The ViP2P Platform: XML Views in P2P

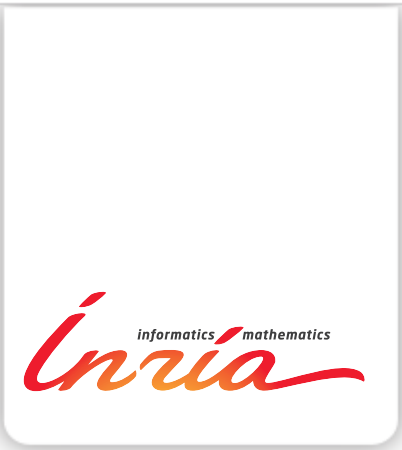
Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, Spyros Zoupanos

The ViP2P Platform: XML Views in P2P

Konstantinos Karanasos, Asterios Katsifodimos,
Ioana Manolescu, Spyros Zoupanos

RESEARCH REPORT
N° 7812
November 2011
Project-Team Leo

ISSN 0249-6399 ISRN INRIA/RR--7812--FR+ENG



The ViP2P Platform: XML Views in P2P

Konstantinos Karanasos*, Asterios Katsifodimos*,
Ioana Manolescu*, Spyros Zoupanos*†

Project-Team Leo

Research Report n° 7812 — November 2011 — 39 pages

Abstract: The growing volumes of XML data sources on the Web or produced by enterprises, organizations etc. raise many performance challenges for data management applications. In this work, we are concerned with the distributed, peer-to-peer management of large corpora of XML documents, based on distributed hash table (or DHT, in short) overlay networks. We present ViP2P (standing for *Vi*ews in *P*eer-to-*P*eer), a distributed platform for sharing XML documents based on a structured P2P network infrastructure (DHT). At the core of ViP2P stand *distributed materialized XML views*, defined by arbitrary XML queries, filled in with data published anywhere in the network, and exploited to efficiently answer queries issued by any network peer. ViP2P allows user queries to be evaluated over XML documents published by peers in two modes. First, a long-running subscription mode, when a query can be registered in the system and receive answers incrementally when and if published data matches the query. Second, queries can also be asked in an ad-hoc, snapshot mode, where results are required immediately and must be computed based on the results of other long-running, subscription queries. ViP2P innovates over other similar DHT-based XML sharing platforms by using a very expressive structured XML query language. This expressivity leads to a very flexible distribution of XML content in the ViP2P network, and to efficient snapshot query execution. ViP2P has been tested in real deployments of hundreds of computers. We present the platform architecture, its internal algorithms, and demonstrate its efficiency and scalability through a set of experiments. Our experimental results outgrow by orders of magnitude similar competitor systems in terms of data volumes, network size and data dissemination throughput.

Key-words: P2P, XML, DHT, distributed query processing, view-based query evaluation

* INRIA Saclay–Île de France and LRI, Université Paris Sud-11
† Max-Planck Institut für Informatik, Saarbrücken, Germany

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

Parc Orsay Université
4 rue Jacques Monod
91893 Orsay Cedex

La plate-forme ViP2P: vues XML en pair-à-pair

Résumé : Les grands volumes de données XML disponibles sur le Web, produites par les organisations ou individus posent des défis importants pour la gestion efficace de données. Ce travail est situé dans le contexte de la gestion de grands volumes de documents XML, dans un réseau décentralisé, distribué, pair-à pair, qui s'appuye sur une table de hashage distribuée (ou DHT). Dans ce rapport, nous présentons ViP2P (*v*ues en *p*air-à-*p*air), une plateforme distribuée pour le partage de documents XML s'appuyant sur un réseau de type DHT. Au cœur de ViP2P sont des vues matérialisées distribuées. Celles-ci sont définies par n'importe quel pair, sous la forme de requêtes XML. Dès que des données XML publiées par un pair quelconque correspondent aux définitions des vues, ces données seront utilisées pour contribuer au contenu des vues. ViP2P fournit deux scénarios d'évaluation de requêtes sur des documents XML. Il existe d'abord un mode "souscription", où une requête enregistréee dans le système reçoit des réponses de façon incrémentale, lorsque des données que l'on vient de publier contribuent aux résultats. En deuxième lieu, une requêtes peut être évaluée uniquement à partir des données déjà publiées, en réécrivant la requête à l'aide des vues matérialisées. Nous avons testé ViP2P déployé dans des réseaux distribués de plusieurs centaines d'ordinateurs. Dans ce rapport, nous présentons son architecture, ses principaux algorithmes, et démontrons son efficacité et son passage à l'échelle par une série d'expériences. Les résultats de nos mesures démontrent la robustesse de ViP2P jusqu'à des volumes de données, débit de dissemination de données, et tailles de réseau, allant au delà (jusqu'à plusieurs ordres de grandeurs) des mesures précédemment publiées sur des systèmes comparables.

Mots-clés : pair-à-pair, XML, THD, execution de requetes distribuée, evaluation de requêtes en terme de vues

1 Introduction

The volumes of data sources available in the form of XML documents has exploded since the W3C's 1998 standard, and so have the languages, tools and techniques for efficiently processing XML data. The interest of distribution in this context is twofold. First, a distributed storage and processing network can accommodate data volumes going far beyond the capacity of a single computer. Second, as organizations and individuals interact more and more, sharing and consuming one another's information flows, it is often the case that (XML) data sources are produced independently by several distributed sources. The set of producers and consumers of data related to a specific topic, e.g., IT journals, blogs and online bulletins, is not only distributed, but also dynamic: sources may join or leave the system, the set of information consumers or their topics of interest may also change in time etc. Thus, we are interested in the large-scale management of distributed XML data in a *peer-to-peer* (P2P) setting. To provide users with *precise, detailed and complete* answers to their requests for information, we adopt a database-style approach where such requests are formulated by means of a structured query language, and the system must return complete results. That is, if somewhere in the distributed peer network, an answer to a given query exists, the system will find it and include it in the query result. To achieve this, our goal is to build a P2P XML data management platform based on a distributed hash table (or DHT, in short [15]).

In this setting, users may formulate two kinds of information requests. First, they may want to *subscribe to interesting data anywhere in the network*, and published before or after the subscription is recorded in the system. Our goal is to persist the subscriptions and ensure that results are eventually returned as soon as possible following the publication of a matching data source. This is in the spirit e.g. of RSS feeds, but extended to a distributed network where the source from which interesting data will come is not a priori known. Second, users may formulate *ad-hoc (snapshot)* queries, by which they just seek to obtain as fast as possible the results which have already been published in the network.

The challenges raised by a DHT-based XML data management platform are:

- building a *distributed resource catalog*, enabling client producers and consumers to "meet" in the virtual information sharing space; such a catalog is needed both for subscription and ad-hoc queries,
- efficiently distributing the data of the network to the consumers that have subscribed to it and
- providing *efficient distributed query evaluation algorithms* for answering ad-hoc queries fast.

In this paper, we present ViP2P, standing for *Views in Peer-to-Peer*, a distributed P2P platform for sharing Web data, and in particular XML data. ViP2P is built on top of a structured P2P network infrastructure, and it allows each peer in the network to share data with all the other peers. Data sharing in ViP2P is twofold. First, each network peer can ask long-running queries which are treated as subscriptions, that is, they receive results if and when a document published in the system matches such queries. Second, once results are stored for such a subscription, they are treated as *materialized views* based on which subsequent ad-hoc queries can be processed with snapshot semantics, i.e., based

only on the data already published in the network. Given such an ad-hoc query, a ViP2P peer looks up the ViP2P network for relevant materialized views, runs an algorithm for equivalently rewriting the query, identifies and evaluates a distributed query evaluation plan which, based on the views, computes exactly the results of the query on the data published in the system prior to the query. ViP2P thus fills two kinds of needs: (*i*) disseminating information in a timely fashion to subscriber peers and (*ii*) re-using pre-computed results to process ad-hoc queries efficiently on the existing data only.

A critical issue when deploying XML data management applications on a DHT is the division of tasks between the DHT and the upper layers. The DHT software running on each machine allows peers to remain logically connected to each other and to look up data based on search keys: a small set of simple, light-weight operations. In contrast, powerful XML data management requires complex languages (such as the W3C's XPath and XQuery standards), and scalable algorithms to cope with complex processing and large data transfers (known to raise performance issues in any distributed data management setting).

Experience with our previous DHT-based XML data management platform KadoP [3] has taught us to load the DHT layer *as little as possible*, and keep the heavy-weight query processing operations in the data management layer and outside the DHT. This has enabled us to build and efficiently deploy an important-size system (70.000 lines of Java code), which we show scales on up to 250 computers in a WAN, and hundreds of GBs of XML data. ViP2P improves over the state of the art in DHT-based XML data management, since: (*i*) it is one of the very few systems actually implemented (together with [3, 33], and opposed to prototypes built on DHT simulators), (*ii*) is shown to scale on data volumes that are orders of magnitude beyond the cited competitor systems and (*iii*) has the most expressive XML query language, and the most advanced capabilities of re-using previously stored XML results, among all similar existing platforms [3, 9, 10, 18, 24, 25, 33].

ViP2P is part of a family of systems aiming at efficient management of XML data in structured peer-to-peer networks [3, 9, 10, 18, 24, 25, 33, 32]. The contributions of this work, with respect to the existing systems, are as follows:

- We present a *complete architecture for query evaluation, both in continuous (subscription) and in snapshot mode*. This architecture enables the efficient dissemination of answers to tree pattern queries (expressed in an XQuery dialect) to peers which are interested in them, regardless of the relative order in time between the data and the subscription publication. As in [25], it also allows to efficiently answer queries in snapshot mode, based on the content of the existing views materialized in the network, but using more expressive views, queries and rewritings.

- We have fully implemented our architecture (about 250 classes and 70.000 lines of Java code), based on the FreePastry [17] P2P infrastructure. We present a *comprehensive set of experiments performed in a WAN*, showing that (*i*) the performance of a fully deployed large-scale distributed system (and in particular a DHT-based XML management platform) is determined by many parameters, beyond the network size and latency which can be set in typical P2P network simulators and (*ii*) the ViP2P architecture scales to several hundreds of peers and hundreds of GBs of XML data, both unattained in previous works.

The paper is organized as follows. Section 2 surveys the state of the art in managing XML data in DHT networks. Section 3 introduces the ViP2P architecture via an example and describes its main modules. Section 4 presents query rewriting in ViP2P through a series of examples, while Section 5 concentrates on the materialization, indexing and look-up of materialized views, at the core of the platform. In Section 6, we present a set of experiments analyzing the performance of ViP2P data management in a variety of settings and demonstrating its scalability, then we conclude.

2 State of the art

In this section we present the current state of the art in XML data management over P2P networks. In Section 2.1 we focus on the differences of structured and unstructured P2P networks and the reasons behind our choice to use a structured P2P network for building our platform. In Section 2.2 we present our closest competitor works focusing on the management of XML data over structured DHT networks. Section 2.3 stresses the challenges of XML data management in a real, deployed platform as opposed to simulations. Finally, in Section 2.4, we present earlier publications of the ViP2P platform.

2.1 Structured vs. unstructured P2P networks

Peer-to-peer content sharing platforms can be broadly classified in two groups. Unstructured peer-to-peer networks allow arbitrary connections among peers, that is, each peer may be connected to (or aware of the existence of) one or more network peers of its choice. Such network structure typically mimics some conceptual proximity between peers interested in similar topics etc. Structured peer networks, on the other hand, impose the set of connections among peers. A survey of (structured and unstructured) P2P XML sharing platforms reflects the state of the art and open issues as of 2005 [24] and a more recent survey of XML document indexing and retrieval in P2P networks can be found in [2].

