Pathfinder – Full Text

or

Extending a Purely Relational XQuery Compiler with a Scoring Infrastructure for XQuery Full Text

Stefan Klinger

Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

an der Universität Konstanz
Mathematisch-Naturwissenschaftliche Sektion
Fachbereich Informatik und Informationswissenschaft

vorgelegt von Stefan Klinger

Tag der mündlichen Prüfung: 22. Dezember 2010
Referent: Prof. Dr. Marc H. Scholl
Referent: Prof. Dr. Torsten Grust

Stefan Klinger. Pathfinder–Full Text *or* Extending a Purely Relational XQuery Compiler with a Scoring Infrastructure for XQuery Full Text. University of Konstanz, November 2010.

Document compiled 2011-Jan-10 11:13:51

Abstract

This work demonstrates the extension of the purely algebraic XQuery compiler Pathfinder with an infrastructure for *implicit score propagation*. This is used to implement a subset of XQuery Full Text, employing the PF/Tijah index for Full Text search.

It is shown that a flexible framework for implicit score propagation can be implemented easily —*i.e.*, minimally invasive— on top of the Pathfinder compiler. The described prototype implementation can be parametrised with different scoring model functions, and should be adaptable to alternative database back-ends and Full Text engines.

At the same time, various systematic problems that arise from implicit score propagation are pointed out, rising the question whether such an approach is useful in general. Flaws in the design of the XQuery language are described that thwart more flexible extensions at the user level.

German: Diese Arbeit demonstriert die Erweiterung des rein algebraischen XQuery Übersetzers Pathfinder um eine Infrastruktur für die implizite Weiterleitung von Bewertungen (*score propagation*). Diese wird verwendet um, unter Verwendung des PF/Tijah Indexes für die Volltext-Suche, eine Teilmenge von XQuery Full Text, zu implementieren.

Es wird gezeigt, dass ein flexibles Rahmenwerk für die implizite *score propagation*, minimal-invasiv auf dem Pathfinder Übersetzer aufbauend, implementiert werden kann. Die hier beschriebene Implementation eines Prototyps kann mit Funktionen unterschiedlicher Bewertungsmodelle parametrisiert werden, und sollte an unterschiedliche Datenbank- und Volltext-Backends anpassbar sein.

Gleichzeitig werden diverse prinzipielle Probleme herausgearbeitet, die sich bei impliziter *score propagation* stellen. Damit wird auch die Frage gestellt, ob solch ein Ansatz überhaupt sinnvoll ist. Schwächen im Design der XQuery Sprache, die flexiblere Erweiterungen auf Benutzerebene vereiteln, werden beschrieben.

Contents

1 Motivation and Overview **9**

1.1 Extrinsic Motivation . . . 9

1.2 Intrinsic Motivation . . . 10

2 Introduction to XQuery Full Text **13**

2.1 What is XQuery Full Text? . . . 13

2.2 Syntax . . . 14

2.2.1 The horizontal language stack . . . 14

2.2.2 How Full Text interacts with XQuery . . . 16

2.3 What is a score? . . . 20

2.3.1 The "second-order aspect" . . . 20

2.3.2 Where are the scores? . . . 23

2.3.3 Semantics of scores . . . 25

2.4 Neither Tuple, nor Record, nor Class . . . 26

2.4.1 Tuples . . . 26

2.4.2 Overloading with Typeclasses . . . 27

2.4.3 Overloading with Records or Objects . . . 27

3 Related Work **29**

4 The Compiler **35**

4.1 Intermediate Languages . . . 35

4.1.1 XQuery Full Text and XQuery Core . . . 35

4.1.2 Relational Algebra . . . 36

4.1.3 NEXI . . . 36

4.2 The Pathfinder compiler . . . 36

4.2.1 Basic XQuery data structures . . . 38

4.2.2 Pathfinder's XML encoding . . . 38

4.2.3 PATHFINDER's XQUERY item sequence encoding 39
4.3 PATHFINDER^{FT} as a compilation phase 41

5 The Compilation Rules **43**
5.1 Notation and Relational Algebra operators 43
5.2 Compilation Framework 45
5.2.1 Fragments 46
5.3 Literals 46
5.4 Variables 47
5.5 Sequences 47
5.6 The `let`-clause 48
5.7 The `for`-clause 49
5.8 Axis steps 50
5.9 Direct score manipulation 51
5.10 Boolean operators 52
5.11 Conditional expression 54
5.12 Node set operations 55
5.12.1 Union 56
5.12.2 Intersection 56
5.12.3 Difference 56
5.13 The function `fn:exists()` 57
5.14 Other built-in functions 57
5.15 Quantified expressions 58
5.16 Predicates 59
5.16.1 Boolean Predicates 59
5.16.2 Existential Predicates 59
5.16.3 Positional Predicates 60
5.17 General comparison 60
5.18 Accessing XML structures 60
5.19 Element construction 61
5.20 Using Documents 64
5.21 Calling the Full Text machine 64
5.21.1 Purely algebraic 65
5.21.2 Relational Algebra function call 65
5.22 Compiling Full Text expressions 67
5.22.1 The direct approach via XML 68
5.22.2 The direct approach via NEXI 69

5.22.3 Variable search terms . . . 70
5.22.4 Unfolding Full Text expressions . . . 71
5.22.5 Variable search terms, again . . . 75
5.22.6 The limits of unfolding . . . 75
5.23 Scoring model parameters . . . 80

6 The Prototype Implementation **95**
6.1 Goals & Achievements . . . 95
6.2 Why Haskell? . . . 96
6.3 Architecture . . . 96
6.4 History of Development . . . 97
6.5 Query data structures . . . 98
6.6 Plan data structures . . . 99
6.6.1 The DAG structure . . . 100
6.6.2 Monadic DAG construction . . . 101
6.7 Compilation . . . 103
6.7.1 Fragment handling . . . 103
6.7.2 Direct score manipulation . . . 103
6.7.3 Sequences . . . 104
6.7.4 Axis steps . . . 105
6.7.5 Pragmas control score propagation . . . 106

7 Future Work **109**
7.1 Performance testing . . . 109
7.2 Non-determinism . . . 110
7.3 Other interpretations of Score . . . 111
7.4 Avoid locality . . . 111

8 Lessons Learnt **113**

Chapter 1

Motivation and Overview

1.1 Extrinsic Motivation

Clearly, XML is *en vogue*, be it as data exchange format on the web, so called "human readable" configuration files, or programming languages (think XSLT), or, actually, markup in documents. No matter what use case, someone will come up with an XML-ified version of it. Thus, naturally, the amount of XML data grew (and is still growing) to an amout making the need for XML databases obvious. Not surprisingly, XML is particularly well suited to encode documents (*i.e.*, semi structured data) since this domain is where its ancestor SGML originates from. And due to its hierarchical structure, a single XML tree naturally hosts collections of documents, and libraries of collections of documents...

With library-scale document collections, stored in a databases, comes the need for information retrieval (IR), *i.e.*, queries no longer follow the traditional database style "give me those things x from database Y with exactly the property p!", but rather a more vague scheme: "What book is about z?", or even just "z".

But not only the kind of asking changed. An XML database may (depending on the query) decide at which granularity (see [18]) the query is answered. Due to the hierarchical concept (potentially storing a complete library below a single root node) there is no need to focus on tuples, or documents, as the retrieval unit. Without request by the user, there is not even justification for doing so. Hence, among the set of "important" items returned, one may find letters, chapters, paragraphs, or even drawings (think SVG).

Several challanges arise from this vison: Store large XML instances, allow for access at node-granularity, and, by adding IR, determine what is relevant, and at which granularity (is a book relevant just because it contains a relevant paragraph?). But also, and tightly coupled with these: Find means to express a query. In other words, a language is necessary that allows the expression of such queries.

For a plain (*i.e.*, non-IR) setting, XQuery seems to have established as a *de facto* standard. It allows for precise navigation in the tree structure of an XML instance, iteration over node sequences, predicates, conditionals, *etc.*, thus answering the language question for two of the above challanges. Furthermore, there are mature implementations of XML databases featuring XQuery as a query language. The Pathfinder/MonetDB couple is the one this thesis builds on.

For the IR setting, things are more in motion. NEXI is one of the more well-known approaches to find a Full Text query language. One implementation of a NEXI system is of special interest for this thesis. The authors of [13] manage to couple their Full Text index (named TIJAH) with the PATHFINDER/MONETDB system mentioned above: An important step to do, since the expressiveness of NEXI is quite limited.

The PF/TIJAH system embeds NEXI queries as plain strings in XQUERY queries, and uses special builtin functions to run the TIJAH engine on these queries, which return scores, or sequences of nodes ordered by relevance. The missing gap in the PF/TIJAH system is that the NEXI *strings* appear as a black box to the compiler. The problem is that XQUERY provides no means to express a Full Text query. In other words, there is means to evaluate vague queries in an XQUERY setting, but no language to sanely express them.

XQUERY FULL TEXT is an extension to the XQUERY language that aims to solve exactly this problem: It extends the XQUERY language with a full-blown IR language.

1.2 Intrinsic Motivation

When I started looking for a PhD topic, the combination of XQUERY and Full Text was still in its infancy, and there was virtually no XQUERY FULL TEXT engine available. Except, of course, GALATEX [7], which came along with the XQUERY FULL TEXT draft, not to say that the XQUERY FULL TEXT draft looks like the documentation of what was done building GALATEX.

With emerging XQUERY FULL TEXT and a connection to the PATHFINDER people (Torsten Grust mentored my diploma thesis, and it was him who started research (see [9]) on what later led to the PATHFINDER project at Marc H. Scholl's chair in Konstanz, who, in turn, hosts me as a PhD student currently), it seemed a reasonable project to find out how the high-performance XQUERY compiler could be extended to digest XQUERY FULL TEXT. Naturally, the environment and my own perspective led to a more DB-ish understanding of the XQUERY FULL TEXT language, as opposed to what IR folks might expect. See Section 2.1 for a discussion of the tension between IR and DB semantics of a query language.

Long before starting to think about Full Text, during a stay at Universiteit Twente in 2005, I learned to know Vojkan Mihajlović, Djoerd Hiemstra, and others, working on information retrieval over XML documents using the NEXI language.

Figure 1 on page 11 tries to depict the various ways of how their work inspired the development of PATHFINDER^{FT}, although only the most influential events are shown. Chapter 3 gives a more thorough overview of related work.

Clearly, the beginning was contact with PATHFINDER [9, 10, 11], and an understanding of how it manages to perform XQUERY on RDBMSs. The Score Region Algebra introduced by [17] gave an impulse to offer implicit score propagation via abstract functions. After some development, the integration of PATHFINDER and TIJAH in the PF/TIJAH project [14] gave rise to the idea of using the TIJAH index as a scoring engine for XQUERY FULL TEXT.

At the time where the early PATHFINDER^{FT} looked like a reasonable approach, it became obvious that the original PATHFINDER compiler had developed quite a lot from the point I had used as foundation for my work. Also, since I never anticipated to rebuild the complete PATHFINDER compiler, but only the most basic core suitable to host the intended extensions instead, my implementation suffered several shortcomings. In other words: The prototype implemented until then was far from able to actually "run on" the desired back-end. A lot of discussion with the

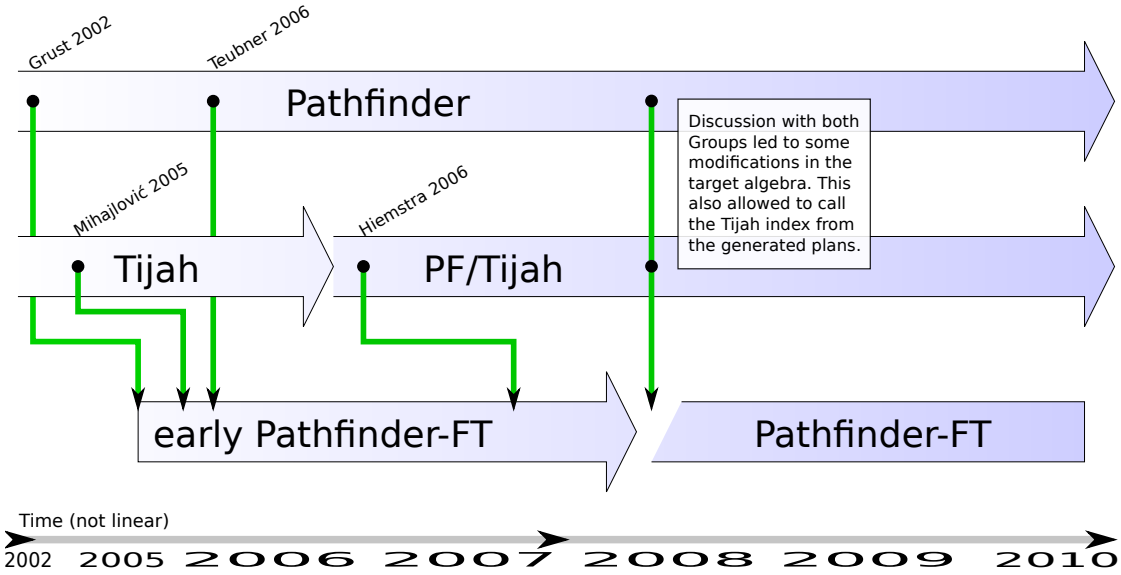


Figure 1: A rough representation of history. The horizontal, shaded bars represent the different projects. Green arrows indicate landmarks in the development that heavily influenced and inspired the development of PATHFINDER^{FT}.

PATHFINDER group helped in adapting PATHFINDER^{FT} to the "real world" PATHFINDER to a reasonable extent. Also, a great deal of communication with the PF/TIJAH group finally allowed me to use their index as scoring back-end for PATHFINDER^{FT}.

Although PATHFINDER^{FT} still lacks a lot of XQUERY, and XQUERY FULL TEXT functionality, it is now a reasonably complete, and runnable proof of concept.

* * *

The remainder of this thesis is organised as follows: **Chapter 2** gives an overview of what XQUERY FULL TEXT actually is, and what challenges arise from extending XQUERY with Full Text in an orthogonal way. This also shines a light on the tension between IR and DB languages, criticising the perception of scores and second-order semantics promoted by the XQUERY FULL TEXT designers. The need for implicit propagation is pointed out. Also, some design choices made in the development of XQUERY, and their rather ugly consequences for XQUERY FULL TEXT are shown. **Chapter 3** outlines related work, before **Chapter 4** gives a more deep description of the PATHFINDER compiler, and the desired extensions. The overall architecture of the PATHFINDER^{FT} compiler is explained here. Finally, **Chapter 5** formally describes the compilation rules used by the PATHFINDER^{FT} compiler to compile XQUERY FULL TEXT to Relational Algebra. The focus is on explaining the extensions made in comparison to the original PATHFINDER compiler. This chapter also points out some of the pitfalls earned from implicit score propagation. **Chapter 6** allows some insights into how the PATHFINDER^{FT} prototype was actually implemented.

Chapter 2

Introduction to XQuery Full Text

2.1 What is XQuery Full Text?

What is XQuery Full Text? Is it a DB or an IR query language? The question boils down to decide what the semantics of an XQuery Full Text query actually is.

In the DB world, a query has semantics formally defined by some more or less simple rules, with a strictly defined set of operators, and their behaviour (the algebra). Together with a bunch of laws that hold for the algebra and that can be exploited for query optimisation. Databases contain structured data, be it relations in the case of an RDBMS, or trees in the case of an XML database.

The IR community takes a more vague approach, some even deny the existence of query semantics that can be derived by a simple parser, or reflected by an ordinary abstract syntax-tree (AST). Put provokingly, guessing and satisfying the user's *information need* is more important to the IR community, than providing means for algebraically correct transformations of a query plan.

The following quote from the makers of the NEXI language (see [24]), describing the difference between XPath and NEXI, makes the tension between a DB and an IR perception of a query language quite clear:

> *The most significant diversion from XPath is semantics. Whereas in XPath the semantics are defined, in NEXI the retrieval engine must deduce the semantics from the query. This is the information retrieval problem – and to do otherwise is to make it a database language. For clarity, strict and loose interpretations of the syntax are included herein, however these should not be considered the only interpretations of the language.*

The perspective of this thesis is to understand XQuery Full Text as an orthogonal extension of XQuery with a Full Text sub-language as described in Section 4.1. *I.e.*, the XQuery portion of the language retains its strict DB semantics, and the Full Text portion can have arbitrarily vague semantics. In fact, the evaluation of the Full Text expressions is not part of this thesis. Instead, different compilation techniques are described to let an IR system do its job on the Full Text expressions, and to hand its findings back to the DB system formed by the Pathfinder/MonetDB couple.

This separation of semantic domains is also described in Section 2.3.1.3: While the *structure* of the Full Text portion of a query may be relevant to its outcome, the structure of the XQuery portion remains impervious to the Full Text engine.

2.2 Syntax

2.2.1 The horizontal language stack

XQuery Full Text is an extension of XQuery with Full Text features. This paragraph presents a notion of XQuery as an interleaving of multiple sub-languages with different focuses. This is referred to as the *horizontal language stack* in this thesis, as opposed to the *vertical language stack* discussed in Section 4.1. Then I will present how XQuery Full Text can be understood as an extension of XQuery with a Full Text language.

2.2.1.1 XQuery

What is the gist of XQuery? If we leave out path expressions and node construction, the remaining thing is a language to work with sequences of items of sorts, binding them to variables, iterating over, filtering, sorting them, offering flow control and function definitions. There is not much else one can do. But after all, XQuery is not XQuery without path expressions and node construction. These are discussed in the following.

2.2.1.2 XML

XML not only is the language used to describe the documents in our database, it can also be used to construct XML fragments in the XQuery language. To allow for an interleaving of XQuery and XML expressions though, the root of the XML grammar does not appear as a terminal symbol in the XQuery grammar. Instead, the XQuery grammar introduces a DirectConstructor[1] to construct XML nodes.

```
DirectConstructor   ::=  DirElemConstructor | ...
DirElemConstructor  ::=  "<" QName DirAttributeList
                         ("/>" | (">" DirElemContent* "</" QName S? ">"))
```

The syntax of such constructions is deliberately based on XML syntax, with one mayor difference: These "XML" expressions may contain an EnclosedExpr[2] to construct CommonContent[3], which can be seen as an "escaping" construct to close the circle back to XQuery.

```
EnclosedExpr   ::=  "{" Expr "}"
CommonContent  ::=  ... | EnclosedExpr
```

The result of a node construction is an item in the XQuery data model.

[1] http://www.w3.org/TR/xquery/#prod-xquery-DirectConstructor
[2] http://www.w3.org/TR/xquery/#prod-xquery-EnclosedExpr
[3] http://www.w3.org/TR/xquery/#doc-xquery-CommonContent

2.2.1.3 XPath

XPath is a language to navigate within XML documents[4], or more precisely, to address arbitrary parts of an XML document. As with XML above, XQuery does not directly use XPath as a terminal symbol in its grammar. Instead, the production of PathExpr[5] introduces axis steps as known from XPath, and the production for Predicate[6] leads back to XQuery expressions. A path expression can be seen as a function mapping an XQuery item sequence containing the context nodes to an item sequence containing the result nodes.

* * *

As said, the XQuery grammar does not *include* the XPath or XML grammars. It is rather constructed deliberately so that the user *has the impression* to embed plain XML/XPath code into XQuery, and to embed XQuery into those embedded XML/XPath expressions.

```
for $b
in doc("library.xml")//book
return <book>
         { $b/title }
         <authors>{ $b/author }</authors>
       </book>
```

In this example "XPath" is used for addressing book, title, and author nodes, "XML" is used for construction of result tuples, and XQuery is used for the looping and for gluing together the other expressions.

2.2.1.4 Adding Full Text

Now XQuery Full Text, as an extension of XQuery, can be seen as the addition of another sub-language (referred to as the Full Text language in this thesis), and means to interleave Full Text with XQuery.

In the grammar, the production FTContainsExpr[7] introduces the Full Text language on its right hand side, through the non-terminal symbol FTSelection[8]. This expression specifies the conditions of a full-text search[9] within a search context specified by the RangeExpr[10] on left hand side of the `contains text` keyword.

FTContainsExpr	::=	RangeExpr ("`contains`" "`text`" FTSelection FTIgnoreOption?)?
FTSelection	::=	[production of the Full Text language]
RangeExpr	::=	[this is plain XQuery]

FTIgnoreOptions are currently not handled by PathfinderFT, but would add another parameter to be passed to the Full Text engine, which is expressed in XQuery.

[4] `http://www.w3.org/TR/xpath/>`
[5] `http://www.w3.org/TR/xquery/#prod-xquery-PathExpr`
[6] `http://www.w3.org/TR/xquery/#prod-xquery-Predicate`
[7] `http://www.w3.org/TR/2010/CR-xpath-full-text-10-20100128/#prod-xquery10-FTContainsExpr`
[8] `http://www.w3.org/TR/2010/CR-xpath-full-text-10-20100128/#prod-xquery10-FTSelection`
[9] `http://www.w3.org/TR/2010/CR-xpath-full-text-10-20100128/#ftselection`
[10] `http://www.w3.org/TR/2010/CR-xpath-full-text-10-20100128/#prod-xquery10-RangeExpr`

The primary search terms can be expressed as literals, or as XQuery expressions, which is implemented by the production FTWordsValue[11], thereby embedding XQuery in the Full Text language.

```
FTSelection   ::=  [via some productions, uses FTWordsValue]
FTWordsValue  ::=  Literal | ("{" Expr "}")
```

This is not the only opportunity where this embedding takes place, *e.g.*, following the `weight` keyword an integer value is expected, which is also expressed by means of XQuery.

The Full Text language offers various ways to further specify the role of a primary search term in a search. Aspects of its compilation are shown in Section 5.22. The exact semantics and evaluation of the (compiled) Full Text expressions is not part of this thesis, but instead depends on the Full Text engine used.

* * *

The proposed partitioning of the XQuery language in the above paragraphs is not purely artificial. Chapter 4 will show more clearly that the different sub-languages correspond to different, although related, compilation strategies and, finally, to different concepts in the back-end.

Within the scope of this thesis, *i.e.*, with respect to the Pathfinder compiler, the following relations exist:

- XML corresponds to document storage and *twigs* used by the Pathfinder compiler to efficiently handle node construction.
- XPath corresponds to the XPath accelerator [9].
- XQuery corresponds to loop-lifted relational item representation.
- Full Text corresponds to PF/Tijah, *i.e.*, the proposed approach expects all Full Text expressions to be evaluated by a Full Text machine that is available to the database back-end.

2.2.2 How Full Text interacts with XQuery

In contrast to XML or XPath sub-expressions, there are *two* points in the syntax, rather than one, where the Full Text language interacts with the query.

One, of course, is the place where the Full Text expression is embedded, *i.e.*, the predicate in the following query.

```
$doc/book[./author contains text "John"]
```

However, and this makes the extension offered by XQuery Full Text special, the Full Text expression does not simply return a Boolean, it also "returns" a score that remains hidden at first. Only the use of the `score` keyword reveals the score and binds it to a variable, say `$s`, which identifies the second syntactic location of interaction between XQuery and Full Text.

```
let score $s := $doc/book[./author contains text "John"] return $s
```

[11] http://www.w3.org/TR/2008/CR-xpath-full-text-10-20080516/#prod-xquery-FTWordsValue

These two extensions to the XQuery syntax are connected only semantically, not on the syntax level, *i.e.*, not explicitly by a form visible to the user. The following paragraphs elaborate on this.

2.2.2.1 Invoking the Full Text machinery

The operator `contains text` introduces a Full Text query language as its second argument, which —applied on `contains text`'s first argument, a plain XQuery expression[12] determining the search context— yields a Boolean together with a score.

```
$lib/book[./title contains text "Hitchhiker" ftand "Guide"]
```

The Full Text language on the right hand side introduces special Full Text operators, such as, *e.g.*, `ftand`, `ftor`, `ftweight`, `ftallWords`, etc., which do not coincide with XQuery operators of similar name. More on the distinction between XQuery and the Full Text language is explained in Section 2.2.2.3.

From the user's perspective, scores come into existence only by using the `contains text` keyword in an XQuery Full Text query. However, they are not accessible by means of the XQuery language itself, because "the Value" returned is a Boolean. The score lurks behind the syntax.

Using only XQuery syntax to operate on the findings of the Full Text machinery will not make any use of the score, *i.e.*, the query above will return those books from the library, whose title is considered by the Full Text machinery to fulfil the requirement `"Hitchhiker" ftand "Guide"`.

However, the Full Text machinery also creates scores that describe "how well" the Full Text requirements are fulfilled by the queried element — although other interpretations of the score are very well possible, see Section 2.3.3. The XQuery Full Text specification [1] restricts this score to a floating point value in the range $[0, 1]$, but other values are thinkable — PF/Tijah actually does use scores outside this range. This score is *attached* to the Boolean returned by the `contains text` operator. This coupling is so tight, that the Pathfinder$^{\text{FT}}$ compiler actually takes the scored Booleans as *pairs of a Boolean and a score*.

There is a clear distinction between these pairs, and XQuery's *item sequences* such as `(true(), 0.3)`: The former are proper pairs of an XQuery Boolean (later relaxed to XQuery singleton items), and a score defined *independent* of XQuery's data model, *i.e.*, XQuery provides no means to instantiate such pairs. The latter resemble lists, XQuery's major data structure, where the list with exactly one element is indistinguishable from the element alone (modulo type), and that hence cannot be nested.

From XQuery's point of view, the scored Boolean is nothing but a Boolean, and consequently, XQuery provides no means to access the scores. This gives rise to the second syntactic extension:

[12] Of course, due to orthogonality, XQuery Full Text expressions would be allowed as well to determine the search context. This further complicates matters, and is discussed later.

2.2.2.2 Getting the scores

The second syntactic extension to the XQuery language can be found in the `for` and `let`-clauses. Both are extended with a keyword `score` to make the hidden score available to the XQuery language by binding it to an XQuery variable. This allows for the construction of XQuery expressions that *depend* on the score calculated by the Full Text machinery.

```
for $i score $s
in $lib/book[./title contains text "Hitchhiker" ftand "Guide"]
where $s > 0.7
order by $s descending
return $i
```

In this example, the predicate `[./title contains text ...]` filters books depending on the Boolean value returned by the Full Text machinery. The `for`-clause iterates over those books that qualify against the predicate, thereby binding variable `$i` to the respective element node, and variable `$s` to the score that was returned by `contains text` together with the Boolean that made the book qualify.

Another example is the `let`-clause, as in the following query:

```
let $i score $s
:= $lib/book[./title contains text "Hitchhiker" ftand "Guide"]
return $s
```

Here, the variable `$s` is bound to a single *combined score* that reflects the scores of all books that qualify. Note that `$i` is bound to an item sequence that potentially contains more than one item, namely all the books for which the Full Text machinery returned true. However, the XQuery Full Text specification [1] requires the score variable `$s` to be bound to a singleton score. Clearly, this requires some way to combine the scores calculated for different qualifying books into a singleton value.

Besides, the XQuery Full Text definition does not allow the binding of a score variable `$s` *and* a sequence variable `$i` in the same `let`-clause, as is done in the above example. Pathfinder^{FT} adds this feature without further effort.

2.2.2.3 Interleaving XQuery and Full Text expressions

Although the `contains text` operator introduces a Full Text language that is syntactically and semantically distinct from the XQuery language, it does allow the use of values calculated by means provided by the XQuery language. The simplest is the use of string literals as in

```
. contains text "Hitchhiker"
```

but all other XQuery expressions of suitable type could be used instead:

```
for $i score $s
in $lib/book[./author contains text $person/surname]
return ($i,$s)
```

Although the XQuery specification [3] is a bit more tight about this, the compilation scheme described in this thesis allows for *arbitrary nesting* of XQuery Full Text expressions.

As a simple example, the expression

```
("foo","bar","qux") contains text "foo"
```

is a completely acceptable XQuery Full Text query to the PathFinder^{FT} compiler, although the Full Text index employed by the current back-end will choke on estimating the score of something not in a document.

Claiming such orthogonality directly triggers one question: What is the score of `42`? More specifically, what happens if the user demands a score that was never calculated:

```
for $i score $s in 42 return $s
```

PathFinder^{FT} compensates for this issue by expecting to know a default score. This could be a neutral element, an invalid score (the XQuery Full Text specification [1] would allow for -1 here), or a marker denoting "unscored" (*e.g.*, `null` if the database back-end supports this). When binding this default score to a score variable by using the `score` keyword, it needs to be mapped to something in the XQuery domain, so `null` might be somewhat difficult to use here.

2.2.2.4 Score propagation

The alert reader probably stumbled over the following peculiarity in the above examples: A query like

```
for $i score $s
in $lib/book[./title contains text "Hitchhiker" ftand "Guide"]
return ($i,$s)
```

uses an iteration over books generated by a path expression, thereby binding the score variable `$s` to scores that come from a syntactically different source. A more terse example is the following:

```
for $i score $s in $list[. contains text $e] return ($i,$s)
```

Note that `$i` iterates over the items drawn from the item sequence `$list`, while `$s` iterates over the scores created inside the predicate expression. So there are two different lists of values (items in `$list` and scores returned by `contains text`), originating at different locations, that are *zipped* together to form value/score pairs, and it is not clear by which means this should happen.

> Here, *zipping* (*aka. convolution*) refers to a canonical, and order-invariant, mapping from a pair of lists of the same length, to a list of pairs. An implementing function is often called `zip`, an instructive equation is the following:
>
> $$\mathsf{zip}([1,2,3],[a,b,c]) \equiv [(1,a),(2,b),(3,c)]$$
>
> One might think of a zip fastener, which employs a similar principle. The more general case maps n lists to a list of n-tuples, or even to a list of results of applying an $n$ary function on each of the tuples.

This zipping threatens the orthogonality of the XQuery Full Text language as described in Section 2.2.2.3: The syntactically different sources correspond to *different* sub-expressions in the abstract expression tree, each of which could be replaced by a different, much more complex one.

There may be different sources for scores in one predicate

```
for $i score $s
in $list[ (./foo contains text "Hitchhiker")
      and (./bar contains text "Guide")
        ]
return ($i,$s)
```

or Full Text predicates may be applied to different steps of a path expression

```
for $i score $s
in $list[. contains text "Hitchhiker"]/foo[. contains text "Guide"]
return ($i,$s)
```

both raising the question about how to deal with all the scores that are created by the different calls to the Full Text machinery.

The PATHFINDER^{FT} compiler employs a translation scheme that uniformly deals with all such cases.

2.3 What is a score?

2.3.1 The "second-order aspect"

The XQUERY FULL TEXT specification [1] claims[13] that it would be impossible to create a function that returns the score attached to the Boolean returned from the Full Text machinery:

> *The use of score variables introduces a second-order aspect to the evaluation of expressions which cannot be emulated by (first-order)* XQUERY *functions. Consider the following replacement of the clause* `let score $s := FTContainsExpr`
>
> ```
> let $s := score(FTContainsExpr)
> ```
>
> *where a function* `score` *is applied on some FTContainsExpr. If the function score were first-order, it would only be applied on the result of the evaluation of its argument, which is one of the Boolean constants true or false. Hence, there would be at most two possible values such a score function would be able to return and no further differentiation would be possible.*

This justification is bogus for two reasons: First of all, if the above argumentation were valid, it would be impossible to use the `score` keyword within a function. Easily, one could declare a `score()` function otherwise:

```
declare function score($arg as item()*) as xs:float {
    let score $s := $arg return $s
}
```

Here the "second-order aspect" carries over to the declared function, which is forbidden by the above argumentation. Disallowing the use of any scoring inside function definitions however, is a restriction I consider too rigorous.

