Score-Consistent Algebraic Optimization of Full-Text Search Queries with GRAFT

Nathan Bales
UC San Diego
La Jolla, CA
nbales@cs.ucsd.edu

Alin Deutsch
UC San Diego
La Jolla, CA
deutsch@cs.ucsd.edu

Vasilis Vassalos
A.U.E.B.
Athens, Greece
vassalos@aueb.gr

ABSTRACT

We address two open problems involving algebraic execution of full-text search queries. First, we show how to correctly apply traditional database rewrite optimizations to full-text algebra plans with integrated scoring, and explain why existing techniques fail. Second, we show how our techniques are applied in a generic scoring framework that supports a wide class of scoring algorithms, including algorithms seen in the literature and user-defined scoring.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Search process

General Terms

Design; Performance

1. INTRODUCTION

Though not commonly used for web-search, search systems that reason about individual word positions (henceforth: *full-text search*) are commonly used in contexts where expert users require more focused queries and precise results [9]. Full-text search systems[3, 27, 14, 22, 19] power search for enterprise, government, academia, scientific applications such as Westlaw[32], PubMed[30], and the U.S. Library of Congress. The choice of a powerful but more complex language is appropriate for sophisticated expert users and for search systems with GUI-generated queries. Expressive power, and thus improved precision, entails complex evaluation plans and higher evaluation cost. To make full-text search feasible despite high evaluation cost, plans must be optimized like database queries.

Expert users who demand the power of full-text search, also expect to be able to use their preferred ranking function [21], or tailor a ranking function [6] to their specific need. It follows that developers of full-text search systems need to support a wide variety of ranking functions, even plug-in ranking functions, to appeal to the widest possible audience. This fact is supported anecdotally; Terrier [22] ships with, as of September 2010, two different weight aggregation techniques, 15 different term-weighting functions, and support for user defined term-weighting functions.

Many state-of-the-art full-text systems do not support multiple ranking algorithms [27, 14], and those that do, like Terrier, support narrow classes thereof. Rigid plan generation, which hard-codes assumptions about the specific ranking algorithm, limits generic ranking support in these systems; such systems only support ranking algorithms that fit their assumptions. Ranking techniques such as proximity metrics [16, 25] do not fit typical evaluation plans.

To support a wider class of ranking algorithms, IR systems must either generate highly generic plans (sacrificing performance), implement multiple rigid generators, one per supported ranking algorithm (sacrificing maintainability and extensibility), or deploy a flexible generator that takes a ranking algorithm as a parameter and adjusts plans to fit it. This paper studies flexible plan generation.

Our approach is to add generic ranking to database-style algebraic strategies for full-text evaluation recently proposed in the database community [2, 7]. Ranking is defined against a 'canonical' plan that, for each document d, first computes the set S of query 'matches' and then scores d based on S. From the canonical plan and a selected ranking algorithm, our optimizer finds a plan which avoids large intermediate results by interleaving matching and scoring. If, for instance, the Lucene ranking algorithm [27] is selected, then our optimizer finds a plan of comparable performance to the one yielded by Lucene's rigid plan generator. If a context-sensitive ranking algorithm is selected, then our optimizer finds an optimized plan consistent with that algorithm. An optimal plan for context-sensitive ranking will look very different from an optimal plan for Lucene's ranking, even for the same query.

Prior works in full-text algebras [2, 7] provide a useful foundation for extension with generic ranking, but these works focus on the boolean retrieval problem (identifying, but not ranking documents that match the query) and, as we show in Section 2, optimizers designed for boolean retrieval cannot be straightforwardly re-purposed for consistent ranked retrieval. Simply put, scoring discussed in prior work depends on intermediate results that optimizers may change. Different scoring schemes are sensitive to different changes in the intermediate results. An optimizer needs to carefully avoid optimizations that lead to score changes, while making full use of those that preserve scores; thus the optimizer should choose which optimizations to apply based on the selected scoring scheme. Building a generic optimizer that correctly applies optimizations for plug-in ranking algorithms, is the primary technical challenge we address.

We call an optimizer score-consistent if all produced execution plans of a particular query yield the same score for the same combination of document and ranking algorithm.

Given the effort usually exerted by application owners into ranking results, and the competitive advantage ranking algorithms often entail, making sure scores do not change due to optimization is not only a technical challenge but also a serious business issue. Ranking algorithms are frequently tuned for specific applications to guarantee some level of precision, recall, and/or 'fairness' that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

must be maintained. Moreover, a score-consistent optimizer and execution engine can be updated and improved without potentially adverse effects on the (carefully tuned) ranking of the results and without requiring changes to ranking implementation.

Desiderata. The goal of this work is to provide a system that: (1) supports expressive full-text search and flexible generic scoring; (2) has an optimizer that exploits both known and novel optimizations techniques without introducing inconsistent scoring; (3) despite overhead from generic scoring, performs competitively with systems using a fixed scoring algorithm; and (4) isolates scoring interface from optimizer such that scoring scheme developers need not understand the optimizer or respond to optimizer changes.

Contributions. This paper presents GRAFT (Generic Ranking Algebra for Full Text), a full-text algebra with integrated generic scoring framework. We show how GRAFT supports known and novel optimizations by providing the optimizer just enough information about a selected scoring implementation – including user-defined implementations – to select only the compatible optimizations. Reaching this result, we make the following contributions:

1. We study the interplay between scoring and match computation and show that blind application of traditional database optimizations to match computation can result in different scores. Such blind application of optimizations is known to arise naturally in systems in which the optimizer is developed initially for boolean retrieval, and scoring is integrated afterward (see Section 2).

2. We present the first formal model for generic ranking of full-text queries. The model comprises a small set of operators (a *scoring algebra SA*). We validate the expressive power of our model by capturing ranking algorithms and techniques found in the literature [7, 13, 16, 20, 22, 25, 28, 29, 34].

3. We propose an integrated algebraic model for the interplay between score computation and match computation. This model allows us to reason about the correctness of traditional optimizations in ranked full-text query systems.

4. We show that our integrated model accommodates novel optimizations involving both scoring and match algebra operators.

5. We identify a small set of SA operator properties relevant to applicability of optimizations. We show how to automatically configure the optimizer, based on these properties, to exploit the available optimization potential without compromising score consistency.

6. Finally, we show experimentally that optimization allows a system that supports generic scoring to compete with a system that does not, and that our novel optimizations are powerful.

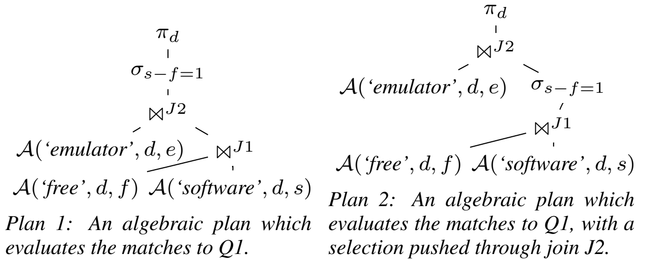
Paper organization. Section 2 motivates our work with examples that illustrate how we solve the score-consistency problem found in existing full-text algebras. Section 3 describes a relational model for full-text based on sequences of match tuples called match tables. Section 4 defines our novel model for generic scoring and outlines the GRAFT framework. Section 5 discusses optimizations, both classical database optimizations, and optimizations that cross the boundary between matching and scoring. With each optimization, we list the scoring scheme properties that must hold for the optimization to be score-consistent. Section 6 discusses the complexity of evaluating relational full-text queries. Section 7 is a study of scoring schemes observed in the literature[7, 13, 16, 20, 22, 25, 28, 29, 34]. Section 8 reports on the implementation and experimental evaluation of GRAFT. We conclude in Section 9.

2. MOTIVATION

We motivate our work with an example that both illustrates why state-of-the-art full-text algebra solutions are unsatisfactory for generic ranking, and gives the intuition behind our solution. This section also introduces vocabulary used in the following sections.

TOKEN	DOC	#INDOC	#DOCS	OFFSETS
'emulator'	d_w	1	2768	[64]
'free'	d_w	1	332335	[3]
'foss'	d_w	1	2044	[179]
'software'	d_w	4	71735	[4, 32, 180, 189]
'windows'	d_w	4	43949	[27, 42, 144, 187]

Figure 1: A fragment of a normalized term-position index for a document d_w. The fragment includes all positions of keywords used in later examples. OFFSETS *is a list of positions of the token in d_w.* #INDOC *is the number of occurrences of the keyword in d_w, and* #DOCS *is the the count of documents in the collection that contain the keyword.*



Plan 1: An algebraic plan which evaluates the matches to Q1. *Plan 2: An algebraic plan which evaluates the matches to Q1, with a selection pushed through join J2.*

Expressive power to reason about term positions is the fundamental difference between full-text search and classical keyword search. The document model for full-text search is a sequence of words, whereas for pure keyword search it is a bag of words. Figure 1 shows part of a full-text index for document d_w[1]. This index contains positional information: it records the position (offset) of each word occurrence in the document.

The expressive power of full-text search is illustrated in Query Q1, a simple query over some collection of documents that includes d_w.

find documents with 'emulator', 'free' and 'software' s.t. 'free' appears immediately before 'software' (Q1)

Keyword search systems that do not index the full-text (word positions) cannot answer this query without scanning the document text because *'appears immediately before'* involves term positions, which the bag-of-words document model does not maintain.

Document d_w is an answer to Q1 because it has one *match* to Q1 using tokens at offsets 64 (emulator), 3 (free), and 4 (software). Informally, matches are tuples consisting of a document id and token offsets that satisfy the query, such as $\langle d_w, 64, 3, 4\rangle$. If not for the *'appear immediately before'* clause, Q1 would have four matches, one of the four different positions for 'software': $\langle d_w, 64, 3, 4\rangle$, $\langle d_w, 64, 3, 32\rangle$, $\langle d_w, 64, 3, 180\rangle$, and $\langle d_w, 64, 3, 189\rangle$. Three of these tuples are not matches because the value of the fourth field minus the value of third field is greater than one, indicating that 'free' does not appear immediately before 'software'.