The different network structures impact the way in which searches (or queries) can be answered in the network. Thus, in unstructured networks, queries are forwarded from each peer to its set of known peers (or neighbors) and answers are computed gradually as the query reaches more and more peers. For instance, in [35] peers are logically organized into clusters that are formed on a document schema-similarity basis. The superpeers of the network are organized to form a tree, where each superpeer hosts schema information about its children. When a query arrives it is forwarded to the superpeers. Every superpeer performs location assignment: it examines the schemas of the documents of its children to detect which peers could possibly contribute results to the query. After the contributing peers have been located, the peer that originally posed the query builds a location aware algebraic plan and ships the corresponding subqueries to their respective peers. The results are then retrieved from each peer and the original query is evaluated by performing operations such as joins over the subquery results.

It is easy to see that if query answers reside on a peer very far (in terms of peer connections) from the peer where the query originated, this may lead to numerous messages and a long query response time. To improve the precision, performance and recall of query answering in this context, many approaches have been proposed, from the earliest [40] to the very recent [16], to name just a few.

In contrast, structured networks (and their best-known representatives, distributed hash tables or DHTs, in short [15]) provide a simple distributed index functionality implemented jointly by all the peers. The simplest DHT interface provides *put(key, value)* and *get(key)* operations allowing the storage of (key, value) pairs distributed over all the network peers. More advanced DHT structures also allow range searches of the form *get(key range)*, such as Baton [21, 22] or P2PRing [13, 14]. In a DHT, to answer a *get* request, a bounded number of messages are exchanged in the network, typically in $O(log_2(N))$, where N is the

number of network peers.

In this work, we consider the setting of a structured network, based on a DHT, and design an efficient platform for *XML query processing in large scale networks, based on P2P XML materialized views*. The main difference between most of the existing platforms and ViP2P is that our system addresses the whole processing chain involved in evaluating queries, as opposed to only locating the interesting documents and shipping the query to those peers for evaluation. The latter approach may, in some cases, require numerous messages at query evaluation time and possibly increased response times. ViP2P, in contrast, considers the complete chain of query processing based on materialized views incrementally built in the network. This enables answering queries by contacting only a few peers and possibly re-using complex pre-computed results, stored in the views.

2.2 XML data management based on DHTs

The first DHT-based platform for XML content sharing was described in [18]. This work proposed a framework for indexing XML documents, based on the parent-child element paths appearing in the document. Processing a query involves (*i*) extracting from the query a set of paths which could serve as lookup keys, (*ii*) obtaining via *get* calls the IDs of all documents matching the paths, (*iii*) shipping the query to all the peers holding such documents and (*iv*) retrieving the results at the query peer. The approach carries some imprecision in the case of queries featuring the descendant axis (//) or tree branches. For instance, the query `/a[b]/c` could be forwarded to documents in which the paths `/a/b` and `/a/c` occur, but the tree pattern `/a[b]/c` does not occur. A very similar approach to DHT-based XML indexing by parent-child paths is taken in [37].

The above discussion illustrates a common aspect in DHT-based content management platforms: imprecision in the indexing method leads to more peers being contacted to process a given query. A previous work on managing relational data based on DHTs [26] has shown that intensive messaging at query time may seriously limit scaling. Therefore, index precision is generally a desirable feature.

The work described in [9, 10] considers the setting where XML documents are divided in fragments distributed among several peers. Each fragment is assigned as identifier the parent-child label path going from the document root to the root of the fragment, and subsequently, fragments are indexed in the DHT by their identifiers. The system uses a particular DHT which can handle *prefix queries*, and thus allows locating XML fragments for which a prefix of the path from the root to the fragment is known. Processing linear queries using only the child axis is simple, however, simple queries using the descendant axis, such as the query `//a`, need to be forwarded to all the network peers.

The KadoP system [3] indexes XML documents at fine granularity. Thus, for any element name `a`, a network peer is in charge of storing the identifiers (or IDs, in short) of all `a`-labeled elements from all the documents in the network. The IDs reflect the position of the elements in the respective documents. Therefore, any tree pattern query can be answered by retrieving the list of IDs corresponding to each tree pattern node, and combining these lists via a holistic twig join [11]. This indexing model has very high precision, since the output of the holistic twig join includes exactly the documents matching the query. However,

the index is much more voluminous than in previous proposals [9, 10, 18, 37], highlighting the severe limitations *in terms of volume of the (key, value) pairs* of the DHT index. Several optimizations in the index structure were introduced in [3], based on which the KadoP platform was tested on hundreds of peers and 1GB of data.

More recently, the *psiX* system [32, 33] proposed an XML indexing scheme based on document summaries, corresponding to the backward simulation image of the XML documents (if a DTD is available, summaries can also be built based on the DTD). An algebraic signature is associated to each summary and to each query. When a query arrives, the algebraic query signature is used to look up in a holistic fashion all document signatures matching the query. The precision of this indexing scheme improves over KadoP [3] by a better treatment of wildcard ($*$) nodes, which KadoP ignores for the most part of query processing. From the matching summaries, one can identify the concrete corresponding documents, and then push query evaluation to the peers hosting the documents. The approach is implemented over the Chord DHT and shown to be effective by experiments on up to 11 peers in the PlanetLab network.

The main difference between the works described in [18, 32, 33, 37] and our work lies in the approach taken for query processing. These works, of which *psiX* [33] can be considered the most advanced, are only concerned with locating the documents relevant for a query. In contrast, [9, 10], KadoP [3] and the ViP2P platform presented here address the P2P XML query processing problem as a whole. They re-distribute data in the P2P network in order to prepare for the evaluation of future queries. KadoP distributes a tag index over the peers independently of the data and the queries, which can be seen as a "one size fits all" approach. ViP2P allows individual peers to choose the particular queries of interest for them, expressed in a rich tree pattern dialect (or, equivalently, a useful XQuery subset) and then allows exploiting the stored results of such queries as views for rewriting future queries. An ongoing development of ViP2P [12] focuses on automatically choosing the views to materialize on each peer in order to improve observed query processing performance. Thus, going beyond the problem of *locating* relevant documents, ViP2P aims at making the most out of the existing network storage and processing capacity in order to *evaluate queries* most efficiently to the peers that need them.

Closer in spirit to our work is the cooperative XPath caching approach described in [25], where peers can store results of a (peer-chosen or system-imposed) XPath query. The definitions of these stored queries (or views) are indexed in the network, enabling subsequent queries to be rewritten and answered based on these views. ViP2P is more general, since (i) our view and query language is an XQuery dialect with many returning nodes, as opposed to the simple XPath subset in [25] and (ii) our approach allows to rewrite a query based on *several* views, whereas [25] can only exploit one view for one query.

DHT-based XML indexing methods [3, 9, 10, 18, 32, 33, 37] are *complete*, i.e., for each query, based on the index, all relevant answers can be computed and returned. In ViP2P and [25], peer-chosen views replace the compulsory index fragments assigned by the network to each peer. Thus, it is possible that some queries cannot be processed due to the lack of appropriate views. Our focus in ViP2P is on efficiently building and exploiting pre-computed query results under the form of materialized views. To guarantee completeness, our approach can be coupled with an efficient and compact document-level index,

such as *psiX* [33], on which to fall back when no suitable views are found for a given query.

We conclude our analysis by considering the *granularity* or level of detail used to index XML, i.e., the granularity of the keys inserted in the DHT. Element labels (or label paths, or document summaries) have been often used. However, this does not allow efficiently locating documents which satisfy specific *value or keyword* search conditions, such as e.g., //item[price=$45] or //item[contains(.,'camera')]. Indexing by keywords or text nodes increases index precision but also significantly increasing the index size, since there are many more keywords in an XML document than distinct tags. Therefore, the approaches of [9, 10, 18, 32, 33, 37] cannot be easily extended to support keyword search and preserve their scalability. A *value summary framework* is proposed in [18] to index element values by trading off precision for index space. KadoP [3] indexes all keywords just like element labels, and proposes index-level optimization techniques to cope with important scale-related problems. ViP2P allows keyword and value conditions both in the materialized views and in the queries.

2.3 Managing XML on a DHT: platforms vs. simulations

Developing distributed systems, and in particular a P2P platform, requires significant efforts. This may be a reason why many previous works in this area validate their techniques based on *simulated peer networks*, where a single computer runs an analytical model configured to simulate a given network size. Our INRIA team has invested significant manpower (of the order of 70 man $\times$ month by now) developing the KadoP and then the ViP2P platforms. Our effort has taught us that many architecture and engineering problems arise due to the mismatch between the initial DHT goals (maintaining large dynamic networks connected and providing minimal messaging), and the data-intensive operations required by indexing, storing, and querying large volumes of XML data. We have addressed these problems in ViP2P by careful architecture and engineering, and report in this paper *experiments at a scale (in peers deployed over a WAN, and in data size) unattained so far by any other platform.* Thus, KadoP [3] scales up to 1 GB of data over 50 computers peers, psiX [33] used 262 MBs of data and 11 computers, and in this paper we report on sharing up 160 GB of data over up to 250 computers (in all cases, the computers were distributed in a WAN).

2.4 Previous publications on ViP2P

A first version of the platform was described in an informal setting (no proceedings) in an international workshop [30] and a national conference [29]. These works used a more restricted query language than we consider here, and described early experiments on a platform which has been much improved since. Two ViP2P applications have lead to demonstrations: P2P management of RDF annotations on XML documents [23] and adaptive content redistribution [12]. The details of view-based query rewriting in ViP2P are described in a separate paper [27]. They can be seen as orthogonal to the architecture and performance issues described here.

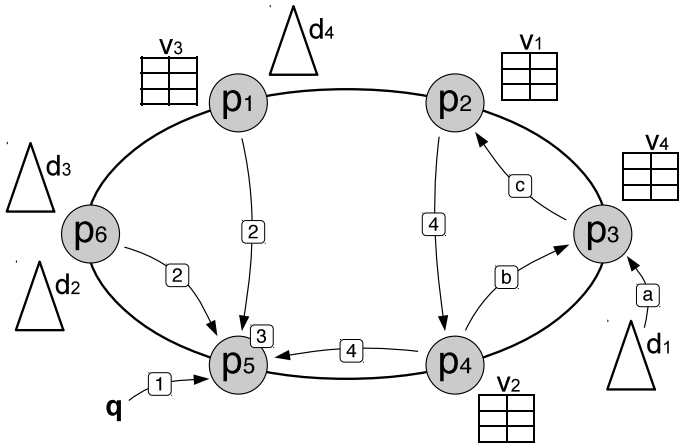


Figure 1: System overview.

3 ViP2P platform overview

XML data flows in ViP2P can be summarized as follows. XML documents are published independently and autonomously by any peer. Peers can also formulate subscriptions, or long-running queries, potentially matching documents published before, or after the subscriptions. The results of each subscription query are stored at the respective peer, and the definition of the query is indexed in the peer network. Finally, peers can ask ad-hoc queries, which are answered in a snapshot fashion (based on the data available in the network so far) by exploiting the existing subscriptions, which can be seen as materialized views. We detail the overall process via an example in Section 3.1. We then proceed to describe the ViP2P modules implementing it in Section 3.2.

3.1 ViP2P by example

A sample ViP2P instance over six peers is depicted in Figure 1 and we use it to base our presentation of the operations which can be carried in each peer. In the Figure, XML documents are denoted by triangles, whereas views are denoted by tables, hinting to the fact that they contain sets of tuples. More details on views and view semantics are provided in Section 5, but they are not required to follow the discussion here. For ease of explanation, we make the following naming conventions for the remainder of this paper:

- **Publisher** is a peer which publishes an XML document
- **Consumer** is a peer which defines a subscription and stores its results (or, equivalently, the respective materialized view)
- **Query peer** is a peer which poses an *ad-hoc* query (to be evaluated over the complete ViP2P network).

