Sharing Content in Structured P2P Networks

Serge Abiteboul Ioana Manolescu Nicoleta Preda
INRIA Futurs – LRI, University Paris 11, France
Projet Gemo, Parc Club Orsay-University, 4 rue Jean Monod,
91893, Orsay Cedex, France
firstname.lastname@inria.fr

Abstract

The current development of Web standards and technologies has brought new opportunities for large-scale integration of *Web resources*: data sources (such as XML, HTML, or PDF files), distributed applications (typically accessed via Web Services), and semantic information (represented in formats such as OWL or RDF). At the same time, new peer-to-peer platforms are being developed, and increasingly used for data management at the network scale.

We present KADOP, a model, architecture, and system for *sharing content in a peer-to-peer environment*. By building on distributed hash tables technology, XML indexing and query optimization techniques, and on the paradigm of active documents, KADOP enables the publication, discovery, and efficient exploitation of content in a P2P environment. In some sense, with KADOP, the network can be seen as a content warehouse (for a domain of interest) as well as an entry point to external resources, in mediator style.

The system presented here as been implemented and is being tested. A novel aspect of the work is the use of active documents, to: (i) automatically maintain and refresh the distributed index, (ii) greatly reduce its overall space occupancy, (iii) provide bridges to other data repositories, such as databases or other peer-to-peer networks, and (iv) dynamically combine data-centric Web services to answer complex queries.

Keywords: XML, ActiveXML, data integration, Web services, DHT, intensional indexing, XML query processing.

1 Introduction

Enterprises, and more generally communities centered around some common interest, increasingly rely on the network for sharing content. Members of a community publish content (reports, emails, contracts, catalogs, phone books, etc.) as Web resources, typically, HTML or XML files and Web services. These standards, and the current Web technology, make content publishing simple and fast. On the other hand, access to published information still relies primarily on full-text search, a main drawback of this approach, in particular for industry applications. We introduce the KADOP system that builds on distributed hash tables [32], XML indexing techniques [6, 19], and active documents [30], to facilitate the publication, discovery, access, and integration of content in a P2P environment.

We are concerned here with a large and varying number of peers sharing content in absence of any centralized authority. The peers provide an integrated access to their heterogeneous, autonomous and distributed content. The first contribution of this work is to combine DHT and XML query technology in a unique setting, together with a semantic model to facilitate the access to content. A second, novel, and we believe most important contribution, is to build on the notion of intensional documents to manage changes, open the system to external resources, and support novel query optimization techniques.

A first prototype of KADOP is now running based on the Free-Pastry [32] and Active XML (AXML) systems [30]; it will be demonstrated shortly [7].

P2P XML query processing The core of KADOP technology is efficient query processing over distributed XML resources.

Indexing consists in publishing in a DHT, an index built in the style of [6, 19], allowing KADOP to efficiently support both full-text and structure-based queries. KADOP takes advantage of AXML and *intensional indexing* to drastically reduce the size of the index. It also uses typing and semantic information as a filter during query processing, to rapidly focus on relevant data sources.

Semantic-based access A first major use of KADOP is to support access to typed XML information. However, in a network setting, one cannot expect to always find homogeneously typed and well-understood information; so the use of semantics is compulsory. Many formats have been proposed to capture semantic information, most notably RDF [33] and OWL [31]. KADOP is based on a very simple semantic model that is only meant as a proof of concept; it does include the notions of abstract concept and namespace, and the most common semantic relationships, *isA*, *partOf* and *relatedTo*. Queries may be expressed directly over the XML types, or over the semantic concepts.

The KADOP system is based on AXML, i.e. documents with embedded Web service calls. We next highlight the main advantages brought by AXML.

Change management In KADOP, publication is controlled by an AXML system, which permits a fine control over document changes. AXML is used to automatically maintain and refresh the distributed index. In particular, KADOP supports collections of resources, which can be thought of as generalized bookmark folders. One can specify how to manage the indexing of a particular collection, i.e. how to update the index after document insertions, deletions or modifications (immediately, in batch mode, etc.).

Web services and openness to external resources KADOP also manages Web service resources, with WSDL [35] signatures. In the spirit of UDDI [28], Web services may be discovered in a KADOP network based on the presence of some keywords in their textual description, their classification in some service registry, or on their inputs and outputs semantics. However, the automatic use of Web services to answer queries goes beyond such descriptions. KADOP allows describing *Web services in context*. Intuitively, such contexts capture the semantic relations between a Web service's inputs and outputs, much in the spirit of a relational binding pattern [23], but for XML-structured, semantically rich data. With this approach, given a query, one can discover query plans that combine extensional information from XML documents and intensional information, provided by Web services. As we will see, this usage of the core AXML paradigm also allows KADOP to establish *bridges* to other content repositories, such as centralized databases or other peer-to-peer networks.

The paper is organized as follows. Section 2 introduces the key technologies underlying KADOP: AXML and DHTs. Section 3 describes the data and query model used in KADOP. We outline the architecture of a KADOP peer in Section 4, and discuss KADOP indexing and query processing. evaluation. We discuss related works in Section 7 and then conclude.

2 Preliminaries

In this section, we introduce two technologies at the core of KADOP: ActiveXML [3, 30] and distributed hash tables.

ActiveXML and intensional information ActiveXML (in short, AXML) documents are XML documents including calls to Web services. A call is represented under a specific XML syntax, and includes all information necessary to actually call the service, such as the service provider, the service and operation name etc.; the detailed syntax can be found at [30]. For ease of exposition, we will designate here service call elements simply as sc elements. When a service call is *activated*, an XML message is composed based on its information, and sent to the service. When the answer is received, its XML content is inserted in the original document, next to the service call element. Parameters of sc elements allow to control [3]:

- the time of the service call activation, e.g.: at the user request, at a given point in time, periodically, only if the result is needed, when some particular event (e.g., some particular call) occurs, etc.
- how to combine the results to the original document and in particular to previous invocations of the call;
- how long a service call result is considered valid;

A sample AXML document and its evolution are shown in Figure 1. The original document (top) contains two service calls: one to a Web service returning the current date, and another one to the *timeout* server returning the movies showing in Paris at the given date. The latter call has two parameters: the first one is the city, and the second one is the date, that must be retrieved from the <date> element. When the first call is activated, the result (shown in bold) is inserted in the document next to the call that produced it. Subsequently, when the call to *timeout.getMovies* is activated, it returns two <movie> elements. Each movie comes with a title and a call to another *timeout* service providing more details on the movie. In this simple example, we suppose for simplicity that all calls are triggered by the user.

The following observations on AXML are of importance in our context. **(1)** ActiveXML document contains: *extensional* information, appearing as plain XML, and *intensional* information, specified by means of sc elements. Such elements stand for the data that may be added to the document by making the associated service call invocation. **(2)** Web services may be called with intensional parameters, and they may return intensional results, as in the case of the service *getMovies* in Figure 1. **(3)** An interesting trade-off exists between intensional and extensional information. On one hand, intensional information tends to more compactly represent the same contents. As an example, compare the top and bottom versions of the sample document in Figure 1. Also, extensional information may get stale, e.g., a few weeks after the invocation of the call to *getMovies*, the list of movies obtained may no longer show. In contrast, intensional information, materialized when needed, produces up-to-date results. On the other hand, intensional data must be materialized, by invoking its service calls, before it can be queried; this may slow down query processing.

Distributed Hash Tables P2P systems can be classified according to the way peers are interconnected. In *centralized catalog* P2P systems, one or a few peers keep track of all resources in the network; this was the case of early systems such as Napster. In *unstructured* peer networks, peers establish arbitrary connections with one another, such as in the file-sharing networks Gnutella. *Structured* peer networks rely on a distributed data structure that keeps all peers logically connected. This data structure is equipped with protocols

```
<movies>
  <date><sc>local.getDate</sc></date>
  <sc>timeout.getMovies
    <param>Paris</param>
    <param>../date</param></sc>
</movies>
```

```
<movies>
  <date>
    <sc>local.getDate</sc>
    '03/03/05'
  </date>
  <sc>timeout.getMovies
    <param>Paris</param>
    <param>../date</param></sc>
</movies>
```

```
<movies>
  <date>
    <sc>local.getDate</sc>
    '03/03/05'
    </date>
  <sc>timeout.getMovies
    <param>Paris</param>
    <param>../date</param>
  </sc>
  <movie title="Alexander">
    <sc>timeout.getDetails(../@title)</sc>
  </movie>
  <movie title="Iznogoud">
    <sc>timeout.getDetails(../@title)</sc>
  </movie>
</movies>
```

Figure 1: Sample evolution of an AXML document.

for introducing new peers in the network, coping with peer withdrawal or failure etc.

