XML Retrieval

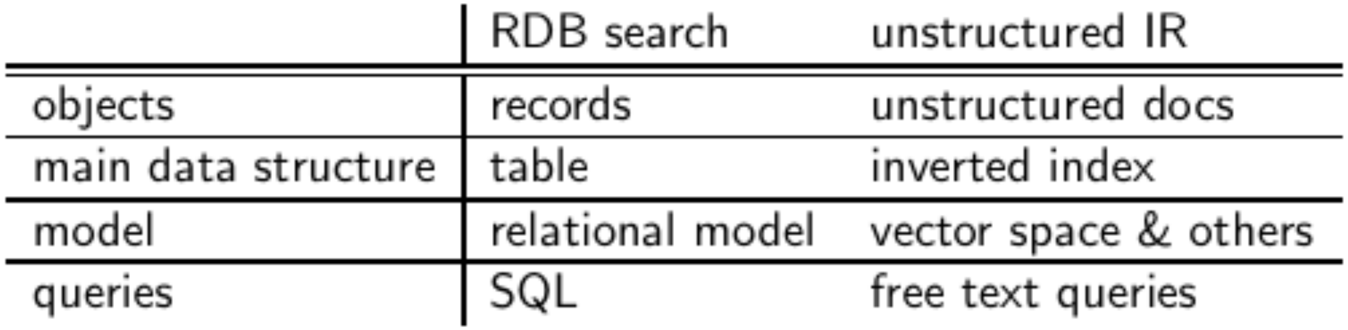
Overview

- Introduction
- Basic XML concepts
- Challenges in XML IR
- Vector space model for XML IR
- Evaluation of XML IR

IR and relational databases

IR systems are often contrasted with relational databases (RDB).

- Traditionally, IR systems retrieve information from *unstructured text* (“raw” text without markup).
- RDB systems are used for querying *relational data*: sets of records that have values for predefined attributes such as employee number, title and salary.



	RDB search	unstructured IR
objects	records	unstructured docs
main data structure	table	inverted index
model	relational model	vector space & others
queries	SQL	free text queries

Some structured data sources containing text are best modeled as structured documents rather than relational data (Structured retrieval).

Structured retrieval

Basic setting: queries are structured or unstructured; documents are structured.

Applications of structured retrieval

Digital libraries, patent databases, blogs, tagged text with entities like persons and locations (named entity tagging)

Example

- Digital libraries: *give me a full-length article on fast fourier transforms*
- Patents: *give me patens whose claims mention RSA public key encryption and that cite US patent 4,405,829*
- Entity-tagged text: *give me articles about sightseeing tours of the Vatican and the Coliseum*

Why RDB is not suitable in this case

Three main problems

❶ An unranked system (DB) would return a potentially large number of articles that mention the Vatican, the Coliseum and sightseeing tours without ranking them by relevance to query.

❷ Difficult for users to precisely state structural constraints – may not know which structured elements are supported by the system.

tours AND (COUNTRY: *Vatican* OR

LANDMARK: *Coliseum*)?

tours AND (STATE: *Vatican* OR BUILDING: *Coliseum*)?

❸ Users may be completely unfamiliar with structured search and advanced search interfaces or unwilling to use them.

Solution: adapt ranked retrieval to structured documents to address these problems.

Structured Retrieval

RDB search, Unstructured IR, Structured IR

	RDB search	unstructured retrieval	structured retrieval
objects	records	unstructured docs	trees with text at leaves
main data structure	table	inverted index	?
model	relational model	vector space & others	?
queries	SQL	free text queries	?

Standard for encoding structured documents: Extensible Markup Language (XML)

