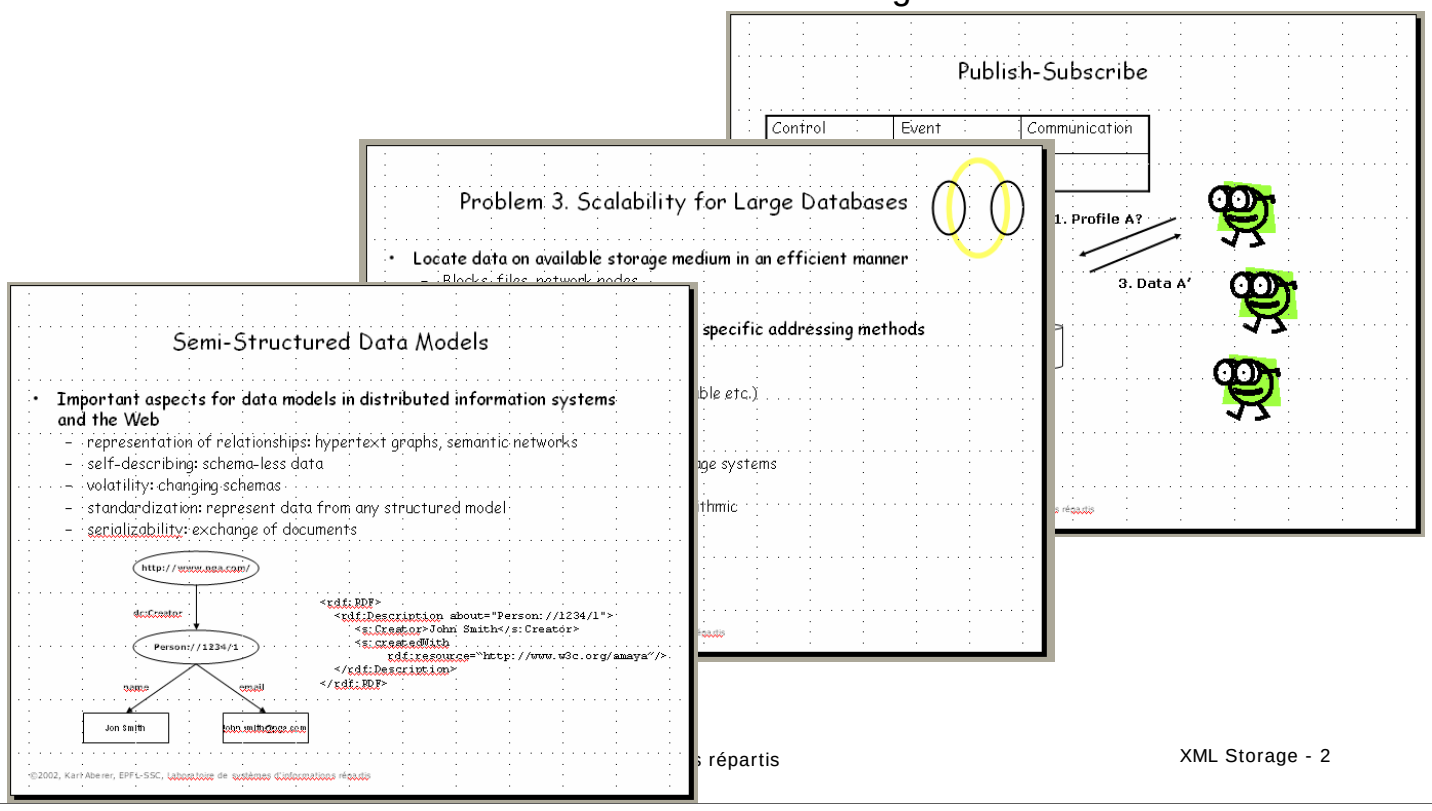
# Semistructured Data Management
# Part 1 – XML Storage and Filtering

# Today's Question

1. Relational XML Data Storage
2. XML Document Filtering




Having a data model such as XML leads to the problem of managing data represented in the data model. Therefore we will investigate in this part two different, exemplary aspects of XML data management: the storage of XML data relying on conventional relational database technology, and the filtering of XML documents against a large number of queries (profiles).

# 1. XML Data Storage

- Problem: persistent storage of (large sets of large) XML documents
- Goal: find a storage system that supports both efficient search and updates
- Alternatives for XML database systems
  - Reuse of existing technologies for persistent data management
    - Relational databases
    - Object-oriented databases
    - Text retrieval systems
  - Native XML data storage systems

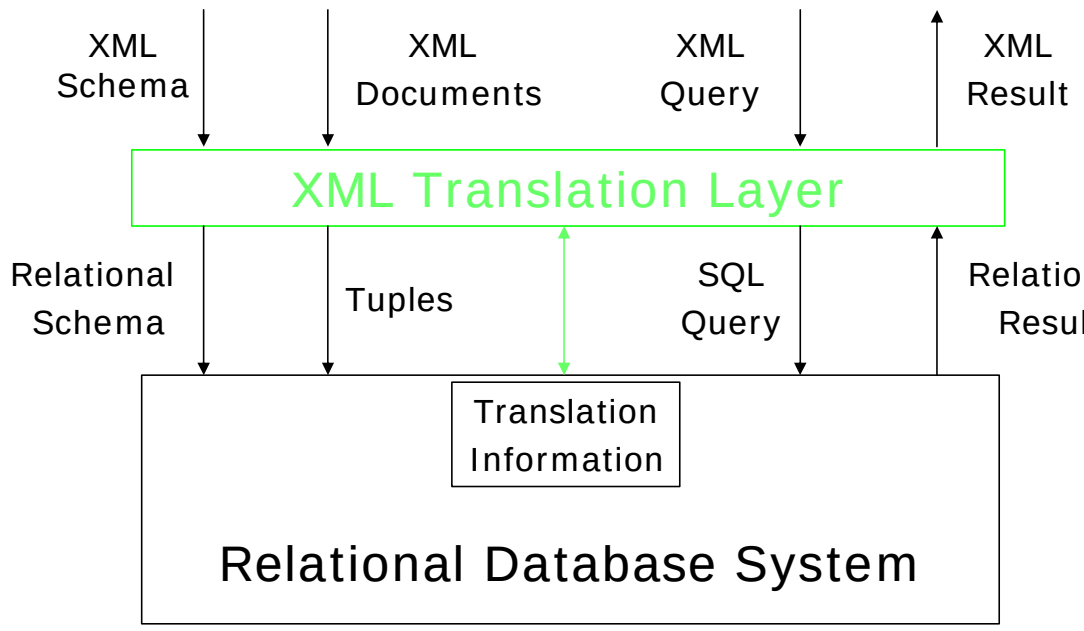



We can expect that much of the data will be represented/available in the future in XML. This is not so problematic for data that is generated from storage systems that keep this data in another format, such as relational data, but it is more problematic if the data is only available in XML, for example, message data that has been received from business partners as part of a message-based business process implementation. There exists two possibilities, of how an XML database system can be implemented:

1. Reusing existing technologies, e.g. relational DBMS. This has the advantage that one can rely on proven and robust technology.
2. Building special purpose (native) XML storage systems, this has the advantage that one can take better advantage of the specific characteristics of XML data.

## Relational XML Data Storage

- Advantage: reuse of existing robust and scalable technology
- Disadvantage: Data model mismatch
  - Hierarchical vs. relational data
  - Optional elements in XML schemas








At the current time the development of native XML storage systems has not reached a stable state, such that for practical purposes reuse of relational database technology is a valid alternative. Storing XML data in the relational data model poses challenging questions on mapping in between the relational and the XML data model.

The architecture of a relational XML store is straightforward. Essentially it requires a translation layer that can translate both schema information, queries and data instances back and forth between the XML and relational data model.

## What Do You Think ?

- How to store XML documents in a relational database ?

- What problems need to be addressed ?

