Chapter 4

Modeling Documents and Queries

The syntax of approXQL simplifies the formulation of queries; the syntax of XML simplifies the serialization of structured data. To define the *semantics* of approXQL, we need a model for the representation of queries and documents that abstracts from their syntactic characteristics. In this chapter, we present such a model. We propose type-value trees as a simple and elegant means to represent XML documents and approXQL queries in a uniform way. The underlying type system captures the syntactic types of XML like elements, attributes, and character data. It is extensible so that user-defined types can be added. We show how a collection of XML documents can be interpreted as a single type-value tree. Based on this interpretation, we are able to shift from the static notion of XML documents to the more flexible concept of logical documents. Logical documents are the units for the similarity valuations and are retrieved as answers to a query. We also demonstrate how approXQL queries can be decomposed into a set of conjunctive queries, and how each conjunctive query can be mapped to an extended type-value tree.

We begin the chapter with a review of important properties of trees. In Section 4.2, we introduce type-value trees, which are subsequently used to model XML documents (Section 4.3) and approXQL queries (Section 4.4). We conclude the chapter with a review of related work.

4.1 Trees and their Properties

We use standard definitions to formalize the notion of trees and their properties. These definitions can be found in several sources that deal with tree algorithms, e.g., [Knu69, AHU74].

A *rooted tree* is a structure $T = (N, E, r)$, where N is a finite set of *nodes*, $E \subseteq (N \times N)$ is a finite set of *edges*, and $r \in N$ is a node that forms the *root* of T. If $(u, v) \in E$ is an edge, then u is the *parent* of v, and v is a *child* of u. T must fulfill the following properties:

- The root has no parent.
- All nodes except the root have exactly one parent.

We use the notation $parent(v)$ to refer to the parent of a node v. The nodes in a tree that have a common parent u are called *children* of u, denoted by

$$children(u) = \{v \in N \mid (u, v) \in E\}.$$

A node without children is a *leaf*; all other nodes are called *inner nodes*. Two nodes with the same parent are *siblings*.

A *path* in a tree $T = (N, E, r)$ is a sequence of nodes $u_1.u_2 \ldots u_k$ such that $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$. The path starts at node u_1, ends at node u_k, and has the *length* k. We write $u \rightsquigarrow v$ if there is a path between u and v. Each path in a tree is unique. In particular, there is a unique path from the root to each other node in the tree. The *depth* of a tree is defined as the length of the longest path starting at the root node. If $u \rightsquigarrow v$ holds, then we say that u is an *ancestor* of v, and v is a *descendant* of u. The set of ancestors of v is defined as

$$ancestors(v) = \{\, u \in N \mid u \rightsquigarrow v \,\};$$

the set of descendants of u is defined as

$$descendants(u) = \{v \in N \mid u \rightsquigarrow v \,\}.$$

Let u be a node in a rooted tree $T = (N, E, r)$. The *subtree* $T' = (N', E', u)$ rooted at u is a tree, where

$$\begin{aligned} N' &= \{u\} \cup descendants(u), \\ E' &= E \cap (N' \times N'). \end{aligned}$$

We use the notation $T[u]$ to refer to the subtree $T' = (N', E', u)$ of T. A subtree $T[u]$ is an *immediate* subtree of a tree T if u is a child of the root of T. Consider the tree depicted in Figure 4.1 on the facing page: The tree within the grey triangle is an (immediate) subtree.

An *included tree* $T' = (N', E', u)$ in T is a tree, where

$$\begin{aligned} N' &\subseteq N, \\ E' &= E \cap (N' \times N'). \end{aligned}$$

We say that a tree includes another tree *directly* if both trees have the same root. Figure 4.2 shows an example: The nodes and edges bordered by the grey line constitute an included tree. It is directly included in an immediate subtree of the entire tree.

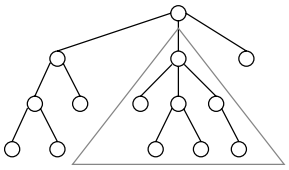

Figure 4.1: A subtree.

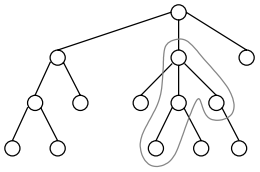

Figure 4.2: An included tree.

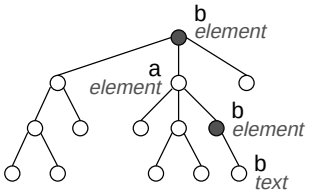

Figure 4.3: A recursive tree.