[13] `http://www.w3.org/TR/2009/CR-xpath-full-text-10-20090709/#doc-xquery-LetClause`

Second, XQuery lacks referential transparency anyway[14], so it is not much good as rationale in the above argument. The reason for this is that XQuery's node constructors do have side effects: Each constructed node has a unique identity, i.e., the appearance of `<a/>` in an XQuery expression denotes neither constant nor function. The following query nicely demonstrates this.

```
declare function local:f($xs as element()*) as element() {
  <a>{ $xs }</a>
};
let $q := <b/>
return ( if ($q is $q)
         then "$q is $q"
         else "$q is not $q"
       , if (local:f($q) is local:f($q))
         then "f($q) is f($q)"
         else  "f($q) is not f($q)"
       )
```

Running this query evaluates to

```
("$q is $q", "f($q) is not f($q)")
```

In contrast to node identity (tested with the `is` operator), *equality is too weak* to recognise the different identities of nodes: Value comparison operator `eq` and general comparison operator `=` both atomise their arguments before performing the actual comparison (see Section 5.17 for general comparison), hence, node identities do not play a role in this case anyway. The following query returns `true`.

```
<a>hello<b/>world</a> eq "helloworld"
```

2.3.1.1 Why node identity?

A rationale for node identity certainly is XPath semantics: An axis-step always maps a set of context-nodes to a duplicate-free sequence of nodes in document-order. It is important to note that duplicates are determined by identity, not by equivalence.

```
let $doc := <doc>
              <x>             – the first intermediate node
                <x>           – and the second one
                  <target/>
                </x>
                <target/>
              </x>
            </doc>
return $doc//x//target
```

In the above query, the intermediate axis step `//x` finds both `x`-nodes, forming the sequence $(\mathtt{x}_1, \mathtt{x}_2)$. From this context set, the final step `//target` finds both `target`-nodes from $\mathtt{x}_1$, and additionally finds the second `target`-node again from $\mathtt{x}_2$. The duplicate occurrence of the second

[14] http://www.w3.org/TR/2007/REC-xquery-20070123/#id-basics

`target`-node is dropped from the result sequence (A smart implementation combines these steps, see [10]).

Comparing by equivalence instead would make it impossible to reliably count the number of occurrences of a tag, as in

```
fn:count($doc//target)
```

which would always return `1`.

Not removing duplicates would lead to the same problem in the opposite direction, pretending the existence of more nodes than actually exist.

2.3.1.2 Semantics of query structure

Another justification for the "second-order aspect" claimed[15] in the XQuery Full Text specification is the assumed relevance of the structure of a query to the result:

> *There are numerous scoring algorithms used in practice. Most of the scoring algorithms take as inputs a query and a set of results to the query. In computing the score, these algorithms rely on the structure of the query to estimate the relevance of the results.*
>
> *In the context of defining the semantics of XQuery and XPath Full Text, passing the structure of the query poses a problem. The query may contain XQuery 1.0 and XPath 2.0 expressions and XQuery and XPath Full Text expressions in particular. The semantics of XQuery 1.0 and XPath 2.0 expressions is defined using (among other things) functions that take as arguments sequences of items and return sequences of items. They are not aware of what expression produced a particular sequence, i.e., they are not aware of the expression structure.*
>
> *To define the semantics of scoring in XQuery and XPath Full Text using XQuery 1.0, expressions that produce the query result (or the functions that implement the expressions) must be passed as arguments. In other words, second-order functions are necessary. Currently XQuery 1.0 and XPath 2.0 do not provide such functions.*

I disagree with the conclusion made in the third paragraph: Having access to the structure of the query neither implies second-order functionality, nor the other way round:

First, to gain access to the query structure, it is enough to have access to the parse tree. From this, the "hard semantics", can be derived by forming an abstract syntax tree. But also the query structure is available therein, *i.e.*, the interpretation of the parse tree is not limited to construct an AST just by not having higher-order functionality in the implementing language. Having access to the parse tree by language means, however, does not imply higher-order functionality: A language does not necessarily provide means to evaluate data, like, *e.g.*, the `eval` functions available in Python[16], or Perl[17].

Second, consider a higher-order function, such as Haskell's `map` function. As its type

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$

[15] `http://www.w3.org/TR/2010/CR-xpath-full-text-10-20100128/#ScoreSec`
[16] `http://docs.python.org/library/functions.html#eval`
[17] `http://perldoc.perl.org/functions/eval.html`

shows, it transforms a unary function $f :: \alpha \to \beta$ to a unary function working on lists of type $[\alpha] \to [\beta]$, by applying the function f to each member of the input list:

```
map f [1,2,3] ≡ [f 1, f 2, f 3]
```

Although `map` is a higher-order function, there are no means it could possibly detect the structure of its argument function f. *E.g.*, if $f \equiv$ `sqrt`, it would be impossible for `map` to find out by which means the root is calculated.

2.3.1.3 Missing the "second-order aspect"

It is important to realise where PATHFINDER^{FT} makes the query structure accessible to the executing engine: Section 5.22 describes, among others, compilation techniques that make Full Text expressions completely accessible to a Full Text engine. But it is an intrinsic feature of the PATHFINDER^{FT} architecture that the XQUERY portion of a query is isolated from interpretation by the Full Text engine.

In the IR community it is common sense that the syntax used by the user to denote the query contains information about the users information need, which is not expressed by an abstract syntax tree, see the quote in Section 2.1.

Allowing a more IR-style interpretation of the Full Text language may lead to diverging semantics for the following two queries. An algebraic style semantics of the (commutative) `and` operator in

```
doc("library")/books[ ./abstract contains text "magic"
                  and ./abstract contains text "technology"
                    ]
```

clearly allows to interchange the two search term literals without changing the semantics of the query, and doing otherwise is potentially breaking the compiler's optimisation strategies.

On the other hand, a Full Text system might want to give more weight to the term `"magic"` than to the term `"technology"` in the query

```
doc("library")/books[./abstract contains text "magic"
                                       ftand "technology"
                     ]
```

simply because "there must be a reason why" the user entered `"magic"` first.

The former query contains the Boolean XQUERY operator `and` combining two Full Text expressions, while the latter contains one Full Text expression employing the Boolean Full Text operator `ftand` to combine two Full Text terms.

PATHFINDER^{FT} is able to pass various representations of a Full Text expression to the Full Text engine, thereby allowing for IR-style semantics of operators such as `ftand`. On the other hand, it resorts to a DB-style perception of semantics for the XQUERY portion of the language, allowing for the interchange of both keywords in the first example without changing its semantics.

2.3.2 Where are the scores?

The approach taken by PATHFINDER^{FT} allows the definition of a function that returns the score of its argument, while not further violating the remains of XQUERY's tendency to referential transparency:

PATHFINDER^{FT} takes each and every XQUERY FULL TEXT item as a proper pair[18] of a native XQUERY value and a score (which would be a float in most cases). The available XQUERY operations only "see" the XQUERY value while the score remains hidden, *i.e.*, its existence is orthogonal to the XQUERY items. Section 2.4 gives a more thorough discussion about why such a design is required.

With this perception, it is perfectly sound to allow a function `score` that maps a scored item to its score, by projection on the second component.

So what about equality? For cases where

$$a = b \Rightarrow f\ a = f\ b$$

holds in XQUERY, we would like it to hold in XQUERY FULL TEXT as well. So there are three possible approaches:

1. Redefine equality to obey the scores.
2. Strip all scores when calling a function, or set them to a fixed default value.
3. Ignore scores for comparison, only compare the value.

PATHFINDER^{FT} currently implements the last option, *i.e.*, the Query[19]

```
("foo" scored 0.3) eq ("foo" scored 0.7)
```

evaluates to `true`, although the scores of the two string values differ.

Thus, *equality is too weak to recognise different scores*, just as explained above for node identity. This analogy is the reason why I consider the approach taken to fit best: XQUERY's equality already had a weak spot before. Equality of scores of two values x and y can still be tested with

```
let score $a := x
  , score $b := y
return $a = $b
```

The other two solutions do not fit as well: If equality would obey scores, *i.e.*, if the above query evaluated to `false`, it would be more difficult for a user to use correctly: For some query result `$q` containing the string `"foo"` with a score different from the default score, the comparison[20] `$q = "foo"` would nevertheless evaluate to `false`, giving the impression that `"foo"` was not part of the result. PATHFINDER^{FT} returns `true` instead.

Stripping all scores from the items before passing them to a function would guarantee referential transparency as far as XQUERY does, simply because functions could observe variations only in the value-component of the passed item, not in the score component. But this would also thwart the efforts taken in the XQUERY FULL TEXT design to *implicitly* pass around scores on, *e.g.*, XPATH axis steps. One would have to draw a line between *functions* that require stripping, and other *operators* that do not employ stripping. Not only is this distinction subject to whim, it would merely reduce the discussed problem to the smaller class of those *operators*, not solve it completely.

[18] in a mathematical sense, *i.e.*, not an XQUERY sequence of two items

[19] introducing the non-standard keyword `scored`, which explicitly sets the score of all items in the sequence on its left hand side to the value given to the right, see Section 5.9.

[20] using XQUERY's existential semantics (see `http://www.w3.org/TR/2007/REC-xquery-20070123/#id-general-comparisons`)

2.3.3 Semantics of scores

Recall one of the earlier examples:

```
for $i score $s
in $lib/book[./title contains text "Hitchhiker" ftand "Guide"]
where $s > 0.7
order by $s descending
return $i
```

If the score reflects, as the XQuery Full Text specification demands, how well the Full Text expression is satisfied, then one has to find a threshold for returning true. If a book does not qualify, the Full Text machinery could return false with an arbitrary score, or true with a very low score. The exact behaviour is a design choice to be made by the implementers.

This however raises the question whether a pair of a Boolean and a score is required at all. If, *e.g.*, a threshold of t implied for Boolean b with score s that $b \equiv s > t$, then the score would carry all information, *i.e.*, the Boolean would be redundant.

Also, one could argue that the Full Text machinery should never return false: If the query asked for `ascending` ordering instead, one would expect the least significant books first. If they are filtered out just because `contains text` returned false, the list would begin somewhere in the middle, dropping relevant (due to irrelevance) results.

One could as well (deviating from the XQuery Full Text specification) argue that the score indicates how confident the Full Text machinery is about its Boolean decision. In that case high scores would make sense with a false return value.

The Pathfinder$^{\text{FT}}$ architecture does not impose any semantics whatsoever on the scores, *i.e.*, it is fit to comply with any of the above interpretations. However, the scoring model functions provided in Chapter 5 share one common perspective: The involved scores shall not interfere with the calculations of values, unless explicitly requested by the user using one of the keywords `score`, or `scored`. More formally, that situation is described as follows: To calculate an item $\langle v|s\rangle$ by application of a function f on some arguments

$$\langle v|s\rangle \equiv f\ \langle v_1|s_1\rangle\ \ldots\ \langle v_n|s_n\rangle$$

there should be a function f' so that

$$\forall s_1, \ldots, s_n.\, v \equiv f' v_1\ \ldots\ v_n$$

holds. The obvious exception being uses of `score`, or `scored` in f.

This is a notable restriction: Consider the simple set difference expressed by the following query.

```
let $list := doc("library.xml")/books
for $b in $list[./author contains text "Wells"]
          except
          $list[./text contains text "gay"]
return $b/title
```

This may not return "The Time Machine"[21] for a simple reason: The text contains the word `"gay"`, and if the Full Text engine adds it to the second node sequence, even with a tiny score

[21] H. G. Wells, 1895. `http://www.gutenberg.org/etext/35`

only, than it will be removed from the first sequence by the `except` operation, even though it may have assigned a high score from the first test.

In this situation, it may be desirable to merely reduce the score of an item, instead of removing it, and the PATHFINDER^{FT} architecture allows doing so. On the other hand, such an interpretation of scores is very much similar to fuzzy logic and fuzzy set theory (see [20]), where membership of an element in a set is defined by a characteristic function $\tilde{\in} :: \mathsf{Universe} \to \mathsf{FuzzySet} \to [0,1]_{\mathbb{R}}$. Now an item sequence with attached scores may be considered a fuzzy set. However, the membership test $x \tilde{\in} S$ returns a number (the score) instead of a Boolean with a score attached. Again, the question is what role the Boolean plays.

2.4 Neither Tuple, nor Record, nor Class

This whole thesis is about extending the PATHFINDER compiler with an infrastructure for implicitly handling scores, *attached* by a scoring engine to the native XQUERY values.

But why are scores *attached* to values at all? Because this is the only means to implement *implicit* score propagation in, or better: on top of XQUERY. The reason for this being the fact that the XQUERY design assumes XML to be more than enough to describe any record type necessary, which, to some extent, is not totally wrong. The remainder of this "what-if" section provides some suggestions why more than flat item sequences and XML might have been helpful in the extension of XQUERY with a scoring infrastructure.

2.4.1 Tuples

An interface to the scoring engine typically wants to return not only a Boolean value, or a node, but also a score or a match position (a pointer into the target document), *i.e.*, a tuple of at least two values. XQUERY, however, does not provide means to do so: A sequence of item/score pairs is flattened automatically, becoming a sequence of alternating scores and values with an even number of members, which is much more difficult to handle since one cannot map a function over it.

[14] uses the workaround described on page 31 to get hold of the result nodes and the associated scores.

Of course a scoring engine could as well pack each of its result value/score pairs $\langle v|s \rangle$ in an XML snippet:

```
<mns:item xmlns:mns="a namespace reserved for this purpose">
  <mns:value>v</mns:value>
  <mns:score>s</mns:score>
</mns:item>
```

This, however, would require the (rather expensive, consider the copy-semantics for node construction) creation of lots of new XML snippets, and it would require the programmer to add rather ugly boilerplate unpacking code to finally access the scores and the values. Additionally, an optimising compiler would be left with the task to remove this packing/unpacking code where the scores are not actually used.

2.4.2 Overloading with Typeclasses

Another means to implement implicit score propagation could be achieved with *ad-hoc* polymorphism[22] along the lines of the HASKELL programming language [23]. Assume a class `Steps` that forms the family of all functions used to perform XPATH axis steps:

```
class Steps α where
    child :: [α] -> [α]
    descendant :: ...
```

and similar classes for numeric and Boolean operations. Any value can then be extended with a score of appropriate type σ by wrapping it with a data constructor as, *e.g.*, the following.

```
data Scored α
    = Scored σ α
```

With this construction, and by defining how scores should propagate on axis steps, we can easily declare that everything in the `Steps` class is still in the `Steps` class when annotated with a score:

```
instance Steps α => Steps (Scored α) where
    child = aggregateScores . groupByNodes . map onestep
        where
        -- for each context node, pair result nodes with context scores
        onestep :: Scored α -> (σ, [α])
        onestep (Scored s x) = (s, child [x])
        -- group scores by result nodes
        groupByNodes :: [(σ, [α])] -> [(α,[σ])]
        groupByNodes = ...
        -- for each result node, aggregate scores of respective context nodes
        aggregateScores :: [(α,[s])] -> [Scored α]
        aggregateScores = ...
    descendant = ...
```

And similar for all other types for which implicit score propagation is desired, *e.g.*, assuming a class `Boolean`:

```
instance Boolean α => Boolean (Scored α) where
  (Scored s1 x1) & (Scored s2 x2) = Scored (min s1 s2) (x1 & x2)
  ...
```

2.4.3 Overloading with Records or Objects

Object oriented languages like, *e.g.*, Java or C++, provide means to extend a class through inheritance, some even allow for overloading of operators. In such a scenario it is an option to extend, *e.g.*, the class for XML nodes, to accommodate an additional score. By overloading the operations that work on the base data types, implicit score propagation could be implemented.

The suggestive example query uses fictional dot-notation to access the members of a record, or the fields of an object:

[22] `http://www.haskell.org/haskellwiki/Ad-hoc_polymorphism`

```
for $item in $list[. contains text "foo"]/child::author
order by $item.score
return $item.value/child::surname
```

An overloaded step operator is used here: The first use fetches author nodes, and propagates scores, the second use fetches surnames and is applied on "pure" values. This notation makes it explicit that no score propagation should be applied for the latter step.

* * *

The benefits of a design that integrates scores by the means naturally available in the programming language should be obvious: Each of the suggested approaches gives more freedom to the programmer, the user, and the library interface designer. The hypothetically available compilers and interpreters could be used without extension.

The XQuery Full Text architecture makes it impossible to add different score types and propagation algorithms by means of the language itself. The Pathfinder^{FT} architecture is at least flexible enough to allow a database administrator to add such extensions by tweaking the compilation rules as described in this thesis. But a scoring infrastructure defined by means of the query language itself would offer this flexibility to the database users, *i.e.*, the XQuery Full Text users.

With the design alternatives in sight, the implicit score propagation suggested by the XQuery Full Text draft merely looks like a hack: It had to be added to the language kernel in an inaccessible, and obscure way (*i.e.*, out of the programmers control) because the original language, XQuery, was never built to act as a friendly host for such extensions.

Chapter 3

Related Work

Relational Algebra is used by [5] to present a formal model for Full Text search. The involved *full-text relations* contain a variable list of attributes, one to represent a context node, and further to represent positions where the query matches. In this setting, each tuple in the relation contains exactly one *context node*, and a *list of positions*.

The model proposed in [5] is designed to capture Full Text semantics with positions of tokens, and to embed them in a relational setting. The authors also associate a score with the tuples in the full-text relations, and they present score transformations for their algebra operators. *I.e.*, they define for each Relational Algebra operator how scores implicitly present in each tuple of a relation shall be mapped to the scores in the result relation.

E.g., for the projection $\pi_{\mathsf{cnode},\mathsf{score},\mathsf{pos}_1,\ldots,\mathsf{pos}_k} R$, the scores of all input tuples $t_1, ..., t_n$ in R that are projected to the same output tuple t shall be combined. In the tf.idf case, the formula

$$t.\mathsf{score} = \Sigma\{t_i.\mathsf{score} | 1 \leq i \leq n\}$$

is proposed. In the setting of my thesis, an aggregation would be required to perform such a computation because the Relational Algebra used here does not provide means for the implicit calculation of *attached* scores.

This is a major difference to the PATHFINDER$^{\text{FT}}$ architecture: While [5] handles score propagation implicitly for each algebra operator, PATHFINDER$^{\text{FT}}$ handles scores explicitly at the algebra level. *I.e.*, the implicit score propagation of XQUERY FULL TEXT is made explicit by PATHFINDER$^{\text{FT}}$'s compilation steps.

The benefit of making score calculations explicit is that rewriting the algebra plan becomes easier: Again for the tf.idf case, [5] suggests the following score transformations for negation, union and difference:

$$\neg\langle a|s\rangle = \langle \neg a|1-s\rangle$$
$$\langle a|s\rangle \texttt{ and } \langle b|t\rangle = \langle a \texttt{ and } b|\mathsf{min}\ s\ t\rangle$$
$$\langle a|s\rangle \texttt{ or } \langle b|t\rangle = \langle a \texttt{ or } b|s+t\rangle$$

Which is clearly not compatible with rewritings à la, *e.g.*, DeMorgan.

This is not due to the model proposed by [5], but rather due to the (unsound, [14]) score combining functions chosen for the algebra operators. The drawback implicit score propagation

at the algebra level does introduce, is that it prevents the optimiser from rewriting the Relational Algebra plan without potentially changing the scores.

In contrast to [5], my thesis does not introduce an algebra to model Full Text queries, nor does it map Full Text languages to Relational Algebra, nor extend Relational Algebra with implicit score propagation. Instead, an existing [16] use of pretty much traditional Relational Algebra, to implement XQuery evaluation (*i.e.*, flat item sequences, nested iterations) on relational database back-ends [4], is extended with a *scoring infrastructure* to serve as an XQuery Full Text back-end. To this end, the fixed-width relations with schema iter|pos|item, used by the Pathfinder compiler to model item sequences and iterations, are extended explicitly with one column named score carrying the score attached to an item. This score column is never handled implicitly by any algebra operator, instead, it is a first-class citizen among all other attributes of a relation.

Thus, the score propagation implicit in an XQuery Full Text expression like

e_1 `or` e_2

is made explicit in the corresponding Relational Algebra[1] expression

$$@_{\mathsf{pos}:1} \triangleleft \pi_{\substack{\mathsf{iter}\\ \mathsf{item}\\ \mathsf{score}}} \triangleleft \underbrace{\mathrm{op}^{\mathsf{score}:}_{\mathsf{score1}+\mathsf{score2}}}_{\text{calculate score}} \triangleleft \underbrace{\mathrm{op}_{\mathsf{item}:\ \mathsf{item1}\vee\mathsf{item2}}}_{\text{calculate value}} (\pi_{\substack{\mathsf{iter}\\ \mathsf{item1}:\mathsf{item}\\ \mathsf{score1}:\mathsf{score}}} [\![e_1]\!] \bowtie \pi_{\substack{\mathsf{iter}\\ \mathsf{item2}:\mathsf{item}\\ \mathsf{score2}:\mathsf{score}}} [\![e_2]\!])$$

using the sum operator for combining scores in an `or` expression.

By making the score computation explicit at the algebra level, Pathfinder$^{\mathrm{FT}}$ facilitates optimisations: XQuery Full Text's implicit score propagation is mapped to explicit Relational Algebra operations, *i.e.*, on a semantic level the Relational Algebra optimiser cannot distinguish query results from scores, and handles them just the same way: as first-class citizens of the tuples. Thus, a rewrite of the Relational Algebra plan is guaranteed to return the same scores as an unoptimised plan.

Of course, this does not magically solve the problem of *rewriting a query* if the score propagation for Boolean operators happens to be defined as above, but it allows the optimiser to *rewrite the plan*, *e.g.*, by separating the computation of scores from the computation of a Boolean expression, and to optimise them independently. Comparison of an unoptimised plan as in Figure 7 on page 90 with the optimised version in Figure 6 on page 88 shows this nicely. If, however, the score propagation remains implicit at the algebra level, correctness of plan rewrites depends on the score propagation used.

The pitfalls introduced by not well-behaved score propagation persist, but have now shifted to query-rewriting instead of plan-rewriting. It is still required to take special care at the point where score computation is made explicit, and the user may still experience strange results when he does not fully understand the effects of rewriting a query to a seemingly equivalent one.

Throughout this work I will point out situations, where the concrete implementation of score propagation influences query rewrites.

Whereas XQuery Full Text is introduced by [1] as an extension of XQuery with Full Text semantics, my thesis emphasises the separation of the Information Retrieval world from the Database world: Pathfinder$^{\mathrm{FT}}$ isolates the potentially sloppy semantics of a Full Text expression on the right hand side of the `contains text` operator from the strict semantics of the

[1] The notation is explained in Section 5.1.

remaining XQuery expressions. This is why the Pathfinder compiler can be used to evaluate the XQuery portion of XQuery Full Text expressions, while the Full Text expressions must be evaluated by a separate *scoring engine*. Following [1], the device of communication with the scoring engine is to provide it with a search context and a Full Text search specification, and to receive a scored Boolean from it. But diverging from [1], this is strictly the only means provided by Pathfinder^{FT}. In particular, there is no "second-order" semantics (in the sense of [1], see Section 2.3.1) involved, *i.e.*, the scoring engine will not see the XQuery part of the query it answers. It may, however, have full access to the Full Text part of the query, and exploit its structure in a very IR-ish way to determine the user's information need.

The fact that Pathfinder^{FT} makes no further assumptions about the scoring engine used should make it versatile in that the Full Text back-end could be replaced with a different engine as the user requires. The current prototype implementation of Pathfinder^{FT} uses PF/Tijah [14] as its scoring engine.

This was an obvious choice: [14] (and in more detail [18]) already describe an integration of Full Text search in the Pathfinder/MonetDB system. In contrast to my thesis, their work focuses on bringing together the Tijah index [15] and the Pathfinder compiler, *without* making the step from XQuery to XQuery Full Text. While [14] does exploit Pathfinder's potential as a high performance XQuery database engine, they do not handle XQuery Full Text's implicit score propagation. Instead, PF/Tijah provides an XQuery interface to the Tijah index by adding several functions at the XQuery level, and requires the user to explicitly handle the scores in his XQuery query:

```
let $context := doc("docname.xml")
  , $query := "//tag[about(.//annot, john doe)]"
  , $result := tijah-query-id($context, $query)
for $node at $rank in tijah-nodes($result)
return <item rank="{ $rank }" score="{ tijah-score($result, $node) }">{
         $node
       }</item>
```

The function `tijah-query-id($context, $query)` would like to return, ordered by relevance, *pairs* of nodes and scores. Unfortunately, this is not possible since XQuery does not know about pairs. To work around this fundamental design flaw in the XQuery specification (see also Section 2.4), a *handle* `$result` is returned, which may be used by function `tijah-nodes($result)` to retrieve the sorted list of result nodes, and, together with a node, by the function `tijah-score($result, $node)` to retrieve the respective score.

The XQuery Full Text operator `contains text` basically suffers the same disease, trying to return pairs of Booleans and scores. As [14] points out, in [1], Section 4.4[2], a construction similar to the one above is suggested. Simply adopting such a construction, however, undermines the concept of implicit score propagation as described in this thesis. Section 2.4 presents some insights about where the XQuery language design gets in the way of adding elegant score propagation means.

The Score Region Algebra, introduced by [17], lies at the heart of the Tijah system. The basic idea is to take the XML document as a set of nested regions, determined by their start and end tags. Then a query on the document can be implemented using set and containment operations on these regions, forming the Region Algebra. [17] annotates the regions with scores, and extends

[2] `http://www.w3.org/TR/2010/CR-xpath-full-text-10-20100128/#ScoreSec`

the operations in the algebra to implicitly handle scores, yielding the Score Region Algebra. In their work, three aspects of XML IR are recognised, namely

- *element relevance score computation*, the calculation of a score for an element with respect to one single search term,
- *element score combination*, the combination of per term scores for an element to form a combined score (think of Boolean queries as in `about(., foo bar)`), and
- *element score propagation*, used to propagate a score from the scored elements to the result.

The relation to this work is as follows: Element relevance score computation, and element score combination are concepts expressed in the Full Text part of the XQuery Full Text language. Consider

```
doc("lib.xml")//book[./title contains text "Wallace" ftand "Gromit"]
```

The Full Text engine will have to estimate the relevance of a title according to the terms `"Wallace"`, and `"Gromit"`. If it is capable to evaluate the conjunction, it may immediately combine the scores to a single one, named *score combination* by [17]. Then, the resulting `book` nodes need to receive the scores from their titles, which is called *score propagation*. (But what do we need if a book hosts multiple titles? Is it score combination, or rather propagation?)

Consider an only slightly different query[3] now:

```
doc("lib.xml")//book[ ./title contains text "Wallace"
                  and ./title contains text "Gromit"
                    ]
```

In PathFinder^{FT} parlance, this is where score propagation enters the predicate: The two `contains text` operators create two Booleans, whose scores have to be combined. It is important to distinguish this operation from score combination, since it happens outside the domain of the Full Text language (see Section 2.2.1 for a more precise explanation). The main point to note here is that PathFinder^{FT} propagates scores not only from elements to other elements, but from operands to results, as, *e.g.*, from the arguments of `and` to its result, but also from a set of context nodes to the result set of an axis step. So, to answer the above question about multiple titles: In PathFinder^{FT}, scores may indeed propagate from several elements to one single element. This more generic approach leads to the fact that PathFinder^{FT} does not have a separate notion of *score combination*: It either performs score propagation, or calls a Full Text engine, which does score computation.

The idea of *abstract functions* that implement a certain scoring model in the Score Region Algebra is reused by PathFinder^{FT}. [17] describes various scoring models, and how corresponding implementations can be achieved by parameterising the Score Region Algebra with concrete definitions of these functions. A similar approach is taken in this work, by defining a set of functions, and interfaces that need to be specified in order to define the concrete score propagation. Section 5.23 summarises the interface that needs to be implemented by a scoring model.

[3] By the way: If the Full Text engine is only capable to estimate according to a single keyword per search context, PathFinder^{FT} allows to *unfold* the former query to the following, see Section 5.22.4. The question whether and why the user might expect the same result is also discussed there.

The BaseX project (see [8]) is also located at the DBIS group in Konstanz. In contrast to this work, BaseX focuses on building an XML database from scratch. Recently, XQuery Full Text was added to the BaseX feature set. Although its storage layer is inspired by a relational encoding quite similar to the one employed by the Pathfinder compiler (see Section 4.2.2), its evaluation engine does not employ relational algebra at all. Thus, although some of my concerns against implicit score propagation certainly do hold for BaseX as well, the technical means of adding a scoring infrastructure to the Pathfinder compiler are not applicable to the BaseX architecture.

This is due to the fact that BaseX maintains XQuery items and item sequences, which are not in the document storage, in a native way, *i.e.*, as Java objects in main memory. All manipulations of item sequences that may occur during query evaluation is done by classes tailored to this end. In contrast, the Pathfinder compiler aims at creating Relational Algebra plans that can be evaluated on a wide variety of RDBMSs, and the Pathfinder^{FT} project tries to extend this compiler. Thus, this work describes how the Relational Algebra code created by the Pathfinder compiler should be modified, instead of elaborating on what should happen with the scores exactly.

Of course, the general idea of *attaching* scores to items, and to take scores from arguments of operators and calculating a new score to be attached to the operator's result, should match any implementation of XQuery Full Text, because that is just what the implicit score propagation is required to do. This thesis is about how to accomplish a suitable scoring infrastructure in the Pathfinder setting.

One major advantage of the Pathfinder^{FT} approach in comparison with BaseX, and maybe any native XQuery Full Text engine, is the ability to easily strip scores and score computations from the compiled plan where they are dispensable. In fact, this optimisation comes for free in the setting of this thesis, by means of a Relational Algebra optimisation phase named *dead code elimination*, which is already present in the Pathfinder compiler.

Chapter 4

The Compiler

4.1 Intermediate Languages

The compilation of XQuery Full Text into Relational Algebra is performed in a number of steps via several intermediate languages. This is referred to as the *vertical language stack*, in contrast to the *horizontal language stack* as discussed in Section 2.2.1.

4.1.1 XQuery Full Text and XQuery Core

The XQuery Full Text input is first parsed into an abstract syntax-tree (AST) that captures the syntactic structure of the source code.

The AST is then transformed into simplified XQuery Core, where some of the XQuery constructs have been replaced by more primitive XQuery constructs with equivalent semantics. Predicates, *e.g.*, can be translated into explicit `for`-loops. The XQuery expression

```
doc("foo.xml")/book[./author = "joe"]/title
```

featuring a Boolean predicate, can be replaced (adding a fresh variable dot) by

```
for dot
in doc("foo.xml")/book
return if dot/author = "joe"
       then dot/title
       else ()
```

The XQuery Core structure employed by the Pathfinder^{FT} compiler does not necessarily coincide with the XQuery Core language used W3C recommendation[1].

[1] http://www.w3.org/TR/xquery-semantics/#sec_core

4.1.2 Relational Algebra

XQuery Core is compiled into Relational Algebra. This phase is subject to the most modifications in the Pathfinder compiler: All compilation rules have to be drilled out to at least accept an additional score column, some of them even need to generate extra code to mangle the scores.

Although the Pathfinder^{FT} compiler could benefit from an additional Relational Algebra primitive tailored towards Full Text, this is not obligatory. The current implementation works without such an operator, and hence requires no extensions to the subsequent Pathfinder compilation phases. Section 5.21.1 shines a light on a possible specialised Relational Algebra operator.

4.1.3 NEXI

Due to the absence of a specialised operator, other means are required to pass information about Full Text expressions to the underlying scoring machinery.

To this end, Pathfinder^{FT} creates code which, at runtime, calculates a representation of the Full Text query, which is then passed to the underlying PF/Tijah engine via a function call. The details of this, and alternatives, are explained in Section 5.22.

* * *

The overall architecture of the Pathfinder^{FT} system is depicted Figure 2 on page 37. The shaded area below the thick line represents the available MonetDB/Pathfinder, and PF/Tijah systems, both consuming XQuery as input (the latter making use of XQuery function calls that trigger the scoring engine in the back-end).

Above the thick line the Pathfinder^{FT} subsystem is shown. The XQuery part of XQuery Full Text is translated to XQuery Core in a pretty usual way, while the Full Text part can be compiled in alternative ways described in Section 5.22. The constructed XQuery Core expression is then compiled into Relational Algebra, which adds the scoring infrastructure. The Relational Algebra code this step generates is fed to the optimiser of the original Pathfinder compiler, speckled with similar function calls to the Full Text engine as employed by the PF/Tijah system.

4.2 The Pathfinder compiler

The Pathfinder compiler is a purely relational XQuery compiler. *Purely relational* means that all document storage and query processing is done on a *relational* database back-end. So Pathfinder's task is to translate data and query into a relational language, and to decode the database's response.

XML documents are first transformed into a relational encoding [9] and stored in the relational database back-end. XQuery queries are then compiled into relational algebra plans that are executed on the database. The database returns the resulting XML document in the aforementioned relational encoding, which is then decoded into plain XML.