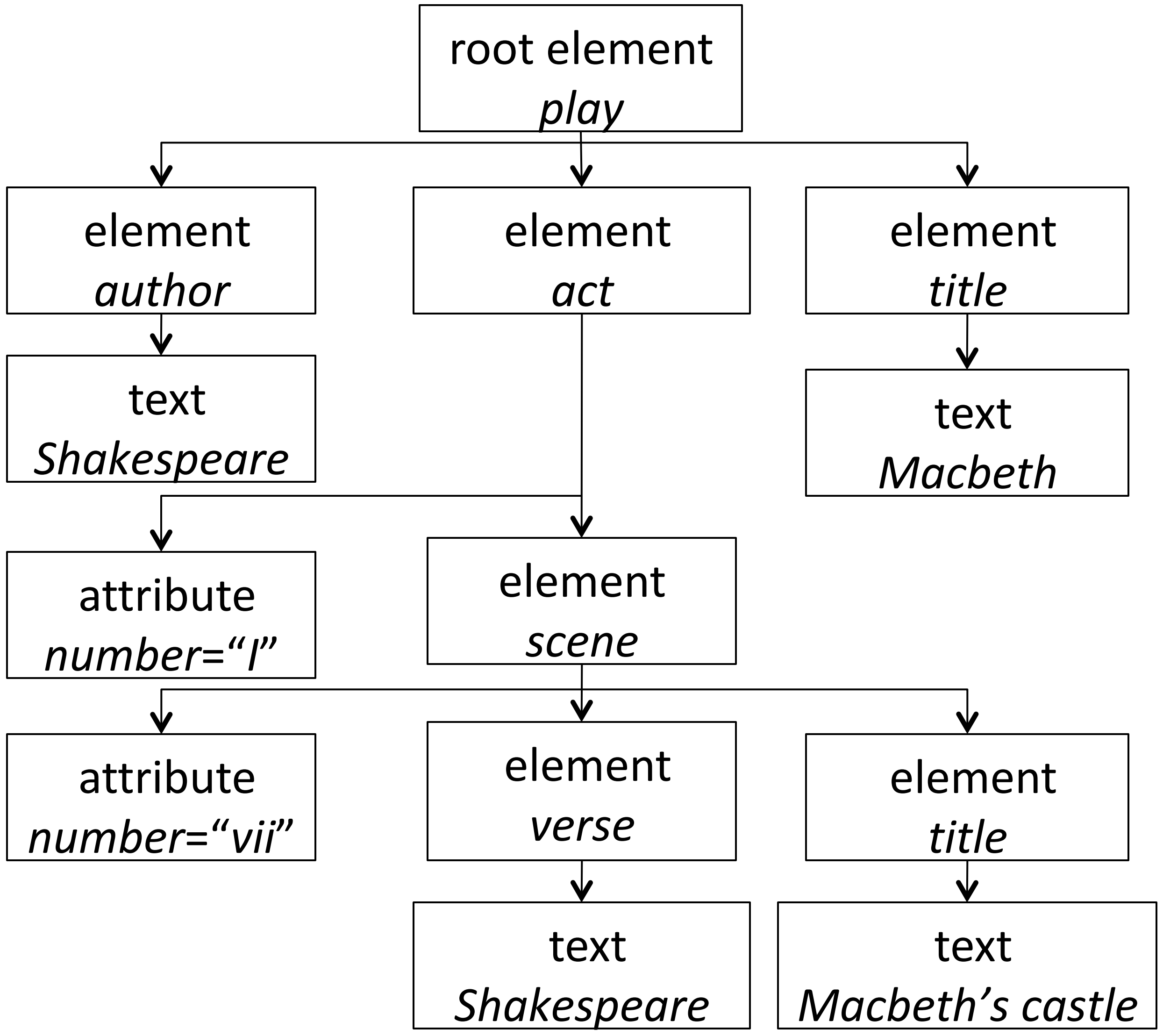
- structured IR → XML IR
- also applicable to other types of markup (HTML, SGML, …)

XML document

- Ordered, labeled tree
- Each node of the tree is an XML element, written with an opening and closing XML tag (e.g. <title…>, </title…>)
- An element can have one or more XML attributes (e.g. number)
- Attributes can have values (e.g. vii)
- Attributes can have child elements (e.g. title, verse)

```
<play>
<author>Shakespeare</author>
<title>Macbeth</title>
<act number="I">
<scene  number=""vii">
<title>Macbeth's castle</title>
<verse>Will I with wine
…</verse>
</scene>
</act>
</play>
```