Clearly, a peer can play any subset of these roles simultaneously or successively.

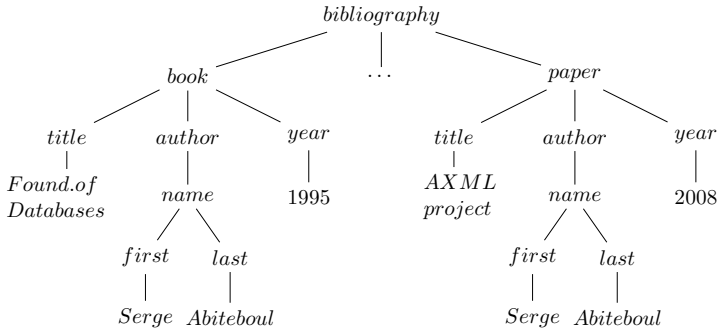


Figure 2: Sample XML document d_1.

3.1.1 View publication

A ViP2P view is a *long-running subscription query* that any peer can freely define. The *definition* (i.e., the actual query) of each newly created view is indexed in the DHT network. For instance, assume peer p_2 in Figure 1 publishes the view v_1, defined by the XPath query $//bibliography//book[contains(.,'Databases')]$. The view requires all the books items from a bibliography containing the word 'Databases'. ViP2P indexes v_1 by inserting in the DHT the following three (key, value) pairs: $(bibliography, v_1@p_2)$, $(book, v_1@p_2)$ and $('Databases', v_1@p_2)$. Here, $v_1@p_2$ encapsulates the structured query defining v_1, and a pointer to the concrete database at peer p_2 where v_1 data is stored. As will be shown below, all existing and future documents that can affect v_1, *push* the corresponding data to its database.

Peers look up views in the DHT in two situations: when publishing documents, and when issuing ad-hoc queries. We detail this below.

3.1.2 Document publication

When publishing a document, each peer is in charge of identifying the views within the whole network to which its document may contribute. For instance, in Figure 1 (step a), peer p_3 publishes the document d_1 (depicted in Figure 2). Document d_1 contains data matching the view v_1 as it contains the element names *bibliography* and *book*, as well as the word $'Databases'$. Peer p_3 extracts from d_1 all distinct element names and all keywords. For each such element name or keyword k, p_3 looks up in the DHT for view definitions associated to k, and, thus, learns about v_1 (step b). In the publication example above, p_3 extracts from d_1 the results matching v_1; from now on, we will use the notation $v_1(d_1)$ to designate such results. Peer p_3 sends $v_1(d_1)$ to p_2 (step c), which adds them to the database storing v_1 data.

A separate mechanism is needed for a view, say v_x, published after d_1 but having results in d_1. One possibility would be for the peer publishing v_x to look up, among all the network documents, for those that could contain terms from v_x and require them to contribute v_x results. The drawback is that this requires indexing all documents on all terms, which may be wasteful since a large part of published content may not be looked up frequently, or not at all.

Instead, ViP2P associates to each view an *interval timestamp*, corresponding to a time interval during which the view was published. Each peer having published a document d must check the DHT for views published after d. To

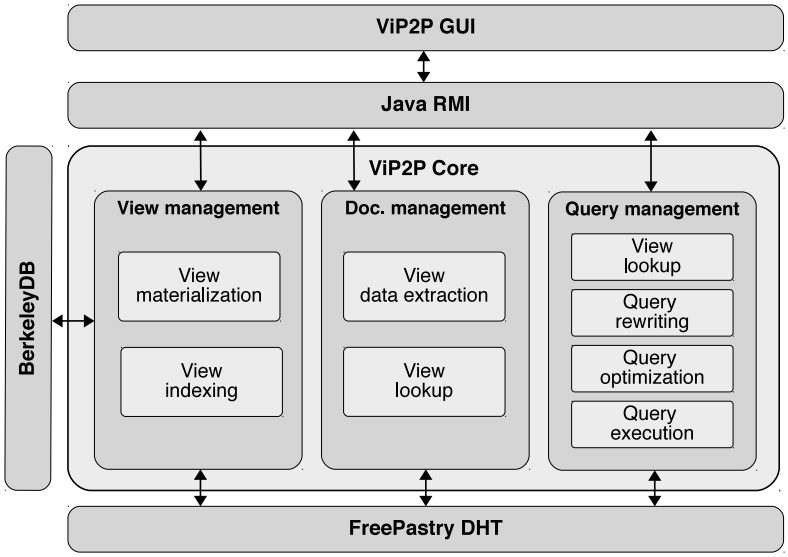


Figure 3: Basic architecture of a ViP2P peer.

that effect, each peer performs regular lookups using as key, the time interval which has just passed. Thus, it retrieves the definitions of all the views published during that interval and contributes its data if it hasn't done it already.

3.1.3 Ad-hoc query answering

ViP2P peers may pose *ad-hoc queries*, which must be evaluated immediately (from the previously published data). To evaluate such queries, a ViP2P peer looks up in the network for views which may be used to answer it. For instance, assume the query $q = //bibliography//book[contains(.,'Databases')]//author$ is issued at peer p_5 (step 1, in Figure 1). To process q, p_5 looks up (step 2) the keys *bibliography*, *book*, $'Databases'$ and *author* in the DHT, and retrieves a set of view definitions (step 3), including that of v_1. Observe that q can be rewritten as $v_1//author$; therefore, p_5 can answer q just by retrieving and extracting q's results out of v_1. A distinguishing feature of ViP2P is its ability to combine *several* materialized views in order to rewrite a query (as we describe in Section 4). A query rewriting (a logical plan based on some views) is translated by the ViP2P query optimizer into a distributed physical plan, specifying which operators will be used and in which peers they will be executed.The ViP2P optimizer is responsible of selecting the most efficient physical plan, as this choice has a significant impact in the query execution time, especially in a distributed setting such as ours where network communication plays an important role. The execution of the physical plan may require the cooperation of various peers, and leads to results being sent at the query peer (step 4).

3.2 ViP2P peer architecture

We now present the main modules of ViP2P peers as well as their functionalities and interaction, outlined in Figure 3. The *ViP2P Core* box includes the main modules, whereas boxes located outside *ViP2P Core* are independent external

subsystems that interact with ViP2P.

3.2.1 External Subsystems

FreePastry DHT [17] provides the underlying DHT layer on which ViP2P is built. FreePastry is an open-source implementation of Pastry [34], an efficient, self-organizing and fault-tolerant overlay network. Pastry provides efficient request routing, deterministic object location, and load balancing. ViP2P nodes index and lookup view definitions on FreePastry's DHT during the view materialization and query processing.

Java RMI is used for all large data transfers. Previous work [3] has shown that the DHT communication primitives were not suitable for such transfers, since (*i*) the DHT *get* and *put* operations are blocking, that is, data sent via the DHT becomes available at the receiver only when it has been completely received and (*ii*) message queues in the DHTs overflow easily even after tuning, in which case the DHT peers re-send them, which further clogs the DHT communication pipes. Beyond the degradation of performance, such message overflows are annoying because a peer that is too busy trying to re-send data, may skip sending the regular "ping" to his neighbors to signal that it is still alive. Then, the neighbors suspect the peer is down, this triggers further loss of messages etc.

For all these reasons, we have decided to split inter-peer communication in two categories. The DHT is used to efficiently send small messages, typically to index and look up view definitions. We use RMI (which we were able to fine-tune by writing efficient serialization/de-serialization methods, properly controlling concurrency at the send and receiver side etc.) to send larger messages containing view tuples, when views are materialized and queried. We also applied specific techniques to reduce the space occupancy of transmitting tuples. Thus, a document ID (or URI) often appears many times in a view, as many times as there are view tuples obtained from that document. Since the URIs are quite large, they make up an important part of the document data. We use dictionary-based encoding of the document URIs, i.e., the tuple sender dynamically builds a dictionary of all document URIs and sends partial dictionaries with each tuple packet, to enable decoding on the receiver side. One could perhaps improve performance even further by coding data-intensive communications at a lower level (e.g. using plain sockets), but the improvements attained by our way of utilizing RMI are already very significant.

BerkeleyDB Within each peer, view tuples are efficiently stored into a native store that we built using the Berkeley DB [8] library. It provides the routines to store, retrieve and sort entries, while guaranteeing ACID transactions when view data are written and read concurrently.

The GUI facilitates the control and inspection of each peer, enabling users to publish views and/or pose queries. Screenshots of the ViP2P GUI, along with other information, can be found on the ViP2P website[1].

We now move to describing the core modules.

3.2.2 Document management module

This module is responsible for looking up for views to which the peer's documents may contribute, extracting the data from the documents and sending it

[1] http://vip2p.saclay.inria.fr/

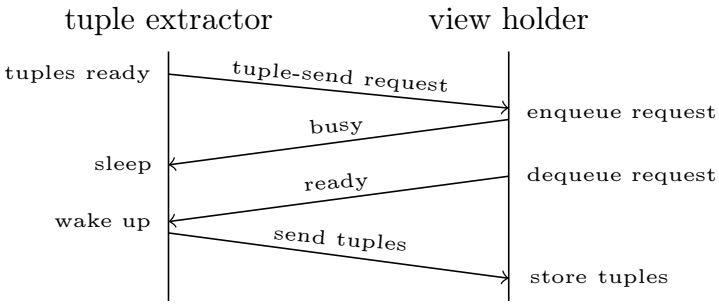


Figure 4: Tuple-send/receive protocol use case between a tuple–sender and a view holder.

to the respective consumers.

View definition lookup When a new document is published by a peer, the *view lookup* module at this peer first, looks up in the DHT the definitions of the views to which the document may contribute data, and then passes these views definitions to the *view data extraction module*.

View data extraction Given a list of view definitions, the *view data extraction module* at a publisher peer extracts from the document the tuples matching each view, and ships them, in a parallel fashion, to the different consumers. The view data extractor is capable of simultaneously matching several views on a given document. Thus, the corresponding tuples are extracted during a single traversal of the document. The extractor maintains a thread pool for setting up RMI communications for shipping tuples to the consumers. As our experiments show in Section 6.3, this parallel tuple sending significantly reduces the time needed to materialize the views.

3.2.3 View management module

This module handles view indexing and materialization.

View indexing This module makes visible to all network peers the definitions of all the views declared in the ViP2P network (of course without broadcasting them, since most peers are typically not interested in all views). When a new view is defined, the indexer inserts in the DHT (key,value) pairs used to describe it, based on one of the *indexing strategies* that we will describe in Section 5.1.

View materialization The *view materialization* module receives tuples from remote publishers and stores them in the respective BerkeleyDB database. In a large scale, real-world scenario, thousands of documents might be contributing data to a single view. To avoid overload on its incoming data transfers, this module implements a back-pressure *tuple-send/receive protocol* which informs the publisher when the incoming tuple buffer is full at the consumer side. Thus, a publisher may have to wait until the consumer is ready to accept the tuples. This makes the most out of the available publisher-to-consumer bandwidth, all the while avoiding costly re-transmissions due to messages lost from overflowing queues.

Figure 4 traces the tuple-send/receive protocol between a tuple extractor and a view holder. First the tuple extractor extracts the tuples and keeps them

in memory being ready to ship them to the view holder. After that, it sends a tuple-send request to the view holder. In this example, the view holder is busy storing tuples (possibly sent by other tuple extractors), thus it enqueues the request and responds to the tuple extractor with a "busy" response. When the view holder is ready to accept the new set of tuples, it dequeues the request and informs the tuple extractor (via a "ready" message). Then, the tuple extractor ships the new tuples to the view holder, who finally stores them in the Berkeley database of the respective view. The view holder can serve multiple tuple-send requests concurrently. Our experiments (Section 6.3) show how the concurrency can affect the time needed for a set of views to be materialized.