4.2 Type-Value Trees

In this section, we formalize the type system introduced in Section 3.2.3. We propose *type-value trees* as a means to enrich the nodes in a rooted tree by a type and a value. The value of a node is a generalization of a node label. It can be, e.g., a string, a number, or a date. The type of a node defines the domain of the node value and the set of predicates that can be applied to nodes of that type.

Definition 4.1 (Node type) *A node type is a pair $\tau = (D, P)$ consisting of a domain D and a set of binary predicates P. Each predicate $\phi \in P$ maps a pair of objects from D to a Boolean value: $\phi : D \times D \rightarrow \{ false, true \}$.*

Because all predicates used in this thesis have intuitive semantics, we omit the definition of the functions represented by the predicate symbols. We simply represent a predicate by its symbol. For example, if $\tau = (\mathbb{N}, \{=, <, \leq, >, \geq\})$, where $\mathbb{N}$ are the natural numbers, then $=, <, \leq, >, \geq$ are predicate symbols that represent the well-known binary operators for the comparison of natural numbers.

Like data types in programming languages, a node type may refine another node type. The refinement relationship between node types spans a *type hierarchy*.

Definition 4.2 (Supertype, subtype) *Let $\tau = (D, P)$ and $\tau' = (D', P')$ be node types. τ is a supertype of τ', denoted by $\tau \prec \tau'$, if and only if*

$$D \supseteq D' \wedge P \subseteq P' \wedge (D \neq D' \vee P \neq P').$$

If $\tau \prec \tau'$ holds, then τ' is a subtype of τ.

Intuitively, a node type is a subtype of another type if it is more specific (its domain comprises less objects) and more expressive (there are more predicates defined). The supertype symbol "$\prec$" hints at the graphical symbol for class inheritance used in the Uniform Modeling Language (UML) [RJB98]. We define the abbreviation $\tau \preceq \tau'$ for $\tau = \tau' \vee \tau \prec \tau'$.

Throughout this thesis, we use the two type hierarchies informally introduced in Section 3.2.3 (Figures 3.2 and 3.3 on page 40). The structural type hierarchy consists of three types: *element*, *attribute*, and their common supertype *struct*. Let Σ_E be the set of all valid element names, and Σ_A be the set of all valid attribute names. We define

$$\begin{aligned} \mathit{struct} &= (\Sigma_E \cup \Sigma_A, \{=\}), \\ \mathit{element} &= (\Sigma_E, \{=\}), \\ \mathit{attribute} &= (\Sigma_A, \{=\}). \end{aligned}$$

We define the data types in a similar way. Let Σ_D be the set of all tokens, Σ_T the set of all text words, $\mathbb{I}$ be the integers, and $\mathbb{R}$ be the real numbers. Then

$$\begin{aligned} \mathit{data} &= (\Sigma_D, \{=\}), \\ \mathit{text} &= (\Sigma_T, \{=\}), \\ \mathit{integer} &= (\mathbb{I}, \{=, <, \leq, >, \geq\}), \\ \mathit{real} &= (\mathbb{R}, \{=, <, \leq, >, \geq\}). \end{aligned}$$

We use the symbol $\mathcal{T}$ to denote the set of all basic structural types, all basic data types, and all user-defined types. The extension of a rooted tree to a type-value tree is straightforward:

Definition 4.3 (Type-Value tree) *Let $\mathcal{T}$ be a finite set of types. A type-value tree $T = (N, E, r, \mathit{type}, \mathit{value})$ is a rooted tree with a function*

$$\mathit{type} : N \to \mathcal{T},$$

which assigns a type to each node in N, and a function

$$\mathit{value} : N \to \bigcup_{(D,P)\in\mathcal{T}} D,$$

which assigns a value to each node in N. The value assignment is constrained:

$$\forall u \in N : \mathit{value}(u) \in D, \text{ where } D \text{ is the domain of } \mathit{type}(u).$$

A *recursive tree* is a type-value tree where a value appears twice or more on a path, and the nodes carrying the values have the same type or supertype:

Definition 4.4 (Recursive tree) *Let $T = (N, E, r, type, value)$ be a type-value tree. T is recursive if and only if*

$$\exists u, v \in N : u \rightsquigarrow v \wedge (type(u) \preceq type(v) \vee type(v) \preceq type(u)) \wedge value(u) = value(v).$$

Figure 4.3 on page 43 shows a type-value tree. For simplicity, only four nodes are annotated with types and values. This tree is recursive because the filled nodes are connected by a path and have the same types and values.