# Edge Table Approach

**Store all edges of the XML document graph in one table**

```
1 <book>
    <booktitle> The Selfish Gene </booktitle>
2   <author id = “dawkins”>
3       <name>
          <firstname> Richard </firstname>
          <lastname> Dawkins </lastname>
        </name>
4       <address>
          <city> Timbuktu </city>
          <zip> 99999 </zip>
        </address>
      </author>
  </book>
```

**Edge table EDGE**

| source | ordinal | name | flag | target |
|---|---|---|---|---|
| 1 | 1 | booktitle | string | v1 |
| 1 | 2 | author | ref | 2 |
| 2 | 1 | name | ref | 3 |
| 2 | 2 | address | ref | 4 |
| 3 | 1 | firstname | string | v2 |
| 3 | 2 | lastname | string | v3 |
| 4 | 1 | city | string | v4 |
| 4 | 2 | zip | int | v5 |

**String table STR**

| id | value |
|---|---|
| v1 | The Self |
| v2 | Richard |
| v3 | Dawkins |
| v4 | Timbuktu |
| v6 | dawkins |

**Integer table INT**

| id | value |
|---|---|
| v5 | 99999 |

**Attribute table ATT**

| source | ordinal | attrname | flag | target |
|---|---|---|---|---|
| 1 | 2 | id | string | v6 |




One possibility is to view the XML document as a graph or more specifically as a tree, and store all the edges of the graph in a relational table. The contents of the document as well as the attributes can be stored in separate tables. One observes that for large documents and document collections these tables will become extremely large.

## Example: Processing of Path Queries

- The XPath query

```
//booktitle/author/name/firstname
```

- Requires evaluation of the following SQL Query

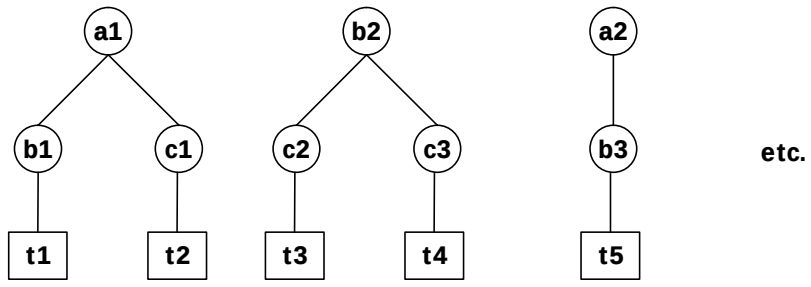
```
SELECT
     S.value
FROM
     EDGE E1, EDGE E2, EDGE E3, EDGE E4, STR S
WHERE
     E1.name="booktitle" AND
     E1.target=E2.source AND E2.name="author" AND
     E2.target=E3.source AND E3.name="name" AND
     E3.target=E4.source AND E4.name="firstname" AND
     E4.target=S.id
```




After mapping XML data into a relational representation, we want also to be able to process queries against the XML documents stored in the relational database. This requires that the XML queries are translated into SQL queries. This translation process is fairly simple for XPath. The problem is that the resulting queries are not necessarily very efficient to evaluate. One can observe that a simple XPath query with four location steps requires the evaluation of 4 self-joins on the Edge table. We know that joins are the most expensive part of query processing in relational databases, therefore this is problematic. In addition, repeatedly selections on potentially large edge table using the element name need to be performed.

## Step 1: Typed Element Tables

- Given three element types A, B, C, no DTD




| Id_A | Parent_A | Content_A |
|---|---|---|
| a1 | NULL | NULL |
| a2 | NULL | NULL |

| Id_B | Parent_B | Content_B |
|---|---|---|
| b1 | a1 | t1 |
| b2 | NULL | NULL |
| b3 | a2 | t5 |

| Id_C | Parent_C | Content_C |
|---|---|---|
| c1 | a1 | t2 |
| c2 | b2 | t3 |
| c3 | b2 | t4 |





A number of variations of the edge table representation have been proposed and evaluated. In the horizontally partitioned edge tables approach the single edge table is partitioned into different ones according to the element types, resulting in typed elements tables. This simplifies query processing insofar as no selections need to be made in order to select the elements of a specific type. When we know the DTD we can derive from it often the possible element types that can be reached in a navigation (location) step of a XPath query and thus simplify the processing of the query.

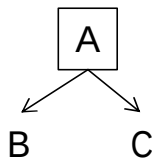
Another problem with the edge table approach is that the document order is lost. This can be dealt with, by using a node-oriented storage scheme, where each node of the XML document tree corresponds to a tuple in a node table. Each tuple can store all the relationships with other nodes, in particular also with sibling nodes (similarly as in the DOM model).

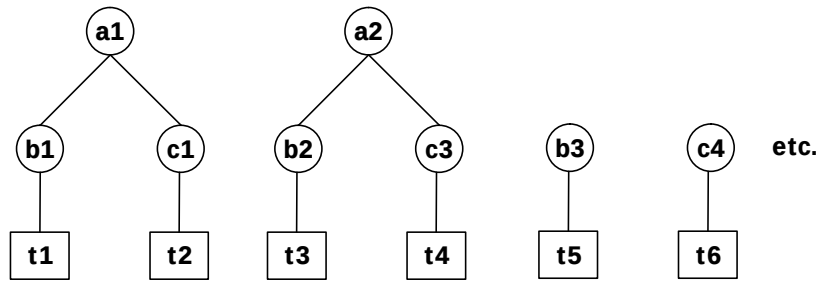
All of these approaches however do not fundamentally change the situation that a large number of joins are required to process queries and to reconstruct the XML data from the relational database. The problem (though also what may be considered an advantage) is that we make no use of additional knowledge on the structure of the documents, in particular of a document type definition, in case it is available. This is what we focus on in the following.

## Step 2: Exploiting a DTD (Inlining)

- Given three element types A, B, C, with DTD

<!ELEMENT A (B, C)>
<!ELEMENT B #PCDATA>
<!ELEMENT C #PCDATA>




```
CREATE TABLE A
(
  Id_A INTEGER,
  Content_B CHAR,
  Content_C CHAR
  PRIMARY KEY Id_A
)
```

| Id_A | Content_B | Content_C |
|---|---|---|
| a1 | t1 | t2 |
| a2 | t3 | t4 |
| _1 | t5 | NULL |
| _2 | NULL | t6 |



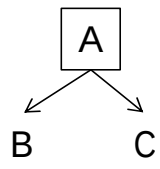

If we consider a DTD we may observe, that an element such as A (in the example given) contains only single occurrences of the elements B and C. A key idea will be, to use inlining for XML elements as a possible technique for optimizing relational representations of XML documents, provided the documents correspond to a DTD. Inlining is illustrated for the table that is used to store the elements of type A which inline B and C. Each attribute in this table corresponds to a path starting from the root element of the table A. Note that using such a mapping seems to be much more natural when one compares to ER modeling: there the elements contained in element A would be most likely modeled as attribute of A, and not as separate entity (1:1 relationship). The resulting relational schema would be similar to the one given in the example.

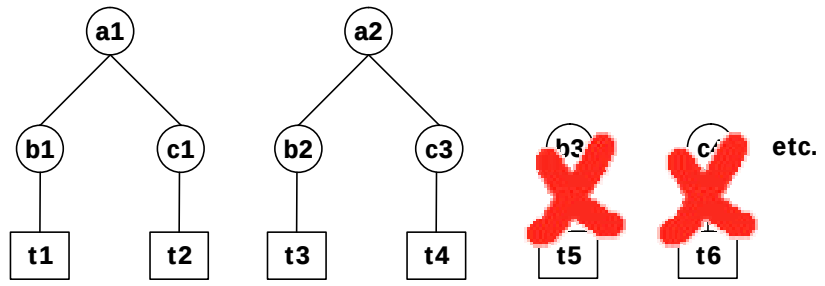
The question is how to determining, given a XML DTD, the proper relational schema that inlines other elements properly.

## Integrity Constraints with Inlining

- Given three element types A, B, C, with DTD

```
<!DOCTYPE A [
 <!ELEMENT A (B, C)>
 <!ELEMENT B #PCDATA>
 <!ELEMENT C #PCDATA>
]>
```




```
CREATE TABLE A
(
  Id_A INTEGER,
  Content_B CHAR NOT NULL,
  Content_C CHAR NOT NULL
  PRIMARY KEY Id_A
)
```

| Id_A | Content_B | Content_C |
|---|---|---|
| a1 | t1 | t2 |
| a2 | t3 | t4 |



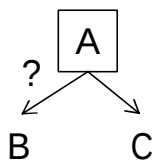


There is a subtle point to consider with inlining when taking into account the declaration of root elements in a DTD (as shown here for element A). With a root element document fragments that satsify only a part of the DTD are no longer possible. As a consequence existence constraints for elements can be formulated. Within the relational representation these result in NOT NULL constraints for elements that must exist necessarily, if the existence of the root element is taken into account. The reason why we will not further consider these constraints, is that for practical purposes of applicability, an XML storage system should always be capable of storing document fragments, as they might occur during processing or editing of XML documents. Then the relational constraints, as shown in this example would be violated, i.e. these fragments could not be stored.