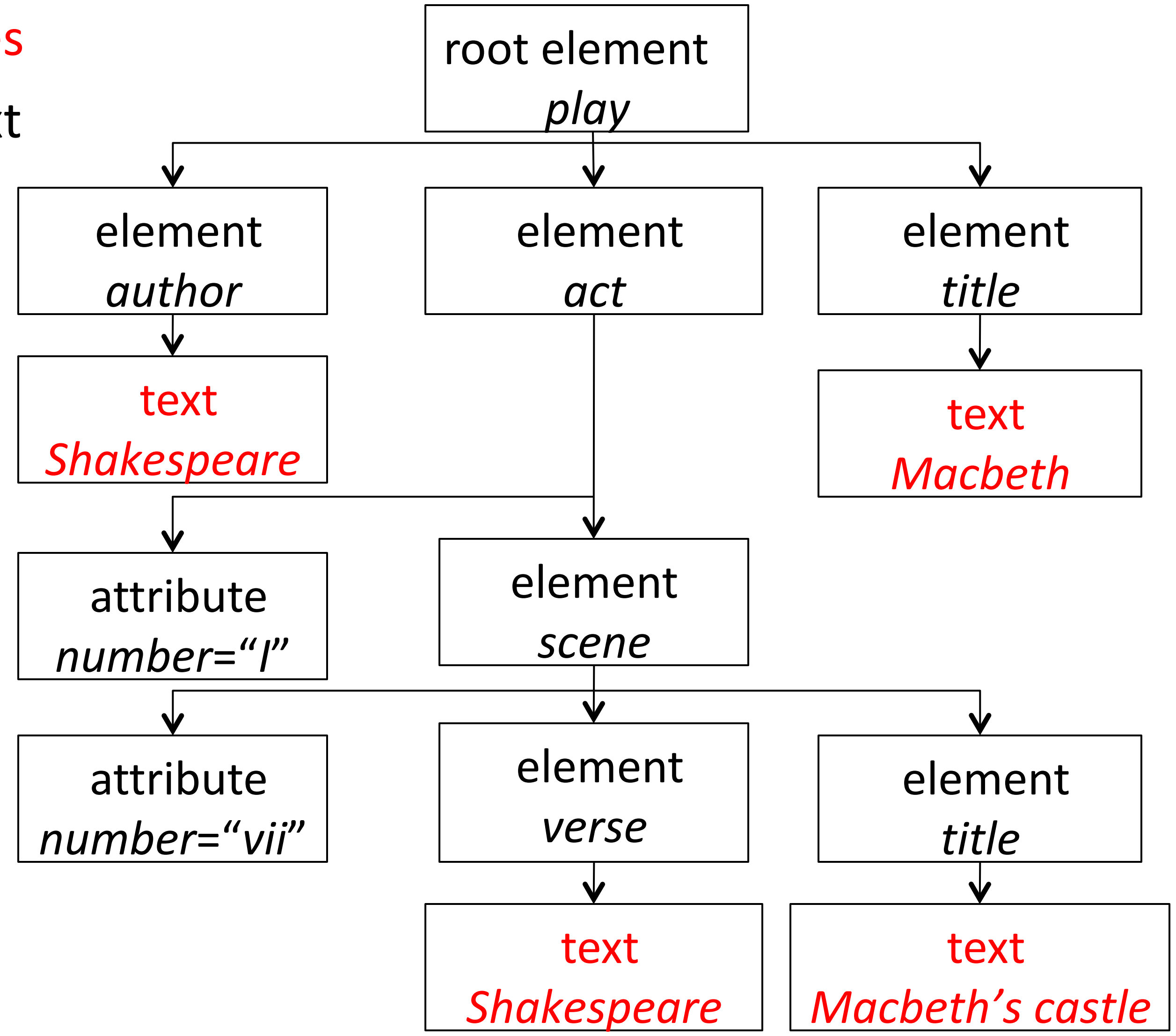
XML document



XML document

The leaf nodes

consist of text

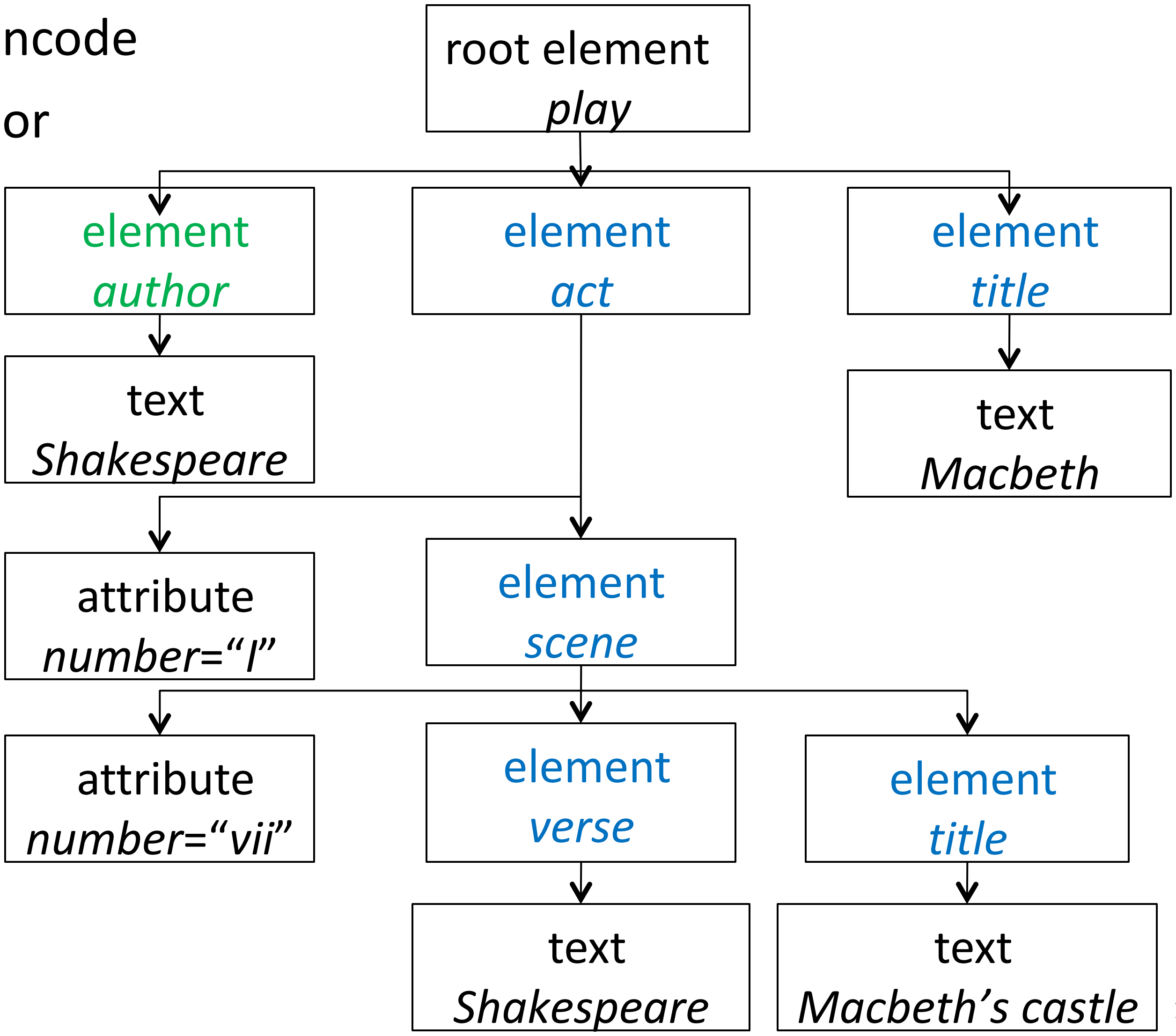


XML document

The internal nodes encode

document structure or

metadata functions



XML basics

- **XML Documents Object Model (XML DOM)**: standard for accessing and processing XML documents
    - The DOM represents elements, attributes and text within elements as nodes in a tree.
    - With a DOM API, we can process an XML documents by starting at the root element and then descending down the tree from parents to children.
- **XPath**: standard for enumerating path in an XML document collection.
    - We will also refer to paths as XML contexts or simply contexts
- **Schema**: puts constraints on the structure of allowable XML documents. E.g. a schema for Shakespeare's plays: scenes can occur as children of acts.
    - Two standards for schemas for XML documents are: XML DTD (document type definition) and XML Schema.

First challenge: document parts to retrieve

Structured or XML retrieval: users want us to return parts of documents (i.e., XML elements), not entire documents as IR systems usually do in unstructured retrieval.

Example