Plan 1 and Plan 2 both compute the answer to Q1 under Boolean semantics. These plans compute 'match tuples' for each document, but project only the document ids (a set of documents is the answer to a boolean search query). $\mathcal{A}(\textit{'emulator'}, d, e)$ is a relation consisting of every pair of document id (d) and offset (e) such that 'emulator' appears in d at position e. $\mathcal{A}(\textit{'emulator'}, d, e)$ abstracts a term-position index scan over all positions of 'emulator' in the document collection. The joins (*J1* and *J2* for reference) are natural inner joins, and the other operators are standard relational algebra[15]. Plan 1 processes selection after the joins in a manner consistent with automatic translations from a calculus. Plan 2 selects before join *J2*, and is derived from Plan 1 using a textbook selection-pushing rewrite that preserves Boolean semantics.

State-of-the-Art. Plans 1 and 2 illustrate why state-of-the-art full-text algebras cannot simultaneously facilitate generic scoring and

[1] d_w is a real document; specifically, the abstract portion of the Wikipedia article `Wine_(software)` as of March 10, 2009.

score-consistent optimizations. The state-of-the-art full-text algebra [7] extends each match tuple with a score, and extends each algebra operator with a function to manipulate the scores. As plan evaluation constructs and combines match tuples, it simultaneously computes and aggregates match scores using the scoring functions. Join, for instance, is extended with the function SJ (Score Join). $\text{SJ}(m_L, m_R)$ computes the score of an output tuple m, such that the arguments m_L and m_R are tuples that join to form m. This framework supports generic ranking. A new ranking algorithm is 'plugged in' by providing new scoring function implementations.

An example implementation, given in [7], for $\text{SJ}(m_L, m_R)$ is $\frac{m_L.s}{|M_R|} + \frac{m_R.s}{|M_L|}$ where m_L is a tuple from the left input, $m_L.s$ is the score of tuple m_L. $|M_R|$ is the number of tuples from the right input that join with m_L. Intuitively, the value $m_L.s$ is distributed equally among the output tuples m_L contributes to, so that the join operator does not create or eliminate score value. Returning to the example, assume m_L is $\langle d_w, 64, s\rangle$ (s is the score of the tuple). In Plan 1, m_L joins (join *J2*) with 4 tuples and the value $m_L.s$ is distributed evenly between them. Three of the resulting tuples are eliminated in the selection, thus only one-quarter of $m_L.s$'s score value contributes to the final score of d_w. In Plan 2, the three tuples have already been eliminated by the selection before they join m_L. The value of $m_L.s$ is distributed to only one tuple, and the whole value of $m_L.s$ contributes to the final score of d_w. Thus, while the selection pushing optimization does not affect which matches are computed by the plan, it affects document scores. To avoid score inconsistency the selection pushing optimization must be disabled when using this scoring function.

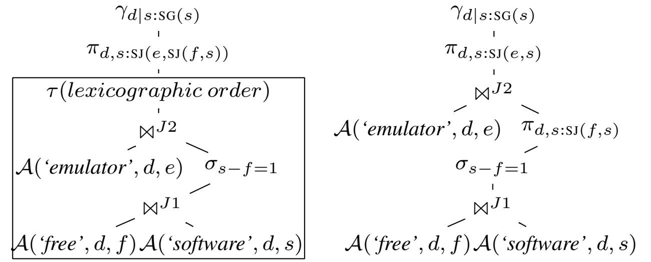
Score inconsistency in this example is not the fault of a poorly chosen scoring function, it is a result of encapsulating score computation into operators that compute matches. Textbook selection pushing was designed for Relational Algebra and preserves the semantics of computing tuples (matches), but it is unaware of scoring, which it does not preserve.

Proposed Solution. We model scoring using similar scoring functions as the framework in [7], but without encapsulating them in the relational algebra operators. Instead, scoring functions are standalone *aggregate functions* that interact with the other relational algebra operators in the standard way.

We define the correctness of scoring based on the principle of *score isolation*. A plan is score-isolated when all matches are computed by a *matching subplan* which contains no scoring. This subplan yields a *match table* containing the matches in a defined order (e.g. lexicographic). Scoring semantics are defined as aggregation of the match table, thus (conceptually) isolating scoring from match computation. Plan 3 is a score-isolated plan for Q1, showing the matching subplan in a box. The match table is well-defined at the semantic, optimization-independent level, thus enabling an optimization-independent definition of the intended scoring semantics, namely score-isolation. This in turn is key to define score consistency meaningfully, by identifying the score to be preserved.

Following the matching subplan, matches for the same document are grouped and aggregated into a document score. Two score aggregation operators are used in Plan 3: SJ and SG (Score Group). SJ still computes the score of joined matches, but not immediately when joined. Instead, SJ is explicitly called outside the matching subplan. SG is an aggregate function (like SUM in SQL) that computes the score of grouped matches. For specifics on algebra notation see Section 3.2 and for scoring semantics see Section 4.

Match tables can be large (see Section 6). Optimizing score-isolated query plans entails interleaving matching and scoring in a way that avoids materializing the entire match table *while computing the same answers and scores*. A crucial part of this work seeks



Plan 3: A score-isolated evaluation plan for Q1 displaying the matching subplan within a box.

Plan 4: A plan derived from Plan 3 by rewrite optimization. (Only for some scoring schemes!)

machinery to optimize such plans in a score preserving way using a mix of classical relational optimizations as well as novel optimizations that cross the boundary between matching and scoring.

Definition 1 (Score Consistency). *An optimizer is* score-consistent *if, given a score-isolated input plan, it produces only plans that compute the same answers and scores as the input plan.*

To interleave matching and scoring without violating score consistency, an optimizer must know which optimizations are tolerated by the selected scoring function implementation. A naïve approach requires each function to specify (in meta-data) which rewrites it tolerates. For instance, to reach optimized Plan 4 from score-isolated Plan 3, it must be true that SG tolerates unsorted matches (since the sort operator is removed), and that SJ tolerates being pushed through joins. But this naïve approach requires the scoring function designer to know and care about the inner workings of the optimizer, and limits the applicability of the scoring function implementation to future optimizations.

A better approach has the scoring designer specify a small set of fundamental properties about her implementation, (e.g. which functions are commutative, fully-associative, idempotent, monotonic, etc.) and allow the optimizer to infer which optimizations will preserve score consistency given a selected scoring function. In Section 5 we describe our solution along these lines.

3. RELATIONAL MODEL FOR FULL-TEXT

The relational model for full-text evaluation has recently been explored within the database community [7, 2]. Full-text queries in the relational model are first-order formulae over term positions. In this section we define MCalc, a full-text calculus used to specify a set of matches, and MA, a full-text algebra used to compute those matches. Ranked retrieval needs this set of matches because scoring functions measure the connection between document and query; matches establish that connection.

3.1 Matching Calculus

The Matching Calculus (MCalc) specifies the set of matches to a full-text query in a collection of documents, in the style of the Domain Relational Calculus. The basic primitive in MCalc is the HAS predicate. $\text{HAS}(d, p, k)$ is true when the word k appears in document d at position p, and is false otherwise. All HAS predicates have one document variable argument, one position variable argument, and one keyword (variable or literal) argument.

Full-text predicates define relationships between the positions of keywords. For example: $\text{DISTANCE}(p_1, p_2, n)$ is true when the distance from position p_1 to p_2 is exactly n, and $\text{PROXIMITY}(p_1, p_2, n)$ is true when the distance from position p_1 to p_2 is at most n. More generally, MCalc supports full-text predicates of the form $\text{PRED}(\bar{p}, \bar{c})$ which specify constraints on position variables in the vector $\bar{p}$. Optional constants, $\bar{c}$, parameterize the constraints. MCalc is general enough to support generic positional predicates [5].

MCalc queries take the form $\{\langle d, \bar{p}\rangle | \Psi(\bar{p}) \text{ is satisfied}\}$, where Ψ is a first-order logic formula over the primitives HAS, EMPTY (described below) and full-text predicates. The free variables in Ψ are exactly the document variable d and the position variables in $\bar{p}$.

Example 1 (MCalc Query):

$$\{\langle d, p_0, p_1, p_2\rangle | \text{HAS}(d, p_0, \text{`emulator'}) \wedge \text{HAS}(d, p_1, \text{`free'}) \wedge \text{HAS}(d, p_2, \text{`software'}) \wedge \text{DISTANCE}(p_1, p_2, 1)\} \quad \text{(Q2)}$$

Query Q2 is a MCalc query that finds all matches to Q1. The HAS predicates restricts position variables p_0, p_1, and p_2 to positions of keywords 'emulator', 'free', and 'software' respectively. The DISTANCE predicate specifies the distance between values of p_1 ('free') and p_2 ('software') must be exactly 1, expressing the requirement that 'free' occur immediately before 'software'. ♢

MCalc also has a built-in EMPTY predicate that is necessary to support optional keywords, and to ensure safe disjunctive queries. $\text{EMPTY}(p)$ is true when p binds to the 'empty position' symbol $\varnothing$. An empty position value does not imply the keyword associated with the variable is absent, just that its presence, or lack thereof, is inconsequential to a particular match.

Example 2 (MCalc Query with Disjunction):

$$\{\langle d, p_0, p_1, p_2, p_3, p_4\rangle | (\Psi^0 \vee \Psi^1) \wedge \text{HAS}(d, p_0, \text{`windows'}) \wedge \text{HAS}(d, p_1, \text{`emulator'}) \wedge \text{WINDOW}(p_0, p_1, 50)\}$$

$$\Psi^0 := \text{EMPTY}(p_2) \wedge \text{EMPTY}(p_3) \wedge \text{HAS}(d, p_4, \text{`foss'}) \quad \text{(Q3)}$$

$$\Psi^1 := \text{HAS}(d, p_2, \text{`free'}) \wedge \text{HAS}(d, p_3, \text{`software'}) \wedge \text{EMPTY}(p_4) \wedge \text{DISTANCE}(p_2, p_3, 1)$$

Q3 asks for matches where the keywords 'windows' and 'emulator' appear within a window of 50 words, and are accompanied by either the keyword 'foss', or by the phrase 'free software'. ♢

The formal meaning of 'match' is given with respect to MCalc:

Definition 2 (Match). *The tuple $\langle d, \bar{p}\rangle$ is a match of query Ψ in document d iff $\langle d, \bar{p}\rangle$ is a satisfying assignment for Ψ that maps the free position variables in Ψ to either word positions in d or to $\varnothing$.*

MCalc vs. State of the Art. State-of-the-art full-text calculi, such as FTC[7], were not designed to specify a set of matches (only documents), and sometimes under-specify matches. Matches to FTC queries may contain positions of keywords not mentioned in the query[5]; such matches are not useful for scoring. MCalc adds a safe-range[1] requirement (similar to SQL) which restricts matches to only those useful for scoring by binding under-specified position variables to $\varnothing$ via the EMPTY predicate. Otherwise MCalc has equivalent expressive power to FTC, which was shown in [7] to subsume other full-text specifications, including predicate-based languages [8, 4], and text-regional algebras[12]. See [5] for details on safety, expressive power, and MCalc's relationship to FTC.