We are concerned with the exploitation of P2P platforms for data sharing. In such platforms, data is kept at the owner peer, and some form of catalog is potentially distributed over the peer network. The basic operation is then to *lookup* a given search term: find which peers in the network own data relevant for the search term. In centralized catalog systems, lookup implies consulting the catalog holder(s), which quickly become a bottleneck. In unstructured P2P networks, in the absence of a catalog, the search is propagated along the connections between peers (also known as "flooding"); search performance degrades when large parts of the networks are traversed. Structured peer networks, in contrast, offer *upper bound guarantees* on the number of messages needed for a lookup. Well-known examples are Chord [25] and Pastry [24], whose upper bounds are in $O(log_2 N)$, N being the number of peers. Thus, we choose to rely on *structured P2P networks* for our P2P platform.

Distributed hash tables (DHTs) are an important class of structured peer networks. Such systems provide to the peers a simple hash table interface, in which each peer can insert *(key,value)* pairs. The DHT applies a hashing function on *key* to find one peer which will be responsible for storing all pairs sharing this key. Peers may issue *lookup(key)* calls to the DHT, to locate the peer responsible of *key*; such calls entail a bounded number of messages. A DHT is used to build a global index in a distributed data management platform, as follows: extract a set of *key*s from each data item, and insert in the DHT *(key,addr)* pairs, where *addr* is the address of the data item in the P2P network [15, 20]. For every *key*, this basic model assigns all *(key,addr)* pairs to one peer (we discuss existing variations in Section 7). DHT-based P2P platforms perform well under a relatively uniform key distribution. If a key is very frequent, the *(key,addr)* pairs on the peer in charge of *key* may get large; if the key is very often looked up, the peer may become a bottleneck. Thus, a proper choice of *(key,value)* pairs is crucial.

3 KADOP data and query model

In this section, we present the model that is used to describe the Web resources managed by KADOP .

3.1 Basic model

KADOP 's data model is outlined in Figure 2. We distinguish between two kinds of resources: *data items*, and *semantic concepts*.

Data items, shown at left in Figure 2, represent data resources of various granularities.

A *document* is typically an XML document. Other formats such as PDF and HTML may be used; in the sequel, the focus is on XML. Documents have associated *DTDs* or *XML schemas* describing their types. When such a schema is absent, KADOP extracts a simple DTD from the document, creating one type for each element name encountered. While both XML Schemas and DTDs are supported, in the following we will refer to DTDs for simplicity. Document *sub-trees* and leaves such as *text nodes* and *attributes* are also data items. The document sub-trees are identified by their roots. The *words* appearing inside a document as well as the labels of attributes and elements are also data items. We will ignore here issues such as stemming and composed words that typically introduce more data items.

A *view* corresponds to a query over some XML documents. We distinguish between *virtual* and *materialized* views. A view that is materialized (typically at a new location) may be defined in any query language; from KADOP's viewpoint, it is a new document (although the connection to the original document remains). On the other hand, we impose that non-materialized views be defined in a simple XPath fragment. This is because virtual view semantics is not clearly defined for more complex languages endowed with new element construction, such as XQuery [36].

A *collection* is a user-defined set of data items. Collections generalize the familiar concept of Web navigator bookmarks. Just like for bookmarks, the insertion of a data item in a collection does not entail its replication.

Finally, *Web services* form the last kind of data item. A Web service is a function taking as input typed XML documents, and returning a typed XML document. Typically, its signature is defined using WSDL [35].

Each data item is uniquely identified by an *item ID*, which in usual Web terminology corresponds to a *Universal Resource Identifier (URI)* [34], that also fully specifies the location of the item, i.e. also serves as an *Universal Resource Locator*. To simplify the presentation, we will assume here that KADOP peers only

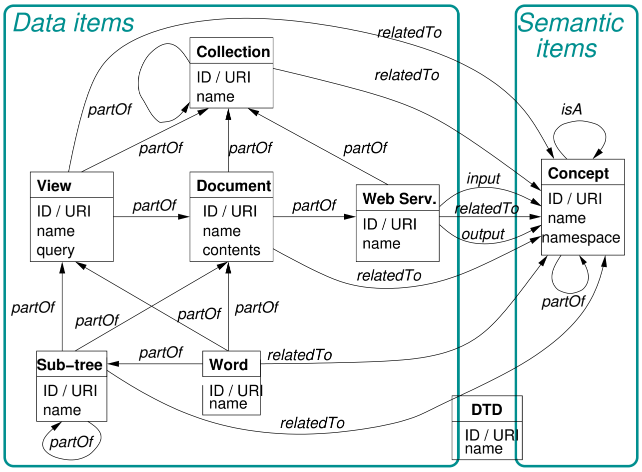

Figure 2: KADOP data model.

publish their own resources. (The system also allows publishing external resources.) In this case, the KADOP URL consists of: the physical ID of the item's peer, and a *intra-peer locator* specifying the item's address within its peer. Thus, all data items, including KADOP -specific ones such as collections and views, are given unique IDs (at the same time, URIs and URLs). The precise ID syntax depends on item granularity; we will present it in Section 4.1).

Observe that the data world of KADOP is centered around XML. Indeed, even collections and views live in the (Active) XML world. Furthermore, we also incorporate the *partOf* relationship of XML to connect data items in the natural way. In particular, a word is part of a sub-tree, which may be part of another sub-tree; a sub-tree may be part of a document, etc. Data items of any type can be part of a collection. A data item residing on one peer may be part of a collection defined on another peer.

Semantic concepts, depicted at right in Figure 2, encapsulate the semantic information of the peer network. We distinguish two classes of semantic concepts. *Abstract concepts* are explicitly created by the users, by providing a name, and an optional text description of the concept. *Concrete concepts* are automatically derived by the KADOP system from DTDs: a concept c_τ is created for each type element or attribute type definition τ appearing in the DTD.

Different concepts may be defined with the same name, but possibly different meanings. To allow distinguishing them, each concept has an associated *namespace* representing its usage context. KADOP creates a *namespace for each DTD*, and assigns this namespace to all concrete concepts issued from the DTD. Abstract concepts may have user-specified namespaces, or be assigned a *default peer namespace* representing all concepts defined on a peer.

Namespaces play the world of information worlds (although relationships such as *isA* may provide bridges). We assume that a document is associated to the namespace of its DTD. Sub-trees, text nodes, attributes and words inherit the namespace of the document where they occur. A Web service is associated to the namespace corresponding to its WSDL description. The namespaces of views and collections are a priori of less interest, unless they are connected to a namespace via an explicit *relatedTo* relationship to a concept in that namespace.

Concepts may be connected by two kinds of relationships: *partOf*, and *isA*. A set of *partOf* relationships are derived from every DTD, as follows. For every parent-child relationship between DTD types τ_1 and τ_2, we create a *partOf* relationship between the concrete concepts c_{τ_1} and c_{τ_2}. This conversion reflects the fact that DTD types correspond to potentially interesting concepts, and parent-child relationships in DTDs typically reflect object nesting.

From data to semantics The main relationship that connects the two worlds is the *relatedTo* relationship from data items to concepts. It reflects the fact that a particular data item is pertinent to a given concept. A *relatedTo* relationship may be produced in number of ways. First, it may be specified by a user. Second, it is automatically inferred by KADOP for an element of type τ, and the corresponding concrete concept c_τ. Third, it may be derived automatically, e.g., by a document classifier, or using relevance functions in an information-retrieval style: above a given threshold of relevance of a data item to a concept, a *relatedTo* relationship is created. In this work, we are concerned with the exploitation of such relationships, and will not discuss further how they are derived.

