

A Parameterised Search System

Roberto Cornacchia and Arjen P. de Vries

CWI, Kruislaan 413, 1098SJ, Amsterdam, The Netherlands
{roberto,arjen}@cwi.nl

Abstract. This paper introduces the concept of a Parameterised Search System (PSS), which allows flexibility in user queries, and, more importantly, allows system engineers to easily define customised search strategies. Putting this idea into practise requires a carefully designed system architecture that supports a declarative abstraction language for the specification of search strategies. These specifications should stay as close as possible to the problem definition (i.e., the retrieval model to be used in the search application), abstracting away the details of the physical organisation of data and content. We show how extending an existing XML retrieval system with an abstraction mechanism based on array databases meets this requirement.

1 Introduction

For many years, information retrieval (IR) systems could be adequately described as software that assign an estimate of relevancy to a pair of document and query, each represented as a ‘bag-of-words’. The implementation of such search systems has been relatively straightforward, and most engineers code the retrieval model directly on top of an inverted file structure.

Trends in research and industry motivate however a reconsideration of the above characterisation of IR. First, modern retrieval systems have become more complex, as they exploit far more than ‘just’ the text. For example, the ranking function combines query and document text with other types of evidence, derived from, e.g., document markup, link structure, or various types of ‘context information’. Also, work tasks supported by search have become diverse. Within organisations, *enterprise search* refers to intranet search, but also search over collections of e-mail, finding expertise, etc. [1]. People use web search indeed for the goal of ‘finding information about’, but also to book a hotel, find a job, hunt for a house, just to name a few. Companies are targeting these niche markets with specialised search engines (known as *vertical search*).

Today, the development of such specialised applications is the job of information retrieval specialists. We expect however that, very soon, *any* software developer should be able to develop applications involving search. Actually, Hawking has stated that ‘*an obvious reason for poor enterprise search is that a high performing text retrieval algorithm developed in the laboratory cannot be applied without extensive engineering to the enterprise search problem, because of the complexity of typical enterprise information spaces*’ [1]. Simplifying the process

of tailoring search to a specific work task and user context should therefore be an important goal of IR research!

This paper proposes that the engineering of search systems may proceed analogous to the development of office automation applications using relational database management systems – define the ‘universe of discourse’; design a conceptual schema; express the user application in terms of this schema; and, design the user interface. So, software developers of a search application should have access to a high-level declarative language to specify collection resources *and retrieval model*. The search engine can then be parameterised for optimal effectiveness: adapted to the work task and user context, optimised for specific types of content in the collection, and specialised to exploit domain knowledge.

1.1 Anatomy of a Parameterised Search System

We refer to this new generation of information retrieval systems as *parameterised search engines* (PSSs). Fig. 1 illustrates how a PSS differs from the traditional search engine, in the so-called *abstraction language*. Its main purpose is to decouple search strategies from algorithms and data structures, bringing what the database field calls data independence to IR systems. This abstraction language should enable the search system developer to specify search strategies declaratively, ideally without any consideration of the physical representation of document structure and content.

Compare this to current practise using IR software like Lucene, Lemur, or Terrier. These systems provide a variety of well-known ranking functions, implemented on top of the physical document representation. The modular design of the IR toolkit allows application developers to select one of these ranking functions suited for their search problem. Design decisions about how to rank, e.g., weighting various types of documents differently with respect to their expected contribution to relevancy, will however be part of the application code.

Summarising, a PSS provides a declarative IR language. Search application developers specify the desired search strategy for a specific user and task context in this language. The system translates expressions in this language into operations on its internal data structures to perform the actual retrieval.

1.2 Approach, Contributions and Outline

This raises many open questions, the obvious one of course what the abstraction language could look like. The main contribution of this work is to demonstrate feasibility of the idea of a PSS through a (preliminary) prototype implementation. We propose a very specific instantiation of a PSS, by extending an existing XML IR system (PF/Tijah) with a formal language for the specification of retrieval models (*Matrix Framework for IR*). Another contribution is to operationalise this theoretical framework for IR on an array database system (SRAM), such that it can be deployed in practical search system implementation.