If we query Shakespeare's plays for *Macbeth's castle*, should we return the scene, the act or the entire play?

- In this case, the user is probably looking for the scene.
- However, an otherwise unspecified search for *Macbeth* should return the play of this name, not a subunit.

Solution: structured document retrieval principle

Structured document retrieval principle

Structured document retrieval principle

One criterion for selecting the most appropriate part of a document:

A system should always retrieve the most specific part of a document answering the query.

- Motivates a retrieval strategy that returns the smallest unit that contains the information sought, but does not go below this level.
- Hard to implement this principle algorithmically. E.g. query: title:*Macbeth* can match both the title of the tragedy, *Macbeth*, and the title of Act I, Scene vii, *Macbeth's castle*.
  - But in this case, the title of the tragedy (higher node) is preferred.
  - Difficult to decide which level of the tree satisfies the query.

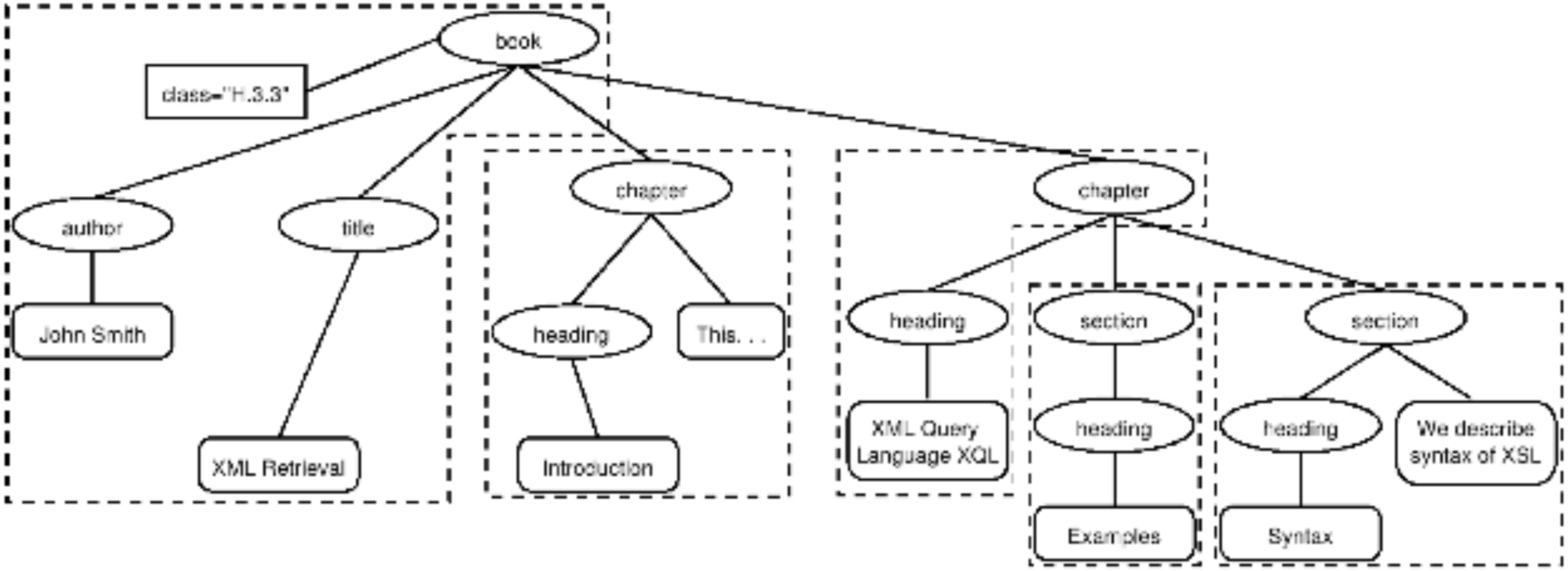
Second challenge: document parts to index

Central notion for indexing and ranking in IR: documents unit or **indexing unit**.