## Step 3: Optional Elements

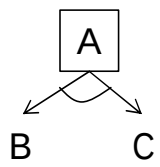
- Given three element types A, B, C, with DTD

<!ELEMENT A (B?, C)>
<!ELEMENT B #PCDATA>
<!ELEMENT C #PCDATA>




```
CREATE TABLE A
(
  Id_A INTEGER,
  Content_B CHAR,
  Content_C CHAR
  PRIMARY KEY Id_A
  )
```

<!ELEMENT A (B | C)>
<!ELEMENT B #PCDATA>
<!ELEMENT C #PCDATA>




**Optional elements have no impact on structure (just on constraints) !**

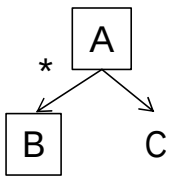


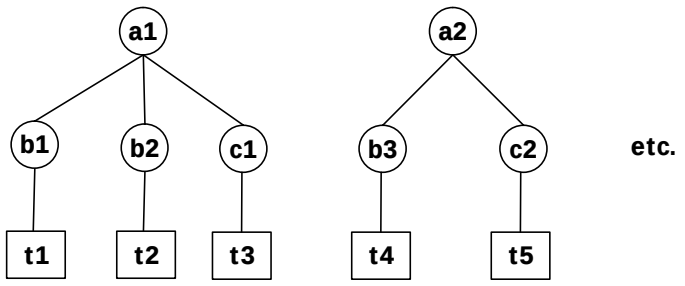

This example shows that certain aspects of a DTD have no impact on the selected storage scheme. If we vary the content model from (A,B) to (A?,B) or (A|B), this has only an impact on constraints that may be imposed on the tables, but not on the structure of the tables themselves. Since we do not consider the constraints for the reasons explained before, the relational schema is identical in these two cases.

## Step 4: Repeated Elements

- Given three element types A, B, C, with DTD

<!ELEMENT A (B*,C)>
<!ELEMENT B #PCDATA>
<!ELEMENT C #PCDATA>




| Id_A | Content_C |
|---|---|
| a1 | t2 |
| a2 | t5 |

| Id_B | Parent_B | Content_B |
|---|---|---|
| b1 | a1 | t1 |
| b2 | a1 | t2 |
| b3 | a2 | t4 |

foreign key

```
CREATE TABLE B
(
  Id_B INTEGER,
  Parent_B INTEGER ,
  Content_B CHAR
  PRIMARY KEY Id_B
  FOREIGN KEY Parent_B
  REFERENCES A(Id_A)
)
```



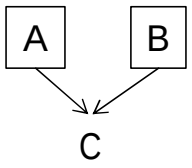


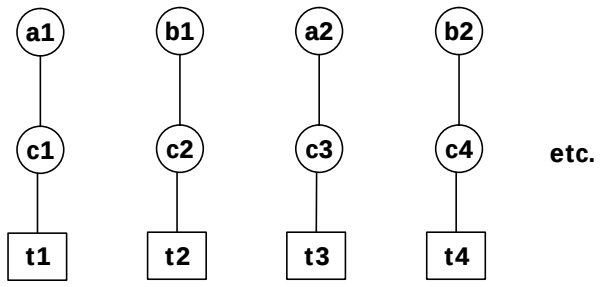
If the content model induces a 1:n relationship among the elements the creation of separate tables for the elements involved in that relationship is unavoidable. The tables are then related by employing a foreign key relationship. Note that the constraints introduced in the table definition are correct but not mandatory, since, as already argued earlier, only systems software will make use of the relational storage schema.

## Step 5: Shared Elements

- Given three element types A, B, C, with DTD

<!ELEMENT A (C)>
<!ELEMENT B (C)>
<!ELEMENT C #PCDATA>




| Id_A | Content_C |
|---|---|
| a1 | t1 |
| a2 | t3 |

| Id_B | Content_C |
|---|---|
| b1 | t2 |
| b2 | t4 |

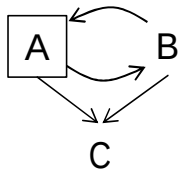


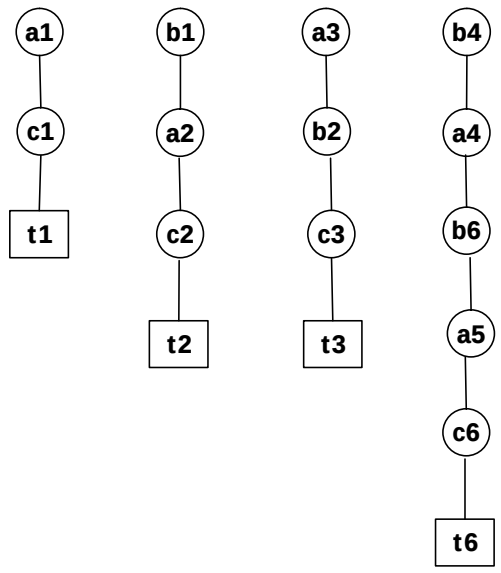

Shared elements in a DTD can without problems be inlined (Note: sharing of elements at the instance level does not exist !)

## Step 6: Cyclic Element Containment

- Given three element types A, B, C, with DTD

<!ELEMENT A (B|C)>
<!ELEMENT B (A|C)>
<!ELEMENT C #PCDATA>




| Id_A | Content_A_C | Content_A_B_C | Parent_A |
|---|---|---|---|
| a1 | t1 | NULL | NULL |
| a2 | t2 | NULL | _1 |
| _1 | NULL | NULL | NULL |
| a3 | NULL | t3 | NULL |
| a4 | NULL | NULL | _2 |
| _2 | NULL | NULL | NULL |
| a5 | t6 | NULL | a4 |





We know that recursive definitions of element content models are possible in XML DTDs. This poses a slight technical problem as the recursion needs to be resolved when mapping to the relational schema. In fact it is not necessary to map all element types along a recursive relationship into separate tables, it is sufficient to "break" the recursion by mapping one element type. In practice, the decision which element type to map to a relation along a cycle will be dictated by the relationships with other elements (e.g. one element taking part in a 1:n relationship).

## Nested Content Models

- Mapping defined for flat content models (without choice)
  - Nested content models ?
- The following is required (Completeness)
  - Any XML DTD X must be mappable to a relational schema R
  - Any XML document XD conforming to DTD X must be mappable to relational tuples in RD conforming to schema R
  - Any XML query against X must be mappable to an SQL query against R
  - The document XD must be recoverable from the database RD
- The following is not required !
  - the DTD X must be recoverable from the schema R !
- Therefore we may modify the DTD before mapping to a relational schema




We have seen how to map in case the content models are flat. However, XML allows nested content models as well. In order to understand how to deal with this case we have to consider which properties the mapping has to satisfy. A mapping must be able to deal with all possible XML DTDs and XML documents. Once a mapping is defined queries must be mapped and query results must be constructed from the relational representation. But: it is not necessary that the DTD must be recoverable from the generated relational schema. This is not required while using a relational DBMS for XML storage and querying.

We can take advantage of this observation by modifying the DTD before defining the mapping. Any modification that does not violate the requirement that all document instances can be mapped is permissible.

## Normalization of Content Models

- Normalization steps:
  - eliminate choice: (A|B) -> (A?, B?)
  - distribute occurrence operators: (A, B)* -> (A*, B*), (A, B)? -> (A?, B?)
  - contract occurrence operators: A** -> A*, A?? -> A?, A*? -> A*, A?* -> A*
  - contract elements: (A, B, A) -> (A*,B) (analogously for A*, A? instead of A)
- Example:

  (A?, (B*, A, (B|C)), B) -> (eliminate choice)

  (A?, (B*, A, (B?, C?)), B) -> (eliminate paranthesis)

  (A?, B*, A, B?, C?, B) -> (contract element A)

  (A*, B*, B?, C?, B) -> (contract element B)

  (A*, B*, C?)