A Web service may be related to some abstract concept, e.g., Gemo:EmailService. But for Web services, we also have information on their input and output. Thus, the system also provides two automatically derived re-

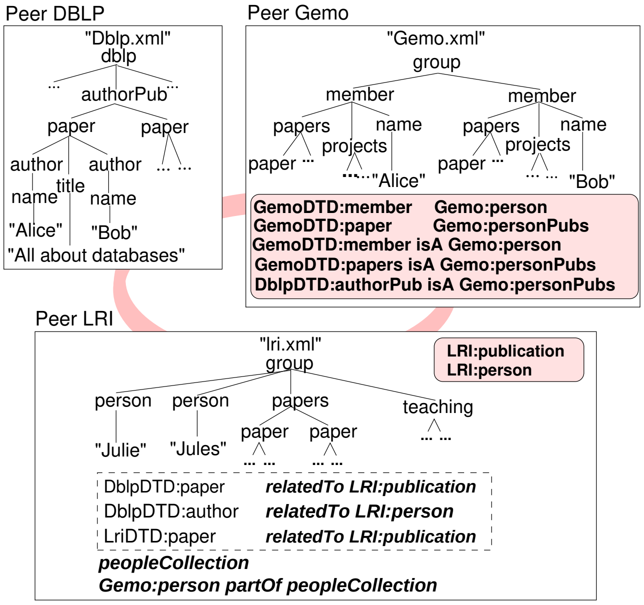

Figure 3: Sample instance of KADOP 's data model.

lationships, *input* and *output*, relating a Web service to the concrete concepts corresponding to its input and output (see Figure 2).

Some applications may require higher-level semantic constructs, expressed in languages such as RDF [33] or OWL [31]. The rather simple model we use here is meant as a proof of concept. More work is needed to check which more elaborate constructs can be mapped to this model and thus supported by the actual system, and which would require more sophisticate reasoning mechanisms .

Example. Figure 3 illustrates an instance over three peers: the French research labs Gemo and LRI, and the DBLP bibliographic server. XML documents are shown as trees, concepts and relationships appear in rounded boxes, collections and collection components are shown in italics, and *relatedTo* relationships appear in dashed boxes. The figure illustrates both concrete concepts, such as GemoDTD:paper, corresponding to the paper type in the DTD of the "Gemo.xml" document, and abstract concepts, such as Gemo:personPub, which stands for "all the papers of a given person". Abstract and concrete concepts can be freely connected. For instance, the concrete concept GemoDTD:papers, which refers to the paper children of member elements, is a specialization of Gemo:personPub.

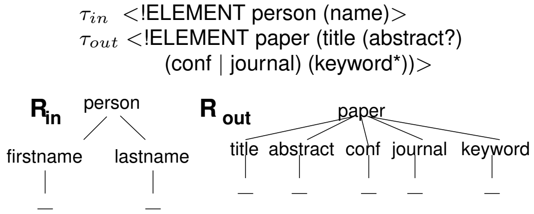

Figure 4: Web service input and output types, and corresponding representative documents.

3.2 Semantics for Web services

Web services may be described at various levels of detail and abstraction. We already saw that a Web service may be described as related to some concept. We now refine this mechanism to simplify service discovery and usage.

A Web service w has an associated WSDL [35] description, that determines its input and output concepts. More precisely, the WSDL description of w determines its input and output types, say τ_{in} and τ_{out}, from which KADOP derives the concepts $c_{\tau_{in}}$ and $c_{\tau_{out}}$.

Furthermore, based on τ_{in} and τ_{out}, we automatically generate *representative input and output XML documents* R_{in} and R_{out} for the Web service. These documents may be seen as *abstract materializations* of w that represent possible *sources of data* for the service, as well as possible *data results* obtained by invoking it.

Figure 4 illustrates representative documents for the service *Gemo:GetPapers* whose input and output τ_{in} and τ_{out} are described in a simple DTD-style syntax. Representative documents feature one instance of each required element type, regardless of its DTD cardinality constraints; text and attribute values are filled in with dummy values, unless the values can be inferred from schema constraints. Recursive types are unfolded up to a limited depth fixed in advance, a pragmatic choice dictated by the fact that XML data recursion is rarely deep [21]. The construction algorithm traverses the type definitions of τ_{in} and τ_{out} top-down. We omit the details.

The person and paper elements in a representative document share the namespace of the *Gemo:GetPapers* service, and are implicitly related to the concrete concepts derived from the service's input and output types.

Representative documents will turn to have many usages. (i) Inserting the R_{out} document in an AXML document, as a sibling of an sc element encoding a call to w, makes explicit the kind of data which may appear at that spot, when the call is activated. This is useful for resource indexing (see Section 4.1). (ii) Publishing the R_{in} and R_{out} documents, e.g., on the service provider peer, makes it easier for the query processor to understand the hierarchical structure of τ_{in} and τ_{out}. We will detail this in Section 4.2. (iii) Finally, representative documents may be used to construct *Web service contexts*; this is described next.

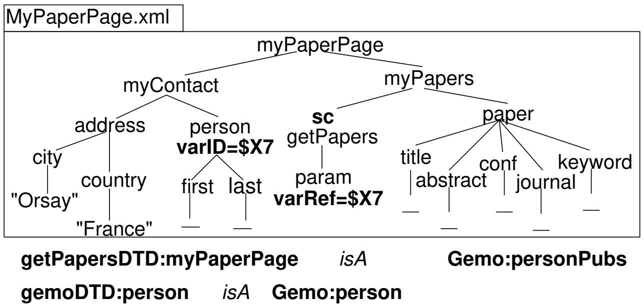

Figure 5: Sample service context for *Gemo:getPapers*.

3.3 Web services in context

We believe that although useful, the discovery of Web services based on related concepts, or input/output types, is too limited for a truly declarative access to information. For instance, if a user is interested in the publications of Alice, a Gemo researcher, the user's need should be answered regardless of whether Alice's publications are actually materialized in a document *AlicePapers.xml*, or whether the *Gemo:GetPapers* service must be invoked in order to obtain them. To enable this uniform access, one must be able to specify, for a Web service, not only the semantics of its input and outputs, but also the *semantic connection* between its input and output, how the service fits in larger data contexts, and/or constraints on its input and output.

Using the KADOP model and its AXML basis, such semantics can be elegantly specified by means of an AXML document, including a call to the service. We introduce this by means of an example.

Figure 5 illustrates a *Web service context* for the *Gemo:GetPapers*. At the top, we depict a (mostly intensional) document *MyPaperPage.xml*. The person element includes a copy of the R_{in} document of this service (see Figure 4). We add to this element a KadoP:varID attribute, which allows to refer to it in other places in the document. The papers element contains a call to *Gemo:GetPapers*; its parameter is annotated with a KadoP:varRef attribute, relating it to the copy of R_{in}. At the bottom, a semantic statement relates the document's root element, identified by *myPaperPage*, to the Gemo:personPubs abstract concept (recall Figure 3).

Writing Web service contexts based on representative documents is a means to specify: (i) *roles of the service's inputs and output within a hierarchical context*, (ii) *how the input and output are related*, and (iii) *possible Web service domain constraints*. For instance, Figure 5 shows how to use *Gemo:GetPapers* to populate the "MyPaperPage.xml" of an unspecified Gemo group member. The personal details under myContact can be filled with an input message for *Gemo:GetPapers*. The actual papers, grouped under the myPapers element, can be obtained by activating the call to *Gemo:GetPapers*, with the person sub-tree as input. Finally, the document in Figure 5 specifies that *Gemo:GetPapers* only applies for researchers located in Orsay, France. Similar constraints can be applied directly on R_{in} and R_{out}. For instance, if the service only returned papers written by Alice, we would replace the _ under first by the value "Alice"; if the service only returned VLDB papers, we would replace the _ under conf by "VLDB".

Web service contexts go beyond WSDL's expressive power, and are in the spirit of more complex formalisms such as RDF or OWL. Still in the spirit of RDF, several context descriptions may exist for a service. One main role of the KADOP query engine is to exploit such descriptions when processing queries, as we will show further.

3.4 Exploiting KADOP resources

Resources published in a KADOP network may be exploited in two ways, corresponding to the casual, respectively, to the more expert user. The simplest way to discover KADOP network resources is *navigational*. Navigation may start from a keyword search, typically specifying some particular words or concepts of interest, possibly also specifying the kind of resources to retrieve. For instance, one may ask for all documents or services containing the words "article" and

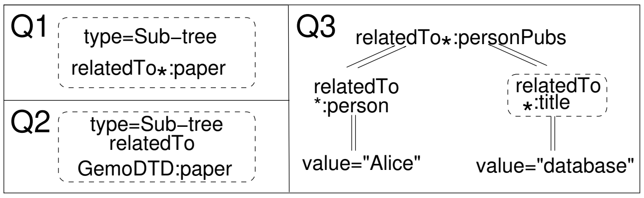

Figure 6: Sample KADOP queries.

