for `Query`. If the query is of the form `Query1 AND Query2`, it is equivalent to $Expr1 \wedge Expr2$, where $Expr1$ and $Expr2$ are calculus expressions for `Query1` and `Query2` respectively. `OR` is defined similarly. As an example, the query `'test' AND NOT 'usability'` is equivalent to the calculus query expression: $\exists p_1(hasPos(n, p_1) \wedge hasToken(p_1, 'test')) \wedge \neg(\exists p_2 hasPos(n, p_2) \wedge hasToken(p_2, 'usability'))$.

Obviously, `BOOL` cannot express position-based predicates. However, we now show that even if we disallow such predicates in the calculus (i.e., $Preds = \phi$), `BOOL` is still incomplete if $\mathcal{T}$ is infinite.

Theorem 3 *If $\mathcal{T}$ is infinite, there exists a full-text query that can be expressed in the full-text calculus with $Preds = \phi$, but which cannot be expressed by* `BOOL`.

Proof Sketch: We shall show that no query in `BOOL` can express the following calculus query: $\exists p(hasPos(n,p) \wedge \neg hasToken(p,t_1))$ (i.e., *find context nodes that contain at least one token that is not* t_1), where $t_1 \in \mathcal{T}$. The proof is by contradiction. Assume that there exists a query $\mathcal{Q}$ in `BOOL` that can express the calculus query. Let $\mathcal{T}_Q$ be the set of tokens that appear in $\mathcal{Q}$. We construct two context nodes CN_1 and CN_2. CN_1 contains only the token t_1. CN_2 contains the token t_1 and one other token $t_2 \in \mathcal{T} - (\mathcal{T}_Q \cup \{t_1\})$ (such a token t_2 always exists because $\mathcal{T}$ is infinite and $\mathcal{Q}$ is finite). By the construction, we can see that CN_1 does not satisfy the calculus query, while CN_2 does. We will now show that $\mathcal{Q}$ 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 $\mathcal{Q}$. We show by induction on the structure of C_Q that 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 $\exists p(hasPos(n,p) \wedge hasToken(p, token))$, it returns true for both CN_1 and CN_2 if $token = t_1$, and false if $token \neq t_1$ (by construction of CN_1 and CN_2 - recall that $token$ appears in $\mathcal{Q}$). 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 we limit $\mathcal{T}$ to be finite, however, we can prove that `BOOL` is complete with $Preds = \phi$.

Theorem 4 *If $\mathcal{T}$ is finite, every query that can be expressed in the full-text calculus with $Preds = \phi$ can be expressed in* `BOOL`.

The proof is presented in Appendix A. The main intuition is that, if $\mathcal{T}$ is finite, we can express queries such as: $\exists p(hasPos(n,p) \wedge \neg hasToken(p,t_1))$ in `BOOL` by explicitly listing all the tokens that are not t_1. Although `BOOL` is complete under this assumption, it is not always practical because even for simple queries such as the one above, we need to explicitly list all possible tokens other than t_1 in the query.

4.2 Incompleteness of Existing Predicate-Based Full-Text Search Languages

We now consider full-text languages that have position-based predicates in addition to Boolean operators [3, 7]. A typical syntax for such a language, which we shall call `DIST`, is given below.

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

The semantics of `DIST` is the same as `BOOL`, except for the addition of `dist(Token,Token,Integer)`. This construct is the equivalent of the $distance$ predicate introduced in the calculus (Section 2.2), and specifies that the number of intervening tokens should be less than the specified integer. More formally, the semantics of dist(t_1,t_2,d) for some tokens t_1 and t_2 and some integer d is given by the calculus expression: $\exists p_1(hasPos(n,p_1) \wedge hasToken(p_1,t_1) \wedge \exists p_2(hasPos(n,p_2) \wedge hasToken(p_2,t_2) \wedge distance(p_1,p_2,d)))$. If t_1 or t_2 is `ANY` instead of a string literal, then the corresponding $hasToken$ predicate is omitted in the semantics. We now show that `DIST` is incomplete with respect to the calculus so long as $\mathcal{T}$ is not trivially small. We can also prove similar incompleteness results for other position-based predicates.

Theorem 5 *If $|\ \mathcal{T}\ | \geq 2$, there exists a full-text query that can be expressed in the full-text calculus with $Preds = \{distance(p_1,p_2,d)\}$, but which cannot be expressed by* `DIST`.

Proof Sketch: We shall show that no query in `DIST` can express the following calculus query: $\exists p_1(hasPos(n,p_1) \wedge \exists p_2(hasPos(n,p_2) \wedge hasToken(p_1,t_1) \wedge hasToken(p_2,t_2) \wedge \neg distance(p_1,p_2,0)))$, where $t_1 \in \mathcal{T}$, $t_2 \in \mathcal{T}$ and $t_1 \neq t_2$ (i.e., *find context nodes where the tokens t_1 and t_2 do not appear next* *to each other at least once*). The proof is by contradiction. Assume that there exists a query $\mathcal{Q}$ in 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_1. CN_2 contains the tokens t_1 followed by t_2 followed by t_1 followed by t_2. By the construction, we can see that CN_1 does not satisfy the calculus query, while CN_2 does. Using induction on the structure of $\mathcal{Q}$ similar to the proof of Theorem 3, we can show that $\mathcal{Q}$ either returns both CN_1 or CN_2 or neither of them. This is a contradiction. $\Box$

4.3 A Complete Full-Text Query Language

We now present a new language COMP based on the full-text calculus that is complete even in the presence of arbitrary position-based predicates. COMP shares the same syntax as BOOL for simple Boolean queries, but naturally generalizes BOOL with position variables to achieve completeness. Thus, simple queries retain the same conventional syntax, while new constructs are only required for more complex queries.

Query := Token | NOT Query | Query AND Query | Query OR Query | SOME Var Query | EVERY Var Query | Preds
Token := StringLiteral | ANY | Var HAS StringLiteral | Var HAS ANY
Preds := distance(Var,Var,Integer) | diffpos(Var,Var) | ...

The main additions to BOOL are the HAS construct in Token, and the SOME, EVERY and Preds constructs in Query (the semantics of the other constructs remain unchanged from BOOL). 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 calculus, where tok is a StringLiteral is: $hasToken(var_1, tok)$. 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 and EVERY constructs 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 calculus expression semantics for Query. The semantics of 'EVERY var_1 Query' is $\forall var_1(hasPos(n, var_1) \Rightarrow Expr)$, where $Expr$ is the calculus expression semantics for Query. Finally, the Preds construct allows for the definition of arbitrary position-based predicates. The semantics of a predicate 'pred(var_1,...,var_p,c_1,...c_q)', is simply: $pred(var_1, \ldots, var_p, c_1, \ldots, c_q)$.

As an illustration of the power of COMP, the following two queries express the calculus queries used to prove the incompleteness of BOOL and DIST in Theorems 3 and 5, respectively.

SOME p_1 (NOT p_1 HAS t_1)
SOME p_1 SOME p_2 (p_1 HAS t_1 AND p_2 HAS t_2 AND NOT distance(p_1,p_2,0))

We can prove that COMP is complete (the proof is in the appendix).

Theorem 6 *Every query that can be expressed in the full-text calculus using predicates $Preds$ can be expressed by* COMP *using $Preds$.*

5 Query Complexity and Evaluation Algorithms

While one important aspect of a full-text language is expressibility (discussed in the previous section), another important aspect is query complexity, i.e., the efficiency of evaluating a query in a full-text language. In this section, we study the query complexity of different full-text languages and develop efficient query evaluation algorithms. Due to space constraints, we will only sketch the algorithms to evaluate NPRED queries.

Like other formal languages, full-text languages have a tradeoff between expressibility and query complexity: the more expressive the language, the greater its query complexity. We formalize this notion by developing a complexity hierarchy of full-text languages based on the inverted list [28] model for query evaluation commonly used in the IR community (see Section 5.1.2). At the top of our complexity hierarchy is COMP, which is the most expressive but which also has the greatest query complexity. At the bottom of the hierarchy is BOOL, which is the least expressive but also has the lowest query complexity. We also identify two new classes of languages between these two extremes: PPRED, which stands for a subset of COMP restricted to "Positive" PREDicates, and NPRED, which stands for a subset of COMP restricted to "Negative" PREDicates (we shall formally define positive and negative predicates in Sections 5.5 and 5.6). PPRED includes most common full-text predicates, such as distance and samepara, but is more powerful than existing full-text languages such as DIST. NPRED is a superset of PPRED and includes the negations of most common full-text predicates.

An interesting result of our study is that the query evaluation complexity of PPRED is linear in the size of the query token inverted lists, and quadratic in the size of the query. Specifically, in Section 5.5, we present an algorithm whereby PPRED queries can be evaluated in a single scan over the query term inverted lists. This illustrates a practical application of our formalism: developing full-text languages such as PPRED that are more powerful than existing full-text predicate languages (such as DIST), but which can still be evaluated efficiently in a single scan over inverted lists. Similarly, we also show in Section 5.6, that the query evaluation complexity of NPRED is linear in the size of the query inverted lists but, in some case, exponential in the size of the query – this additional complexity is the price paid for negation.

We note that our focus in this section is on establishing query complexity *upper bounds* for the various full-text languages by developing concrete, efficient, and practical query evaluation algorithms (especially for PPRED and NPRED). Exploring query complexity lower bounds is beyond the scope of this paper, and is part of future work. We now start by describing our complexity model.

5.1 Complexity Model

Our study of the complexity of full-text search languages is similar in spirit to the vast body of work on the complexity of database query languages (e.g., [8, 9, 23, 36]). However, there are two main reasons why the complexity results for database query languages do not directly apply to our setting.

First, our focus is specifically on full-text search using the inverted list model for data organization (which is the commonly used model in the IR community). Thus, our complexity parameters are expressed in terms of this model, and we establish upper bounds by developing concrete query evaluation algorithms based on this model. In contrast, most database query languages work with arbitrary relations (not just full-text relations and inverted lists); while this leads to more general results, these results do not isolate the complexity of full-text primitives in the context of the inverted list model.

Second, most database query language complexity studies treat *expression complexity* (i.e., the complexity of evaluating a query as a function of the size of the query, assuming the database is the same) [9, 36] and *data complexity* [23, 36] (i.e., the complexity of evaluating a query as a function of the size of the data, assuming that the query is the same) separately. In contrast, we are interested in *combined complexity* (defined, but not explored in [36]), whereby we want to determine the complexity of evaluating a query as a function of *both* the query size and the data size in order to study their *relative impact* on query performance.

5.1.1 Query Model

We characterize a COMP query Q by the following parameters. Since the other full-text languages that we consider are subsets of COMP, these parameters apply to these languages as well.

- $toks_Q$: The number of tokens in Q, including string literals and the universal token ANY.

"usability" inverted list

cn	*PosList*
1	**3 12 39**
51	**56 59**
89	**96 102 108**

"software" inverted lis

cn	*PosList*
1	**25 29 42**
75	**81**

Figure 2: Inverted Lists Examples

- $preds_Q$: The number of predicates in Q.
- ops_Q: The number of operations in Q, where an operation can be `NOT`, `AND`, `OR`, `SOME`, or `EVERY`.

The above parameters characterize the total size of a `COMP` query since they capture all the primitives that can appear in a query.