Element content specifications in an XML DTD are (nested) regular expressions that constrain the order and cardinality in which sub-elements may occur: Cardinality constraints are relevant for the storage scheme (1:1 vs 1:n relationships, not null constraints): On the other hand we have seen that order is not relevant for storage, only the order in which elements occur within an *XML instance* matters. Thus order information can be captured as separate data values for instances if required. Cardinality constraints on the other hand are very important for the mapping, since the relational model can represent these directly, or better, depending on the cardinality of relationships different relational schemas are constructed. Therefore we can use any transformation that relaxes order constraints (naturally not further constrains them as we would exclude documents!) and maintains cardinality constraints.

The modification of DTDs is performed by using rewriting rules. The rewriting rules can be distinguished into three classes:

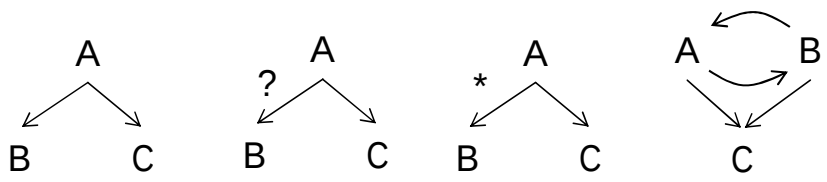
1. Elimination of choice by introducing an optional occurrence operator
2. Distribution or flattening rules, that are used to eliminate nesting in content models. Note that these rules only affect the possible ordering. For example the content model on the left hand side of the first rule allows only contents of the order A,B,A,B,… whereas the right hand side allows A,A,B,B…
3. After eliminating nesting it may occur that multiple unary operators are applied to an element. These operators can be contracted without changing the cardinality constraint
4. After eliminating nesting it may occur that the same element occurs in multiple places in the content model. With reordering those can be reduced.

Note that the rules are designed such that they are confluent (Church-Rosser property). Therefore the repeated application necessarily terminates in a normal form of the content model.

Remark: Some of the simplification rules could be formulated in a more stringent manner, e.g. (A,B,A) -> (A+,B). However, these more stringent rules would have no impact on the kind of relational tables that are generated (since optional occurrence is not distinguished from mandatory occurrence), and therefore the more relaxed rules are used in practice.

## DTD Graph

- For a DTD with normalized content models we can generate a DTD graph
  - each element type is represented by exactly one node
  - arrows connect elements with their subelements
  - arrows are marked with *, + or ? according to the content model
  - attributes are connected to their element



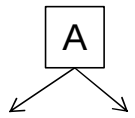




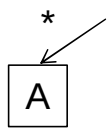
For describing the general algorithm for generating a relational schema from an XML DTD it is useful to introduce the concept of DTD graph on which the algorithm operates. The DTD graph is the graph induced by the element relationships in the content model of a DTD. We have already used DTD graphs in the earlier examples.

## Creation of Relations (Top Nodes)

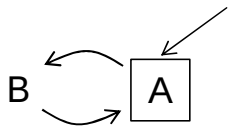
1. not reachable from another node



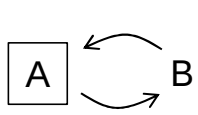

2. direct child of * or + marked edge



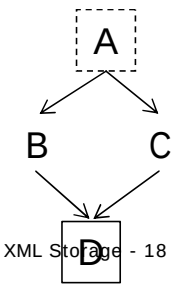

3. recursive node with in-degree >1




4. one node among two (or more) mutually recursive nodes with in-degree = 1




5. nodes with in-degree >1 that are reachable from the same top node by different paths composed exclusively of non-top nodes







The key step in the algorithm is the selection of element types for which relations are created, so-called top nodes. In our examples we have already identified the main prototypical situations for that. Here we generalize these examples to rules that cover all possible cases. These cases correspond to all situations where not creating a table for the element type satisfying one of these rules, results in the impossibility of representation of the respective data or in an ambiguous representation. For example, in the case of rule 5, if not creating a table for element type D, elements of D would be inlined into A, but it would be not clear whether they were reached via B or C.

The strategy to choose top nodes given by these rules correspong to so-called "hybrid" inlining. Other strategies (e.g. creating relations for every node) are possible, but are typically less efficient in terms of introducing to many unnecessary relatinos (and thus over-normalizing the relational schema).

## The "Hybrid" Inlining Algorithm

1. Normalize all content models
2. Create the DTD graph
3. Mark top nodes
4. Create a table for top nodes with a (system-generated) identifier
5. Create attributes for leaf nodes; compose names by concatenating the element/attribute names along path from top node to leaf node
6. Create an attribute root_type to record the root of the document
7. If a top node is direct descendant of several top nodes, add a parent_type attribute to denote the type of the parent top node[1]
8. If a top node is direct descendant of a top node add a parent_element attribute for keeping the foreign key of the parent top node[1]
9. If a node is child of multiple nodes but direct descendant of a single top node add a childof_type attribute to denote the type of the parent node

[1] here direct descendant means direct descendant in the graph of top nodes only




This is the complete hybrid inlining algorithm for generating from a DTD a corresponding relational schema. When generating the relational tables from top nodes the algorithm has to introdcue a number of different attributes required both for proper and efficient processing of queries. For top node elements system identifiers are generated. Inlined elements are identified by their path from the top node. The root-type attribute is added to improve the efficiency of query processing. If it were not available certain queries might be forced to perform exhaustive searches over multiple types in order to determine the root type. In case a top node is shared by multiple other elements the parent type attribute identifies where the parent of the element is found, which also optimizes the processing of queries.

## Example DTD

```
<!ELEMENT book (booktitle, author, citation)>
<!ELEMENT article (title, author, citation*)>
<!ELEMENT author (name)>
<!ATTLIST author id ID #REQUIRED>
<!ELEMENT name ((firstname, lastname)| lastname)>
<!ELEMENT citation (article|book)>
```

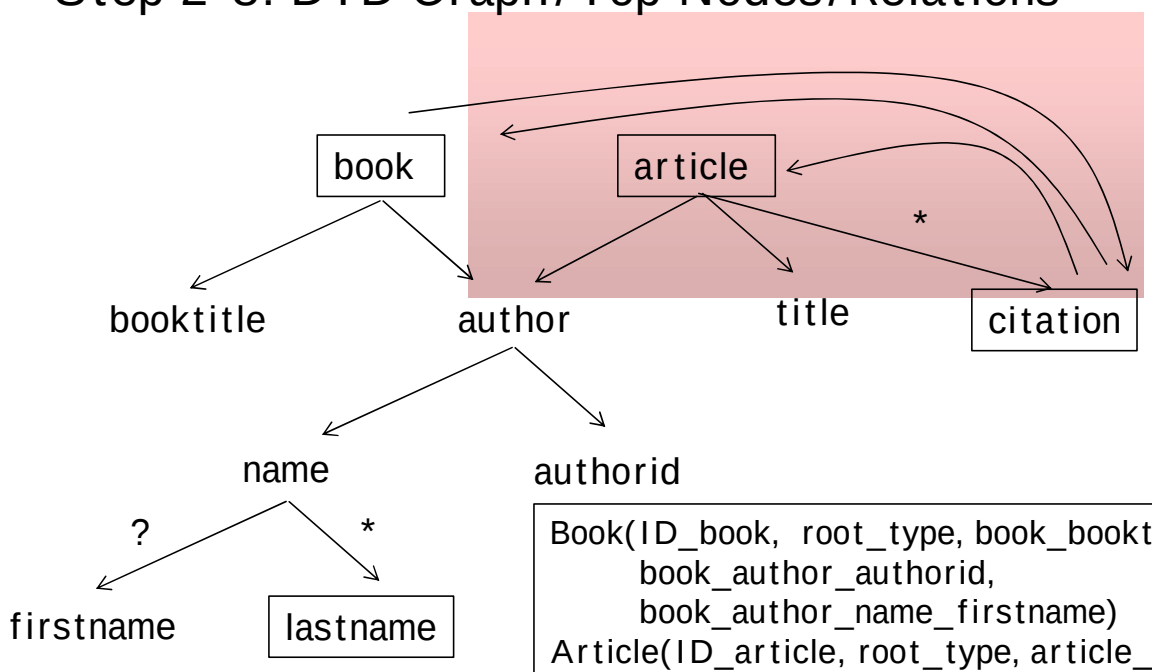
Step 1: Normalizing the DTD

((firstname, lastname)| lastname)
- -> ((firstname,lastname)?,lastname?)
- -> (firstname?, lastname?, lastname?)
- -> (firstname?, lastname*)