4.3 Tree Representation of XML Documents

Our model of XML documents aims at combining two objectives: First, we want to map an entire collection of documents to a single tree. This mapping allows us to define arbitrary subtrees (logical documents) to be results of a query. The differentiation between physical XML files and their internal representation is very helpful, because the granularity of a physical XML document often does not meet the desired result granularity: A single XML file may contain an entire media catalog; another file may contain a single CD; a third one may contain only a single track of a CD. Second, we want to enable the mapping of query selectors to tokens within the character data of documents. This is usual for search engines and retrieval systems, but not for most XML query languages, which bind query variables to values. A value typically comprises the entire text content of an element or the entire value of an attribute.

In the following, we describe the mapping of an XML document to a type-value tree, which is called *document tree*. Each element is modeled as a subtree of the document tree, where the root of the subtree is annotated with the type *element*. The name of the element is assigned as value to the subtree root. Direct-containment relationships between elements are modeled as parent-child relationships between the roots of the subtrees representing the elements.

Sequences of character data are split up into tokens using whitespaces as delimiters. The type of a token is determined by its syntactic characteristics. All tokens are by default of type *data*. If a token consists of digits only, then its type is refined to *integer*. Similar rules are used to figure out whether a token has type *text* or *real*. If a token has type *text*, then the stem of the word is derived. For each pair consisting of a token and a type, a node is created and annotated with the token and the type. All nodes derived from a sequence of character

data are added as children to the root of the subtree that represents the element containing the data.[1]

Each attribute is modeled as subtree of depth two: The attribute name is mapped to a node of type *attribute*. This node is then added as a child to the node that represents the element the attribute belongs to. The attribute value is split up into tokens as described for sequences of character data, and the tokens are mapped to nodes. These nodes form the children of the node representing the attribute name. Figure 4.4 shows an XML document and its mapping to a type-value tree.

```
<catalog>
  <cd year="2001">
    <performer> Ashkenazy </performer>
    <composer> Rachmaninov </composer>
    <title> Piano concerto </title>
    <tracks>
      <track length="13:25">
        <title> Vivace </title>
      </track>
      ...
    </tracks>
  </cd>
  <mc> ... </mc>
  ...
</catalog>
```

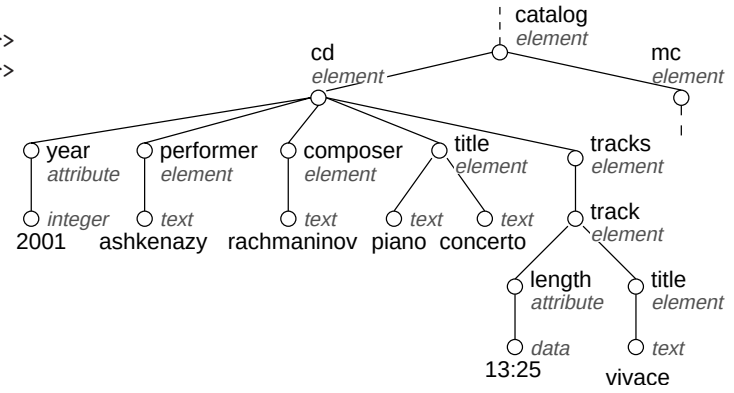

Figure 4.4: Mapping of an XML document to a type-value tree.

We add a new node of type *struct* with a unique value to the collection of type-value trees constructed for an XML document collection, and establish an edge between this node and the root of each document tree. The resulting tree is called *data tree*:

Definition 4.5 (Data tree) *Let C be a collection of (physical) XML documents and $C' = \{T_1, T_2, \ldots, T_n\}$ be the set of type-value trees constructed for the documents in C. The data tree $T_D = (N_D, E_D, r_D, type_D, value_D)$ of C is a type-value tree, where $type_D(r_D) = struct$, and the value of r_D is unique in the codomain of $value_D$. Each T_i $(1 \leq i \leq n)$ is an immediate subtree of T_D; no other immediate subtree of T_D exists.*

Finally, we replace the physical notion of documents by a more flexible logical notion:

Definition 4.6 (Logical document) *Let $T_D = (N_D, E_D, r_D, type_D, value_D)$ be a data tree.*

[1] The one-to-one mapping of tokens to nodes assumed here is a simplification of our model, but not a restriction. The value of a node may also consist of sequences of words, and a single word may be part of several text sequences assigned to nodes. In this way, we can implement the matching of query phrases.

Each subtree $T_D[u_D]$, $u_D \in N_D$, struct $\preceq u_D$, is a logical document with u_D as the document root.