The motivation behind this is to exploit the maturity of available relational engines for XQuery processing. To see that this is not a trivial task, observe some of the fundamental differences between XML and relations:

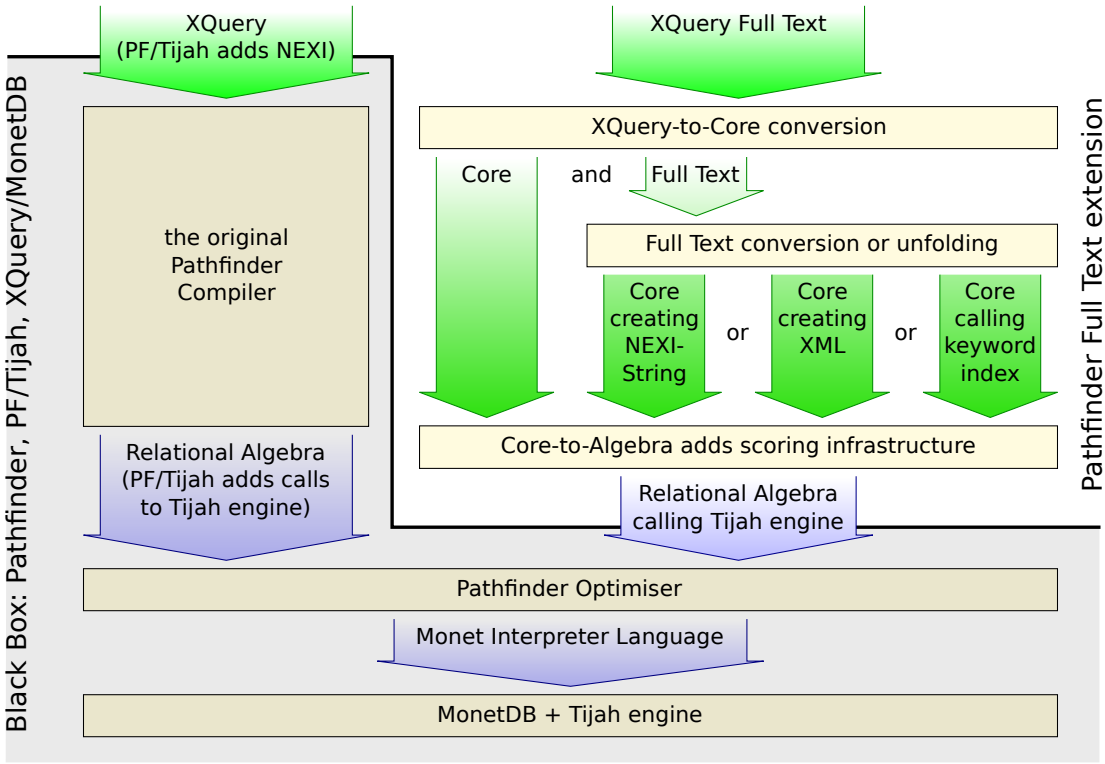


Figure 2: Architecture of the PATHFINDER^{FT} system, and its location within the environment formed by PATHFINDER, MONETDB, and PF/TIJAH. The shaded area below the thick line shows previously existing systems.

- Relations are unnested, while XML data is nested.
- Relational algebra is set-oriented, whereas XQUERY is sequence-oriented.
- A relational database schema is a set of attribute-name to type mappings, in contrast, an XML schema is expressed via an augmented context-free grammar (*e.g.*, RELAX-NG, XML Schema).

Despite these challenges, the PATHFINDER developers came up with a scalable, high-performance solution for XQUERY [16]. The PATHFINDER^{FT} project described in this thesis extends the PATHFINDER technology to XQUERY FULL TEXT.

This chapter describes the PATHFINDER technology to an extent sufficient for understanding the extensions provided by the PATHFINDER^{FT} system. It does not present new work, and it may deviate from the original PATHFINDER implementation for brevity and simplicity of presentation. A more thorough explanation of the PATHFINDER system can be found in [21]

4.2.1 Basic XQuery data structures

When processing XQuery queries, one has to deal with two separate data structures: XML data forms the queried trees, and is held in the database.

XQuery item sequences are the principal kind of data used in the XQuery language. An item is a value of a primitive type, such as Integer, String, Boolean, or Node. An item sequence is an ordered, flat list of zero or more items, potentially of heterogeneous type. These sequences are never nested, *i.e.*, they are always flat, and a singleton item sequence is indistinguishable from the contained item alone. The following sequences are thus equivalent:

```
(1, 2, "foo")  ≡  ((1), (2, "foo"))
           42  ≡  (42, ())
```

We say that a node in an item sequence is represented by its node ID γ, also called *node surrogate*, which is a pointer into the document storage. The storage architecture is not of interest in this work, the only requirements we make use of is that

- a node can be uniquely identified by its ID, and
- the database engine provides means to navigate within the trees by using a dedicated interface function (the step operator $\text{⌐}_{\alpha,n}$, see Section 5.8).

Just for completeness the next section briefly describes the encoding.

4.2.2 Pathfinder's XML encoding

Pathfinder/MonetDB employs a schema-oblivious relational encoding of XML data, which allows for fast XPath axis steps [9]. The basic idea is to perform a depth-first traversal of the XML document tree, and to annotate each node with its pre-order and post-order numbers. The pre-order number reflects document order, and also serves as node identifier. The post-order number of a node is used to reconstruct the tree structure.

A simple example document is encoded as follows:

<a>foo<c>bar<d/></c> ≡

pre	post	node
0	5	<a>
1	1	
2	0	"foo"
3	4	<c>
4	2	"bar"
5	3	<d>

Different encodings are feasible without loosing the tree structure. *E.g.*, one might trade the post-order number for the size of the corresponding subtree.

Within the scope of this thesis it is sufficient to consider the XML storage a black box, with the property that each node in an XML tree can be identified with its node ID, and that XPath axis steps can be performed. Here, the notation γ_1 is used to refer to the node with ID 1, *i.e.*, foo.

4.2.3 Pathfinder's XQuery item sequence encoding

The PATHFINDER compiler uses a technique called *loop-lifting* to represent an XQUERY item sequence for all iterations it appears in at once, *i.e.*, the relational encoding of a sequence spans all iterations of its lifetime. For each item that is contained in the sequence, the corresponding relation denotes the value of the item, the position it occupies in the sequence, and the iteration during which this judgement is valid.

An example:

```
for $i in (10,20,30)
return ($i,"x")
```

The whole query, and hence the sequence `(10,20,30)`, appears in one top-level iteration. The sequence iterated over is encoded into a relation with schema iter|pos|item, and an accompanying loop relation enumerates the iterations for the scope of the sequence.

$$\texttt{(10,20,30)} \equiv \begin{array}{ccc} \text{iter} & \text{pos} & \text{item} \\ 1 & 1 & 10 \\ 1 & 2 & 20 \\ 1 & 3 & 30 \end{array} \quad , \quad \text{loop} \equiv \begin{array}{c} \text{iter} \\ 1 \end{array}$$

The variable `$i` bound by the `for`-loop appears in three iterations, containing only one value at a time. Thus, the according encoding is

$$\texttt{\$i} \equiv \begin{array}{ccc} \text{iter} & \text{pos} & \text{item} \\ 1 & 1 & 10 \\ 2 & 1 & 20 \\ 3 & 1 & 30 \end{array} \quad , \quad \text{loop} \equiv \begin{array}{c} \text{iter} \\ 1 \\ 2 \\ 3 \end{array}$$

The sequence `($i,"x")` represents a 2-sequence during each of the three iterations. The encoding reads

$$\texttt{(\$i,"x")} \equiv \begin{array}{ccc} \text{iter} & \text{pos} & \text{item} \\ 1 & 1 & 10 \\ 1 & 2 & \texttt{"x"} \\ 2 & 1 & 20 \\ 2 & 2 & \texttt{"x"} \\ 3 & 1 & 30 \\ 3 & 2 & \texttt{"x"} \end{array} \quad , \quad \text{loop} \equiv \begin{array}{c} \text{iter} \\ 1 \\ 2 \\ 3 \end{array}$$

The whole query, as said, appears in one top-level iteration. Hence a back-mapping step transforms the in-loop representation of the sequence `($i,"x")` to a representation of the constructed sequence:

```
for $i in (10,20,30)
return ($i,"x")
```
$\equiv$
iter	pos	item
1	1	10
1	2	"x"
1	3	20
1	4	"x"
1	5	30
1	6	"x"

, loop $\equiv$
iter
1

which represents the query result `(10,"x",20,"x",30,"x")`.

An empty sequence is represented by iterations that are present in the loop-relation, but not in the sequence encoding. In the following example, variable `$v` lives in a scope that is iterated over three times, and it represents the sequences `("a")`, `()`, and `(23,"b")` during the first, second, and third iteration.

$v $\equiv$
iter	pos	item
1	1	"a"
3	1	23
3	2	"b"

, loop $\equiv$
iter
1
2
3

4.2.3.1 Sequences that contain XML nodes

If an XQuery item sequence contains an XML node, the corresponding encoding simply refers to the respective entry in the pre|post table. The notation γ_1 is used to depict the *node surrogate* of the XML node with ID 1.

A small example demonstrates this: Consider the sequence containing the number 42, and the element node `<c>` from the document used on page 38. With the document encoded as above, this very sequence is encoded by

`(42, <c>bar<d/></c>)` $\equiv$
iter	pos	item
1	1	42
1	2	γ_3

, loop $\equiv$
iter
1

where γ_3 identifies node `<c>` in the document.

* * *

With the relational encodings in place, the remaining task for the Pathfinder compiler is to translate the XQuery expressions into Relational Algebra expressions that work on the encoded data to return a result that is an encoding of the result to be expected from evaluating the XQuery expression. This is completely covered in [21]. To show the extensions made for Pathfinder^{FT} in a proper context, Chapter 5 partially replicates that work by showing the complete compilation rules.

4.3 Pathfinder^{FT} as a compilation phase

The compilation rules provided by the PATHFINDER^{FT} compiler can be seen as an alternative initial phase for the original PATHFINDER compiler, see Figure 2 on page 37. PATHFINDER^{FT} compiles XQUERY FULL TEXT into Relational Algebra plans as used by the original PATHFINDER compiler. Intentionally, no optimisation takes place in this phase. The created plan is then further processed by the PATHFINDER compiler, which cares about optimisation. Hence, the PATHFINDER compiler does not need to know about scoring and scoring models. The only required property of PATHFINDER is that it allows the extension of its XQUERY sequence representation with one further column to host scores.

The PATHFINDER^{FT} compilation separates parts of the query that can be handled by the PATHFINDER compiler, from those parts that need processing by the Full Text machinery. The PATHFINDER compiler optimises the created plan, without further knowledge about the Full Text language or its interpretation. Only the executing engine is required to have access to a Full Text back-end, which is called via function calls embedded in the emitted Relational Algebra code. PF/TIJAH is used in the current implementation.

Section 2.2.2 already mentions the idea of regarding XQUERY FULL TEXT items as pairs of an XQUERY value and a score. The main idea of this work is to extend PATHFINDER's iter|pos|item scheme presented in the previous section with one additional, and ubiquitous score attribute that stores the score of the item in the corresponding row.

The compilation from XQUERY FULL TEXT to Relational Algebra makes the score propagation explicit, which was previously implicit in XQUERY FULL TEXT (see Section 2.2.2.4). The required changes made to the original PATHFINDER's compilation form the core of this work.

The scores are ubiquitous in that this addition of a score column is done *for all* items that occur in an XQUERY FULL TEXT query, *i.e.*, PATHFINDER's sequence encoding is changed from iter|pos|item to iter|pos|item|score in *all* compilation rules. Even though some of the attached scores will never be used, doing so improves the orthogonality of the generated Relational Algebra expressions. It is a matter of optimisation to remove the scores, and their calculation, from all items and expressions that do not make use of them.

In fact PATHFINDER will later prune unused score columns, and related calculations from the query plans. Ideally, the Relational Algebra plan of a query that does not make use of full-text extensions should look the same, no matter whether it was compiled with PATHFINDER^{FT} or the original PATHFINDER compiler. Unfortunately, this is not true for the current prototype implementation, because it already generates different plans for plain XQUERY queries. It is the case, however, that a plan compiled by PATHFINDER^{FT} from an XQUERY query does not contain scoring related operations after optimisation, even though they have been introduced for each and every subexpression in the initial plan generation.

Chapter 5

The Compilation Rules

In this chapter, the compilation rules of the PATHFINDER^{FT} compiler are explained, as well as their modification compared to the original PATHFINDER compiler. Those parts of the rules that form the score propagation are outhoused, to form an interface for a *scoring model*. This is done here by giving them function names starting with the prefix `sm`.

While examples of possible implementations of a scoring model are given, the intention is not to describe precisely how score propagation should happen, but rather how it can be implemented on top of PATHFINDER. Hence, the given implementations of the scoring model functions should be considered examples, they are not intended to work as a top-notch IR system. Instead, I have tried to point out peculiarities and pitfalls that come along with implicit score propagation. See Section 2.3.3 for a discussion of a different interpretation of scores. To implement them, the line between score propagation and value calculation would blur, and the compilation rules below would need some rewriting.

Being a proof of concept prototype, PATHFINDER^{FT} lacks several features. Among these, I consider the complete absence of a type system most crucial. This bans language constructs like type-switches, correct recognition of predicates (although I do show their compilation, once they are identified), or functions like `fn:boolean()`.

Most of the XQUERY constructs have the same form in XQUERY Core. For brevity I describe their compilation as if the intermediate step via XQUERY Core would be omitted, *i.e.*, I show a direct compilation of a core language. Only for those cases where the intermediate step is required, *e.g.*, quantified expressions (Section 5.15), or accessing XML structures (Section 5.18), it is explained in detail.

5.1 Notation and Relational Algebra operators

PATHFINDER^{FT}'s target language is the Relational Algebra understood by the PATHFINDER compiler. Table 1 on page 44 summarises the Relational Algebra operators used, which does not completely display the operators available in the real-world PATHFINDER compiler, see Chapter 1.

Juxtaposition is used for function application (*i.e.*, $f\ x$ instead of $f(x)$), and multiple arguments are just named one after the other (*i.e.*, $f\ x\ y$ instead of $f(x, y)$). Curried notation is used, and

a | b 1 | "foo" 2 | 42	Relations may be specified literally. Here: Schema a|b, and the tuples $\{(1, \texttt{"foo"}), (2, 42)\}$. The attributes are of polymorphic type.
$\pi_{\substack{\mathsf{foo} \\ \mathsf{bar:qux}}}\ r$	Projection of relation r on the attributes foo, and qux, the latter being renamed to bar. Multiple attribute renamings can be specified.
$@_{\mathsf{a}:v}\ r$	Attach a new attribute a with constant value v to relation r.
$\sigma_{\mathsf{a}}\ r$	Select those tuples from the relation r where the Boolean attribute a is true.
$r \uplus s$	The disjoint union of relations r, and s.
$r \setminus s$	Difference, *i.e.*, all tuples in r that are not in s.
$r \underset{\mathsf{a=b}}{\bowtie} s$	The join of the relations r and s, the predicate being equivalence of the attributes a and b.
$r \times s$	The Cartesian product of the relations r and s.
$\varrho_{\mathsf{e}:(\mathsf{a}_1,\ldots,\mathsf{a}_n)/\mathsf{g}}\ r$	Row *numbering* of relation r groups the tuples by a grouping attribute g if it is provided, and (group-wise) densely enumerates the tuples in r, according to the sort key $(\mathsf{a}_1, ..., \mathsf{a}_n)$. The enumeration is stored in attribute e, tuples with duplicate sort keys receive *different* numbers.
$\rho_{\mathsf{k}:(\mathsf{a}_1,\ldots,\mathsf{a}_n)}\ r$	Row *ranking* ranks the tuples in r, according to the sort key $(\mathsf{a}_1, ..., \mathsf{a}_n)$. The ranking is stored in attribute k. In contrast to ϱ, tuples with duplicate sort keys receive *the same* ranking. Grouping is not available. This operator can be used to identify groups in a relation which are determined by multiple keys.
$\delta\ r$	Duplicate removal. PATHFINDER uses bag semantics Relational Algebra.
$\text{⌐}\!\!\text{┐}_{\alpha,n}\ r$	An XPATH axis step with axis α and node test n. See Section 5.8.
$\text{ELEM}_{\substack{\mathsf{iter} \\ \mathsf{item}}}\ r$	Element construction used for *twigs*, see Section 5.19.
$\text{CONT}_{\substack{\mathsf{iter} \\ \mathsf{item} \\ \mathsf{pos}}}\ r$	Content construction used for *twigs*, see Section 5.19.
$\text{agg}_{\mathsf{a}:f\ \mathsf{b/g}}\ r$	Aggregation of relation r groups the tuples by a grouping attribute g if it is provided, and (group-wise) aggregates the values in column b, using function f, and returns the result value in attribute a. The schema of the resulting relation is a|g if grouping is performed, or just a otherwise.
$\text{op}_{\mathsf{a}:\mathsf{b}+\mathsf{c}}\ r$	This tuple-wise operation adds a new attribute a, which contains the sum of the attributes b and c.
$\text{FUN}_f\ l\ [r_1, ..., r_n]$	Call a built-in Relational Algebra function f, passing loop-relation l and a list of argument relations r_1 through r_n. See Section 5.21.2.

Table 1: Operators of the target Relational Algebra language. The infix operators (*i.e.*, $\uplus$, $\setminus$, $\bowtie$, and $\times$) associate to the left, and have lower precedence than the prefix operators, which bind just as function application does.

function application associates to the left:

$$f\ x\ y \quad \equiv \quad (f\ x)\ y \quad \not\equiv \quad f\ (x\ y)$$

Parenthesis are used for grouping, function application always binds tighter than infix operators, prefix operators bind as function application does. Infix operators associate to the left unless otherwise noted. The usual precedence rules apply for arithmetic expressions, and I'd rather use parenthesis for grouping than defining a precedence for each of the Relational Algebra operators in Table 1 on page 44.

Function *composition* is denoted by the infix operator $\circ$, *i.e.*, $(f\ 3 \circ g)\ x \equiv f\ 3\ (g\ x)$.

The infix *application* operator $\lhd$ associates to the *right*. Basically $f \lhd x$ means just $f\ x$, but $\lhd$ has lowest precedence, and thus serves as a means to reduce parenthesis:

$$e\ (f\ (g\ (h\ x))) \quad \equiv \quad (e \circ f \circ g \circ h)\ x \quad \equiv \quad e \lhd f \lhd g \lhd h\ x \quad \equiv \quad e \lhd (f \lhd (g \lhd (h\ x)))$$

Haskell programmers recognise their precious `$` operator here, others may understand it as a "data flow" operator, which provides the output of its right argument as input to its left argument — since $\lhd$ associates to the right, data "flows" from right to left.

Tuples are represented as usual $(1, 2, 3)$, lists are denoted by brackets instead of parenthesis: $[1, 2, 3]$. Note that this is Relational Algebra notation, XQuery, of course, uses parentheses for item sequences, and does not know about tuples.

Types Types of functions are given (only rarely, but then) in a curried style, *i.e.*, $f :: \alpha \to \beta \to \gamma$ declares a function f consuming something of type α and returning a function of type $\beta \to \gamma$. To this end, the type constructor $\to$ associates to the right.

To avoid the restriction of scores to the domain of floating point values, the type Score is used to denote the type used to represent scores.

The type of a relation with schema iter|pos|item is represented by $\mathsf{Rel}_{\mathsf{iter,pos,item}}$. The attributes are of polymorphic type.

The type of a list of values of type α is denoted by $[\alpha]$.

5.2 Compilation Framework

Compilation is defined by the function $[\![\cdot]\!]_{\Gamma,L}$, which maps its argument, an XQuery expression, to the compiled target Relational Algebra expression. Compilation is further parametrised with an *environment* Γ, L, shown as subscript.

As explained in Section 4.2.3, a loop relation L is maintained for each variable scope. It contains a single column named loop that contains all the iterations the respective scope occurs in.

Variables can be bound by `for` and `let`-clauses. The set of bound (*i.e.*, available) variables is stored in the *variable lookup function* Γ in a loop-lifted manner, *i.e.*, it contains the representation of all item sequences a variable refers to during its life in all iterations. Hence, it maps a variable name to an iter,pos,item relation.

5.2.1 Fragments

An XML fragment is a consecutive part of an XML document, or, in other words, a sequence of XML nodes. This may refer to fragments of a document in the database, or to nodes constructed by the query. Different subexpressions of a query may create different fragments:

```
( doc("foo.xml")//a, <b>42</b> )
```

Here, the subexpression `doc("foo.xml")//a` calculates an item sequence that contains node surrogates, each pointing to an `<a>`-node from the fragment which represents the document `foo.xml`. Subexpression `<b>42</b>` returns a singleton item, pointing to a node in a newly constructed fragment. The top-level expression yields an item sequence that refers to nodes from different fragments.

PATHFINDER handles this by calculating a *fragment union* that contains the disjoint union of the fragments of the subexpressions. Disjointness is guaranteed by never reusing node identities. Hence, each compilation of both subexpressions in the above example returns a Relational Algebra expression calculating the corresponding item sequence encoding, and additionally returns a fragment containing the XML nodes referred to by this sequence. Compilation of the top-level expression then returns a Relational Algebra expression calculating the combined sequence, plus the united fragments returned from the subexpressions.

To simplify the presentation, the fragment handling is silently ignored in the following discussion. Basically, one can assume that the compilation of an expression not only returns a Relational Algebra plan to calculate the item sequence, but also a Relational Algebra plan to calculate the respective fragment, *i.e.*, a pair of plans is returned. The step operator discussed in Section 5.8, behaves like a join, and makes use of both plans. In Section 6.7 snippets from the prototype implementation are shown that do demonstrate fragment handling.

* * *

The compilation rules for XQUERY to Relational Algebra compilation have been developed by Torsten Grust *et al.*, [16]. The compilation rules presented here are modifications of the original work as presented in [21], with adaptations that were necessary to keep up with PATHFINDER development, see Figure 1 on page 11.

5.3 Literals

To achieve orthogonality, each item is annotated with some score. Of course, this also applies to literals, although a score on a literal might not make much sense from an IR point of view. However, PATHFINDER$^{\text{FT}}$ will happily accept a query like

```
let score $s := "foo" return $s
```

and therefore needs some *default score* value μ to return.

Depending on the scoring model and the data type used for scores, different values may be used here, see Section 5.23. To represent "not scored", or "no valid score", the XQUERY FULL TEXT draft would allow for -1 here. Supporting back-ends might want to use `null`, but then a mapping to the XQUERY space is required when binding this to a variable.

A literal XQuery item c, is compiled into an iter,pos,item,score relation, where the iter column depends on the current loop-relation.

$$[\![c]\!]_{\Gamma,L} = L \times \left(\mathsf{smDefault}\ \begin{array}{c|c} \mathsf{pos} & \mathsf{item} \\ \hline 1 & c \end{array}\right)$$

where

$\mathsf{smDefault} = @_{\mathsf{score}:1}$

The adoption made is to apply a function smDefault to the pos|item table, which adds the default score. (Throughout this work, the prefix sm is used for function names that implement score propagation of the scoring model. Section 5.23 gives an overview.) This implementation of smDefault attaches a score of 1 to the argument.

5.4 Variables

The variable lookup function Γ can be taken directly from the original Pathfinder compiler. The returned values will already carry the attached score columns. Also, loop-lifting has already been performed on all variables in Γ.

$$[\![v]\!]_{\Gamma,L} = \Gamma\ v$$

Although not depicted in the formula, a variable not only carries the encoded item sequence, but also the associated fragment information.

5.5 Sequences

A sequence of expressions is compiled, by attaching a column ord to each of them, containing an integer value that represents the position of the expression within the sequence. The disjoint union is calculated, and the newly introduced columns are projected away. As the only extension, during this last step it is required to keep the score column alive.

$$= \pi_{\substack{\mathsf{iter} \\ \mathsf{pos}:\mathsf{pos1} \\ \mathsf{item} \\ \mathsf{score}}} \triangleleft\ \varrho_{\substack{\mathsf{pos1}:(\mathsf{ord},\mathsf{pos}) \\ /\mathsf{iter}}} \triangleleft\ \left(@_{\mathsf{ord}:1}[\![e_1]\!]_{\Gamma,L} \uplus \ldots \uplus @_{\mathsf{ord}:n}[\![e_n]\!]_{\Gamma,L}\right)$$
$$[\![(e_1, \ldots, e_n)]\!]_{\Gamma,L}$$

5.6 The `let`-clause

XQuery Full Text adds a new keyword `score` to the `let`-clause, which can be used to bind a score to a variable. The proposed usage is

```
let score $s := $doc/a[. contains text "foo"] return $s
```

to bind the variable `$s` to a *single combined* score, which is some sort of combination of the scores of the items returned by the assigned expression.

It is worth noting that the expression `$doc/a[. contains text "foo"]` may return an item sequence containing multiple `<a>`-nodes with different scores. Nevertheless variable `$s` is bound to a singleton score.

Although the XQuery Full Text draft does not allow for the binding of a score variable and a "normal" variable by the same `let`-clause, PathFinder$^{\text{FT}}$ happily compiles

```
let $x score $s := $doc/a[. contains text "foo"]
return ($x,$s)
```

which would —according to the XQuery Full Text draft— require the more verbose but less elegant syntax

```
let $x := $doc/a[. contains text "foo"]
let score $s := $x
return ($x,$s)
```

In the proposed data model, this is only a matter of calculating the new score from a list of already calculated scores, and to enrich the environment Γ with bindings for the newly introduced variables i and s. Thus, the compilation of a `let`-clause is as follows:

$$[\![\texttt{let } i \texttt{ score } s \texttt{ := } e_1 \texttt{ return } e_2]\!]_{\Gamma,L} = [\![e_2]\!]_{\Gamma',L}$$

where

$$\Gamma'\ v = \begin{cases} [\![e_1]\!]_{\Gamma,L} & \text{if } v \equiv i \\ \mathsf{smDefault} \lhd @_{\mathsf{pos}:1} \lhd \pi_{\substack{\mathsf{iter} \\ \mathsf{item}:\mathsf{score}}} \lhd \mathsf{smLet}\ L\ [\![e_1]\!]_{\Gamma,L} & \text{if } v \equiv s \\ \Gamma\ v & \text{otherwise.} \end{cases}$$

Adding the case where $v \equiv s$ to the lookup function is the only extension required. Note that the scores from e_1 become values during this step, and hence need a score attached. This is done by applying smDefault.

Unfortunately, a score must be provided even for the empty sequence. Consider the following, where a score is to be constructed out of nothing:

```
let score $s := () return $s
```

The loop relation L is used to calculate the empty sequences in e_1, and the following implementation of smLet returns the default score for empty sequences. For non-empty lists, *e.g.*, the scores can be averaged.

$$\mathsf{smLet}\ L\ e_1' = \mathrm{agg}^{\substack{\mathsf{score}: \\ \mathsf{avg\ score} \\ /\mathsf{iter}}}\ e_1' \uplus \mathsf{smDefault}\ (L \setminus \pi_{\mathsf{iter}}\ e_1')$$

5.7 The `for`-clause

The `for`-loop implements the concept of *loop lifting* as exemplified in Section 4.2.3. XQuery Full Text adds a new keyword `score` to the `for`-clause, as in

```
for $v at $p score $s
in $doc/a[. contains text "foo"]
order by $s
return <item>
         <pos>{ $p }</pos>
         <score>{ $s }</score>
         <value>{ $v }</value>
       </item>
```

Its semantics is that in the scope of the return and order-by clauses, the variable `$s` is bound to the score of the current item (which is bound to `$v` here; its position is bound to `$p`).

Compilation is easily extended with the scoring infrastructure: The outermost projection is extended with the target column score, while the back-mapping step remains unchanged.

$$= \pi_{\substack{\mathsf{iter:outer}\\ \mathsf{pos:pos1}\\ \mathsf{item}\\ \mathsf{score}}} \lhd \varrho_{\substack{\mathsf{pos1:(sort1,\ldots,sort}n\mathsf{,inner,pos)}\\ /\mathsf{outer}}} \lhd M_1 \underset{\mathsf{inner=iter}}{\bowtie} e_2' \qquad \llbracket \texttt{for } i \texttt{ at } p \texttt{ score } s \texttt{ in } e_1 \texttt{ order by } o_1, \ldots, o_n \texttt{ return } e_2 \rrbracket_{\Gamma,L}$$

The input expression e_1, the loop relation L' and the order-retaining map relation M are translated just as in the original compiler.

$$e_1' = \llbracket e_1 \rrbracket_{\Gamma,L}$$
$$K = \varrho_{\mathsf{inner:(iter,pos)}}\ e_1'$$
$$L' = \pi_{\mathsf{iter:inner}}\ K$$
$$M = \pi_{\substack{\mathsf{outer:iter}\\ \mathsf{inner}}}\ K$$

The only prominent extension happens to the variable lookup function Γ', which is enriched with a new binding for the score variable s. The newly introduced items, which are bound to the variables p and s respectively, are annotated with a new default score, as already discussed for the `let`-clause and literals.

$$\Gamma'\ v = \begin{cases} @_{\mathsf{pos:1}} \lhd \pi_{\substack{\mathsf{iter:inner}\\ \mathsf{item}\\ \mathsf{score}}} K & \text{if } v \equiv i, \\ \mathsf{smDefault} \lhd @_{\mathsf{pos:1}} \lhd \pi_{\substack{\mathsf{iter:inner}\\ \mathsf{item}}} \lhd \varrho_{\substack{\mathsf{item:pos}\\ /\mathsf{iter}}} \lhd \pi_{\substack{\mathsf{inner}\\ \mathsf{iter}\\ \mathsf{pos}}} K & \text{if } v \equiv p, \\ \mathsf{smDefault} \lhd @_{\mathsf{pos:1}} \lhd \pi_{\substack{\mathsf{iter:inner}\\ \mathsf{item}}} \lhd \pi_{\substack{\mathsf{inner}\\ \mathsf{item:score}\\ \mathsf{iter}\\ \mathsf{pos}}} K & \text{if } v \equiv s, \\ \pi_{\substack{\mathsf{iter:inner}\\ \mathsf{pos}\\ \mathsf{item}\\ \mathsf{score}}} (M \underset{\mathsf{outer=iter}}{\bowtie} \Gamma\ v) & \text{otherwise.} \end{cases}$$

The algebraic expression for the case "$v \equiv s$" is almost only a copy of the expression used for "$v \equiv p$", the only difference being that the item column is not calculated by row numbering, but projected from the score column instead, which is exactly the semantics intended.

To comply with orthogonality, the positions bound to variable p need to have the default score attached as well.

With Γ' and L' the compilation of the return expression e_2 and the order-by expressions o_i are straightforward,

$$e_2' = [\![e_2]\!]_{\Gamma',L'} \quad \text{and} \quad o_i' = [\![o_i]\!]_{\Gamma',L'} \text{ where } 1 \leq i \leq n.$$

Also, the respectively reordered back-mapping relation M_1 is calculated as in the PATHFINDER compiler with

$$M_i = \begin{cases} M & \text{if } i \equiv n+1, \\ \pi_{\substack{\mathsf{outer} \\ \mathsf{inner} \\ \mathsf{sort}i\mathsf{:item} \\ \mathsf{sort}(i+1) \\ \vdots \\ \mathsf{sort}n}} (o_i' \underset{\mathsf{iter}=\mathsf{inner}}{\bowtie} M_{i+1}) & \text{if } 1 \leq i \leq n. \end{cases}$$

5.8 Axis steps

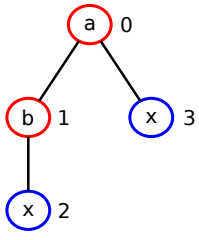
It is desirable to propagate the score of a context node to the target nodes found through an axis step. While one context node can have several target nodes in the direction of the axis, it is also possible that multiple context nodes lead to the same target node. According to XPATH specifications, there must not be duplicates in the result of an axis step. This gives rise to a function smStep that maps the scores of all related context nodes to the score of the target node.

PATHFINDER's step operator behaves like a join between the relation encoding the document (the type of which is denoted by $\mathcal{D}$), and the sequence of context nodes respectively:

$$\text{⏍}_{\alpha,n} :: \mathcal{D} \to R_{\mathsf{iter,item}} \to R_{\mathsf{iter,item,ctx}}$$

This operator does *not* remove duplicates as required by XPATH semantics. Instead, for each context node all target nodes are returned, and annotated with the respective context node in the ctx column. The pos columns are not used here, since argument and result are considered sets of nodes. Only when calculating the final item sequence the document order needs to be restored by adding a pos column.

Consider the XML document $d =$ `<a><b><x/></b><x/></a>`, with the displayed tree representation. The numbers indicate node IDs.



Performing an axis step `/descendant::x` from the context set (γ_0, γ_1) shown in red, returns the result set (γ_2, γ_3) shown in blue. Note that γ_2 is a target from both context nodes, while γ_3 is not a descendant of γ_1. This situation is reflected in the context attribute ctx returned from the step operator.

iter	item
1	γ_0
1	γ_1

$\xrightarrow{\text{⏍}_{\texttt{/descendant::x}}\ d}$

iter	item	ctx
1	γ_2	γ_0
1	γ_3	γ_0
1	γ_2	γ_1

The introduced duplicates need to be removed by another Relational Algebra operator, and the result nodes have to be annotated with pos values reflecting the document order.