Note that in this example after normalization the content model of name relaxes the original constraints imposed. More document instances would satisfy it (i.e. documents with multiple lastname elements)

## Step 2-8: DTD Graph/Top Nodes/Relations




```
Book(ID_book,  root_type, book_booktitle,
        book_author_authorid,
        book_author_name_firstname)
Article(ID_article, root_type, article_title,
        article_author_authorid,
        article_author_name_firstname,
        parent_article)
citation(ID_citation, root_type,
        parent_type, parent_citation)
lastname(ID_lastname, root_type,
        parent_type, parent_lastname, lastname)
```

# Example Documents

```
<book>
 <booktitle>Complexity</booktitle>
 <author authorid=1>
  <name><lastname>Karp</lastname></name>
 </author>
</book>

<article>
 <title>NP=P</title>
 <author authorid=2>
  <name><firstname>Li</firstname><lastname>Yu</lastname></name>
 </author>
 <citation>
  <book>
   <booktitle>Complexity Theory</booktitle>
   <author authorid=3>
    <name><lastname>Shamir</lastname></name>
   </author>
  </book>
 </citation>
</article>
```

# Relational Tables

| ID_book | root_type | book_booktitle | book_author_authorid | book_author_name_firstname |
|---|---|---|---|---|
| b1 | book | Complexity | 1 | NULL |
| b2 | book | Complexity Theory | 3 | NULL |

| ID_article | root_type | article_title | article_author_authorid | article_author_name_firstname |
|---|---|---|---|---|
| a1 | article | NP=P | 2 | Li |

| ID_citation | root_type | parent_type | parent_citation |
|---|---|---|---|
| c1 | article | book | b2 |

| ID_lastname | root_type | parent_type | parent_lastname | lastname |
|---|---|---|---|---|
| l1 | book | book | b1 | Karp |
| l2 | article | article | a1 | Yu |
| l3 | article | book | b2 | Shamir |

## Processing of XML Queries

- Requires 3 Steps
  - Step 1: Conversion of XML Queries against the DTD to SQL queries against the relational storage schema
  - Step 2: Evaluation of the relational query
  - Step 3: Conversion of the relational query result into an XML document
- Difficulty
  - Step 1 is straightforward
  - Step 2 performed by RDBMS
  - Step 3 simple for XPath queries (only elements are returned), but very difficult for structured query results




The main difficulty in processing queries against the relationally stored XML data is the creation of result structures of a complex result type. (e.g. as it can become necessary with XQuery). We will not go further into this problem.

## Conversion of XPath Queries

- Conversion of path expressions
  - Start from the root of the query
  - Inlined elements can be directly accessed
  - Elements in other relations are accessed via joins
- Recursive path expressions (queries containing // )
  - Require transformation into an SQL fixpoint query (nonstandard feature)
- Example: The XPath query

```
/book/author/name/lastname
```

is converted to the following SQL Query

```
SELECT L.lastname
FROM book B, lastname L
WHERE B.ID_book=L.parent_lastname
```



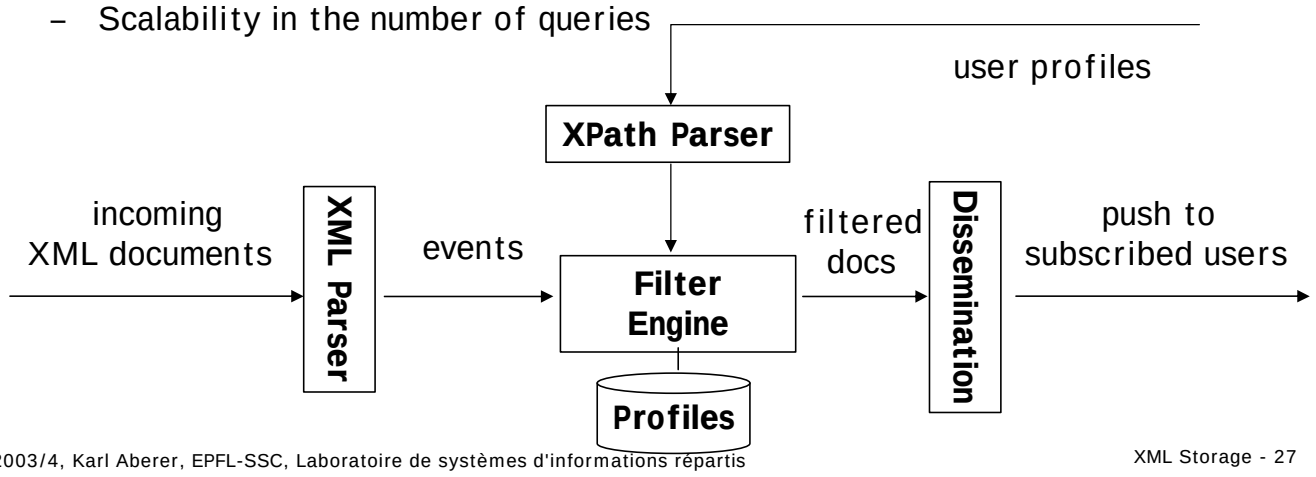

In the example we see that the author element needs to be accessed by means of a join whereas the other navigation can be directly performed within the table created for the author element by accessing the inlined elements.

## Summary

- What are advantages and disadvantages of reusing existing database system technology ?
- What are requirements when designing a relational schema to store XML data ? What is not required ?
- Which XML constraints can be ignored in order to simplify the DTD before creating a relational storage schema ?
- What are potential weaknesses of the "edge table" approach ?
- What is inlining of elements ?
- For which element types is a relation created in hybrid inlining ?
- Can a standard RDBMS process all XPath queries on relationally stored XML data ?

## 2. XML Document Filtering

- A large number of users wants to receive XML documents from a document stream according to individual profiles
  - News feeds, stock tickers, … (compare data broadcast)
  - Documents are distributed individually to users (unicast)
  - Individual profiles expressed as XPath queries
- Goal: efficiently match each single incoming document against a large number of user profiles
  - Minimize processing time
  - Scalability in the number of queries








One key application of XML is it's use as a message format. A potential application of that is the management of real-time data that is made available in form of XML messages. For example, this messages could contain news, stock information etc. Assuming that relevant messages are collected at large scale by some central information providers (e.g. as it is done by publishers today, e.g. Reuters) who intend to provide this information in real time to a large community of information consumers, the question arises how this could be done efficiently.

Simply broadcasting all messages to all users is clearly not the solution, as it would generated much unnecessary traffic. A better approach would be to send only those messages to the clients that are of interest to them. This can be done if the clients can provide in advance the information which they will be interested in. This information on the clients are called user profiles, and for an information filtering scenario the user profiles will be specified as queries against the message stream that arrives at the server. The server acts then as a filter engine that has to filter for each user the information that is potentially interesting to him according to his user profile.

This leads to a new situation which in a certain sense swaps the roles of data and queries in data management systems. We have no longer a large database containing data against which a stream of client requests (queries) need to be processed, but rather we have a large database containing queries (i.e. the user profiles) against which a stream of data messages needs to be processed, in order to identify for which queries the data is relevant. In the following we assume that the query language that is used is XPath, and that for distribution of documents, a unicast mechanism is used (such that it is sufficient to select the documents matching a profile, with more complex distribution mechanisms like multicast, also the most efficient distribution scheme would have to be determined).