- In unstructured retrieval, usually straightforward: files on your desktop, email massages, web pages on the web etc.
- In structured retrieval, there are four main different approaches to defining the indexing unit
  1. non-overlapping pseudodocuments
  2. top down
  3. bottom up
  4. all

XML indexing unit: approach 1

Group nodes into non-overlapping pseudodocuments.



Indexing units: books, chapters, section, but without overlap.

Disadvantage: pseudodocuments may not make sense to the user because they are not coherent units.

XML indexing unit: approach 2

Top down (2-stage process):

- ❶ Start with one of the latest elements as the indexing unit, e.g. the book element in a collection of books
- ❷ Then, postprocess search results to find for each book the subelement that is the best hit.

This two-stage retrieval process often fails to return the best subelement because the relevance of a whole book is often not a good predictor of the relevance of small subelements within it.

XML indexing unit: approach 3

Bottom up:

Instead of retrieving large units and identifying subelements (top down), we can search all leaves, select the most relevant ones and then extend them to larger units in postprocessing.

Similar problem as top down: the relevance of a leaf element is often not a good predictor of the relevance of elements it is contained in.

XML indexing unit: approach 4

Index all elements: the least restrictive approach. Also problematic:

- Many XML elements are not meaningful search results, e.g., an ISBN number.
- Indexing all elements means that search results will be highly redundant.

Example

For the query *Macbeth's castle* we would return all of the *play*, *act*, *scene* and *title* elements on the path between the root node and *Macbeth's castle*. The leaf node would then occur 4 times in the result set: 1 directly and 3 as part of other elements.

We call elements that are contained within each other **nested elements**. Returning redundant nested elements in a list of returned hits is not very user-friendly.

Third challenge: nested elements

Because of the redundancy caused by the nested elements it is common to restrict the set of elements eligible for retrieval.

Restriction strategies include:

- discard all small elements
- discard all element types that users do not look at (working XML retrieval system logs)
- discard all element types that assessors generally do not judge to be relevant (if relevance assessments are available)
- only keep element types that a system designer or librarian has deemed to be useful search results

In most of these approaches, result sets will still contain nested elements.

Third challenge: nested elements

Further techniques:

- remove nested elements in a postprocessing step to reduce redundancy.
- collapse several nested elements in the results list and use highlighting of query terms to draw the user’s attention to the relevant passages.

Highlighting

- Gain 1: enables users to scan medium-sized elements (e.g., a section); thus, if the section and the paragraph both occur in the results list, it is sufficient to show the section.
- Gain 2: paragraphs are presented in-context (i.e., their embedding section). This context may be helpful in interpreting the paragraph.

Nested elements and term statistics

Further challenge related to nesting: we may need to distinguish different contexts of a term when we compute term statistics for ranking, in particular inverse document frequency (idf).

Example

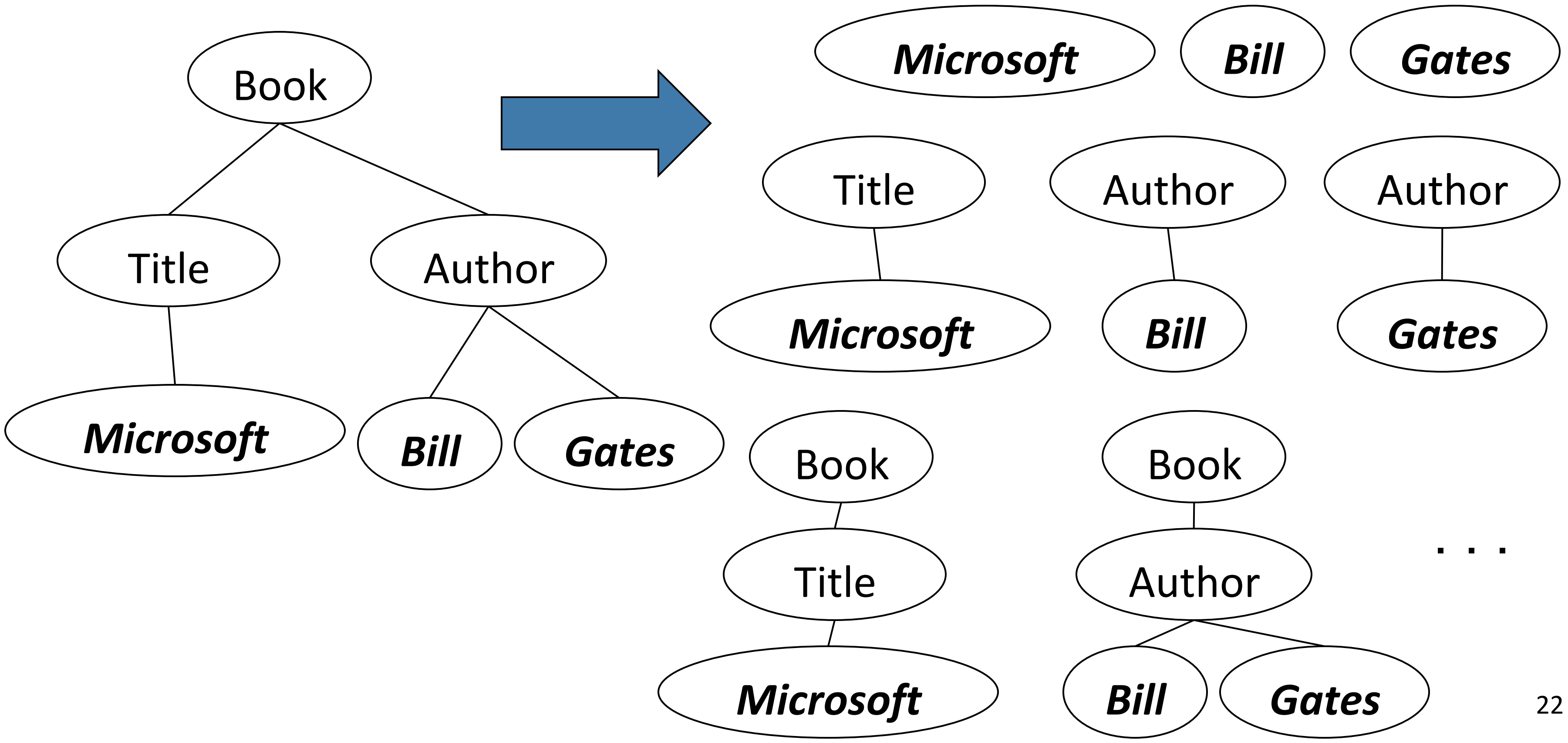
The term *Gates* under the node *author* is unrelated to an occurrence under a content node like *section* if used to refer to the plural of *gate*. It makes little sense to compute a single document frequency for *Gates* in this example.