4.4 Tree Representation of approXQL Queries

The modeling of an approXQL query is slightly more complicated than the modeling of an XML document, because a query may contain the Boolean operators "`and`" and "`or`". However, there is a simple correspondence between a query and a tree if the query is *conjunctive*, i.e., does not contain "`or`" operators. Then, the subqueries connected by "`and`" operators can be mapped to adjacent subtrees. A query with "`or`" operators can be transformed into an equivalent disjunctive normal form, and each conjunct of this normalized query can be mapped to a tree. We first introduce the modeling of conjunctive queries, and then show how arbitrary queries are interpreted as sets of trees.

A conjunctive query is mapped to a tree as follows: For each query selector, a node is created and annotated with the value and the type of the selector. If no explicit type is specified, then the appropriate default type (*text* for text selectors, *integer* for numerical selectors, *struct* for structural selectors) is used. For the operator of the selector a predicate is created and assigned to the node. If restrictions or relaxations for the selector exist, then the node is annotated with a set of modifiers taken from the set

$$M = \{\, \mathsf{insres}, \mathsf{insrel}, \mathsf{delres}, \mathsf{valres} \,\}.$$

For example, the set $\{\, \mathsf{insrel}, \mathsf{valres} \,\}$ assigned to a node indicates that an insertion relaxation and a value-change restriction for this node are defined. A value-change relaxation in a query is indicated by a sequence of alternative values separated by "`|`" signs. Queries with value-change relaxations are treated like queries with "`or`" operators (see below). A containment relationship between query selectors is interpreted as a parent-child relationship between query-tree nodes. If the expression enclosed by a containment operator includes "`and`" operators, then each subquery in the conjunctive expression is mapped to a subtree of the parent node.

Figure 4.5 on the next page shows the interpretation of a conjunctive approXQL query as a query tree. Because the query does not explicitly define a type for the selectors `cd`, `title`, and `composer`, they get the most general type *struct*. For simplicity, selection predicates that test for equality are not shown in the graphical representation of the query tree. For the rest of the thesis, we will omit the type names in the graphical representations of query trees and document trees if they are clear from context.

```
cd[attribute:year[> 2000:!]  and
   title["piano" and "concerto"] and
   *composer![“rachmaninov"]
```

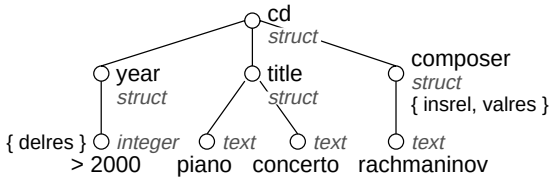

Figure 4.5: Mapping of a conjunctive approXQL query to a tree.

To represent a query tree formally, we extend the notation introduced in Section 4.1:

Definition 4.7 (Query tree) *Let $\mathcal{T}$ be a finite set of types, and M be a set of modifiers. A query tree $T_Q = (N_Q, E_Q, r_Q, type_Q, value_Q, pred_Q, mod_Q)$ is a type-value tree with a function*

$$pred_Q : N_Q \rightarrow \bigcup_{(D,P)\in\mathcal{T}} P,$$

which assigns a predicate to each node in N_Q, and a function

$$mod_Q : N_Q \rightarrow 2^M,$$

which assigns a set of modifiers to each node in N_Q. The predicate assignment is constrained:

$$\forall u_Q \in N_Q : pred_Q(u_Q) \in P, \text{ where } P \text{ is the set of predicates of } type(u_Q).$$

Consider the query tree depicted in Figure 4.5. If u_Q is the leftmost leaf, then $type_Q(u_Q)$ returns the type $integer = (\mathbb{I}, \{=, <, \leq, >, \geq\})$, $value_Q(u_Q)$ returns the value 2000, $pred_Q(u_Q)$ returns the predicate represented by the symbol ">", and $mod_Q(u_Q)$ returns the set $\{$ delres $\}$.

A query that contains "or" operators is partitioned into a set of query trees, which is called *separated query representation*. Each query tree in a separated query representation represents a conjunctive query in the hierarchical disjunctive normal form (HDNF) of the original query. The HDNF is similar to the disjunctive normal form (DNF) of a Boolean expression, but additionally includes the containment operators.