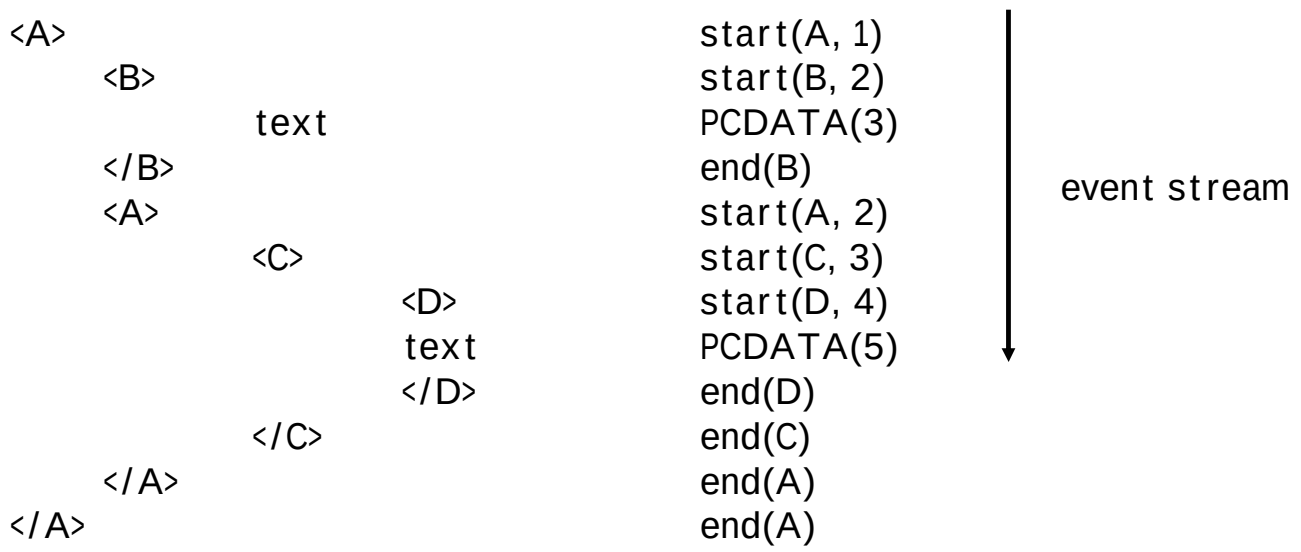
The architecture of such a filter engine is depicted above. XML documents would enter the filter engine and create a stream of document events (i.e. the document parts that are detected by the document parser). These events are sent to the filter engine. The filter engine obtains also (offline) from a XPath parser the structure of the XPath queries. From this structure it builds a "profile database" which is used to identify the queries which are successful (i.e. which produce a result for the message). The successful queries are forwarded to the data dissemination component which then distributes the messages according to the profile information (e.g. by email)

With this new scenario a new technical problem arises: one has to find ways to process large numbers of queries against a single document in short time.

## What Do You Think ?

- Possible approaches to filter XML documents given a large number of XPath expressions as profiles ?

## Event-based XML Parsing (SAX)








The approaches we study will make use of event-based parsing of XML documents, such as with SAX. Event-based parsing sequentially traverses the XML document and produces as a result "document events" which indicate where elements start and end, and provide the depth of the element in the document tree.

## Step 1: Prefiltering Event Stream

Example Query Set:

Q1 = /A/B//C
Q2 = //B/*/C/D
Q3 = /*/A/C//D
Q4 = //B/D/E
Q5 = /A/*/*/C//E

```
<A>                                   start(A, 1)      Q1, Q3, Q5 ok
    <B>                               start(B, 2)      Q2, Q4, ok
    text                              PCDATA(3)
    </B>                              end(B)
    <A>                               start(A, 2)
        <C>                           start(C, 3)
            <D>                       start(D, 4)
            text                      PCDATA(5)
            </D>                      end(D)
        </C>                          end(C)
    </A>                              end(A)
</A>                                  end(A)
```

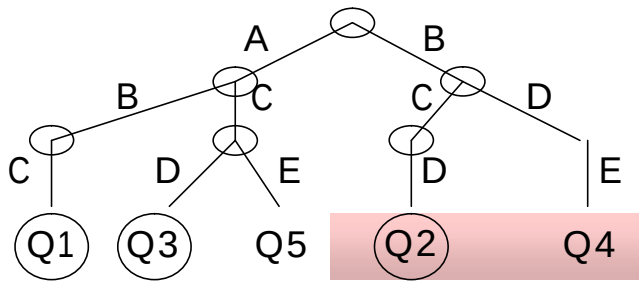




The naive approach would be to process all the queries against the document. This does not scale in the number of queries (and is in fact the counterpart of scanning a complete database when answering a query). Also simple techniques for improving sequential processing, like elimination of documents that do not share any element with the query do not substantially help.

Here the use of event-based parsing for simple prefiltering of documents is illustrated. It is checked whether the first element of the Xpath query can be found in the document. This is already more selective than just checking all queries against the document, but still does not very effectively filter the queries.

## Step 2: Building a Query Trie




Example Query Set:
Q1 = /A/B//C
Q2 = //B/*/C/D
Q3 = /*/A/C//D
Q4 = //B/D/E
Q5 = /A/*/*/C//E

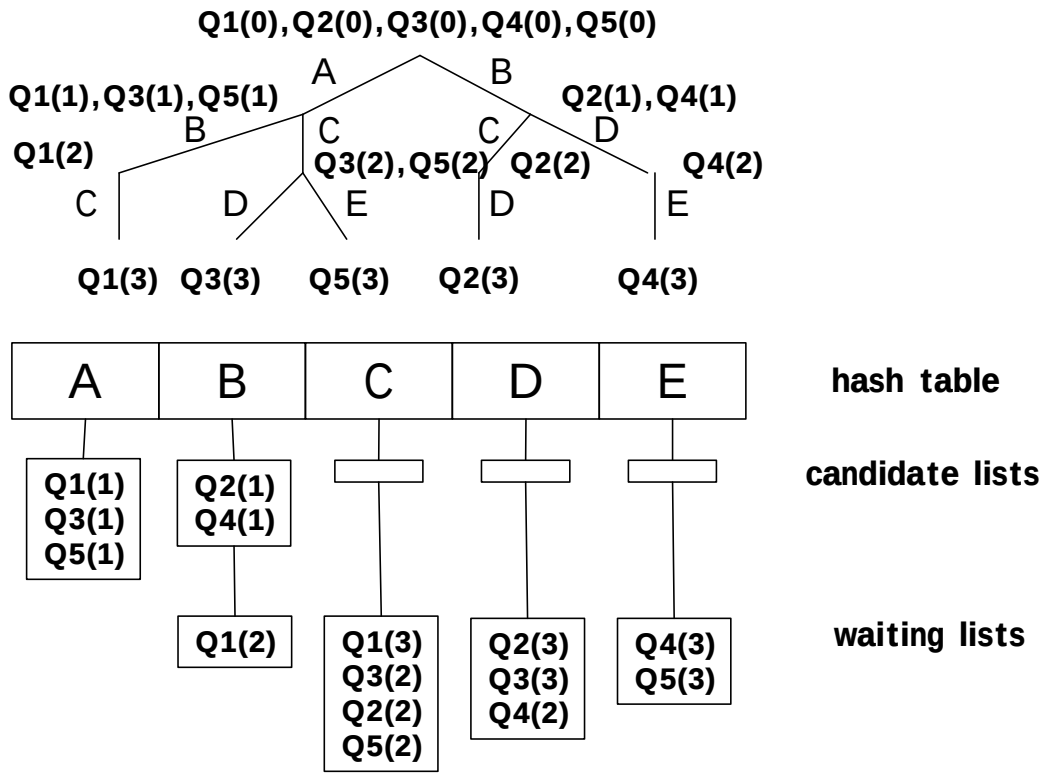
| | | | | |
|---|---|---|---|---|
| <A> | | | start A | |
| | <B> | | start B | |
| | text | | PCDATA | |
| | </B> | | end B | |
| | <A> | | start A | |
| | | <C> | start C | Q1 ok |
| | | | <D> | start D | Q2, Q3 ok |
| | | | text | PCDATA | |
| | | | </D> | end D | |
| | | </C> | | end C | |
| | </A> | | | end A | |
| </A> | | | | end A | |




The solution will be to build an index on the query (profile) database, such that efficiently all queries that match a document can be found. Such an index would be embedded into a filter engine.

A first idea of how a query index can be constructed is based on the observation that an XPath expression contains always a sequence of elements. If we just consider this sequence and ignore the rest of the expression, we arrive at the standard problem of indexing a set of strings. For this problem a well-known approach is to construct a trie. We illustrate the trie resulting from the set of example queries above. Once such a trie is available it can be used to identify while processing the event stream which queries qualify for a given document. Whenever the next required start element tag is observed for a query one can progress in the trie one level down. This is done in parallel for all prefixes of strings that qualify. Only queries at the leaves of the trie, that are reached in this manner, qualify to match the document.