3.2.4 Query management module

A sequence of steps are required to evaluate queries, each performed by a dedicated module, as follows.

View lookup This module, given a query, performs a lookup in the DHT network retrieving the view definitions that can be used to rewrite the query.

Query rewriting This module takes a given ad-hoc query and a set of available view definitions and produces a logical rewriting plan which, evaluated on some views, produces exactly the results required by the query (algorithm detailed in [27] and illustrated in Section 4).

Query optimization This module receives as input a logical rewriting plan which is output by the query rewriting module and translates it to an optimized physical plan. The optimization takes place both at the logical (join reordering, push selections and projections etc.) and physical (dictating the exact flow of data during query execution) level.

Query execution This module provides a set of physical operators which can be deployed at any ViP2P peer, implementing the standard iterator-based execution model [20]. In particular, the *send*/*receive* operators are implemented using asynchronous communication buffers, data is transferred via RMI, and document URIs (present in each view tuple, to identify the document the tuple was extracted from) are compressed using a dictionary by the *send* (and decompressed by the *receive*) in order to reduce the transferred data volumes. Other notable physical operators are binary structural joins [5], holistic twig joins [11], hash- and merge-joins etc.

4 Views, queries and rewritings

Once an ad-hoc query is issued by a peer, as described in Section 3, a DHT lookup retrieves the definitions of the existing ViP2P network views which could be used to answer the query. Then, the query peer runs its own algorithm for rewriting the query using the respective materialized views. The algorithm used in ViP2P is presented in [27] and its details are out of the scope of this paper, where we are mainly concerned with the platform and its scalability. However, to make this paper self-contained, we present the XQuery dialect we consider (Section 4.1), we present a joined tree pattern formalism that conveniently represents queries (Section 4.2) and describe our algebraic rewritings based on views (Section 4.3).

4.1 XQuery dialect

Let $\mathcal{L}$ be a set of XML node names, and $\mathcal{XP}$ be the XPath$^{\{/,//,[]\}}$ language [31]. We consider views and queries expressed in the XQuery dialect described in Figure 5. In the for clause, $absVar$ corresponds to an absolute variable declaration, which binds a variable named x_i to a path expression $p \in \mathcal{XP}$ to be evaluated starting from the root of some document available at the URI uri. The non-terminal $relVar$ allows binding a variable named x_i to a path expression $p \in \mathcal{XP}$ to be evaluated starting from the bindings of a previously-introduced variable x_j. The optional where clause is a conjunction over a number of predicates, each of which compares the string value of a variable x_i, either with the string value of another variable x_j, or with a constant c.

The return clause builds, for each tuple of bindings of the for variables, a new element labeled l, having some children labeled l_i ($l, l_i \in \mathcal{L}$). Within each such child, we allow one out of three possible information items related to the current binding of a variable x_k, declared in the for clause: (1) x_k denotes the full subtree rooted at the binding of x_k; (2) string(x_k) is the string value of the binding; (3) id(x_k) denotes the ID of the node to which x_k is bound.

There are important differences between the *subtree* rooted at an element (or, equivalently, its *content*), its *string value* and its *ID*. The content of x_i includes all (element, attribute, or text) descendants of x_i, whereas the string value is only a concatenation of n's text descendants [39]. Therefore, string(x_i) is very likely smaller than x_i's content, but it holds less information. Second, an XML ID does not encapsulate the content of the corresponding node. However, XML IDs enable joins which may stitch together tree patterns into larger ones. *We assume structural IDs*, i.e., comparing the IDs $id(n_1)$ and $id(n_2)$ allows determining if n_1 is a parent (or ancestor) of n_2. Our XQuery dialect distinguishes structural IDs, value and contents, and allows any subset of the three to be

1	$q :=$	for $absVar$ (, $(absVar\|relVar)$)* (where $pred$ (and $pred$)*)? return ret
2	$absVar :=$	x_i in doc(uri) p
3	$relVar :=$	x_i in x_j p // x_j *introduced before* x_i
4	$pred :=$	string(x_i) = (string(x_j) | c)
5	$ret :=$	$\langle l \rangle$ $elem$* $\langle /l \rangle$
6	$elem :=$	$\langle l_i \rangle${ (x_k | id(x_k) | string(x_k)) }$\langle /l_i \rangle$

Figure 5: Grammar for views and queries.

q	for $p in doc("confs")//confs//SIGMOD/paper, $y1 in $p/year,
	$a in $p//author[email], $c1 in $a/affiliation//country,
	$b in doc("books")//book, $y2 in $b/year, $e in $b/editor,
	$t in $b//title, $c2 in $b//country
	where $e='ACM' and $y1=$y2 and $c1=$c2
	return ⟨res⟩ ⟨tval⟩{string($t)}⟨/tval⟩ ⟨/res⟩

v_1	for $p in doc("confs")//confs//paper, $a in $p/affiliation return ⟨v1⟩ ⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩ ⟨acont⟩{$a}⟨/acont⟩ ⟨/v1⟩

v_2	for $b in doc("books")//book, $c in $b//country, $e in $b/editor,
	$t in $b/title, $y1 in $b/year, $p in doc("confs")//SIGMOD/paper,
	$y2 in $p/year, $a in $p//author[email]
	where $e='ACM' and $y1=$y2
	return ⟨v2⟩ ⟨cval⟩{string($c)}⟨/cval⟩ ⟨tval⟩{string($t)}⟨/tval⟩
	⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩ ⟨/v2⟩

r	for $v1 in doc("v1.xml")//v1, $p1 in $v1/pid, $af1 in $v1/aid,
	$c1 in $v1//acont//country, $v2 in doc("v2.xml")//v2,
	$c2 in $v2/cval, $t2 in $v2/tval, $p2 in $v2/pid, $a2 in $v2/aid
	where $p1=$p2 and parent($a2,$af1) and $c1=$c2
	return ⟨res⟩ ⟨tval⟩{$v2}⟨/tval⟩ ⟨/res⟩

Figure 6: XQuery query, views, and rewriting.

returned for any of the variables, resulting in significant flexibility.

For illustration, Figure 6 shows a query q in our XQuery dialect, as well as two views v_1 and v_2. The parent custom function returns true if and only if its inputs are node IDs, such that the first identifies the parent of the second. Moreover, as usual in XQuery, the variable bindings that appear in the where clauses imply the string values of these bindings (e.g. $e='ACM' is implicitly converted to string($e)='ACM').

4.2 Joined tree patterns

We use a dialect of joined tree patterns to represent views and queries. Formally, a tree pattern is a tree whose nodes carry labels from $\mathcal{L}$ and may be annotated with zero or more among: ID, *val* and *cont*. A pattern node may also be annotated with a value equality predicate of the form $[=c]$ where c is some constant. The pattern edges are either simple for parent-child or double for ancestor-descendant relationships. A joined tree pattern is a set of tree patterns, connected through value joins, which are denoted by dashed edges. For illustration, Figure 7 depicts the (joined) tree pattern representations of the query and views shown in XQuery syntax in Figure 6. In short, the semantics of an annotated tree pattern against a database is a list of tuples storing the ID, *val* and *cont* from the tuples of database nodes in which the tree pattern embeds. The tuple order follows the order of the embedding target nodes in the database. The detailed semantics feature some duplicate elimination and projection operators (from the algebra we will detail next), in order to be as close to the W3C's XPath 2.0 semantics as possible. The only remaining difference is that tree patterns return tuples, whereas standard XPath/XQuery semantics uses node lists. Algebraic operators for translating between the two are by now well understood [28]. The semantics of a joined tree pattern is the join of the semantics of its component tree patterns.

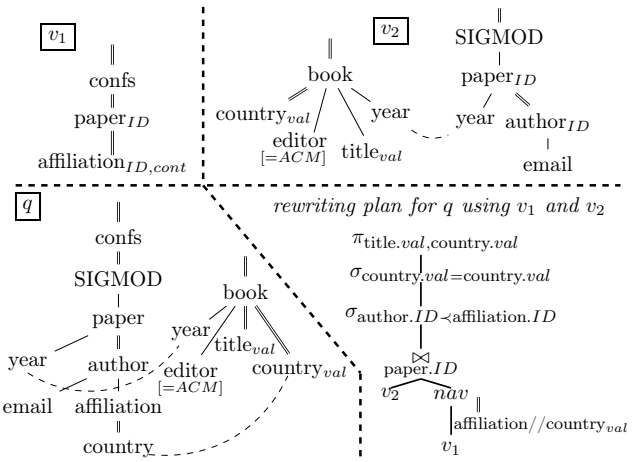


Figure 7: Pattern query and views, and algebraic rewriting.

Translating from our XQuery dialect to the joined tree patterns is quite straightforward. The only part of the XQuery syntax *not* reflected in the joined tree patterns is the names of the elements created by the return clause. These names are not needed when rewriting queries based on views. Once a rewriting has been found, the query execution engine creates new elements out of the returned tuples of XML elements, values and/or identifiers, using the names specified by the original query, as explained in [36]; we will not discuss this further. From now on, for readability, we will only use the tree pattern query representations of views and queries.

4.3 Rewritings & algebra

A rewriting is an XQuery query expressed in the same dialect as our views and queries, but formulated against XML documents corresponding to materialized views. For instance, the rewriting XQuery expression r in Figure 6 is an equivalent rewriting of the query q using the views v_1 and v_2 in the same Figure.

An alternative, more convenient, way to view rewritings is under the form of *logical algebraic plans*. Before presenting plans, we introduce some useful logical operators. We denote by $\prec$ the *parent comparison operator*, which returns true if its left-hand argument is the ID of the parent of the node whose ID is the right-hand argument. Similarly, $\prec\!\!\prec$ is the *ancestor comparison operator*. Observe that $\prec$ and $\prec\!\!\prec$ are only abstract operators here (we do not make any assumption on how they are evaluated).

We consider an algebra on tuple collections (such as described in the previous Section) whose main operators are: (1) scan of all tuples from a view v, denoted $scan(v)$ (or simply v for brevity, whenever possible), (2) cartesian product, denoted $\times$; (3) selection, denoted σ_{pred}, where $pred$ is a conjunction of predicates of the form $a \odot \underline{c}$ or $a \odot b$, a and b are tuple attributes, $\underline{c}$ is some constant, and $\odot$ is a binary operator among $\{=, \prec, \prec\!\!\prec\}$; (4) projection, denoted π_{cols}, where $cols$ is the attributes list that will be projected; (5) navigation, denoted $nav_{a,np}$. nav is a unary algebraic operator, parameterized by one of its input columns' name a, and a tree pattern np. The name a must correspond to a *cont* attribute

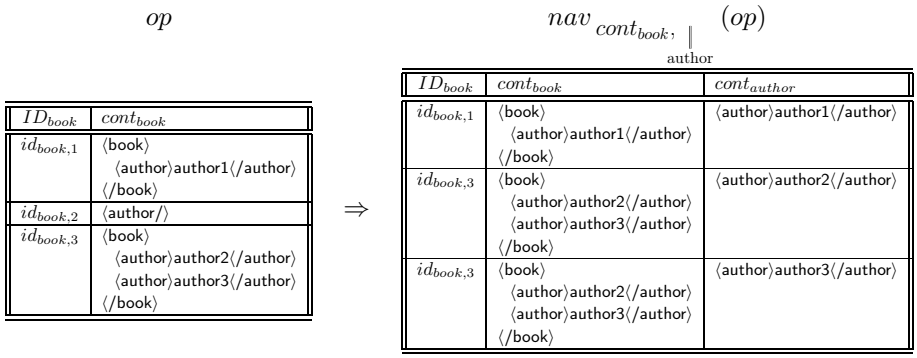


Figure 8: Sample input and output to a logical *nav* operator.

in the input of *nav*. Let t be a tuple in the input of *nav*, and $np(t.a)$ be the result of evaluating the pattern np on the XML fragment stored in $t.a$. Then, $nav_{a,np}$ outputs the tuples $\{t \underset{a}{\bowtie} np(t.a)\}$.

Figure 8 illustrates the functioning of *nav* on a sample input operator *op*. The parameters to this *nav* are $cont_{book}$, the name of the column containing ⟨book⟩ elements, and the tree pattern $//author$. The first tuple output by *nav* is obtained by augmenting the corresponding input tuple with a $cont_{author}$ attribute containing the single author-labeled child of the element found in its $cont_{book}$ attribute. The second and third *nav* output tuples are similarly obtained from the last tuple produced by *op*. Observe that the second tuple in *op*'s output has been eliminated by the *nav* since it had no ⟨author⟩ element in its $cont_{book}$ attribute.

The algebra also includes the join operator, defined as usual, sort and duplicate elimination. For illustration, in the bottom of Figure 7, we depict the algebraic representation of the rewriting r shown in XQuery syntax at the bottom of Figure 6.

4.4 Minimal rewritings

Each equivalent rewriting r of a query q produced by our rewriting algorithm [27] is *minimal*, that is: one cannot obtain an equivalent rewriting of q using only *a strict subset of the view occurrences appearing in* r. For instance, the rewriting r in Figure 6, of the form $\sigma(v_1 \bowtie_{paper.ID} v_2)$ is minimal. In contrast, a rewriting r' of the form $\pi(\sigma(v_1 \bowtie_{paper.ID} v_1 \bowtie_{paper.ID} v_2))$, using the v_1 view twice, in a self-join on paper.ID, is not minimal. Considering only minimal rewriting allows keeping view storage space *and* query evaluation costs low.

5 ViP2P view management

Materialized views stand at the heart of data sharing in ViP2P. Sections 5.1 and 5.2 show how view definitions are indexed and looked up in the DHT in order to be retrieved for view materialization and query rewriting, respectively.

5.1 View definition indexing & lookup for view materialization

This Section describes how published data and views "meet", i.e., how ViP2P ensures that for each view v, the data obtained by evaluating v over d, denoted $v(d)$, is *eventually* computed and stored at the peer having defined v. Two cases arise, depending on the publication order among v and d.

View published before the document In this case, the view definitions are indexed using as keys all the labels (node names and words) of the view. Figure 10 shows eight views. One of them, v_8 is a value join (depicted as a dashed line) between two tree patterns. To index v_1, ViP2P issues the calls *put(book,* v_1*)* and *put(title,* v_1*)* to the DHT. These calls index the syntactic definition of v_1 (not its data) on the keys *book* and *title*. Similarly, v_2 is indexed on the keys *book*, *author* and *last*, v_3 using the keys *paper*, *author* and *last* etc. When the document in Figure 2 is published, *get* calls are issued with the keys *bibliography*, *book*, *paper*, *title*, *author*, *year*, *Found. of Databases* and all the other labels and keywords of the document. The result is a superset of view definitions of the views that the document might affect. In this case the views v_1 to v_8 are retrieved.

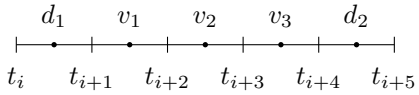


Figure 9: Sample timeline of view and document publication.

View published after the document ViP2P ensures that views are up to date (providing for some time to circulate data across the network). Thus, when a view is published, it should be filled in with data from all the previously published documents, that match the view. To achieve this, ViP2P associates to each view an *interval timestamp*, corresponding to a time interval during which the view was published. This is illustrated in Figure 9. Here, v_1 was published in (belongs to) the interval $(t_{i+1}, t_{i+2}]$, v_2 belongs to the interval $(t_{i+2}, t_{i+3}]$ and v_3 belongs to the interval $(t_{i+3}, t_{i+4}]$.

Each peer having published a document d must check the DHT for views that may have appeared after d. To that effect, each peer performs regular lookups using as key, the time interval which has just finished. This retrieves the definitions of all the views published during that interval. The peer then checks, for each of its documents, if the document has already contributed to that view (this information is stored locally at the peer). If this is not the case, the peer checks whether the document holds any data for these views and if yes, extracts and sends the corresponding data etc. In Figure 9, we see that document d_1 arrives during the $(t_i, t_{i+1}]$ time interval. With the help of the timestamped view index, we discover the views v_1, v_2 and v_3 which arrived

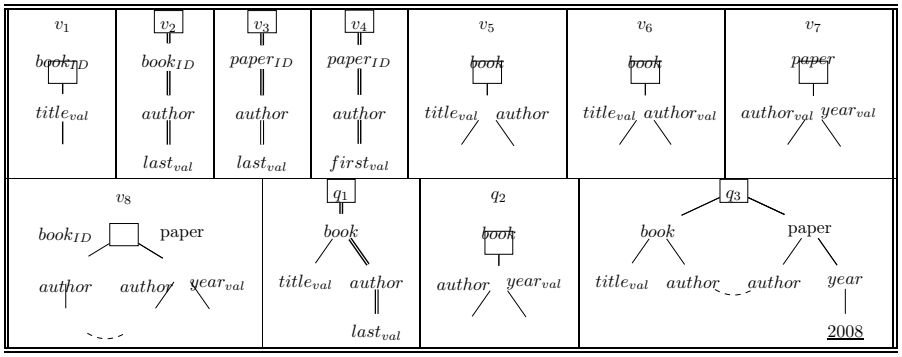


Figure 10: Sample views and queries.

later. (Document d_2 is published after the views and thus is treated according to the first case above.)

5.2 View definition indexing & lookup for query rewriting

View definitions are also indexed in order to find views that may be used to rewrite a given query. In this context, a given algorithm for extracting (key, value) pairs out of a view definition is termed a *view indexing strategy*. For each such strategy, a *view lookup* method is needed, in order to identify, given a query q, (a superset of) the views which could be used to rewrite q. Many strategies can be devised. We present four that we have implemented, together with the space complexity of the view indexing strategy, and the number of lookups required by the view lookup method. We also show that these strategies are *complete*, i.e. they retrieve at least all the views that could be embedded in q and, thus, lead to q rewritings.

5.2.1 Label indexing (LI)

In this strategy we index v by each v node label (either some element or attribute name, or word). The number of (key, value) pairs thus obtained is in $O(|v|)$.

View lookup for LI The lookup is performed by all node labels of q. The number of lookups is $\Theta(|q|)$, where $|q|$ is the number of nodes in the query. Figure 10 depicts some sample queries. The LI lookup keys for q_1 are *book*, *title*, *author* and *last*, retrieving all the views of Figure 10. Note that some of these can not be used to equivalently rewrite q_1. For instance, v_1 has data about papers, while q_1 is interested in books etc. Similarly, LI indexing and lookup for q_2 and q_3 leads to retrieving all the views. This shows that LI has many false positives.

LI completeness If LI is not complete, then there exists a view v that can be used to rewrite a query q, and v is not retrieved when searching by all q labels. It has been shown [38] that in order for a view to appear in an equivalent rewriting of a query, there must exist an embedding (or homomorphism) from the view into the query, which entails that some node labels must appear in both. If in our case v and q have no common node label, this contradicts the hypothesis that v was useful to rewrite q.

The LI strategy coincides with the view definition indexing for document-driven lookup (described previously). An interesting variant can furthermore

be elaborated.

5.2.2 Return label indexing (RLI)

Here, we index v by the labels of all v nodes which project some attributes (at most $|v|$). For instance, in Figure 10, the index keys for v_1 are *book* and *title*, for v_2 they are *book* and *last*, for v_3 *paper* and *last* etc. up to v_7 which is indexed by RLI on the keys *book* and *year*.

View lookup for RLI The view definition lookup is the same as for LI (look up on all query node labels). In Figure 10, the definitions of $v_1 - v_3$, and $v_5 - v_8$ will be retrieved for q_1. For q_2, the definitions of v_1, v_2, v_6, v_7 and v_8 will be retrieved. A RLI lookup for q_3 will retrieve $v_1 - v_8$. Observe that RLI lead to less view definitions retrieved than LI.

RLI completeness Suppose that there is a view v which can be used to rewrite a query q, yet the definition of v is not retrieved by RLI lookup. This means that either (i) v does not store any attributes or (ii) the labels of v nodes that project an attribute do not appear in q. (i) is not possible because a view that participates to a rewriting should store at least an attribute and (ii) is also not possible since it contradicts the existence of an embedding from v to q, required for v to be useful in rewriting q.

5.2.3 Leaf path indexing (LPI)

Let $LP(v)$ be the set of all the distinct root-to-leaf label paths of v. Here, a path is just the sequence of labels encountered as one goes down from the root to the node, and does not reflect the type of the edges. We index v using each element of $LP(v)$ as key. The number of (key, value) pairs thus obtained is in $\Theta(|LP(v)|)$. Going back to Figure 10, v_1 is indexed on the key *book.title*, v_2 with the key *book.author.last* etc. The view v_8, composed of two tree patterns, is indexed using the keys *book.author*, *paper.author* and *paper.year*.

View lookup for LPI Let $LP(q)$ be the set of all the distinct root-to-leaf label paths of q. Let $SP(q)$ be the set of all non-empty sub-paths of some path from $LP(q)$, i.e., each path from $SP(q)$ is obtained by erasing some labels from a path in $LP(q)$. Use each element in $SP(q)$ as lookup key. For example, q_1 of Figure 10 LPI lookup uses the keys *book.title*, *book*, *title*, *book.author.last*, *book.author*, *author.last*, *book.last*, *book*, *author* and *last* etc. Note that LPI lookup for q_1 does not retrieve the definitions of the views v_3, v_4, and v_7, which previous strategies retrieved, although they are not useful to rewrite q_1. LPI can still have some false positives though: a lookup for q_2 retrieves v_5, v_6 and v_8, none of which can be used to rewrite q_2 (in this example, q_2 simply has no rewriting). The lookup for q_3 retrieved the views v_1, v_5, v_6, v_7 and v_8. The filtering is very good in this case because among these only v_5 can not be used to rewrite q_3.

Let $h(q)$ be the height of q and $l(q)$ be the number of leaves in q. The number of LPI lookups is bound by $\Sigma_{p \in LP(q)} 2^{|p|} \leq l(q) \times 2^{h(q)}$. If the query q is a join of tree patterns (tpqs) then the bound becomes $\Sigma_{tpq \in q}(\Sigma_{p \in LP(tpq)} 2^{|p|})$.

LPI completeness is guaranteed by the fact that if a view v can be embedded in the query q, then $LP(v) \subseteq SP(q)$.

5.2.4 Return path indexing (RPI)

RPI is the last strategy that we consider. Let $RP(v)$ be the set of all rooted paths in v which end in a node that returns some attribute. Index v using each element of $LP(v)$ as key. The number of (key,value) pairs is also in $\Theta(|RP(v)|)$. The indexing keys for v_1 are *book* and *book.title*, for v_2 are *book* and *book.author.last* etc.

View lookup for RPI coincides exactly with the lookup for LPI. The lookup of q_1 retrieves the definitions of the views v_1, v_2, v_5, v_6 and v_8, the same as LPI. For q_2, RPI lookup retrieves the definitions of v_1, v_2, v_6 and v_8. Observe that unlike LPI, RPI in this situation does not return v_5, which indeed is not useful! We end by noting that this increase of precision of RPI over LPI is not guaranteed. For example, an RPI lookup for q_3 retrieves the definitions of all views in Figure 10, which is much less precise than LPI.

RPI completeness is established in a similar fashion to the LPI case.

6 Experimental results

In this Section we present a set of experiments studying ViP2P performance. Section 6.1 outlines the experimental setup. ViP2P attempts to speed up query processing by exploiting pre-computed materialized views. This shifts the complexity of extracting and sending interesting data across the network, from query processing to view materialization, to which we devote the most attention in our experiments. Several parameters determine view materialization performance: the distribution of the documents and views in the network, the documents which contribute to each view, the documents and views size etc. Section 6.2 starts by studying view materialization in the context of a single peer. Then, Section 6.3 examines view materialization in the large, in widely different network configurations, varying the number and the distribution of publisher and consumer peers. Section 6.4 presents an evaluation of the indexing strategies presented in Section 5.2. Finally, Section 6.5 presents experiments that evaluate the performance of the query execution engine.

6.1 Experimentation settings

Infrastructure setup We have carried our experiments on the Grid5000 infrastructure (https://www.grid5000.fr), providing computational resources distributed over nine major cities across France. Sites are interconnected with a 10Gbps network. Within each site, nodes are are interconnected with (at least) 1Gbps Ethernet network. The hardware of Grid5000 machines varies from dual-core machines (of at least 1.6 GHz clock speed) with 2GBs of RAM to 16-core machines with 32GBs of RAM. We settled for a random and heterogeneous distribution of hardware, in order to be close to real P2P deployment scenarios.

Data generation To have fine control over all the parameters impacting our experiments, we have used synthetic data, produced by two existing XML data generators: ToXGene [7] and MemBeR [4].

Experimentation parameters We summarize the main parameters characterizing our experiments in Table 1. For each set S, we use $|S|$ to denote the size of the set. Thus, $|P|$ is the number of peers in the network etc. Finally, for a document d, we use $|d|$ to denote the size of d, measured in Megabytes (MBs).

Evaluation metrics In our measurements, we use the following metrics to characterize the system performance:

- **Materialization time** is the time needed for the network to materialize a set of views populating them with the data extracted by all the documents published in the network. The materialization time starts at the time instant that a peer initiates the first extraction of data and ends at the time that all peers have extracted and shipped the tuples to the appropriate view holders.

- **Tuple extraction time** for a view v and a document d is the time needed for the publisher of d to extract from d the tuples which make up $v(d)$.

- **Storage time** for a document d and a view v is the time taken by the consumer holding v, to add to the corresponding BerkeleyDB database the set of tuples corresponding to $v(d)$.

Symbol	Description
P	The set of peers in the network
P_D	The set of peers holding at least one document
V	The set of views in the network
P_V	The set of peers holding at least one view
D	The set of all published documents
D_V	The set of documents matching at least one view

Table 1: Parameters characterizing the experiments.

- **Data exchange time** for a document d and view v is the time needed for the tuples $v(d)$ to be transferred across the network from the publisher of d to the consumer holding v.
- **Lookup time** for a query q is the time needed for the peer asking q to lookup in the DHT the views that may be useful to rewrite q.
- **Embedding time** for a query q and a set of views V is the time needed by the query peer to verify which of the views may actually be used to rewrite q. Recall from Section 5.2 that this is established by checking for the presence of embeddings between each view $v \in V$ and the query q [38].
- **Query response time** for a query q is the time elapsed between the moment when the query has been posed, and the moment when its execution has finished (as observed at the query peer).
- **Time to first result** for a query q is the time between the moment when the query has been posed, and the moment when its first result tuple has been received at the query peer.

Whenever the query, view, or document are not specified for a given metric, *the metric value is understood to be the sum, over all the documents, views, and queries used in the respective experiment, of the respective metric, with the exception of the materialization time*. By nature, this metric accounts for many materialization processes running *in parallel*, and therefore is not the sum of individual materialization times. For instance, assume publisher p_1 publishes a document which contributes data to a view at p_2, while publisher p'_1 similarly contributes to a view at p'_2. The peers p_1 and p'_1 will start at about the same time the materialization process by looking up views to which they could contribute etc. One of them will be the last to report that all its tuples have been stored and acknowledged by the respective consumer peer. The materialization time of this experiment spans between the first materialization start event, and the last materialization end event, while the two processes run in parallel.

6.2 View materialization in the small

We start by studying the performance of extracting from a document d, the tuples corresponding to a view v, and sending these $v(d)$ tuples from the peer holding d to the one storing v. To focus exactly on the process of extraction, we use very simplistic network settings. View materialization in more complex settings and larger scale will be studied next.

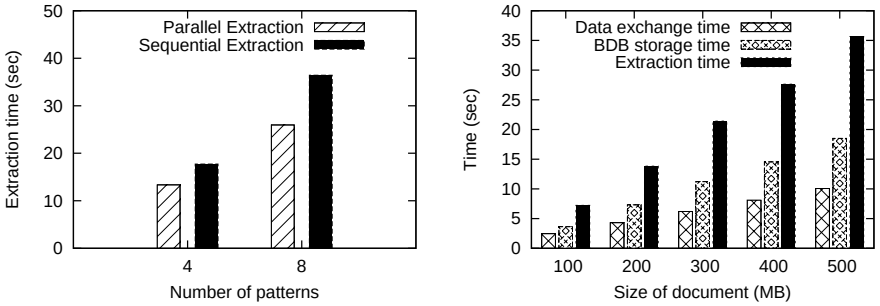


Figure 11: Experiment 1: parallel vs. sequential extraction time (left); experiment 2: view materialization over different-size documents (right).

Experiment 1: sequential vs. parallel extraction of views As described in Section 3.2, a ViP2P peer p is capable of simultaneously matching several views $v_1, v_2, \ldots, v_k$ on a given document d residing at p. The corresponding tuples $v_1(d)$, $v_2(d)$, ..., $v_k(d)$ are extracted during a single traversal of the document d, instead of k traversals, one for each document. This is important when publishing a document d in case the publisher finds out that many previously defined views could match d, and therefore it has to match all of them against d. While parallel extraction is faster, it may require more memory, since matches for the various views have to be constructed and kept in memory at the same time.

Our first experiment studies the effect of extracting data for several views in parallel. We use a document d and two distinct sets of views. First, we consider a four-view set of the form $\{//t_{i\,ID}\}$ for $i \in \{1, \ldots, 4\}$. Second, we consider a larger set including views of the form $\{//t_{i\,ID}\}$ for $i \in \{1, \ldots, 8\}$. The views and d are chosen so that d contributes 130.000 tuples to each published view v_i. The parameters characterizing the experiment are as follows:

$\lvert P\rvert$	$\lvert P_D\rvert$	$\lvert V\rvert$	$\lvert P_V\rvert$	$\lvert D\rvert$	$\lvert D_V\rvert$	$\lvert d\rvert$
2	1	{4, 8}	1	1	1	100MB

Figure 11 (left) depicts the extraction time when extracting data out of d for four and for eight views, in a parallel and sequential fashion. We observe that parallel extraction accelerates data extraction (in this case, up to 40%). Therefore, we will always use parallel extraction in the subsequent experiments.

Experiment 2: studying one data transfer pipe We now study the materialization of documents of various sizes, in order to identify the bottleneck of the materialization process. Possible bottlenecks are (*i*) data extraction at the document publisher; (*ii*) network bandwidth between a consumer and a publisher; (*iii*) view storage time at the consumer. For this experiment, the following parameters are used:

$\lvert P\rvert$	$\lvert P_D\rvert$	$\lvert V\rvert$	$\lvert P_V\rvert$	$\lvert D\rvert$	$\lvert D_V\rvert$	$\lvert d\rvert$
2	1	1	1	1	1	$\{100, \ldots, 500\}$MB

One peer plays the role of the publisher, while the other is the consumer. The peers are located at two opposite ends of France (Lille and Grenoble). The

document and the view are chosen so that the complete content of the document is extracted and sent to the consumer, thus, the materialized view size increases linearly to the size of the document.

Let us now detail the synchronization of the various processes involved when a publisher sends data to a consumer to be added in a view.

1. The publisher extracts data locally. *After* all the tuples from $v(d)$ have been computed, the publisher starts sending them to the consumer[2].
2. Packets of tuples are sent over the network to the consumer in an asynchronous way using buffers at the consumer side.
3. At the consumer, a thread picks packets of tuples from the buffer and stores them in the BerkeleyDB database.

The buffer at the consumer can be parameterized to control the data transfer speed: when the buffer is full because the storage thread is not sufficiently fast, data transfer stalls. For this experiment, the size of the data buffer was set to *unlimited* (making sure in advance that the memory of the consumer is enough to store all the produced tuples), so that the data exchange thread can use as much as possible of the available bandwidth between the two peers.

Figure 11 (right) depicts the time needed for the view tuples to be (*i*) extracted from the document, (*ii*) sent over the network and (*iii*) stored in BerkeleyDB at the consumer. We observe that the three times increase linearly in the size of the data. Data extraction is the slowest component, however, overall, times were comparable (also recall that the network connection is fast within the Grid, thus transfer times may be higher in other contexts).

From the above two experiments, we conclude that (*i*) parallelizing data extraction does speed up the time to compute view tuples; (*ii*) extraction time grows linearly to the size of the input document and (*iii*) data transfer and data storage time grow linearly with the size of the extracted tuples.

6.3 View materialization in larger networks

We now consider view materialization in larger and more complex environments, with many publishers and/or many consumers.

Documents For these experiments, we needed to tightly control which parts of the published data are relevant to which views on each peer. Therefore, unless stated otherwise, we rely on documents whose shape is outlined on the left of Figure 12. There are always 64 camera elements under one *catalog*, and each *camera* has 4 children. To obtain different document sizes, we insert text of varying length in the *description* of each *camera*.

Experiment 3: one publisher, fixed data, varying number of consumers In this experiment, we use a single publisher, a fixed data set (5 documents of 50 MBs each), and a varying number of consumers (from 1 to 64). Each consumer always holds exactly one view. All the published data is relevant for some view and moreover, the view contents do not overlap, i.e., the data is practically "partitioned" over the views. Thus, when there is a single consumer,

[2]This could be improved to parallelize extraction and sending in some cases, but there are fundamental limitations: for some of the views we support, one needs to wait for the full traversal of the document before producing an output tuple [19].

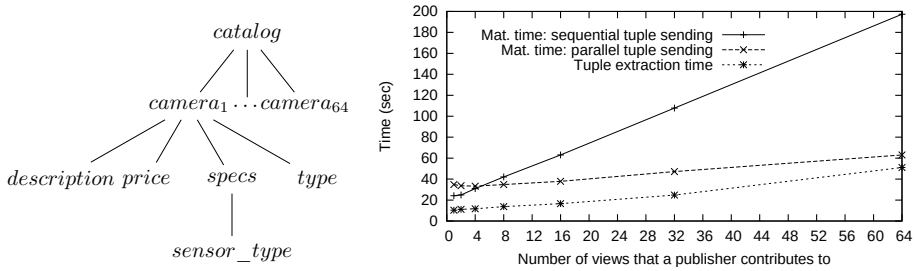


Figure 12: Outline of a controlled synthetic document for our experiments (left); experiment 3: view extraction and materialization time depending on the number of consumers (right).

its view stores the *cont* of all cameras from the catalog. When there are two consumers, the view of the first consumer stores the *cont* of the cameras from camera_1 to camera_{32}, while the other consumer's view stores the *cont* of the rest of the cameras (camera_{33} to camera_{64}) and so on. This way, the views absorb all the data published. The producer is located in Lille and the consumers in Sophia-Antipolis (two opposite ends of France). The parameters values for this experiment are given in the table below:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
65	1	{1,2,4,...,32,64}	$\{1, 2, 4, \ldots, 32, 64\}$	5	5	50MB

Once the tuples are extracted by a publisher, they can be shipped to the view holders sequentially (the publisher contacts the consumers one after the other) or in parallel (the publisher ships all the tuples to all consumers concurrently). At right in Figure 12, we show the time needed to extract the tuples, and the materialization time for the two variations of tuple sending: sequential or parallel. In both cases, as expected, the extraction time is the same and it increases linearly with the number of consumers.

When sending tuples sequentially, we observe that the materialization time increases linearly with the number of consumers (views). In the case of 64 consumers, data extraction takes about 45 seconds, but materialization takes about 200 seconds. Materialization time increases drastically with sequential tuple sending since more and more consumers need to be contacted one after another.

When sending tuples in parallel, we observe that the materialization time is notably lower than in the case of sequential tuple shipping and that its slope is almost the same as the one of the extraction time. This is because, as soon as the tuples are extracted, a pool of threads (one thread for each packet of tuples) takes over the task of shipping all the tuples in parallel. The bottleneck in this situation is the upload link of each consumer.

Experiment 4: one publisher, varying data size, 64 consumers We study how materialization time is affected when the total size of published data is increased. We use one publisher. The size of the published data varies from 64MBs to 1024MBs.

Each of the 64 consumers holds one view of the form $//catalog//camera_{K\ cont}$ where K varies according to the peer that holds the view. For example, the

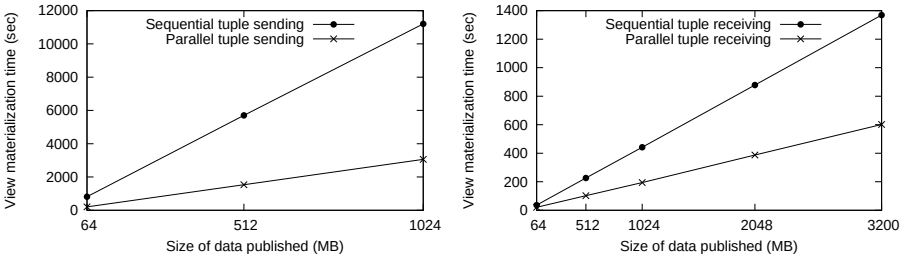


Figure 13: Experiment 4: one publisher, varying size of data, 64 consumers (left); experiment 5: 64 publishers, varying data size, one consumer (right).

first consumer holds the view $//catalog//camera_{1\ cont}$, the second holds the view $//catalog//camera_{2\ cont}$ etc. This way, from each document the publisher extracts 64 tuples, each of which is sent to a different consumer. All the content of the documents is absorbed by the 64 views. The parameter values used for this experiment are:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
65	1	64	64	$\{64, 512, 1024\}$	$\{64, 512, 1024\}$	1MB

Like in Experiment 3, we run two variations of the same experiment: (*i*) one for sequential tuple sending and (*ii*) one for parallel tuple sending. The graph at left in Figure 13 shows, as expected, that the materialization time increases linearly with the size of data published in the network in both cases. It also shows that the materialization time in the case of parallel tuple sending is considerably shorter (about 3000 sec. instead of 11500 sec. for absorbing 1024MBs of data).

Experiment 5: 64 publishers, varying data size, one consumer We now study the potential for parallel publishing, i.e., the impact of the number of (simultaneous) publishers on the capacity of absorbing the data into a single view. The published data size varies from 64MBs to 3.2GBs, and all the published data ends up in the view. The parameter values for this experiment are:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
65	64	1	1	$\{64, \ldots, 3200\}$	$\{64, \ldots, 3200\}$	1MB

Recall from Section 3.2 that the view materialization module maintains a queue of tuple-send requests and allows only a certain number of concurrent tuple-extractors to send data to it concurrently. In this experiment we test 2 modes of tuple-receiving concurrency: (*i*) the consumer accepts only one tuple-send request at any given time (sequential tuple receiving); (*ii*) the consumer accepts at most 64 tuple-send requests concurrently (parallel tuple receiving).

Figure 13 (right) depicts the materialization time as the size of the published data increases. We observe that the materialization time increases proportionally to the size of published data in both sequential and parallel tuple receiving modes. Also, parallel tuple receiving reduces the view materialization time by more than 50% (600 sec. instead of about 1400 sec. to absorb 3.2GBs of data).

From the two graphs in Figure 13, we conclude that it is faster for the network to absorb data using one consumer and many publishers rather than

many consumers and one publisher. For example, for absorbing 1024MBs of data, the view materialization time is less than 200 seconds (Figure 13 right) for 64 publishers and one consumer, and about 3000 seconds in the case of one publisher and 64 consumers (Figure 13 left). This is explained by the fact that, data extraction is proven to be a slow process (Experiment 2) thus it is slow for a peer to extract all the available data by itself and ship them to the consumers.

Experiment 6: varying number of publishers, fixed data, one consumer The purpose of this experiment is to study the impact that the parallelization of document publication has on the view materialization time. We use 250MBs of data distributed evenly across an increasing number of publishers. First, one peer publishes all the data, then two peers publish half of the data each, then 4, then 8 peers etc. The parameter values for this experiment are as follows:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
65	{1,2,...,64}	1	1	512	512	0.49MB

Figure 14 (left) shows how materialization time varies depending on the number of parallel publishers. The time decreases as the data is distributed to two and then 4 publishers, as the extraction effort is parallelized. From 8 publishers onwards, the materialization time increases again, until it stabilizes from 32 to 64 publishers. This increase is due to publishers simultaneously trying to connect to the consumer and making the consumer's storage module the bottleneck.

Experiment 7: community publishing We now consider a more complex scenario. We study materialization time in a setting with (logical) sub-networks, i.e., such that no single publisher has data of interest to all views, and no single view needs data from all publishers. The parameters of this experiment are:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
250	250	50	50	$\{20K, \ldots, 160K\}$	$\{20K, \ldots, 160K\}$	1MB

We use a network of 250 peers, each of which holds the same number of 1MB documents. We logically divide the network into 50 groups of 5 peers each, such that in each group there are five publishers and one consumer (one peer is both a publisher and a consumer). The data of all publishers in a group is of interest for the consumer of that group, but it is not relevant for any of the other groups' views. The group peers are randomly chosen, i.e., they do not enjoy any special geographic or network locality etc. The total amount of data published (and shipped to the views) varies from 20GBs to 160GBs. Figure 14 (right) shows that the materialization time grows linearly with the published data size.

Conclusion This Section has studied several extreme cases of view materialization (very skewed / very evenly distributed, with one or many publishers or consumers etc.), in order to traverse the space of possibilities. Overall, the experiments demonstrate the good scalability properties of ViP2P as the data volume increases, and that ViP2P exploits many parallelization opportunities when extracting, sending, receiving and storing view tuples. Table 2 summarizes the results by providing a global metric, the view materialization throughput, or the quantity of data that can be published (from documents to views) simultaneously in the network. Table 2 demonstrates that ViP2P properly exploits all opportunities for parallelism in the "community publishing" scenario: the

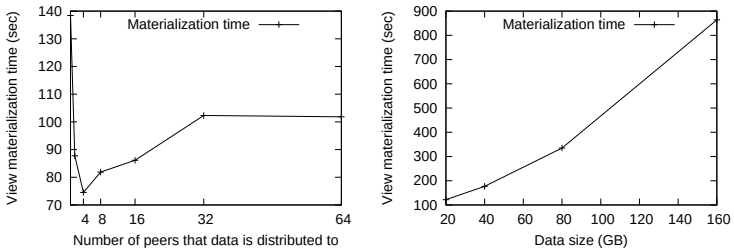


Figure 14: Experiment 6: publishing the same amount of data from an increasing number of publishers (left); experiment 7: publishing varying size of data in 50 groups of 5 peers each (right).

Exp. No.	Experiment description	Throughput (MB/sec)
3	One publisher, fixed data, varying number of consumers	10.30
4	One publisher, varying data size, 64 consumers	0.34
5	64 publishers, varying data size, one consumer	5.31
6	Varying number of publishers, fixed data, one consumer	8.05
7	Community publishing	238.80

Table 2: Maximum data absorption throughput during view materialization.

throughput is of 238 MB/s, while the best comparable result in this area from KadoP is of .33 MB/s only [3].

6.4 View indexing and retrieval evaluation

We now compare the view indexing and lookup strategies LI, RLI, LPI and RPI described in Section 5.2.

Experiment 8: view indexing and retrieval We start with a random synthetic query q of height 5, having 30 nodes labeled $a_1, \ldots, a_{30}$. Each node of q has between 0 and 2 children. We then create three variants of q:

- q' has the same labels as q, but totally disagrees with q on the structure (if a_i is an ancestor of a_j in q, a_i is not an ancestor of a_j in q')
- q'' coincides with q for half of the query, while the other half conserves the labels of q but totally disagrees on the structure (as in q')
- q''' has the same structure as q, half of it has the same labels $a_1, \ldots, a_{15}$, while the other half uses a different set of labels $b_1, \ldots, b_{15}$ (that replace $a_{16}, \ldots, a_{30}$ respectively).

From each of q, q', q'' and q''' we automatically generate 360 views of 2 to 5 nodes, totaling 1440 views, such that: the views can all be embedded into their respective queries, i.e. those generated from q can be embedded in q, those generated from q' can be embedded in q' and so on. We, thus, obtain a mix of views resembling the original query q to various degrees.

We have indexed the resulting 1440 views in a network of 250 peers, following the LI, RLI, LPI and RPI strategies described in Section 5.2. We then

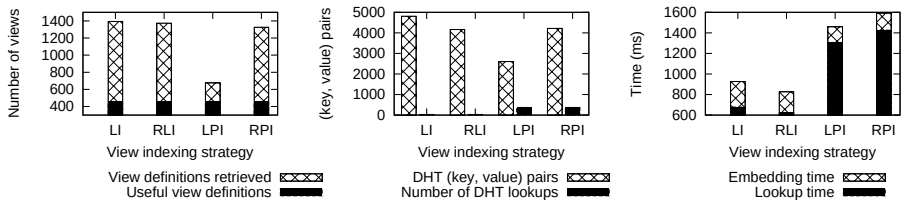


Figure 15: Experiment 8: view definition retrieval (left); index entries and lookups generated for the views (center); embedding vs lookup time(right).

performed lookups using the four different indexing strategies. The parameters characterizing this experiment are the following:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
250	0	1440	250	0	0	0

Figure 15 (left) depicts the number of views retrieved by each strategy, compared to the number of useful views, which can be embedded into q. We observe, as expected, that the path indexing-lookup strategies (LPI and RPI) are more precise than the label based ones (LI and RLI). Moreover, LPI is the most precise, since it uses as keys longer paths, describing views more precisely.

Figure 15 (center) depicts the number of (key, value) pairs indexed in the DHT by each view indexing strategy, and the number of lookups performed by each strategy for the query we consider. As expected, LI inserts the largest number of DHT entries. With respect to query-driven lookup, LI and RLI perform 30 lookups, much less than LPI and RPI that perform 370 lookups each.

Figure 15 (right) depicts the time spent looking up in the DHT the set of (possibly) useful views in order to rewrite q, as well as the time spent to check whether embeddings exist from those views into q. We observe that from this angle, the label strategies (LI and RLI) perform better than the path strategies, since the more numerous lookups performed by the path strategies take up too much time when processing queries.

From this experiment, we conclude that leaf-based strategies are preferable, since the savings at query processing time more critical than the DHT index size (which is very modest in all cases) or the precision of look-up, as the retrieved view definitions are further filtered at the query peer.