Solution: compute idf for XML-context term pairs.

- sparse data problems (many XML-context pairs occur too rarely to reliably estimate df)
- compromise: consider the parent node x of the term and not the rest of the path from the root to x to distinguish contexts.

Main idea: lexicalized subtrees

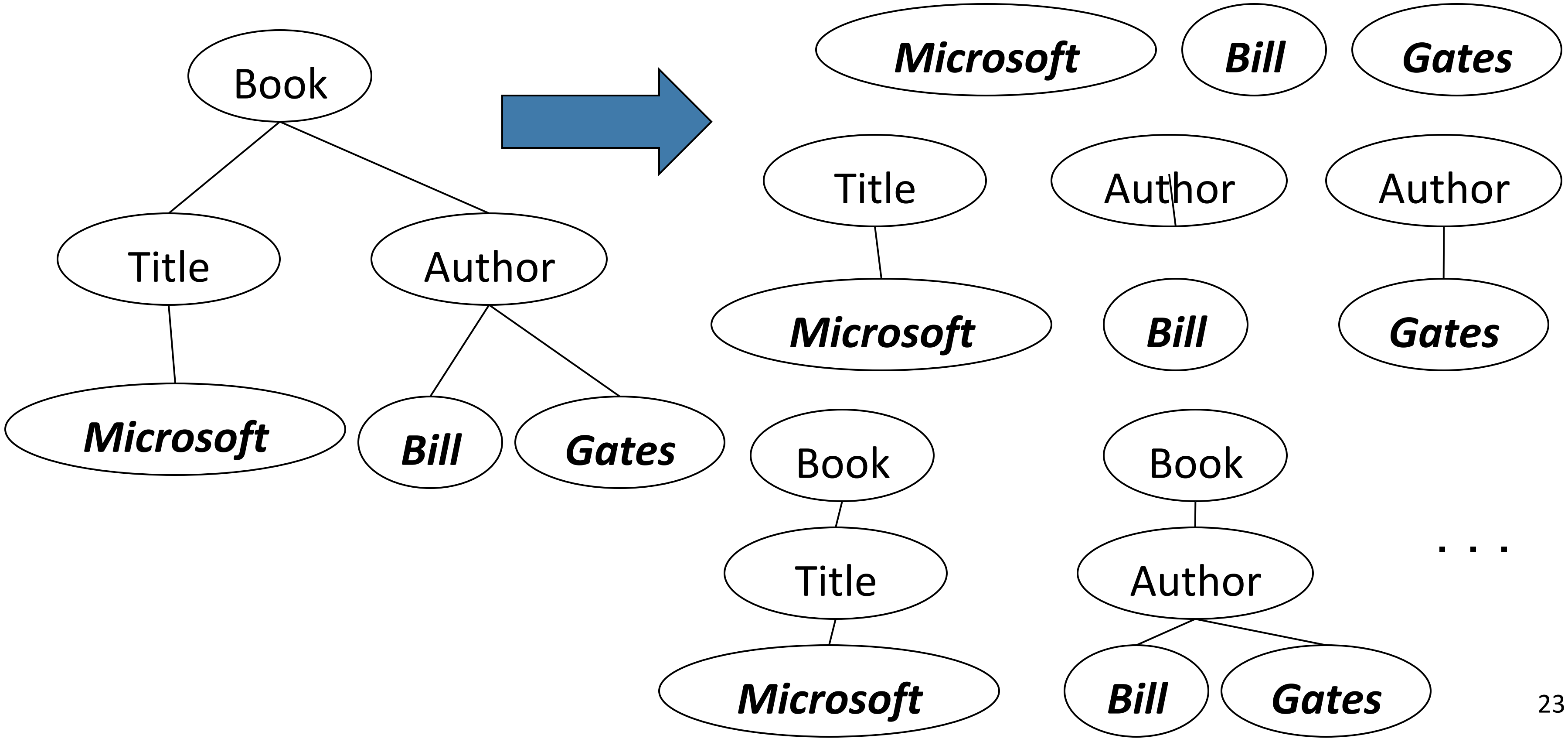
Aim: to have each dimension of the vector space encode a word together with its position within the XML tree.
How: Map XML documents to lexicalized subtrees.