Before doing so, PATHFINDER$^{\text{FT}}$ simply adds a score column to the input relation, and the step operator willingly copies the scores along with the context information. This not only introduces duplicates as above, it also propagates the score information from the context nodes to the target nodes. Assuming the context set with scores to be $(\langle\gamma_0|0.3\rangle, \langle\gamma_1|0.5\rangle)$, the calculation reads

iter	item	score
1	γ_0	0.3
1	γ_1	0.5

$\xrightarrow{\text{◿//x } d}$

iter	item	ctx	score
1	γ_2	γ_0	0.3
1	γ_3	γ_0	0.3
1	γ_2	γ_1	0.5

It is now the responsibility of the surrounding Relational Algebra code to remove duplicates, add a pos column, and aggregate the scores of identical target nodes:

$$[\![e/\alpha::n]\!]_{\Gamma,L} = \pi_{\substack{\text{iter}\\ \text{pos}\\ \text{item}\\ \text{score}}} \triangleleft \varrho_{\substack{\text{pos:item}\\ /\text{iter}}} \triangleleft \delta(\pi_{\substack{\text{iter}\\ \text{item}\\ \text{group1:group}}} g) \underset{\text{group=group1}}{\bowtie} \text{smStep } g$$

where

$$g = \rho_{\text{group:(iter,item)}} \triangleleft \text{◿}_{\alpha,n} \triangleleft \pi_{\substack{\text{iter}\\ \text{item}\\ \text{score}}} [\![e]\!]_{\Gamma,L}$$

$$\text{smStep} = \text{agg}_{\substack{\text{score:}\\ \text{avg score}\\ /\text{group}}}$$

Fragment handling is not shown in this formula. The truth is that ◿$_{\alpha,n}$ uses the fragment information (d in the above example) returned from the compilation of e to perform the axis step on the nodes identified by the context set. This technical detail is omitted here to unclutter the formula, see Section 5.2.

Note that the result of the axis step is of type Rel$_{\text{iter,item,ctx,score}}$, and must be partitioned into groups identified by all *pairs* of iter|item — for each iteration, identical items (*i.e.*, nodes) must have their scores aggregated. Since, however, the Relational Algebra operation for aggregation only supports grouping according to *one* attribute, the groups must be identified first, which is done when calculating g.

The above implementation of smLet uses the average to aggregate the scores of identical nodes.

5.9 Direct score manipulation

The PATHFINDER$^{\text{FT}}$ compiler adds a keyword `scored` to the XQUERY FULL TEXT language, which is not defined in the XQUERY FULL TEXT draft. e_1 `scored` e_2 returns the item sequence e_1 with all scores set to e_2, which is thus required to be a singleton value of type Score.

```
let score $s := "foo" scored 0.3
return $s
```

evaluates to `0.3`. The implementation is straightforward:

$$[\![e_1 \texttt{ scored } e_2]\!]_{\Gamma,L} = \pi_{\substack{\text{iter}\\ \text{pos}\\ \text{item}\\ \text{score}}} (\pi_{\substack{\text{iter}\\ \text{pos}\\ \text{item}}} [\![e_1]\!]_{\Gamma,L} \underset{\text{iter}=\text{iter1}}{\bowtie} \pi_{\substack{\text{score:item}\\ \text{iter1:iter}}} [\![e_2]\!]_{\Gamma,L})$$

The score of e_2 is lost in this operation. To allow for the application of scored scores, the following construction may be used:

```
let $s score $ss := e2 return e1 scored $s * $ss
```

To merely alter the scores of the items in a sequence, *e.g.*, by uniformly scaling them by e_2, the user may resort to a map construction as the following, which can be wrapped around any XQuery item sequence e_1.

```
for $i score $s in e1 return $i scored e2 * $s
```

The transition from value-space to score-space offered by the `scored` operator is somewhat dual to what `for` and `let` offer: These make a score available as value, and allow its interpretation by XQuery means. `scored` works in the opposite direction by moving a value beyond XQuery's horizon, and attaching it as score to another item.

5.10 Boolean operators

In XQuery, functions and operators are used to calculate some outcome based on their arguments. XQuery Full Text score propagation adds the task of assigning a meaningful score to the result value, potentially depending on the scores of the arguments.

The need for score propagation is most evident with Boolean operators (`and`, `or`, and `not`) when used inside a predicate to combine the results of Full Text expressions:

```
for $i score $s
in doc("lib.xml")//book[
             ./authors contains text "Wallace" ftand "Gromit"
         and ./abstract contains text "cheese"
                        ]
order by $s descending
return $i/title
```

Note the distinction between the two "and" operators used: `ftand` is part of the Full Text language, and it is used by the Full Text engine to rate the `<authors>` nodes depending on their combined relevance with respect to the search terms `"Wallace"`, `"Gromit"`. In a separate operation, the Full Text engine rates `<abstract>` nodes depending on their relevance with respect to the search term `"cheese"`. These two findings are then combined *outside* the Full Text machinery in the final scored Boolean.

Pathfinder$^{\text{FT}}$ achieves this by adding simple score propagation functions to the involved compilation rules. *E.g.*, the operator `and` can be compiled as follows:

$$= \begin{array}{l} [\![e_1 \texttt{ and } e_2]\!]_{\Gamma,L} \\ @_{\mathsf{pos}:1} \triangleleft \pi_{\substack{\mathsf{iter}\\ \mathsf{item}\\ \mathsf{score}}} \triangleleft \mathsf{smAnd} \triangleleft \mathrm{op}_{\substack{\mathsf{item}:\\ \mathsf{item1}\wedge\mathsf{item2}}} (\pi_{\substack{\mathsf{iter}\\ \mathsf{item1}:\mathsf{item}\\ \mathsf{score1}:\mathsf{score}}} [\![e_1]\!]_{\Gamma,L} \underset{\mathsf{iter}=\mathsf{iter2}}{\bowtie} \pi_{\substack{\mathsf{iter2}:\mathsf{iter}\\ \mathsf{item2}:\mathsf{item}\\ \mathsf{score2}:\mathsf{score}}} [\![e_2]\!]_{\Gamma,L}) \\ \text{where} \\ \mathsf{smAnd} = \mathrm{op}_{\substack{\mathsf{score}:\\ \mathsf{score1}\cdot\mathsf{score2}}} \end{array}$$

This implementation of smAnd multiplies the scores of its argument to calculate the resulting score. The compilation rule for `or` is identical, modulo the obvious adaptation. Score propagation is discussed below. Also, the unary operator `not` entails no surprises:

$$[\![\texttt{not } e]\!]_{\Gamma,L} = \pi_{\substack{\mathsf{iter}\\ \mathsf{pos}\\ \mathsf{item}\\ \mathsf{score}}} \triangleleft \mathsf{smNot} \triangleleft \mathrm{op}_{\substack{\mathsf{item}:\\ \neg\mathsf{item1}}} \triangleleft \pi_{\substack{\mathsf{iter}\\ \mathsf{pos}\\ \mathsf{item1}:\mathsf{item}\\ \mathsf{score1}:\mathsf{score}}} [\![e]\!]_{\Gamma,L}$$

where

$$\mathsf{smNot} = \mathrm{op}_{\substack{\mathsf{score}:\\ \mathsf{score2}-\mathsf{score1}}} \circ @_{\mathsf{score2}:1}$$

Here, the score is inverted by calculating $\texttt{not}(\langle v|s\rangle) = \langle \neg v|1-s\rangle$. To leave the score unmodified, a simple projection is sufficient.

Unfortunately, careless implementation of the three score propagation functions smAnd, smOr, and smNot may cause confusion: We all know that $\neg(a \wedge b) \equiv \neg a \vee \neg b$, and the user *might* also expect this to hold when scores are involved. These two queries should then yield the same result.

```
  let score $s := not (a and b) return $s
≡
  let score $s := (not a) or (not b) return $s
```

Using angle bracket notation $\langle v|s\rangle$ to denote an item with value v and score s, the score propagation for Boolean operators is bound by DeMorgan's law: Given

$$\langle v_1|s_1\rangle \texttt{ and } \langle v_2|s_2\rangle = \langle v_1 \wedge v2|s_1 \cdot s_2\rangle \qquad \text{and} \qquad \texttt{not}(\langle v|s\rangle) = \langle \neg v|s\rangle$$

it immediately follows that

$$\langle v_1|s_1\rangle \texttt{ or } \langle v_2|s_2\rangle \equiv \langle v_1 \vee v_2|s_1 \cdot s_2\rangle$$

should hold as well, at least if the user expects such rewrites to yield the same result.

If the user chooses to invert the score on negation, and leave conjunction as above, it follows by the same argumentation that

$$\langle v_1|s_1\rangle \texttt{ or } \langle v_2|s_2\rangle \equiv \langle v_1 \vee v_2|s_1 + s_2 - s_1 \cdot s_2\rangle$$

PATHFINDER$^{\mathrm{FT}}$ implements both choices, and it is a rather simple task to add further alternatives. The user may set the pragma[1] `pfft:ftBool` to switch between them. If set to `"negInv"` the score is inverted on Boolean negation, if set to `"negId"` the score is left unchanged.

[1] `declare namespace pfft = "http://stefan-klinger.de/ns/pfft/0.1";`

In both cases the scores for conjunction and negation are as discussed above.

If the scores are multiplied on disjunction, the rule reads as the one for conjunction. The following implements the calculation of $s = s_1 + s_2 - s_1 \cdot s_2$:

$$\mathsf{smOr} = \mathrm{op}^{\mathsf{score:}}_{\mathsf{item3-item4}} \circ \mathrm{op}^{\mathsf{item4:}}_{\mathsf{score1 \cdot score2}} \circ \mathrm{op}^{\mathsf{item3:}}_{\mathsf{score1+score2}}$$

PATHFINDER$^{\mathrm{FT}}$ offers a third option, which, however, assumes positive and negative scores. Setting the pragma `pfft:ftBool` to `"extreme"`, the following equations are used:

$$\langle v_1|s_1\rangle \texttt{ and } \langle v_2|s_2\rangle = \langle v_1 \wedge v_2 | \mathsf{min}\ s_1\ s_2\rangle$$
$$\langle v_1|s_1\rangle \texttt{ or } \langle v_2|s_2\rangle = \langle v_1 \vee v_2 | \mathsf{max}\ s_1\ s_2\rangle$$
$$\texttt{not(}\langle v|s\rangle\texttt{)} = \langle \neg v| - s\rangle$$

Using extremes for score propagation on Boolean operations becomes relevant when weights are used to scale scores, (see Section 5.22.4.4). If scores were limited to the range [0..1], one may also choose $1 - s$ instead of just $-s$ for the negation. This would look a bit like fuzzy logic (see, *e.g.*, [20]).

While neither the XQUERY FULL TEXT draft, nor PATHFINDER$^{\mathrm{FT}}$, impose any semantics on scores, it is reasonable to think of scores as probabilities. [19] uses similar formulas as the ones described here to aggregate probabilities. Depending on the aggregation assumption, *i.e.*, whether these are *independent*, *disjoint*, or *subsumed*, a different formula is used. The authors present a modified SQL `select` statement that allows to specify the assumption. Similar, PATHFINDER$^{\mathrm{FT}}$ allows the use of XQUERY FULL TEXT pragmas to specify the aggregation algorithm.

5.11 Conditional expression

For a conditional expression `if` e_0 `then` e_1 `else` e_2, evaluation of the condition e_0 yields a Boolean for each iteration, indicating which of the two branches (exactly one in each iteration) is to be used to calculate the result. The strategy here is to calculate two separate loop relations L_1, L_2, for each case, and to compile the two branches e_1, and e_2 with the environment updated respectively.

$$= \begin{array}{l} [\![\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2]\!]_{\Gamma,L} \\ \pi^{\mathsf{iter}\ \mathsf{pos}\ \mathsf{item}\ \mathsf{score:score2}} \triangleleft \mathsf{smConditional} \triangleleft \pi^{\mathsf{score1:score}\ \mathsf{iter1:iter}} [\![e_0]\!]_{\Gamma,L} \underset{\mathsf{iter=iter1}}{\bowtie} \left([\![e_1]\!]_{\Gamma_1,L_1} \uplus [\![e_2]\!]_{\Gamma_2,L_2} \right) \\ \text{where} \\ L_1 = \pi_{\mathsf{iter}} \triangleleft \sigma_{\mathsf{item}} [\![e_0]\!]_{\Gamma,L} \quad , \quad L_2 = L \setminus L_1 \quad , \quad \Gamma_i\ v = \pi^{\mathsf{iter}\ \mathsf{pos}\ \mathsf{item}\ \mathsf{score}} \triangleleft \Gamma\ v \underset{\mathsf{iter=iter1}}{\bowtie} \pi^{\mathsf{iter1:iter}}\ L_i \\ \mathsf{smConditional} = \mathrm{op}^{\mathsf{score2:score1 \cdot score}} \end{array}$$

The variable lookup functions Γ_i are restricted versions of Γ, where the iterations not used in a branch are stripped from the variable representations.

The score propagation shown multiplies the score of the predicate with the score of the result of the chosen branch.

This violates some equivalences the user might expect, if combined with the choices offered in Section 5.10. To see this, *e.g.*, consider the following two expressions for Booleans `$a`, `$b`, and `$c`:

```
($a and $b) or (not($a) and $c)   vs.   if $a then $b else $c
```

Without scoring involved, the user will expect identical results here. With scoring, however, the left hand side expression always carries score information that is influenced by `$a`, `$b`, *and* `$c`. For the right hand side, *only* the score from `$a`, and either `$b` *or* `$c` will influence the result score.

Doing otherwise is not an option though, because this would leverage the semantics of conditional expressions: First, both branches would have to be evaluated to find the score of the branch that was not selected by the predicate. Second, what should happen with the *value* from that branch?

Other examples that bear the same connection are:

```
if $a then false() else true()   vs.   not($a)
if $a then $b else false()       vs.   $a and $b
if $a then true() else $b        vs.   $a or $b
```

and also the symmetric versions of the latter two. With scoring, these are *not* necessarily equivalent.

I was unable to find conditional expressions in IR literature (*e.g.*, [2]), thus it appears to me that `if-then-else` constructs are not a feature widely used in IR languages. In fact, they do not even appear in the Full Text part of XQuery Full Text. Only since scores are attached to items, it is required for the (formerly pure) XQuery conditional expressions to somehow deal with them.

From a probabilistic point of view, [6] describes how probability inference must look, and the author does not present conditional expressions. The only other paper I have found in that direction, is [12]. The author links programming, Information Theory, and (Subjective) Bayesian probability, and thereby makes use of conditional statements. However, the expressions in his languages always depict statements about some state (*i.e.*, values assigned to variables). This makes it easier to combine both branches of a conditional expression, since both represent probabilities of values being assigned to a set of variables.

To adopt his strategy, it is required to combine two XQuery items into one. An approach to do so is sketched in Section 7.2.

5.12 Node set operations

XQuery offers operations on node sequences that resemble set operations: Two duplicate free sequences of XML nodes can be combined using one of the operators `union`, `intersect`, or `except` (XQuery's node set difference). The result is the result of a set operation (*i.e.*, duplicates are removed), ordered in document order. While these operations are easy to implement in Relational Algebra for the unscored case, they become more expensive as soon as scoring is involved.

Diverging from what is presented here (see Section 2.3.3 why), scores can be seen as a probability of sequence membership. Again, as for Boolean operators (Section 5.10), it may be appropriate do distinguish how these probabilities relate. The corresponding aggregation functions are described in [19] for a Probabilistic Relational Algebra, or, *e.g.*, in [20] for Fuzzy Logic semantics.

5.12.1 Union

To calculate the union it is sufficient to aggregate the scores grouped by iter/item pairs, a construction already used for smStep in Section 5.8:

$$[\![e_1 \texttt{ union } e_2]\!]_{\Gamma,L} = \varrho_{\mathsf{pos:item/iter}} \triangleleft \pi_{\substack{\mathsf{iter}\\ \mathsf{item}\\ \mathsf{score}}} \triangleleft \delta\,(\pi_{\substack{\mathsf{iter}\\ \mathsf{item}\\ \mathsf{group}}}\, g) \underset{\mathsf{group=group1}}{\bowtie} \pi_{\substack{\mathsf{group1:group}\\ \mathsf{score:score1}}}\,(\mathsf{smUnion}\ g)$$

where

$$g = \rho_{\mathsf{group:(iter,item)}} \triangleleft [\![e_1]\!]_{\Gamma,L} \uplus [\![e_2]\!]_{\Gamma,L}$$

$$\mathsf{smUnion} = \mathrm{agg}_{\substack{\mathsf{score1:}\\ \mathsf{avg\ score}\\ \mathsf{/group}}}$$

Since aggregation is used to calculate the resulting score, it would be easy to extend the binary `union` operator to an n-ary one, calculating the union of multiple node sequences in one go.

5.12.2 Intersection

The intersection of two sets of scored elements is more difficult: The intersection is to be calculated on partial tuples, *i.e.*, tuples are to be identified by only some of their attributes, since scores shall not be used to distinguish elements (see the discussion of equality, Section 2.3.2). This bans plain Relational Algebra intersection from being used. Instead, a multi-predicate equi-join[2] is required, pairing those nodes with the same iter/item pairs, and leaving the score information intact.

$$= [\![e_1 \texttt{ intersect } e_2]\!]_{\Gamma,L}$$

$$\varrho_{\substack{\mathsf{pos:item}\\ \mathsf{/iter}}} \triangleleft \pi_{\substack{\mathsf{iter}\\ \mathsf{item}\\ \mathsf{score:score2}}} \triangleleft \mathsf{smIntersection} \triangleleft \pi_{\substack{\mathsf{iter}\\ \mathsf{item}\\ \mathsf{score}}} [\![e_1]\!]_{\Gamma,L} \underset{\substack{\mathsf{iter=iter1}\\ \mathsf{item=item1}}}{\bowtie} \pi_{\substack{\mathsf{iter1:iter}\\ \mathsf{item1:item}\\ \mathsf{score1:score}}} [\![e_2]\!]_{\Gamma,L}$$

where

$$\mathsf{smIntersection} = \mathrm{op}_{\mathsf{score2:score\cdot score1}}$$

5.12.3 Difference

Nodes that occur on the right hand side are simply removed from the set of nodes on the left hand side. The scores of the remaining nodes need to be removed to calculate the intersection, and reattached afterwards.

$$[\![e_1 \texttt{ except } e_2]\!]_{\Gamma,L} = \varrho_{\substack{\mathsf{pos:item}\\ \mathsf{/iter}}} \triangleleft \pi_{\substack{\mathsf{iter}\\ \mathsf{item}\\ \mathsf{score}}} \triangleleft \pi_{\substack{\mathsf{iter1:iter}\\ \mathsf{item1:item}\\ \mathsf{score}}} e'_1 \underset{\substack{\mathsf{iter=iter1}\\ \mathsf{item=item1}}}{\bowtie} (\pi_{\substack{\mathsf{iter}\\ \mathsf{item}}} e'_1 \setminus \pi_{\substack{\mathsf{iter}\\ \mathsf{item}}} [\![e_2]\!]_{\Gamma,L})$$

where

$$e'_1 = [\![e_1]\!]_{\Gamma,L}$$

The scores are not modified in this implementation, which may look strange at first. This, however, is a consequence of the design decision made by the W3C to attach scores to values, see Section 2.3.3.

[2] The current PATHFINDER$^{\mathrm{FT}}$ implementation uses a theta-join, since the available equi-join does not support multiple comparisons

5.13 The function `fn:exists()`

The function `fn:exists()` takes an XQUERY item sequence as argument, and returns false iff the sequence is empty. So the implementation has to check which of the iterations mentioned in the loop-relation are present in the sequence encoding, see Section 4.2.3.

If the semantics of the score is confidence in the accuracy of a value, then a non-empty sequence should be mapped to true with the default score denoting absolute confidence:

```
fn:exists( $doc contains text "foo" )
```

No matter what the outcome of the Full Text expression is, there certainly is a value, namely the Boolean returned from the Full Text engine. *I.e.*, the user might expect true with absolute confidence, no matter what the score assigned by the Full Text machinery actually is.

In contrast, if scores are used to define a graduation of truth, the result score might very well depend on the scores available in the input sequence:

```
fn:exists( $doc/chapter[. contains text "Wallace" ftand "Gromit"] )
```

The returned score might depend on how well some of the chapters match the Full Text expression. In principle, `fn:exists` has *no chance* to relate the number of matching documents to the number of non-matching ones.

Hence, it may be unwise to return false with the default score for an empty input sequence. To this end, PATHFINDER$^{\text{FT}}$ offers two functions, smExist, and smMiss, to calculate the return score of a non-empty or an empty input sequence:

$$\llbracket \texttt{fn:exists(}\ e\ \texttt{)} \rrbracket_{\Gamma,L} = t \uplus f$$

where

$$t = @_{\mathsf{item:true}} \triangleleft \mathsf{smExist} \triangleleft \pi_{\substack{\mathsf{iter}\\ \mathsf{pos}\\ \mathsf{score1:score}}} \llbracket e \rrbracket_{\Gamma,L}$$

$$f = @_{\mathsf{item:false}} \triangleleft \mathsf{smMiss}\ (L \setminus \pi_{\mathsf{iter}} t)$$

An implementation that averages the scores of non-empty sequences, and that uses the default score for empty sequences can be achieved by choosing:

$$\mathsf{smExist} = \mathrm{agg}^{\substack{\mathsf{score:}\\ \mathsf{avg\ score1}\\ /\mathsf{iter}}}$$

$$\mathsf{smMiss} = \mathsf{smDefault}$$

5.14 Other built-in functions

Basically, the concept of score propagation described for the Boolean operators can be extended to handle most built-in functions in the same way (though this might not be wise to do in all cases, see Section 5.13 for an example).

If scores denote confidence in a value, every calculation based on these values requires the confidence to propagate to the result. This can even be done for simple arithmetics.

With the PATHFINDER$^{\text{FT}}$ architecture, it is quite simple to handle score propagation for all such operators. Let $\otimes \in \{+, -, *, \texttt{div}, \texttt{mod}, \texttt{lt}, \texttt{le}, \texttt{gt}, \texttt{ge}, \texttt{ne}, \texttt{eq}\}$ be a binary operator between

singleton items (XQuery uses existential semantics for comparison operators such as `<`, `<=`, `>`, `>=`, `!=`, and `=`. These are handled in Section 5.17, here, only the corresponding value comparison operators are considered).

$$= \begin{array}{l} [\![e_1 \otimes e_2]\!]_{\Gamma,L} \\ @_{\mathsf{pos}:1} \triangleleft \pi_{\substack{\mathsf{iter}:\mathsf{iter1} \\ \mathsf{item} \\ \mathsf{score}}} \triangleleft \mathsf{smFun}_{\otimes} \triangleleft \mathrm{op}_{\substack{\mathsf{item}: \\ \mathsf{item1} \otimes \\ \mathsf{item2}}} (\pi_{\substack{\mathsf{iter1}:\mathsf{iter} \\ \mathsf{item1}:\mathsf{item} \\ \mathsf{score1}:\mathsf{score}}} [\![e_1]\!]_{\Gamma,L} \underset{\mathsf{iter1}=\mathsf{iter2}}{\bowtie} \pi_{\substack{\mathsf{iter2}:\mathsf{iter} \\ \mathsf{item2}:\mathsf{item} \\ \mathsf{score2}:\mathsf{score}}} [\![e_2]\!]_{\Gamma,L}) \end{array}$$

Suitable implementations of all the functions in the $\mathsf{smFun}_{\otimes}$ family certainly depend on the intended score semantics.

* * *

The following sections use XQuery Core as an intermediate compilation step, *i.e.* an XQuery Full Text expression is replaced by an equivalent XQuery Core expression, which is then compiled as discussed above. It is evident that doing so "inherits" the score propagation defined for the more primitive XQuery Core constructs.

5.15 Quantified expressions

Quantified expressions are translated to a combination of `fn:exists()` and a loop:

```
some $v in e1 satisfies e2
```

is translated to

```
fn:exists( for $v
           in e'1
           return if e'2
                  then 1
                  else ()
         )
```

where the loop creates a sequence that contains the integer `1` for each item in e_1 that makes e_2 evaluate to true when bound to variable `$v`. e'_i is the XQuery Core version of e_i.

The all-quantification

```
every $v in e1 satisfies e2
```

is simply translated to

```
not(some $v in e1 satisfies not(e2))
```

which is then processed as above.

These compilation rules inherit score propagation from the `fn:exists` function (Section 5.13), and from conditional expressions (Section 5.11). *I.e.*, smExist, smMiss and smConditional are involved here.

The scores of values in e_1 is lost in the term

```
for $v in e1 return if (e2) then 1 else ()
```

since the integer `1` is generated with the default score. A slight adaption can be used to propagate the scores from e_1 as well:

```
fn:exists( for $v score $s
           in e1'
           return if e2'
                  then 1 scored $s
                  else ()
         )
```

In this transformation the score extracted from the current item in e_1 and bound to variable `$s` by the `for`-loop, is reattached to the generated integer `1` by the `scored` keyword (Section 5.9).

5.16 Predicates

XQuery offers three kinds of predicates, namely *Boolean*, *existential*, and *positional* ones. All three are subject to score propagation.

5.16.1 Boolean Predicates

For a Boolean predicate e_2, the XQuery Core representation of e_1`[`e_2`]` is

```
for dot
in e1'
return if e2'
       then dot
       else ()
```

where dot is the iteration variable available as "`.`" in the predicate, and e_1' and e_2' refer to XQuery Core versions of e_1 and e_2 respectively.

This inherits the score propagation from `for`-loops and conditionals. Since the empty sequence is returned from the false branch the way scores are calculated there is irrelevant.

5.16.2 Existential Predicates

Existential predicates can be traced back to Boolean predicates by using the function `fn:exists`. For a (potentially empty) sequence of nodes e_2, the XQuery Core representation of e_1`[`e_2`]` is

```
for dot
in e1'
return if fn:exists(e2')
       then dot
       else ()
```

with dot, e_1', and e_2' as above. This inherits score propagation as do Boolean predicates, and also from function `fn:exists`.

5.16.3 Positional Predicates

Positional predicates are compiled in a similar way, using the positional variable available from the `for`-clause. If e_2 is a singleton numeric, the expression e_1`[`e_2`]` is compiled to

```
for dot at $cp
in e'1
return if $cp eq e'2
       then dot
       else ()
```

with dot, e'_1, and e'_2 as for Boolean predicates. This inherits score propagation as do Boolean predicates, and also from comparison operation, *i.e.*, via $\mathsf{smFun}_{\mathsf{eq}}$.

5.17 General comparison

The XQuery *general comparison* operators `<`, `<=`, `>`, `>=`, `!=`, and `=` come with existential semantics[3]. This can be implemented with quantified expressions: Any XQuery expression containing one of the above operators is translated into an XQuery expression using the corresponding *value comparison* operator (see Section 5.14):

The expression

e_1 `=` e_2

is translated to

```
some $i
in e'1
satisfies some $j
          in e'2
          satisfies $i eq $j
```

introducing, as usual, new variables `$i` and `$j`, and e'_i the XQuery Core version of e_i.

This translation inherits the score propagation implemented for quantified expressions. Thus, the resulting score depends on smExist, smMiss, smConditional and the translation to XQuery Core chosen for quantified expressions (see Section 5.15).

5.18 Accessing XML structures

Every time XML structures are used in an XQuery query, be it via document access, or by explicit node construction, Pathfinder implements this via *constructors*. The idea is that such a constructor, member of the Relational Algebra, returns something thatt contains an XML fragment encoded in the way that Pathfinder uses for document encoding (see Section 4.2.2), together with a representation of an XQuery item sequence (see Section 4.2.3) referring to the roots of the fragment by node surrogates.

[3] `http://www.w3.org/TR/2007/REC-xquery-20070123/#id-general-comparisons`

Two Relational Algebra operators are used then to split the constructor's output in two: FRAG returns the relational encoding of the fragment, the details of which are not of interest here (see Section 4.2.3), ROOTS returns the XQUERY item sequence that "contains" nodes from the fragment.

Constructors are created by two Relational Algebra constructor functions

consTwig :: Twig → Constr

consDoc :: Core → Constr

the former is used to create XML nodes (see Section 5.19), the latter to access the database's document table (see Section 5.20).

Unfortunately, however, the Relational Algebra understood by the PATHFINDER compiler handles the result of a constructor as something special. As a consequence of that, one cannot simply attach a score column to a constructor, which is why I use the type name Constr here to distinguish it from the other relations with known schema. Thus, the scores of all relational input to the constructor must be collected, combined, and the resulting score must be attached to the relational output of the "ROOTS" branch of the constructor.

In other words: Take the whole constructor as one single big Relational Algebra operator, with the number of arguments depending on the constructor's structure. Bypass the scores around that operator.

To translate a constructor with score propagation, PATHFINDER$^{\text{FT}}$ resorts to a different translation scheme $[\![\cdot]\!]^{*}_{\Gamma,L}$, which returns a pair of two relational algebra plans: One for the constructor construction, and one for the score propagation. The scores appear in a loop-lifted fashion here, *i.e.*, they are encoded in an iter,score relation.

Compilation of twigs and document access is described in Section 5.19, and Section 5.20 respectively. What remains when one of these constructors, name it c, is compiled with the extended compilation scheme, is to access its root nodes and to add the scores returned from its compilation. So let $(c', s) = [\![c]\!]^{*}_{\Gamma,L}$, then:

$$[\![c]\!]_{\Gamma,L} = @_{\mathsf{pos}:1} \triangleleft \pi_{\substack{\mathsf{iter}\\ \mathsf{item}\\ \mathsf{score}}} (s \underset{\mathsf{iter}=\mathsf{iter1}}{\bowtie} \pi_{\mathsf{iter1}:\mathsf{iter}}(\mathrm{ROOTS}\ c'))$$

This rule, for simplicity and as all other rules in this discussion, does not show the use of FRAG.

5.19 Element construction

For element construction, PATHFINDER uses *twigs*, specialised operators available at the Relational Algebra level, utilised to construct larger portions of XML documents —twigs— in one go.

Taking a simplified view here, twigs are constructed by two constructor functions:

twigElem :: Core → [Twig] → Twig

twigContent :: Core → Twig

To create an element node, twigElem takes an XQUERY Core expression that calculates the name of the element node, and a list of twigs that calculate the contents of the node. The returned twig represents the calculation that builds the complete XML fragment. Element content calculated by an XQUERY Core expression can be added to a twig with the twigContent function.

The goal of this construction is to create XML structures bigger than a single node in one go, to avoid the cost of creating and encoding too many XML fragments.

If score propagation is desired on node construction, *i.e.*, if the score of a newly created node shall depend on the scores of its content, then a way must be found to hand on scores between the twig constructor functions.

So the compilation of element construction takes two steps:

- Construct a twig covering as much constructors as possible.
- Construct Relational Algebra code that collects input scores and attaches the result score to the root of the constructed twig.

No scores are assigned to the inner nodes of the twig. The returned twig is represented as a node surrogate in a singleton item sequence (see Section 4.2.3.1), thus bears only one single score. The XML fragment constructed by the twig is in the fragment storage, and PATHFINDER$^{\text{FT}}$ does not assign scores to XML nodes there.

Thus, descending into a constructed twig "rearranges" the scores:

```
for $i score $s
in <a>{ <x/> scored 0.3
      , <y/> scored 0.7
      }</a>/*
return ($i, $s)
```

returns `(<x/>, 0.5, <y/>, 0.5)` if the average of the content scores is used as the score of the twig. The axis step then propagates the accumulated score at the twig's root down to its result nodes (which does not necessarily distribute the scores evenly, see Section 5.8).

The twig construction is almost directly mapped to Relational Algebra primitives, the remaining task for PATHFINDER$^{\text{FT}}$ is to collect the scores at all entry points of the twig, and to pass them on to the twig result via some score propagation functions:

$$[\![\mathsf{twigElem}\ e\ [t_1, ..., t_n]]\!]^*_{\Gamma,L} = \left(\text{ELEM}^{\mathsf{iter}}_{\mathsf{item}}\ \ e'\ [t'_1, ..., t'_n]\ \ ,\ \mathsf{smElem}\ L\ e'\ (s_1 \uplus ... \uplus\ s_n)\right)$$

The node name construction is compiled using the normal scheme

$$e' = [\![e]\!]_{\Gamma,L}$$

returning an item sequence using the usual encoding.

The twigs that form the content are compiled using the extended scheme, yielding twig/score pairs

$$(t'_i, s_i) = [\![t_i]\!]^*_{\Gamma,L}$$

The scores $s_1, ..., s_n$ returned from compiling the content twigs are handed on to the score propagation function smElem *together* with the score returned from compiling the XQUERY Core code for the element name. The score of the constructed twig can thus depend on the score of the element name as well:

```
for $i score $s
in element { "a" scored 0.3 } { 42 scored 0.7 }
return ($i, $s)
```

returns `(<a>42</a>, 0.21)`, if the result score is the product of the element name score and the element content score.

An implementation of smElem that multiplies the element name score with the average of the content scores is given next. The average of the empty content cannot be calculated, which is why the default score is used here.

$$\mathsf{smElem}\ L\ e'\ c = \pi_{\substack{\mathsf{iter}\\ \mathsf{score:score2}}} \triangleleft\ \mathrm{op}_{\substack{\mathsf{score2:}\\ \mathsf{score\cdot score1}}} \triangleleft\ (a \uplus s) \underset{\mathsf{iter=iter1}}{\bowtie} \pi_{\substack{\mathsf{iter1:iter}\\ \mathsf{score1:score}}}\ e'$$

where

$$a = \mathsf{smDefault}\ \triangleleft\ L \setminus \pi_{\mathsf{iter}}\ c$$

$$s = \mathrm{agg}_{\substack{\mathsf{score:}\\ \mathsf{avg\ score}\\ \mathsf{/iter}}}\ c$$

The calculation of a delivers the content scores for those iterations where the content is empty, *i.e.*, whenever an iteration is mentioned in the loop relation, but not in the table c of content scores. s contains the aggregate scores of non-empty contents.

The constructor function for content is also translated using $[\![\cdot]\!]^*_{\Gamma,L}$:

$$[\![\mathsf{twigContent}\ c]\!]^*_{\Gamma,L} = \left(\mathrm{CONT}_{\substack{\mathsf{iter}\\ \mathsf{pos}\\ \mathsf{item}}}\ c'\ ,\ \mathsf{smContent}\ c'\right)$$

where

$$c' = [\![c]\!]_{\Gamma,L}$$

$$\mathsf{smContent}\ s = \mathrm{agg}_{\substack{\mathsf{score:}\\ \mathsf{avg\ score}\\ \mathsf{/iter}}}\ s$$

This calculates the average of all content scores. Empty contents yield missing iterations in the returned relation, which is accounted for when calculating a in smElem.

The truth, however, is a bit more complicated than what is described here: The XPath data model, and hence XQuery Full Text, does not allow empty text nodes, nor consecutive text nodes amongst a node's children[4], which requires text nodes following each other to be joined. At the time of writing, and in contrast to the Pathfinder compiler that correctly implements this, Pathfinder$^{\mathrm{FT}}$ still offers only an approximation of the desired behaviour.

5.20 Using Documents

When using a document from the database, it may make sense to propagate the score of the document name to the document root, as in the next query.

```
for $c score $s
in for $url
   in doc("list.xml")/entry[./abstract contains text "Burkina"]/url
```

[4] `http://www.w3.org/TR/xpath-datamodel/#Node`

```
    return doc($url)//chapter[. contains text "Ouagadougou"]
order by $s descending
return $c/title
```

Multiple documents are used here, and the relevance of a result item depends on more than one of them: One document, `"list.xml"`, contains a list of abstracts and URLs. The abstracts are searched for the term `"Burkina"`, which creates a score that is propagated to the URL node of the entry.

The identified documents are then searched for chapters containing `"Ouagadougou"`, which creates another score. To implicitly combine both scores, the function `doc()` must offer score propagation from its argument to the returned document.

This is achieved by simply passing the input score around the Relational Algebra DOC operator. Since the PATHFINDER compiler handles document access as a constructor, the compilation scheme $[\![\cdot]\!]^*_{\Gamma,L}$ is used again.

$$[\![\texttt{doc(}\ e\ \texttt{)}]\!]^*_{\Gamma,L} = \left(\mathrm{DOC}_{\mathsf{item:item1}}\ (\pi_{\substack{\mathsf{iter}\\ \mathsf{item1:item}}}\ e')\ ,\ \pi_{\substack{\mathsf{iter}\\ \mathsf{score}}}\ e'\right)$$

where

$e' = [\![e]\!]_{\Gamma,L}$

5.21 Calling the Full Text machine

The `contains text` operator is the connective link between the database query language XQUERY and the information retrieval language defined in the XQUERY FULL TEXT draft.

Fed with a search context expressed by an XQUERY expression, and a search specification expressed by a Full Text expression, it returns a scored Boolean item in the XQUERY FULL TEXT domain, *i.e.*, in the XQUERY domain enriched with scored items. XQUERY FULL TEXT's Ignore Option[5] is currently unsupported. PATHFINDER$^{\mathrm{FT}}$ would consider this part of the search specification.

One prevalent design goal of PATHFINDER$^{\mathrm{FT}}$ is to be independent of the scoring model and the Full Text machinery used to calculate scores. The required adaptations to fit in a Full Text engine culminate in the compilation strategy for the `contains text` operator, and different design choices can be made here, depending on the abilities of the Full Text engine, its preferred intermediate language, and the available operators in the Relational Algebra:

- The `contains text` operator, and the Full Text expression specifying the search may be compiled to specially crafted Relational Algebra primitives (Section 5.21.1).
- The `contains text` operator can be compiled to a Relational Algebra function call (Section 5.21.2).
- The Full Text expression may be compiled to an XML- (Section 5.22.1) or string- (Section 5.22.2) representation, which can be handled with existing PATHFINDER means.

[5] `http://www.w3.org/TR/xquery-full-text/#ftignoreoption`

- The Full Text expression tree can be decomposed into multiple Full Text expressions with their own calls to `contains text`, by pushing Full Text operators towards the root of the query expression tree, past the initial `contains text` operator (Section 5.22.4).

5.21.1 Purely algebraic

The purely algebraic approach would compile the `contains text` operator and the Full Text operators to special relational algebra primitives. Given that the PATHFINDER compiler could be extended with the required knowledge, it would be able to perform cross-language optimisations.

This, however comes at a high cost: For each Full Text operator a Relational Algebra counterpart would have to be implemented. Additionally, the PATHFINDER compiler's carefully crafted optimisations would have to be extended to deal with the new operators. Equivalence laws on the extended algebra would have to be described, proven, and implemented in the PATHFINDER compiler's optimiser. Finally, the PATHFINDER compiler, would be burdened with mapping the new operators to adequate physical operators in the RDBMS, *i.e.*, the level where score propagation is implemented would be one step closer to the database core, which contradicts the approach taken by this work.

Hence, the result would not be a minimally invasive extension of the PATHFINDER compiler, but a complete rewrite to make it a Full Text engine instead.

5.21.2 Relational Algebra function call

The current implementation of PATHFINDER$^{\text{FT}}$ maps the `contains text` operator to a function call at the Relational Algebra level.

Function calls are provided by the Relational Algebra operator FUN, which is parametrised with the name of the function to call, and consumes a loop relation and a list of arguments. The latter have to be provided in an iter,pos,item encoding, *i.e.*, scores cannot be passed to a Relational Algebra function. The returned relation, however, may carry additional columns:

$$\text{FUN}_{\text{name}} :: \mathsf{Rel}_{\mathsf{iter}} \rightarrow [\mathsf{Rel}_{\substack{\mathsf{iter}\\\mathsf{pos}\\\mathsf{item}}}] \rightarrow \mathsf{Rel}_{\substack{\mathsf{iter}\\\mathsf{pos}\\\mathsf{item}\\\dots}}$$

The loop relation is required to determine the empty sequences in the arguments, see Section 4.2.3.

The PATHFINDER$^{\text{FT}}$ compiler makes use of the Relational Algebra function `pftijah`, which is the interface provided by PF/TIJAH. This function takes two arguments, the former being an encoded XQUERY item sequence that represents the search context, the latter being an XQUERY string.

The semantics of the second argument varies: If the first character of the string is a percent character (`%`), the remainder is considered a NEXI query that is interpreted completely inside the PF/TIJAH system. Otherwise, the string is interpreted as a space-separated list of search terms.

To be more flexible when playing with the concrete implementation, the `contains text` operator is mapped to an XQUERY Core function call in a first step, instead of extending the XQUERY Core language with a new keyword.

Thus, at XQuery Core level, the expression will contain one or more occurrences of the function[6] call `pftijah(`c`,` t`)`, which is then compiled to Relational Algebra in another step. In the simplest case, on would translate the XQuery query

e_1 `contains text` e_2

into the XQuery Core expression

`pftijah(`e_1'`,` e_2'`)`

with suitable translations e_i'. This works fine if the second argument e_2 is a single literal search term. For more complex cases, however, a more sophisticated translation scheme is needed. Some strategies are described in Section 5.22.

Score Propagation Since the search context may carry non-default scores, it might make sense to adjust the scores returned from a `contains text` depending on the scores of its arguments. Consider

```
doc("books.xml")
    [./title contains text "something"]
    [./author contains text "John"]
```

where two consecutive filters are applied on a collection of books. The user may expect the scores of the result to reflect the outcome of both Full Text operations. One might even require equivalence with

```
doc("books.xml")
    [ ./title contains text "something"
      and
      ./author contains text "John"
    ]
```

to hold. But this is, again, subject to the design of the score propagation functions.

A user might argue that the score of the search terms must be considered as well, a simple consequence of XQuery Full Text's orthogonality:

```
doc("books.xml")
    [ ./title
      contains text
      { $termcollection[./group contains text "nonsense"]/terms }
    ]
```

In the above query, the search terms are determined by a Full Text search on a collection of search terms in `$termcollection`.

[6] No namespace prefix is used for the function `pftijah`: The function name is introduced on XQuery to XQuery Core translation, and removed on XQuery Core to Relational Algebra translation. Hence, outside of the PathfinderFT implementation the function's name is invisible, and, thus, in fact arbitrary. It can be considered outside of any namespace.

Compilation Score propagation for the `contains text` operator is accounted for when compiling the XQuery Core function `pftijah` to Relational Algebra:

$$[\![\texttt{pftijah}(e_1,e_2)]\!]_{\Gamma,L} = \mathsf{smContainsText}\ L\ e_1'\ e_2' \triangleleft \mathrm{FUN}_{\texttt{pftijah}}\ L\ [\pi_{\substack{\mathsf{iter}\\\mathsf{pos}\\\mathsf{item}}}\ e_1', \pi_{\substack{\mathsf{iter}\\\mathsf{pos}\\\mathsf{item}}}\ e_2']$$

where

$e_i' = [\![e_i]\!]_{\Gamma,L}$

$$\mathsf{smContainsText}\ L\ c\ t\ r = \pi_{\substack{\mathsf{iter}\\\mathsf{pos}\\\mathsf{item}\\\mathsf{score:score2}}} \triangleleft \mathrm{op}_{\substack{\mathsf{score2:}\\\mathsf{score}\cdot\mathsf{score1}}} \triangleleft (s \uplus a) \bowtie_{\mathsf{iter1}=\mathsf{iter}} \pi_{\substack{\mathsf{iter1:iter}\\\mathsf{pos}\\\mathsf{item}\\\mathsf{score1:score}}}\ r$$

$a = \mathsf{smDefault} \triangleleft L \setminus \pi_{\mathsf{iter}}\ c$

$$s = \mathrm{agg}_{\substack{\mathsf{score:}\\\mathsf{avg\ score}\\/\mathsf{iter}}}\ c$$

This implementation of smContainsText ignores scores from the search terms t, and multiplies the scores returned from the Full Text engine with the average score s of the search context items in c. For the empty search context, a provides the default score.

* * *

Up to this point, the focus was on how to *call* the Full Text machinery, and how to propagate scores across this call. The remainder of this section is about what is *passed to* the Full Text machinery.

5.22 Compiling Full Text expressions

If the purely algebraic approach was taken, see Section 5.21.1, there would not be much to worry about here: The Full Text expression has already been translated to some Relational Algebra expression that could readily be optimised by the Pathfinder compiler, and executed by MonetDB.

As described in Section 5.21.2 however, a function call is used whose second parameter represents the Full Text expression. To this end, it is required to compile the Full Text expression into something that is in the XQuery Core domain, and is understood by the Full Text machine interface, namely the `pftijah` function.

This task can be approached from two different angles:

1. The direct approach, presented in the following sections, is to translate the Full Text language introduced by XQuery Full Text to the Full Text language understood by the Full Text engine. For this to work precisely, it is required that the former represents a semantic subset of the latter. In the real world, one should be happy if not too much frictional loss occurs: A sloppy handling may be acceptable, since the Full Text semantics are somewhat fuzzy anyway. Another obstacle is the orthogonal nature of XQuery Full Text, which may require a loop-back opportunity in the target language to hook in expressions from the XQuery domain, see Section 5.22.3.

2. The *unfolding* approach, Section 5.22.4, tries to unfold Full Text expressions occurring on the right hand side of the `contains text` operator to valid XQuery Core expressions while retaining semantics. Defining semantic equivalence, however, is difficult for languages that

lack a definition of semantics. A more serious problem is that some Full Text expressions simply cannot be represented in XQuery Core, see Section 5.22.6.

3. A combination of both approaches, *i.e.*, unfolding exactly those XQuery Full Text constructs that cannot be expressed in the target Full Text language, gains the advantages of both, but also their difficulties: Again, orthogonality allows an unfoldable construct to occur below a directly usable construct, and unfolding past the upper expression might prove difficult, see Section 5.22.5.

5.22.1 The direct approach via XML

Without adding extensions crafted towards XQuery Full Text, the XQuery Core domain offers only two structures that seem suitable for representing an expression in the Full Text target language: A plain string, or an XML fragment.

A common drawback of both of them is that the actual representation of the Full Text expression is calculated at runtime. As a consequence, static errors in the Full Text target language may appear as runtime errors of the generating XQuery Full Text query. Also, the representation is opaque to the compiler, which deprives the compiler of optimisation opportunities: The Pathfinder compiler will handle the Full Text representation as ordinary data. Furthermore, performance issues arise, which are discussed later on in Section 5.22.3.

Another drawback is that the Full Text representation has to be parsed by the Full Text engine each time the interface function is called. *I.e.*, the Full Text expression parsed by Pathfinder^{FT} is compiled into something that is later parsed again, potentially repeatedly, by the Full Text engine.

Using XML fragments is an approximation of adding special operators (see Section 5.21.1), and can be carried through to the Relational Algebra plan: Instead of adding operators to the Relational Algebra vocabulary that represent Full Text operators, a twig is constructed that represents the Full Text expression. The element names are drawn from a distinct namespace.

An advantage of this approach is that the structure of the Full Text expression becomes visible in the Relational Algebra plan, where it appears as a series of twigs. Thus, a later compilation phase may be able to apply rewrites to the generated structure, if it is able to recognise those twigs that represent Full Text expressions. This can be done via a dedicated namespace.

Another advantage is that it is rather easy to embed XQuery expressions in the Full Text expression, and to express this in the Relational Algebra plan: There XQuery expression simply appears as the content construction of a suitable element node.

```
declare namespace wns = "http://www.mediawiki.org/xml/export-0.4/";
declare namespace pfft = "http://stefan-klinger.de/ns/pfft/0.1";

(# pfft:ftComp toXml #) {

  for $i score $s
  in doc("enwikiquote.xml")//wns:page
     [ .//wns:text contains text
       "technology" ftand "magic" ftand "indistinguishable" ]
  order by $s descending
  return $i/wns:title

}
```

The plan generated from this query is shown in Figure 3 on page 82. The shown pragma triggers the compilation of Full Text expressions into twigs. The twig generated from the Full Text expression `"technology" ftand "magic" ftand "indistinguishable"` appears as a group of green nodes in the plan, zoomed in on at the right hand side of the figure.

The bright green nodes are the node constructors, the topmost of which represents the `ftand` operation. This can be seen from its left descendants, which construct the element name `pfft:and`.

The text node constructors at the very bottom refer to the three search terms `"technology"`, `"magic"`, and `"indistinguishable"`. Since these are XQuery string literals (or the result of some other XQuery calculation), they are simply hooked into the Full Text expression via `pfft:embed` element nodes, which make up the middle row of constructors.

Of course, the XQuery Core function used to interface the Full Text engine is required to understand the passed XML fragments, and `pftijah`, unfortunately, lacks this ability.

Hence, although Pathfinder^{FT} can generate such plans, they are of no use, since there is currently no system available that could evaluate them, nor benefit from the described advantages.

5.22.2 The direct approach via NEXI

The `pftijah` function accepts the search specified in NEXI, passed as a string argument prefixed with a percent character (`%`). To this end, the Full Text expression given in the XQuery Full Text query needs to be compiled into a NEXI expression.

This is an easy task, if the Full Text expression is constant, *i.e.*, if it does not require any calculations in the XQuery domain to be performed at run time. In that case, the NEXI query can be calculated completely at compile time, *i.e.*, by Pathfinder^{FT} when generating the Relational Algebra plan.

```
declare namespace wns = "http://www.mediawiki.org/xml/export-0.4/";
declare namespace pfft = "http://stefan-klinger.de/ns/pfft/0.1";

(# pfft:ftComp toNexi #) {

for $i score $s
in doc("enwikiquote.xml")//wns:page
   [ .//wns:text
     contains text
     "technology" ftand "magic" ftand "indistinguishable"
   ]
order by $s descending
return $i/wns:title

}
```

The same query as in Section 5.22.1, however compiling the Full Text expression to a NEXI string as directed by the pragma, yields the plan shown in Figure 4 on page 84. The bigger grey box contains all the text snippets that need to be concatenated to form the final NEXI query. This concatenation is performed by the `fn:string_join` function, which is part of the Pathfinder compiler's Relational Algebra.

Note that the compiled NEXI query

```
.[(about(.,technology) and about(.,magic)
                       and about(.,indistinguishable))]
```

is constant, *i.e.*, no computation (set aside string concatenation) is required to determine its contents.

The PF/Tijah system is able to work in a loop-lifted fashion in this situation, which leads to internally running one NEXI query on the loop-lifted representation of all node sets returned from the axis steps at once. Executing this plan, and returning the result

```
<XQueryResult>
  <title>Technology</title>
  <title>Paranormal</title>
  <title>Arthur C. Clarke</title>
  <title>December 16</title>
  <title>Programming</title>
  <title>Creationism and evolution</title>
  <title>Creationism and Intelligent Design</title>
</XQueryResult>
```

from the 164MB document takes about 1.7s.

5.22.3 Variable search terms

This strategy, however, fails badly when the search term is not constant.

```
declare namespace wns = "http://www.mediawiki.org/xml/export-0.4/";
declare namespace pfft = "http://stefan-klinger.de/ns/pfft/0.1";

(# pfft:ftComp toNexi #) {

for $t
in ("technology", "magic", "indistinguishable")
return for $i score $s
       in doc("enwikiquote.xml")
             //wns:page[.//wns:text contains text {$t}]
       order by $s descending
       return $i/wns:title

}
```

In this query, the search terms are drawn from a list that is iterated over. The compiled Relational Algebra plan is shown in Figure 5 on page 86, the region zoomed in on depicts the construction of the NEXI query. The larger grey box on the lower right hand side contains the loop-lifted search terms.

The problem here is that PF/Tijah cannot optimise in this situation, and runs the NEXI queries for each of the nodes in the search context repeatedly. The document contains 33316 pages, hence as many NEXI queries are executed by the PF/Tijah back-end, which takes about 2h26′ on the same document as above.

5.22.4 Unfolding Full Text expressions

A completely different approach is to "unfold" the Full Text expression into multiple XQuery Core expressions. In an extreme case this leads to plans where the Full Text engine is fed solely with a search context and a single keyword. In other words, by mapping the operators of the Full Text language to XQuery Core, parts of the score calculation are moved from the Full Text engine to the Relational Algebra engine.

An advantage of this strategy is that the structure of the complete expression becomes visible to the Pathfinder compiler, making it subject to potential optimisations.

Pathfinder^{FT} currently supports unfolding of Boolean operators and weights, the above example query can thus be compiled setting the `pfft:ftComp` to `unfold`:

```
declare namespace wns = "http://www.mediawiki.org/xml/export-0.4/";
declare namespace pfft = "http://stefan-klinger.de/ns/pfft/0.1";

(# pfft:ftComp unfold #) {

for $i score $s
in doc("enwikiquote.xml")//wns:page
   [ .//wns:text
     contains text
     "technology" ftand "magic" ftand "indistinguishable"
   ]
order by $s descending
return $i/wns:title

}
```

The plan generated by this query is shown in Figure 6 on page 88. Three areas are highlighted this time: Interfacing the Full Text engine (bottom), Boolean computation (middle) and score computation.

The most significant change is that the PF/Tijah engine is called three times, once for each keyword, but also the score and value calculation are visible in this plan. The `pftijah` function calls appear in the wider box at the bottom of the plan: The search context enters the box from the left, and and is used as first argument of each function calls. The respective keyword appears as second argument

This plan already underwent Pathfinder's optimisations, and one of them can be observed here nicely: The Boolean conjunctions have turned into three select statements, which appear in the middle of the plan. And the score computation, which was interleaved with the Boolean computation in the non-optimised plan, has been separated, and pushed towards the top, working only on those items that passed the Boolean conjunction. For comparison, see the unoptimised plan in Figure 7 on page 90.

5.22.4.1 Equivalence

Unfolding, as discussed here, means nothing but replacing a computation in the Full Text domain with a computation performed in the Relational Algebra engine, no matter whether this unfolding happens automatically (as described in these sections) or by hand. The obvious question is whether the user may expect the same result for both variants of a query.

Consider the following two examples:

```
$canteen[./food contains text "cheap"
                        ftand "tasty"
        ]

$canteen[ ./food contains text "cheap"
      and ./food contains text "tasty"
        ]
```

While I would clearly prefer to consider them equivalent, one might also argue the converse: *"I want the food to be cheap, and tasty."* has a different meaning than *"I want the food to be cheap, and I want the food to be tasty."*, where repetition is used to emphasise the conjunction. Similarly, the `and` might implement a somewhat more strict version of conjunction than the concept used by the Full Text engine. In other words, to implement such behaviour, a *different implementation* of score propagation needs to be employed for the `and` operator, than what the Full Text engine uses for `ftand`.

This can be done, but the user needs to be aware of the fact that unfolding a query changes its semantics. Hence, there is no guarantee that unfolding leaves the semantics untouched. If this is desired, though, the score propagation functions (see Section 5.23) have to be tailored to match the Full Text engine used.

5.22.4.2 Unfolding Booleans

It is a little bit incorrect to speak of the Full Text operators `ftand`, `ftor`, and `ftnot` as "Boolean operators": Although the intended semantics is of course "Boolean" in nature, they *do not* operate on Booleans (nor scored Booleans). A more precise perception is to see them as search term combiners, *i.e.*, as function combiners for binary functions, the input of which is the search term (a string) *and* the search context (an item sequence).

To this end, the unfolding scheme $[\![\cdot]\!]^{\mathsf{uf}}_{c}$ requires to pass the search context c down to the individual search terms t_i. This, however, would introduce multiple copies of the query term c. To avoid the side-effect of node construction to appear several times, a `let`-clause is used, binding a fresh variable to the search context. This variable is passed on instead of the search context expression.

Let q be Full Text expression containing only "Boolean" operators (arbitrarily nested) and search terms t_i. Then the XQuery Full Text expression

c `contains text` q

is translated to

`let $c :=` c' `return` $[\![q]\!]^{\mathsf{uf}}_{\$c}$

with a fresh variable `$c` and an XQuery Core representation c' of c. The unfolding scheme $[\![q]\!]^{\mathsf{uf}}_{\$c}$, (*i.e.*, unfold q with context variable `$c`) is defined by the following set of equations:

$$\begin{aligned}
[\![e_1 \texttt{ ftand } e_2]\!]^{\mathsf{uf}}_{\$c} &= [\![e_1]\!]^{\mathsf{uf}}_{\$c} \texttt{ and } [\![e_2]\!]^{\mathsf{uf}}_{\$c} \\
[\![e_1 \texttt{ ftor } e_2]\!]^{\mathsf{uf}}_{\$c} &= [\![e_1]\!]^{\mathsf{uf}}_{\$c} \texttt{ or } [\![e_2]\!]^{\mathsf{uf}}_{\$c} \\
[\![\texttt{ftnot } e]\!]^{\mathsf{uf}}_{\$c} &= \texttt{not } [\![e]\!]^{\mathsf{uf}}_{\$c} \\
[\![t_i]\!]^{\mathsf{uf}}_{\$c} &= \texttt{pftijah(\$c, } t_i\texttt{)} \qquad \text{, if } t_i \text{ is a search term}
\end{aligned}$$

The result of such unfolding can be observed in the unoptimised plan given in Figure 7 on page 90, where the relations returned from the three calls to the Full Text engine are combined using the appropriate operations to calculate the final value and score.

* * *

Although the Relational Algebra plan (Figure 6 on page 88) of the unfold query invokes the Full Text engine multiple times, this approach seems to benefit from PATHFINDER's optimisations: A speedup of 25% can be observed[7] when comparing the unfold version with the version that uses NEXI as intermediate language (Figure 4 on page 84). On the other hand, the latter may as well suffer from the overhead due to generating and parsing the NEXI query string.

5.22.4.3 Unfolding weights

Another operation for which unfolding is implemented in PATHFINDER$^{\text{FT}}$ is the weighting of Full Text expressions. According to the XQuery Full Text draft, "the effect of weights on the resulting score is implementation-dependent"[8]. The constraints actually imposed by the draft are chosen in such a ridiculously whimsy manner, that it is totally unclear what the intention of the designers was. *E.g.*, although its semantics is unclear, a weight must be in the range $[-1000.0..1000.0]$, otherwise an error has to be raised[9].

The unfolding of weights is achieved by extending the unfolding scheme $[\![\cdot]\!]^{\mathsf{uf}}_{\$c}$, introduced in Section 5.22.4.2, with one more rule:

$$[\![e \texttt{ weight } \{w\}]\!]^{\mathsf{uf}}_{\$c} = \mathsf{smWeight}\ w\ [\![e]\!]^{\mathsf{uf}}_{\$c}$$

This introduces the function smWeight, which needs to be provided by the scoring model.

An implementation that simply scales the score by the weight is implemented by

$$\mathsf{smWeight}\ w\ e' = \texttt{for \$i score \$s in } e' \texttt{ return \$i scored (\$s * } w\texttt{)}$$

making use of the non-standard operator `scored`, introduced in Section 5.9, allowing a simple map on the score-part of the items in e'. Of course, `$i`, and `$s` are fresh variables again.

[7] Test was: Seeding MONETDB/PATHFINDER's cache with a run of the query; then timing 100 consecutive runs of the same query. This does not claim to be a full-fledged performance test.

[8] `http://www.w3.org/TR/xquery-full-text/#section-using-weights`

[9] `http://www.w3.org/TR/xquery-full-text/#ftweight`

5.22.4.4 Weight borrow

At this point I would like to point out an issue hinted at previously: By unfolding, a Full Text expression is mapped to an XQuery Core expression, which is very likely to have slightly different semantics (Section 2.1). One particularly nasty pitfall is the potential interference of weighting and Boolean operators.

When using a score propagation for Booleans (Section 5.10) that multiplies the scores of the arguments for an operator, say `and`, together with unfolding and smWeight as described in the previous section, then the weight carries over to the neighbouring terms.

Consider the following XQuery Full Text expression, where a weight is applied on the term `"technology"`:

```
.//wns:text contains text "technology" weight {0.3} ftand "magic"
```

By unfolding, this is first transformed as follows, where `$c = .//wns:text`:

$$
\begin{aligned}
&= [\![\texttt{"technology" weight \{0.3\} ftand "magic"}]\!]^{\mathsf{uf}}_{\$c} \\
&= [\![\texttt{"technology" weight \{0.3\}}]\!]^{\mathsf{uf}}_{\$c} \texttt{ and } [\![\texttt{"magic"}]\!]^{\mathsf{uf}}_{\$c} \\
&= (\mathsf{smWeight}\ 0.3\ [\![\texttt{"technology"}]\!]^{\mathsf{uf}}_{\$c}) \texttt{ and } [\![\texttt{"magic"}]\!]^{\mathsf{uf}}_{\$c} \\
&= (\mathsf{smWeight}\ 0.3\ \texttt{pftijah(\$c,"technology")}) \texttt{ and pftijah(\$c,"magic")}
\end{aligned}
$$

Now, assuming that `pftijah($c,"technology")` and `pftijah($c,"magic")` return the items $\langle v_{\mathsf{tech}}|s_{\mathsf{tech}}\rangle$ and $\langle v_{\mathsf{magic}}|s_{\mathsf{magic}}\rangle$ respectively, the overall result is computed as follows:

$$
\begin{aligned}
&= (\mathsf{smWeight}\ 0.3\ \langle v_{\mathsf{tech}}|s_{\mathsf{tech}}\rangle) \texttt{ and } \langle v_{\mathsf{magic}}|s_{\mathsf{magic}}\rangle \\
&= \langle v_{\mathsf{tech}}|s_{\mathsf{tech}} \cdot 0.3\rangle \texttt{ and } \langle v_{\mathsf{magic}}|s_{\mathsf{magic}}\rangle \\
&= \langle v_{\mathsf{tech}}{}^{\wedge} v_{\mathsf{magic}}|s_{\mathsf{tech}} \cdot 0.3 \cdot s_{\mathsf{magic}}\rangle
\end{aligned}
$$

Following the very same argumentation for the similar Full Text expression

```
.//wns:text contains text "technology" ftand "magic" weight {0.3}
```

which only differs from the one above by weighting the other term, leads to the computation

$$\langle v_{\mathsf{tech}}{}^{\wedge} v_{\mathsf{magic}}|s_{\mathsf{tech}} \cdot s_{\mathsf{magic}} \cdot 0.3\rangle$$

which is, of course, the same.

Note that the weight *carries over* to the other search term, due to a change in associativity of the operators in different domains: XQuery Full Text's associative `weight` operator on one side, and multiplication in the domain of XQuery arithmetics on the other.

This can be fixed by using a non-associative operator for Boolean score propagation, as provided when setting the pragma `pfft:ftBool` to `extreme` (Section 5.10). Then the respective results of the given expressions are

$$\langle v_{\mathsf{tech}}{}^{\wedge} v_{\mathsf{magic}}|\mathsf{min}\ (s_{\mathsf{tech}} \cdot 0.3)\ s_{\mathsf{magic}}\rangle \qquad \textit{vs.} \qquad \langle v_{\mathsf{tech}}{}^{\wedge} v_{\mathsf{magic}}|\mathsf{min}\ s_{\mathsf{tech}}\ (s_{\mathsf{magic}} \cdot 0.3)\rangle$$

depending on which term is weighted.

5.22.5 Variable search terms, again

Unfortunately, unfolding does not solve the performance problem observed with variable search terms in Section 5.22.3.

```
declare namespace wns = "http://www.mediawiki.org/xml/export-0.4/";
declare namespace pfft = "http://stefan-klinger.de/ns/pfft/0.1";

(# pfft:ftComp unfold #) {

for $t
in ("technology", "dupery", "legerdemain")
return for $i score $s
       in doc("enwikiquote.xml")//wns:page
          [ .//wns:text
            contains text
            {$t} ftand "magic" ftand "indistinguishable" ]
       order by $s descending
       return $i/wns:title

}
```

Although the plan (Figure 8 on page 92) generated for this query looks cleaner than the one shown in Figure 5 on page 86, the runtime is about the same. This indicates that string concatenation is not the runtime killer, as previously guessed.

5.22.6 The limits of unfolding

Just as some Full Text expressions may not be expressible in language understood by the Full Text engine, there are also Full Text expressions that cannot be handled by unfolding. This is the case for all situations, where the information required to perform a Full Text operation is not returned by the `pftijah` function to the XQuery Core domain. These are, *e.g.*, distance and context information:

XQuery Full Text offers the `not in` operator, providing a *mild-not* selection[10]. The ubiquitous use case for this is

```
.../city[./name contains text "York" not in "New York"]
```

expressing a search for cities whose name contains an occurrence of `"York"` that is not directly preceded by `"New"`. An unfold query had to validate this constraint for each result matching `"York"`. Since —in the described architecture— no context information of the retrieved matches is provided, PathfinderFT is unable to do this.

The same problem turns up when distances are involved. Typical queries are

```
.//text contains text "technology" ftand "magic"
                      distance at most 2 words
```

using distance based on word counts, or

[10] `http://www.w3.org/TR/xquery-full-text/#sec-ftmildnot`

```
.//text contains text "technology" ftand "magic" same sentence
```

using a notion of distance based on scopes.