"databases", search for "book" concepts, retrieve their instances, etc. Once some items are obtained by a user, she may want to refine the query or navigate along relationships, and so on. While navigating, the search interface allows the user to create new resources, such as views, collections, concepts, relationships etc.

More expert users can issue KADOP *queries*. A query consists of a *set of tree patterns* and (possibly) a *set of join conditions*. We start by describing tree patterns.

A tree pattern is meant to match some (possibly virtual) data items. Edges marked by / correspond to the direct *partOf* relationship, while //-annotated edges correspond to the transitive closure of *partOf*. The following constraints may be placed on the nodes of a tree pattern:

1. *type*=t specifies the type of data items matching this node, e.g., collection, document, view, subtree, word, or Web service.
2. *value*=v specifies the textual value of the data item, typically a word. In the case of an attribute or an element, it is required to match its content.
3. *relatedTo* ns:*concept* is satisfied by all data items *relatedTo* this concept, or to a narrower concept ns':c' such that ns':c' *isA* ns:*concept*. Instead of the namespace ns, a $*$ may be used to refer to *concept* within an arbitrary namespace.
4. *URL*=u specifies a precise data item URL. It may be the URL of a document or a Web service. It may also be the URL of a KadoP view or collection, e.g., mypeer.com:/KadoP/collections/mine.xml. One could also consider URL patterns such as www.inria.fr/*, which matches all resources published in the system with an URL matching the particular pattern.
5. *input* or *output*=ns:*concept* that matches any Web service with the specified input or output behavior.

To avoid queries that we consider under-specified, we require that at least one *value*, *relatedTo*, *URL* or *input/output* condition be specified on each pattern node.

A KADOP query may use *join conditions* to connect nodes from different tree patterns. For now, we consider only *value-based* semantics, i.e., the data items must have the same value. For simplicity, we consider queries returning values for a single *return node*, typically called n_{ret}.

Sample KADOP queries are shown in Figure 6. Q_1 returns all sub-trees instances of some concept "paper" in the peer network. In the sample instance in Figure 3, such sub-trees can be found in all three peers, related to the concrete concepts DblpDTD:paper, GemoDTD:paper and LriDTD:paper.

Q_2 is more specific: it only refers to the GemoDTD:paper concept, thus it will only return Gemo paper subtrees.

Q_3 asks for papers written by Alice, having "database" in the title. Notice that the author, and the papers, must be part of a common data item related to some personPubs concept. In the sample instance in Figure 3, Q_3 is matched by data items in the DBLP document (notice that dblp:authorPubs is a specialization of Gemo:personPubs. Moreover, Q_3 is also matched by the context document associated to *Gemo:getPapers*, shown in Figure 5. We will explain how KADOP 's query engine computes answers from such intensional documents in the next Section.

We will use $\mathcal{I}$ to refer to the data in a particular KADOP system. For a tree pattern t, N_t is the set of nodes in t, and similarly N_q for a query q. The set of nodes in $\mathcal{I}$ is denoted $N_{\mathcal{I}}$. We next define the query semantics using the auxiliary notion of embedding:

Definition 1 (Query semantics) *An* embedding *of a tree pattern t in $\mathcal{I}$ is a function $\phi : N_q \rightarrow N_{\mathcal{I}}$, such that:*

- *for each $n \in N_t$, $\phi(n)$ simultaneously satisfies all conditions attached to n.*
- *for each $n_1, n_2 \in N_t$ such that n_2 is a / child of n_1, $\phi(n_1)$ is connected by a* partOf *relationship to $\phi(n_2)$.*
- *for each $n_1, n_2 \in N_t$ such that n_2 is a // child*

of n_1, a chain of partOf *relationships goes from $\phi(n_2)$ to $\phi(n_1)$.*

For a query t with $N_t = \{n_1, n_2, \ldots, n_k\}$, the image *of $\mathcal{I}$ by t is defined by:*

$$t(\mathcal{I}) = \{\ (\phi(n_1), \ldots, \phi(n_k)) \mid \phi \text{ embeds } t \text{ in } \mathcal{I}\}$$

Let $q = (t_1, \ldots, t_l, C, n_{ret})$ be a query where the t_is are tree patterns over disjoint nodes, C are join conditions (equalities between nodes) and n_{ret} a node in one of the tree patterns. The result of q on $\mathcal{I}$ *is defined by:*

$$q(\mathcal{I}) = \Pi_{q_{ret}}(\sigma_C(t_1(\mathcal{I}) \times t_2(\mathcal{I}) \times \ldots \times t_l(\mathcal{I})))$$

The language and its semantics can be extended in a standard way to queries returning more than one nodes and to queries constructing more complex results.

Observe that the semantics was defined on extensional data. However, KADOP query evaluation does not distinguish between intensional and extensional data; this will be described in Section 4.2.

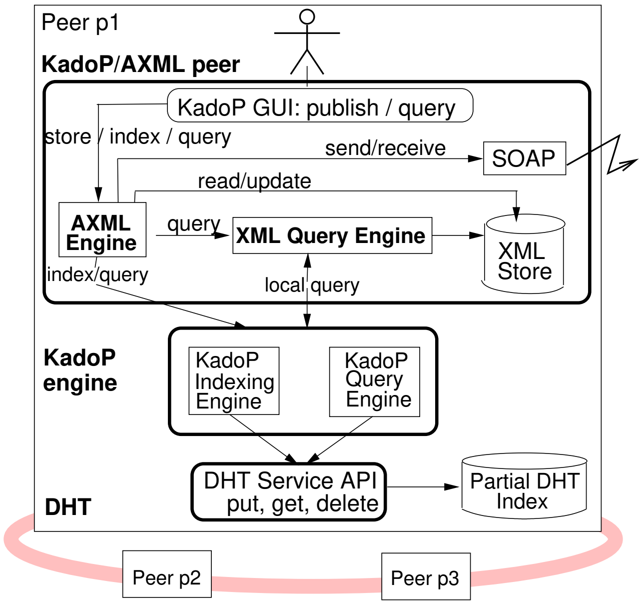

Figure 7: KADOP architecture.

4 The core of the KADOP system

In this section, we outline the architecture of our system, and detail two main aspects of this work: indexing and query processing. More details on the architecture, implementation, and a performance study can be found in [8].

The overall architecture of KADOP is outlined in Figure 7. It consists of several superposed layers: the DHT, the KADOP engine, and the KADOP/AXML peer.

The DHT is responsible for keeping the network of peers connected, in the event of peers joining or leaving the network, peer failures etc. This layer provides a simple API allowing inserting and deleting (key, value) pairs, and looking up a given key. In order for the DHT to function, each peer stores a fraction of the key-value pairs defined in the network. The DHT distributes the *resource index*, but the resources themselves remain on the originating peers.

The KadoP/AXML peer is an AXML application used to manage KADOP data items and semantic concepts. As any AXML application, it relies on an XML store, an XML query engine, and an AXML engine. It provides users with a GUI for publishing, editing, querying and navigation.

All information about data items (possibly after XML or SOAP wrapping) and semantic concepts defined on the peer are stored in the XML store; this also concerns collections and views defined on the peer. Some of the information may be partially intensional; we will discuss this further. Web service messaging is provided by the Axis SOAP engine [29]. The AXML engine is in charge of scheduling and executing Web service calls, for instance, re-indexing peer documents that have changed. It interacts with the XML store to maintain the AXML documents, and controls the evaluation of XML queries against this store [2].

The KadoP engine mediates between AXML and the DHT. Its indexing and query engine are crucial for the functioning of the whole platform.

4.1 Indexing in KADOP

Publishing a resource on a KADOP peer means sharing the information about the resource with the other peers. To that effect, the indexing engine extracts from each resource a set of *(key, value)* pairs, describing the resource's contents, and publishes them into the DHT. The pairs must compactly encode the complex structure of information, and at the same time support efficient KADOP query processing.

An essential component of indexing is the identification of items. The IDs of documents, views, and collections (also serialized as documents) consist of the peer's physical ID, and (an integer code of) the respective XML document. IDs of sub-trees, words, and abstract concepts (stored as XML elements) also include a structural ID identifying the collection within its document. Structural IDs are of the form [pre, post, depth], commonly used in XML query processing [6, 14, 18, 19]. We assign consecutive pre and post numbers uniformly to words and sub-trees. For concrete concepts, the ID contains the peer ID, (an integer code of) the enclosing DTD file, and the concept name.