Main idea: lexicalized subtrees

❶ Take each text node (leaf) and break it into multiple nodes, one for each word. E.g. split *Bill Gates* into *Bill* and *Gates*

❷ Define the dimensions of the vector space to be lexicalized subtrees of documents – subtrees that contain at least one vocabulary term.



Lexicalized subtrees

We can now represent queries and documents as vectors in this space of lexicalized subtrees and compute matches between them, e.g. using the vector space formalism.

Vector space formalism in unstructured VS. structured IR

The main difference is that the dimensions of vector space in unstructured retrieval are vocabulary terms whereas they are lexicalized subtrees in XML retrieval.

Structural term

There is a tradeoff between the dimensionality of the space and the accuracy of query results.

- If we restrict dimensions to vocabulary terms, then we have a standard vector space retrieval system that will retrieve many documents that do not match the structure of the query (e.g., *Gates* in the title as opposed to the author element).
- If we create a separate dimension for each lexicalized subtree occurring in the collection, the dimensionality of the space becomes too large.

Compromise: index all paths that end in a single vocabulary term, in other words all XML-context term pairs. We call such an XML-context term pair a structural term and denote it by <*c*, *t*>*:* a pair of XML-context *c* and vocabulary term *t*.

Context resemblance

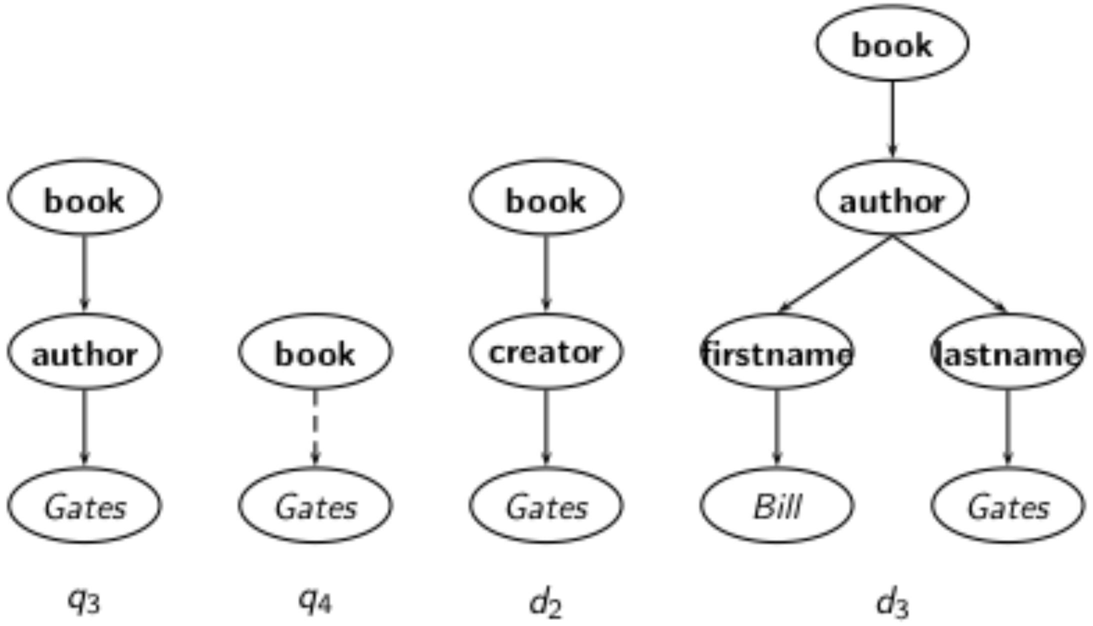
A simple measure of the similarity of a path c_q in a query and a path c_q in a document is the following *context resemblance* function CR:

$$\mathrm{CR}(c_q, c_d) = \begin{cases} \frac{1+|c_q|}{1+|c_d|} & \text{if } c_q \text{ matches } c_d \\ 0 & \text{if } c_q \text{ does not match } c_d \end{cases}$$

$|c_q|$ and $|c_d|$ are the number of nodes in the query path and document path, resp.

c_q matches c_d iff we can transform c_q into c_d by inserting additional nodes.

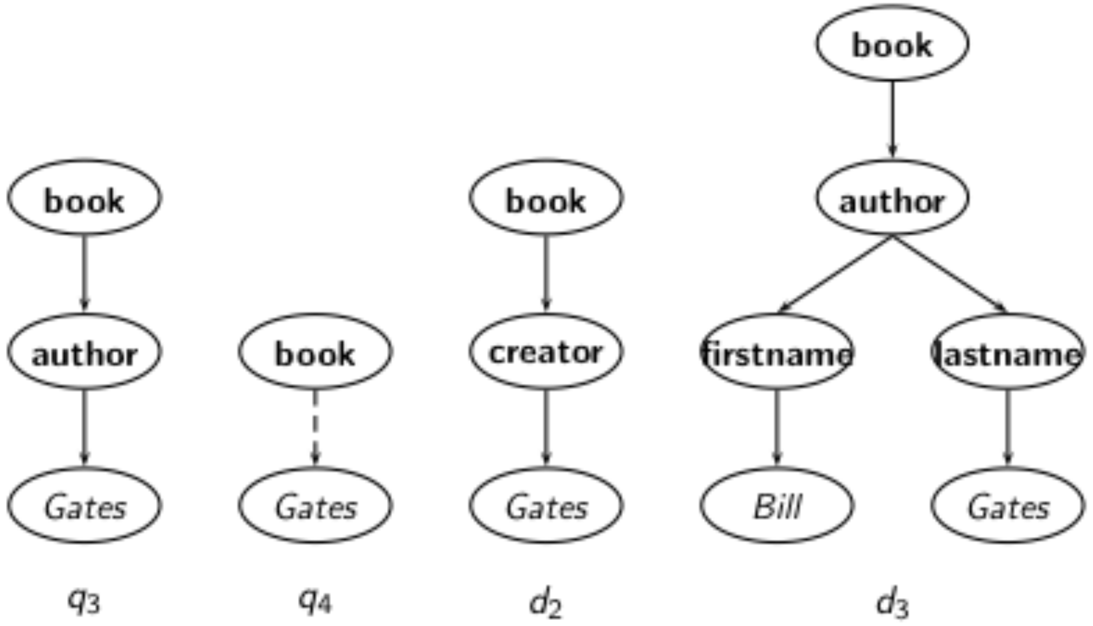
Context resemblance example



$$\mathrm{CR}(c_q, c_d) = \begin{cases} \frac{1+|c_q|}{1+|c_d|} & \text{if } c_q \text{ matches } c_d \\ 0 & \text{if } c_q \text{ does not match } c_d \end{cases}$$

CR(c_q, c_d) = 3/4 = 0.75. The value of CR(c_q, c_d) is 1.0 if *q* and *d* are identical.

Context resemblance example



$$\mathrm{CR}(c_q, c_d) = \begin{cases} \frac{1+|c_q|}{1+|c_d|} & \text{if } c_q \text{ matches } c_d \\ 0 & \text{if } c_q \text{ does not match } c_d \end{cases}$$

CR(c_q, c_d) = ? CR(c_q, c_d) = 3/5 = 0.6.

Document similarity measure

The final score for a document is computed as a variant of the cosine measure, which we call SIMNOMERGE.

SIMNOMERGE(q, d) =

$$\sum_{c_k \in B} \sum_{c_l \in B} \mathrm{CR}(c_k, c_l) \sum_{t \in V} \mathrm{weight}(q, t, c_k) \frac{\mathrm{weight}(d, t, c_l)}{\sqrt{\sum_{c \in B, t \in V} \mathrm{weight}^2(d, t, c)}}$$

- V is the vocabulary of non-structural terms
- B is the set of all XML contexts
- weight (q, t, c), weight(d, t, c) are the weights of term t in XML context c in query q and document d, resp. (standard weighting e.g. idf_t x $\mathrm{wf}_{t,d}$, where idf_t depends on which elements we use to compute df_t.)

SIMNOMERGE(q, d) is not a true cosine measure since its value can be larger than 1.0.

SimNoMerge algorithm

ScoreDocumentsWithSimNoMerge(*q*, *B*, *V*, *N*, *normalizer*)

```
 1  for $n \leftarrow 1$ to $N$
 2  do $score[n] \leftarrow 0$
 3  for each $\langle c_q, t \rangle \in q$
 4  do $w_q \leftarrow \textsc{Weight}(q, t, c_q)$
 5      for each $c \in B$
 6      do if $\textsc{Cr}(c_q, c) > 0$
 7            then $postings \leftarrow \textsc{GetPostings}(\langle c, t \rangle)$
 8                 for each $posting \in postings$
 9                 do $x \leftarrow \textsc{Cr}(c_q, c) * w_q * weight(posting)$
10                    $score[docID(posting)] += x$
11  for $n \leftarrow 1$ to $N$
12  do $score[n] \leftarrow score[n]/normalizer[n]$
13  return $score$
```

Recap

- Structured or XML IR: effort to port unstructured (standard) IR know-how onto a scenario that uses structured (DB-like) data
- Specialized applications (e.g. patents, digital libraries)
- A decade old, unsolved problem