6.5 Query engine evaluation

Experiment 9: query response time vs. query selectivity and number of results We now investigate the query processing performance as the data size increases. We use 20 peers, all of which are publishers, 2 are consumers and 1 is a query peer. The query peer and the 2 consumers are located in 3 different locations of France (Bordeaux, Lille and Orsay). The parameter values characterizing this experiment are the following:

$\|P\|$	$\|P_D\|$	$\|V\|$	$\|P_V\|$	$\|D\|$	$\|D_V\|$	$\|d\|$
20	20	2	2	$\{20, \ldots, 500\}$	$\{20, \ldots, 500\}$	0.5MB

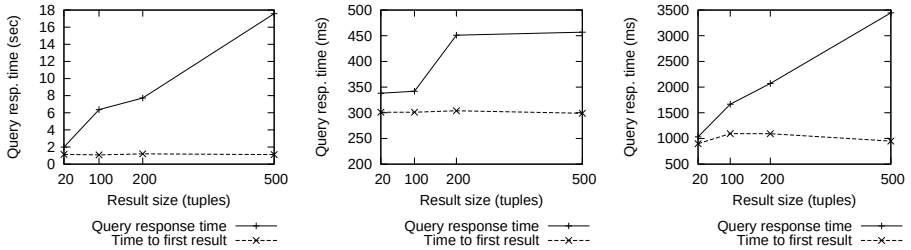


Figure 16: Experiment 9: query execution time vs. number of result tuples for three queries.

The document used in this experiment is the same as Figure 12 (left) with a slight difference: the root element *catalog* has only one child, named *camera*.

The views defined in the network are the following:

- v_1 is $//catalog_{ID}//camera_{ID}//description_{ID,cont}$
- v_2 is $//catalog_{ID}//camera_{ID}//\{description_{ID}, price_{ID,val}, specs_{ID,cont}\}$

Each view stores one tuple from each document. A v_1 tuple from document d roughly contains all of d (since the *description* element is the most voluminous in each *camera*). A v_2 tuple is quite smaller since it does not store the full camera descriptions. We use three queries:

- q_1 asks for the $description_{cont}$, $specs_{cont}$ and $price_{val}$ of each *camera*. To evaluate q_1, ViP2P joins the views v_1 and v_2. Observe that q_1 returns full XML elements, and in particular, product descriptions, which are voluminous in our data set. Therefore, q_1 returns roughly all the published data (from 10MB in 20 tuples, to 250MB in 500 tuples).
- q_2 requires the $description_{ID}$, $specs_{ID}$ and $price_{ID}$ of each *camera*. This is very similar to q_1 but it can be answered based on v_2 only. The returned data is much smaller since there are only IDs and no XML elements: from 2KB in 20 tuples, to 40KB in 500 tuples.
- q_3 returns the $specs//sensor\_type_{val}$ of each *camera*. The rewriting of q_3 applies *navigation* over $specs_{cont}$ that is stored by v_2. The result size varies from 2KB in 20 tuples to 40KB in 500 tuples.

Figure 16 shows the query response time and the time to get the first result for the 3 queries. The low selectivity query q_1 (at left in Figure 16) takes longer than q_2, due to the larger data transfers and the necessary view join. The time to first result is always constant for both q_1 and q_2 and does not depend on the result size. For q_1, a hash join is used to combine v_1 and v_2, and thus no tuple is output before the view v_2 has been built into the buckets of the hash join. This is done in more or less one second in the case of q_1 and about 300ms for q_2. Note that the join is performed on the peer holding v_1 as it is faster to transfer v_2 at the peer holding v_1. Increases in the total running time appear when more data-sending messages are needed to transfer increasing amounts of results. For q_3, which applies navigation on the view v_2, the time to the first

tuple is the time to evaluate the navigation query locally at v_2's peer and send the first message with result tuples to the query peer, and this does not grow with the data size.

Conclusion The ViP2P query processing engine scales quite linearly answering queries in a wide-area network. The fact that ViP2P rewrites queries into logical plans which are then passed to an optimizer enables it to take advantage from known optimization techniques used in XML and/or distributed databases, to reduce the total query evaluation time, and (depending on the characteristics of the particular physical operators chosen) the time to the first answer. Given the ViP2P architecture, the peers involved in processing a query are only those holding the views used in the query rewriting; this is why using only 20 peers for this experiment does not affect its interpretation, since ViP2P query processing involves only three peers. The network size may only impact the view look-up time, which is very modest (Section 6.4).

6.6 Conclusion of the experiments

Our study leads to concluding that the ViP2P architecture scales up well. Thus, view materialization scales in the number of publishers and consumers, in the size of the network, and in the size of the data; high contention at a single consumer receiving data from many publishers, and especially at a single publisher contributing to many consumers' views, degrades the capacity to vehiculate data across the peer network, to be absorbed by the views. These contention effects to be expected in a large distributed system. We show that when interest in the published data is more evenly distributed among sub-communities, ViP2P takes advantage of all parallelization opportunities to increase the data transfer rate between publishers and consumers by 3 orders of magnitude. Our view materialization experiments also show the importance of carefully tuning all stages in the data extraction and data transfer process, including asynchronous communication and parallelization whenever possible. The cumulated impact of these optimizations on the data transfer rate between peers are dramatic (more than 4 orders of magnitude increase).

Our query processing experiments show that label-based view indexing strategies are preferable, and indeed we use RLI by default. They also demonstrate that the ViP2P execution engine scales linearly up to large data volumes, orders of magnitude more than in previous real DHT deployments [3, 33].

7 Conclusion and perspectives

The efficient management of large XML corpora in structured P2P networks requires the ability to deploy data access support structures, which can be tuned to closely fit application needs. We have presented the VIP2P approach for building and maintaining structured materialized views, and processing peer queries based on the existing views in the DHT network. Using DHT-indexed views adds to query processing the (modest) cost of locating relevant views and rewriting the query using the views, in exchange for the benefits of using precomputed results stored in views. We studied several view indexing strategies and associated complete view lookup methods. Moreover, we did an extensive study of our platform's main aspects (view materialization, indexing and retrieval, and query processing) in different scenarios and settings. ViP2P was able to extract and disseminate 160GB of data in less than 15 minutes over 250 computers in a WAN network [1]. These results largely improve over the closest competing XML management platforms based on DHTs, and actually implemented and deployed (1 GB of data indexed in 50 minutes in KadoP [3], hundreds of MB of data on 11 peers in psiX [33] which focused only on document indexing and look-up).

Many avenues for further research are open. An ongoing work built on ViP2P, LiquidXML [12] automatically selects and continuously adapts a set of materialized views on each peer, to improve query processing performance in the network. Handling documents that contain references to each other and evaluating tree pattern queries that extend to many documents are other interesting developments.

Acknowledgements Part of the ViP2P code comes from ULoad [6]. We thank Alin Tilea, Jesús Camacho-Rodríguez, Alexandra Roatis, Varunesh Mishra and Julien Leblay for their help developing and testing ViP2P. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

Contents

References

[1] Grid'5000 network infrastructure. https://www.grid5000.fr/.

[2] K. Aberer. Peer-to-peer data management. *Synthesis Lectures on Data Management, Morgan & Claypool Publishers*, Volume 3:p.87–94, 2011.

[3] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, 2008.

[4] L. Afanasiev, I. Manolescu, and P. Michiels. MemBeR: A Micro-benchmark Repository for XQuery. In *EXPDB*, 2005.

[5] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[6] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured Materialized Views for XML Queries. In *VLDB*, 2007.

[7] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *SIGMOD*. ACM, 2002.

[8] Oracle Berkeley DB Java Edition. http://www.oracle.com/technetwork/database/berkeleydb/overview/.

[9] A. Bonifati and A. Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data Knowl. Eng.*, 59(2), 2006.

[10] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *WIDM*, 2004.

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

[12] J. Camacho-Rodríguez, A. Katsifodimos, I. Manolescu, and A. Roatis. LiquidXML: Adaptive XML Content Redistribution. In *CIKM (demo)*, 2010.

[13] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. An indexing framework for peer-to-peer systems. In *SIGMOD*, 2004.

[14] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-Ring: An efficient and robust P2P range index structure. In *SIGMOD*, 2007.

[15] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *IPTPS*, 2003.

[16] W. K. Dedzoe, P. Lamarre, R. Akbarinia, and P. Valduriez. ASAP Top-k query processing in unstructured P2P systems. In *Peer-to-Peer Computing*, 2010.

[17] Freepastry, an open-source implementation of Pastry. http://freepastry.org/FreePastry/.

[18] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB*, 2003.

[19] O. Gauwin, J. Niehren, and S. Tison. Bounded delay and concurrency for earliest query answering. In *LATA*, 2009.

[20] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD Conference*, 1990.

[21] H. Jagadish, B. Ooi, K. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *SIGMOD*, 2006.

[22] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: a balanced tree structure for peer-to-peer networks. In *VLDB*, 2005.

[23] K. Karanasos and S. Zoupanos. Viewing a world of annotations through AnnoVIP. In *ICDE (demo)*, 2010.

[24] G. Koloniari and E. Pitoura. Peer-to-peer management of XML data: issues and research challenges. *SIGMOD Record*, 34(2), 2005.

[25] K. Lillis and E. Pitoura. Cooperative XPath caching. In *SIGMOD Conference*, 2008.

[26] 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.

[27] I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, 2011.

[28] I. Manolescu, Y. Papakonstantinou, and V. Vassalos. XML tuple algebra. In *Encyclopedia of Database Systems*, pages 3640–3646. Springer, 2009.

[29] I. Manolescu and S. Zoupanos. Materialized views for P2P XML warehousing. In *BDA (informal proceedings)*, 2009.

[30] I. Manolescu and S. Zoupanos. XML materialized views in P2P. In *DataX workshop (not included in the proceedings)*, 2009.

[31] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1), 2004.

[32] P. Rao and B. Moon. An Internet-Scale Service for Publishing and Locating XML Documents. In *ICDE (demo)*, 2009.

[33] P. R. Rao and B. Moon. Locating XML documents in a peer-to-peer network using distributed hash tables. *IEEE TKDE*, 21, 2009.

[34] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *ICDSP*, Nov. 2001.

[35] C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. Xpeer : A self-organizing xml p2p database system. In W. Lindner, M. Mesiti, C. TÃ¼rker, Y. Tzitzikas, and A. Vakali, editors, *Current Trends in Database Technology - EDBT 2004 Workshops*, Lecture Notes in Computer Science, pages 429–432. Springer, 2005.

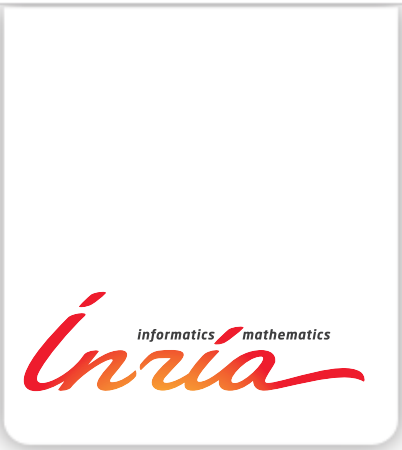
[36] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *VLDB*, 2000.

[37] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient processing of XPath queries with structured overlay networks. In *OTM Conferences (2)*, 2005.

[38] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple Materialized View Selection for XPath Query Rewriting. In *ICDE*, 2008.

[39] XPath Functions and Operators. www.w3.org/TR/xpath-functions, 2007.

[40] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *ICDE*, 2003.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

Parc Orsay Université
4 rue Jacques Monod
91893 Orsay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399