For each data item of type t, identified by *ID*, and related to a concept *ns:c*, we publish (*ns:c*$||$*t, ID*) in the DHT, where $||$ stands for concatenation. Observe that we introduce the type information in the key. This is because we often need to retrieve items of specific types. The possible types are the six from Figure 2; we use, e.g., t_{wd} to denote the word type, t_{st} for the subtree type etc. Furthermore, we introduce: intensional types, e.g. $t_{int\text{-}st}$ for intensional sub-trees encountered in R_{out} documents, and variable types, e.g., for subtrees encountered in R_{in} documents. The set of possible types is still very small, so it can be very compactly encoded.

Users often omit to specify namespaces in queries. Consider some concept, say *person*. To allow retrieving the items related to *person*, we could publish a pair *(person,ID)* for each such item. However, all such pairs would accumulate in the partial index of a single peer, and degrade DHT performance. Instead, we take the following more efficient approach. For each concept *c* and for each namespace *ns* where this concept occurs, we publish in the DHT the pair *(c,ns*$||\mathcal{T}$*(ns:c))*, where $\mathcal{T}$*(ns:c)* is the set of types in which it occurs. The $\mathcal{T}$ sets will be used during query processing.

Indexing relationships between data model items is more complex. To limit the index size, we do not index in the DHT the implicit *partOf* relationships between sub-trees and their parent/ancestor subtrees and document, or between words and their enclosing subtrees. Neither do we index *relatedTo* relationships between nodes/words and their associated concrete concepts. Such relationships will be directly handled by the KADOP query processor.

concept	**ns**$||\mathcal{T}$**(ns:concept)**
member	GemoDTD$||\{t_{st}\}$
paper	GemoDTD$||\{t_{st}\}$
person	Gemo$||\{t_{st}\}$
person	getPaperDTD$||\{t_{st}\}$
PersonPubs	Gemo$||\{t_{st}\}$
title	GemoDTD$||\{t_{st}\}$
title	getPaperDTD$||\{t_{st}\}$
ns:concept$||t$	**ID**
GemoDTD:member$||t_{st}$	GPID$||$/Gemo.xml$||[2, 15, 1]$
GemoDTD:name$||t_{st}$	GPID$||$/Gemo.xml$||[10, 14, 2]$
GemoDTD:Alice$||t_{wd}$	GPID$||$/Gemo.xml$||[11, 13, 3]$
Gemo:PersonPubs$||t_{st}$	GPID$||$/Gemo.xml$||[3, 10, 3]$
DblpDTD:authorPubs$||t_{st}$	DPID$||$dblp.xml$||[50, 70, 1]$
Gemo:PersonPubs$||t_{st}$	GPID$||$/myGetPap.xml$||[1, 26, 0]$
getPapersDTD:person$||t_{st}$	GPID$||$/myGetPap.xml$||[10, 8, 2]$ $||$**$X7**
getPapersDTD:paper$||t_{int\text{-}st}$	GPID$||$/myGetPap.xml$||[24, 16, 2]$ $||$**Gemo:getPapers($X7)**
getPapersDTD:**$X7**$||t_{var\text{-}st}$	GPID$||$/myGetPap.xml$||[10, 8, 2]$
getPapersDTD:title$||t_{int\text{-}st}$	GPID$||$/myGetPap.xml$||[15, 17, 3]$ $||$**Gemo:getPapers($X7)**
isA$||$**ns**$_1$**:c**$_1$	**ns**$_2$**:c**$_2$
isA$||$Gemo:PersonPubs	DblpDTD:authorPubs
isA$^{rev}||$**ns**$_2$**:c**$_2$	**ns**$_1$**:c**$_1$
isA$^{rev}||$DblpDTD:authorPubs	Gemo:PersonPubs

Table 1: Sample *(key,value)* pairs extracted by KADOP.

For other relationships, *isA*, *input*, *output* and for the non implicit *partOf* and *relatedTo* relationships, we now illustrate how this is achieved. Consider the *isA* relationship between two concepts ns_1:c_1 and ns_2:c_2. Our indexing scheme must allow both finding specializations of ns_1:c_1, and generalizations of ns_2:c_2, in other words, traversing the relationship in both directions. Thus, we create who pairs:

$$(isA\|ns_1{:}c_1,\ ns_2{:}c_2) \quad (isA^{rev}\|ns_2{:}c_2,\ ns_1{:}c_1)$$

where special bit codes *isA* and isA^{rev} represent the relationships *isA* and the reverse one. Publishing the first pair allows finding specializations of ns_1:c_1, while publishing the second one allows finding generalizations of ns_2:c_2.

Table 1 shows some of the pairs published by KADOP from the sample instance depicted in Figure 3. GPID identifies the Gemo peer that also publishes the context document; and and DPID, the DBLP peer. For readability, we used the full XML document names; keep in mind that only document numbers are actually inserted in the DHT. The last rows in the table index the statement DblpDTD:authorPubs *isA* Gemo:personPubs.

Indexing representative and context documents Such documents are indexed just like regular documents, except that we introduce extra annotations to support query processing. For instance, consider in Table 1 the indexing of the context document for *Gemo:getPapers* of Figure 5. Observe that the variables that relate the service call inputs and outputs in context documents are given unique variable IDs, e.g., $X7. In this namespace, the fact that the parameter $X7 of the *getPapers* service may be instantiated to a <person> is captured by appending the ID of the parameter, to the value in the index entry for getPapersDTD:person. Similarly, the fact that, for instance, <paper>s or <title>s may be retrieved by this service call is captured by publishing in the index, entries for the context document. Observe that such entries also include information about the service call and its parameters.

The total size of the DHT pairs is roughly linear with the total size of $\mathcal{I}$. Structural identifiers require a space occupancy in $O(N\ log_2 N)$, but at the same time, a key is only stored once, for all its associated values. The specific contribution of KADOP for reducing the index size, however, is based on the usage of intensional information, as we explain in Section 5.

4.2 The query engine

We now describe KADOP query processing is based on an index built as previously described. Queries are processed in successive stages.

(1) Scoping. We first seek to refine the query's tree patterns, by finding the possible types and namespaces associated to query nodes. To this purpose, we issue in parallel several lookup calls to the DHT, as follows.

For every node with a *relatedTo* *:c test, *get(c)* call, to retrieve all namespaces *ns* associated to *c*, and the sets $\mathcal{T}$(*ns*:*c*). We do exactly the same for nodes with *value*=v tests; thus, from now on, we explain query processing based on *relatedTo* tests, but the same applies for value tests. This step produces a set of concepts (with namespaces) $\mathcal{C}(n)$, and if, for a node n, $\mathcal{C}(n)$ is empty, the node cannot be embededed in $\mathcal{I}$, and query processing stops.

(2) IsA enhancement. For every pattern node n and precise concept $ns{:}c \in \mathcal{C}(n)$, we issue *get*($isA^{rev}\|$*ns*:*c*) calls to find *ns*:*c* specializations, and add the resulting concepts to $\mathcal{C}(n)$. We continue the backward traversal of *isA* relationships from the concepts in $\mathcal{C}(n)$, until a fixed depth (specified in the system initialization); the processor keeps track of the concepts that are encountered, to avoid cycles. All concepts thus gathered are added to $\mathcal{C}(n)$.

(3) Refinement and pruning. Each query tree pattern is reformulated into a union of *refined tree patterns*. A refined tree pattern is a copy of the original tree pattern, but refined pattern nodes carry more information. Namely, a refined pattern node n is annotated with: all the tests associated to n in the original query; a concept *ns*:*c* $\in \mathcal{C}(n)$; and a data item type $t \in \mathcal{T}$(*ns*:*c*), only if type t is compatible with the query-specified conditions on n. For instance, if n has an *input* condition, then its type can only be Web service; if n has a *type=sub-tree* test in the query, then only t_{st} can be associated to n in a refined pattern.

From a tree pattern tp such that $N_t = \{n_1, n_2, \ldots, n_k\}$, we obtain a set of refined tree patterns whose size is bounded by: $\Pi_{i=1,2,\ldots,k}(\sum_{ns:c\in\mathcal{C}(n_i)}(|\mathcal{T}(ns:c)|))$.

Not all refined patterns may have embeddings (and thus, contribute to $q(\mathcal{I})$). We check two necessary conditions.