3.2 Matching Algebra

We express evaluation plans for MCalc queries in the Matching Algebra (MA). MA has enough expressive power to express all safe MCalc queries, as Relational Algebra has enough expressive power to express safe Relational Calculus queries. Queries may be translated between MCalc and MA using traditional translation methods developed for the Relational Calculus and Relational Algebra [1].

MA operates on, and outputs *match tables*. Match tables are lists of matches, which are tuples of form $\langle d, p_0, ..., p_n\rangle$ where d ranges over documents, and p_i ranges over term positions and $\varnothing$. Match tables are *lists* (rather than sets) of tuples; table rows and columns are both sequenced, and tables may contain duplicate rows. We use "matches", "tuples" and "rows" interchangeably. The use of tables is consistent with practical relational algebra and SQL implementations. The match table for query Q3 on document d_w (from Section 2) is shown in Figure 2.

The Matching Algebra consists of familiar Relational Algebra operators: natural join $\bowtie$, outer bag-union $\uplus$ [11], selection σ, generalized projection π, anti-join $\triangleright$, group γ, and sort τ. A formal specification and description of each operator is available[5]. Due to space constraints, we focus this discussion on the novel Atomic Match Factory ($\mathcal{A}$) and clarify our notationfor π and γ.

Atomic Match Factory: $\mathcal{A}$, is the terminal operator in MA, corresponds logically to the MCalc HAS predicate, and abstracts a scan of the term-position index. The expression $\mathcal{A}(d, p, k)$ produces a match table with two columns, specifically, a document id named d and a term position named p. The match table contains one tuple $\langle d, p\rangle$ for each d and p satisfying $\text{HAS}(d, p, k)$ – one for each occurrence of keyword k in the index. To simplify presentation we assume that a system has a single library of documents indexed, and that all queries are applied to the entire library.

Notation. The subscript of π contains a list of attribute names and assignments with the form $a{:}f(b)$. The attribute names are the usual fields that are preserved through a projection, while the assignment means attribute a in the output tuple gets the value of function f applied to attribute b in the input tuple.

The subscript of γ has the form $A|B$, where A is the list of group-by attributes, and B a list of assignments of the form $a{:}f(b)$. In this case, f is an aggregate function (e.g. SUM), and field a in the output tuple is the result of using f to aggregate the values of b in the grouped input tuples. We use the following shorthand for binary aggregation operators: $a{:}{+}(b) \equiv a := g_0.b + g_1.b + \cdots + g_n.b$, where $g_0 \ldots g_n$ are the grouped tuples.

4. SCORING

Scoring functions measure evidence that establishes the connection between a document and a query. For GRAFT, we consider a class of scoring algorithms, called *match-scoring* algorithms, that measure specifically the evidence contained in the list of matches to the query against the document.

We choose match-scoring algorithms for several reasons. Match-scoring captures many real world algorithms because the list of matches is a superset of the evidence they require. The match table, which is already computed for boolean evaluation, contains the list of matches. Finally, we can model match-scoring algorithms using a simple algebra of scoring operators that can be integrated with the Match Algebra, uncovering optimization opportunities.

We validate the expressiveness of our algebra, and of match-scoring, in Section 7 by demonstrating how several scoring algorithms seen in the literature [7, 13, 16, 20, 25, 22, 28, 29, 34] are implemented as *scoring schemes* in our algebra. A scoring scheme is an implementation of the operators of our scoring algebra.

Section 4.1 introduces the Scoring Algebra (SA) and explains the underlying intuition, Section 4.2 discusses subtleties of score aggregation which must be considered when choosing a scoring algebra plan, and Section 4.3 explains how SA and MA work together to evaluate ranked full-text queries. Due to space constraints, we relegate to [5] the formalization of the intuitive presentation in Section 4.1 and Section 4.2. [5] defines the score of a match table inductively, as an aggregation of scores of match subtables, and summarizes the rules that govern the allowed choices of subtables.

4.1 Scoring Algebra

The Scoring Algebra (SA) is an algebra of composable abstract scoring operators, whose implementations, called *scoring schemes*, express match-scoring functions. A *canonical SA plan* takes as input a match table (the result of computing a MA query) and outputs

a list of scored documents. An optimized plan interleaves SA and MA operators, as discussed in Section 4.3, to avoid materializing full match tables because, as we show in Section 6, their size can be quite large. A score for document d is computed from the query matches to d using the following three-step process:

Step 1: Initialization. The term position p in each table cell is replaced with an initial score value $\alpha(p)$. α is the *initializer function* in SA, and typically implements a term weighting function such as TF-IDF[18], BM25[18], KL Divergence[18], etc.

Step 2: Aggregation. Initial scores are aggregated into a single, but not final, aggregate score value. Modeling this step using a single aggregation operator would be too crude, because the match table has two dimensions with different semantics. Term positions within the same row have a different relationship than positions in the same column. Furthermore, term positions in the same row that match conjunctive sub-expression *'foo'*$\wedge$*'bar'* have a different relationship than positions matching disjunctive sub-expression *'foo'*$\vee$*'bar'* (details in Section 4.2.1). We developed three binary aggregation operators to reflect the relationships between positions.

We call term positions within the same match table column *alternates* of each other. Within the same row, we say that positions in columns corresponding to a conjunctive (disjunctive) sub-expression of the query are *conjuncted (disjuncted)*. The relationship among positions induces a relationship among scores. If positions p, p' are alternates/conjuncted/disjuncted, we say that so are the scores $\alpha(p)$,$\alpha(p')$. Conjuncted, disjuncted, and alternate scores are combined using, respectively, the conjunctive combinator $\owedge$, the disjunctive combinator $\ovee$, and the alternate combinator $\obar$.

Step 3: Finalization. The finalization function ω computes a final, floating point, score for a document from the aggregate score produced by Step 2. The aggregate score is a structure, called an *internal score*, composed of one or more values that are aggregated independently. Internal scores are necessary so that score aggregation, which is defined using binary operators, can express complicated aggregation functions that do not normally compose when expressed as binary operators. For instance, the mean of $\{a \dots y, z\}$ cannot be computed given the mean of $\{a \dots y\}$ and value z. It can be computed given z and both the sum and count of $\{a \dots y\}$ as the pair (s, c). Adding z to s and incrementing c yields a new pair (s', c'). The internal score type for mean is the pair $\langle sum, count\rangle$ and $\omega((s', c')) = \frac{s'}{c'}$ yields the final score.

The ω function also performs other post-processing including normalization and incorporation of match-unrelated score components such as document age, PageRank[23], etc.

Summary. SA comprises six operators (α, ω, $\owedge$, $\ovee$, $\obar$). The initialization function α scores individual match table cells. Three binary score aggregation operators $\owedge$, $\ovee$, and $\obar$ aggregate the scores of the individual cells into a single (internal) aggregate score. Finally, the finalization function ω post-processes the internal score.

Example 3 (MEANSUM Scoring Scheme Implementation): We present a scoring scheme called MEANSUM, chosen because it helps illustrate our points effectively in a single example. More scoring schemes reflecting the real world scoring algorithms in [7, 13, 16, 20, 22, 25, 28, 29, 34] can be seen in Section 7.

$$
\begin{array}{l}
\alpha(d,c,p) : \textbf{if } p \textbf{ is } \varnothing \textbf{ then} : \textbf{return } \langle 0.0, 1\rangle \\
\qquad\qquad \textbf{else} : \textbf{let } tfidf := \frac{p.\#InDoc}{d.length} \times \frac{d.collectionSize}{p.\#Docs} \\
\qquad\qquad\quad \textbf{return } \langle tfidf, 1\rangle \\
\obar(s_1, s_2) : \textbf{return } \langle s_1.sum + s_2.sum, s_1.count + s_2.count\rangle \\
\owedge(s_1, s_2) : \textbf{return } \langle s_1.sum + s_2.sum, s_1.count\rangle \\
\ovee(s_1, s_2) : \textbf{return } \langle s_1.sum + s_2.sum, s_1.count\rangle \\
\omega(d, s) : \textbf{let } mean := \frac{s.sum}{s.count} \\
\qquad\quad \textbf{return } 1 - \frac{1}{ln(mean+e)}
\end{array}
$$

MEANSUM defines the score of a document as the average score of all its alternate matches, and the score of a match as the total score of the individual positions in the match. Term positions in MEANSUM are scored by *tfidf*[18].

The initializer function α produces a score for a single match table cell populated by a term position or $\varnothing$. The function takes three arguments: a document d, a match table column c, and a term position p. Position p must appear in document d and match the query variable corresponding to column c. Each of these arguments is not merely an id, but a collection of relevant statistics (e.g. the document argument d includes the document length as $d.length$). When the position argument is $\varnothing$, the implementation returns the pair of zero and one. Otherwise, p contains a word position summary, and the implementation returns a pair of a tfidf value and the value one. MEANSUM uses pairs as its internal score type. The members of the pair are the two components of a mean computation: the sum of tfidf scores, and the $count$ of aggregated matches (rows).

The alternate combinator combines alternate scores s_1 and s_2. Each input score is a sum of tfidfs, and a count of matches. The sums are added, as are the counts (alternate match sets must be disjoint by definition.)