Algorithm 4.1 on the facing page shows a simple recursive function that constructs the HDNF of a query Q. The algorithm traverses the query top-down. If the current subquery Q has no further subqueries, then the selector s is returned (Line 1). Otherwise, there are further subqueries connected by an arbitrary Boolean expression $expr(Q_1, Q_2, \ldots, Q_m)$ (Line 2). The function performs a recursive call, passing all subqueries as parameters (Line 4). Having transformed all subqueries, the algorithm creates the DNF (in the Boolean sense) of the expression, treating the subqueries connected by the formulae F_{Q_i} as literals (Line 6). Each

Algorithm 4.1 creates the hierarchical disjunctive normal form of a query Q.

function `create_HDNF`(Q)

param: Q – a query,
returns: the hierarchical disjunctive normal form of Q.

1: **if** Q is a selector s **then return** s
2: Q has the form $s[expr(Q_1, Q_2, \ldots, Q_m)]$
3: **for** $i := 1$ **to** m **do**
4: $\quad F_{Q_i} :=$ `create_HDNF`(Q_i)
5: $\quad$ Substitute Q_i by F_{Q_i}.
6: Create the DNF F of $expr(F_{Q_1}, F_{Q_2}, \ldots, F_{Q_m})$
7: **foreach** conjunct F_C of F **do**
8: $\quad$ Substitute F_C by $s[F_C]$
9: **return** F

conjunct F_{Q_i} is syntactically enclosed by the root selector s of the current subquery (Lines 7 and 8). Finally, F is returned (Line 9) and is used as substitute of the original subquery (Line 4).

Value-change relaxations are transformed to disjunctions. For each alternative value of a selector, the subquery rooted at the selector is replicated. The alternative subqueries are then connected by "`or`" operators. Consider the query

```
cd[(title | category)["piano" and ("concerto" or "sonata")] and
   (composer["rachmaninov"] or performer["ashkenazy"])].
```

The replacement of the value-change relaxation by a disjunction results in the query

```
cd[(title["piano" and ("concerto" or "sonata")] or
    category["piano" and ("concerto" or "sonata")]) and
   (composer["rachmaninov"] or performer["ashkenazy"])].
```

This query is then passed to Algorithm 4.1, which creates the HDNF

```
cd[title["piano" and "concerto"] and composer["rachmaninov"]] or
cd[title["piano" and "concerto"] and performer["ashkenazy"]] or
cd[title["piano" and "sonata"] and composer["rachmaninov"]] or
cd[title["piano" and "sonata"] and performer["ashkenazy"]] or
cd[category["piano" and "concerto"] and composer["rachmaninov"]] or
cd[category["piano" and "concerto"] and performer["ashkenazy"]] or
cd[category["piano" and "sonata"] and composer["rachmaninov"]] or
cd[category["piano" and "sonata"] and performer["ashkenazy"]].
```

If n is the number of "`or`" and "`|`" operators in a query Q, then the HDNF of Q has up to 2^n conjuncts. From the HDNF, we construct the separated query representation of Q:

Definition 4.8 *The separated representation* $\mathbb{Q} = \{ T_{Q_1}, T_{Q_2}, \ldots, T_{Q_n} \}$ *of a query* Q *is a set of query trees, where each* T_{Q_i}, $1 \leq i \leq n$, *represents a conjunct of the HDNF of* Q.

To construct the separated representation of our example query Q, we assign a number to each conjunctive query in the HDNF of Q, map it to a query tree T_{Q_i}, and construct the separated query representation $\mathbb{Q} = \{ T_{Q_1}, T_{Q_2}, T_{Q_3}, T_{Q_4}, T_{Q_5}, T_{Q_6}, T_{Q_7}, T_{Q_8} \}$.

4.5 Related Work

Labeled trees and graphs are commonly used structures for the modeling of XML data (see [ABS99] for a survey). In the XML version [GMW99] of the OEM data model [PGW95], each element is mapped to a subgraph. The root of the subgraph is annotated with the set of attributes of the element; the edge leading to the root is annotated with the name of the element. If character data occurs within the element, a leaf node annotated with the character data is created. Similar data models are used in many projects (see, e.g., [FFK+97, CDSS98, ACVW00]). Our concept of type-value trees differs from those models in two aspects: First, sequences of character data are split up into tokens to allow fine-grained access to words and numbers. Second, elements, attributes, words, and numbers are uniformly modeled as subtrees, in which the nodes are annotated with types and values. The type system allows the users to expose or to hide the distinction between, e.g., elements and attributes, or integers and reals. We are not aware of another data model with equivalent properties.

Many researchers use labeled trees to model queries (see, e.g., [Kil92, Meu00, NS00]). However, to the best of our knowledge, there is no other model that supports (i) the interpretation of disjunctive queries as sets of query trees and (ii) the assignment of types, selection predicates, and transformation modifiers to query-tree nodes.