First, refined pattern node types must be coherent with the *partOf* relationships between the nodes. For instance, if the node n_1 is a child of n_2 in a refined tree pattern rtp, n_1 is of type collection, while n_2 is of type node, then rtp cannot have an embedding. All refined patterns whose type annotation, together with the pattern structural relationships, contradicts the KADOP data model, are discarded.

Second, the namespaces associated to related nodes and words must coincide. Indeed, only nodes and words from the same document can be related by *partOf*; thus, they must share a namespace. A refined pattern rtp where n_1 is of type node, n_2 is a child of n_1 and is of type node or word, and n_1, n_2 have different namespaces, is discarded.

The above pruning steps may still leave patterns for which no embedding is found, but they are *safe*: only patterns without any embeddings are eliminated.

(4) Refined tree pattern evaluation. In each refined pattern, some nodes are annotated with a precise concept, e.g. $ns{:}c$. For each such nodes, and associated type t, we issue *get*($ns{:}c \| t$) calls to obtain item ID lists. In particular, for word and sub-tree types, structural ID lists are returned, sorted by the document ID and then by the pre ID number.

The lists are combined by a single efficient holistic twig join [11], minimizing intermediary join results.

For the remaining nodes in the pattern, the query has specified URIs(IDs) directly. We check the remaining *partOf* relationships between these items and those retained by the join, by issuing *get*($partOf \|$*ID)* calls to verify the required *partOf* relationships between them and the IDs resulting from the join. Again, chains if *partOf* relationships are only explored up to a bounded depth.

(5) Value joins are applied on the refined pattern results.

(6) Data fetch. Once the result of the join is available, for each ID associated to n_{ret} (a result node), we issue a Web service call *getItem(ID)* directly to the peer in charge of this data item, to retrieve the corresponding item.

Variation: query pushing The above evalation steps are performed by the peer receiving the query, say p_1. If the peers owning useful data items have query evaluation capabilities (as it is the case for KADOP peers), then Step (4) can be pushed to peer where data resides.

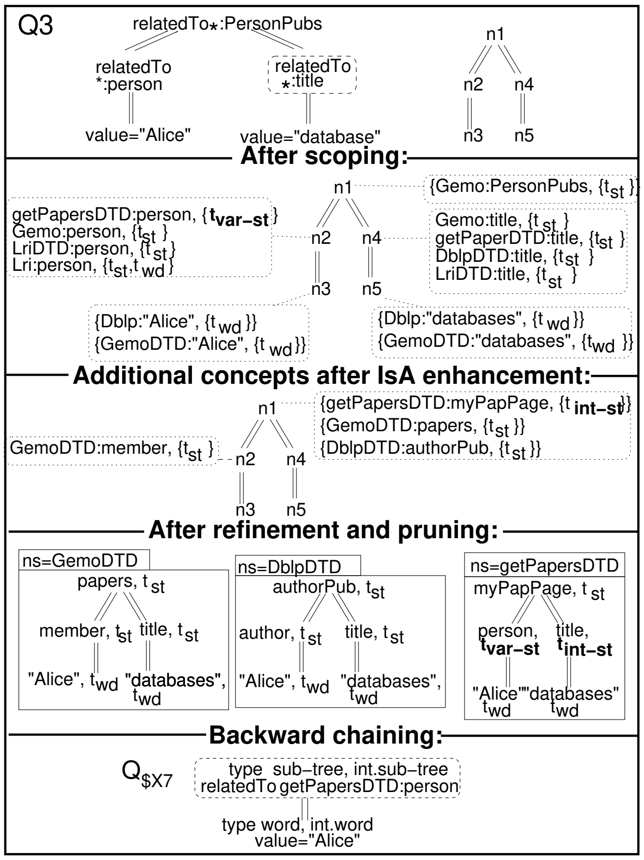

Figure 8: Main stages in processing Q_3.

Consider, for instance, a peer p_2 containing a lot of items in the *ns* namespace, and a pattern query over *ns* concepts. Instead of processing the query at p_1, one could push it to p_2, and thus avoir transferring the large ID lists to p_1. This is a classical distributed query optimization technique [26].

Handling intensional data. We now explain how KADOP exploits intensional data, and in particular, Web service contexts. In some refined pattern, we may obtain for some node n, an intensional data type, such as $t_{int\text{-}word}$ (intensional text). Similarly, we may find some variable data type, such as $t_{var\text{-}st}$ (variable subtree). However, in the pruning stage, we do not discard a refined pattern if it is missing some match under a node of variable or intensional type; the intuition is that matches for such nodes *may still be obtained* later in the process, e.g., as the result of a service call, even if they do not (yet) exist at this stage.

To evaluate a refined tree pattern involving intensional data types, we start by obtaining IDs for all nodes annotated with precise concepts, just like for regular data. Since variable and intensional items are given regular structural IDs (as shown in Table 1), we are able to verify, via the holistic twig join, that variable and intensional items satisfy the structural relationships required by the query.

At this stage only, the engine starts exploring and materializing intensional results. As Table 1 shows, intensional item IDs contain information about the service calls to activate in order to materialize the items. Two cases may occur. (i) If the service call parameters are fully materialized, the engine activates the call, inserts the result in a temporary document, and applies all query-specified predicates on the result. (ii) If some service call's parameters are variables, the engine must instantiate these variables before making the call. Let $Z be such a variable. Values for $Z must be related to the concrete WSDL-derived concept associated to $Z; thus, an *auxiliary query* asking for all sub-trees related to this concept is issued. Furthermore, the query might have also specified conditions on $Z; in this case, the conditions are also added to the auxiliary query. Such variable instantiations bring the engine to (i) above.

Notice that an auxiliary query may return intensional items. In such case, a step like (i) or (ii) above is triggered anew for such items, and so on recursively. This recalls the classical ***backward chaining*** algorithm for evaluating logical programs, set here in an XML-based peer-to-peer environment. Each such chain of service invocations is followed until no intensional data remains to be evaluated. Observe that there is no guarantee of termination. In KADOP , we consider only chains of bounded depth.

Subsequently, value-based joins, and fetching of the (non-intensional) items matching n_{ret} take place. (The intensional ones have already been materialized.)

Example. Figure 8 outlines the evaluation of query Q_3 previously introduced, on our sample instance (Figure 3 and Figure 5). We recall Q_3, next to its simplified form, wherein nodes are denoted as n_1, n_2 etc. After scoping, a dotted box connected to each n_i shows the set of precise concepts $\mathcal{C}(n_i)$, and for each *ns:c* in this set, the types $\mathcal{T}(ns{:}c)$. Notice the variable type associated to n_2. *IsA* enhancement yields new concepts and type sets for n_1 and n_2, e.g., next to n_1, an intensional subtree type has been obtained, due to the context document for *Gemo:getPapers* (Figure 5). Overall, 4 concrete concepts have been found for n_1, 5 for n_2, 2 for n_3, 4 for n_4, and 2 for n_5, which yields a total of 320 refined patterns. Of these, only 3 survive pattern pruning, in the *GemoDTD*, *DblpDTD*, respectively, *getPapersDTD* namespaces. The former two are finalized by pattern evaluation and data fetch.

The third one is more interesting since n_4 has intensional type. The engine calls *get(GemoPapersDTD:title*$\|t_{int\text{-}st}$*)*, and obtains the ID [24,16,2] (Table 1), plus the call to *Gemo:getPapers($X7)*. Similarly, the call to *get(GemoPapersDTD:person*$\|t_{int\text{-}st}$*)* produces the ID [15,17,3] and $X7. After verifying the required structural conditions on these two IDs, and [1,26,0] obtained for the extensional element myGetPapers, the intensional papers must be materialized. To that effect, the additional query $Q_{\$X7}$ is issued, which also applies the *"Alice"* test from Q_3; the engine has understood that n_2 matches are to be used as inputs to obtaind n_4 matches, thanks to the the common variable $X7. The answer to $Q_{\$X7}$ includes the member sub-tree from "gemo.xml", corresponding to Alice. The engine must align this sub-tree to the required structure of the person sub-tree corresponding to $X7. In the example, this simply means erasing all children of the sub-tree but the name; in general, data alignment may be more complex. If alignment fails, then the third refined pattern is discarded alltogether.

Assume now that the answer to $Q_{\$X7}$ contains intensional item IDs, corresponding, for instance, to the R_{out} document of a service *Gemo:getMembers()* returning members of the Gemo project. In this case, backward chaining will continue, retrieving the members, then selecting Alice among the members etc.