MEANSUM does not differentiate between conjuncted and disjuncted scores, thus $\owedge$ and $\ovee$ have the same implementation. The arguments s_1 and s_2 are conjuncted or disjuncted scores. The intuition behind the implementation is similar to that of $\obar$, only that in this case s_1 and s_2 by definition refer to the same set of matches, so they have the same counts, which are preserved by $\owedge$ and $\ovee$.

The finalizer function ω computes the final score value as a floating point number. ω takes two arguments, d is the document, and s is the internal score computed by aggregating the entire match table. The sum member of the pair score is divided by the count member to compute the mean, and the final score is computed by normalizing the mean to the range [0:1]. ◇

4.2 Match Table Aggregation

SA operators aggregate match (sub)tables into single aggregate scores. This aggregation is formally defined from the top down, but practically implemented using bottom-up plans. For now, we confine our discussion to the top-down definition.

Match table scoring is defined inductively. Two *column-wise* or *row-wise* subtables are chosen and scored recursively. The scores of the subtables are combined using $\owedge$, $\ovee$, or $\obar$ to reach a score for the full table. *Row-wise* subtables partition the match table rows. *Column-wise* subtables partition the match table columns (and thus the query variables, which provide the column names). The query variables are partitioned into *conjuncted* or *disjuncted* variable sets (see Section 4.2.1 for more).

Match tables with more than one column and/or more than one row allow multiple partitions into subtables. The choice of subtables cannot be arbitrary, rather it must consider the selected scoring scheme. As a simple example, consider a match table with two rows a and b. There are two possible row-wise subtable pairs: $(\{a\}, \{b\})$ and $(\{b\}, \{a\})$ which differ merely in order. Given these two choices, the score of the match table is either $score(a) \obar score(b)$ or $score(b) \obar score(a)$. Clearly, if $\obar$ is commutative, then either pair results in the same score, otherwise the score will depend on which pair is chosen, and a score-consistent optimizer must choose so as not to perturb the desired order among matches. A generic score-consistent optimizer must therefore know if the $\obar$ implementation of the selected scoring scheme is commutative.

While the commutativity of $\obar$ provides a simple example, the following two sub-sections detail two far more subtle issues that guide the choice among potential subtables.

p_0	p_1	p_2	p_3	p_4
27	64	∅	∅	179
27	64	3	4	∅
42	64	∅	∅	179
42	64	3	4	∅

Figure 2: The match table for Q3 over d_w. Each cell contains either the empty position symbol ∅, or a term position. Only term offset is shown for each term position, the other statistics (See Figure 1) are hidden for brevity.

4.2.1 Choosing Column-Wise Subtables

The structure of the query expression defines relationships between match table columns and between column-wise subtables. Two columns q and r are *conjuncted* if $q \wedge r$ is a subexpression of the query and *disjuncted* if $q \vee r$ is a subexpression of the query. Similarly, two column-wise subtables with variable (column) sets Q and R are conjuncted if $\Psi(Q) \wedge \Psi(R)$ is a subexpression of the query, where $\Psi(Q)$ is a subexpression over the free variable set Q. Since the columns in a full match table have a one-to-one mapping to the free variables in a valid query expression, valid conjuncted or disjuncted subtables are identified using the query's syntax tree.

A *scoring plan* $\Phi(\bar{p})$ is a syntactic transformation of a query $\Psi(\bar{p})$ which provides information needed to determine column-wise subtables: the structure of conjunctions and disjunctions between free position variables. The transformation procedure is as follows: erase all non-HAS predicates, erase HAS predicates with quantified position variables, erase all negations, erase dangling local connectives, replace each remaining HAS predicate with its position variable argument, and finally, replace the remaining $\wedge$ and $\vee$ with $\circledwedge$ and $\circledvee$ respectively. The safety condition on MCalc[5] guarantees this procedure yields a meaningful scoring plan.

Example 4 (Scoring Plan):
The scoring plan for Q3 is obtained by following these steps. First remove the non-HAS predicates from Ψ:

$$(\text{HAS}(d, p_0, \text{'windows'}) \wedge \text{HAS}(d, p_1, \text{'emulator'})) \wedge \Big([\text{HAS}(d, p_2, \text{'free'}) \wedge \text{HAS}(d, p_3, \text{'software'})] \vee \text{HAS}(d, p_4, \text{'foss'}) \Big)$$

Then replace $\text{HAS}(d, p_x, k)$ with p_x, and $\wedge$ and $\vee$ with $\circledwedge$ and $\circledvee$ to arrive at the scoring plan Φ: $(p_0 \circledwedge p_1) \circledwedge ([p_2 \circledwedge p_3] \circledvee p_4)$ ◇

Generic scoring support constrains the selection of column-wise subtables. In the absence of scoring, MCalc queries obey FO-logic equivalences. Query $q \wedge r$ is equivalent to $r \wedge q$ because $\wedge$ is commutative. These two FO-equivalent queries result in the scoring plans $q \circledwedge r$ and $r \circledwedge q$, that are equivalent only when $\circledwedge$ is commutative for the selected scoring scheme. Similar issues occur with associativity, commutativity, and monotonicity of both $\circledwedge$ and $\circledvee$.

Since query equivalence is based on properties of FO-logic and scoring plan equivalence is not, a question arises: from which of the many potential syntax trees for Ψ is Φ derived? The matching plan is obtained from a syntax tree derived using standard FO-logic equivalences. The scoring plan is obtained from a syntax tree derived using the properties (see Section 5.1) of the selected scoring scheme. The decoupling of the Matching and Scoring Algebras thus allows an optimizer to reorder joins (using the flexibility of FO-logic equivalence) even when a rigid scoring scheme is selected that does not allow reordering of the score aggregation operators.

Example 5 (Score Computation):
In this example we walk through the process of computing a score using MEANSUM. Score computation starts from the match table. Figure 2 shows the match table used in this example, which contains matches in document d_w (introduced in Section 2) for Q3.

We score the match table by aggregating the scores for its subtables. For the example, we choose column-wise subtables until only single-column subtables remain. We first compute the scoring plan for Q3: $\Phi = (p_0 \circledwedge p_1) \circledwedge ([p_2 \circledwedge p_3] \circledvee p_4)$. Each individual column is a column-wise subtable that must be first scored by aggregating the initial scores of its rows. We show how column p_4 is scored.

Column p_4 has four alternate position values: [179, ∅, 179, ∅]. We split p_4 into two, then four row-wise subtables: [([179], [∅]), ([179], [∅])]. Each new subtable is a single, uninitialized cell; we continue the example by showing how these cells are initialized.

The first and third single-cell subtables both contain term position 179. α uses statistics from 179's index record (Figure 1, Section 2): $\langle \mathit{Offset}{=}179, \#\mathit{Docs}{=}2044, \#\mathit{InDoc}{=}1, \dots \rangle$. α also requires two document parameters, which are $d_w.\mathit{length}{=}207$ and $d_w.\mathit{collectionSize}{=}4{,}638{,}535$. Finally,
$\alpha(d_w, p_4, \langle 179, \dots \rangle) = \langle \frac{1}{207} \times \frac{4638535}{2044}, 1 \rangle = \langle 10.96, 1 \rangle$.

The second and fourth single-cell subtables each have the empty term position ∅. Based on the α implementation for MEANSUM, $\alpha(d_w, p_4, \varnothing) = \langle 0.0, 1 \rangle$.

We next aggregate the initial scores of the alternate positions in column p_4 using the alternate combinator:

$$(\langle 10.96, 1 \rangle \oplus \langle 0, 1 \rangle) \oplus (\langle 10.96, 1 \rangle \oplus \langle 0, 1 \rangle)$$
$$= \langle 10.96, 2 \rangle \oplus \langle 10.96, 2 \rangle$$
$$= \langle 21.92, 4 \rangle$$

The scores of the other columns are computed similarly, bringing us back to the computation for the whole match table which is completed with the conjunctive and disjunctive combinators:

$$(\langle 8.156, 4 \rangle \circledwedge \langle 32.38, 4 \rangle) \circledwedge ((\langle 0.134, 4 \rangle \circledwedge \langle 2.498, 4 \rangle) \circledvee \langle 21.92, 4 \rangle)$$
$$= \langle 40.536, 4 \rangle \circledwedge (\langle 2.632, 4 \rangle \circledvee \langle 21.92, 4 \rangle)$$
$$= \langle 40.536, 4 \rangle \circledwedge \langle 24.552, 4 \rangle$$
$$= \langle 65.086, 4 \rangle$$

Finally, the finalizer function ω computes the final, normalized document score from the aggregated score:
$\omega(d, \langle 65.086, 4 \rangle) = 1 - \frac{1}{ln(\frac{65.086}{4} + e)} = 0.660.$ ◇

4.2.2 Scoring Directionality

Scoring directionality has to do with whether row-wise subtables or column-wise subtables are selected first. Some scoring schemes are sensitive to this choice. There are two simple scoring patterns: row-first, and column-first. *Row-first* scoring involves first computing the score of each row, and combining those scores. For row-first scoring, row-wise subtables are always chosen when a match table has more than one row. *Column-first* scoring involves first computing the score of each column, and combining those scores. For column-first scoring, column-wise subtables are always chosen when a match table has more than one column. In *hybrid scoring* row-wise and column-wise score aggregation may be interleaved.

Many scoring schemes are *directional*; they will compute different scores under row-first aggregation than under column-first aggregation. When a scoring scheme is directional, one direction (row-first or column-first) is 'correct' – in that it is the intention of the scoring scheme designer – and hybrid scoring is incorrect.

Example 6 (Scoring Directionality):
Directionality is easy to illustrate with a simple example. Consider the match table M^{bool} of conjunctive query $a \wedge b$, where scores are either T (true) or F (false). We define $\circledwedge := \wedge$, and $\oplus := \vee$. If we choose row-first aggregation, then the score computation is shown as s_r. Similarly, s_c shows column-first aggregation.