5.1.2 Data Model

As mentioned earlier, we use the inverted list model [28] for representing context nodes. For each token tok that appears in at least one context node in $\mathcal{N}$, there is an associated inverted list IL_{tok}. IL_{tok} contains a list of one or more *entries*. Each entry is a pair: $(cn, PosList)$, where cn is the id of a context node that contains tok, and $PosList$ is the list of positions in cn that contain tok. The positions in $PosList$ are ordered based on their order of occurrence in cn, and the entries in IL_{tok} are ordered based on the ids of the context nodes. Intuitively, IL_{tok} corresponds to the physical representation of the full-text relation R_{tok} in the FTA. Figure 2 shows example inverted lists for the `usability` and `software` tokens, where the document in Figure 1 is one of the context nodes and has id 1.

In addition to the inverted lists, there is also a list, IL_{ANY}, which contains one entry for each context node in $\mathcal{N}$. Each entry is the pair: $(cn, PosList)$, where cn is the id of a context node, and $PosList$ is the list of positions that occur in cn. Again, the positions in $PosList$ are ordered based on their order of occurrence in cn, and the entries in IL_{ANY} are ordered based on the ids of the context nodes. Intuitively, IL_{ANY} corresponds to the physical representation of the `ANY` full-text relation in the FTA.

One important restriction on inverted lists is that they can only be accessed sequentially (some IR implementations allow restricted random accesses, but we do not consider these extensions here). Specifically, the only way to access an inverted list IL_{tok} (similarly, for IL_{ANY}) is to open a *cursor*. Each cursor sequentially scans IL_{tok} and supports the following two operations.

- **nextEntry()**: The first nextEntry() call moves the cursor to the first entry e in IL_{tok}, and returns the id of the context node in e. Each subsequent call advances the cursor to the next entry in IL_{tok} and returns the corresponding context node id. When all entries have been scanned, nextEntry() returns NULL. The entry pointed to by the cursor at any time is called the *current entry*.
- **getPositions()**: This call returns the list of all positions ($PosList$) in a given entry int he inverted list.

We assume that each invocation of the above operations is executed in $O(1)$ (i.e., constant) time.

Finally, to quantify the size of the inverted lists, we use the following parameters. We use $\mathcal{T}$ to denote the set of all tokens that appear in the context nodes $\mathcal{N}$.

- $\mathbf{cnodes}$: $|\mathcal{N}|$ (the number of context nodes).
- $\mathbf{pos\_per\_cnode}$: $max_{(cn,PosList) \in IL_{ANY}}(|PosList|)$ (the maximum number of positions in a context node).
- $\mathbf{entries\_per\_token}$: $max_{tok \in \mathcal{T}}(|\{e|e \in IL_{tok}\}|)$ (the maximum number of entries in a token inverted list).
- $\mathbf{pos\_per\_entry}$: $max_{tok \in \mathcal{T}} max_{(cn,PosList) \in IL_{tok}}(|PosList|)$ (the maximum number of positions in an entry in a token inverted list).

We note that the above parameters are conservative in the sense that they use the maximum values for the number of positions per context node, etc.; we do this to keep the model simple. However, as we shall soon see, these conservative parameters are still sufficient to clearly separate the query evaluation complexity of the full-text languages that we consider.

5.2 Summary of Complexity Results

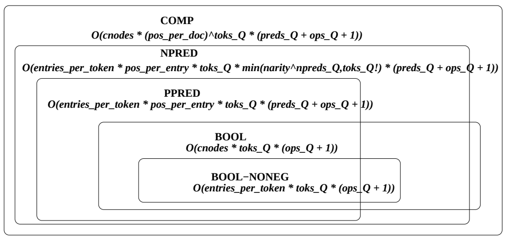
COMP
O(cnodes * (pos_per_doc)^toks_Q * (preds_Q + ops_Q + 1))
NPRED
O(entries_per_token * pos_per_entry * toks_Q * min(narity^npreds_Q,toks_Q!) * (preds_Q + ops_Q + 1))
PPRED
O(entries_per_token * pos_per_entry * toks_Q * (preds_Q + ops_Q + 1))
BOOL
O(cnodes * toks_Q * (ops_Q + 1))
BOOL−NONEG
O(entries_per_token * toks_Q * (ops_Q + 1))

Figure 3: Complexity Hierarchy

Figure 3 summarizes our complexity results; we present the details in the subsequent sections. We represent each language by a bounding box and depict the query complexity of that language (expressed in terms of our complexity parameters) within the bounding box; note that these are upper bounds on the query complexity. If the bounding box of a language A encloses the bounding box of another language B, then all queries in B can be expressed in A. If the bounding boxes of two languages A and B intersect, but no one bounding box contained in the other, then there are some queries that can be expressed in A but not in B, and vice versa.

As shown, the main results are:

- The query complexity of `COMP` is polynomial in the size of the inverted lists and exponential in the size of the query.
- The query complexity of `BOOL` without negation (`BOOL-NONEG`) is linear in the size of the query token inverted lists, and linear in the size of the query.
- The query complexity of `BOOL` (with negation) is linear in the size of the `ANY` list, and linear in the size of the query.
- The query complexity of `PPRED` is linear in the size of the query token inverted lists, and linear in the size of the query.

- The query complexity of NPRED is linear in the size of the query token inverted lists, and possibly exponential in the size of the query.

The above complexity results demonstrate the potential for performance savings using PPRED and NPRED: they reduce the query complexity from *polynomial* in the size of the data (for COMP) to *linear* in the size of the data.

We now discuss the complexity results and query evaluation algorithms in more detail. Due to space constraints, we only briefly discuss BOOL and COMP, and focus mainly on developing efficient query evaluation algorithms for PPRED. We will only sketch the implementation of NPRED due to space limitations.

5.3 BOOL: Evaluation and Complexity

As mentioned in Section 4.1, BOOL is commonly used in IR systems. We first consider a subset of BOOL called BOOL-NONEG, which does not have ANY and does not allow NOT as the first operator. It has the following grammar (note that NOT can only appear along with an AND).

Query := Token | Query AND NOT Query | Query AND Query | Query OR Query

Token := StringLiteral

A common way to evaluate queries in the above language is to merge [28] the inverted lists for the query tokens. For example, consider the query `('software' AND 'users' AND NOT 'testing') OR 'usability'`. The query can be evaluated by merging $IL_{software}$ and IL_{users} (for the first AND) to determine the context node ids that contain both tokens. This result can then be merged with $IL_{testing}$ to determine the context node ids that do not contain `'testing'` (for NOT). Finally, this result can be merged with $IL_{usability}$ to determine the union of the context node ids (for OR). Since the inverted list entries are sorted by the context node ids, each merge can be done in a single scan over the query token inverted lists. Since the total size of the relevant parts of the query token inverted lists is $entries\_per\_token \times toks_Q$ (since BOOL-NONEG ignores positions), and each inverted list entries are scanned at most once for each operator, the query evaluation complexity of BOOL-NONEG is: $O(entries\_per\_token \times toks\_Q \times (ops_Q + 1)$.

In contrast to BOOL-NONEG, BOOL allows ANY and NOT to appear anywhere in the query (Section 4.1). Since ANY and NOT require access to IL_{ANY} (to find all positions in a context node), and IL_{ANY} has $cnodes$ entries, the query complexity of BOOL is: $O(cnodes \times toks\_Q \times (ops_Q + 1))$.

A scoring formula is associated with each Boolean operator in BOOL and BOOL-NONEG as is defined in [19]. Initially, a score is associted with each entry in the inverted lists and are modified by each Boolean operator in the query plan.

5.4 COMP: Evaluation and Complexity

As discussed in Section 4.3, COMP has a one-to-one mapping to the FTC. Since the FTC is a Quantified Boolean Formula (QBF), it 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) [36]. It is an open question as to whether LOGSPACE is a strict subset of PTIME (polynomial time), and whether PSPACE is a strict subset of EXPTIME (exponential time). Thus, for all practical purposes given our current knowledge, 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. We now outline one such algorithm.

The basic idea is to translate the FTC expression corresponding to a COMP query into an equivalent FTA expression (using the equivalence of FTC and FTA given in Section 2.4). The FTA expression can then be evaluated using regular relational operators. As an illustration, consider the following COMP query: SOME p_1 SOME p_2 (p_1 HAS 'usability' AND p_2 HAS 'software' AND

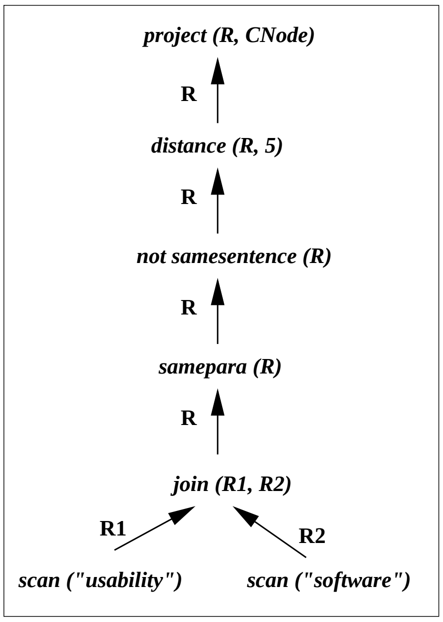

Figure 4: Query Plan Example

`samepara(`p_1`,`p_2`) AND` $\neg$ `samesentence(`p_1`,`p_2`) AND distance(`p_1`,`p_2`,5))` (return context nodes that contain the words ”usability” and ”software” in the same paragraph, in that order, within at most 5 words of each other). The resulting FTA expression tree is shown in Figure 4.

The complexity of evaluating a `COMP` query is thus bounded by the complexity of the FTA operators. Since all FTA operators except the join operator have complexity linear in the size of their input, we focus on the join operator. If the join operator takes in two inputs I and J, and I has p tuples for context node cn, and J has q tuples for context node cn, then the result has $p \cdot q$ tuples for context node cn (since the full-text join operator always performs an equi-join on the context nodes). Thus, the worst case complexity of a join is a cartesian product of the number of tuples *per context node*. Since there can be at most $toks_Q$ joins in a `COMP` query, and the query can access the IL_{ANY} relation in general (with size $cnodes \times pos\_per\_cnode$), the complexity of a `COMP` is:

$O(\mathbf{cnodes} \times (\mathbf{pos\_per\_cnode})^{\mathbf{toks_Q}} \times (\mathbf{preds_Q} + \mathbf{ops_Q} + 1))$

Scoring in `COMP` is handled by each operator in the query plan as defined in Section 3. This also applies to `PPRED` and `NPPRED`.

5.5 PPRED: Evaluation and Complexity

`PPRED` is a subset of `COMP` that restricts the use of negation and only allows “positive” predicates (which will shall formalize soon), which actually include most common predicates used in the IR community. The surprising aspect of `PPRED` is that, by placing these restrictions, it can guarantee that queries can be run in *linear* time over the size of the query inverted lists, instead of in *polynomial* time; for large, practical data sets, this translates to a huge gain in performance. The grammar for `PPRED` is given below, where Query* refers to a Query with no free variables.

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

Token := StringLiteral | Var `HAS` StringLiteral

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

Like `BOOL-NONEG`, `PPRED` only allows negations to appear in the context of an `AND` and cannot explicitly specific `ANY`. Both of these restrictions ensure that IL_{ANY} does not need to be accessed during query processing. Since query operations can be implemented in a linear scan over the query inverted lists in the presence of "positive" predicates, we have the following query complexity for `PPRED`:

$$O(\mathbf{entries\_per\_token} \times \mathbf{pos\_per\_entry} \times toks_Q \times (\mathbf{preds_Q} + \mathbf{ops_Q} + 1))$$

We now describe the intuition behind positive predicates and how it enables an efficient linear query evaluation algorithm. We then formalize the properties and algorithms.

5.5.1 Positive Predicates: Intuition

Intuitively, positive predicates are those that are true in a contiguous region in the position space, and are false outside of this region. For instance, the `distance` predicate is true in the region where both positions are within the distance limit, and false outside this region. A more complex example of a positive predicate is `ordered`, where the region specifies the part of the position space where the positions are in the required order. Other common full-text predicates such as `samepara`, `window`, etc. are also positive predicates. Given positive predicates, how can we use their property to devise efficient query evaluation algorithms?

Recall from the complexity discussion in `COMP` that the main source of complexity stems from the evaluation of the join operation, which computes the cartesian product of the number of tuples *per context node*. If the query contains only positive predicates, we can avoid computing this cartesian product, while still producing the correct results. The key idea is to *skip over* continuous regions of positions in the cartesian product by exploiting the property of positive predicates, without missing any answer to a query. This skipping over is done in *increasing order* of positions, and hence can be done in a linear scan over the inverted lists.

As an illustration, consider the following query: `SOME` p_1 `SOME` p_2 `(`p_1 `HAS 'usability' AND` p_2 `HAS 'software' AND distance(`p_1`,`p_2`,5))` (return context nodes that contain the words "usability" and "software" within at most 5 words of each other). Consider evaluating the query over the inverted lists shown in Figure 2. A naive evaluation plan would be to join $IL_{usability}$ and $IL_{testing}$ on the context node, and compute the cartesian product of positions, and then apply the distance predicate. For the context node with id 1, this corresponds to computing 9 pairs of positions (3 in each inverted list), and then only selecting the final pair (39,42) that satisfies the distance predicate. However, it is sufficient to determine the answer by only scanning 6 pairs of positions (3 + 3 instead of 3 * 3).

Specifically, we start with the smallest pair of positions (3,25) and check whether it satisfies the distance predicate. Since it does not, we move *the smallest position* to get (12,25). Since this does not satisfy the predicate again, we move the smallest position to get (39,25), and so on until we find the solution (39,42). Note that we only scan each inverted list position exactly once, so the complexity is linear in the size of the inverted lists. The reason we were able to move the smallest position is because the distance predicate is true in a contiguous region, and if the predicate is false for the smallest position in the region, we can infer that it is also false for other positions without having to explicitly enumerate them.

5.5.2 Positive Predicates: Definition

We now formally define positive predicates.

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

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

Intuitively, the property states that for every tuple of positions that do not satisfy the predicate (a) there exists a contiguous area, in which all tuples do no satisfy the predicate; this area is specified in terms of the functions $f_i(p_1, ..., p_n)$, which specifies the lower bound of the next possible solution, and (b) at least one $f_i(p_1, ..., p_n)$ has value greater than p_i; this specifies which position inverted list can be moved forward without compromising correctness.

As mentioned earlier, predicates such as $distance$, $samepara$, $ordered$ are positive predicates. For instance, for the 2-ary distance predicate (we only count position parameters in the arity), $f_1(p_1, p_2) = p_1 + 1$ if $p_2 > p_1$, and $= p_1$ otherwise. Similary, $f_2(p_1, p_2) = p_2 + 1$ if $p_1 > p_2$, and $= p_2$ otherwise. $samepara$ and $ordered$ have similar f_i functions.

5.5.3 PPRED Query Evaluation Algorithms

We now present the algorithm for evaluating a PPRED query. The query is first rewritten to push down projections wherever possible so that spurious positions are not propagated. Given the resulting PPRED query, an operator tree is constructed based on the FTA operators. Figure 4 shows a sample query evaluation 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 $p_1, ..., p_n$, 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, $p_1, ..., p_n$ such that: $(node, p_1, ..., p_n) \in R \wedge \forall p'_1, ..., p'_n (node, p'_1, ..., p'_n) \in R \Rightarrow p'_1 \geq p_1 \wedge ... \wedge p'_n \geq p_n$ (i.e., it sets positions $p_1, ...p_n$ 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 $p_1, ..., p_n$ are updated as before.

- **getNode()**: Returns the current value of $node$.

- **advancePosition(i,pos)**: It sets the values of $p_1, ..., p_n$ such that they satisfy: $(node, p_1, ..., p_n) \in R \wedge p_i > pos \wedge \forall p'_1, ..., p'_n (node, p'_1, ..., p'_n) \in R \wedge p'_i \geq pos \Rightarrow (p'_1 \geq p_1 \wedge ... \wedge p'_n \geq p_n)$ (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_is to be NULL and returns false.

- **getPosition(i)**: Returns the current value of p_i.

Given the operator evaluation tree in Figure 4, the general evaluation scheme proceeds as follows. To find a solution **advanceNode** is called on the top project operator which simply forwards this call to the distance selection operator below it. The latter tries to find a solution by continuously calling **advancePosition** on the ordered selection operator below it until it finds a satisfying tuple of positions (see more details about the exact algorithm below). The ordered selection operator behaves in a similar manner: it advances through the result tuples of the underlying operator 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. Notice that the entire evaluation is pipelined and no intermediate relations need to be materialized.

We now show how the different `PPRED` operators can implement the above API. The API implementation for the scan operator is straightforward since it directly operates on the inverted list. We will thus focus on the join operator (Algorithm 1) and the select operator for evaluating predicates (Algorithm 2). The algorithms for the project (Algorithm 3), union (Algorithm 4), and set difference (Algorithm 5) operators are essentially the same as in the relational model.

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 sort-merge 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 the select operator implementing predicate $pred$ with functions f_i (see definition in the beginning of the section). Each invocation of `advanceNode`, advances $node$ until one that satisfies the predicate is found, or there are no $node$s 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 $pred$. The implementation of `advancePosition` is similar. It first advances the position on its input, and then invokes `advancePosUntilSat` until a set of positions that satisfy $pred$ are found.

The `advancePosUntilSat` function 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 is the core operation in the select operator: scanning the input positions until a match is found. The properties of positive predicates enable us to do this in a single pass over the input.

Algorithm 3, Algorithm 4, and Algorithm 5 contain a typical implementation similar to the one often used for relational algebra operators. The **advanceNode** function for the project operator is trivial. The **advancePosition** function for the same operator saves the current values of the projected-out columns and advances the specified cursor until a new set of values for the projected-out columns is found. The union operator performs a merge between the two inputs, always returning the smaller node identifier (for **advanceNode**) or the smaller tuple in lexicographic order (for **advancePosition**). Finally, the difference operator implements only the **advanceNode** function (it works only at the level of nodes) by always returning the first node from the first input not found in the second input.

5.5.4 Correctness and Complexity

We now present a sketch of the proof of correctness of the above algorithm. The proof has two parts: (1) soundness, i.e., every result returned by the algorithm is a result of evaluating the corresponding `PPRED`

Algorithm 1 PPRED 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();
 3: node2 = inp2.advanceNode();
 4: while node1 != NULL && node2 != NULL && node1 != node2 do
 5:    if node1 $<$ node2 then node1 = inp1.advanceNode();
 6:    else node2 = inp2.advanceNode(); end if
 7: end while
 8: if node1 == NULL || node2 == NULL then
 9:    return NULL;
10: else {node1 == node2}
11:    set $p_i$ ($i < c_1$) to inp1.getPosition(i);
12:    set $p_i$ ($i \geq c_1$) to inp2.getPosition($i - c_1$);
13:    node = node1;
14:    return node1;
15: end if}
16:
17: boolean advancePosition(i,pos) {
18: if $i < c_1$ then
19:    result = inp1.advancePosition(i,pos);
20:    if result then
21:       $p_i$ = inp1.getPostion(i);
22:    end if
23:    return result;
24: else
25:    //Similary for inp2
26: end if}
```

query and (2) completeness, i.e., proving that the algorithm does not miss any query results.

First, to prove the soundess we use structural induction on the structue of the operator evaluation tree.

- If the current operator is a scan operator for the token t, the corresponding FTA expression is R_t. Then, **advanceNode** moves the cursor to the first inverted list entry corresponding to the next context node, and **advancePosition** moves the cursor to the next inverted list entry. Given the direct correspodence between the inverted list and the token relation R_t, the new position obviously belongs to the result of the FTA expression.

- If the current operator is a selection operator for the positive predicate $pred(p_1, ..., p_m, c_1, ..., c_q)$, then the corresponding FTA expression is $AlgExpr = \sigma_{\texttt{pred(att}_1,...,\texttt{att}_m,\texttt{c}_1,...,\texttt{c}_q\texttt{)}}(\texttt{Expr}')$, where $Expr'$ corresponds to the nested operator sub-tree. Let's consider **advanceNode**. Lines 2 and 4 guarantee that the current result always satisfies $Expr'$ (induction hypothesis). The loop inlines 15-20 quarantees that the current solution also satisfies the predicate $pred$. The loop in lines 3-5 will not end until both the nested sub-expression and the predicate are satisfied. Therefore, **advanceNode** always produces correct results. Further, similar conclusions can be made for **advancePosition**.

- If the current operator is a join operator between te sub-trees T_1 and T_2, it corresponds to the FTA expression $AlgExpr = E_1 \bowtie E_2$, where E_i corresponds to T_i for $i = 1, 2$. The **advanceNode** algorithm is a trivial sort-merge join. Therefore, the algorithm-produced result is correct (it is a result of $AlgExpr$) iff the results produced by the evaluation of T_1 and T_2 are correct. This is true by

Algorithm 2 PPRED Predicate Evaluation Algorithm

```
Require: $inp$ is API inputs to the predicate with $c$ position columns
 1: Node advanceNode() {
 2: node = inp.advanceNode();
 3: while node != NULL && !advancePosUntilSat() do
 4:    node = inp.advanceNode();
 5: end while
 6: return node; }
 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);
12: return advancePosUntilSat(); }
13:
14: boolean advancePosUntilSat () {
15: while !$pred(p_1, ..., p_c)$ do
16:    find some $i$ such that $f_i(p_1, ..., p_c) > p_i$
17:    success = inp.advancePos($i$, $f_i(p_1, ..., p_c)$);
18:    if !success then return false; end if
19:    $p_i$ = inp.getPosition(i);
20: end while
21: return true; }
```

the induction hypothesis. **advancePosition** simply dispatches the call to the correct cursor from the nested sub-trees and thus, it trivially preserves the correctness.

- If the current operator is a project operator that projects out the columns $i_1, ..., i_m$, then the corresponding FTA expression is $AlgExpr = \pi_{\mathtt{CNode},\mathtt{att_1},...,\mathtt{att_{n-m}}}(\mathtt{E'})$, where the att_i's are the remaining columns and E' corresponds to the nested operator sub-tree T'. **advanceNode** is trivially true because moving the context node always produces a new tuple. With respect to **advancePosition**, we can say that the loop in lines 14-30, guarantees that the algorithm produces the next distinct tuple of projected columns. Again, using the induction hypothesis for E' and T', the correcness is trivial.

- If the current operator is a union operator between the sub-trees T_1 and T_2, then the correspoding FTA expression is $AlgExpr = E_1 \cup E_2$,where E_1 and E_2 correspond to T_1 and T_2. The implementations of both **advanceNode** and **advancePosition** get the next smallest (in lexicographic order) tuple from the input streams. By the induction hypothesis for T_1, T_2, E_1, E_2, our algorithm trivially preserves the correcness.

- If the current operator is a set-difference operator between the sub-trees T_1 and T_2, then the correspoding FTA expression is $AlgExpr = E_1 - E_2$,where E_1 and E_2 correspond to T_1 and T_2. The implementations of **advanceNode** gets the next smallest (in lexicographic order) tuple from the first input stream that is not in the second input stream. By the induction hypothesis for T_1, T_2, E_1, E_2, our algorithm trivially preserves the correcness.

The second part of the proof proves the completeness of the algorithm and is a bit more complex. We prove completeness by inductively showing for each operator that **advanceNode** and the **advancePosition** preserve the invariants shown in the beginning of Section 5.5.3, i.e. they always find minimal solutions for the corresponding operator tree. Therefore, they cannot ”miss” solutions.

Algorithm 3 PPRED Project Evaluation Algorithm

```
Require: $inp$ is API inputs to the project operator; $i_1$, ..., $i_m$ are the columns to be projected out
 1: Node advanceNode() {
 2: node = inp.advanceNode();
 3: return node; }
 4:
 5: boolean advancePosition(i,pos) {
 6: if $i \in \{i_1, ..., i_m\}$ then
 7:    error();
 8: end if
 9:
10: //save current positions
11: for all $j = 1, ..., m$ do
12:    $q_j$ = inp.getPosition($i_j$);
13: end for
14: repeat
15:    success = inp.advancePosition(i,pos);
16:    if !success then return false; endif
17:    for all $j = 1, ..., m$ do
18:       $nq_j$ = inp.getPosition($i_j$);
19:    end for
20: until $\exists j \; q_j \neq nq_j$ }
```

- Let the current operator be a scan operator for the token t. Then, **advanceNode** moves the cursor to the first inverted list entry corresponding to the next context node, and **advancePosition** moves the cursor to the next inverted list entry. Given the direct correspodence between the inverted list and the token relation R_t, the new position obviously belongs to the result of the FTA expression. Further, the minimality of the result (in lexicographic order) is implied by the presence of "the first inverted list entry corresponding to next context node" and "next inverted list entry".

- Let the current operator be a selection operator for the positive predicate $pred(p_1, ..., p_m, c_1, ..., c_q)$. The soundness of the algorithms guarantees the $(node, p_1, ..., p_n \in R$ part of the properties. We will show minimality. We will focus on the **advancePosUntil** algorithm. If it finds a minimal solution, the completeness of both **advanceNode** and **advancePosition** trivially follows. Line 15 guarantees that $!pred(p_{i_1}, ..., p_{i_m}, c_1, ..., c_q)$. Let **advancePosUntil** chooses index i_0 in line 16. The positive-predicates property guarantees that there is at least one such index. Further, the same property guarantees that $!pred(p'_{i_1}, ..., p'_{i_m}, c_1, ..., c_q)$ for every $p'_{i_1}, ..., p'_{i_m}, p_{i_0} \leq p'_{i_0} < f(p_{i_1}, ..., p_{i_n})$, i.e. there is no solution for $p_{i_0} \leq p'_{i_0} < f(p_1, ..., p_n)$. Further, let line 17 moves the cursors to $(p''_1, ..., p''_n)$. The induction hypothesis guarantees that this is the smalles tuple that satisfies the nested sub-expression. Further, we will loop in lines 15-20 until $pred(p''_{i_1}, ..., p''_{i_m}, c_1, ..., c_q)$ gets satisfied. Thus, we have minimality also with respect to $pred$.

- Let the current operator be a join operator between te sub-trees T_1 and T_2. As already pointed out, the **advanceNode** algorithm is a trivial sort-merge join, which guarantees the minimality of the solution (given the induction hypothesis). **advancePosition** simply dispatches the call to the correct cursor from the nested sub-trees and thus, it trivially preserves the minimality too.

- Let the current operator be a project operator that projects out the columns $i_1, ..., i_m$ and with a nested operator sub-tree T'. **advanceNode** is trivially true because moving the context node always produces a new tuple with the minimal possible context node (given the induction hypothesis for T'. Similarly,

Algorithm 4 PPRED Union Evaluation Algorithm

Require: inp_1, inp_2 are API inputs to the union operator; $minIdx$ is the index of the last input advanced (-1 initially)

```
1:  Node advanceNode() {
2:  if minIdx == -1 ||$inp_1$.getNode() == $inp_2$.getNode() then
3:      $hasMore_1$ = $inp_1$.advanceNode() != NULL;
4:      $hasMore_2$ = $inp_2$.advanceNode() != NULL;
5:  else
6:      $hasMore_{minIdx}$ = $inp_{minIdx}$.advanceNode() != NULL;
7:  end if
8:
9:  //do a merge
10: if ! $hasMore_1$ then
11:     if ! $hasMore_2$ then
12:         return NULL;
13:     else
14:         return $inp_2$.getNode();
15:     end if
16: else
17:     if ! $hasMore_2$ then
18:         return $inp_1$.getNode();
19:     else if $inp_2 \prec inp_1$ then
20:         //$inp_2$ precedes lexicographically $inp_1$
21:         $minIdx$ = 2;
22:         return $inp_2$.getNode();
23:     else
24:         $minIdx$ = 1;
25:         return $inp_1$.getNode();
26:     end if
27: end if}
28:
29: boolean advancePosition(i,pos) {
30: //Similar to advanceNode but calling $inp_1$.advancePosition and $inp_2$.advancePosition }
```

the loop in lines 14-30 guarantees that the algorithm produces the next (in lexicographic order) distinct tuple of projected columns. Again, we use the induction hypothesis for T'. Therefore, the algorithm **advancePosition** is also complete.

- Let the current operator be a union operator between the sub-trees T_1 and T_2. The implementations of both **advanceNode** and **advancePosition** get the next smallest (in lexicographic order) tuple from the input streams. By the induction hypothesis for T_1 and T_2, our algorithms trivially preserve the minimality, i.e. they are complete.

- Let the current operator be a set-difference operator between the sub-trees T_1 and T_2. The implementations of **advanceNode** gets the next smallest (in lexicographic order) tuple from the first input stream that is not in the second input stream. By the induction hypothesis for T_1, our algorithm trivially preserves the minimality. Therefore, it is complete.

The query evaluation complexity for the PPRED evaluation algorithm is given by:

$$O(\mathbf{entries\_per\_token} \times \mathbf{pos\_per\_entry} \times \mathbf{toks_Q} \times (\mathbf{preds_Q} + \mathbf{ops_Q} + 1))$$

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. Note how the complexity compares with the naive

Algorithm 5 PPRED Difference Evaluation Algorithm

```
Require: $inp_1, inp_2$ are API inputs to the difference operator;
Node advanceNode() {
repeat
    node = $inp_1$.advanceNode();
    if node == NULL then
        return NULL;
    end if

    while node $>$ $inp_2$.getNode() do
        $inp_2$.advanceNode();
    end while
until node $<$ $inp_2$.getNode()

return node; }

boolean advancePosition(i,pos) {
error(); //only node-level cursor movement is allowed }
```

approach in Section 5.1.2.

5.6 NPRED: Evaluation and Complexity

We now define the second class of full-text search predicates called *negative predicates*, which are designed to capture the negations of common full-text search predicates. For instance, *not-distance*, which is the negation of the *distance* predicate, is a negative predicate. Similarly, *not-ordered* and *not-samepara* are also negative predicates.

We show that even for this rich class of negative predicates, query evaluation can still be done in a linear scan over the positions in the inverted list, where the number of scans depends on the size of the query (and does not depend on the size of the data). However, unlike the case of positive predicates, query evaluation cannot always be done in a single scan of the positions. The extra scans are the price paid for negation. We note that the proposed algorithms can also support positive predicates in addition to negative predicates.

5.6.1 Negative Full-Text Predicates

Definition (Negative Predicates): An n-ary position-based predicate $pred$ is said to be a negative predicate iff

$$\begin{aligned}
&\forall p_1, ..., p_n \in \mathcal{P}\ \neg pred(p_1, ..., p_n) \Rightarrow \\
&\quad \exists i_1, ..., i_n\ p_{i_1} \leq ... \leq p_{i_n} \\
&\quad\quad \wedge \forall p'_{i_n} \in \mathcal{P}\ p_{i_1} \leq p'_{i_n} \leq p_{i_n} \Rightarrow \\
&\quad\quad\quad \forall p'_{i_1}, ..., p'_{i_{n-1}} \in \mathcal{P} \\
&\quad\quad\quad\quad Bounded(p'_{i_1}, ..., p'_{i_n}, p_{i_1}, ..., p_{i_n}) \\
&\quad\quad\quad\quad \Rightarrow \neg pred(p'_1, ..., p'_n) \text{ where} \\
&Bounded(p'_{i_1}, ..., p'_{i_n}, p_{i_1}, ..., p_{i_n}) \equiv \\
&\quad p_{i_1} \leq p'_{i_1} \leq p'_{i_2} \wedge p_{i_2} \leq p'_{i_2} \leq p'_{i_3} \wedge ... \\
&\quad \wedge p_{i_{n-1}} \leq p'_{i_{n-1}} \leq p'_{i_n}
\end{aligned}$$

Intuitively, the property says that if a negative predicate is false for a given set of positions ordered as $(p_{i_1} \leq ... \leq p_{i_n})$, then it is also false for every other set of positions $p'_{i_1}, ..., p'_{i_n}$ bounded by $p_{i_1} \leq ... \leq p_{i_n}$ (denoted through $Bounded(p'_{i_1}, ..., p'_{i_n}, p_{i_1}, ..., p_{i_n})$). A list of positions $p'_{i_1}, ..., p'_{i_n}$ is said to be bounded by another list of positions $p_{i_1} \leq ... \leq p_{i_n}$ if the ordering of the positions is preserved and each p'_{i_j} is bounded by (less than) its corresponding p_{i_j}. In other words, negative predicates can only be made true by *extending* the interval between the smallest and the largest positions. Note that for positive predicates, we needed to *contract* this interval.

Consider *not-distance*, the negation of *distance*, which returns true if the positions exceed a certain distance. *not-distance* is a negative predicate because it can only be made true by extending the window (distance). Similarly, the negation of other positive predicates such as *order* and *samepara*, referred to as *not-order* and *not-samepara*, are also negative predicates.

The NPRED language for negative predicates is similar to the PPRED language, except that it allows for both positive and negative predicates in the selection operators.

5.6.2 Query Evaluation Overview

The addition of negative predicates to the query language increases the complexity of query evaluation when compared to positive predicates. To see why this is the case, consider the query $\pi_{node}(\sigma_{not-distance(att_1,att_2,40)}(\mathtt{R}_{\mathtt{assignment}} \bowtie \mathtt{R}_{\mathtt{judge}}))$ (find nodes that contain tokens "assignment" and "judge" that are at least 40 positions apart). Now consider the inverted lists in Figure 2. The PPRED evaluation strategy (Section 5.5.3) of moving the smallest of the two positions p_1 and p_2 does not work in this case because the distance between p_1 and p_2 may never grow (recall that we want the distance to exceed 40, and moving the smallest position may always keep the positions withing 40 tokens of each other).

Instead of the PPRED evaluation strategy of moving the smallest position for *distance*, for *not-distance*, we wish to fix one position and move the other one until the predicate is satisfied. But which of p_1 or p_2 do we fix and which one do we move? Obviously, we have to try both alternatives because both alternatives (we do not know a priori which one) could lead to valid solutions: (100, 34) and (50, 97). Consequently, instead of scanning the inverted lists just once, we may have to scan them *as many times as the arity of the negative predicate* (in this case, twice). For each scan, we fix a partial order among the cursors: the positions pointed by the cursors must be ordered as specified by the partial order. Later if we need to evaluate another negative predicate, we may either use the existing partial order or extend it if the order is not sufficient (i.e., it does not specify the order between a couple of cursors). Since multiple partial orders enforce a total order in the worst-case, we may have to scan the inverted list position up to $toks\_Q!$ times, where $toks\_Q$ is the number of query tokens.

Below we present an algorithm for evaluating NPRED queries. It resolves the non-determinism outlined in the previous paragraph by running $toks\_Q!$ threads of the evaluation algorithm, where n is the number of search tokens. Each thread uses an *ordering permutation* $i_1, ..., i_n$ of $\{1, ..., n\}$. The latter specifies an ordering of the cursors over the query token inverted lists. If $p_1, ..., p_n$ are the current positions, then the invariant is that $p_{i_1} \leq ... \leq p_{i_n}$. Thus, when trying to satisfy a negative predicate, the algorithm moves always the iterator over the inverted list that points to the largest position.

It must be noted that the presented algorithm is not the most efficient. As we discussed above, we need orderings only among cursors that are used in negative predicates, i.e., we need a partial order among these cursors. On the other hand, the ordering permutation imposes a total order which is needed only if all positions are used in negative predicates. We chose to present this less efficient algorithm because it demonstrates the main points of the query evaluation while keeping the presentation simple. Our implementation generates only the necessary partial orders.

Algorithm 6 NPRED Join Evaluation Algorithm

```
Require: $inp1, inp2$ are the two API inputs to the join, and have $c_1$ and $c_2$ position columns, respectively; $i_1, ..., i_n$ specifies a
    permutation of the position columns
 1: boolean advancePosition(index,pos) {
 2: repeat
 3:   if $index < c_1$ then
 4:     result = inp1.advancePosition(index,pos);
 5:     if result then
 6:       $p_{index}$ = inp1.getPosition(index);
 7:     end if
 8:   else
 9:     result = inp2.advancePosition(index-$c_1$,pos);
10:     if result then
11:       $p_{index}$ = inp2.getPosition(index-$c_1$);
12:     end if
13:   end if
14:   if $result$ then
15:     $k = \{j \mid i_j = index\}$
16:     violated = ($p_{index} < p_{i_{j+1}}$)
17:   end if
18:   if violated then
19:     pos = $p_{index}$
20:     $index$++
21:   end if
22: until !$result$ || !$violated$
23: return result; }
```

5.6.3 NPRED Query Evaluation Algorithms

The query evaluation algorithm for NPRED is similar to PPRED with two exceptions: (1) each query evaluation thread is associated with a unique total order of query inverted list positions, and (2) the NPRED selection operators in a given thread only move the cursor that corresponds to the largest position in the total order associated with that thread. We only describe the join algorithm and the predicate evaluation algorithm; the other operator algorithms are only minor modifications of the corresponding PPRED evaluation algorithm to take cursor ordering into account.

Algorithm 6 presents the join algorithm for NPRED. It is based on the same evaluation interface as the one defined for PPRED in Section 5.5. The **advanceNode** method is identical to the PPRED case and is omitted. The **advancePosition** method is also similar to the one used for PPRED but it also ensures that the positions are always in the order specified by the permutation $i_1, ..., i_n$.

Algorithm 7 presents the predicate evaluation algorithm for NPRED. It differs from Algorithm 2 only in the **advancePosUntilSat** method which, unlike for positive predicates, moves the cursor pointing to the largest position to "extend" the gap between positions.

5.6.4 Correctness and complexity

Intuitively, the proof for correctness of the above algorithms is similar to the one for PPRED. Again, we have two parts: soundness and completeness. The soundness can be proven per evaluation thread and the proof is analogous to the soundness proof for PPRED. The difference is just in the join algorithm where lines 15-21 ensure that the ordering among inverted-lists cursors is preserved. It is not hard to see that in the case of **advancePosition**, we need to check whether the order is violated only for the position that has been

Algorithm 7 NPRED Predicate Evaluation Algorithm

Require: inp is API inputs to the predicate with c position columns; $i_1, ..., i_n$ specifies a permutation of all position columns

```
1: boolean advancePosUntilSat () {
2: while !$pred(p_1, ..., p_c)$ do
3:     index = Max $\{j \mid p_{i_j}$ is one of $p_1, ..., p_c\}$
4:     success = inp.advancePos($index, f_i(p_1, ..., p_c)$);
5:     if ! success then
6:         return false;
7:     end if
8: end while
9: return true; }
```

moved (lines 14-21). Indeed, the order for the other positions is guaranteed to be correct by the induction hypothesis. As before, **advancePosUntilSat** loops until it finds a satisfying position-variable assignment.

For the completeness part, we will consider only the presented algorithms that have a non-trivial difference to their PPRED counterparts. Intuitively, we need to prove the minimality of the found solution only on a per-thread basis. If every thread finds a minimal solution, then minimal solution among all thread solutions is the global minimal solution. The latter holds because we have a thread for every possible ordering of the positions in a solution.

Thus, we need to show that given a thread, the algorithms for the join and selection operators preserve the minimality.

- Let the current operator be a selection operator for the positive predicate $pred(p_{i_1}, ..., p_{i_m}, c_1, ..., c_q)$ with a sub-tree T'. The soundness of the algorithms guarantees the $(node, p_1, ..., p_n \in R$ part of the properties. We will show minimality. As before, we will focus on the **advancePosUntil** algorithm. If it finds a minimal solution, the completeness of both **advanceNode** and **advancePosition** trivially follows. Now, line 2 ensures that for the current set of positions $(p_1, ..., p_n)$, $pred(p_{i_1}, ..., p_{i_m}, c_1, ..., c_q) = false$. First, observe that the property of negative predicates guarantees that for the minimal solution $(p'_1, ..., p'_n)$, we have $p_{i_m} < p'_{i_m}$. It follows from the induction hypothesis, that the solution $(p'_1, ..., p'_n)$ is the minimal solution for T' such that $p''_{i_m} > p_{i_m}$. Then, the loop in lines 2-8 guarantees that we stop moving i_m-th cursor once we find the first (minimal) set of positions that is a solution to T' and satisfes the predicate. Thus, we have minimality.

- Let the current operator be a join operator between te sub-trees T_1 and T_2. The difference from the PPRED case stems from the fact that we need to ensure that the ordering permutation is satisfied by the current set of positions. It can be seen from lines 14-21 that in case of an order violation, it is resolved in an order of increasing positions. Therefore, given the induction hypothesis, the join algorithm always finds the minimal set of positions that satisfy the ordering permutation.

The query evaluation complexity for NPRED is similar to PPRED, except that there are $toks_Q$! different evaluation threads. Thus, the resulting complexity is:

$O(\mathbf{entries\_per\_tok} \times \mathbf{pos\_per\_entry} \times \mathbf{toks} \times \mathbf{toks_Q}! \times (\mathbf{preds_Q} + \mathbf{ops_Q} + 1))$

The scoring method presented in Section 5.5 can be directly applied for NPRED. As before, the computation of scores can be done in constant time and does not affect the complexity of the query evaluation algorithm.

6 Experiments

We performed experiments on both real and synthetic data sets. Due to lack of space, we report our experiments on real data and only mention similar results on synthetic data. The goals of our experiments are (1) to compare the performance of the evaluation algorithms in Section 5 and study the trade-offs between language expressiveness and complexity and, (2) to study the effect of the query parameters listed below on each algorithm.

- $\mathbf{tok_Q}$: The number of tokens in Q, including string literals and the universal token ANY.
- $\mathbf{pred_Q}$: The number of predicates in Q.
- $\mathbf{op_Q}$: The number of operations in Q, where an operation can be NOT, AND, OR

6.1 Summary of Results

Our results validate the complexity study presented in Section 5. We show that we can order our algorithms by performance: BOOL $\preceq$ PPRED $\preceq$ NPRED $\preceq$ COMP. This was expected given the expressibility of the languages. On the other hand, the interesting fact is that PPRED achieves greater expressibility (the ability to evaluate predicates) than BOOL at a marginally larger cost. On average, PPRED performs better than NPRED for positive predicates due to the fact that PPRED does not need to generate all permutations of the inverted lists. In general, NPRED has noticeably better performance than COMP for both positive and negative predicates. We also observe that in practice, our algorithms perform better than their worst case complexity. In particular, COMP might find a solution early and hence, avoid performing a Cartesian product which explains that sometimes COMP is not much worse than NPRED. Our experiments also show that all of the algorithms perform very similarly when queries do not contain predicates.

6.2 Experimental Setup

We implemented the algorithms for BOOL, PPRED, NPRED, and COMP in C++. The evaluation algorithm for BOOL follows the method outlined in the example in Section 5.3. The algorithm for COMP converts the query to an FTA expression and evaluates the latter as in the relational algebra. We ran our experiments on a AMD64 3000+ computer with 1GB RAM and one 200GB SATA drive, running under Linux 2.6.9.

To quantify the size of the scanned inverted lists, we use the following parameters ($\mathcal{T}$ denotes all tokens that appear in the context nodes $\mathcal{N}$).

- $\mathbf{cnodes}$: $|\mathcal{N}|$ (the number of context nodes).
- $\mathbf{pos\_per\_cnode}$: $max_{(cn,PosList)\in IL_{ANY}}(|PosList|)$ (the maximum number of positions in a node).
- $\mathbf{entries\_per\_token}$: $max_{tok\in\mathcal{T}}(|\{e|e \in IL_{tok}\}|)$ (the maximum number of entries in a token inverted list).
- $\mathbf{pos\_per\_entry}$: $max_{tok\in\mathcal{T}}max_{(cn,PosList)\in IL_{tok}}$ $(|PosList|)$ (the maximum number of positions in an entry in a token inverted list).

We present the experiments for the effects of the $\mathbf{tok_Q}$, $\mathbf{pred_Q}$, $\mathbf{cnodes}$, and $\mathbf{pos\_per\_entry}$ query parameters given above. The experiments on the other parameters supported the conclusions from the summary above and are omitted in the interest of space. To test the influence of each parameter on query evaluation performance, we fixed the other parameters to their default values and varied the values of the studied

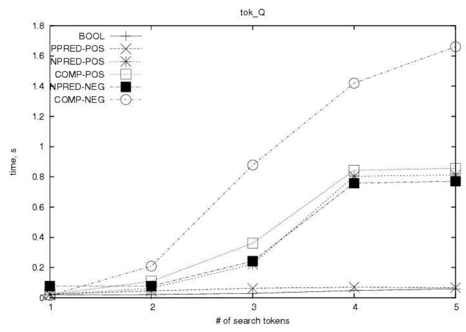

Figure 5: Varying Number of Query Tokens (INEX)

parameter. In particular, we used: queries with 1 to 5 query tokens (default 3) to test $\mathbf{tok_Q}$; queries with 0 to 4 predicates (default 2) to test $\mathbf{pred_Q}$; 2500, 6000 (default), and 10000 context nodes to test $\mathbf{cnodes}$; query tokens with at most 5, 25 (default), and 125 positions per inverted list entry to test $\mathbf{pos\_per\_entry}$.

In order to understand the comparative behavior of our algorithms, we plot them all on the same graph. Each algorithm is run with a different query. While the labels `BOOL`, `PPRED`, `NPRED` and `COMP` represent each algorithm, `PPRED-POS`, `NPRED-POS` and `COMP-POS` (resp., `NPRED-NEG` and `COMP-NEG`) report queries with positive predicates only (resp., negative predicate only) for each algorithm. Since the performance of our algorithms is similar when queries have no predicates, we only report `BOOL` for such queries.

6.3 Data, Queries and Results

We used the INEX 2003 XML document collection dataset [1] which is 500MB large with a little over 12000 documents that contain articles from 17 IEEE journals between 1997 and 2001. Since we are interested in full-text search, we ignored the XML structure and indexed the documents as flat.

Figure 5 shows the performance of our algorithms when varying the number of tokens in the input query and keeping the input data fixed. Figure 6 shows the performance of our algorithms when varying the number of predicates in the input query and keeping the input data fixed. Both experiments show that `BOOL` and `PPRED` grow slowly linearly in each of the query size parameters, while `COMP` and `NPRED` grow exponentially, the former is faster. Both figures show that `PPRED` can achieve greater expressibility than `BOOL` at marginally worse performance.

The big difference in the evaluation time for positive and negative predicates can be explained with the difference in the selectivity of negative predicates: it is higher than the selectivity of the positive predicates. In fact, we used the negation of the positive predicates to generate the negative predicates queries. This explains why the performance of `COMP-NEG` is bad: large selectivity means it needs to scan many tuples to find a solution. In this case, `NPRED` is better than `COMP` for the same queries because it does a more intelligent scan of the inverted lists. It ”searches” for the solution, while `COMP` just blindly enumerates the entire join. The pruning that does `NPRED` decreases significantly the influence of selectivity.

Although not reported, our experiments on synthetically generated data had similar results when varying the number of tokens and the number of predicates in queries.

Figure 7 shows the performance of our algorithms when varying the number of context nodes. As it can be seen, `PPRED` and `BOOL` offer the best scalability: slow linear decrease in performance. The scalability of

[1]http://www.is.informatik.uni-duisburg.de/projects/inex03/

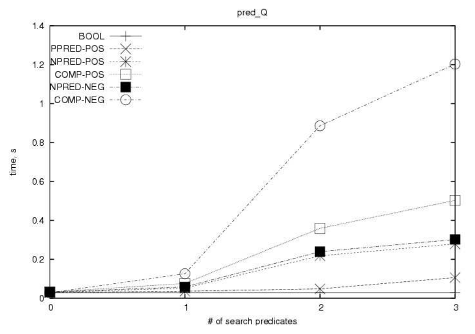

Figure 6: Varying Number of Query Predicates (INEX)

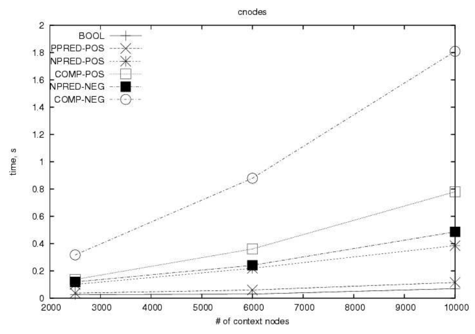

Figure 7: Varying Number Context Nodes (INEX)

NPRED is acceptable (the evaluation time increases linearly in the size of the database) while COMP does not scale very well – exponentially. The results for the scalability when we increase the number of positions per inverted list entry (Figure 8) show similar results. This directly influences the size of the join of the inverted lists, thus increasing the number of potential results. Again, PPRED and BOOL are the best, but NPRED also displays only a small increase.

7 Related Work

Most IR research [3][31] has focused on methods for relevance estimation and efficient evaluation of keyword queries. In this context, full-text languages have been developed to implement specific primitives, but their formal properties have not been studied. This observation also applies to XML full-text search languages such as XQuery/IR [6], XIRQL [18], XSEarch [15], XRANK [21], XXL [33] and Niagara [37]. In fact, we can represent these languages (see Sections 4.1 and 4.2). Several other works have used relational systems to store inverted lists and translate keyword queries to SQL [11, 17, 22, 29, 30, 37] but they do not study completeness.

Clarke et al. [5] propose a formal model for full-text search with some leverage of structure such as chapters and paragraphs. The model is based on intervals of positions and supports a fixed set of predicates *(not) containing, (not) contained in, one of/both of, followed by*. Thus, this model is less general than ours and it may be hard to extend it because it is based on intervals of positions. This coarser granularity inherently looses some information since not all positions in an interval may be relevant to the query. No study of expressiveness is provided.

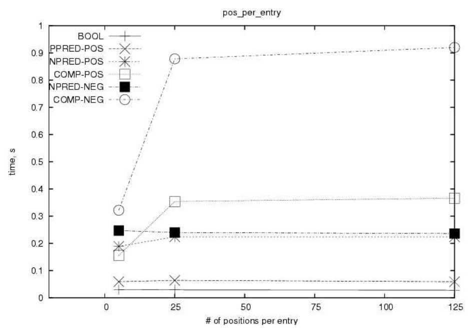

Figure 8: Varying Number of Positions Per Inverted List Entry (INEX)

8 Conclusion

We presented a simple, yet powerful formalization of full-text search languages as a basis for studying expressiveness and efficiency. We believe that this work is an important first step for full-text search much like the relational model laid the foundation of extensive database research. We are planning to add new full-text primitives such as stemming, thesaurus and stop-words. We would also like to explore how our formalization in terms of the relational model enables the joint optimization of structured and full-text queries. Finally, we want to study the complexity implications of scoring and top-k techniques [10, 16, 25, 32].

References

[1] S. Amer-Yahia, C. Botev, J. Robie, J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. http:/www.cs.cornell.edu/database/TeXQuery/.

[2] S. Amer-Yahia, C. Botev, J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. WWW 2004.

[3] R. Baeza-Yates, B. Ribiero-Neto. Modern Information Retrieval. Addison-Wesley, 1999.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. ICDE 2002.

[5] 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)

[6] J. M. Bremer, M. Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. WebDB 2002.

[7] E. W. Brown. Fast Evaluation of Structured Queries for Information Retrieval. SIGIR 1995.

[8] A. Chandra, P. Merlin Optimal Implementation of Conjunctive Queries in Relational Databases. STOC 1977.

[9] A. Chandra, D. Harel. Structure and Complexity of Relational Queries. FOCS 1980.

[10] S. Chaudhuri, L. Gravano. Evaluating Top-k Selection Queries. In Proc. of 25th VLDB Conf.

[11] T. T. Chinenyanga, N. Kushmerick. Expressive and Efficient Ranked Querying of XML Data. WebDB 2001.

[12] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13(6): 377-387 (1970).

[13] E.F. Codd. Relational Completeness of Database Sublanguages. In R. Rustin (ed.), Database Systems, Prentice-Hall, 1972.

[14] W.W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. SIGMOD 1998.

[15] S. Cohen et al. XSEarch: A Semantic Search Engine for XML. VLDB 2003.

[16] R. Fagin, R. Kumar, D. Sivakumar. Efficient Similarity Search and Classification Via Rank Aggregation. SIGMOD 2003.

[17] D. Florescu, D. Kossmann, I. Manolescu. Integrating Keyword Search into XML Query Processing. WWW 2000.

[18] N. Fuhr, K. Grossjohann. XIRQL: An Extension of XQL for Information Retrieval. SIGIR 2000.

[19] N. Fuhr, T. Rölleke. A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. ACM TOIS 15(1), 1997.

[20] Y. Hayashi, J. Tomita, G. Kikui. Searching Text-rich XML Documents with Relevance Ranking. SIGIR Workshop on XML and Information Retrieval, 2000.

[21] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003.

[22] V. Hristidis, L. Gravano, Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. VLDB 2003.

[23] N. Immerman. Relational Queries Computable in Polynomial Time Information and Control. 68(1-3): 86-104 (1986).

[24] Initiative for the Evaluation of XML Retrieval. http://www.is.informatik.uni-duisburg.de/projects/inex03/.

[25] C. Li, K. Chang, I. Ilyas, S. Song. Query Algebra and Optimization for Relational Top-k Queries. SIGMOD 2005

[26] Library of Congress. http://lcweb.loc.gov/crsinfo/xml/.

[27] S.-H. Myaeng, D.-H. Jang, M.-S. Kim, Z.-C. Zhoo. A Flexible Model for Retrieval of SGML Documents. SIGIR 1998.

[28] G. Salton. Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer. Addison Wesley, 1989.

[29] J. Melton, A. Eisenberg. SQL Multimedia and Application Packages (SQL/MM). SIGMOD Record 30(4), 2001.

[30] A. Salminen. A Relational Model for Unstructured Documents. SIGIR 1987.

[31] G. Salton, M. J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, 1983.

[32] M. Theobald, G. Weikum, R. Schenkel. Top-k Query Evaluation with probabilistic guarantees. In Proc. of VLDB 2004.

[33] A. Theobald, G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. EDBT 2002.

[34] The World Wide Web Consortium. XQuery 1.0: An XML Query Language. W3C Working Draft. http://www.w3.org/TR/xquery/.

[35] The World Wide Web Consortium. XQuery and XPath Full-Text Use Cases. W3C Working Draft. http://www.w3.org/TR/xmlquery-full-text-use-cases/.

[36] M. Vardi. The Complexity of Relational Query Languages. STOC 1982.

[37] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD 2001.

[38] E. Zimanyi. Query Evaluations in Probabilistic Relational Databases. Theoretical Computer Science, 1997.

A Proofs of Theorems

Theorem 1: Equivalence of the Full-Text Calculus and Algebra

Lemma 1. For every full-text algebra expression that only uses position-based predicates from the set $Preds$, there exists an equivalent full-text calculus expression that only uses position-based predicates from the same set $Preds$.

Proof Sketch: We will prove that for every algebra expression $AlgExpr$, which evaluates to a relation $R(CNode, att_1, att_2, ..., att_k)$, $k \geq 0$, there exists a calculus query expression $CalcExpr(n, p_1, ..., p_k)$ with free variables $\{n, p_1, ..., p_k\}$, such that $\{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge hasPos(n, p_1) \wedge ... \wedge hasPos(n, p_k) \wedge CalcExpr(n, p_1, ..., p_k)\} = R$.

The proof is by induction on the structure of $AlgExpr$.

- If $AlgExpr = \texttt{SearchContext}$, then $CalcExpr(n) = (\exists p\ hasPos(n,p) \wedge hasPos(n,p)) \vee \neg(\exists p\ hasPos(n,p) \wedge hasPos(n,p))$. $CalcExpr(n)$ is always true; therefore, $\{n \mid SearchContext(n) \wedge CalcExpr(n)\}$ is equal to the full-text relation $SearchContext$ by its definition.

- If $AlgExpr = \texttt{HasPos}$, then $CalcExpr(n, p_1) = hasPos(n, p_1)$, i.e. $\{(n, p_1) \mid SearchContext(n) \wedge hasPos(n, p_1)\}$ is equal to the full-text relation $HasPos$ by its definition.

- If $AlgExpr = \texttt{R}_{\texttt{token}}$, then $CalcExpr(n) = hasToken(p,'token')$. $\{(n, p_1) \mid SearchContext(n) \wedge hasPos(n, p_1) \wedge hasToken(p_1,'token')\}$ is equal to the full-text relation R_{token} by its definition.

- If $AlgExpr = \pi_{\texttt{CNode,att}_1\texttt{,...,att}_\texttt{i}}(\texttt{AlgExpr}')$, where $AlgExpr'$ is a full-text algebra expression whose equivalent calculus query expression is $CalcExpr'(n, p_1, ..., p_m)$ and $AlgExpr'$ evaluates to $R'(CNode, att_1, att_2, ..., att_m)$, $m \geq i$, then $CalcExpr(n, p_1, ..., p_i) = \exists p_{i+1}\ hasPos(n, p_{i+1}) \wedge ... \exists p_m\ hasPos(n, p_m) \wedge CalcExpr'(n, p_1, ..., p_m)$. $\{(n, p_1, ..., p_i) \mid SearchContext(n) \wedge \bigwedge_{j=1,...,i} hasPos(n, p_j) \wedge CalcExpr(n, p_1, ..., p_i)\} = \pi_{CNode,p_1,...,p_i}\{(n, p_1, ..., p_m) \mid SearchContext(n) \wedge \bigwedge_{j=1,...,m} hasPos(n, p_j) \wedge CalcExpr'(n, p_1, ..., p_m)\} = \pi_{CNode,p_1,...,p_i}(R')$.

- If $AlgExpr = \texttt{AlgExpr}_1 \bowtie \texttt{AlgExpr}_2$, where $AlgExpr_1$ and $AlgExpr_2$ are full-text algebra expressions that evaluate to $R_i(CNode, att_1, ..., att_{m_i})$ for $i = 1, 2$, and their equivalent calculus query expressions are $CalcExpr_i(n, p_1, ..., p_{m_i})$ for $i = 1, 2$,then $CalcExpr(n, p_1, ..., ..., p_{m_1+m_2}) = CalcExpr(n, p_1, ..., p_{m_1}) \wedge CalcExpr(n, p_{m_1+1}, ..., p_{m_1+m_2})$. $\{(n, p_1, ..., p_{m_1+m_2} \mid SearchContext(n) \wedge \bigwedge_{j=1,...,m_1+m_2} hasPos(n, p_j) \wedge CalcExpr(n, p_1, ..., p_{m_1}) \wedge CalcExpr(n, p_{m_1+1}, ..., p_{m_1+m_2}\} = R_1 \bowtie R_2$.

- If $AlgExpr = \sigma_{\texttt{pred(att}_1\texttt{,...,att}_\texttt{m}\texttt{,c}_1\texttt{,...,c}_\texttt{q})}(\texttt{Expr}')$, where $Expr'$ is a full-text algebra expression that evaluates to the relaton R' and the equivalent calculus query expression is $CalcExpr'(n, p_1, ..., p_k)$, then $CalcExpr(n, p_1, ..., p_k) = CalcExpr'(n, p_1, ..., p_k) \wedge pred(att_1, ..., att_m, c_1, ..., c_q)$. $\{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge \bigwedge_{j=1,...,k} hasPos(n, p_j) \wedge CalcExpr'(n, p_1, ..., p_k) \wedge pred(att_1, ..., att_m, c_1, ..., c_q)\} = \sigma_{pred(att_1,...,att_m,c_1,...,c_q)} R'$.

- Let $AlgExpr = \texttt{AlgExpr}_1 \cup \texttt{AlgExpr}_2$, where $AlgExpr_1$ and $AlgExpr_2$ are full-text algebra expressions that evaluate to R_i for $i = 1, 2$ and their equivalent calculus query expressions are $CalcExpr_i(n, p_1, ..., p_k)$ for $i = 1, 2$, then $CalcExpr(n, p_1, ...p_k) = CalcExpr_1(n, p_1, ...p_k) \vee CalcExpr_2(n, p_1, ...p_k)$. $\{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge \bigwedge_{j=1,...,k} hasPos(n, p_j) \wedge (CalcExpr_1(n, p_1, ...p_k) \vee CalcExpr_2(n, p_1, ...p_k))\} = R_1 \cup R_2$.

- Let $AlgExpr = \texttt{AlgExpr}_1 \cap \texttt{AlgExpr}_2$, where $AlgExpr_1$ and $AlgExpr_2$ are full-text algebra expressions that evaluate to R_i for $i = 1, 2$ and their equivalent calculus query expressions are $CalcExpr_i(n, p_1, ..., p_k)$ for $i = 1, 2$, then $CalcExpr(n, p_1, ...p_k) = CalcExpr_1(n, p_1, ...p_k) \wedge CalcExpr_2(n, p_1, ...p_k)$. $\{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge \bigwedge_{j=1,...,k} hasPos(n, p_j) \wedge(CalcExpr_1(n, p_1, ...p_k) \wedge CalcExpr_2(n, p_1, ...p_k))\} = R_1 \cap R_2$.

- Let $AlgExpr = \texttt{AlgExpr}_1 - \texttt{AlgExpr}_2$, where $AlgExpr_1$ and $AlgExpr_2$ are full-text algebra expressions that evaluate to R_i for $i = 1, 2$ and their equivalent calculus query expressions are $CalcExpr_i(n, p_1, ..., p_k)$ for $i = 1, 2$, then $CalcExpr(n, p_1, ...p_k) = CalcExpr_1(n, p_1, ...p_k) \wedge \neg CalcExpr_2(n, p_1, ...p_k)$. $\{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge \bigwedge_{j=1,...,k} hasPos(n, p_j) \wedge (CalcExpr_1(n, p_1, ...p_k) \wedge \neg CalcExpr_2(n, p_1, ...p_k))\} = R_1 - R_2$.

This completes the structural induction. The requirement that full- text algebra queries evaluate to a relation with a single $CNode$ attribute ensures that the corresponding $CalcExpr$ expression will have only one free variable - n. Therefore, $\{n|SearchContext(n) \wedge CalcExpr(n)\}$ is a valid calculus query. □

Lemma 2. For every full-text calculus expression that only uses position-based predicates from the set $Preds$, there exists an equivalent full-text algebra expression that only uses position-based predicates from the same set $Preds$.
Proof Sketch: We will prove that for every query calculus expression $CalcExpr(n, p_1, ..., p_k)$ with free variables $\{n, p_1, ..., p_k\}$, $k \geq 0$, there exists an algebra expression $AlgExpr$, which evaluates to a relation $R(CNode, att_1, att_2, ..., att_k)$, such that $\{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge \bigwedge_{j=1,..,k} hasPos(n, p_j) \wedge CalcExpr(n, p_1, ..., p_k)\} = R$.

The proof is by induction on the structure of $CalcExpr$.

- If $CalcExpr(n, p) = hasPos(n, p)$, then $AlgExpr = \texttt{HasPos}$. The proof of the equivalence is the same as the analogous case from Lemma 1.

- If $CalcExpr(n, p) = hasToken(p, 'token')$, then $AlgExpr = \texttt{R}_{\texttt{token}}$. The proof of the equivalence is the same as the analogous case from Lemma 1.

- If $CalcExpr(n, p_1, ..., p_k) = pred(p_1, ..., p_k, c_1, ..., c_q)$, then $AlgExpr = \sigma_{\texttt{pred}(\texttt{p}_1,...,\texttt{p}_k,\texttt{c}_1,...,\texttt{c}_q)}$ $(\texttt{HasPos} \bowtie ... \bowtie \texttt{HasPos})$, where the number of joins is k. Obviously, $R = \{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge \bigwedge_{i=1,...,k} hasPos(n, p_i) \wedge pred(p_1, ..., p_k, c_1, ..., c_q)\}$.

- If $CalcExpr(n, p_1, ..., p_l, q'_1, ..., q'_m, q''_1, ..., q''_c) = CalcExpr_1(n, p_1, ..., p_l, q'_1, ..., q'_m) \wedge CalcExpr_2(n, p_1, ..., p_l, q''_1, ..., q''_c)$, where $k = l + m + c$, $CalcExpr_1$, and $CalcExpr_2$ are calculus query expressions with equivalent algebra expressions $AlgExpr_1$ and $AlgExpr_2$, which evaluate to $R_1(CNode, att_1, ..., att_k, att'_1, ..., att'_m)$ and $R_2(CNode, att_1, ..., att_l, att''_1, ..., att''_c)$, then $AlgExpr = (\texttt{AlgExpr}_1 \bowtie \pi_{\texttt{CNode},\texttt{q}''_1,...,\texttt{q}''_c} \texttt{AlgExpr}_2) \cap (\pi_{\texttt{CNode},\texttt{q}'_1,...,\texttt{q}'_m} \texttt{AlgExpr}_1 \bowtie \texttt{AlgExpr}_2)$. $R = \{(n, p_1, ..., p_l, q'_1, ..., q'_m, q''_1, ..., q''_c) \mid (n, p_1, ..., p_l, q'_1, ..., q'_m) \in R_1 \wedge (n, p_1, ..., p_l, q''_1, ..., q''_c) \in R_2\}$ = $\{(n, p_1, ..., p_l, q'_1, ..., q'_m, q''_1, ..., q''_c) \mid CalcExpr_1(n, p_1, ..., p_l, q'_1, ..., q'_m) \wedge CalcExpr_2(n, p_1, ..., p_l, q''_1, ..., q''_c)\}$, which is exactly what we wnated to show.

- If $CalcExpr(n, p_1, ..., p_l, q'_1, ..., q'_m, q''_1, ..., q''_c)$ = $CalcExpr_1(n, p_1, ..., p_l, q'_1, ..., q'_m) \vee CalcExpr_2(n, p_1, ..., p_l, q''_1, ..., q''_c)$, where $k = l + m + c$, $CalcExpr_1$, and $CalcExpr_2$ are calculus query expressions with equivalent algebra expressions $AlgExpr_1$ and $AlgExpr_2$, which evaluate to $R_1(CNode, att_1, ..., att_k, att'_1, ..., att'_m)$ and $R_2(CNode, att_1, ..., att_l, att''_1, ..., att''_c)$, then

$AlgExpr = (\texttt{AlgExpr}_1 \bowtie \pi_{\texttt{CNode},q''_1,...,q''_c}\texttt{AlgExpr}_2) \cup (\pi_{\texttt{CNode},q'_1,...,q'_m}\texttt{AlgExpr}_1 \bowtie \texttt{AlgExpr}_2)$. $R = \{(n, p_1, ..., p_l, q'_1, ..., q'_m, q''_1, ..., q''_c) \mid (n, p_1, ..., p_l, q'_1, ..., q'_m) \in R_1 \vee (n, p_1, ..., p_l, q''_1, ..., q''_c) \in R_2\}$ = $\{(n, p_1, ..., p_l, q'_1, ..., q'_m, q''_1, ..., q''_c) \mid CalcExpr_1(n, p_1, ..., p_l, q'_1, ..., q'_m) \vee$ $CalcExpr_2(n, p_1, ..., p_l, q''_1, ..., q''_c)\}$, which is what we wanted to show.

- Let us consirder the case $CalcExpr(n, p_1, ..., p_k) = \neg CalcExpr'(n, p_1, ..., p_k)$, where $CalcExpr'$ is a calculus query expression that is equivalent to the algebra expression $AlgExpr'$, which evaluates to $R'(n, p_1, ..., p_k)$. If $k > 0$, then $AlgExpr = (\texttt{HasPos} \bowtie ... \bowtie \texttt{HasPos}) - AlgExpr'$, where the number of joins is k. $R = \{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge \bigwedge_{i=1,...,k} hasPos(n, p_i) \wedge \neg CalcExpr'(n, p_1, ..., p_k)\}$, which is what we wanted to show.

  If $k = 0$, then $AlgExpr = \texttt{SearchContext} - \texttt{AlgExpr}'$ and $R = \{n \mid SearchContext(n) \wedge SearchContext(n) \wedge \neg CalcExpr'(n)\}$

- If $CalcExpr(n, p_1, ..., p_k) = \exists p_{k+1}\ hasNode(n, p_{k+1}) \wedge CalcExpr'(n, p_1, ..., p_{k+1})$, where $CalcExpr'$ is a calculus query expression that is equivalent to the algebra expression $AlgExpr'$, which evaluates to R', then $AlgExpr = \pi_{\texttt{CNode},p_1,...,p_k}\texttt{R}'$ and $R = \{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge \exists p_{k+1}\ (n, p_1, ..., p_{k+1}) \in R'\}$ = $\{(n, p_1, ..., p_k) \mid SearchContext(n) \wedge \exists p_{k+1}\ CalcExpr'(n, p_1, ..., p_{k+1})\}$.

- Let $CalcExpr(n, p_1, ..., p_k) = \forall p_{k+1}\ hasNode(n, p_{k+1}) \Rightarrow CalcExpr'(n, p_1, ..., p_{k+1})$, where $CalcExpr'$ is a calculus query expression that is equivalent to the algebra expression $AlgExpr'$. We use the equation $CalcExpr(n, p_1, ..., p_k) = \neg\exists p_{k+1}\ \neg CalcExpr'(n, p_1, ..., p_{k+1})$ and apply the previous case..

For every calculus query, its query expression has only one free variable, n, therefore the equivalent algebra query evaluates to a relation that contains a single column, $CNode$. Therfore, it is a valid algebra query. $\square$

The above two Leammas prove the equivalence of the full-text calculus and algebra.

Theorem 4: Completeness of BOOL when $\mathcal{T}$ is finite

Proof Sketch: Let $F = \{n \mid SearchContext(n) \wedge P(n)\}$ be a calculus query expression. We will prove that there exists an equivalent *Query* expression E in BOOL. Without loss of generality, we assume that every quantified variable in F has a unique name. Let these position variable names be $p_1, p_2, ..., p_m$.

We first normalize $P(n)$ using the sequence of equivalence transformations presented below.

1. *(Sink Negations)* Move all negations down to the predicates $hasPos(n, p_i)$ and $hasToken(p_i, t)$. Replace any repetitive negations $\neg\neg A$ with A. Invert quantifiers: $\neg\exists p\ hasPos(n, p) \wedge A$ is replaced with $\forall p\ hasPos(n, p) \Rightarrow \neg A$ and $\neg\forall p\ hasPos(n, p) \Rightarrow A$ is replaced with $\exists p\ hasPos(n, p) \wedge \neg A$.

2. *(Group)* Move every expression of the form $hasToken(p_i, t)$ and $\neg hasToken(p_i, t)$ out of the scope of any quantifier over a variable different from p_i. This is possible because $hasToken$ is applied on only one position variable. Formally, the transformation is a repeated application of $\mathcal{Q}p_j\ A \circ B \mapsto B \circ \mathcal{Q}p_j\ A$ where $\mathcal{Q} \in \{\exists, \forall\}$, $\circ \in \{\wedge, \vee\}$, and B has no free variable p_j. Use the commutativity of $\wedge$ and $\vee$ to group the above predicate expressions next to each other and right after $hasPos(n, p_i)$. We get a propositional formula with propositions of the form $\mathcal{Q}_i p_i\ A_i(n, p_i)$ where $\mathcal{Q}_i \in \{\exists, \forall\}$.

3. *(Remove universal quantification)* Remove any universal quantifers by replacing $\forall p_i\ hasPos(n, p_i) \Rightarrow X$ with $\neg\exists p_i\ hasPos(n, p_i) \wedge \neg\ X$. We get a propositional formula over propositions of the form $\exists p_i\ hasPos(n, p_i) \wedge B_i(n, p_i)$.

4. *(Local DNF)* Convert each $B_i(n, p_i)$ to DNF.

5. *(Split)* Replace $\exists p \;\; hasPos(n,p) \wedge (X(n,p) \vee Y(n,p))$ with $(\exists p' \;\; hasPos(n,p') \wedge X(n,p')) \vee (\exists p'' \;\; hasPos(n,p'') \wedge Y(n,p''))$ to $\exists p_i \;\; hasPos(n,p_i) \wedge B_i(n,p_i)$ for every disjunct in $B_i(n,p_i)$. Let the new position variables be $q_1, ..., q_k$. We get a propositional formula over propositions of the form $\exists q_j \; hasPos(n,q_j) \wedge C_j(n,q_j)$ where C_j is a conjunction.

6. *(Global DNF)* Consider F to be a propositional formula over propositions of the form $\exists q_j \; hasPos(n,q_j) \wedge C_j(n,q_j)$. Convert it to a DNF.

We define $QE(F)$ for a calculus query expression F as the equivalent query in BOOL.

We observe that after the normalization $F = \{n \mid SearchContext(n) \wedge (\bigvee_i \bigwedge_j D_{i,j})\} = \bigcup_i \bigcap_j \{n \mid SearchContext(n) \wedge D_{i,j}\}$, where each $D_{i,j}$ is either of the form $\exists q \;\; hasPos(n,p) \wedge C(n,q)$ or of the form $\neg\exists q \;\; hasPos(n,p) \wedge C(n,q)$, as in step ***GlobalDNF*** from the normalization. Therefore, F can be decomposed into the calculus expressions $F_{i_j} = \{n \mid SearchContext(n) \wedge D_{i_j}\}$ and it is not difficult to see that $QE(F)$ = $(QE(F_{1,1})$ AND ... AND $QE(F_{1,r_1}))$ OR ... OR $(QE(F_{s,1})$ AND ... AND QE(F_{s,r_s})). Thus, we can focus only on converting each $F_{i,j}$.

As seen above, each $F_{i,j}$ is of the form $\exists p \; hasPos(n,p) \wedge \bigwedge_r H_r(n,p)$ or of the form $\neg\exists p \; hasPos(n,p) \wedge \bigwedge_r H_r(n,p)$, where $H_r(n,p)$ is $hasToken(p,t)$ or $\neg hasToken(p,t)$. In either case, we can consider there are no duplicates among $H_r(n,p)$; otherwise, we can simply eliminate them.

Let us first consider the case where $F_{i,j} = \exists p \; hasPos(n,p) \wedge \bigwedge_t H_t(n,p)$.

- If there exists r_1 and r_2 such that $H_{r_1}(n,p) = hasToken(p,t_1)$, $H_{r_2}(n,p) = hasToken(p,t_2)$, and $t_1 \neq t_2$, then the condition "one token per position" (Section 2.2) is violated. Therefore, $F_{i,j}$ is the empty set. $QE(F_{i,j})$ = `ANY AND NOT(`$t_1 OR ... OR t_c$`))`, where $\mathcal{T} = \{t_1, ..., t_c\}$ is the set of all tokens. Intuitively, this query returns the empty set because it requires the result nodes to contain a token that is not in $\mathcal{T}$, which is impossible.

- If there exists r_1 and r_2 such that $H_{r_1}(n,p) = hasToken(p,t)$ and $H_{r_2}(n,p) = \neg hasToken(p,t)$ then this is an obvious contradiction and $F_{i,j}$ is the empty set. We define $QE(F_{i,j})$ = `ANY AND NOT(`$t_1 OR ... OR t_c$`))` as above.

- Let there exists r_1 and there does not exist r_2 such that $H_{r_1}(n,p) = hasToken(p,t)$ and $H_{r_2}(n,p) = \neg hasToken(p,t)$. Then we can ignore any $H_r(n,p)$ which contains $\neg hasToken(p,t')$ for some token $t' \neq t$. The latter are trivially satisfied. In this case, $F_{i,j}$ = $\{n \mid SearchContext(n) \wedge \exists p \; hasPos(n,p) \wedge hasToken(p,t)\}$, which is exactly the semantics for $QE(F_{i,j})$ = t.

- The last case is $F_{i,j} = \{n \mid SearchContext(n) \wedge \exists p \;\; hasPos(n,p) \wedge \neg hasToken(p,t_{i_1}) \wedge ... \wedge \neg hasToken(p,t_{i_v})$. This expression can be interpreted as the condition that n contains a token from the complement $\{t_{j_1}, ..., t_{j_u}\}$ of $\{t_{i_1}, ..., t_{i_v}\}$ with regards to the set $\mathcal{T} : \{t_{j_1}, ..., t_{j_u}\} = \mathcal{T} - \{t_{i_1}, ..., t_{i_v}\}$. Due to the finiteness of $\mathcal{T}$, $F_{i,j} = \{n \mid SearchContext(n) \wedge \exists p \; hasPos(n,p) \wedge (hasToken(p,t_{j_1}) \vee ... \vee hasToken(p,t_{j_u}))\}$ = $\bigcup_r \{n \mid SearchContext(n) \wedge \exists p \; hasPos(n,p) \wedge hasToken(p,t_{j_r})\}$. The latter is trivially equivalent to the query $QE(F_{i,j})$ = t_{j_1} `OR ... OR` t_{j_u}.

In case $F_{i,j} = \neg\exists p \;\; hasPos(n,p) \wedge \bigwedge_t H_t(n,p)$, then $QE(F_{i,j})$ = `NOT` $QE(\neg F_{i,j})$, where $\neg F_{i,j}$ is transformed as in the previous case.

Theorem 6: Completeness of COMP

Proof Sketch: We will prove that every calculus query can be represented by a COMP query $CompQuery$. We use induction on the structure of the query expression $CalcExpr(n, p_1, ..., p_k)$.

- If $CalcExpr(n, p) = hasPos(n, p)$, then $CompQuery$ = p `HAS ANY`. This is equivalent to $CalcExpr$ by definition.
- If $CalcExpr(n, p) = hasToken(p, 'token')$, then $CompQuery$ = p `HAS` $'token'$. This is equivalent to $CalcExpr$ by definition.
- If $CalcExpr(n, p_1, ..., p_k) = pred(p_1, ..., p_k, c_1, ..., c_q)$, then $CompQuery = \mathtt{pred}(\mathtt{p_1}, ..., \mathtt{p_k}, \mathtt{c_1}, ..., \mathtt{c_q})$. This is equivalent to $CalcExpr$ by definition.
- If $CalcExpr(n, p_1, ..., p_l, q'_1, ..., q'_m, q''_1, ..., q''_c)$ = $CalcExpr_1(n, p_1, ..., p_l, q'_1, ..., q'_m) \wedge CalcExpr_2(n, p_1, ..., p_l, q''_1, ..., q''_c)$, where $k = l + m + c$, $CalcExpr_1$, and $CalcExpr_2$ are calculus query expressions with equivalent COMP queries be $CompQuery_1$ and $CompQuery_2$, then $CompQuery$ = $CompQuery_1$ `AND` $CompQuery_2$. This is equivalent to $CalcExpr$ by definition.
- If $CalcExpr(n, p_1, ..., p_l, q'_1, ..., q'_m, q''_1, ..., q''_c)$ = $CalcExpr_1(n, p_1, ..., p_l, q'_1, ..., q'_m) \vee CalcExpr_2(n, p_1, ..., p_l, q''_1, ..., q''_c)$, where $k = l + m + c$, $CalcExpr_1$, and $CalcExpr_2$ are calculus query expressions with equivalent COMP queries be $CompQuery_1$ and $CompQuery_2$, then $CompQuery$ = $CompQuery_1$ `OR` $CompQuery_2$. This is equivalent to $CalcExpr$ by definition.
- If $CalcExpr(n, p_1, ..., p_k) = \neg CalcExpr'(n, p_1, ..., p_k)$, where $CalcExpr'$ is a calculus query expression that is equivalent to the COMP query $CompQuery'$, then $CompQuery$ = `NOT` $CompQuery'$. This is equivalent to $CalcExpr$ by definition.
- If $CalcExpr(n, p_1, ..., p_k)$ = $\exists p_{k+1}\ hasNode(n, p_{k+1}) \wedge CalcExpr'(n, p_1, ..., p_{k+1})$, where $CalcExpr'$ is a calculus query expression that is equivalent to the COMP query $CompQuery'$, then $CompQuery$ = `SOME` p_{k+1} `(` $CompQuery'$ `)`. This is equivalent to $CalcExpr$ by definition.
- If $CalcExpr(n, p_1, ..., p_k) = \forall p_{k+1}\ hasNode(n, p_{k+1}) \Rightarrow CalcExpr'(n, p_1, ..., p_{k+1})$, where $CalcExpr'$ is a calculus query expression that is equivalent to the COMP query $CompQuery'$, then $CompQuery$ = `EVERY` p_{k+1} `(` $CompQuery'$ `)` . This is equivalent to $CalcExpr$ by definition. $\square$