5 Intensional information in P2P

A main novelty of KADOP is its use of intensional/active information supported by AXML [3]. While detailing KADOP internals, we already encountered some uses of active or intensional information. We now step back to a more general level and summarize the benefits that intensional information brings to

Technique	*Applications*
Dynamic publishing	(1) Adapting to data dynamicity (2) Adapting to network dynamicity
Caching Web service results	(3) Optimizing DHT-based XML queries
Representative documents	(4) Web service discovery and composition (5) Indexing intensional data
Service contexts	(6) Integrated P2P querying of intensional and extensional data (7) Bridges to other data sources
Intensional DHT entries	(8) Index size reduction

Table 2: Techniques enabled by intensional information usage, and some of their applications.

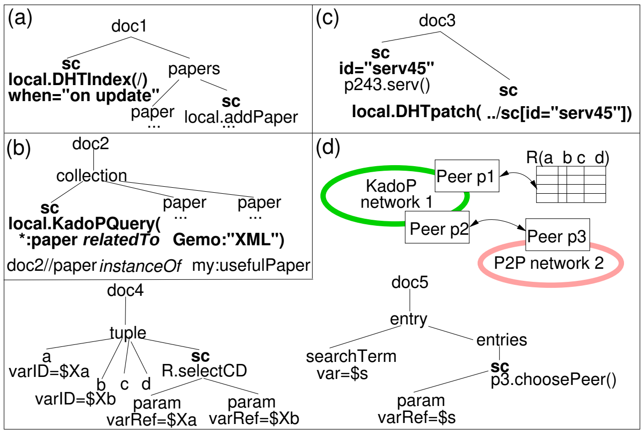

Figure 9: Intensional techniques in structured networks.

a structured P2P network. New techniques, enabled by the usage of intensional information, and their applications, are summarized in Table 2. We briefly discuss them next.

The dynamic nature of AXML is essential to control two main facets of changes: changes to the information and changes to the network of peers. **(1)** Using AXML, the system may rapidly adapt to *change in the local data and knowledge*. In particular, AXML may be used to refresh the DHT index for a particular item, when the item changes. For example, in Figure 9(a), the document may change as the result of a user update or when the call to *addPaper* is activated and new papers are added. When such an update occurs, the call to *local.DHTindex()* is activated, and the document is re-indexed. Furthermore, data items may be dynamically defined. For instance, the collection in Figure 9(b) is defined by a call to a KADOP query retrieving all papers related to Gemo:XML. (The activation of this call, not given in the figure, may be specified as, say activate daily.) The statement under the document declares that all (present and future) papers in doc_2, defined as a view over doc_2, are instances of the concept my:usefulpaper.

(2) Let us now consider *network changes*. Note that doc_3 in Figure 9(c) contains a call to the service *p243.serv*. If the peer *p243* becomes unavailable, evaluating the call yields an error. When this happens, the second service call patches the first one: it searches the DHT for a similar service (e.g., with the same name, and same representative documents), and replaces *p243.serv* with such a service.

(3) Now consider another aspect of AXML, namely *the caching of Web service results*. AXML allows specifying how long the result of a call remains valid. During this period, the call needs not be activated again, instead, the previous result is used. In the context of KADOP query processing, Web service calls (in particular, calls to the DHT) are issued in the scoping, *isA* enhancement, and data fetch stages. For instance, doc_2 includes a call to a KADOP query service. The results of the calls entailed by processing this query could be cached for some time, e.g., in the document itself (hidden to the user); this would avoid repeating the DHT lookup, as long as the respective values are considered valid, leading to important performance benefits.

(4),(5) and (6) were already discussed in Section 4.2.

(7) One most useful application of Web service contexts is the possibility to provide bridges between a KADOP network and other data sources, such as a relational database, or another P2P network. Figure 9(d) illustrates this feature. Peer p_1 in a KADOP network is connected to a relational data source $R(a, b, c, d)$. To give other peers access to R, p_1 could export R as an XML document and index it in the DHT. However, this would be very time and space-consuming. Furthermore, p_1 may want to disallow scanning R, for confidentiality or performance reasons, and instead allow other peers to access it only if they provide values for the a and b attributes. To efficiently share this information about R, p_1 may publish the Web service context document doc_4, which states: for every pair of a, b values, a call to *selectCD* returns corresponding c and d values.

(7-continued) The same technique applies if we consider *another structured network*, not necessarily KADOP, as an external source of information, as in Figure 9(d). Assume KADOP peer p_2 knows a peer p_3 from the second network, and wants to make this second network contents available on the KADOP network, *without replicating* the entire index of the second network. p_2 can achieve this by publishing in the KADOP network the context document doc_5, stating that entries for any term $s may be found in the second network. To avoid overloading p_3 with searches from the KADOP network, p_2 inserts in doc_5 (and in the KADOP DHT) a call to *p3.choosePeer*, that chooses a random peer in the second network for every new search. will thus bridge to different peers in the second network.

(8) Last but not least, a simple, but very useful application of intensional data is building *intensional DHT entries*. The idea is the following. For any peer p, holding several items related to key *k*, instead of inserting all corresponding *(k,ID)* pairs in the DHT, we only insert *(k,p.getItemIDs(k))*. Thus, we replace a potentially large ID list with a single (intensional) value. This last technique is very effective in reducing the index size, especially for popular DHT keys. We demonstrate this in [8].

6 Experimental validation

In this section, we briefly report on the status and indexing performance of the KADOP system. For space reasons, we delegate other experiments, in particular regarding querying, simulated in a larger network of peers, to [8].

The platform. KADOP is implemented in Java based on FreePastry [32] DHT, and on the AXML platform for PCs [30]. The indexing and query engine are fully working, with the exception of the backward chaining functionality, whose components are implemented, but not yet integrated. KADOP will be demonstrated shortly [7]. We report results obtained over 10 actual peers deployed in a LAN.

Data sets. We used several collections of XML documents: (i) the **DBLP.xml** document, of size **130 Mb**, with the well-known bibliographic data; (ii) an **IEEE-CS** collection of $2,300$ computer science XML-ized articles, of a total size of **55 Mb**; and (iii) the **QMUL.xml** bibliographic data from the Queen Mary U. of London. These are all real data sources, used in the INEX 2004 [17] benchmark initiative. These collections are representative for a large variety of XML documents. QMUL is very regular; in fact, it is an XML-ized relation. DBLP structure is more complex, and IEEE-CS is extremely complex, due to the complex nesting of presentation tags such as section, paragraph etc. DBLP and QMUL illustrate "data-centric" XML, while QMUL is representative for document-centric applications.

Indexing. To obtain a scenario where several peers share data, we have split the (single) DBLP document into 320 individual files, each with several bibliographic entries (the IEEE-CS data is already split in $2,300$ files), and distributed them over the peers. The resulting total index sizes, over all peers, are reported in the following table.

Data set	*Size*	*# Pairs*	*Index size*
DBLP	130 Mb	$1,433,142$	22.20 Mb
IEEE-CS	55 Mb	$5,743,730$	120 Mb
QMUL	1 Mb	$131,975$	0.6 Mb

Notice that the index-to-document size ratio is much smaller for regular QMUL and DBLP data, featuring words from a restricted vocabulary, than for IEEE-CS, which is full-text. The above index size makes no optimization, and even indexes e.g., stop words. Eliminating these would clearly reduce the index resulting from IEEE-CS.