$M^{bool} :=$

a	b
T	F
F	T

$$s_r := (T \wedge F) \vee (F \wedge T) = F \vee F = F$$
$$s_c := (T \vee F) \wedge (F \vee T) = T \wedge T = T$$

Scores computed by row-first and column-first aggregations are different even in this overly-simplistic scenario. ◇

Score evaluation must respect the appropriate pattern for directional scoring schemes or computed scores will be wrong.

Diagonal scoring schemes compute the same score using a row-first, column-first, or hybrid scoring pattern, and are highly desirable because they provide the optimizer more flexibility. Some optimizations discussed in Section 5 only work for diagonal schemes. Several scoring schemes we study in Section 7 are diagonal.

Definition 3 (Diagonal Scoring). *A scheme is diagonal iff the following properties hold:*

$$(w \owedge x) \obar (y \owedge z) = (w \obar y) \owedge (x \obar z)$$
$$(w \ovee x) \obar (y \ovee z) = (w \obar y) \ovee (x \obar z)$$

4.3 Integrating Matching & Scoring Algebras

We now introduce GRAFT (Generic Ranking Algebra for Full Text), which integrates MA and SA. In GRAFT, the operators of the scoring algebra are *hosted* by π and γ operators (which in score-isolated plans are performed outside the matching subplan). Hosted aggregations are a familiar concept. The SQL query '*SELECT a+b as c, SUM(d) FROM foo GROUP BY a,b*' translates to a relational algebra expression where the `SUM` aggregate is hosted by a group-by operator and the addition `(a+b)` is hosted by a generalized projection. In GRAFT, $\obar$ is hosted by the group operator γ, while $\ovee$, $\owedge$, α and ω are hosted by projection π.

Plans produced by automatic MCalc-to-GRAFT translation are score-isolated (introduced in Section 2). *Score-isolated plans* consist of a matching subplan with no scoring operations that computes a match table, and a scoring portion (everything above the matching subplan) which computes the match table score. The optimizer starts with a *canonical* score-isolated plan, rewriting it iteratively.

There are two canonical score-isolated plans for any MCalc query which compute scores in a row-first (column-first) manner. Which one is used depends on the directionality of the selected scoring scheme as discussed in Section 4.2.2. Both plans share the same matching subplan. Canonical matching subplans use a right-deep join tree and join order follows the order of keywords in the query. Selections follow joins, and a sort follows selections. Plan 7 shows the matching subplan of the canonical score-isolated plan for Q3.

The scoring portion of the row-first canonical plan first scores each match table cell and row by evaluating α and the scoring plan Φ in the context of a π. It then aggregates the row scores using $\obar$ in the context of γ. Finally, it computes the final scoring using ω in the context of π. Plan 6 shows the scoring portion of the row-first canonical score-isolated plan for Q3.

The scoring portion of the column-first canonical plan first scores each match table cell by evaluating α in the context of π. It then uses $\obar$ in the context of γ to compute column scores. It then evaluates the scoring plan Φ to combine column scores, and computes the final scoring using ω in the context of π. Plan 5 shows the scoring portion of the column-first canonical plan for Q3.

5. OPTIMIZATION

Optimizing GRAFT queries involves interleaving matching and scoring to avoid materializing the entire match table, while computing the same answers and scores as the canonical score-isolated plan. The performance of GRAFT queries stems from the size of the match table – a crucial intermediate result in score-isolated plans. We base our semantics on the match table for two reasons: it allows us to use an unrestricted, fully expressive set of full-text predicates; and we consider expressive ranking algorithms that use all of the matches as evidence for scoring. In the worst case, the match table is the cross product of the position list for each query keyword. Since the number of positions of each keyword scales linearly with the size of the data, the size of this cross product is $O(\mathcal{W}^{\mathcal{Q}})$ where $\mathcal{W}$ is the size of the library in words, and $\mathcal{Q}$ is the size of the query. Eager materialization of the match table, as in a score-isolated query plan, is a costly step, potentially incurring exponential complexity. Match tables are, however, only conceptual, they need not always be materialized, and certainly not eagerly.

In Section 5.1 we describe some design-time-specified properties of scoring scheme implementations that are relevant to the applicability of optimizations. In Section 5.2 we list useful relational optimizations, both classical and novel, and discuss how each is related to properties from Section 5.1.

5.1 Optimization-Relevant Properties

With respect to a selected scoring scheme, some optimizations are valid (preserve score consistency) or invalid (do not preserve score consistency). The optimizer must be able to discriminate and only apply valid optimizations. To do this, the optimizer needs to know some properties of each scoring scheme implementation. These properties are declared by the scoring scheme developer.

We keep the set of properties simple, small, and high level to reduce burden on the developer and improve maintainability across updates. The scoring scheme developer can specify the properties without understanding the workings of the optimizer, and changes in the optimizer will not break previously specified scoring schemes.

The set of specified properties includes basic mathematical properties of aggregation operators: associativity, commutativity, monotonicity, and idempotency of $\owedge$, $\ovee$, and $\obar$. If the scoring scheme is not diagonal (see Section 4.2.2), the developer must specify whether she intends the scoring to be row-first or column-first. Finally, we define here three additional properties that are useful for determining the validity of interesting optimizations.

A $\obar$ operator *multiplies* if a sequence of equal scores can be aggregated in one operation. Specifically, if there exists and operator $\otimes$, implementable in constant time, such that:

$$s_1 \obar s_2 \obar ... \obar s_k \equiv s_0 \otimes k \text{ where } \forall i,j : s_i = s_j$$

A scoring scheme is *constant* when all matches for a document have the same score, and the alternate combinator $\obar$ is idempotent. The constant property implies that finding one match for a document is enough to score the document. Specifically:

$$\forall m_1, m_2 : (matches(m_1, doc) \wedge matches(m_2, doc)) \rightarrow$$
$$(score(m_1) = score(m_2) = score(m_1) \obar score(m_2))$$

A scoring scheme is *positional* if term positions factor into scores. Specifically, a scoring scheme implementing α is positional iff:

$$\exists p_0, p_1, d, k : \textsc{Has}(d, p_0, k) \wedge \textsc{Has}(d, p_1, k) \wedge \alpha(p_0) \neq \alpha(p_1)$$

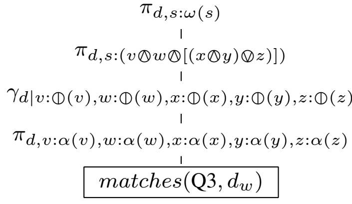
5.2 Study of Optimizations Under Scoring

We first relate some classical relational optimizations to the relevant properties an optimizer must consult to check validity of each optimization for a selected scoring scheme. Then we do the same for both existing and novel full-text-specific optimizations.

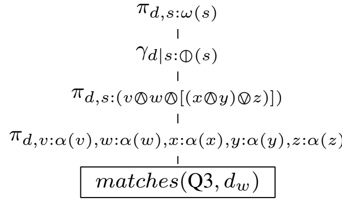
5.2.1 Classical Optimizations

Sort Elimination. Canonical GRAFT plans have a single sort operator which guarantees a well-defined order to matches in the match table. This order is necessary for scoring schemes where $\obar$ is non-commutative. When $\obar$ commutes, the order is irrelevant and the sort operator may be removed. If $\obar$ does not commute, sorting can sometimes be eliminated using classical techniques [24].

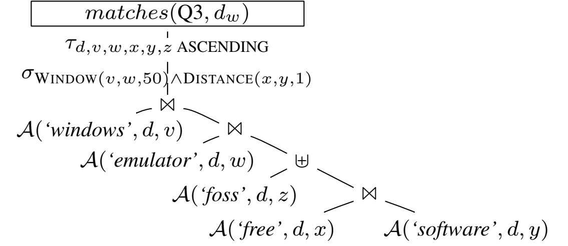
Selection Pushing & Join Reordering. Selection pushing and join reordering, are both textbook relational algebra optimizations. Since score aggregation in GRAFT is decoupled from join and selection operators, these optimizations are not prohibited by any scoring schemes. They must still be applied carefully because, for instance, some join orders are more amenable than others to optimizations that push score aggregation (described below).



Plan 5: Scoring portion of the column-first canonical score-isolated plan for Q3.



Plan 6: Scoring portion of the row-first canonical score-isolated plan for Q3.



Plan 7: Canonical matching subplan for Q3

Eager Aggregation. One way to avoid full materialization of match tables is to eagerly aggregate the matches in intermediate results by pushing group-bys down the plan. See Yan and Larson [33] for details on eager aggregation. Yan and Larson also studied eager aggregation in the context of generic aggregate functions [33]. We can directly map their aggregate function classifications to our properties and assert that eager aggregation is applicable when $\obar$ is fully associative. Additionally, and unique to our application, when the selected scoring scheme requires row-first scoring, γ operators hosting $\obar$ may not be pushed down through π operators hosting $\owedge$ or $\ovee$ (as doing so would violate the row-first requirement).

Eager Counting. Eager counting [33] is a technique that groups n *identical* tuples into a single tuple with a count value n. When two eagerly counted tuples join, their counts are multiplied. Counted tuples are expanded for aggregation, but otherwise eagerly counted tuples act as regular tuples in the system. The expansion step prior to aggregation can be avoided for score aggregation if the $\obar$ operator multiplies (see Section 5.1).

Zig-Zag Joins. The zig-zag join[15] is a special case of sort-merge join useful in fully streaming plans/subplans when each join attribute is indexed. The zig-zag join consumes its inputs by exploiting order to signal the index scan over one join attribute to skip directly to the value of the other join attribute. Zig-zag join signals the index scan operator even if it is several levels down the operator tree, thus bypassing large swaths of intermediate results from earlier joins. Zig-zag is a powerful join technique for GRAFT since every plan leaf is an ordered index scan.

In the context of relational full-text, zig-zag joins perform the same function as the skip pointers[18] commonly used in IR systems to accelerate inverted list intersection.

Rank Joins. Top-k optimizations speed up query execution by first exploring the documents that show the highest potential for a high score, and avoiding further exploration of lower scoring documents once the top-K are established. The relational rank-join[17] is a state-of-the-art algorithm for efficiently joining two rank-order tuple streams, producing a rank-order tuple stream.

Since MA uses a standard relational join, the rank-join may, under some circumstances, be used to implement a match join. The GRAFT rank-join hosts the $\owedge$ operator which it uses to combine the scores of joining tuples. Therefore, a rank-join may only be used where $\owedge$ may be pushed into the matching subplan. Additionally, rank-join only works with monotonically increasing ranking functions [17], so $\owedge$ must be monotonically increasing for rank-join to apply. A similar rank-union operator is also possible. Rank-union hosts the $\ovee$ operator, requiring it to be monotonically increasing.

5.2.2 Full-Text Specific Optimizations

Forward-Scan Joins. Botev et al. [7] identified a set of common full text predicates (PPREDS) that can be executed in a single forward pass over the index and can be used as join predicates to zig-zag joins. They developed an efficient evaluation plan for a full-text language restricted to PPRED. The language restriction enables the *forward-scan join* technique: a stateless zig-zag join that advances both its inputs in a forward-only manner. We discuss the complexity results for the PPRED algorithm in Section 6.

The forward-scan join may be used as a physical join operator in GRAFT queries, but only for very specific scoring schemes. Specifically, the scoring scheme must be constant (see Section 5.1) since the forward-scan join may miss some matches [5].

5.2.3 Novel Full-Text Optimizations

Alternate Elimination. For constant scoring schemes, alternate aggregation is unnecessary since the score of any match is the document score. In this case, group-by operators used to aggregate scores may be replaced by an alternate elimination operator δ_A:

$$\gamma_{A|B}(P) \equiv \delta_A \text{ when all aggregation functions in B are } \obar$$

The implementation of alternate elimination differs from group-by in two crucial ways: (1) it emits a new result match as soon as a new group is seen instead of waiting to see all group members, and (2) it signals its child operators to skip any further tuples in the group. Alternate elimination is similar to relational-algebra's duplicate elimination, except that matches (unlike tuples) are by definition never duplicates. Because matches behave like duplicates under a constant scoring scheme we can treat them as such, and in doing so we exploit the scoring scheme property.

Pre-Counting. Positions are not always used by a query. For example, consider a keyword k that is involved in no full-text predicates. Unless the positions are needed by the scoring scheme, they may be eliminated by the rewrite $\mathcal{A}(d,p,k) \rightsquigarrow \pi_d(\mathcal{A}(d,p,k))$. The projection creates duplicates, which may be eagerly counted by the rewrite $\pi_d(\mathcal{A}(d,p,k)) \rightsquigarrow \gamma_{d|c:\mathrm{COUNT}(*)}(\pi_d(\mathcal{A}(d,p,k)))$. In this case, the Match Factory, projection, and eager-counting group-by can be replaced by a much more efficient Pre-Counting Atomic Match Factory $\mathcal{CA}$:

$$\gamma_{d|c:\mathrm{COUNT}(*)}(\pi_d(\mathcal{A}(d,p,k))) \equiv \mathcal{CA}(d,p,k)$$

The novelty and efficiency of pre-counting is at the physical level: instead of the term-position index, $\mathcal{CA}$ scans a much smaller term-document index (a logical subset of the term-position index). Pre-counting yields significant performance gains over eager counting; in Section 8 we report a query with twenty-fold runtime speedup.

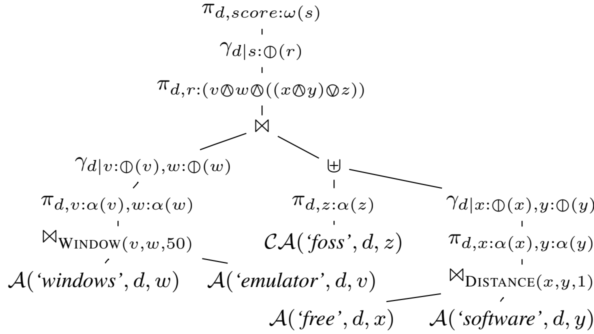
Since it forgets positions, pre-counting is valid only for non-positional scoring schemes.

5.2.4 Discussion of Optimizations

Table 1 summarizes specifically which scoring scheme properties are required by each optimization discussed in this section. One positive feature of Table 1 is that there are no restrictions on classical optimizations (join reordering, selection pushing, zig-zag joins, and eager counting). This fact is a direct consequence of our aggregation scoring model that decouples scoring from match computation as much as possible. As shown in Section 2, this is in contrast to state-of-the-art systems that encapsulate score computation in join and selection operators: they must either give up these optimizations or score-consistent generic ranking.

OPTIMIZATION	OPERATOR REQ.	DIRECTION REQ.
τ elim.	$\obar$ commutes	
$\bowtie$ reordering		
σ pushing		
zig-zag $\bowtie$		
forward-scan $\bowtie$	constant	
alt. elim.	constant	
eager agg.	$\obar$ fully associative	not row-first
eager count		
pre-count	non-positional	
rank-join	$\owedge$ monotonic increasing	diagonal
rank-union	$\ovee$ monotonic increasing	diagonal

Table 1: Each optimization listed can be applied when the selected scoring scheme satisfies the operator and direction requirements listed in the same row. Entries in this table are limited to properties that determine the optimization's correctness. An optimizer may consult other properties to assist optimization heuristics (e.g. eager count is not generally not helpful for positional scoring schemes).



Plan 8: An optimized evaluation plan for Q3 for a diagonal, non-positional scoring scheme.

Optimizations involving grouping and aggregation, show quite a few restrictions in Table 1. This is not a surprise, as $\obar$ is inseparable from the group-by operator. Four optimizations are described (in the middle row block of Table 1), each with different scoring scheme restrictions as well as different applicability within queries. Given their different applicability, these optimizations should be viewed as complementary.

Example 7 (Optimized Plan):
Plan 8 is a plan for Q3 showing various optimizations. The Pre-Counting Atomic Match Factory has been chosen for the keyword 'foss' since it is not involved in any predicates. Selections are pushed down into join predicates and joins are reordered. Group-by operators are pushed down beneath joins. Finally, the sort operator has been eliminated. $\diamondsuit$

In general, GRAFT's extensible design allows incorporating as optimizations various techniques from IR systems with rigid plan generation and/or restricted query language. GRAFT broadens the applicability of these techniques to queries and scoring schemes such systems do not support, by identifying the query subplans that are score-consistent with these techniques.

6. COMPLEXITY

The evaluation complexity of full-text languages has been extensively studied [7]. MCalc and MA have the same evaluation complexity as the similarly expressive FTC[7]: LOGSPACE-complete for the data and PSPACE-complete for the expression size. This result should not be a surprise. First-order relational calculus evaluation has the same data and expression complexities[31]. Despite this complexity, optimizers have made SQL and relational algebra practical even for very large datasets. The relational framework of GRAFT opens similar optimization opportunities for MCalc.

Lower complexity evaluation plans for restricted full-text languages have also been studied, but the restrictions placed on scoring generality for these plans have not been studied. Here we consider two such plans from [7] focusing specifically on the scoring scheme restrictions implied by the plans.

Languages in the class BOOL (no full-text predicates), have an evaluation plan in $O(\mathcal{D} \times \mathcal{Q}^2)$[7] where $\mathcal{D}$ is the number of documents in the library and $\mathcal{Q}$ the number of keywords in the query. This algorithm scans a term-document index instead of a term-position index. Because term positions are not scanned, positional scoring schemes [16, 25] cannot be used with this algorithm. The BOOL algorithm is simulated in GRAFT using the pre-counting optimization with the same restriction.

Languages in the class PPRED (discussed in Section 5.2.2) have an evaluation strategy that is $O(\mathcal{W} \times \mathcal{Q}^2)$[7] where $\mathcal{W}$ is the number of words in the collection. This algorithm is guaranteed to find a match in a document if one exists[7], but will not necessarily find all matches[5]. Because matches are missed, this algorithm is compatible only with scoring schemes that are constant and thus do not require all matches. The PPRED algorithm is simulated in GRAFT using forward-scan joins with the same restriction.

Clever evaluation techniques are crucial to low complexity plans for restricted languages, and for optimizing plans for expressive languages, but they must be applied carefully when using generic scoring. With some scoring schemes, the restricted language plans and similar optimizations become invalid. To illustrate how careful one must be, we point out that the paper which describes the efficient plans for BOOL and PPRED [7] also describes a scoring scheme (which we call Join-Normalized in Section 7) that is not score-consistent for either plan.

7. STUDY OF REAL SCORING SCHEMES

To validate the expressiveness of our Scoring Algebra, we considered scoring algorithms from the literature [7, 13, 16, 20, 22, 25, 27, 28, 29, 34]. From these, we identified seven appropriate scoring schemes that capture all the scoring algorithms. We implemented all seven schemes in our prototype and analyzed the set of scoring-relevant properties of the implementations. As shown in Table 2, even in our sample of scoring schemes there is significant variance in these properties. We combined our list of optimization requirements (Table 1) with the scoring scheme property analysis (Table 2) and obtained a list of rewrite optimizations that each scoring scheme allows (Table 3).

The schemes we implemented are as follows:

AnySum.

$\alpha(d, a, p)$: **return** $Bm25(d, p)$
$\owedge(s_L, s_R)$: **return** $s_L + s_R$
$\ovee(s_L, s_R)$: **return** $s_L + s_R$
$\obar(s_L, s_R)$: **return** s_L
$\omega(s)$: **return** s

AnySum is a scoring scheme typical of keyword-search systems that find a single match per document, and do not differentiate between different positions of a term. Thus all positions (including $\varnothing$) for a keyword have the same term weight, and consequently all matches to a document have the same score. AnySum defines the score of a document as the same as the score of its matches; the number of matches in a document does not factor into score. The score of a match is the sum of the BM25[18] (similar to tfidf) measures of the term positions that together form the match.

The scoring schemes we studied from Terrier[22] (DFR models), Timber[34] (as implemented for INEX) are instances of AnySum. Terrier also uses a similar scoring scheme for language model scoring where the score of a match is the product (vs sum) of the term position scores.

SumBest.

Extends AnySum; overrides:

$\alpha(d, a, p)$: **if** p **is** $\varnothing$: **return** 0.0 **else return** $Bm25(d, p)$
$\obar(s_L, s_R)$: **return max** (s_L, s_R)

SumBest is column-first, initializes the score of non-$\varnothing$ positions to BM25[18] and the score of $\varnothing$ to 0. It defines a column score as the maximum score in that column, and the document score as the sum of the column scores.

Lucene. Lucene [27] is a respected open-source keyword search engine with limited support for full-text predicates. This consists of the PROXIMITY and the phrase predicate, and, in contrast to GRAFT, allows no "plug-in" full-text predicates.

The scoring scheme used by Lucene goes slightly beyond the scoring model we present here. Specifically, for the proximity predicate, imperfect matches are allowed, whose scores reflect the divergence from the proximity parameter. Because we feel that no evaluation of our GRAFT prototype is complete without a comparison to Lucene, and because we didn't want to rule out proximity predicates in this comparison, we have implemented in our prototype an extension to capture this special matching behavior. We omit presentation of this extension because it is an ad-hoc solution to a more general problem: fuzzy matching. Fuzzy matching for MCalc queries is beyond the scope of this paper, and an interesting follow-up topic. Excluding the special handling of proximity predicates, the Lucene scoring scheme coincides with SumBest.

Join-Normalized Weighting.

$\alpha(d, a, p)$: **if** p **is** $\varnothing$: **return** $\langle 0.0, d.occurrences(a)\rangle$
  let $scr := TfIdf(d, p)$
  let $size := p.countInDoc$
  return $\langle scr, size\rangle$
$\owedge(s_L, s_R)$: **let** $scr := \frac{s_L.scr}{s_R.size} + \frac{s_R.scr}{s_L.size}$
  let $size := s_L.size * s_R.size$
  return $\langle scr, size\rangle$
$\ovee(s_L, s_R)$: **let** $size := (s_L.size * s_R.size) + s_L.size + s_R.size$

$$\textbf{let } src := \begin{cases} \frac{s_L.scr}{2} & \textbf{if } s_R.scr = 0.0 \\ \frac{s_R.scr}{2} & \textbf{if } s_L.scr = 0.0 \\ \frac{s_L.scr}{2*s_R.size} + \frac{s_R.scr}{2*s_L.size} & \textbf{else} \end{cases}$$

  return $\langle scr, size\rangle$
$\obar(s_L, s_R)$: **return** $\langle s_L.scr + s_R.scr, s_R.size\rangle$
$\omega(s)$: **return** $s.scr$

Join-Normalized weighting implements the scoring from [7] as discussed in Section 2, and a similar scoring scheme from [20]. When implemented in the GRAFT framework, the Join-Normalized scoring scheme does not have access to the size of intermediate results (because our scoring model does not include an explicit API to this statistic). To overcome this, the scoring scheme maintains the desired statistic in the *size* field of the internal score structure. As detailed in Section 2 the intermediate result size changes under optimizations. To obtain a well-defined score, we compute the size intermediate results would have in a canonical, score-isolated plan (i.e. the intermediate results are subtables of the match table).

Event Model.

$\alpha(d, a, p)$: **if** p **is** $\varnothing$: **return** 0.0 **else return** $Bm25(d, p)$
$\owedge(s_L, s_R)$: **return** $s_L * s_R$
$\ovee(s_L, s_R)$: **return** $s_L + s_R - (s_L * s_R)$
$\obar(s_L, s_R)$: **return** $s_L + s_R - (s_L * s_R)$
$\omega(s)$: **return** s

The probabilistic event model found in [13] and [29] treats the initial term weights as probabilistic events. The score of a match is the conjunction and/or disjunction of the term weights according to the scoring plan, using the standard inclusion-exclusion principle under the independence assumption. Finally, a document score is a disjunction of the scores to all matches.

	Any Sum	Sum Best	Lucene	Join Normal	Mean Sum	Event Model	BestSum +MinDist
directional		col				row	row
positional			✓[2]				✓
$\obar$ *associates*				left			
$\obar$ *commutes*	✓	✓	✓	✓	✓	✓	✓
$\obar$ *monotonic inc*		✓	✓	✓		✓	✓
$\obar$ *idempotent*	✓	✓	✓				✓
$\obar$ *multiplies*	✓	✓	✓	✓	✓	✓	✓
$\obar$ *constant*	✓						
$\owedge$ *associates*							
$\owedge$ *commutes*	✓	✓	✓	✓	✓	✓	✓
$\owedge$ *monotonic inc*	✓	✓	✓	✓	✓		✓
$\ovee$ *associates*							
$\ovee$ *commutes*	✓	✓	✓	✓	✓	✓	✓
$\ovee$ *monotonic inc*	✓	✓	✓	✓	✓	✓	✓

Table 2: Optimization-relevant properties of scoring schemes that we implemented in our prototype for our study.
[2] *Lucene is positional only for queries with phrase or proximity predicates.*

	Any Sum	Sum Best	Lucene	Join Normal	Mean Sum	Event Model	BestSum +MinDist
τ *elim.*	✓	✓	✓	✓	✓	✓	✓
$\bowtie$ *reordering*	✓	✓	✓	✓	✓	✓	✓
σ *pushing*	✓	✓	✓	✓	✓	✓	✓
zig-zag $\bowtie$	✓	✓	✓	✓	✓	✓	✓
forward-scan $\bowtie$	✓						
alt. elim.	✓						
eager agg.	✓	✓	✓	✓	✓		
eager count	✓	✓	✓	✓	✓	✓	✓
pre-count	✓	✓	✓	✓	✓	✓	
rank-join	✓		✓	✓	✓		
rank-union	✓		✓	✓			

Table 3: By combining Table 1 and Table 2 we derive the set of optimizations that may be consistently applied for each scoring scheme.

BestSum+MinDist.

$\alpha(d, a, p)$: **if** p **is** $\varnothing$: **return** $\langle 0.0, \infty, [\,]\rangle$
  let $scr := Bm25(d, p)$
  let $pos := [p.offset]$
  let $dist := MinDist(pos)$
  return $\langle scr, dist, pos\rangle$
$\owedge(s_L, s_R)$: **let** $scr := s_L.scr + s_R.scr$
  let $pos := s_L.pos@s_R.pos$
  let $dist := MinDist(pos)$
  return $\langle scr, dist, pos\rangle$
$\ovee(s_L, s_R)$: **return** $s_L \owedge s_R$
$\obar(s_L, s_R)$: **let** $scr := \mathbf{max}(s_L.scr, s_R.scr)$
  let $dist := \mathbf{min}(s_L.dist, s_R.dist)$
  return $\langle scr, dist\rangle$
$\omega(s)$: **return** $s.scr + \mathbf{log}(1 + e^{-s.dist})$

BestSum+MinDist uses the MinDist proximity measure from [25]. MinDist gives a high score to matches where two matching terms are very close, and a low score when no two matching terms are very close. MinDist over full-text is interpreted as applying to individual matches, since the proximity of keywords that do not occur in the same match is irrelevant. BestSum+MinDist computes the score of an individual match as the sum of the BM25 score of each term position in the match, multiplied by the MinDist metric. The score of a document is the score of its highest-scoring match. MinDist concerns term position so BestSum+MinDist is positional.

All the proximity measures found in [25] and the scoring schemes from [28, 16] are implemented similarly to BestSum+MinDist.

8. EXPERIMENTAL RESULTS

We report on experiments showing both that (a) our novel optimizations effectively improve query performance beyond classical optimizations and (b) despite additional overhead from generic scoring, GRAFT performance frequently exceeds state-of-the-art full-text search systems that do not implement generic scoring.

Our contributions regarding classical optimizations (join order, selection pushing, eager aggregation, eager count, zig-zag and rank-joins) are in building a framework to (correctly) exploit them, not their development. We do not validate their potential here.

Data. As a test dataset, we used a snapshot of the English Wikipedia from September 2010 [26]. We indexed the text from all articles, talk pages, disambiguation pages, and figure detail documents. The index constitutes 2.4 billion words (12 million unique terms) distributed over 5.2 million documents.

Queries. We give results with respect to eight queries formulated over Wikipedia, listed below in a shorthand syntax that is more concise than MCalc. Position variables are implicit. Keywords are conjuncted unless separated by a vertical bar. Quotes imply a PHRASE predicate. Other predicates are preceded by keyword arguments in parenthesis and followed by constant arguments in brackets. Q8 is the translation of MCalc query Q3 to this shorthand.

Q4. san francisco fault line
Q5. dinosaur species list (image | picture | drawing | illustration)
Q6. "orange county convention center" orlando
Q7. "san francisco" "fault line"
Q8. (windows emulator)WINDOW[50] (foss | "free software")
Q9. (free wireless internet)PROXIMITY[10] service
Q10. arizona ((fishing | hunting) (rules | regulations))WINDOW[20]
Q11. "rick warren" (obama inauguration)PROXIMITY[4] (controversy invocation)PROXIMITY[15]

Q4 and Q5 are simple boolean keyword queries, used as yardsticks to measure the overhead introduced by support for full-text predicates. Q6 and Q7 have phrase predicates which, from the full-text perspective, are syntactic sugar over a series of DISTANCE predicates. Q8 through Q11 have predicates typically only found in full-text search systems.

Besides Q8 our queries are non-artificial. (Q8 was constructed for the examples in earlier sections.) Q4 and Q7 were chosen at random from a friend's web search history (with permission). The rest are loosely based on topics from the TREC 2009 Web Track [10]. Web track queries are simple keyword search, not full-text search, so we rephrased topic descriptions into full-text queries.

Measurement Methodology. Each measurement was repeated nine times in succession, and we report the average of the five median times. This methodology was chosen to minimize the chance that a garbage collection or JIT event would occur during one measurement and not during another.

All systems tested cache index entries in RAM. Measurements are all taken on a warm cache; *no measured times include disk access*.

Platform. Experiments ran on a Phenom II 940 CPU, using the IcedTea6 1.8.1 JVM restricted to 4GB of RAM on Linux 2.6. Our GRAFT implementation is single-threaded.

Plans and Optimizer. Starting with a canonical plan, first the selection pushing rewrite is applied iteratively until the plan converges. Then either the eager aggregation or eager counting rewrite, is applied similarly. Eager counting is used when the scoring scheme is constant (in this case eager counting always performs better) or if the scoring scheme does not support eager aggregation. Plans used in experiments are listed in [5] to ensure repeatability. We expect a cost-based optimizer to outperform the heuristic optimization we used. Cost-based optimization is beyond the scope of this work.

Alternate Elimination. To measure the benefit of alternate elimination, we started with plans optimized as described above. We used the AnySum scoring scheme, and replaced each group-by operator with the alternate elimination operator per the equivalence listed in Section 5.2. We use AnySum for this experiment because it is the only scheme compatible with alternate elimination that we list in Section 7. We report the *execution time reduction*, i.e. the difference between unoptimized and optimized execution time as percentage of the unoptimized time. These results are the first (red) column of each cluster in Figure 3 (taller is better).

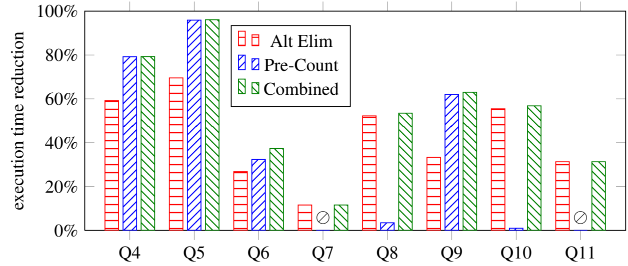


Figure 3: Execution time reduction provided by Alternate Elimination optimization, Pre-Counting optimization, and a combination of both over the classical eager count optimization.

Alternate elimination improves the performance of all eight queries. It is most effective, e.g. in Q5, when it replaces a group-by operator with moderate to large group sizes. In this case, a regular grouping operator waits for its inputs to produce every tuple in the group. The alternate elimination operator does not wait, rather it recursively signals operators in its input subplan to skip further members of the same group, speeding up the subplan.

Pre-Counting. To test pre-counting, we again started with plans optimized as described above for the AnySum scoring scheme, and then applied the pre-counting rewrite listed in Section 5.2 iteratively until the plan converged. The performance provided by pre-counting, measured in the same manner as alternate elimination, is shown in the middle (blue) bars in Figure 3. Pre-counting does not apply to Q7 or Q11 because they have no *free* keywords; all keywords in these queries occur in full-text predicate arguments.

Pre-counting substantially improves the performance of queries Q4 and Q5 since all of their keywords are free. Q6, Q9, Q8 and Q10 have one free keyword each. Q8 and Q10 do not benefit from pre-counting due to Amdahl's law: the free keywords represent only 3% and 2% of the positions scanned for the unoptimized Q8 and Q10, as opposed to 31% and 69% for Q6 and Q9 respectively.

Combined Effect. The third (green) bar of each cluster in Figure 3 is the combined evaluation time reduction of our two novel optimizations. Alternate elimination and pre-counting both produce results with a single tuple per document. Alternate elimination's effectiveness depends on having an input that produces multiple tuples per document. In Q4, Q5 and Q9 alternate elimination opportunities follow pre-counting; thus pre-counting minimizes alternate elimination's effect. In Q8 and Q10 alternate elimination opportunities follow full-text predicates, and consequently keywords are not pre-counted. An additive effect occurs for queries like Q6 with both predicate-free and predicate-full sub-plans.

Comparison to State-of-the-Art. Compared to many traditional IR search systems, our matching and scoring model require additional bookkeeping to support more expressive queries and generic ranking. We tested our system against two state-of-the-art IR systems Lucene[27] and Terrier[22] to show that the additional bookkeeping does not negatively impact our performance. We also tried to compare against Egothor[14] and FTA[7], but Egothor repeatedly failed to index our collection, and we were unable to obtain the FTA implementation.

Let us first note the difference in expressive power. Lucene and Terrier support only the PROXIMITY and PHRASE predicates.

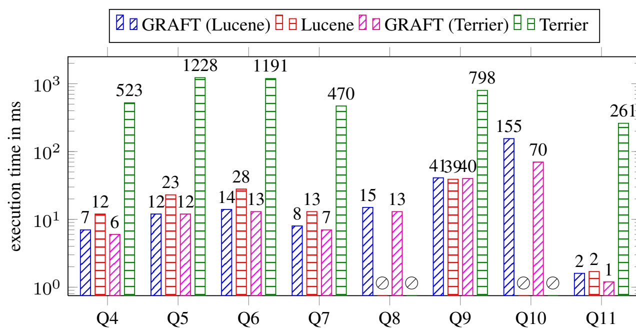


Figure 4: Execution time Lucene, GRAFT optimized for Lucene's scoring scheme, Terrier, and GRAFT optimized for Terrier's scoring scheme.

GRAFT additionally supports DISTANCE, ORDER, WINDOW, and can support as plug-ins virtually any predicate on positions. It can be easily extended to support such predicates as SAMESENTENCE or SAMEPARAGRAPH, assuming the index supports sentence and paragraph offsets. Of the seven scoring schemes we studied, Terrier only supports AnySum, and Lucene supports its own.

Figure 4 shows the comparative execution times for Q4 through Q11 on GRAFT optimized for Lucene's scoring scheme, Lucene, GRAFT optimized for Terrier's scoring scheme, and Terrier. In all cases, results are fast enough for interactive use. Properly optimized GRAFT plans run as fast, if not faster than both Lucene and Terrier. Lucene and Terrier do not support Q8 or Q10 because they do not support the WINDOW predicate.

It is worth noting that Lucene and Terrier are multi-year, multi-person projects tuned for traditional keyword and phrase search queries such as Q4 through Q7. Remarkably, GRAFT is very competitive against both systems even on such simple queries, and even using the systems' own custom scoring schemes. The fact that this holds despite the overhead stemming from GRAFT's support for higher query expressivity, score genericity, and score consistency, is due to unlocking the optimization potential allowed by each scheme.

9. CONCLUSIONS

This work is motivated by the observation that state-of-the-art algebraic execution of full-text queries creates a conflict between support for generic scoring on one hand, and score-consistent optimization on the other. To resolve this conflict, the same query must be optimized differently for different scoring schemes.

We show how to build an optimizer that takes as plug-in parameter a scoring scheme, and exploits its inherent potential for optimization without perturbing scores. Towards a well-defined scoring semantics, we first introduce an execution-independent scoring model based on the principles of *match-scoring* and *score isolation*. We show that our model supports score-consistency for classical relational algebra optimizations, IR techniques adapted as algebra optimizations, and even novel optimizations that cross the boundary between matching and scoring. We show that our scoring model can capture different interesting scoring algorithms from the literature [7, 13, 16, 20, 22, 25, 28, 29, 34] in a generic manner, and we describe a technique by which an optimizer can correctly optimize a plan given different scoring schemes.

Compared to state-of-the-art IR engines tuned for classical keyword and phrase search, GRAFT incurs overhead due to its support for more expressive full-text predicates and for a wider class of scoring schemes under score-consistency. However such overhead is more than made up for by the proposed optimizations, which yield an all-around competitive engine.

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient xml search with complex full-text predicates. In *SIGMOD*, 2006.

[3] Autonomy Corporation. Verity K2. http://www.verity.com.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1st edition, May 1999.

[5] N. Bales, A. Deutsch, and V. Vassalos. Score-consistent algebraic optimization of full-text search queries with graft: Extended technical report. Technical report, UC San Diego, http://db.ucsd.edu/graft/TechReportMar2011.pdf, 2011.

[6] R. Barrows and J. Traverso. Search considered integral. *ACM Queue*, 4(4), 2006.

[7] C. Botev, S. Amer-Yahia, and J. Shanmugasundaram. Expressiveness and performance of full-text search languages. In *EDBT*, 2006.

[8] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *SIGIR*, 1995.

[9] P. Case. Enhancing xml search with xquery 1.0 and xpath 2.0 full-text. *IBM Systems Journal*, 45(2), 2006.

[10] C. L. Clarke, N. Craswell, and I. Soboroff. Overview of the trec 2009 web track. Technical report, 2009, http://trec.nist.gov/pubs/trec18/papers/WEB09.OVERVIEW.pdf.

[11] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4), 1979.

[12] M. P. Consens and T. Milo. Algebras for querying text regions: Expressive power and optimization. *J. Comp. Sys. Sci.*, 57(3), 1998.

[13] N. Fuhr and K. Großjohann. Xirql: An xml query language based on information retrieval concepts. *ACM Trans. Inf. Syst.*, 22(2), 2004.

[14] L. Galambos. Egothor. http://www.egothor.org/, 2009.

[15] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, 2008.

[16] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD*, 2003.

[17] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3), 2004.

[18] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2007.

[19] Microsoft Corporation. Fast ESP. http://www.microsoft.com/enterprisesearch/en/us/fast.aspx.

[20] V. Mihajlovic, G. Ramírez, A. P. de Vries, D. Hiemstra, and H. E. Blok. Tijah at inex 2004 modeling phrases and relevance feedback. In *INEX*, 2004.

[21] R. Mukherjee and J. Mao. Enterprise search: Tough stuff. *ACM Queue*, 2(2), 2004.

[22] I. Ounis, G. Amati, P. V., B. He, C. Macdonald, and Johnson. Terrier Information Retrieval Platform. In *ECIR 2005*. Springer.

[23] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[24] D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, 1996.

[25] T. Tao and C. Zhai. An exploration of proximity measures in information retrieval. In *SIGIR*, 2007.

[26] M. Technologies. Freebase wikipedia extraction (wex). http://download.freebase.com/wex/, 2009.

[27] The Apache Foundation. Lucene. http://lucene.apache.org/.

[28] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. Topx: efficient and versatile top-*k* query processing for semistructured data. *VLDB J.*, 17(1), 2008.

[29] M. Theobald, R. Schenkel, and G. Weikum. Topx and xxl at inex 2005. In *INEX*, 2005.

[30] NIH. Pubmed: U.S. National Library of Medicine. pubmed.gov.

[31] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.

[32] Westlaw. http://www.westlaw.com.

[33] W. P. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.

[34] C. Yu, H. Qi, and H. V. Jagadish. Integration of ir into an xml database. In *INEX Workshop*, 2002.