## Compacting the Trie




**whenever a member of the candidate list is matched promote the next member of the waiting list for the respective query**




In practice the number of elements occurring in a large set of queries may be very large, and thus the internal nodes of the trie are correspondingly large. In addition we have seen that we have to keep track of multiple string prefixes while processing a document event stream. Therefore a better data structure for performing this processing (in terms of storage space required) is to use a hash table for all elements where we keep track of which nodes in the trie are currently active. We illustrate the transformation from the one into the other presentation in the figure. First, annotate each internal node in the trie with the state associated with each query. For example, for query Q1, the root is associated with state 0, thus Q1(0), the first level of the trie along the branch A is annotated with Q1(1) and so forth. In the hash table we enter then the states that are currently possible candidates for matching into a candidate list. That is, in the candidate lists we keep exactly those trie nodes that can be properly matched next to a document event. In the waiting lists we keep all the other trie nodes. Whenever a node in the candidate list is matched the next corresponding node of the query is promoted from the waiting list to the candidate list in the corresponding element entry of the hash table.

## Step 3: Adding Level Information

Q(p,l,r): p position of element in XPath expression
l required level of element in document
r relative level with respect to previous element
(na = not applicable, undet = undetermined)

Q(p,l,r) is called a pathnode

| | |
|---|---|
| Q1 = /A/B//C | Q1(1,1,na), Q1(2,2,1), Q1(3,undet,undet) |
| Q2 = //B/*/C/D | Q2(1,undet,na), Q2(2,undet,2), Q2(3,undet,1) |
| Q3 = /*/A/C//D | Q3(1,2,na), Q3(2,3,1), Q3(3,undet,undet) |
| Q4 = //B/D/E | Q4(1,undet,undet), Q4(2,undet,1), Q4(3,undet,1) |
| Q5 = /A/*/*/C//E | Q5(1,1,na), Q5(2,4,3), Q5(3,undet,undet) |

level l can be dynamically determined (using r) while document is parsed




The trie approach discussed before allows still incorrect matches. To make the selection more precise we also take into account in detail the information about the level at which elements occur in the query. We annotate the trie nodes with information on absolute and relative level of elements that we can extract from the query.

With respect to the absolute level we can observe that for queries starting at the root and having possibly some wildcards at the beginning the levels of elements are perfectly known as long as no descendant operator (//) occurs.

With respect to the relative level we can derive from the query the relative level with respect to the previous element in the query again as long as no descendant operator is in between. At the first level the relative level attribute is not applicable.

As a result we have now in some cases level information on the elements available. Some of this information may also be updated while processing an incoming document and matching elements. Then it can occur that an element of which the level is not fixed by the query (since it follows a // operator) after a match the level is fixed and the subsequent elements' levels can be derived using the relative position.

## Processing of Start Element Events

If start(element, level) arrives:

Step 1: obtain candidate list for element

Step 2: for each candidate entry Q(p, l, r)
if l > 0 then
if l = level then continue else no match
else continue

Step 3: check other conditions if applicable
(attributes, text content)

Step 4: if continue and conditions are satisfied
if last node of query path: successful match of profile
else promote the next node corresponding to Q from the
waiting list to the candidate list




Now we have to describe what exactly happens with the query index when a document event arrives, for each possible event, i.e. start and end element and character data events. For a start element the following steps are required

(1) Use the hash table to access the element name in the query index and obtain the candidate list

(2) For each entry check whether the level condition is satisfied, if this is not the case nothing happens

(3) If the level condition is satisfied other conditions could be checked (filters, however we did not allow them till now)

(4) If all the conditions have been successfully checked we have two possibilities: either the entry in the candidate list was the last one for this query, then the query has been successfully matched with the document and the profile is selected, or there exists a next node in the waiting list for this query that needs to be promoted to the candidate list.

## Promotion from Waiting List

Promoting the next entry Q( p+1, $l_w$, $r_w$)

Step 1: add a copy of Q( p+1, $l_w$, $r_w$) from the waiting list to the candidate list
/* Copying is required as the node may appear again later (multiple occurrences of the same element) */

Step 2: update the level value

if $r_w$ not undet then $l_w$ = $r_w$ + level else $l_w$ = undet

/* The character elements are processed analogously */




The promotion requires two main steps

(1) Copying the node from the waiting list to the candidate list

(2) Update the level value. Since we know now from the document at which level the event occurred we can fix this value (which could have been variable due to the occurrence of // operators). This also explains why the node is copied from the waiting list and not removed from it. It could, for example, be the case that multiple occurrences of the same element occur in the document at subsequent levels. For example, assume the query is //A/B and the document contains a path /A/A/A/B. Only the third time the node corresponding to element A is copied to the candidate list will eventually succeed. The other two entries (with different level values) cannot succeed as they will not find an element B at the next level.

## Processing of End Element Events

If end(element) arrives:

Remove the entry in the candidate list that has been entered when start(element,level) was encountered

- This is the pathnode of the query following the pathnode corresponding to the element
- This restores the list to the state when the start node was encountered
- Exploits the fact that the XML document is well-formed !

Example: Q1 = /A/B//C  Q1(1,1,na), Q1(2,2,1), Q1(3,undet,undet)

At the beginning Q1(1,1,na) corresponding to A is in the candidate list
When A is encountered Q1(2,2,1) is promoted
When end A is encountered Q1(2,2,1) is removed (the pathnode corresponding to B which is following A in the query)




When an end element event arrives (and the query has not been successful in the meanwhile) this means that the candidate list has to be reset to the state it was in when the corresponding start element had been processed.

## Example

Document

<A>
<B>text</B>
<C></C>
<B><C><D></D></C></B>
</A>

Document events

start(A, 1), start(B, 2), PCDATA(3), end(B, 2), start(C, 2), end(C, 2),
start(b, 2), start(C, 3), start(D, 4), end(D, 4), end(C, 3), end(B, 2), end(A, 1)

Query Q = /A/B//D

Pathnodes Q(1, 1, na), Q(2, undet, 1), Q(3, undet, undet)



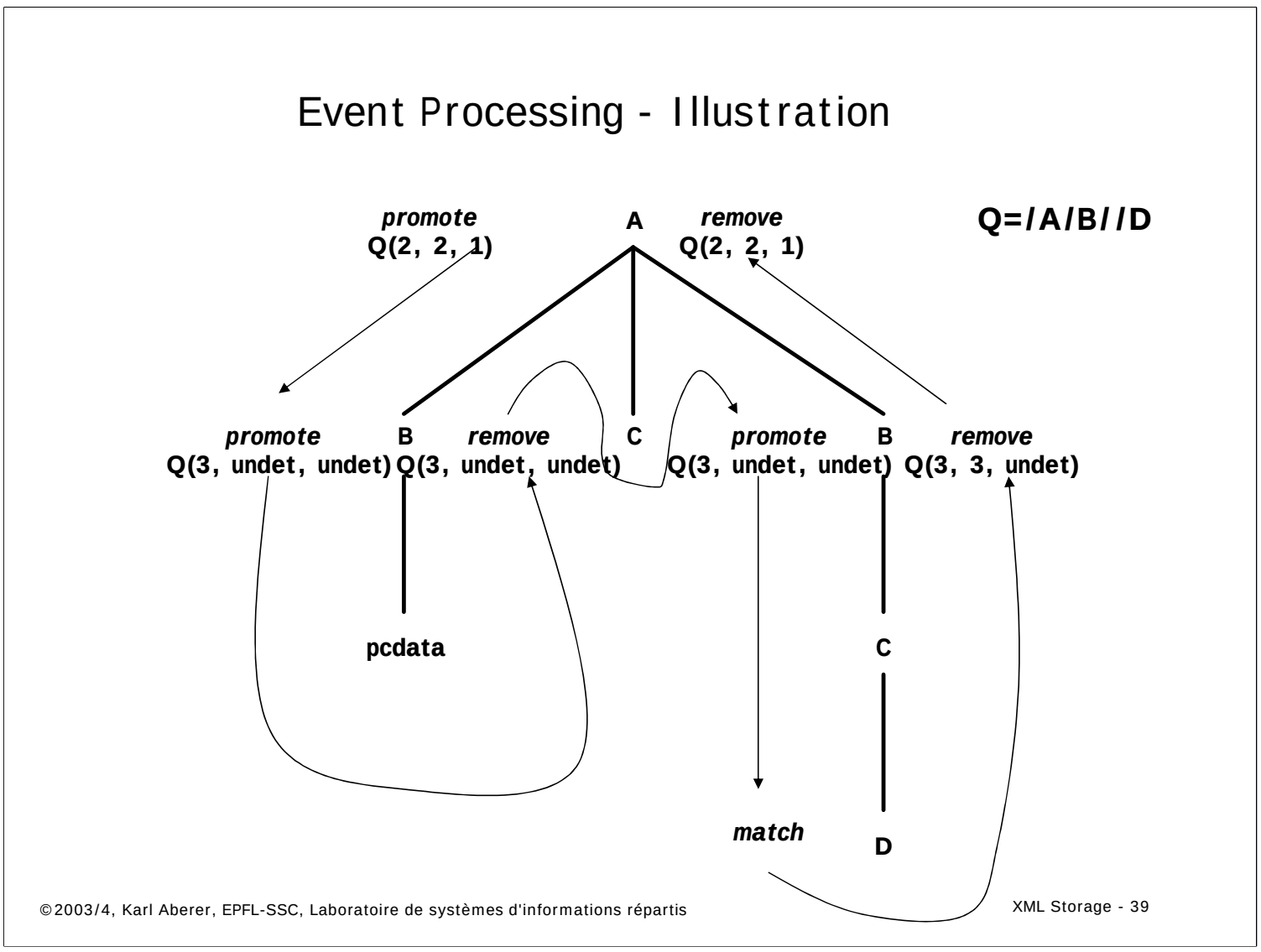

Note that instead of Q(2,undet,1) we could already determine the level and set Q(2,2,1). But the algorithm will in any case correctly determine the level value from the relative level information, therefore it can be left as well undetermined initially.

## Event Processing - Example

| Event | Processing | Candidate list |
|---|---|---|
| init | | A: Q(1,1,na), B:_, C:_, D:_, E:_ |
| start(A,1) | promote Q(2, undet, 1) | A: Q(1,1,na), B: Q(2, 2, 1), C:_, D:_, E:_ |
| start(B,2) | promote Q(3, undet, undet) | A: Q(1,1,na), B: Q(2, 2, 1), C:_, D:Q(3, undet, undet), E:_ |
| PCDATA(3) | | |
| end(B) | remove Q(3, undet, undet) | A: Q(1,1,na), B: Q(2, 2, 1), C:_, D:_, E:_ |
| start(C,2) | | |
| end(C,2) | | |
| start(B,2) | promote Q(3, undet, undet) | A: Q(1,1,na), B: Q(2, 2, 1), C:_, D:Q(3, undet, undet), E:_ |
| start(C,3) | | |
| start(D,4) | match | A: Q(1,1,na), B: Q(2, 2, 1), C:_, D:Q(3, undet, undet), E:_ |
| end(D) | | |
| end(C) | | |
| end(B) | remove Q(3, 3, undet) | A: Q(1,1,na), B: Q(2, 2, 1), C:_, D:_, E:_ |
| end(A) | remove Q(2, 2, 1) | A: Q(1,1,na), B:_, C:_, D:_, E:_ |




Note: for promoting B the level can be fixed, whereas for C it remains undetermined




This figure illustrates how the document parsing is performed in depth first order and how the corresponding process steps on the query index are performed.

## Handling Nested Path Expressions

- Nested Queries in XPath Filters are treated like separate queries
  - For absolute path expressions in filters this is sufficient to check whether the condition is satisfied
  - Otherwise it has to be processed with respect to the node of the parent query, i.e. the filter condition enters the candidate list when this node has been reached
- Complication
  - Parent query needs to be further processed before filter is fully evaluated
  - In that case the filter is marked and re-evaluated when the complete document has been processed




We shortly mention necessary extensions of the elementary processing model. The first concerns the processing of queries with filter conditions that involve themselves XPath expressions. The queries occurring in the filters need to be treated like separate queries. The two cases of absolute path expressions (starting with / or //) and relative path expressions (starting with an element name or .) need to be treated differently. This might lead to the problem, that the parent query can be further processed without knowing whether the filter is true. In such a case the query is further processed as if the filter were true, and the filter is marked for later evaluation. When the complete document is processed for all marked filters it has to be checked whether they have (in the meanwhile) also been successful, and only in that case the query is considered as being successfully evaluated.

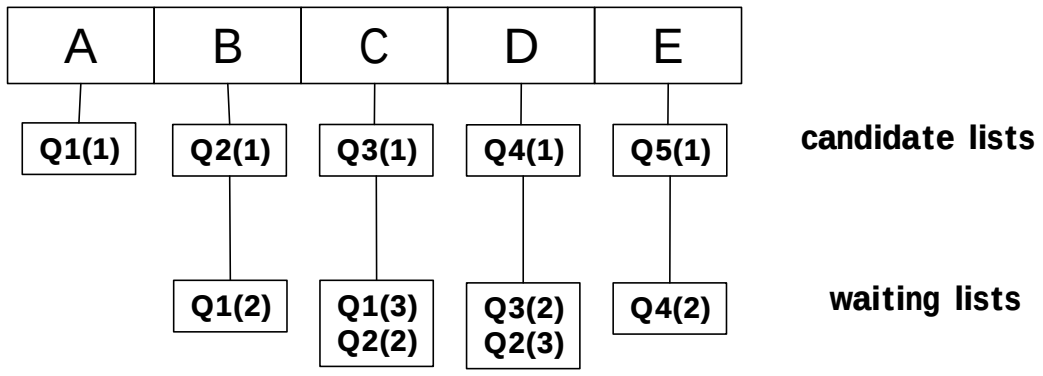
## Load Balancing the Index

- Problem: the candidate list will be highly skewed towards top-level elements in documents and queries
  - Low selectivity at the start of processing
- Solution: balance the lengths of the initial candidate lists
  - If a query is added the element of the query is chosen which has shortest current candidate list (pivot node)
  - This node is entered into the candidate list
- Modified processing
  - When the pivot node is encountered the prefix needs to be checked
  - To do that efficiently a stack of the traversed elements is kept




A practical problem that will occur with the approach described so far, is that the top level elements will at the beginning tend to have many more entries in the candidate lists (as query also tend to start with some top level elements) This would compromise the efficiency of the query indexing scheme as it relies on a high selectivity of the element name based access, whereas the candidate lists need to be sequentially processed. A solution to this problem is to add entries to the candidate lists by selecting some intermediate path nodes, called pivot nodes, such that the candidate lists remain balanced. This means however that the lists no longer are processed in the proper order. To compensate for this, one needs to check as soon as such a pivot node is encountered, whether the document events that have occurred so far actually match the prefix of the query. This can be done by keeping a stack of all the elements that have been traversed from the root to the currently processed document node.

# Load Balancing Example




| | |
|---|---|
| Q1 = /A/B//C | Q1(1,1,na,_), Q1(2,2,1), Q1(3,undet,undet) |
| Q2 = //B/*/C/D | Q2(1,undet,na,_), Q2(2,undet,2), Q2(3,undet,1) |
| Q3 = /*/A/C//D | Q3(1,3,1,/*/A), Q3(2,undet,undet) |
| Q4 = //B/D/E | Q4(1,undet,1,//B), Q4(2,undet,1) |
| Q5 = /A/*/*/C//E | Q5(1,undet,undet,/A/*/*/C) |

**first pathnode holds prefix information**

order of entry

## Summary

- How can database query processing and message filtering be compared ?
- Which approaches did we introduce to perform XML filtering ?
- Which properties are stored in path nodes ? Why is the relative position required in a path node ?
- Which properties of path nodes are modified during document event processing ? What needs to be done in document event processing when an end element tag arrives ?
- Why ist postprocesssing of filter conditions required ?
- Why is it not efficient to build candidate lists based on the initial elements of query paths ?

# References

- Course material based on
  - Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, Jeffrey F. Naughton: Relational Databases for Querying XML Documents: Limitations and Opportunities. VLDB 1999: 302-314
  - Dongwon Lee, Wesley W. Chu: CPI: Constraints-Preserving Inlining algorithm for mapping XML DTD to relational schema. DKE 39(1): 3-25 (2001)
  - Daniela Florescu, Donald Kossmann: Storing and Querying XML Data using an RDMBS. IEEE Data Engineering Bulletin 22(3): 27-34 (1999)
  - Mehmet Altinel, Michael J. Franklin: Efficient Filtering of XML Documents for Selective Dissemination of Information. VLDB 2000: 53-64
- More background material
  - http://www.acm.org/sigmod/record/index.html
    SIGMOD RECORD, Volume 30, Number 3, September 2001
    Special Section on Advanced XML Data Processing
    (ed. K. Aberer)