An approach to solve this issue is to extend the iter|pos|item|score schema with further attributes to describe, *e.g.*, token positions, and many Full Text engines ([7, 15, 8]) readily provide token position information. To extend PATHFINDER's item sequence encoding with positional information, a set-valued (or, at least, list-valued) approach is required: Since a token may occur at different positions below an XML node, a match returned from the Full Text engine may carry any number of positions.

Clearly, since we want to stay in the world of atomic-valued attributes, a list-valued attribute (*i.e.*, tok :: [Int]) is not an option[11]. Also, it is desirable to leave the schema fixed, *i.e.*, not to add attributes tok1, tok2, ..., tok$n$ for n token positions (this has been done in [5]).

Following the PATHFINDER design, the way to go is to multiply each item with its positions. This would also allow to annotate each position with a different score.

Consider the next example document, with node ids annotated as superscript, and a running token enumeration shown as subscript. Like γ for node surrogates, we shall use τ with a subscript to identify tokens.

```
<doc>0
  <a>1 magician0 magic1 technician2 theurgist3 2</a>
  <a>3 lawyer4 technologist5 magician6 broker7 Rincewind8 4</a>
</doc>
```

Running the query

```
doc(...)//a[. contains text "magician"]
```

will invoke the Full Text engine with two search contexts (γ_1, γ_3) inside a loop that represents the predicate expression (Section 5.16). Let's look inside this loop: The search context is represented (assuming $\mu = 1.0$) as

lifted context =	iter	pos	item	score
	1	1	γ_1	1.0
	2	1	γ_3	1.0

Using a thesaurus, this is what the Full Text engine may calculate

scoring result =	iter	pos	item	score	tok
	1	1	true	1	τ_0
	1	1	true	0.5	τ_3
	2	1	true	1	τ_6

indicating, that there are full matches (*i.e.*, score = 1) at τ_0, τ_6, and a partial match at τ_3 (score = 0.5, because it required a dictionary lookup).

Several difficulties arise here. First of all, I was cheating with the orthogonality: The input relation did not have a tok attribute, but the output relation has. To regain orthogonality, it is required to add the tok attribute to all relations representing item sequences. A null value

[11] For demonstration I talk about lists here, and in the following, assuming ordering by the order of tokens. Talking about sets is probably more correct.

could be used here. This issue can be mitigated by arguing that token positions are not relevant all the time, but only below the distance constraints created by certain unfolding operations. However, it will prove necessary to extend the core language with certain new *primitives* below, which in turn would require token positions on some of their arguments. Having two kinds of item sequence encodings aggravates composition of expressions, and requires to reason about schemas, *i.e.*, something like a type system, distinguishing the two kinds, would be required. On the other hand, token positions is a valuable information, and passing them to the root of the query expression allows to direct the user to relevant results more precisely (think about the text snippets presented by a search engine, with the matches highlighted).

A minor effect of orthogonality is that even string literals occurring in the query need to be enumerated, because the `contains text` operator may very well be applied to a literal (see Section 2.2.2.3), or a constructed fragment. Just for the argument, assume the token enumeration to span the query as well.

Also, the scores still need aggregation: The result, after score propagation to the nodes, aggregation and back-mapping (see Section 4.2.3) could be

$$\text{query result} = \begin{array}{ccccc} \text{iter} & \text{pos} & \text{item} & \text{score} & \text{tok} \\ 1 & 1 & \gamma_1 & 0.75 & \tau_0 \\ 1 & 1 & \gamma_1 & 0.75 & \tau_3 \\ 1 & 2 & \gamma_3 & 1 & \tau_6 \end{array}$$

by simply calculating the averages of the scores, grouped by iteration and item, and then listing all tokens from the same group with the same score, now matter how much it contributed.

Not only does this loose information (which token is relevant?), looking at the document again, this seems unlikely to be the desired result, because it does not relate the number of matching tokens to the number of failing ones.

One might consider the following to be more precise:

$$\text{query result}' = \begin{array}{ccccc} \text{iter} & \text{pos} & \text{item} & \text{score} & \text{tok} \\ 1 & 1 & \gamma_1 & 0.25 & \tau_0 \\ 1 & 1 & \gamma_1 & 0.125 & \tau_3 \\ 1 & 2 & \gamma_3 & 0.2 & \tau_6 \end{array}$$

Which, of course, could have been calculated by the Full Text engine directly. But with the unfolding perspective, we want to move calculation from the Full Text engine to the Relational Algebra engine. Alternatively, the Full Text engine might thus have included the failing tokens as well:

$$\text{scoring result}' = \begin{array}{ccccc} \text{iter} & \text{pos} & \text{item} & \text{score} & \text{tok} \\ 1 & 1 & \text{true} & 1 & \tau_0 \\ 1 & 1 & \text{false} & 0 & \tau_1 \\ 1 & 1 & \text{false} & 0 & \tau_2 \\ 1 & 1 & \text{true} & 0.5 & \tau_3 \\ 2 & 1 & \text{false} & 0 & \tau_4 \\ 2 & 1 & \text{false} & 0 & \tau_5 \\ 2 & 1 & \text{true} & 1 & \tau_6 \\ 2 & 1 & \text{false} & 0 & \tau_7 \\ 2 & 1 & \text{false} & 0.1 & \tau_8 \end{array}$$

This contains a score for *every* single token, and it requires aggregation not only on the scores, but also (and more important) on the items, which is further discussed in Section 7.2.

Assuming that this particular Full Text engine trusts Rincewind to do only insignificant magic (*i.e.*, item = false, although score > 0 for τ_8 in the very last tuple) makes the required aggregation difficult: Just averaging the scores yields 0.22 instead of 0.2 for γ_3 in query result$'$ above. But prior selection on the item attribute will just generate the query result we have already deemed phony.

One could simply divide the score by the number of tokens (count grouped by (iter, pos)), and then drop the false items. This returns the result as expected above, but comes with a loss of information. The question remains what role the failed tokens should play whan calculating the overall score and token list.

5.22.6.1 The need for null

A potential solution is to add a notion of scores that are not related to any (successful) token, *i.e.*, the aggregate of false items could be stored with a null token:

query result$''$ =

iter	pos	item	score	tok
1	1	γ_1	0.25	τ_0
1	1	γ_1	0.125	τ_3
1	2	γ_3	0.2	τ_6
1	2	γ_3	0.02	null

This, by the way, represents the XQuery item sequence

```
( <a>magician magic technician theurgist </a>
, <a>lawyer technologist magician broker Rincewind</a>
)
```

with some attached information: Both items know which token caused them to gain which score, the first comes with $[(0.25, \tau_0), (0.125, \tau_3)]$, the second one with $[(0.2, \tau_6), (0.02, \text{null})]$.

This was the second opportunity where the need for a null value in the database back-end arose, both triggered only by adding token positions to the sequence encoding (elsewhere in this thesis, null values are not required).

The third occasion is by negation: Negation is successful when its argument fails. So if there are no matching tokens, a high score is expected. But unlike the representation of an empty item sequence (which simply omits tuples with an iter value mentioned in the loop relation, see Section 4.2.3) it is required to pass some information, namely the score. As above, a simple approach is to use one entry with the null token to account for scores that cannot be traced back to a token. Of course this mixes the tokens matched by negation with those that failed, as τ_8 above. An approach to solve this is sketched in Section 7.3.

5.22.6.2 Making use of token positions

With all this work, we have introduced new problems: Recall that the need for passing token information arose from unfolding an XQuery Full Text expression into an XQuery Core expression hosting multiple Full Text expressions. Unfortunately, neither XQuery, nor XQuery Core provide means to deal with token positions. For scores, there is a `score` keyword, but there is nothing similar for token positions. In other words, the token positions are not accessible outside Full Text expressions. It is thus required to add special functions or keywords to the

language, which receive special treatment by the compiler: It must be aware of the fact that some language forms request calculations on the token positions.

Let's have a brief look at what these new primitives could be. Examine the following XQuery Full Text predicate:

```
$x[. contains text "foo" ftand "bar" distance at most 1 word]
```

The naive unfolding approach is to deal with the Boolean first, and check the distances later, by using a function `distMax` which takes a maximum distance —constructed by a "unit" function— as first argument:

```
distMax(words(1),$x contains text "foo" and $x contains text "bar")
```

This, however, doesnot work, which becomes clear when looking at the results of the two Full Text expressions, which might be something like the following:

`$x contains text "foo"` =	iter	pos	item	score	tok
	1	1	true	0.1	τ_1
	1	1	true	0.2	τ_2

`$x contains text "bar"` =	iter	pos	item	score	tok
	1	1	true	0.3	τ_3
	1	1	true	0.4	τ_4

Now the distance attribute we have chosen above (`at most 1 word`) requires a certain pairing of the tokens: $(\tau_1, \tau_3), (\tau_2, \tau_3), (\tau_2, \tau_4)$. But the `and` operator does not know that it is used inside a function constraining distances[12], and hence cannot perform this pairing. It will produce something like

lhs. `and` *rhs.* =	iter	pos	item	score	tok
	1	1	true	0.1	τ_1
	1	1	true	0.2	τ_2
	1	1	true	0.3	τ_3
	1	1	true	0.4	τ_4

maybe with different scores. For the same reason, the function `distMax` must not make assumptions about the origin of its argument, hence it will add the (invalid) pairs $(\tau_1, \tau_2), (\tau_3, \tau_4)$ to the pairings determined above.

What is required to solve this issue, is a means to relate the tokens to each other, *i.e.* to annotate the tuples in the relation with information about which of them come from the same subexpression. This would require to add another attribute to the relation.

Alternatively, one may consider the text "`distance at most 1 word`" a modifier on the `ftand` operator. Then, unfolding would look like this:

```
andWithDistMax( words(1)
              , $x contains text "foo"
              , $x contains text "bar"
              )
```

[12] It must not know due to orthogonality. And telling it means to use another operator, which happens in the next paragraph.

This demonstrates that, in general, it is not possible to just aggregate token positions across an operator. Here it was required to supplement the `and` operator with a special `andWithDistMax` function which requires special treatment by the compiler, *i.e.*, it must be a primitive. But still, information about how these tokens relate to each other is missing.

To retain orthogonality in the XQuery part of the language, it is necessary to make the same consideration for every XQuery operator. Basically, adding a tok column requires to replicate the efforts undertaken to add the score column, *i.e.*, rethink every compilation rule. Let alone the difficulties introduced by rewriting the queries to seemingly equivalent ones, the added difficulty in comparison to scores is that it appears less obvious what propagation functions to choose.

* * *

The conclusion to draw from this section is that attaching just another column replicates and amplifies the trouble introduced by adding the score column: An extension to the core language is required (aka. the `score` keyword), the database should support null values, the relation between token positions still cannot be expressed, and the exact mechanics of token position propagation needs to be determined.

Section 7.3 sketches an approach to overcome this issue by encapsulating more information in a value of some Score type.

5.23 Scoring model parameters

This section summarises the scoring model parameters available for tweaking in the current implementation of the PathfinderFT architecture.

- smDefault attaches the default score μ to all values in an item sequence. Used, among others, for literals (Section 5.3), the creation of positions (Section 5.7), or when a score becomes a value, and needs a new score attached (Section 5.6).
- smLet aggregates the scores of an item sequence to form the item to be bound to a variable by the `score` keyword in a `let`-clause. Section 5.6.
- smStep aggregates the scores of all those context nodes that led to the same result node when an axis step is performed. Section 5.8.
- smAnd, smOr, and smNot provide score propagation across Boolean operators.
- smFun propagates the scores from the arguments to the result of a function or operator x. Used for the operators $x \in \{+, -, *, \texttt{div}, \texttt{mod}, \texttt{lt}, \texttt{le}, \texttt{gt}, \texttt{ge}, \texttt{ne}, \texttt{eq}\}$ (Section 5.14).
- smConditional merges the score of the predicate of a conditional expression with the score of the items returned from respective successful branch. Section 5.11.
- smUnion, and smIntersection combine the scores of elements appearing in both arguments of a respective operation. Section 5.12.
- smExist and smMiss are used to propagate the scores through the function `fn:exists`. The former aggregates scores of existing items, the latter invents a score if no items were passed. Section 5.13.

- smElem, and smContent are used for element construction to combine the score returned from computing the element's name with the scores returned from computing the content. Section 5.19.
- smContainsText combines the scores of a Full Text search context with the scores of the results of the search. Section 5.21.2.
- smWeight is used when weighting expressions are unfold from the Full Text language. It adjusts the score of an item(sequence) according to a provided weight. Section 5.22.4.3.

So, how to parametrise these functions to implement, say, tf-idf? The short answer is, you cannot do this: This is not the place to implement tf-idf, nor any other scoring mechanism.

It is important to realise that the above functions implement the calculation of a score for an expression, depending on the scores of subexpressions. This, however, is something beyond tf-idf, because it lacks a concept of subexpressions. The scoring model functions implement how the results of *different runs of the Full Text engine* shall be combined.

To clarify this, let us assume that there is a Full Text engine implementing something like tf-idf, *e.g.*, an expression e `contains text` t could trigger a run of tf-idf to calculate the relevance of the atomised text contents of e with respect to the term t and the "document collection" of all nodes in the database. Finally, a Boolean is calculated from the score, which could be done by checking whether it exceeds a certain threshold.

> Basically, tf-idf estimates the relevance of a *document* according to a *term* and a *collection* of documents, by relating the number of occurrences of the term within the document to the number of documents from the collection containing that term. Several variants of the actual formula exist, see [2].

Now consider the following query, containing two separate Full Text expressions:

```
$list[./y contains text "foo" and ./z contains text "bar"]
```

What happens here is that the contents of y are rated, and the contents of z are rated separately, the only connection between the calculations being the document collection used by tf-idf. The outcome of the conjunction is rather the product of two numbers (or whatever is chosen for smAnd) than a tf-idf measure, and it shall not be misconceived as such.

So, it is impossible to use tf-idf with PATHFINDER$^{\text{FT}}$? Not quite, since it is perfectly sane to multiply numbers. The implementer just has to be aware of the fact that the implementation of the abstract scoring model functions *adds semantics on the outside* to what the Full Text engine does. In other words, a *new algorithm* is created, which *uses* tf-idf on a lower level.

The above excuse was that tf-idf has no concept of subexpressions. But the argument also holds for algorithms that do have such a notion, *e.g.*, the TIJAH engine, which understands NEXI expressions. Here it is especially important to make a decision about whether the queries discussed in Section 5.22.4.1 are expected to return the same result, or not.

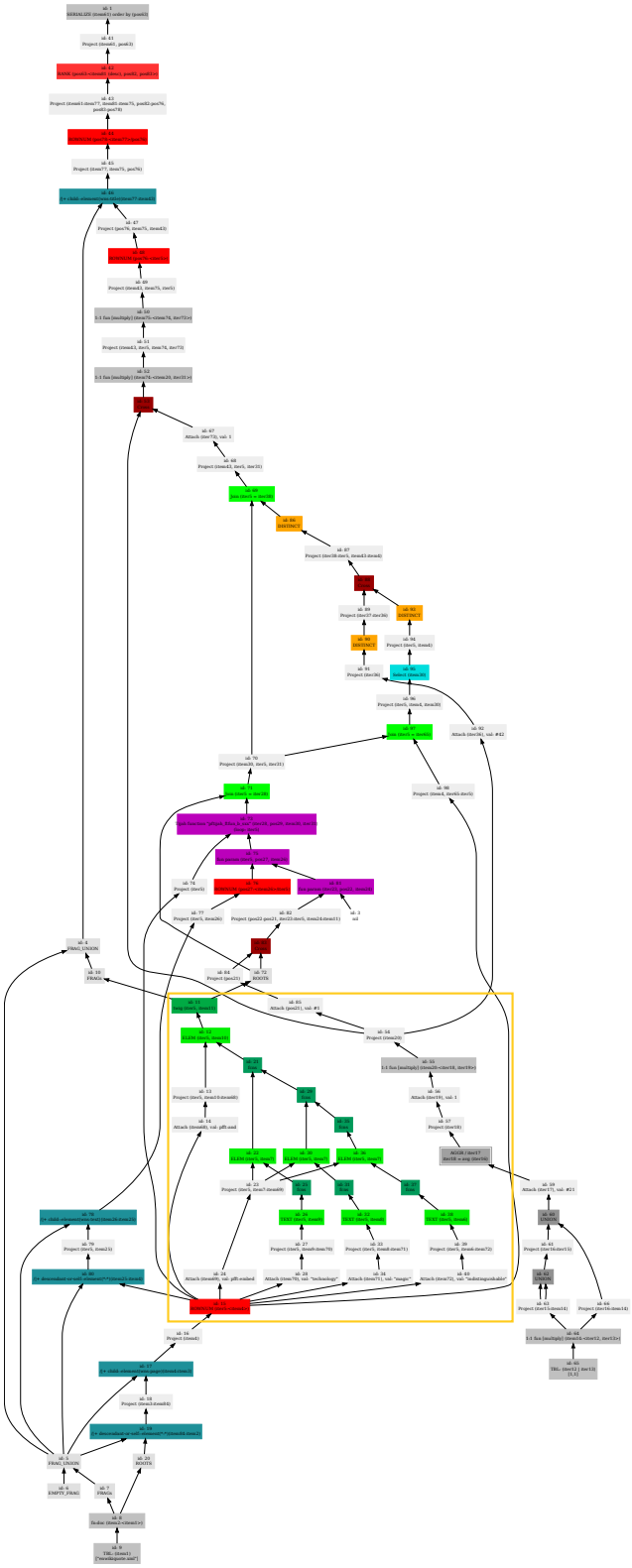


Figure 3: The Relational Algebra plan for the example query from Section 5.22.1 shows the twig representation of Full Text expression. (See enlarged detail in Figure 3a on page 83)

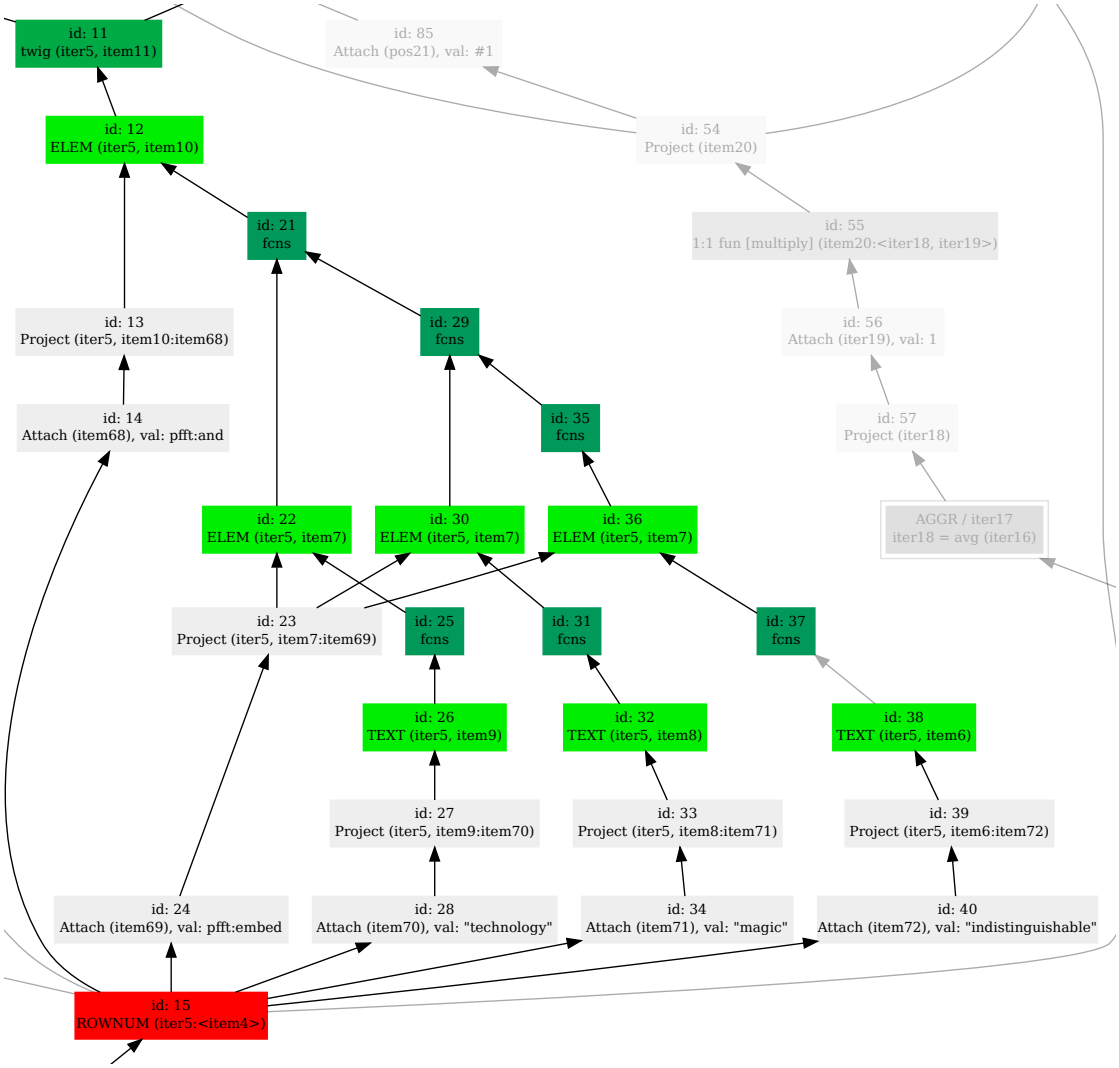


Figure 3a: (Detail of Figure 3 on page 82)

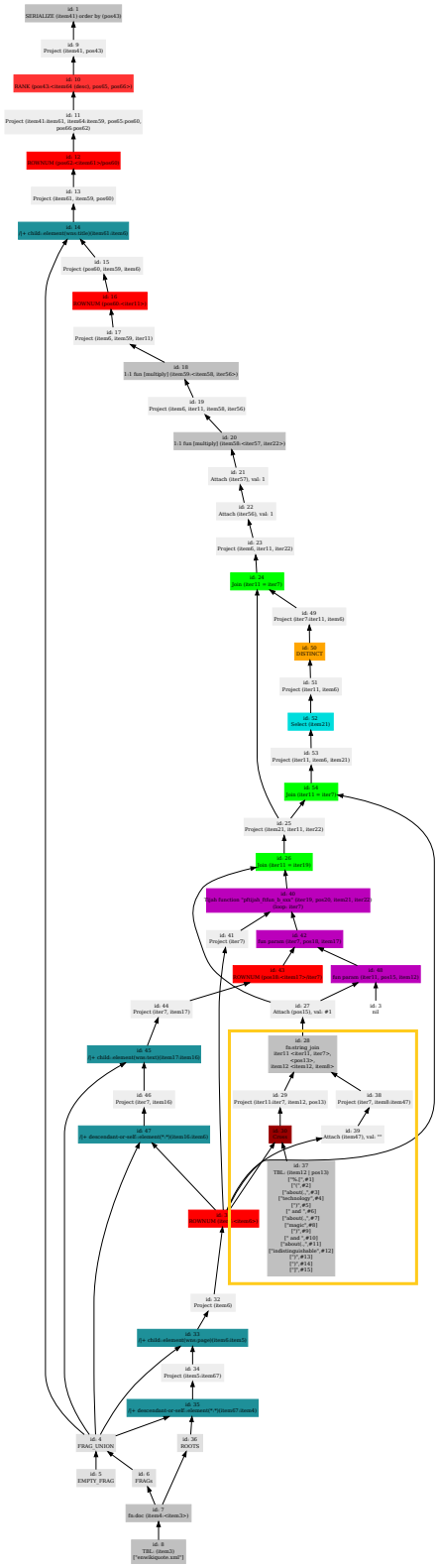


Figure 4: The Relational Algebra plan for the example query from Section 5.22.2 shows the NEXI query generated from the Full Text expression. (See enlarged detail in Figure 4a on page 85)

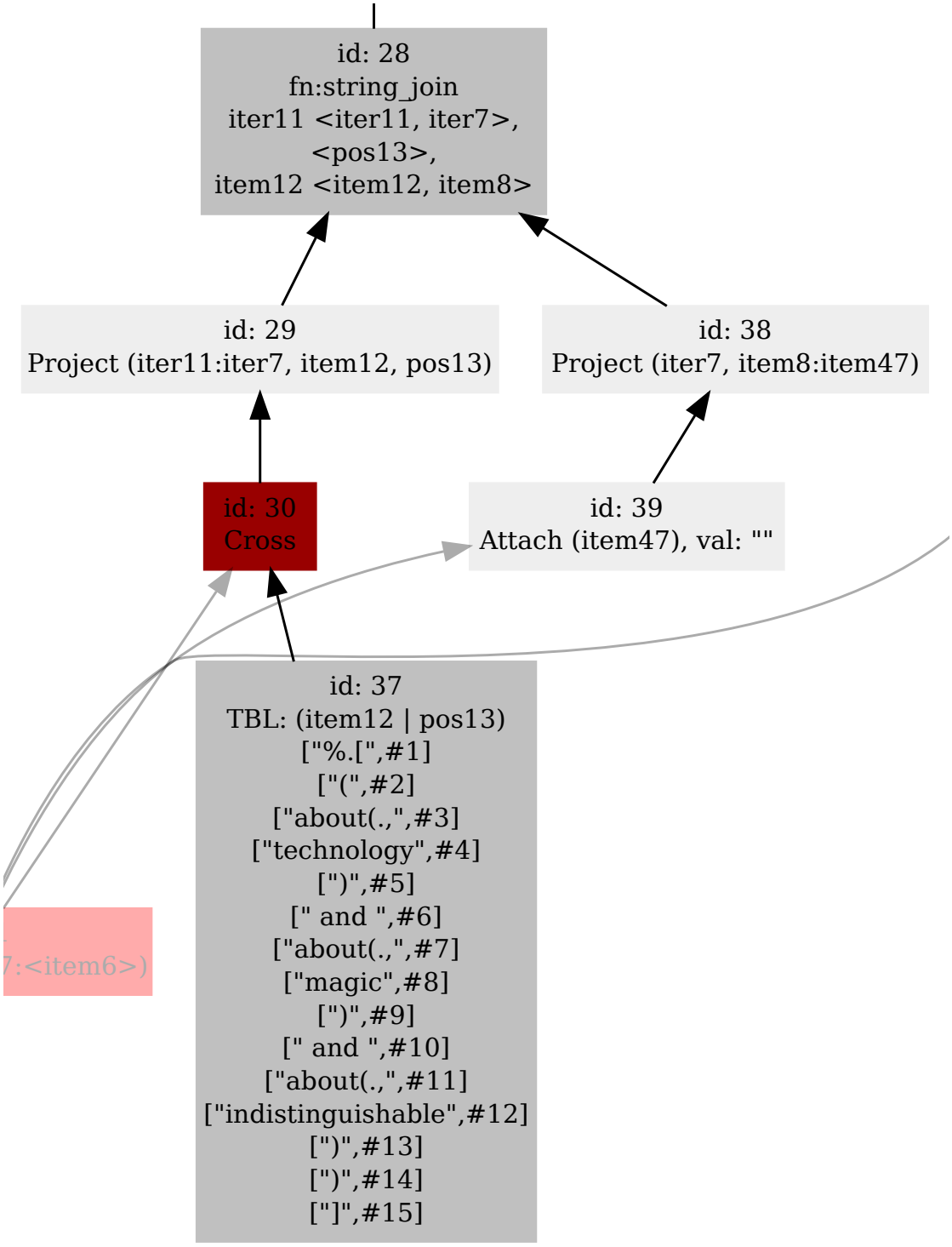


Figure 4a: (Detail of Figure 4 on page 84)

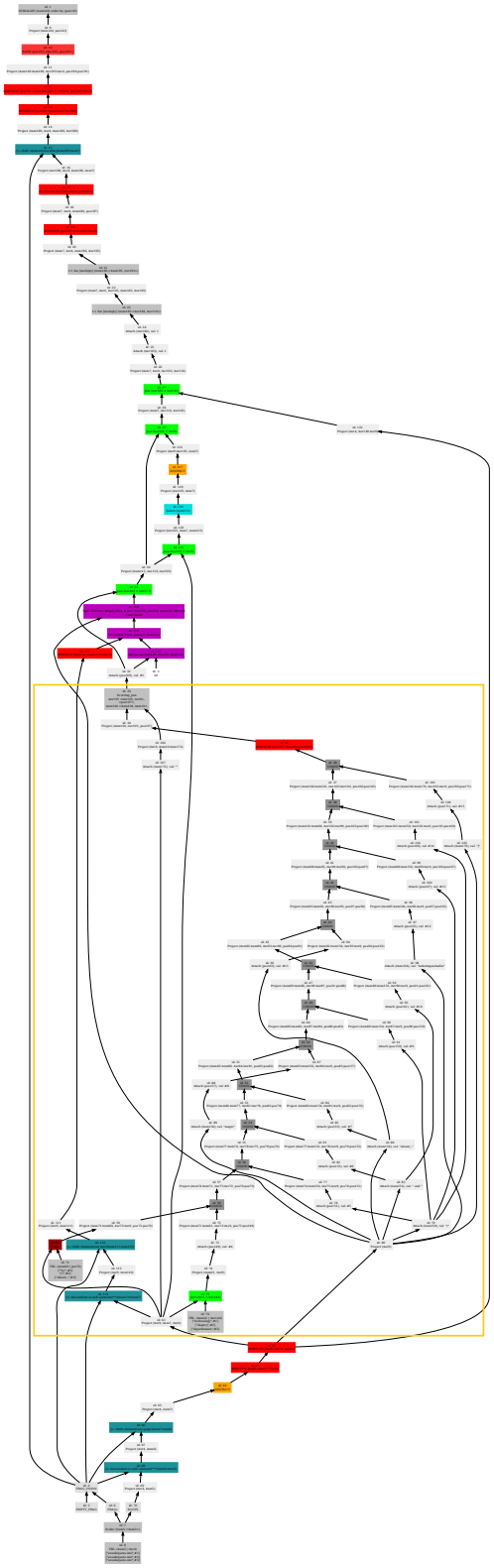


Figure 5: The Relational Algebra plan for the example query from Section 5.22.3 shows how the construction of the NEXI query requires calculations in the Relational Algebra domain. (See enlarged detail in Figure 5a on page 87)

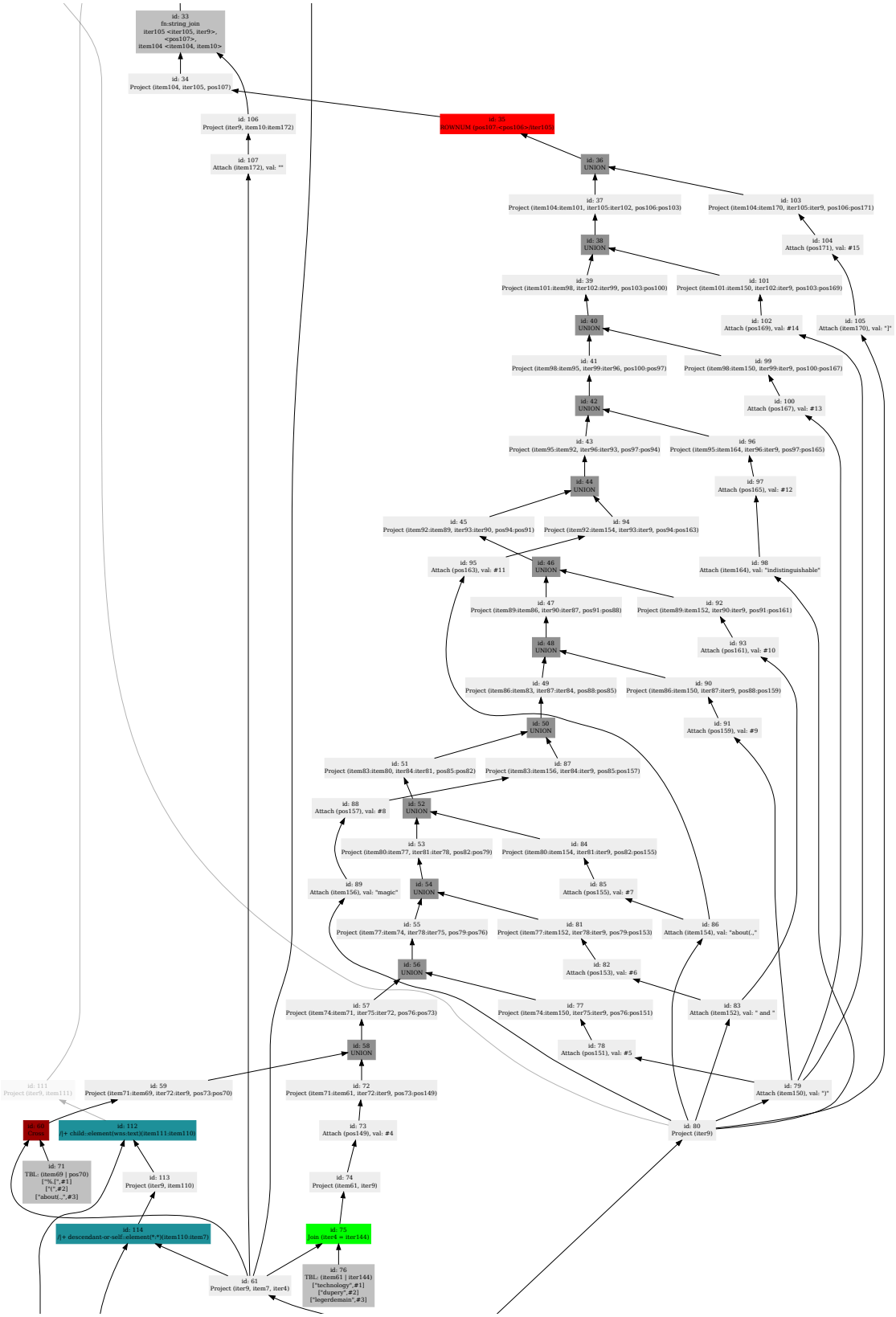


Figure 5a: (Detail of Figure 5 on page 86)

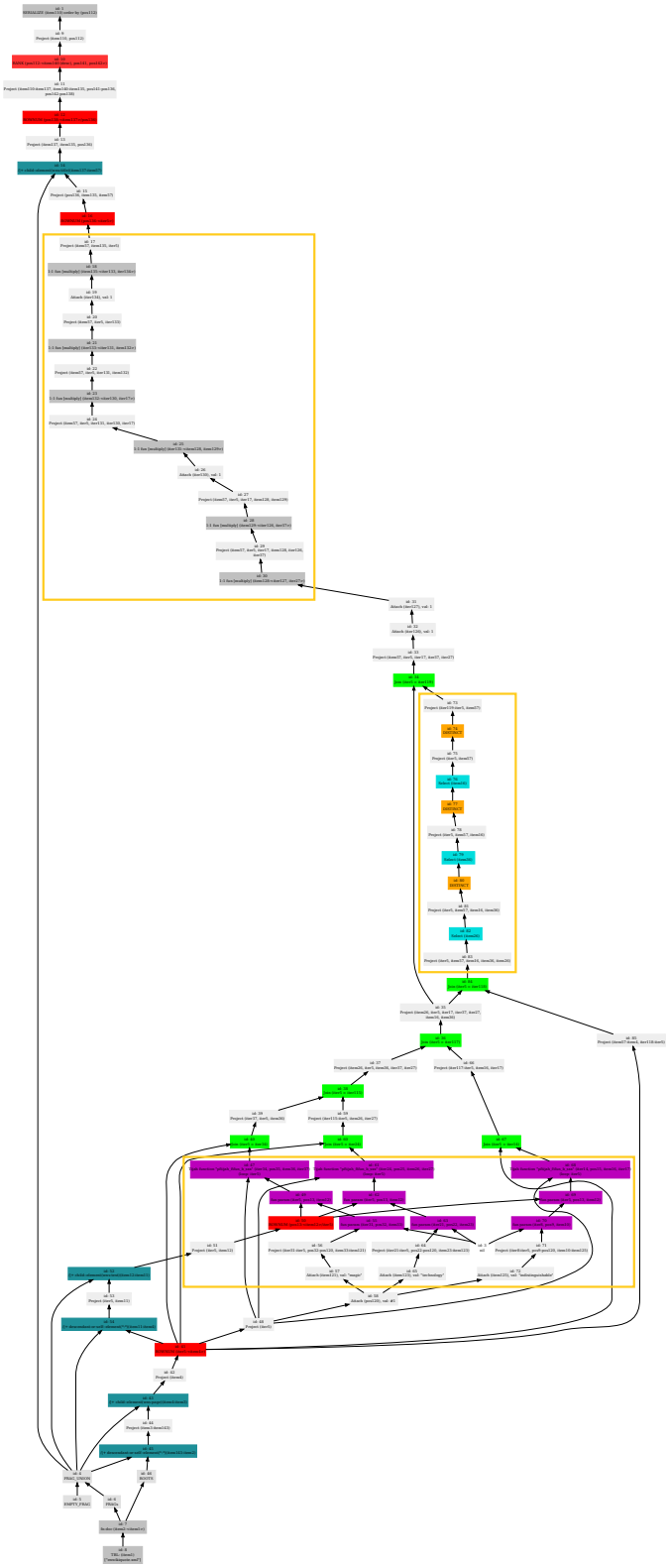


Figure 6: Unfolding (Section 5.22.4) creates one `pftijah` function call for each keyword (bottom). The Boolean operators are optimised to a sequence of selects (middle), the score computation is performed on the surviving nodes only (top). (See enlarged details in Figure 6a on page 89)

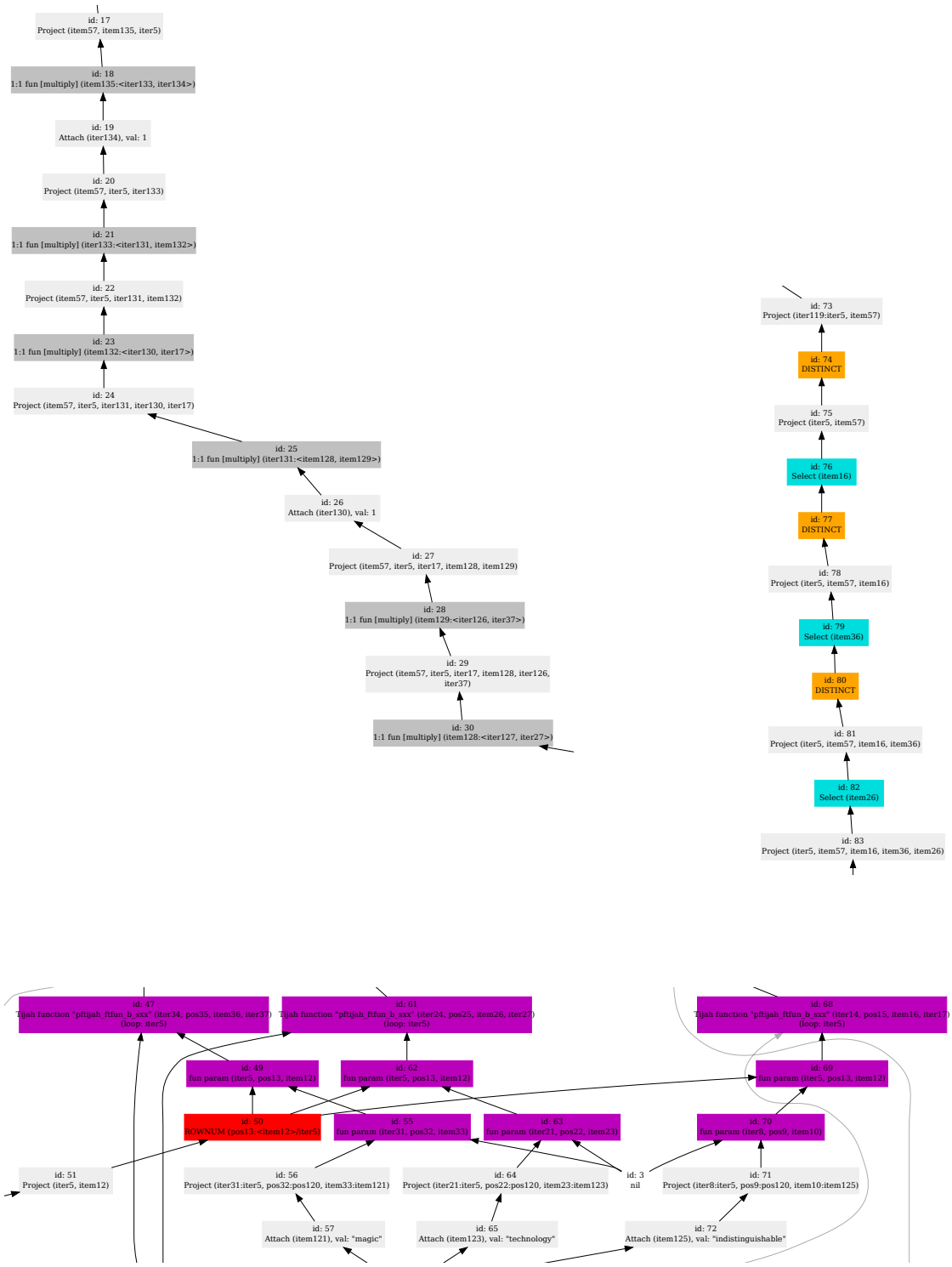


Figure 6a: (Details of Figure 6 on page 88)

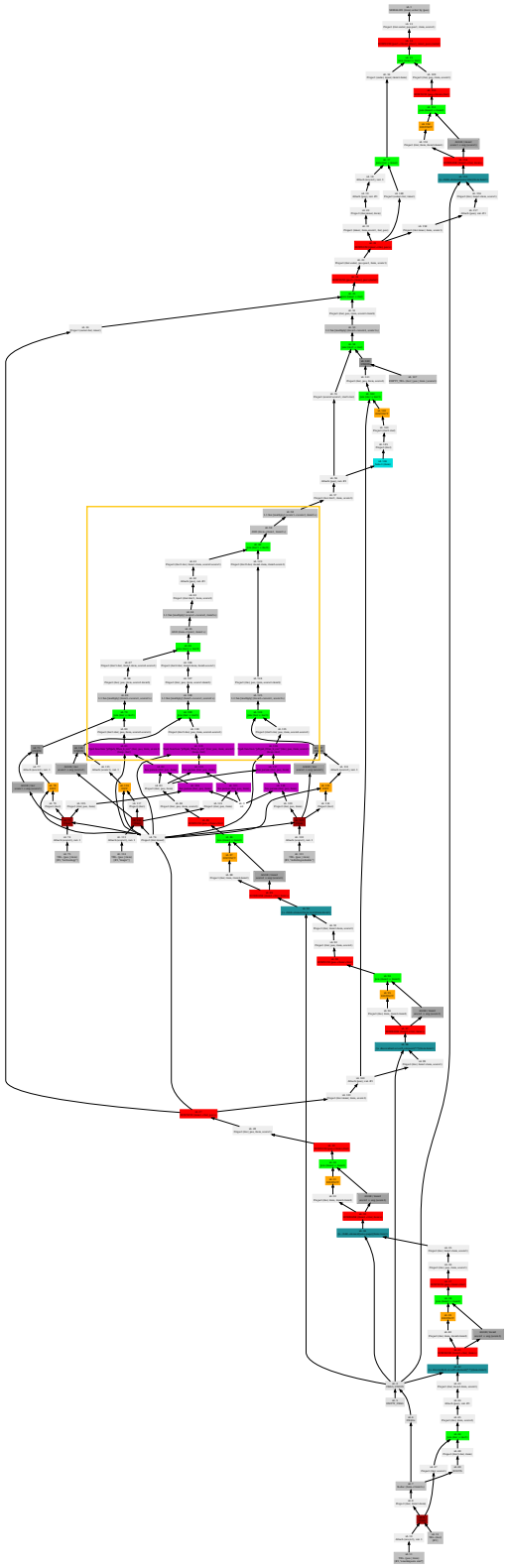


Figure 7: The unoptimised version of the plan given in Figure 6 on page 88. The AND operators are explicit, and interleaved with the multiplications for the corresponding score propagation. (See enlarged detail in Figure 7a on page 91)

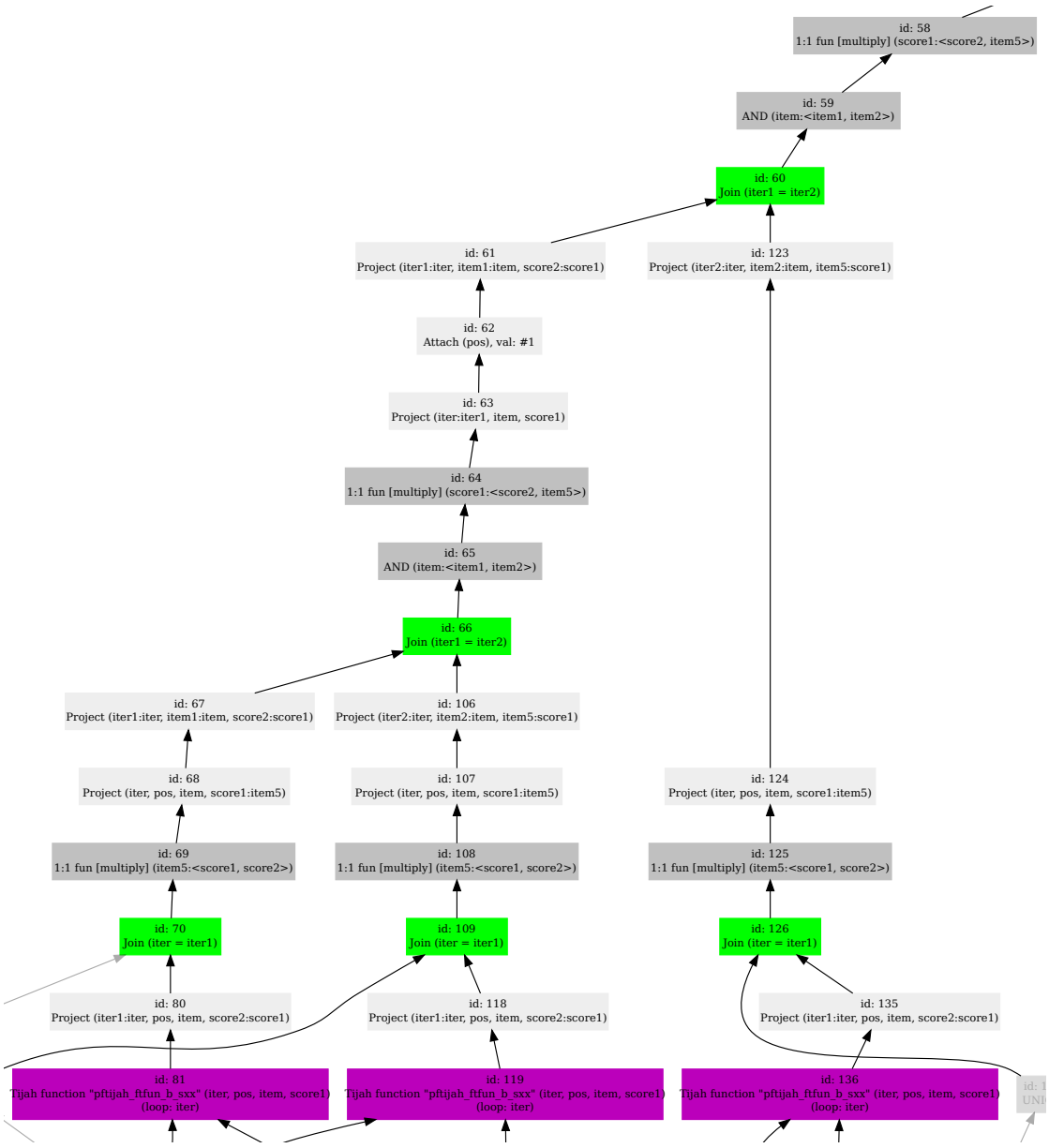


Figure 7a: (Detail of Figure 7 on page 90)

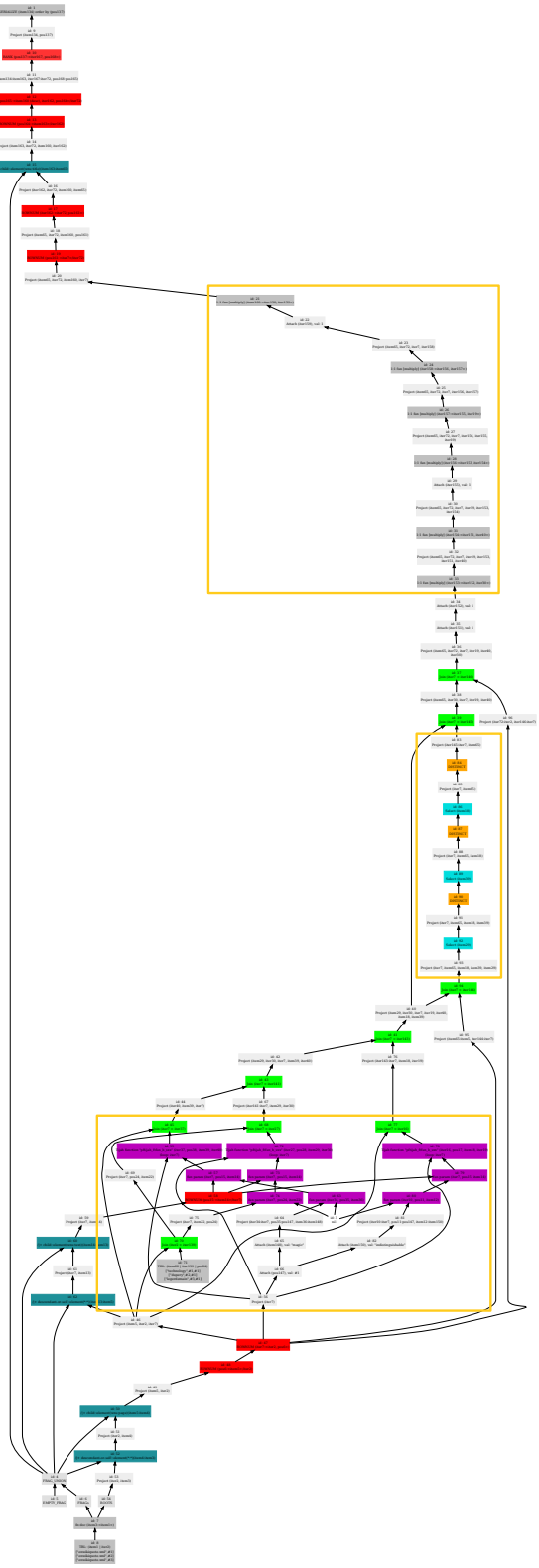


Figure 8: The plan generated from the query in Section 5.22.5. The separation of score and Boolean calculation is clearly visible. The variable search terms originate in the dark grey box on the right. (See enlarged details in Figure 8a on page 93)

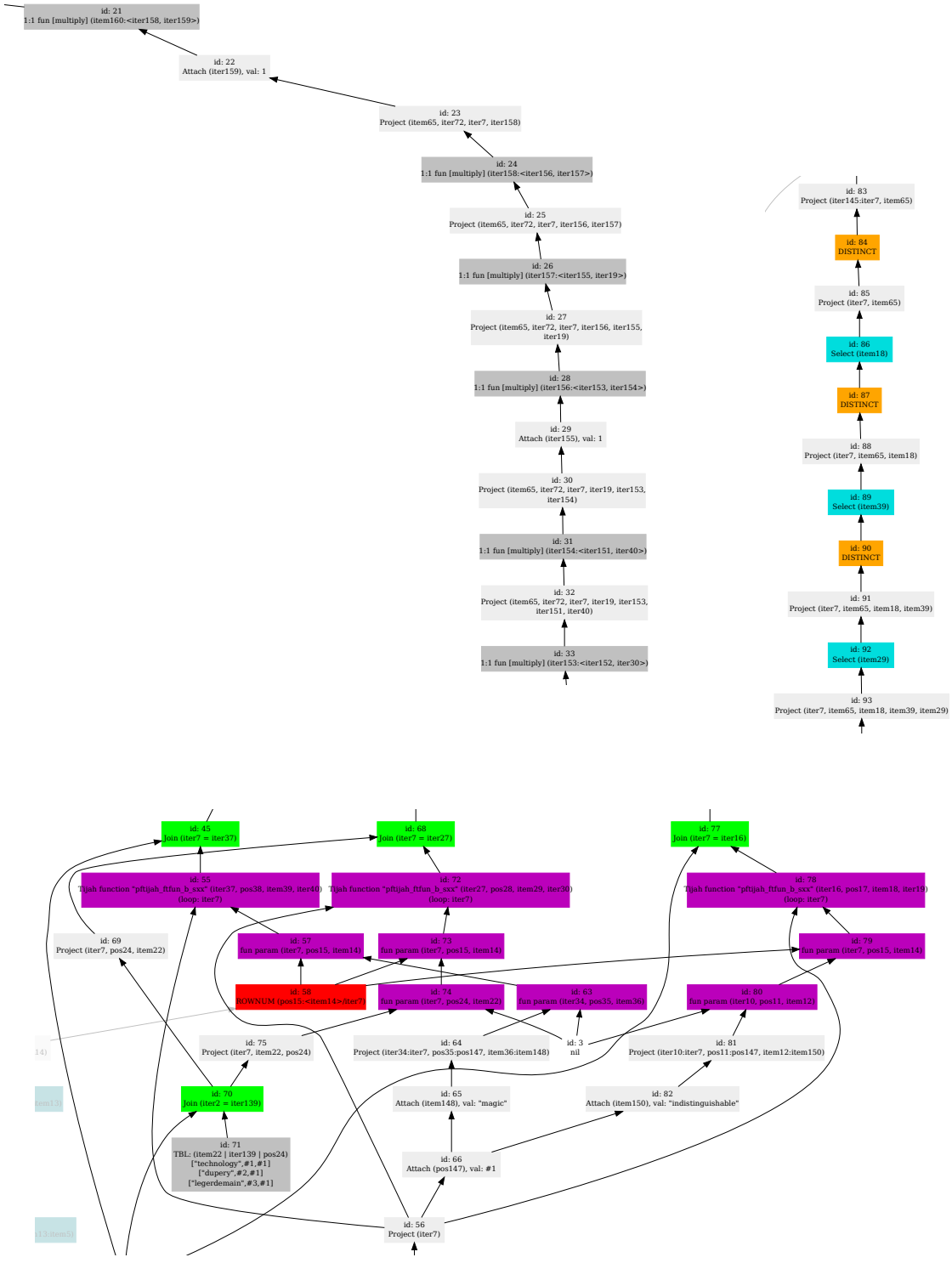


Figure 8a: (Details of Figure 8 on page 92)

Chapter 6

The Prototype Implementation

The ideas proposed in this thesis have been implemented in the PATHFINDER^{FT} compiler. Its current version still is a proof-of-concept only, *i.e.*, neither intended nor suitable for production use. The following discussion does not provide insights into the principal PATHFINDER^{FT} architecture, which has been discussed thoroughly in the previous chapters.

6.1 Goals & Achievements

On paper, the idea of simply extending PATHFINDER's relational item sequence encoding with another column looks nice. But is it a viable approach? To prove its feasibility, a prototype was implemented. The goals of this effort were

- to show that score propagation can actually be implemented in the proposed way,
- to verify the locations where the scoring model needs to be instructed about how to handle the scores.
- to demonstrate that the required changes are indeed quite isolated, *i.e.*, minimally invasive to the original PATHFINDER compiler, and
- to provide a platform for further development and evaluation of scoring models in the XQUERY FULL TEXT environment.

Proof of concept To the extent to which PATHFINDER^{FT} actually implements XQUERY FULL TEXT, the first objective can be considered accomplished. Although neither XQUERY, nor XQUERY FULL TEXT are implemented to their full scope, PATHFINDER^{FT} now is a reasonably complete, and runnable proof of concept. Unless otherwise noted, all compilation rules described in this thesis are implemented in the prototype.

Locality of required changes The required extensions gather in a very early compilation phase: The abstract syntax tree (generated by an adoption of the XQUERY FULL TEXT parser from Christian Grün's BASEX XML database [8]) is compiled into a Relational Algebra plan, an XML representation of which is handed on to the *original* PATHFINDER compiler. From there on,

query processing proceeds as for the non-Full Text case. Only the Full Text back-end must be capable to understand calls present in the Relational Algebra plan, and to return scored Booleans as required by the XQuery Full Text definition. Of course, the Pathfinder compiler must be capable to generate such calls, but this feature has already been available from the PF/Tijah project [14].

Although the Relational Algebra plans triggered one or two bugs in the Pathfinder compiler —which have not been observed until then, because the Pathfinder compiler has been confronted exclusively with plans generated by itself— I would not consider their fixing an *adoption* of the Pathfinder compiler to the needs of Pathfinder^{FT}.

Adoption was, however, required on the side of the Full Text index: The PF/Tijah index has not been used before in the very restricted way it is employed by Pathfinder^{FT}, but rather in a much broader way as depicted on page 31. But it is no surprise to find the need for adaptations here: The XQuery Full Text specification defines a certain return type for the `contains text` operator, and if a Full Text engine does not obey here, it naturally requires modification.

Identification of scoring model parameters The identification of scoring model parameters (see Section 5.23), the clean implementation in an accessible programming language (Haskell, further described in the following sections), and the description of issues summoned by the design of the XQuery and XQuery Full Text languages should open the door for further investigations in the XQuery Full Text environment.

6.2 Why Haskell?

At the time of writing, Haskell [23] seems to be the state-of-the-art functional programming language (FPL). It comes with an elaborate (*i.e.*, rich, clean, *and* comprehensible) syntax, a very active community, and a far more than sufficient set libraries.

The nature of a FPL suits the structure of an algebraic-style compiler very well, and Haskell proves being a useful tool for compiler prototyping: The 80.3kB Pathfinder^{FT} source is made up of 2288 lines[1] (including maintenance code and helper modules) of very readable code, plus comments, summing up to just above 144kB in total. Most of the compiler logic is assembled in the three modules ToCore, PathfinderFT, and Scoring, see Figure 9 on page 97. Together, these account for only 784 lines (40071kB) of code.

This chapter shows some of the code actually used in those modules, and it demonstrates a very close similarity between the actual implementation, and the formulas used in this thesis to describe the ideas.

After all: Programming with Haskell is fun.

6.3 Architecture

While Figure 2 on page 37 shows the overall architecture of the Pathfinder^{FT} compiler within its ecosystem, Figure 9 on page 97 shows some of the Haskell modules used to build the Path-

[1] Counted November 2010, using: `find PathfinderFT -type f -exec grep '^>' {} ';' | wc -l -c`

FINDER^{FT} compiler, and how they relate. Blue, oval nodes depict modules that model a language, while red, box-shaped nodes refer to processing modules.

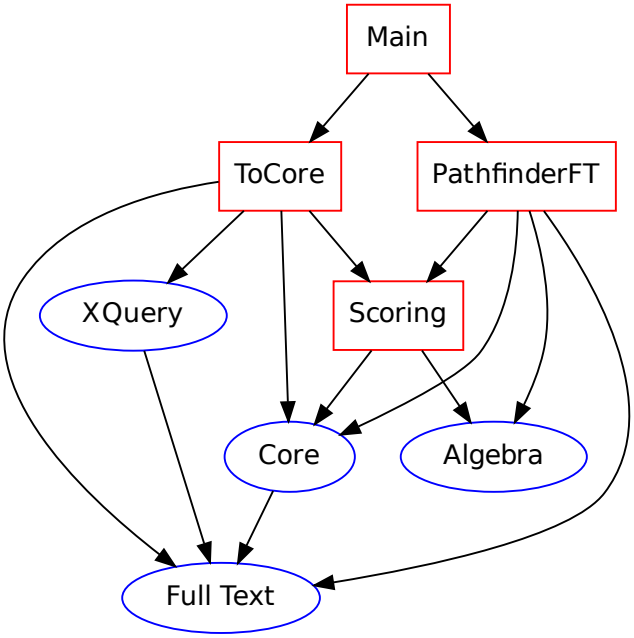


Figure 9: Dependencies between the most important HASKELL modules of the PATHFINDER^{FT} compiler.

The languages XQUERY, and XQUERY Core, both embed Full Text expressions. The main module first triggers the conversion from XQUERY to XQUERY Core, and then the core PATHFINDER^{FT} module which transforms XQUERY Core to Relational Algebra. Both of them make use of the scoring module, which hosts all information required to implement a scoring model. It is used during conversion to XQUERY Core (*e.g.*, to unfold weights, Section 5.22.4.3), and during compilation to Relational Algebra (*e.g.*, to attach the default score to a relation).

6.4 History of Development

Initially, I have partially reconstructed the PATHFINDER compiler as described in [21], to gain a deeper insight in the workings of the PATHFINDER architecture, and to explore possibilities for the desired extension. To this end, the HASKELL language [23] was a natural choice: It is very simple to build algebraic-style compilers in this language. A few examples will be shown in the remainder of this section.

After having a very basic re-implementation of the PATHFINDER compiler, I started exploring where scores would have to be propagated, and what consequences this had for the rest of the compiler. At that time, PATHFINDER^{FT} was lacking only a parser, a database back-end, an optimiser, the type system, and a scoring engine, *i.e.*, XQUERY ASTs expressed in HASKELL were compiled and the resulting Relational Algebra executed by a naive implementation of a relational interpreter. Scores were calculated with a Levenshtein algorithm.

When the concept proved feasible, it became desirable to actually have a version of PATHFINDER^{FT} that would be able to benefit from the Relational Algebra optimisations available in the original PATHFINDER compiler, and to work on real-life data, *i.e.*, larger XML instances than the ones processable with the naive interpreter available at that time. Also, the PF/TIJAH project was alive at that point in time, and it was tempting to use it as the Full Text scoring engine.

Sadly (or, from any other point of view, rather luckily) the PATHFINDER compiler had improved a lot in the meantime, and it took some time to actually synchronise the Relational Algebra produced by PATHFINDER^{FT} with the Relational Algebra understood by the PATHFINDER compiler. The two relational algebras were in fact so incompatible, that all compilation rules of the PATHFINDER^{FT} compiler had to be changed. While conceptually still the same idea was implemented, the target language to express it had subtly changed in too many places.

Fortunately, due to the modular and clean design of the HASKELL code, implementing the required changes was rather easy. So the sheer amount, the identification, and the comprehension of the misfits was the biggest obstacle here. During that phase, Jan Rittinger, from the Database Group in Tübingen, was a constant source of help, pointing out a lot of errors the plans generated by the PATHFINDER^{FT} compiler contained, and showing me some technical subtleties inherent in the PATHFINDER compiler's plans.

At about the same time Jan Flokstra, from the Database Group in Twente, provided me with the `pftijah` Relational Algebra function to interface the PF/TIJAH index, which is used in Section 5.21.2. He also allowed me to pass either keywords, or complete NEXI queries to the scoring engine. Finally, Leonard Wörteler, student of Computer Science in Konstanz, adopted the XQUERY FULL TEXT parser from Cristian Grün's BASEX XML database [8] to emit XQUERY FULL TEXT ASTs understood by the PATHFINDER^{FT} implementation.

Together with the optimising PATHFINDER compiler, the TIJAH index and the MONETDB RDBMS, the PATHFINDER^{FT} system forms a reasonable, though very basic, implementation of the architecture described in this thesis.

6.5 Query data structures

XQUERY FULL TEXT, and XQUERY Core expressions are modelled as algebraic data types `XQ` and `Core` respectively, reflecting the query structure in a very natural way. *E.g.*, an XQUERY item sequence is constructed with the constructor function

```
X_seq :: [XQ] -> XQ
```

In a first step, the query represented by an expression of type `XQ` is transformed into an expression of type `Core`, thereby trading some XQUERY expressions for more primitive XQUERY Core constructs:

```
toCore :: Pragmas -> XQ -> Core
```

The first argument `prg` is used to maintain pragmas, which may be used to control variations of the compilation rules.

One example of its use is the removal of Boolean predicates, which are replaced by explicit `for`-loops in the XQUERY Core language, as discussed in Section 5.16:

```
toCore prg (X_predBool e2 e1)              — core counterpart:
    = C_for dot "" "" e1' []               —   for . in e1'
```

```
  $                                    —   return
  C_if e2' (C_var dot) (C_seq [])      —   if e2' then . else ()
where
e1' = toCore prg e1
e2' = toCore prg e2
```

A Boolean predicate query like e_1`[`e_2`]` is represented by the HASKELL expression `X_predBool` e_2' e_1', with e_i' the expression derived for e_i. Note the twist in the ordering of arguments: The predicate e_2 is applied on the expression e_1.

Recursive calls to `toCore` provide XQUERY Core versions `e`i`'` of the subexpression `e`i. A dot occurring in a predicate is represented by a special variable `dot`. This variable is used as iteration variable in the created `for`-loop.

The constructor function

```
C_for :: String -> String -> String -> Core -> [Sort Core] -> Core -> Core
```

consumes three variable names for the iteration variable, the positional variable, and the score variable, then a core expression that computes the sequence to iterate over, a potentially empty list of expressions to calculate sort keys, and finally an expression that computes the return value for each iteration. An empty variable name means that no binding is created. The variable `dot` is special, in that it represents the `.` seen often in XQUERY predicates.

The representation of conditional expressions is even simpler, provided for by the constructor function

```
C_if :: Core -> Core -> Core -> Core
```

consuming the condition, and two branches for the true- and false-cases.

The other language constructs are simply implemented along these lines.

6.6 Plan data structures

For the construction of Relational Algebra plans, a *directed acyclic graph* (DAG) structure is required: Of course, this comes in handy to collapse identical sub-plans, and to avoid repetitive calculations. But the main reason is that *common subtree elimination* (CSE) provides means to simply handle the not referentially transparent semantics of XQUERY's node construction. The idea, explained in [21], is to

- identify all Relational Algebra operators witch have the same semantics and the same arguments,
- with the exception of node constructors, which are never identified.

The rationale of this being the fact that exactly the Relational Algebra operators that are free from side effects return the same result for the same arguments (*i.e.*, they are referentially transparent). Node constructors, however, always return different results, *i.e.*, they must not be identified, even if they are applied on the same arguments. See also Section 2.3.1 for a discussion of referential transparency.

6.6.1 The DAG structure

The DAG structure is simply a map from References (Integers do) to *pathfinder algebra nodes* (PAN), and a plan is a DAG with a pointer to the root node. The `M` prefix points to names from the `Data.Map` module.

```
type DAG = M.Map Ref PAN
type Plan = (DAG, Ref)
```

A PAN contains a *pathfinder node description* (PND), describing the operation to be performed, and a list of arguments to the operator, identified by their references in the DAG. A schema is added for sanity-checking of the created plans, a tool useful for debugging, but currently not used otherwise by the PATHFINDER^{FT} compiler.

```
data PAN
    = PAN { pnd :: PND
          , schema :: Schema    — not discussed in the following
          , args :: [Ref]
          }
    deriving (Eq, Ord, Show)
```

Now the `PND` data type simply enumerates all Relational Algebra operators, providing means to describe their semantic arguments, *e.g.*, what columns to use for projection, or which function to use for aggregation. Only an excerpt of the 38 operators is shown here. `Name` encodes attribute names, `Val` holds primitive, though polymorphic values (*i.e.*, integers, strings, node surrogates, *etc.*):

```
data PND
    | P_attach Name Val
    | P_union
    | P_difference
    | P_eqjoin Name Name
    | P_select Name
    | P_project [OnTo]
        — data OnTo = On Name | To Name Name describes the projections
    | P_agg Agg Name Name (Maybe Name)
        — aggregation: agg_{score:avg score/iter} becomes P_agg A_avg Score Score (Just Iter)
    ...
    | P_doctbl Name Name
    | P_element Name Name
    | P_textnode Name Name
    | P_twig Name Name
    | P_roots
    | P_fragUnion
    | P_step Axis Nodetest Name Name  — axis step
    deriving (Eq, Show, Ord)
```

One point to observe in the above definitions is the fact that `PND` can be ordered, by deriving `Ord`. This is an important feature for the intended DAG construction with built-in CSE: By maintaining a reverse mapping (which requires ordering for the keys)

```
type Signature = M.Map (PND,[Ref]) Ref
```

it is easy to determine whether an operator with a given set of arguments (identified by the pair `(PND,[Ref])`) has already been added to the DAG, and, if so, find its reference number.

6.6.2 Monadic DAG construction

To simplify the creation of Relational Algebra plans, a monadic DAG constructor is defined, which automatically provides CSE where appropriate, guarantees well-formedness of the generated plan (no dangling pointers, *i.e.*, no references to undefined nodes), and which can further provide sanity checks not discussed in this work. The result is an embedded domain-specific language, examples of its use being shown in Section 6.7.

A simple monadic state transformer is used, with the additional ability to throw/catch error messages. The infrastructure for error handling is not discussed here. While it is useful for debugging plan generation, it is not required to perform the principal task.

```
data ST e s a = ST (s -> Either e (s, a))
instance (STError e) => Monad (ST e s)
```

The monadic DAG constructor is a specialisation of the state transformer

```
type Planner a = ST String (Ref, DAG, Signature) a
```

the state being a triple of

1. `Ref`, the last reference number used, simplifies finding the next free reference to assign to a new node,
2. `DAG`, the DAG constructed so far, and
3. `Signature`, the reverse mapping introduced on page 100.

The `Planner` is controlled via two functions, only the signature of which is given here:

```
known :: PND -> [Ref] -> Planner (Maybe Ref)
append :: PND -> [Ref] -> Planner Ref
```

Given a PND and a list of argument references to apply it on, the `known` function returns `Just` r bearing its reference, iff such an operation is already present in the DAG, by simply looking it up in the `Signature` map. Otherwise `Nothing is returned`.

The `append` function adds a PAN constructed from the PND and the argument reference list to the DAG, no matter whether a node with the same signature already exists. A reference to the newly created PAN is returned, and may be used as argument for further operations, or as DAG root.

These two functions are used by the `add` function, which determines whether CSE should be used for the operator to be added, or not. Its complete definition is:

```
add :: PND -> [Ref] -> Planner Ref
-- always add ops with side effects
add p@(P_element {}) as = append p as
add p@(P_textnode {}) as = append p as
add p@(P_content {}) as = append p as
-- perform CSE for all other ops
add p as = known p as >>= maybe (append p as) return
```

The last line makes `add` return the reference to an existing node with the same signature if one exists, or the reference of a newly constructed node otherwise.

Finally, `add` and the PND constructor functions are used to define the front-end functions in a rather boring and repetitive style. Again, only some operators are shown:

```
attach n v r       = add (P_attach n v) [r]
cross l r          = add P_cross [l,r]
union r1 r2        = add P_union [r1, r2]
difference r1 r2   = add P_difference [r1,r2]
eqjoin n1 r1 n2 r2 = add (P_eqjoin n1 n2) [r1,r2]
select n r         = add (P_select n) [r]
project ps r       = add (P_project ps) [r]
...
```

As a toy example, consider the following Relational Algebra expression. Given two relations $r_1, r_2 :: \mathsf{Rel}_{\mathsf{iter,item}}$, find those iterations where the items are equal:

$$\pi_{\mathsf{iter}} \triangleleft \sigma_{\mathsf{item2}} \triangleleft \mathrm{op}^{\mathsf{item2:}}_{\mathsf{item1=item}} \triangleleft (\pi^{\mathsf{iter1:iter}}_{\mathsf{item1:item}} r_1 \bowtie_{\mathsf{iter1=iter}} r_2)$$

The join operator can be applied partially to one operator, returning a function (this style of partial operator application is known as *currying*):

$$\pi_{\mathsf{iter}} \triangleleft \sigma_{\mathsf{item2}} \triangleleft \mathrm{op}^{\mathsf{item2:}}_{\mathsf{item1=item}} \triangleleft (\bowtie_{\mathsf{iter1=iter}} r_2) \triangleleft \pi^{\mathsf{iter1:iter}}_{\mathsf{item1:item}} r_1$$

Given two references `r1, r2 :: Ref`, referring to the calculations of r_1, r_2 respectively, the implementing Haskell code is simply a copy of the above formula, in reverse order, and using slightly different syntax.

```
project [To Iter1 Iter, To Item1 Item] r1
>>=
eqjoin Iter r2 Iter1
>>=
op O_eq Item2 [Item1, Item]
>>=
select Item2
>>=
project [On Iter]
```

This expression is of type `Planner Ref`, indicating that it (probably) modifies the DAG, and returns a reference for further usage.

Of course, this style allows for imperative style "variable binding". A similar implementation of the above example is the less elegant, but maybe more catchy version below:

```
do r1' <- project [To Iter1 Iter, To Item1 Item] r1
   r2' <- project [To Iter2 Iter, To Item1 Item] r2
   eqjoin Iter2 r2' Iter1 r1' >>= op O_eq Item [Item1, Item2]
                              >>= select Item >>= project [On Iter]
```

Here, the references from calculating the projections are stored in variables `r1', r2' :: Ref` respectively. The last two lines form a single monadic action whose result is returned by the `do`-block.

Although I have not introduced the types nor semantics of the complete embedded language, it should be possible now to grasp how the Relational Algebra plans are constructed.

6.7 Compilation

This section shows some of the code actually present in the PATHFINDER^{FT} prototype implementation. As such, it is a little bit more technical to read and understand than the formulas presented in Chapter 5. However, the resemblance between HASKELL code and formulas should be obvious by now, and make it easy to grasp the additional details.

6.7.1 Fragment handling

As mentioned before (see Section 5.2.1), a compilation step not only returns the Relational Algebra code to calculate an item sequence, but also information about all the fragments incarnated by the subexpression. This is reflected by the type of the main compilation function

```
comp :: Env -> Core -> Planner (Ref, FragU)
```

which takes an environment containing the variable lookup function Γ, the loop relation, and pragma information as first argument, and then maps an XQUERY Core expression to a monadic DAG constructor that returns a reference to the root of the generated expression, and also fragment information.

```
data Env = Env {gamma :: Gamma, loop :: Ref, pragma :: Pragmas}
```

For now, one can consider `FragU` to represent a disjoint union of all fragments generated by the compiled expression. Such fragment unions can be united further with the function

```
mbFragUnion :: FragU -> FragU -> Planner FragU
```

as will be demonstrated in the next sections.

The truth is a bit more complicated than that, because the PATHFINDER compiler does not actually accept an arbitrarily nested expression of unions of fragments, but insists on a linked list instead. Thus, `FragU` rather contains a list of fragments generated so far, which is united only when used, and then in a degenerated left-deep style. Also `mbFragUnion` refuses to add empty fragments to that list.

A convention for readability: If an XQUERY Core expression stored in variable `e` is compiled, the returned reference and fragments are often named `e'`, and `e_` respectively.

6.7.2 Direct score manipulation

One of the simplest, though interesting, compilation rules is probably direct score manipulation, Section 5.9. The HASKELL implementation reads

```
comp env (C_scored e1 e2)
    = do (e1',e1_) <- comp env e1
         e1' <- cast Score1 Item T_dbl e1'
                >>=
```

```
        project [To Score Score1, To Iter1 Iter]
(e2',e2_) <- comp env e2
e2' <- project [On Iter, On Pos, On Item] e2'

seq <- eqjoin Iter1 e1' Iter e2'
       >>=
       project [On Iter, On Pos, On Item, On Score]
frg <- mbFragUnion e1_ e2_
return (seq,frg)
```

There are some things to note here:

Imperative style "variable assignment with overwriting" is used two times when calculating `e1'`, and `e2'` respectively. Of course these are different variables, the latter incarnation *obscuring* the earlier one, rather than overwriting it.

The item attribute from e_1 is casted to a double when being introduced as score. Casting is never shown in the compilation rules given in Chapter 5. Hence, the projection shown in Section 5.9 looks a bit different here.

The fragment union is calculated as explained in the previous section, and the pair of the reference `seq` to the resulting Relational Algebra expression and the calculated fragment union is returned.

6.7.3 Sequences

Another example, which also shows HASKELL's usefulness for simply expressing complex construction tasks, is the compilation of sequence expressions. The ellipsis given in the compilation rule in Section 5.5 is replaced with a combination of `mapM` and `zipWithM` in the real code:

```
comp env (C_seq xs) — xs not empty
    = do (xs',xs\_) <- mapM (comp env) xs >== unzip
         seq <- zipWithM (attach Ord) (map N [1..]) xs'
                >>=
                foldl1M union
                >>=
                rownum Pos1 [Asc Ord,Asc Pos] (Just Iter)
                >>=
                project [On Iter, To Pos Pos1, On Item, On Score]
         frg <- mbFragUnions xs_
         return (seq,frg)
```

For a monadic structure m (such as monadic DAG construction), the function `mapM` defined in the module `Control.Monad`, has the type

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

and it applies the provided function on each element of its list argument, forming a list of results. Furthermore, the monadic computations of each such application are sequenced by the `>>=` operator, making sure that all modifications to the generated plan are applied in order.

With this, it is easy to compile a list `xs` of XQUERY Core expressions in one go, yielding a list of type `[(Ref,FragU)]` that can be split into a list `xs'` of references and a list `xs_` of fragment

unions by `unzip`[2].

The binary version of `mapM` is

```
zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
```

also defined in `Control.Monad`. It pairs the elements from the two list arguments by application of the provided function, and constructs a list from the results. Just as `mapM` does, the monadic computations sequenced. Thus, the above HASKELL code calculating `seq` implements something rather similar to the following Relational Algebra expression:

$$\pi_{\substack{\mathsf{iter}\\ \mathsf{pos:pos1}\\ \mathsf{item}\\ \mathsf{score}}} \triangleleft \varrho_{\substack{\mathsf{pos1:(ord,pos)}\\ /\mathsf{iter}}} \triangleleft \biguplus_{i=1}^{n} @_{\mathsf{ord}:i} [\![e_i]\!]_{\Gamma,L}$$

6.7.4 Axis steps

An example that actually performs some score propagation and fragment union is the axis step explained in Section 5.8.

Consider the XQUERY FULL TEXT expression $e/\alpha::n$. To perform an axis step on the fragments made available by expression e, the union of fragments returned from compiling e must be actually built, see Section 6.7.1. This is done via

```
mkFragUnion :: FragU -> Planner Ref
```

which is also a monadic DAG construction function, *i.e.*, if code for calculating the same fragment union has already been added to the DAG, only its reference will be returned. Otherwise the required calculation is added to the DAG.

The attribute `ctx` described in Section 5.8 is named `Item1` here[3], and is used as argument to the step operator, which then adds an `Item` column containing the result nodes.

```
comp env (C_step a n e)
    = do (e',e_) <- comp env e
         f <- mkFragUnion e_
         g <- project [On Iter, To Item1 Item, On Score] e'
              >>=
              step a n Item Item1 f
              >>=
              rowrank Item2 [Asc Iter, Asc Item]
         d <- project [On Iter, On Item, To Item3 Item2] g
              >>=
              distinct
         r <- smStep (pragma env) g
              >>=
              eqjoin Item3 d Item2
              >>=
              rownum Pos [Asc Item] (Just Iter)
              >>=
```

[2]Here, `>==` is a shorthand for lifting `unzip` into the monad.

[3]For technical reasons, the PATHFINDER compiler only accepts a limited set of attribute names.

```
            project [On Iter, On Pos, On Item, On Score]
        return (r,e_)
```

As all score propagation functions, `smStep` is defined in a dedicated module `PathfinderFT.Scoring`. Its type

```
smStep :: Pragmas -> Ref -> Planner Ref
```

indicates that it consumes pragmas (taken from the compilation environment by `pragma env`) that could be used to steer its exact behaviour.

The implementation of smStep given in Section 5.8 is straight forward, `A_avg` names the aggregator function avg to be used, and the pragmas are ignored:

```
smStep _ = agg A_avg Score Score (Just Item2)
```

6.7.5 Pragmas control score propagation

The Boolean operators are a good example to demonstrate the use of pragmas to control the score propagation. PATHFINDER^{FT} comes with the following implementation of smOr, which contains all the alternatives described in Section 5.10:

```
smOr :: Pragmas -> Ref -> Planner Ref

smOr prg r

    | pragmaTest prg ftBool_negInv — ⟨a|x⟩^∨⟨b|y⟩ = ⟨a^∨b|x + y − x · y⟩
      = return r
        >>=
        op O_add Item3 [Score1, Score2]
        >>=
        op O_mul Item4 [Score1, Score2]
        >>=
        op O_subtract Score [Item3, Item4]

    | pragmaTest prg ftBool_negId — ⟨a|x⟩^∨⟨b|y⟩ = ⟨a^∨b|x · y⟩
      = op O_mul Score [Score1, Score2] r

    | pragmaTest prg ftBool_extreme — ⟨a|x⟩^∨⟨b|y⟩ = ⟨a^∨b|max x y⟩
      = do x <- return r
                >>=
                op O_gt Item3 [Score1, Score2]
                >>=
                op O_not Item4 [Item3]
           a <- select Item3 x
                >>=
                project [On Iter1, On Item, To Score Score1]
           b <- select Item4 x
                >>=
                project [On Iter1, On Item, To Score Score2]
           union a b
```

```
    | otherwise
        = error "Scoring.smOr: need pragma"
```

The user may choose any of the three branches by setting the XQuery Full Text pragma `pfft:ftBool` to one of the values `negInv`, `negId`, or `extreme`. All currently set pragmas are passed to `smOr` in the `prg` variable. The above code uses `pragmaTest` in the guard expressions to discriminate amongst known pragmas.

Chapter 7

Future Work

7.1 Performance testing

So how does this approach perform? While, of course, performance is an important aspect of query processing, the focus of this work clearly lies on the conceptual difficulties and challenges.

Albeit I have shown (*e.g.*, see Section 5.22.4.2) that proper plan generation may have a tremendous effect on query evaluation speed, performance testing is not part of this thesis. For solid performance measurements, at least the following aspects would need prior consideration.

Performance of the compiler Currently, parsing is done by an adoption of the BaseX parser, which serialises the AST in form of Haskell code. This is parsed by the Pathfinder^{FT} compiler before plan generation starts. The generated Relational Algebra plan is printed as an XML representation, which is then consumed by the original Pathfinder compiler.

Obviously, in this setting, measuring the performance of plan generation of the prototype Pathfinder^{FT} compiler —whose main design goal is comprehensibility, not performance— is somewhat pointless. To actually compare the performance of the underlying idea, it is certainly required to first re-implement Pathfinder^{FT} *inside* the original Pathfinder compiler.

Performance of the Full Text engine Since Pathfinder^{FT} is designed to be somewhat independent of the actual Full Text engine used, its performance should appear as a factor in the overall performance findings. One the other hand, different Full Text engines may provide different services, *e.g.*, one may provide a NEXI interface, another one may only allow for simple keyword search. Hence, completely abstracting from the Full Text engine may become difficult, since its capabilities restrict the set of feasible plans.

To isolate the performance of any Pathfinder^{FT}-based XQuery Full Text implementation, one would have to measure the used Full Text engine on its own, embedded in another environment. Only then the gain, or loss, of performance caused by the Pathfinder^{FT} architecture could be estimated.

Plan optimisation Pathfinder^{FT} does not perform any optimisations on the generated plan. All these tasks are left to the original Pathfinder compiler (*e.g.*, see Figure 7 on page 90 and

Figure 6 on page 88). It may be worthwhile to investigate whether the PATHFINDER optimiser could make use of further knowledge about the plans PATHFINDER$^{\text{FT}}$ generates.

Interfacing the Full Text engine Currently, the Full Text engine is called via a function call (see Section 5.21.2), and the PATHFINDER optimiser makes no assumptions about the algebraic properties of the called function. Making the Full Text interface a primitive Relational Algebra operator, and teaching PATHFINDER about its properties, could possibly open the door for much more comprehensive optimisations.

7.2 Non-determinism

The example scoring result′ on page 77 introduces the rather nasty issue of non-determinism, *i.e.*, multiple values of truth for each iteration and position in the item sequence. So how to make a Boolean decision?

Recall the compilation rule for conditional expressions (Section 5.11). We used a select and a difference on the loop relation to discriminate between the two branches to choose from. This is not sufficient here any more, because different truth values may occur "simultaneously". For each iteration, one could count the trues and falses, and relate their numbers. One could also weight this process by the scores (which makes sense if the score represents confidence in the item, but does not, if the score is just a finer grained measure of truth).

A more natural means to deal with this situation would be to skip the aggregation of the item column, and explicitly *allow* different values. From a probabilistic point of view, one might say that the score attached to an item does not only represent its membership in the item sequence (*i.e.*, fuzzy set membership), but instead the probability of its occurrence at exactly this position in the sequence.

This approach *en passant* answers the question for the semantics of conditional expressions with a fuzzy condition, because the combination of the results from both branches becomes quite natural. As an example, consider this query:

```
if doc("schroedingers-cat.xml") contains text "poison"
then "dead"
else "alive"
```

In case the Full Text engine determines the condition to yield

iter	pos	item	score	tok
1	1	true	0.5	τ_1
1	1	false	0.5	τ_2

representing a 50% chance for the cat to die, then the superposition of the cat's status is easily represented by

iter	pos	item	score	tok
1	1	"dead"	0.5	τ_1
1	1	"alive"	0.5	τ_2

The question is how to represent measurement. In the end, *i.e.*, after evaluating the complete query, one will see a relation representing several possible outcomes of an experiment.

This approach can be seen as an adoption of what [12] presents, see Section 5.11: Only one variable is used (*i.e.*, the expression result), and the scores annotated to the result represent the probability of this assignment becoming true.

7.3 Other interpretations of Score

Section 5.22.6 discussed how the sequence encoding schema could be extended with an additional tok column, containing token positions. The trouble encountered renders this approach less desirable.

Instead of adding a tok column, one might extend the Score type, *i.e.*, move the extension to the already attached scores. With XML at our fingertips, it seems feasible to use an XML structure to encode the findings of the Full Text engine. Although the XQuery Full Text specification [1] requires `contains text` to return a Boolean with a score, it describes an XML representation[1] of the AllMatches found by the Full Text engine.

Of course, score propagation becomes much more difficult, because it would not combine numbers, but rather complex expressions of the AllMatches type. In fact, it may become necessary to use a different representation than AllMatches to describe the outcome of a score propagation.

On the other hand, the expressive power of the attached information solves some of the problems introduced in Section 5.22.6: This not only allows to add lists of tokens to a single tuple in the sequence encoding relation, but also store how these tokens are related. Further, we do not *multiply* items with the tokens, which helps avoid non-determinism (also discussed in Section 7.2).

If the variables bound by the `score` keyword shall remain numbers (which is handy, *e.g.*, for sorting), then it is required to calculate a score from the information attached in the score column. This defers the loss of information until the number is calculated. But even then, enclosing expressions will see the matches attached to the items.

7.4 Avoid locality

A general limit of this approach is the locality of score propagation: Each XQuery operator comes with some knowledge about how to propagate scores, and although pragmas can be used to "parametrise" this, there is no means an operator could know where its arguments came from.

This issue can be eased by making the Score type more expressive. The previous section already discussed the use of XML to achieve this. Instead of dealing with AllMatches, one may construct the expression tree in a bottom-up fashion, and store its XML representation (*i.e.*, a node surrogate for the root node of this representation) in the score column.

Thus, to evaluate an expression e with subexpressions $e_0, \ldots, e_n$, we evaluate the subexpressions which returns the respective results annotated with how they were calculated:

$$\langle e'_0 | s_0 \rangle, \ldots, \langle e'_n | s_n \rangle$$

From this, one may calculate $e' = f\ e_0\ \ldots\ e_n$, where f represents the function combining the subexpressions, and annotate it with

[1] `http://www.w3.org/TR/xquery-full-text/#tq-ft-XML-representation`

```
<application fun="a description of f">
  <argument>s_0</argument>
  ...
  <argument>s_n</argument>
</application>
```

To achieve this, it is not even required to have access to the parse tree: Any implementation of an operator or a function f is free to return a pair of its "natural" result, and some kind of identifier referring to the function itself. *E.g.*, with $\oplus$ being the primitive operation employed to calculate addition, one may implement

$$[\![e_1 + e_2]\!] = \langle x_1 \oplus x_2 | \gamma \rangle$$

where

$$\langle x_i | s_i \rangle = [\![e_i]\!]$$

$$\gamma = \left[\begin{array}{l} \texttt{<application fun="addition">} \\ \quad \texttt{<argument>}s_1\texttt{</argument>} \\ \quad \texttt{<argument>}s_2\texttt{</argument>} \\ \texttt{</application>} \end{array}\right]$$

Hence, although this does not introduce second order functionality (see Section 2.3.1.2), it may allow to work around the restrictions imposed by the PATHFINDER$^{\mathrm{FT}}$ architecture (see Section 2.3.1.3). Those, however, came for a good reason: With operators "knowing" their arguments, query rewriting becomes a black art. Following this road would blur PATHFINDER$^{\mathrm{FT}}$'s strict separation between a fuzzy Full Text language, and a somewhat clean database query language.

Chapter 8

Lessons Learnt

The query language XQuery is not easy to extend, and I would not consider it suitable as a general purpose programming language[1], because it lacks certain data structures, see Section 2.4.

The extension from XQuery to XQuery Full Text proposed by the W3C lacks reasoning about the semantics of the extension. While leaving many aspects abstract seems a good idea at first, as this allows for flexibility in implementations of XQuery Full Text, it makes the implementation of a suitable framework quite hard. Then again, some aspects are regulated to a superfluous level of detail, *e.g.*, weights *must* be in the range $[-1000, 1000]$ although their semantics is deliberately undefined. One could even argue about whether a (scored) Boolean is actually the right thing to return from the Full Text engine, see Section 2.3.3.

The implicit score propagation deemed necessary in Section 2.2.2.4 not only means to semantically parallelise two different computations, namely that of the value and that of the score. It also implies that *every time* the same syntactic construct is used, the same computations are performed for *both*, the value and the score. *I.e.*, the syntax determines two different computations instead of just one. This can be alleviated only by the excessive usage of pragmas, which reduces the syntactic benefit of implicit score propagation to absurdity.

Also, to achieve orthogonality consistent with scores, this parallelisation implies the algebra on scores (we used floats with arithmetics) to be compatible with all the algebras of types that the scores are paired with (*e.g.*, Boolean, axis-steps, *etc.*), which seems difficult to achieve.

A score can bear different meanings, and I doubt that restriction to one is sufficient in general. Why not allow different types of scores in one query, and handle them differently by explicitly programming how they should combine? The approach of explicitly dealing with scores (taken in the PF/Tijah project, see page 31) seems much more sane to me now: Calling the Full Text engine returns first-class citizens of the XQuery domain (scores and nodes), which can —and need to— be further processed by XQuery means.

On the other hand, this work proves that the Pathfinder compiler can be extended to deal with implicit score propagation easily. The extensions made line up very well with the overall

[1] Of course, I do not deny its completeness, but other languages, *e.g.*, brainfuck [22], are complete as well, although I hope that nobody uses them as general purpose programming languages.

PATHFINDER/MONETDB architecture. More important: All required changes are isolated to an early compilation phase, given that a Full Text engine is in place. Different compilation techniques for the Full Text language have been sketched in Section 5.21, which make the PATHFINDER^{FT} architecture usable even when the Full Text engine changes, and even when different interpretations of the Full Text language should be implemented.

Consider a user being expert enough to question the scoring model used. Of course, the implementation of the scoring model functions (see Section 5.23) should be accessible by that user. But this may not be enough: In principle, the user may request *any* interaction of scores and values thinkable (see Section 2.3.3). A consequence of this is that the complete calculation of values and scores should be under the user's control. This is probably achieved best by keeping PATHFINDER^{FT} a separate, early compilation phase (see Section 4.3), defined independent of the PATHFINDER compiler itself.

Within the XQUERY FULL TEXT ecosystem, PATHFINDER^{FT} may be a tool to further develop, and assess scoring models, and Full Text engines. Future work here includes to make PATHFINDER^{FT} cover more of the XQUERY language, adding a type system is probably the most pressing issue. Also, the support for the Full Text language is very limited currently.

If a clean, elegant, and sober language design is desired, it seems necessary to rethink XML IR at large, and outside of XQUERY FULL TEXT. Due to the shortcomings of XQUERY towards extensibility (see Section 2.4), this discussion should be led without regard to integration with XQUERY.

Bibliography

[1] Sihem Amer-Yahia et al., eds. *XQuery and XPath Full Text 1.0, W3C Candidate Recommendation*. URL: http://www.w3.org/TR/xquery-full-text/.

[2] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. 1st. Addison Wesley, 1999. ISBN: 020139829X. URL: http://www.worldcat.org/isbn/020139829X.

[3] Scott Boag et al., eds. URL: http://www.w3.org/TR/2007/REC-xquery-20070123/.

[4] Peter A. Boncz et al. "MonetDB/XQuery: a fast XQuery processor powered by a relational engine". In: *SIGMOD Conference*. Ed. by Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis. ACM, 2006, pp. 479–490. ISBN: 1-59593-256-9.

[5] Chavdar Botev, Sihem Amer-Yahia, and Jayavel Shanmugasundaram. "Expressiveness and Performance of Full-Text Search Languages". In: *EDBT*. Ed. by Yannis E. Ioannidis et al. Vol. 3896. Lecture Notes in Computer Science. Springer, 2006, pp. 349–367. ISBN: 3-540-32960-9.

[6] Richard T. Cox. *Algebra of Probable Inference*. Johns Hopkins University Press, 2002. ISBN: 080186982X. URL: http://yaroslav.hopto.org/papers/cox-algebra.pdf.

[7] Emiran Curtmola et al. "GalaTex: a conformant implementation of the XQuery full-text language". In: *WWW (Special interest tracks and posters)*. Ed. by Allan Ellis and Tatsuya Hagino. ACM, 2005, pp. 1024–1025. ISBN: 1-59593-051-5.

[8] Christian Grün et al. "XQuery Full Text Implementation in BaseX". In: *XSym*. Ed. by Zohra Bellahsene et al. Vol. 5679. Lecture Notes in Computer Science. Springer, 2009, pp. 114–128. ISBN: 978-3-642-03554-8.

[9] Torsten Grust. "Accelerating XPath location steps". In: *SIGMOD Conference*. Ed. by Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki. ACM, 2002, pp. 109–120. ISBN: 1-58113-497-5.

[10] Torsten Grust, Maurice van Keulen, and Jens Teubner. "Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps". In: *VLDB*. 2003, pp. 524–525.

[11] Torsten Grust and Jens Teubner. "Relational Algebra: Mother Tongue - XQuery: Fluent". In: *TDM*. Ed. by Vojkan Mihajlovic and Djoerd Hiemstra. CTIT Workshop Proceedings Series. Centre for Telematics and Information Technology (CTIT), University of Twente, Enschede, The Netherlands, 2004, pp. 9–16.

[12] Eric Hehner. "A probability perspective". In: *Formal Aspects of Computing* (2010). ISSN: 0934-5043. DOI: 10.1007/s00165-010-0157-0. URL: http://dx.doi.org/10.1007/s00165-010-0157-0.

[13] D. Hiemstra et al. “PFTijah: text search in an XML database system”. In: *Proceedings of the 2nd International Workshop on Open Source Information Retrieval (OSIR), Seattle, WA, USA*. Seattle, WA, USA: Ecole Nationale Supérieure des Mines de Saint-Etienne, 2006, pp. 12–17. ISBN: not assigned.

[14] D. Hiemstra et al. “Sound ranking algorithms for XML search”. In: *Proceedings of the 2nd SIGIR workshop on Focused Retrieval, Singapore*. Singapore: University of Otago, 2008, pp. 15–21.

[15] Johan A. List et al. “TIJAH: Embracing IR Methods in XML Databases”. In: *Inf. Retr.* 8.4 (2005), pp. 547–570.

[16] Vojkan Mihajlovic and Djoerd Hiemstra, eds. *First Twente Data Management Workshop (TDM 2004) on XML Databases and Information Retrieval, Enschede, The Netherlands, June 21, 2004*. CTIT Workshop Proceedings Series. Centre for Telematics and Information Technology (CTIT), University of Twente, Enschede, The Netherlands, 2004.

[17] Vojkan Mihajlovic et al. “Score region algebra: building a transparent XML-R database”. In: *CIKM*. Ed. by Otthein Herzog et al. ACM, 2005, pp. 12–19. ISBN: 1-59593-140-6.

[18] Henning Rode. “From document to entity retrieval : improving precision and performance of focused text search”. PhD thesis. Enschede, 2008. URL: `http://doc.utwente.nl/60765/`.

[19] Thomas Rölleke et al. “Modelling retrieval models in a probabilistic relational algebra with a new operator: the relational Bayes”. In: *VLDB J.* 17.1 (2008), pp. 5–37.

[20] Tanaka. *An Introduction to Fuzzy Logic for Practical Applications*. Springer-Verlag, 1997.

[21] Jens Teubner. “Pathfinder: XQuery Compilation Techniques for Relational Database Targets”. PhD thesis. 2006.

[22] *The Brainfuck Programming Lanuage*. URL: `http://www.muppetlabs.com/~breadbox/bf/`.

[23] *The Haskell Programming Lanuage*. URL: `http://www.haskell.org`.

[24] Andrew Trotman and Börkur Sigurbjörnsson. “Narrowed Extended XPath I (NEXI)”. In: *INEX*. Ed. by Norbert Fuhr et al. Vol. 3493. Lecture Notes in Computer Science. Springer, 2004, pp. 16–40. ISBN: 3-540-26166-4.