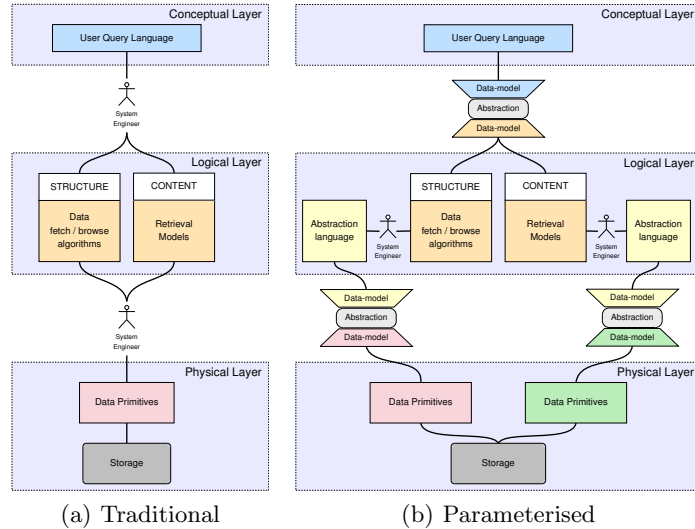


Fig. 1. Comparison of Search System architectures

The remainder is organised as follows. Section 2 describes the layered architecture of the XML retrieval system PF/Tijah. Section 3 details how we turn PF/Tijah into a PSS named *Spiegle*. We introduce its two main ingredients, the Matrix Framework for IR formalism and the array data-model abstraction of SRAM, and explain how these are integrated. Section 4 provides implementation details. Related research is discussed in Section 5, before we conclude and outline future work in Section 6.

2 Querying content and structure in PF/Tijah

A preliminary requirement of implementing a PSS is that it allows to express search strategies that refer to structure *and* content. *Spiegle* meets this requirement by building upon PF/Tijah [2], a flexible environment for setting up XML search systems.

PF/Tijah integrates XQuery and NEXI: XQuery to query and transform the *structure* of XML documents, NEXI (Narrowed Extended XPath) [3] to rank XML elements by their *content*. The resulting query language provides a powerful way to customise (mainly) the structural aspects of a retrieval strategy. Its layered system architecture, depicted in Fig. 2(a), uses the PathFinder (PF) XQuery system [4] to query by structure, as well as to construct the preferred result presentation. PathFinder translates the XQuery (non-ranked) part of a query into a relational query plan, independently from the ranking component of the system.

The Tijah XML retrieval system provides the XML IR support, by processing the NEXI expressions in a query. NEXI is a subset of XPath (it allows only descendant and self axis steps) that allows an additional `about()` clause, which

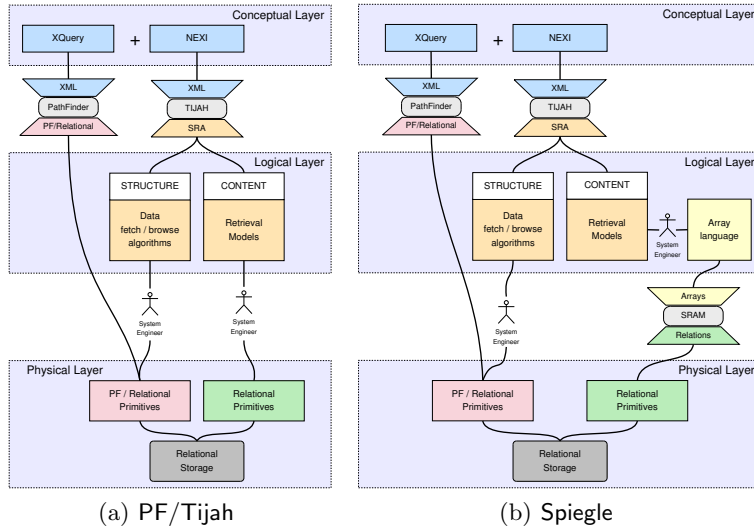


Fig. 2. Architectures of PF/Tijah and Spiegle

ranks the selected XML elements by their *content*. The following example retrieves, from a database of scientific publications, *The title of documents written by John Smith, where the abstract is about “IR DB integration”*:

```
let $c := doc("papers.xml")//DOC[author = "John Smith"]
let $q := "//*[text[about(../Abstract, IR DB integration)]];";
for $res in tijah-query($c, $q)
return $res/title/text()
```

PF/Tijah translates the NEXI expression $\$q$ into a logical query plan consisting of SRA (Score Region Algebra) operators [5]. The SRA algebra includes operators that perform the following tasks: (i) selection of the XML elements that are to be ranked; (ii) a parameterised *element score computation*, implementing the `about()` clause according to the desired retrieval model; (iii) a parameterised *element score combination*, i.e. compute the score of **AND** and **OR** combinations of `about()` clauses; (iv) a parameterised *element score propagation*, needed when scores need to be propagated to a common ancestor before being combined. As depicted in Fig. 2(a), structured information retrieval queries involve operations that act on *structure* (like (i)) and on *content* (e.g. (ii)). Each SRA operator implementation is given in terms of relational database operators (by a system engineer). Several retrieval models are supported out-of-the-box, selected by an extra ‘options’ parameter to the `tijah-query()` function.

3 Spiegle: Turning an XML retrieval system into a PSS

While PF/Tijah provides a powerful query language to embed search functionality in data processing, it does *not* support customisation of the retrieval model. Although advanced users may in principle modify the pre-defined mapping of

SRA operators to relational query plans to implement new ranking functions, doing so is far from trivial. In other words, PF/Tijah supports the customisation of the structural aspects of various search strategies, but it is inflexible with respect to modifying the *content* aspects.

Spiegle overcomes this limitation in two steps. First, it supports the declarative specification of retrieval models, by employing the Matrix Framework for IR. This way, the search system engineer may implement information retrieval models at a high level of abstraction. Second, SRAM translates the specified retrieval model automatically into a relational query plan.

3.1 The Matrix Framework for IR: a formalism for search strategies

The Matrix Framework for IR [6] (abbreviated to Matrix Framework) is a mathematical formalism that maps a wide spectrum of IR concepts to matrix spaces and matrix operations, providing a convenient logical abstraction that facilitates the design of IR systems. Indexing, retrieval, relevance feedback and evaluation measures are aspects described within the Matrix Framework. Also, it establishes a consistent notation for frequencies in event spaces, readily available as building blocks for IR applications in common matrix operation libraries.

We introduce only the part of the Matrix Framework that is focused on indexing and retrieval. First, we define three vectors, one for each of the dimensions used in the framework: $D = [w_d]_{N \times 1}$ for *documents*, $T = [w_t]_{S \times 1}$ for *terms* and $L = [w_l]_{R \times 1}$ for *locations*, with $1 \leq d \leq N$, $1 \leq t \leq S$ and $1 \leq l \leq R$. The quantities $w_d \geq 0$, $w_t \geq 0$ and $w_l \geq 0$ are the weight of document d , term t and location l , respectively. In the simplest case, these weights are boolean values that denote presence/absence in the collection. The term “location” is rather generic and covers concepts indicating document components of varying granularity, such as section, paragraph, position, or XML element.

The content and the structure of each data collection are entirely described by the two boolean matrices $LD_{L \times D}$ (location-document) and $LT_{L \times T}$ (location-term), whereas each query is described by a vector $Q_{T \times 1}$. As defined in (1), each value $LD(l, d)$ tells whether location l belongs to document d and each value $LT(l, t)$ encodes the occurrence of term t at location l of the collection. Finally, each query Q is described as a bit-vector of all collection-terms.

$$LD(l, d) = \begin{cases} 0, & \text{if } l \notin d \\ 1, & \text{if } l \in d \end{cases}, \quad LT(l, t) = \begin{cases} 0, & \text{if } t \notin l \\ 1, & \text{if } t \in l \end{cases}, \quad Q(t) = \begin{cases} 0, & \text{if } t \notin Q \\ 1, & \text{if } t \in Q \end{cases} \quad (1)$$

Standard IR statistics are defined as simple operations on matrices LD and LT :

DT_{nl}, DT – #term occurrences and term presence

$$DT_{nl} = LD^T \cdot LT, \quad DT = \min(DT_{nl}, \text{Ones}[|D| \times |T|]),$$

D_{nl}, T_{nl} – #per-document and #per-term locations

$$D_{nl} = LD^T \cdot L, \quad T_{nl} = LT^T \cdot L, \quad (2)$$

D_{nt}, T_{nd} – #terms per document and #documents per term

$$D_{nt} = DT \cdot T, \quad T_{nd} = DT^T \cdot D,$$

where $\text{Ones}[A \times B]$ defines a matrix of size $A \times B$, filled with 1’s.

A number of standard IR frequencies are easily derived from the quantities defined above:

DT_f – within-document term frequency

$$DT_f(d, t) = \frac{DT_{nl}(d, t)}{D_{nl}(d)},$$

D_{tf}, D_{idf} – term frequency and inverse term frequency of a document

$$D_{tf}(d) = \frac{D_{nt}(d)}{|T|}, \quad D_{idf}(d) = -\log D_{tf}(d), \quad (3)$$

T_{df}, T_{idf} – document frequency and inverse document frequency of a term

$$T_{df}(t) = \frac{T_{nd}(t)}{|D|}, \quad T_{idf}(t) = -\log T_{df}(t),$$

T_{nl} – collection frequency of a term

$$T_f(t) = \frac{T_{nl}(t)}{|L|}.$$

Finally, it is possible to define a number of retrieval models using the quantities and the frequencies described above. The *tf.idf* approach is specified as $RSV_{tf.idf} = DT_f \cdot \text{diag}(T_{idf}) \cdot Q$. Refer to [6] for many other retrieval models.

The Matrix Framework expresses indexing and retrieval tasks declaratively, starting from a simple matrix-based representation of collection and queries, and following a set of generic definitions that can be stored as a well-structured system library. This facilitates the engineering of extensible retrieval systems, as it allows a programming interface based on the array data-model (suited for IR). Physical implementation is delegated to another, dedicated layer.

3.2 SRAM: An array data-model implementation

The SRAM system is a prototype tool for mapping sparse arrays to relations and array operations to relational expressions. While SRAM syntax allows to express array operations on an element by element basis, the system translates such element-at-a-time operations to collection-oriented database queries, suited for (potentially more efficient) bulk processing.

The SRAM language defines operations over arrays declaratively in comprehension syntax [7], which allows to declare arrays by means of the following construct: $A = [\langle \text{array-cell value} \rangle \mid \langle \text{array axes} \rangle]$. The section $\langle \text{array axes} \rangle$ specifies the *shape* \mathcal{S}_A of result array A , i.e., its number of dimensions and their domain, denoted as: $i_0 < N_0, \dots, i_{n-1} < N_{n-1}$. The value of each dimension variable i_j ranges from 0 to $N_j - 1$. The section $\langle \text{array-cell value} \rangle$ assigns a value to each cell indexed by the index values enumerated by the $\langle \text{array axes} \rangle$ section. For example, the expression $Tidf = [-\log(Tdf(t)) \mid t < nTerms]$ defines a new array $Tidf[nTerms]$, where the value of each cell is computed by applying the function $-\log$ to corresponding cells of array Tdf . The explicit domain specification may be omitted when its range is clear from the context, e.g., the more convenient $[-\log(Tdf(t)) \mid t]$ is equivalent to $[-\log(Tdf(t)) \mid t < nTerms]$.

The language supports aggregations over any array dimension (**sum**, **prod**, **min**, **max**). For example, summing per document the term frequency counts (in a document-term matrix DT_f) of query terms (in binary vector Q) is expressed

as: $P = [\text{sum}([\text{DTf}(d,t) * Q(t) \mid t]) \mid d]$. The shape \mathcal{S}_P of array P is determined by the rightmost axis d .

Retrieving the *top-N* values is allowed for one-dimensional arrays only. The result consists of the *positions* (in order) of the values in the original array. So, $T = \text{topN}(P, N, \langle \text{ASC} | \text{DESC} \rangle)$ returns a *dense* array T with $\mathcal{S}_T = [N]$. The actual values can then be fetched by dereferencing the original array, $S = P(T)$.

The SRAM syntax allows definitions and assignments. Definitions are indicated by the symbol “=”, as in all the expressions seen above. They are expanded symbolically by a preprocessor at every occurrence in the array expression. Assignments, indicated by the symbol “:=”, translate to database expressions whose result is stored permanently in tables, named as indicated by the left part of the assignment expression: `<array name> := <comprehension expression>`.

3.3 How Spiegle applies the Array data-model

Fig. 2 shows how Spiegle inherits PF/Tijah’s logical layer, which provides an algebraic abstraction (SRA) between the conceptual and physical layers. The Spiegle architecture takes the point of logical abstraction further, by exploiting the clean distinction between *structure* and *content* operations offered by SRA. In the following, we first explain how the Matrix Framework can handle XML documents. Next, we see how to bootstrap, index and rank XML collections using the array data-model. The SRAM scripts shown are translated on-the-fly into database queries executed by the backend.

The Matrix Framework and XML data. The formalism of the Matrix Framework is not restricted to flat text search. Mapping the XML structure to the three dimensions location, document and term is all that is needed to apply the Matrix Framework to semi-structured collections. Each XML file can be seen as a collection of documents, where each distinct XML element represents a different one. The main difference from modelling plain text collections is that locations are now shared among different (nested) elements. Therefore, locations are defined as *term and XML tag* position in the file. Consider the following excerpt of the `papers.xml` collection of scientific publications, where location, document and term identifiers are annotated next to each piece of text for the reader’s convenience, with `l`, `d` and `t` prefixes respectively:

```
<PAPERS>[10,d0]
  <DOC>[11,d1]
    <text>[12,d2]
      <Abstract>[13,d3] IR[14,t0] as[15,t1] simple[16,t2] as[17,t1] cakes[18,t3]</Abstract>[19]
      ...
    </text>[140]
  </DOC>[141]
  <DOC>[142,d15] ..... </DOC>[1120] ...
</PAPERS>[130000000]
```

Notice that (i) the two DOC elements have different identifiers; (ii) the two `as` occurrences have the same term identifier; (iii) locations 3 to 9 belong to documents 0,1,2 and 3.

Bootstrapping XML collections. A Matrix Framework representation of a collection is obtained from vectors L , D and T and matrices LD and LT . For the example collection `papers.xml`, this corresponds to a SRAM script with the following definitions.

```
papers.ram :
// global information for collection "papers.xml"
nLocs=30000000 , nDocs=2000000 , nTerms=400000
L = [ 1 | l < nLocs ]
D = [ 1 | d < nDocs ]
T = [ 1 | t < nTerms ]
LD = [nLocs,nDocs], bool, sparse("0"), "LD_table"
LT = [nLocs,nTerms], bool, sparse("0"), "LT_table"
```

First, the length of each dimension is defined as `nLocs`, `nDocs`, `nTerms`. Then, vectors L , D and T are defined, using a constant weight 1. Finally, matrices LD and LT are declared as persistently stored in the database, as they are supposed to be the outcome of an earlier parsing and pre-processing phase on the XML file. For each persistent matrix, the dimensionality, the element type, the information on sparsity (LD and LT are both sparse with common value 0) and the name of the corresponding relational table are specified. Section 4 gives further details on the automatic creation of matrices LD and LT .

A script file similar to `papers.ram` is created automatically for each collection and it is included in every SRAM script that uses that particular collection. Notice that the script file above only contains SRAM *definitions* and no assignments (see Section 3.2), which result in simple in-memory declarations.

Array System Libraries. The uniform approach to IR tasks like indexing and retrieval that the Matrix Framework provides is easily operationalised by collecting the formulae presented in Section 3.1 in a system library composed by SRAM expressions (automatically made collection-dependent using a unique prefix for the collection's stored arrays). Fig. 3 shows an excerpt of such a library, that is used in the subsequent paragraphs about indexing and retrieval of XML collections. One can observe that the SRAM expressions are an almost direct transcription of mathematical formulae to ASCII characters, which demonstrates the intuitiveness of array comprehensions as an IR query language.

Indexing XML collections. The indexing phase creates statistical information about a given collection. As described in Section 3.1, this entails the computation of the matrices defined in (2) and (3). The definition of such matrices in SRAM syntax is given in Fig. 3, file `MF_Indexing.ram`. Notice that this file contains array *assignments*, that create persistent arrays in the database. The SRAM script for indexing the example `papers.xml` collection becomes straightforward: load the collection's global definitions, followed by the generic index-creation assignments:

```
#include "papers.ram" // global definitions for papers.xml
#include "MF_Indexing.ram" // create index arrays
```

Ranking XML collections. Recall the example query of Section 2, where NEXI expression `//text[about(//Abstract, IR DB integration)]` ranks the

```

#include "MF_DocContext.ram"
bm25(d,k1,b) = sum( [ w(d,t,k1,b) * Q(t) | t ] )
w(d,t,k1,b) = log( nDocs / Tdf(t) ) * (k1+1) * DTf(d,t)
              / ( DTf(d,t) + k1 * (1 - b + b * Dnl(d) / avgDnl) )

langmod(d,a) = sum( [ lp(d,t,a) * Q(t) | t ] )
lp(d,t,a)   = log( a*DTf(d,t) + (1-a)*Tf(t) )

MF_RetrievalModels.ram

#include "LinearAlgebra.ram"
DTnl := mxMult(mxTrnsp(LT), LD)
Dnl  := mvMult(mxTrnsp(LD), L)
Tnl  := mvMult(mxTrnsp(LT), L)
DT   := [ min(DTnl(d,t),1) | d,t ]
DTf  := [ DTnl(d,t)/Dnl(d) | d,t ]
Tf   := [ Tnl(t)/nLocs | t ]

MF_Indexing.ram

DTnl = [ DTnl(d,t) * DX(d) | d,t ]
Dnl  = [ Dnl(d)    * DX(d) | d ]
DT   = [ DT(d,t)  * DX(d) | d,t ]
DTf  = [ DTf(d,t) * DX(d) | d,t ]

MF_DocContext.ram

mxTrnsp(A) = [ A(j,i) | i,j ]
mxMult(A,B) = [ sum([ A(i,k) * B(k,j) | k ]) | i,j ]
mvMult(A,V) = [ sum([ A(i,j) * V(j) | j ]) | i ]

LinearAlgebra.ram

```

Fig. 3. SRAM libraries (excerpts)

“text” elements that contain an “Abstract” element about “IR DB integration”. This structured IR part of the query is translated to SRA algebra. Selection of text and their containing Abstract elements is performed by *structure* operations implemented using PathFinder primitives. We call the resulting node-set the *ranking-context*, represented in the Matrix Framework by a binary vector *DX*.

Ranking the Abstract elements of the *ranking-context* is performed by a *content* SRA operation, implemented in Spiegle by a function with signature:

```

Function rank(collection, rankingContext, queryTerms, N,
              retrievalModel, param1, param2, ...) := rankedContext

```

This function turns the *ranking-context* of a given query into a top-N *ranked-context* against the specified query terms, by applying the desired retrieval model. Its body executes a SRAM script, customised at each call with the value of the current function parameters. The script of the example NEXI query above, with parameters *collection*=“papers”, *retrievalModel*=“langmod” and *N*=20, corresponds to:

```

// global definitions for collection papers.xml
#include "papers.ram"
// ranking-context and query terms
DX = [nDocs], bool, sparse("0"), "DX_table"
Q   = [nTerms], bool, sparse("0"), "Q_table"
// include retrieval model definitions
#include "MF_RetrievalModels.ram"
// retrieve top N documents using "langmod" (Language Modelling)
S = [ langmod(param1,param2,...) | d ]
D20 := topN( [S(d) | d<20], DESC )
S20 := S(D20)

```

First, global definitions for the collection *papers.xml* are loaded. Then, the *ranking-context* and the query terms are declared as persistent arrays, previously

stored in the database by the system. Definitions of the available retrieval models are loaded (file `MF_RetrievalModels.ram`) and the selected one used to rank documents. Finally, the top 20 document identifiers and scores are computed and used together as the *ranked-context* returned by the SRA operator.

Fig. 3, file `MF_RetrievalModels.ram` shows an excerpt of the available retrieval model expressions. These get customised to the current querying scenario by declaring the two arrays DX (*ranking-context*) and Q (*query-terms*) before each query is executed. The first line includes library file `MF_DocContext.ram`. The included script limits the document axes of the index arrays to the current *ranking-context* by multiplying the document axes with the binary vector representing the *ranking-context* (the result of this multiplication corresponds to precisely those portions of index arrays that contain information about the node-set to be ranked).

4 Implementation details

Spiegle uses PathFinder’s efficient ‘document shredder’ to turn any XML data file into database relations (also from remote sources). PF/Tijah extends the standard PathFinder data-model by indexing text words (terms) in addition to XML nodes. XML elements are encoded as *regions*. Regions are stored as tuples $\langle \text{start}, \text{end}, \text{id}, \text{type} \rangle$, where *id* is the identifier associated with the XML element (a node tag or a term) and *type* can be either `node` or `term`. The *start* and *end* positions identify the text region in the original text file (each term is a text region of length 1). An additional relation is created for efficient term access (representing an inverted file structure). A more detailed description of PathFinder and PF/Tijah indexing schemes can be found in [4,2].

To use the Matrix Framework as described above, matrices LD and LT should be prepared during this indexing phase. Clearly, all required information is available in the tables created by PF/Tijah. Positions delimited by *start* and *end* values become “locations”, whereas the combination of *id* and *type* values become the document and term identifiers for matrices LD and LT .

Many storage schemes have been proposed for sparse multi-dimensional arrays, with strong emphasis on the special case of two-dimensional arrays [8]. SRAM represents an n -dimensional sparse array A as relation R_A , with ε_A denoting its default value. Each tuple encodes a vector $\mathbf{i} = (i_0, \dots, i_{n-1})$ of index values and a cell value $A(\mathbf{i})$; only those cells for which $A(\mathbf{i}) \neq \varepsilon_A$ are stored:

$$A \mapsto R_A(i_0, \dots, i_{n-1}, v) = \{(i_0, \dots, i_{n-1}, A(\mathbf{i})) \mid A(\mathbf{i}) \neq \varepsilon_A\}.$$

This storage scheme naturally extends to dense arrays, for which all tuples are stored physically. Data access patterns are optimised by creating standard relational indexing structures on top of such relations, or by explicit tuple clustering/sorting (the index columns form the relation’s primary key).

The life-cycle of array queries through the SRAM architecture can be summarised by the following sequence of transformations: Array comprehension \mapsto Array algebra \mapsto Relational algebra \mapsto Relational plan. The first translation step

generates an array-algebra tree, which represents the sequence of operations performed on the stored arrays (reshaping, selection, aggregation, or function application). A cost-based optimiser normalises and rewrites the array-algebra tree. No physical details are involved in the process yet: only arrays' size and density are taken into account. The next step maps the array data-model to the relational data-model, by means of translation rules that take into account the storage details: although common values of a sparse arrays are not stored, they may affect the final result. Standard relational optimisation techniques finalise the relational expression. The last step of the translation process maps relational algebra expressions to physical query plans for the database engine. Data access paths and algorithms are chosen for the physical implementation of relational operators in the query tree. For more in-depth details about the array-algebra and the mapping and optimisation rules used in the translation process, see [9].

5 Related work

This work addresses some of the main issues about IR&DB integration [10,11]. We are not aware of previous work that provides a declarative language for the definition of the retrieval model as part of an XML retrieval system. However, Inquiry's [12] (now Indri) query language can be considered to provide some early version of a parameterised search system, and it has often been used for precisely this flexibility (for example, in cross-language IR).

Already ten years ago, Fuhr argued in favour of data independence in IR [13], pointing out how this would reduce problems in plain text search with noun phrase search and treatment of compound words, and (semi-)structured data types to capture attributes like author, journal title or publication year.

De Vries defined the notion of 'content independence' [14] to refer to the decoupling between search strategies and content representation. His definition has been refined by Mihajlovic into the two related concepts 'retrieval model independence' and 'content description independence' [5]. Wen et al. use 'media independence' for a similar separation of concerns as Mihajlovic's content description independence [15]. Yet, none of these authors has proposed a declarative language that actually achieves the goal of separating the retrieval model from its actual implementation.

6 Conclusions and Future Work

This paper has argued that modern IR application requirements force us to reconsider the design characteristics of search systems. We promote an innovation in the search system engineering process, by introducing more flexibility in the IR system's architecture. The increased flexibility aims to reduce the effort of adapting search functionalities to work task and user context.

We defined the architectural requirements of so-called *Parameterised Search Systems*: (i) a layered architecture that allows structural *and* content information to be exploited for the search task; (ii) a convenient abstraction from the

physical details that discloses the retrieval engine's capabilities to the unique needs of each particular combination of collection characteristics, user preferences, and search strategies.

We indicated the architecture of the PF/Tijah XML IR system as a possible foundation to build upon, and the *Matrix Framework* for IR as a very well-suited abstraction to express retrieval strategies. Finally, we showed how *SRAM* can operationalise the *Matrix Framework* on top of a database system. The result of this effort is *Spiegle*: the first prototype of parameterised XML search system.

Future work includes further exploiting the array abstraction for the implementation of structure operations. This will allow for simpler inclusion of structural information in the retrieval strategy and better opportunities for optimising the final relational query plan. Early results of [9] have demonstrated that *SRAM*'s multi-stage query translation process can give excellent run-time performance on a large collection of web-data, given an efficient back-end system. Integrating this new back-end into the *Spiegle* architecture is on-going work.

References

1. Hawking, D.: Challenges in enterprise search. In: Proc. ADC. (2004) 15–26
2. Hiemstra, D., Rode, H., van Os, R., Flokstra, J.: PFTijah: text search in an XML database system. In: Proc. OSIR. (2006)
3. R.A.O'Keefe, Trotman, A.: The simplest query language that could possibly work. In: Proc. INEX. (2004)
4. Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery. In: Proc. SIGMOD. (2006) 479–490
5. Mihajlovic, V.: Score Region Algebra. A Flexible Framework for Structured Information Retrieval. PhD thesis, University of Twente (2006)
6. Rölleke, T., Tsikrika, T., Kazai, G.: A General Matrix Framework for Modelling Information Retrieval. *IP&M* **42**(1) (2005) 4–30
7. Buneman, P., Libkin, L., Suciu, D., Tannen, V., Wong, L.: Comprehension syntax. *SIGMOD Record* **23**(1) (1994) 87–96
8. Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H.: Templates for the solution of algebraic eigenvalue problems: a practical guide. SIAM (2000)
9. Cornacchia, R., Héman, S., Zukowski, M., de Vries, A., Boncz, P.: Flexible and efficient IR using Array Databases. Technical Report INS-E0701, CWI (2007)
10. Chaudhuri, S., Ramakrishnan, R., Weikum, G.: Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In: Proc. CIDR, Asilomar, CA, USA (2005) 1–12
11. Amer-Yahia, S., Case, P., Rölleke, T., Shanmugasundaram, J., Weikum, G.: Report on the DB/IR Panel at Sigmod 2005. *SIGMOD Record* **34**(4) (2005) 71–74
12. Callan, J.P., Croft, W.B., Harding, S.M.: The INQUERY retrieval system. In: Proc. DEXA. (1992) 78–83
13. Fuhr, N.: Object-oriented and database concepts for the design of networked information retrieval systems. In: Proc. CIKM. (1996) 164–172
14. de Vries, A.: Content independence in multimedia databases. *JASIST* **52**(11) (2001) 954–960
15. Wen, J., Li, Q., Ma, W., Zhang, H.: A multi-paradigm querying approach for a generic multimedia database management system. *SIGMOD Record* **32**(1) (2003) 26–34