We have measured the *increase in the number of keys* as more and more data was indexed in the network. The results for the more interesting DBLP and IEEE data sets are reported in Figure 10. We plot the increasing ratio of the total number of keys created in the system, as a function of the quantity of data indexed. The curve corresponding to DBLP is almost linear, while the IEEE-CS curve levels off faster. This illustrates different behavior of data-centric, respectively, text-centric documents: the former contain structured data, and every record has some new values; in contrast, keys extracted from IEEE-CS articles reflect natural language and thus increase less starting from some point. As expected, an overwhelming majority of keys

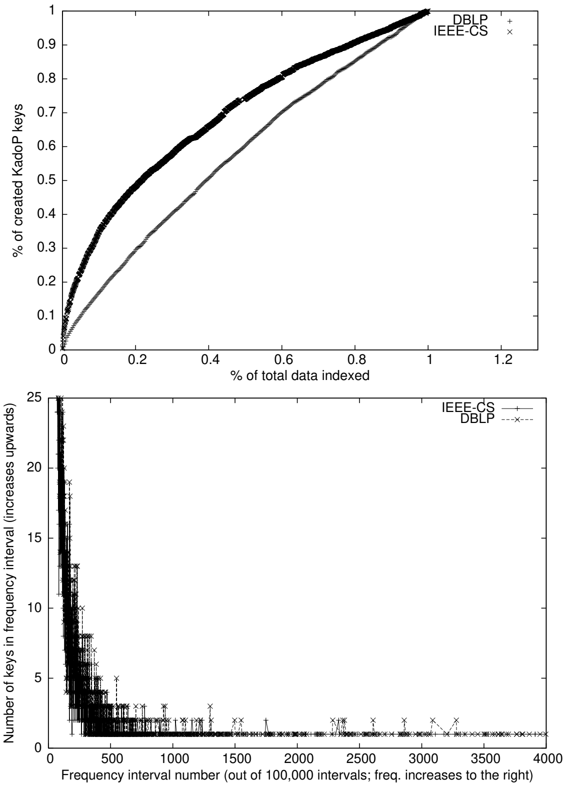

Figure 10: Incremental evolution of the number of KADOP keys (top); key frequency distribution (bottom).

comes from individual words. DBLP only produces 38 concrete concepts, and IEEE only

We now study key frequency in the IEEE and DBLP data sets. The maximum frequency is $315,000$, respectively, $97,000$. In the IEEE set, the most frequent keys include stopwords, whose indexing should be avoided alltogether, such as *a*, *the* etc. On the contrary, the top 20 keys on DBLP include year values, and useful concrete concepts such as *conference*, *journal* etc.

We have also analyzed the frequency distribution of KADOP keys in the the IEEE-CS and DBLP data sources; the results are in the lower part of Figure 10. For each data set, we have divided the frequency interval $[0, f_{max}]$ in $100,000$ equal-width intervals (i.e., each interval has length $f_{max}/100,000$). The numbers of these intervals are on the x axis, with low frequencies to the left. The y dimension corresponds to the number of keys whose frequency falls in the corresponding interval. We only plot the lower-left corner of the curve for readability. Notice the skewed distribution: only the lowest $1,000$ intervals (thus, only in the lowest 1% of all frequencies) contain more than one key. Turning the entries for the top 50 DBLP keys into intensional entries would eliminate $1,000,000$ entries.

7 Comparison with related works

Data-sharing peer-to-peer platforms Recent works have developed P2P data management platforms; PeerDB [22], Piazza [27], and SomeWhere [16] are built on unstructured networks, unlike our work. P2P XML management application based on a DHT is addressed in [15]. This work improves the basic partial index structure maintained by the DHT on a peer, to fetch IDs on a given XML path (as opposed to tags in KADOP). This technique could also be adopted in KADOP, however, joins would still be needed in the presence of tree pattern queries. [15] is also concerned with peer overloading due to popular keys, and recommends *splitting* index entries over neighbor peers. This could also be adopted in KADOP, which, however, relies on intensional indexing to reduce or eliminate keys above a certain frequency treshold. In [20], transferring large ID lists on DHT networks is shown to be a performance problem, and a hybrid model is proposed, with frequent keys handled by a unstructured network, and rare ones in the DHT. To address the same problem, KADOP applies intensional indexing, replacing large ID lists with a few calls to peers owning the respective items. The most important innovation of KADOP is the usage of intensional information; we outlined benefits in Section 5.

Previous works on AXML Several aspects of the AXML platform have been previously studied [2, 4, 1]. One specific contribution of KADOP consists of using DHT technologies to enable access of a document or service regardless of its location in the peer network. The second KADOP contribution is a query processing framework that dynamically discovers and calls Web services whose results, combine, may answer the query. While query optimization has been studied in [2] in a centralized setting, KADOP brings it truly to the Web scale, and show how to bridge separate information worlds based on semantics.

8 Conclusion and future work

We presented he KADOP system for sharing content in a P2P environment that combines DHT, AXML and XML query processing technology, in a unique framework. In Section 5, we discussed a number of advantages that the use of intensional information brings to the system. In particular, it offers a fine control of changes both in the available information and in the peer network. Also, it enables a number of query optimization techniques.

The implementation is in progress. At the present time, it is fully operational with respect to static data and knowledge. The implementation of the powerful functionalities brought by context documents is ongoing; more experimental results are described in [8]. We are also currently deploying KADOP on XtremWeb, a simulator of large WAN P2P systems built on a cluster of machines at U. Paris 11 [13].

XML query optimization for data distributed on a WAN opens up an array of new research issues. Techniques developped in a LAN context such as citeXyleme often present serious drawbacks when communications are slow. We already mentioned a number of optimization techniques that are implemented in KADOP . We intend to introduce some of the techniques described in [5, 1] in KADOP .

As already mentioned, the semantic model of KADOP is rather simple. Although we do handle knowledge, the focus was more on the use of intensional information and on query optimization. It would be interesting to consider more sophisticate semantics models, e.g., in the style of [9], in the context of KADOP .

Some issues ignored here deserve consideration in a P2P setting. First, we need to consider more information-retrieval style queries [10] and in particular K-ranked queries, e.g., [12]. Also, in the actual system, there is no control of publication. In particular, any peer may publish data about a concept from any namespace, possibly against the will of the owner of that namespace. One should consider the possibility of selectively limiting access rights and the issue of trust.

References

[1] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems - datalog to the rescue! In *ACM PODS*, 2005.

[2] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *ACM SIGMOD*, pages 227–238, 2004.

[3] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: A data-centric perspective on Web services. In *Web Dynamics*, pages 275–300, 2004.

[4] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *ACM PODS*, pages 35–45, 2004.

[5] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *ACM SIGMOD*, 2003.

[6] S. Abiteboul, S. Cluet, G. Ferran, and M.-C. Rousset. The xyleme project. *Computer Networks*, 39:225–238, 2002.

[7] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and querying warehouses of XML resources. In *ICDE*, 2005.

[8] S. Abiteboul, I. Manolescu, and N. Preda. Sharing content in structured P2P networks (extended version), 2005. Available at http://www-rocq.inria.fr/ npreda.

[9] P. Adjiman, P. Chatalic, F. Goasdou, M.-C. Rousset, and L. Simon. Distributed reasoning in a peer-to-peer setting. In *European Conference on Artificial Intelligence*, 2004.

[10] R. Baeza-Yates. *Modern Information Retrieval*. Addison-Wesley, 1999.

[11] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *ACM SIGMOD*, 2002.

[12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *ACM PODS*, 2001.

[13] G. Fedak, C. Germain, V. Néri, and F. Cappello. XtremWeb : A generic global computing system. CCGRID, 2001.

[14] T. Fiebig, S. Helmer, C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *The Very Large Databases Journal*, 11(4):292–314, 2002.

[15] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.

[16] F. Goasdoue and M.-C. Rousset. Answering queries using views: a KRDB perspective for the Semantic Web. *ACM TOIT*, 2003.

[17] Initiative for the evaluation of XML information retrieval. http://inex.is.informatik.uni-duisburg.de:2004.

[18] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, , and C. Yu. Timber:a native XML database. *The Very Large Databases Journal*, 11(4), 2002.

[19] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, 2001.

[20] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P file-sharing with an internet-scale query processor. In *VLDB*, 2004.

[21] L. Mignet, D. Barbosa, and P. Veltri. The XML web: a first study. In *WWW*, 2003.

[22] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *ICDE*, 2003.

[23] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *ACM PODS*, pages 105–112, 1995.

[24] A. Rowstron and P. Druschel. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[26] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1999.

[27] I. Tatarinov and A. Halevy. Efficient query reformulation in peer-data management systems. In *ACM SIGMOD*, 2004.

[28] Universal Description, Discovery, and Integration of Business for the Web (UDDI). http://www.uddi.org.

[29] The AXIS SOAP engine. www.apache.org.

[30] ActiveXML home page. www.activexml.net.

[31] Web Ontology Language. www.w3.org/TR/owl-ref/.

[32] The FreePastry system. www.cs.rice.edu/CS/Systems/Pastry/FreePastry/.

[33] The resource description framework. www.w3.org/RDF.

[34] Naming and addressing: URI, URLs. http://www.w3.org/Addressing/.

[35] The Web Service Definition Language. www.w3.org/TR/wsdl.

[36] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery.