Interoperability in Peer Data Management Systems[*]

Katja Hose
Faculty of Computer Science
and Automation
TU Ilmenau
Germany
katja.hose@tu-ilmenau.de

Jana Quasebarth
Faculty of Computer Science
and Automation
TU Ilmenau
Germany
jana.quasebarth@tu-ilmenau.de

Kai-Uwe Sattler
Faculty of Computer Science
and Automation
TU Ilmenau
Germany
kus@tu-ilmenau.de

ABSTRACT

Interoperability plays an important role for a variety of applications. One of them are Peer Data Management Systems, where autonomous data sources (peers) interact with each other based on semantic mappings between their schemas. The building blocks that enable interoperability and thus the main challenges in such systems are mapping representation, query rewriting, and efficient query processing. While most approaches regard these aspects in separate this paper presents a comprehensive study of the interactions between these blocks. Our considerations try to provide a holistic view on semantic interoperability in distributed environments such as PDMS. We discuss techniques for distributed query processing and rewriting that consider high-level query operators such as top-N and skyline. Furthermore, we discuss how to increase efficiency by applying routing indexes and relaxation of result completeness/correctness.

1. INTRODUCTION

In recent years there has been a lot of research with respect to semantic data integration. Eventually, Peer Data Management Systems (PDMS) have attracted attention as they promise to combine aspects from P2P systems with data integration. PDMS consist of autonomous peers representing data sources that are semantically connected via mappings. Each peer is allowed to use its individual local data schema so that it might be unique in the whole network. Thus, it is one of the main problems in PDMS to define mappings between schemas. Several techniques to define such mappings have been proposed for data integration in general and can be used in PDMS. The most popular approaches are local-as-view (LAV) and global-as-view (GAV) [8, 18].

Each peer and its neighborhood in a PDMS can be regarded as a stand-alone data integration system where the local schema of the peer is regarded as the global schema in the integration system. Figure 1 shows an example with highlighted data integration systems for peers P_0, P_8, and P_{10}. Because of the overlap, queries and data can be exchanged between the integration systems.

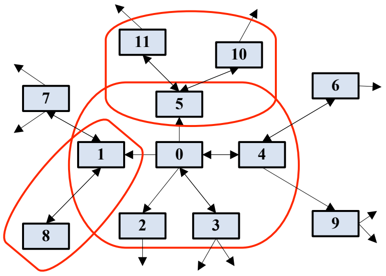


Figure 1: Peer Data Management System

Peers in PDMS are not only autonomous with respect to local data sovereignty but also in the way they participate in the network. There are no designated peers with special roles. On the contrary, all peers are equal and able to issue and answer queries. PDMS are not assumed to be static but have to deal with dynamic behavior: peers might update their local data and leave/join the network at any time. As PDMS are built upon unstructured P2P systems of autonomous peers, we should not assume the existence of a central instance or any other kind of global knowledge. Thus, query processing and especially query optimization has to take place in a completely decentralized fashion with the only help of knowledge that is available locally at each peer.

Although several systems propose a variety of techniques and algorithms for query rewriting, most of them simply disregard lessons learned from distributed query processing in P2P networks. Thus, although there is no global knowledge peers must nevertheless try to minimize query execution costs. One possibility to reduce network load and consequently costs is to reduce the number of peers that the query is forwarded to. In most PDMS peers only prune neighboring peers from consideration if the query cannot be rewritten into the schema of the neighbor (schema-level pruning). However, additional peers can be pruned by using routing indexes [4] (data-level pruning). Doing so each peer holds information about the data that is provided by its neighbors. Based on that information a peer can decide whether a neighbor may contribute to the result of a given query. If not, the peer does not need to be involved in processing the query.

In P2P systems retrieving all results for a given query might end up in a huge result set with a lot of records that the user does not need. Thus, applying a ranking function and limiting the size of the result set not only reduces the number of needless result records but may also help to reduce execution costs. Representatives for the class of rank-aware queries that we consider in this paper are top-N and skyline queries [3]. The additional application of relaxation [14] might reduce execution costs even further.

In this paper, we present an overview of SmurfPDMS [9, 13] in that we consider all aspects mentioned above. In contrast to other PDMS peers in SmurfPDMS apply schema-level pruning as well

[*] This work was supported by the BMBF under grant 03WKBD2B.

International Workshop on Semantic Data and Service Integration (SDSI'07), September 23, 2007, Vienna, Austria..

as data-level pruning. Furthermore, we consider rank-aware query operators not only on top of conjunctive queries but as part of the query that the rewriting process pays attention to. Finally, we also propose the use of relaxation to further reduce execution costs.

This paper is structured as follows. At first, we review related work in Section 2. After Section 3 has defined the basic model of our system, Section 4 discusses some high-level query processing techniques that work on top of query rewriting. The algorithm that we use for query rewriting is presented in Section 5. Section 6 presents some results of our evaluation. Finally, Section 7 concludes this paper.

2. RELATED WORK

In this section we briefly discuss some existing approaches for PDMS. We limit our considerations to the most prominent examples.

2.1 PDMS

Piazza [7] is a PDMS that aims to combine the two data integration formalisms LAV and GAV into one system. Data is provided by the data sources either in XML or RDF-based formats. Queries are formulated using a fragment of XQuery. Piazza knows two types of schema mappings: *peer descriptions* and *storage descriptions*. The former relate two or more peer schemas (schemas that peers publish to make their local data accessible for other peers) whereas the latter relate peer schemas and stored schemas (schemas of the local data that the peers possess). There are two kinds of peer descriptions: *equality descriptions* ($Q_1(P_1) = Q_2(P_2)$ with Q_1 and Q_2 being conjunctive queries with the same arity and P_1 and P_2 being sets of peers) and *inclusion descriptions* ($Q_1(P_1) \subseteq Q_2(P_2)$), where equality can always be regarded as two inclusions.

Queries are given in the form of conjunctive queries. A conjunctive query can be considered as a logical function or rule applied to the relations of a database. An example according to [19] is:

```
q(director) :- Movie(title,director,year) &
   Oscar(title,year1) & year1 ≥ 1965
```

`q(director)` is called the *head* of the query and its argument `director` is its *distinguished variable*. The distinguished variables of a query correspond to attributes appearing in the SELECT clause of a corresponding SQL query. `Oscar(title,year1)`, `Movie(title,director,year)`, and `year1 ≥ 1965` are *atoms* in the *body* of the query. Predicates of the WHERE clause of an SQL query can be expressed by a comparison operator in conjunction with either two variables or a pair of one variable and one constant value, e.g., `year1 ≥ 1965`. Equality predicates are represented by multiple occurrences of the same variable in different atoms, e.g., `title` and `year1` in the example given above. Since atoms in conjunctive queries are connected with logical *and* operators unions are expressed by multiple conjunctive queries having the same head.

For rewriting such queries Piazza uses two techniques with respect to GAV and LAV mappings: *query unfolding* and *query answering using views*. The basic idea of query unfolding in GAV is to construct a tree of query subgoals and expand each query subgoal recursively according to the relevant peer mappings. In the case of LAV mappings the *MiniCon* algorithm [6] is used. Both techniques are combined and applied recursively on the result until no more rewriting is possible using peer descriptions. Then, the storage descriptions are used for the final step of reformulation. The result of this rewriting process is a query on stored relations only (i.e., relations included in the storage descriptions).

Piazza uses a centralized index – that can be regarded as a summary of the data stored at the peers. Each participating peer uploads a summary of its local data to a central instance (index engine) and refreshes it periodically. Users perform searches by submitting queries to the index engine (that also knows all peer mappings). The index consists of objects that contain sets of attribute-value pairs of the form: $d ::= [A_1 = v_1, A_2 = v_2, \ldots, A_n = v_n]$ where $A_1, \ldots, A_n$ are attributes and $v_1, \ldots, v_n$ are atomic values or patterns (with wildcards). Using the peer descriptions relationships between attributes can be derived and queries of the form $q = [B_1 = w_1, B_2 = w_2, \ldots, B_p = w_p]$ with attributes $B_1, \ldots B_n$ and constants $w_1, \ldots, w_n$ can be supported.

In contrast to Piazza, we strictly avoid any kind of central instance that might represent a single point of failure. We argue that the peers are not willing to share their mappings and thus information about their local schema with a central instance that they cannot control. On the contrary, we assume that peers want to keep exclusive control over what peers know about their local schemas and data. Thus, although we can reuse some of the techniques, we need to apply other concepts since appropriate techniques for rewriting and query planning in SmurfPDMS have to work in a completely decentralized manner.

Just like Piazza, **SystemP** [25] uses both LAV and GAV mappings. Likewise, mappings as well as queries are formulated as conjunctive queries using datalog rules. However, although SystemP adopts the rule-goal-tree approach of [8] – that is also used by the Piazza system – SystemP implements query rewriting in a decentralized manner without any kind of global optimization. For query processing SystemP provides a budget-driven approach, where peers are assigned budgets for query answering. The main idea is to prefer peers that potentially contribute a large number of records to the result. Result cardinalities are estimated using multidimensional histograms [1] – applying query feedback to keep them up-to-date.

Using budgets and cardinality estimation, SystemP tries to reduce execution costs by trading off result completeness. On the contrary, our approach applies the idea of routing indexes [4] or rather distributed data summaries [12]. Doing so our optimization techniques are not only restricted to trading off result completeness but may also trade off result quality – neglecting result records according to a ranking function which indicates their importance to the user. Hence, our approach is not limited to only select-project-join queries but also integrates rank-aware query operators. Considering such operators for rewriting is another difference to both Piazza and SystemP.

Another two systems worth mentioning in this context are **Hyperion** [24] and **HePToX** [2] because of their peculiarities. Hyperion uses a completely different approach to formulate mappings between peers. For this purpose, mapping tables [16] are used that list pairs of corresponding values for data residing on different peers. Mapping tables represent expert knowledge and are typically created by domain specialists. HePToX users define mappings using a graphical interface drawing a set of visual annotations between DTDs for the peers' local XML data. Mapping rules are then derived automatically from these annotations. Mappings are represented as datalog-like rules adapted to tree structured data. Rewriting is realized with algorithms related to those used by Piazza.

3. MODEL DEFINITION

Data integration and view-based query rewriting have been investigated for relational databases and stable systems [6, 26]. The two general approaches are global-as-view (GAV) and local-as-view (LAV) [18], both using a mediated global schema to integrate data from different sources. Queries are formulated in the mediated schema. In GAV each global relation of the mediated schema is designed as a view over the local data sources. Query processing results in view unfolding or view expansion. As each joining and leaving of data sources leads to a change of the view definitions, the GAV approach is not applicable to dynamic networks. On the contrary, in LAV the local data sources are described by views over the relations of the mediated schema. Dynamics only affect the view definitions related to one peer. Using the LAV approach results in the need for query reformulation [6]. Since we want to provide a general solution applicable to dynamic environments, we decided to use LAV style mappings for SmurfPDMS. Consequently, mappings are directed and expressed by views. That is why in the following we use the two terms mapping and view as synonyms. However, additionally integrating GAV mappings and query unfolding would be straightforward.

As native data format for SmurfPDMS we use XML because of its immense popularity for many applications that exchange data. Besides, most database systems offer the possibility to export data in XML format such that also non-XML data sources can participate and make their data accessible. Based on this assumption this section presents how queries and mappings between peers are formulated in SmurfPDMS.

Figure 2 illustrates an example network consisting of 7 peers. The data that the peers share describes artists and their works. Arrows between peers indicate that a mapping between them exists. The direction of the arrow indicates the direction of the mapping, e.g., there exists a mapping from P_0 to P_1 but not vice versa. In this example, P_0 has mappings to all other peers. Thus, this network comes very close to a standard data integration scenario with a mediator. However, since it is a PDMS queries can be issued at any peer in the system. Furthermore, peers might leave the network at any time and peers joining the network do not have to establish mappings to P_0. We will use this network as a running example throughout this paper and get back to it later.

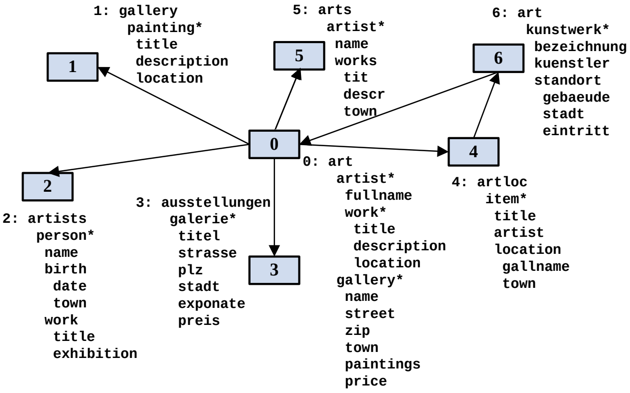
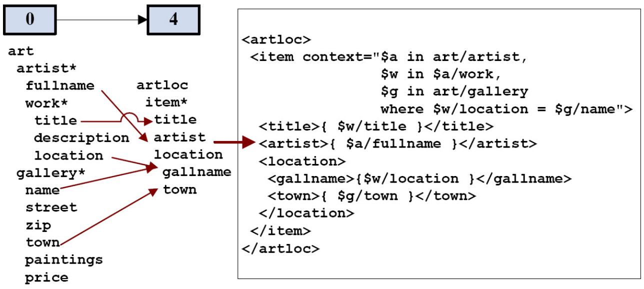


Figure 2: Example Network

3.1 Mapping/View Definition

Although we are aware that mappings might change – replaced or improved – over time, we do not yet consider this aspect in our system. However, the integration of this aspect is straightforward as long as the replacement or improvement originates from a user interaction. Future work might consider this aspect and provide techniques for automatic schema matching [23]. In this context future work might also address the consequences of information loss that originates from incorrect or incomplete mappings. However, for the time being we assume that mappings are correct and complete.



```
<artloc>
 <item context="$a in art/artist,
                $w in $a/work,
                $g in art/gallery
                where $w/location = $g/name">
  <title>{ $w/title }</title>
  <artist>{ $a/fullname }</artist>
  <location>
   <gallname>{$w/location }</gallname>
   <town>{ $g/town }</town>
  </location>
 </item>
</artloc>
```

Figure 3: Mapping Representation with View Definitions

Let us consider a sample LAV mapping that P_0 uses to rewrite queries from its local schema into the schema of P_4. Like Piazza [7] we use XQuery-like blocks in our view definitions. A view definition is well-formed and uses the XML structure corresponding to the schema of the neighboring (target) peer as basis (P_4 in our example). Such a definition contains XQuery-like elements (*for*, *where*, and *return* clauses). The *for* clause defines a set of context nodes. These are nodes contained in the schema of the source peer (P_0). The *where* clause contains predicates that describe how the data is stored at the target peer. For example a predicate with a constant ("`$art/artist/fullname='Leonardo da Vinci'`") indicates that the target's local data is restricted – in this case to information about Leonardo da Vinci. A join in a *where* clause indicates that local structures of the source are stored in a joined format at the target. Both *for* and *where* clauses are contained in an XML attribute named *context*.

Figure 3 shows the mapping that P_0 uses to rewrite queries into the schema of P_4. The *context* attribute is assigned to an XML node *item* that is part of P_4's local schema. This declares *item* as the target context node. For simplicity, we omit the keyword "*for*". In our example the *for* clause defines the following source context nodes and symbols: `$a in art/artist, $w in $a/work, $g in art/gallery`. The *where* clause defines the predicate `$w/location = $g/name` that represents a join in the schema of P_0.

In general, the context attribute is assigned to an XML element and tags the parent element that contains all elements of the *return* clause. However, we do not have an explicit *return* clause and mark the insertion of text with braces {...} enclosing exported symbols (paths in the XML document) that refer to the content of the source context nodes defined in the *for* clause, e.g., {`$w/title`}.

Nesting of views is not allowed, i.e., in the descendent axis of an XML element with a *context* attribute there must not exist any other element with a *context* attribute. Additionally (not shown in Figure 3), a view definition might contain constraints. For example, assume that P_1 defines a mapping to P_0 (Figure 2 shows more details about the local schemas of P_0 and P_1). P_1 does not store the name of the artist since it only stores data about works of Leonardo da Vinci. Thus, a mapping from P_1 to P_0 (defined using the local schema of P_0 as basis) must contain additional information about the restriction on works of Leonardo da Vinci. As a consequence, a *constraint* attribute would be integrated into the mapping, e.g.:

```
<art>
 <artist context="$p in gallery/painting">
  <fullname constraint="($p)='Leonardo da Vinci'" />
  <work>...</work>
 </artist>
</art>
```

As a counterpiece to this constraint P_0 would have the following predicate in its view definition to P_1:

$art/artist/fullname='Leonardo da Vinci'.

In summary, view definitions consist of: literals defining context nodes ($a), symbols ($a/work), exported symbols ({$w/title}), constants ('Leonardo da Vinci'), conditions corresponding to predicates ($w/location = $g/name), and constraints ((\$p) = 'Leonardo da Vinci').

3.2 Query Formulation

Queries in SmurfPDMS are formulated using plan operators (POPs). POPs can be combined resulting in POP trees that represent queries. In SmurfPDMS we do not explicitly distinguish between query plans and queries: the POP tree that is given as input is assumed to be preoptimized by an external component. However, integrating such a component is part of our future work. The result of a query is processed in a bottom-up fashion such that leaf nodes are computed first. The parent POP uses the output of its child (a sequence of XML structures) as input. Finally, the root node of the tree computes its result. This represents the answer to the query that is output to the user or forwarded to a neighboring peer as an answer to the query.

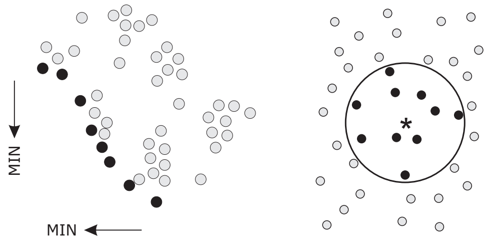
There are only a few basic rules that a query must adhere to. As mentioned above, the POP representation corresponds to a tree structure, i.e., each POP has exactly one parent POP except the root node. The number of children depends on the type of the POP. Leaf nodes of the tree are always *select* POPs because only these POPs are allowed to refer directly to the local data of a peer. Thus, they select the data that all POPs on higher levels operate on. Consequently, they represent a starting point where select expressions (XPath) are evaluated on the peers' local data. Table 1 summarizes the POPs that are currently supported by SmurfPDMS.

POP	#children	parameters
select/project	1	1 XPath expression
union	2	—
join	2	1 condition
skyline	1	$\geq$ 2 ranking functions
topn	1	integer n, 1 ranking function
construct	1	1 expression
remote query	1	neighborID

Table 1: List of Algebra Operators

There are only two POPs that require two child POPs: the *union* POP, which unifies the two result sets of its child POPs, and the *join* POP. The *join* POP receives two sequences of XML structures as parameter and joins any two XML structures that originate from different sets and fulfill the join condition. All the other POPs may only have one child POP. Since the *select* POP is given an XPath expression as input it not only performs selection but also projection and may also occur as inner node of a query POP tree. The *construct* POP is used to restructure XML data according to the associated expression. For example, assume the following expression is assigned to a *construct* POP: <res><artist>{artist/name}</artist> <painted>{artist/work/title}</painted></res>. In this example, all XML structures in the input sequence represent data records about artists (identifiable by their name artist/name) and their works (artist/work/title). For each input XML structure the artist's name and the title of his/her work is selected and transformed into another structure with res as outer element, which has two child elements artist (containing the artist's name) and painted containing the title of the artist's work. The result set of this *construct* POP contains one such XML structure for each input element.

SmurfPDMS also supports high-level query operators such as skyline [3, 14] and top-N [10, 11]. These rank-aware operators are represented by *topn* and *skyline* POPs. Figure 4 shows examples for both operators. Data records are represented as grey shaded circles, those belonging to the result set are shaded black. Figure 4(a) shows the example of a two-dimensional skyline where the two ranking functions simply state that the attribute values in both dimensions are to be minimized. However, much more complex ranking functions – even using multiple attributes – are possible. Formally, the result of the skyline operator is defined as the set of records that are not dominated by any other record. One record dominates another one if its attribute values are at least as good in all dimensions and better in at least one dimension. This means that the result set of a skyline contains those records that represent "good" combinations with respect to the ranking functions. Figure 4(b) shows an example of a top-N query that queries those 10 ($N = 10$) records that are closest to the asterisk. Thus, the corresponding ranking function in this example is defined as the Euclidean distance to a two-dimensional point in the data space.



(a) Sample Skyline Query (b) Sample Top-N Query

Figure 4: Examples for Rank-Aware Query Operators

In contrast to all other POPs the *remote query* POP cannot be used to formulate queries by the user. This POP is only generated by the rewriting algorithm (Section 5) to denote those query subtrees that are forwarded to neighbor peers.

To enable user interaction with the system queries can be issued in a console window in text format. The following example is issued at P_0 and queries a skyline over galleries that offer many but cheap paintings:

```
(skyline 'MIN("gallery/price")'
'MAX("gallery/paintings")'
  (select 'art/gallery')
```

The upper part of Figure 7(c) shows the correct POP tree representation.

4. QUERY PROCESSING

The main difference to query processing in P2P systems is that answers to a query must be routed back the same way as the query. The reason is that not only the query itself needs to be rewritten but also the data records of an answer such that finally the initiator receives a set of records in its local schema. An interesting approach would be to combine mappings that are used to rewrite the query on its way from the initiator to its destination (by applying schema composition techniques). Then, the destination peer could rewrite the answer records and send them directly to the initiator – or derive a permanent mapping to it. However, investigating possible solutions remains future work.

In P2P systems there are two basic approaches for processing queries: *data shipping* (DS) and *query shipping* (QS) [17]. In case all data that is identified as being relevant to a query is sent/shipped to the initiator and all operators are applied at that side, we are talking of data shipping. Applying the query shipping approach the operators are applied at the peers where the data resides on. Only the data that cannot be processed any further is shipped to the initiator. It is obvious that the query shipping approach is more efficient in terms of network load und thus execution costs.

However, query shipping has great difficulties in dynamic environments since peers have to wait for the answers of all those peers that they have forwarded the query to. Not until all these neighbors have answered a peer sends its own answer. In the presence of dynamic behavior this strategy is problematic since peers have to wait for a time-out before sending the answer. Consequently, the user has to wait a long time. In [15] we have proposed an incremental strategy (*incremental message shipping*, IMS) to overcome these shortcomings. Peers forward result records as soon as they are known. Thus, a queried peer is likely to send more than just one answer message. Consequently, this results in a higher number of messages but has the advantage that even when peers crash the answers arrive at the initiator and first results are output to the user at an early stage.

Whatever strategy is used a peer still needs to identify neighbors that are likely to provide relevant data to a given query with the help of local information. In structured (DHT-based) P2P networks – where data is distributed among peers according to a common rule, e.g., a hash function – that rule can be used to route queries efficiently to only those peers that contribute to the result. As SmurfPDMS assumes an unstructured P2P network to underly a PDMS, redistributing or replicating data is not possible. On the contrary, peers have to make routing decisions with the only help of schema mappings (to the schemas of neighboring peers on schema-level) and routing indexes (describing the data of a neighbor on instance-level). However, it is still possible to consider a super peer architecture where only the super peers as a backbone network participate in a PDMS. Then, super peers make the data of their subordinate peers accessible to other super peers in the network.

Routing indexes are data summarizing structures that each peer maintains for its neighbors (one routing index per neighbor). In their original sense routing indexes [4] are used to index files in P2P environments based on a set of keywords. Given an indexed keyword the routing index provides information about how many files that contain the keyword can be accessed by forwarding the query to the corresponding neighbor peers. Note that this information is not restricted to the data of the neighbor itself but also subsumes the data that is stored in a distance of several hops but accessible via the neighbor.

Later works [20] extended this concept to index semi-structured data. As base structure no longer keyword-hits lists are used but one-dimensional histograms. Given a range query these histogram based routing indexes can be used to effectively determine which neighbors can provide relevant data and which not – enabling data-level pruning. In [12] we have identified a class of routing indexes with qualities that make them applicable to rank-aware queries such as top-N and skyline. In summary, a suitable base structure fulfills the following requirements: summarization of records reducing memory consumption, support of efficient lookups, caption of attribute correlations, straightforward maintenance and construction, and applicability to a wide range of query operators. Although in principle any data summarizing structure can be used as basis for routing indexes and the techniques presented in this paper can be used with any such structure, we focus on multidimensional histogram-based structures as they excellently fulfill all these requirements. In accordance with [12] we call this class of routing indexes *Distributed Data Summaries* (*DDS*).

In SmurfPDMS routing indexes are defined on the local schema and index one or multiple attributes. Since queries are also formulated in the local schema the information provided by the routing indexes can easily be used by query processing strategies.

4.1 Strategy

Assuming that each peer has only a small number of neighbors and that for each of these neighbors the peer holds a mapping and a routing index, each peer reacts the same way upon receiving a query. That reaction adheres to the algorithm sketched in Figure 5.

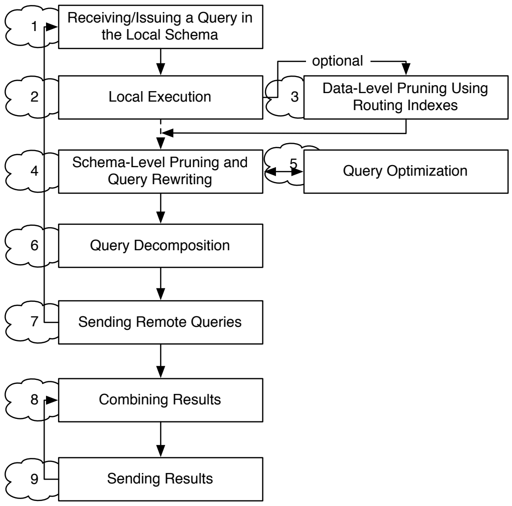

1
Receiving/Issuing a Query in the Local Schema
2
Local Execution
optional
3
Data-Level Pruning Using Routing Indexes
4
Schema-Level Pruning and Query Rewriting
5
Query Optimization
6
Query Decomposition
7
Sending Remote Queries
8
Combining Results
9
Sending Results

Figure 5: Query Processing in SmurfPDMS

1. Receiving/Issuing a Query and 2. Local Execution. At first, a query is issued or received in a peer's local schema. This is always true since the query's sender always rewrites the query into the receiver's local schema before forwarding the query. Then, the query is processed locally applying it to the peer's local data. As already mentioned in Section 3.2 the query POP tree is processed in a bottom-up fashion.

3. Data-Level Pruning. After having processed the query locally a peer needs to identify a set of neighbors to forward the query to. The simplest set consists of all neighbors and is the starting point for this step that uses routing indexes to prune neighboring peers from consideration – performing pruning on data-level. However, routing indexes are optional. Thus, if there are none available we can still use the same strategy for query processing and simply go to step 4. Depending on the top-level operator of the query and the result records that have already been determined in the last step, some of the neighbors can be pruned [14]. Especially for range queries and rank-aware query operators data-level pruning is extremely useful to reduce costs as the following example shows.

Assume a simple query with a tree of only two levels: at the bottom level is a select operator and at the top level a top-N operator with a ranking function and a number n. The routing index indicates that only two out of three neighbors provide relevant data that matches the select expression. This already means that the third neighbor can be pruned. In the best case the indexes indicate that none of the remaining neighbors can provide any result record that could be ranked better than those n result records found in step 2. In that case the query would not have to be forwarded at all and the result could already be output to the user. In a less lucky case there is still the chance that one of the remaining two neighbors can be pruned.

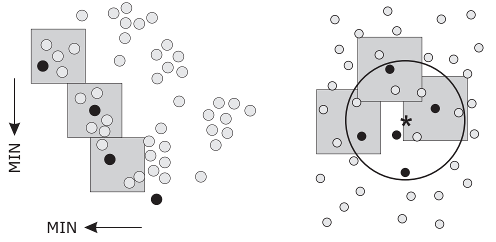
4. Schema-Level Pruning, Query Rewriting, and 5. Query Optimization. Due to the fact that some of the neighbors might have been pruned in the previous step, the overhead for rewriting queries may be reduced so that only a subset of neighbors and thus mappings have to be considered. The query operator tree is then rewritten (Section 5) into a plan that contains *remote query* POPs each representing the root nodes of a subquery tree. As subqueries and original query are combined with *union* POPs, the result of the rewriting step still contains the original query with the local result – attached to the root node of the original query.

6. Query Decomposition and 7. Sending Remote Queries. The rewritten query operator tree is decomposed into several remote queries – each rooted by a *remote query* POP. Until all receiving peers have answered, the sender needs to remember the complete query plan. Thus, it is stored into a local cache. Since it is possible that the rewritten query contains the same subquery multiple times, a peer needs to identify such subqueries before sending them to the corresponding neighbor so that the query is sent and processed only once.

8. Combining and 9. Sending Results. Since a remote query always has a construct operator as root node that transforms the results into the schema of the query's sender, no further adaptions to the results are necessary (Section 5). The received results are inserted into the cached query plan replacing the corresponding subquery plans. Now the parent operators take the results as input, use them to compute their results and propagate them to their parent POPs and so on. Although this basics hold for any concrete query processing strategy we have to distinguish between incremental and non-incremental strategies (QS and IMS). When an answer from a neighbor is received using a non-incremental strategy, the answer is stored into the cache. The final result is computed not until the last queried neighbor has sent its answer. On the contrary, using an incremental strategy the local result is sent as an answer even before any queried neighbor has answered. Further messages containing the "new" result records are sent whenever an answer from a neighbor is received.

4.2 Relaxation

As we have already argued in the introduction relaxing correctness/completeness requirements may help to reduce execution costs even more. This is especially true with respect to rank-aware query operators such as top-N and skylines. The basic concept of relaxation is the same for both operators. We illustrate the concept of relaxation using two examples – one for each of the two operators, details can be looked up in [14].



(a) Relaxed Skyline (b) Relaxed Top-N

Figure 6: Relaxation for Rank-Aware Query Opererators

Figure 6(a) shows the relaxed version of the skyline shown in Figure 4(a), Figure 6(b) the relaxed version of the top-N query of Figure 4(b). Now, some of the result records (solid black) represent not only themselves but also some other records nearby – all those represented records are located within the gray shaded area.

Given a set D of data objects, a top-N or skyline query $\mathfrak{T}$, a distance function $d : D \times D \rightarrow \mathbb{R}$, and a user-defined maximum relaxation $\varepsilon \in \mathbb{R}$, any subset R of D for that

$$\forall t \in \mathfrak{T}(D) \; \exists r \in R : \; d(t, r) \leq \varepsilon \qquad (1)$$

holds, is called a *relaxed top-N/skyline result*. Furthermore, if for $r,\ t \in D$: $d(r, t) \leq \varepsilon$ holds, r is called a *representing record* of t. A *representative* is the combination of such a representing record and the region that is represented. Thus, a relaxed top-N/skyline result can be defined as a set R that contains a representative for each result record $t \in \mathfrak{T}(D)$. There are usually many R for a data set D that fulfill the above equation. Furthermore, several records can be represented by one single representative.

The guarantee that is output to the user for such a result is an inherent part of any representative. It guarantees that all records that are represented by the representative are located within the region that is part of its definition. The maximum distance between the representing record and any point in the region never exceeds ε with respect to distance function d.

With respect to distributed query processing step 3 is affected. In addition to the techniques for data-level pruning a peer tries to find representatives for the data provided by its neighbors – using the information provided by the routing indexes [14]. If this is possible additional neighbors can be pruned. This kind of relaxation is especially useful when we have to deal with data clusters. Thus, many records can be represented by only a few representatives but the user still gets an overview of the data. However, the concept of relaxation reduces costs at the expense of accuracy. Thus, the higher the user-defined relaxation ε the lower are the costs but the higher is the loss of accuracy. In general, that loss can be neglected and the user can work with the representatives. If not, the user is free to define an ε of 0, which means that no relaxation is applied.

5. QUERY REWRITING

In this section we sketch the algorithm that we developed to rewrite queries [22]. Before going into detail let us first review some existing algorithms for LAV mappings.

5.1 Query Rewriting using Views

There are three main rule-based algorithms for LAV mappings proposed in literature dealing with conjunctive queries: the Bucket Algorithm [6], Inverse-Rules [5, 6] and MiniCon [6, 21]. The former considers the subgoals of a query (the global relations and attributes named in the body and the head of the query), builds a bucket for each of them, determines which view may be relevant to a query subgoal, and puts it into the corresponding bucket. To reformulate the whole query, a Cartesian product of the views in the buckets is built. This Cartesian product may be rather large and may as well contain a lot of combinations that would lead to an empty answer, e.g., because of missing attributes for joins or conflicting predicates. In order to find rewritings the algorithm performs a query containment test for each candidate, which is the main disadvantage of the Bucket Algorithm with regard to performance.

Inverse-Rules basically inverts the information given by the view definitions to build rules describing how to get tuples out of the data sources into the global relations to answer a query. Unfortunately, data joined locally is separated into the global relations as well. If there is the same join condition in both query and view, the tuples have to be recomputed and joined again. To avoid this and to obtain a more efficient rewriting, the inversed rules have to be unfolded.

The third algorithm, MiniCon, starts like the Bucket Algorithm and considers the subgoals of a query. For each known view definition it determines which query subgoals are answered. After finding a partial mapping from the query to the view, the join predicates of the query are considered in order to find out, which additional set of view subgoals is needed for rewriting the whole query. This set and the mapping information is called a MiniCon Description (MCD). In a second phase the MCDs are combined and query rewritings are built. Compared to the Cartesian product of the buckets in the Bucket Algorithm fewer combinations of MCDs have to be considered. A portion of the work done in the second phase of the Bucket Algorithm has been shifted into the first phase of building the MCDs in MiniCon. Thus, views that cannot be combined because of missing attributes for join conditions are left out at an early stage. The Bucket Algorithm on the contrary needs to find every combination such a view is involved in.

The presented algorithms are rule-based and work well with respect to relational data and conjunctive queries. For Semantic Web applications and data exchange in a wide-spread network of heterogeneous data sources we believe that XML data rather than relational data is the better basis to choose. Additionally, indexes and high-level operators should be integrated to offer an overview over the data without flooding the network and transforming/transmitting all the data that is available. If we did not consider operators like top-N and skyline in the rewriting process, peers that receive the query could not apply the operators to their local data. Consequently, those peers would have to transform and transmit all their local data such that the operator can be computed at the initiator. Obviously, this is not desirable and causes unnecessarily high network traffic and computational load.

As we are considering PDMS there should be no global component. Instead, each peer should work on its own local schema that serves as the mediated schema for each query received at the peer. Schema mappings are defined as views between a peer and its neighbors. Within the Piazza project [7] directed views are likewise defined between pairs or a small set of peers. Piazza uses these views in both directions, thus integrating data of heterogeneous data sources in a LAV- and a GAV-like manner. By building a rule-goal-tree views are expanded for a peer relation or MCDs are built for rewriting the query. A global system catalog is assumed to provide access to all of the mappings needed for the expansion of a rule-goal tree node.

SmurfPDMS on the contrary has no global component or global knowledge at all. The behavior of the system results from the local interactions of peers. Each peer rewrites queries using the views to its neighbors only. If a query is received at a peer it is processed locally, rewritten and parts of the rewritten query are sent to neighbors. Queries and answers are chained through the network.

5.2 Query Rewriting using Subgoal Trees

The algorithm for rewriting queries we propose in this paper basically combines the Bucket Algorithm [6] with the advantages of Minicon [6, 21]. It works on operator trees instead of datalog rules and pays attention to all query operators supported by SmurfPDMS including rank-aware operators.

Before going into detail we need to discuss how to create subgoals from queries and views and how these subgoals are represented. The subgoal representation is the basis for our modified bucket algorithm. As SmurfPDMS works on (semi-structured) XML data rewriting has to deal with nesting and unnesting of elements or structures. As a consequence, we represent subgoals using a tree structure. Subgoal trees for views always consist of:

- one root node with: (i) the path to the context node of the view's owner (source), and (ii) the path to the context node of the target peer,
- multiple inner nodes with: (i) the name of an element of the owner's schema or (ii) the name of an element of the target peer's schema,
- leaf nodes with: (i) the name of a text node of the owner's schema, (ii) the name of a text node of the target peer's schema or a constant that corresponds to the target peer's node, and (iii) optional predicates.

This structure helps us find a symbol mapping. The root node of a subgoal tree contains the name or path of the XML element, which contains additional elements and text nodes. This context element is comparable to literals or relation names in the relational case. Inner subgoal tree nodes show the nesting of elements at the owner or at the target. The names of text nodes (exported symbols) and predicates or constants are always contained in leaf nodes. As an example, consider the view definition of Figure 3 that we have used in Section 3.1. The corresponding subgoal trees are illustrated in Figure 7(a). In order to compare a view to a query the query has to be transformed into a subgoal tree representation as well. It is basically constructed the same – the main difference is that in case of queries there are no target elements that need to be considered. Figures 7(b) and 7(c) show examples of queries and their subgoal trees.

A view subgoal may be useful for answering a query subgoal, if a mapping can be found

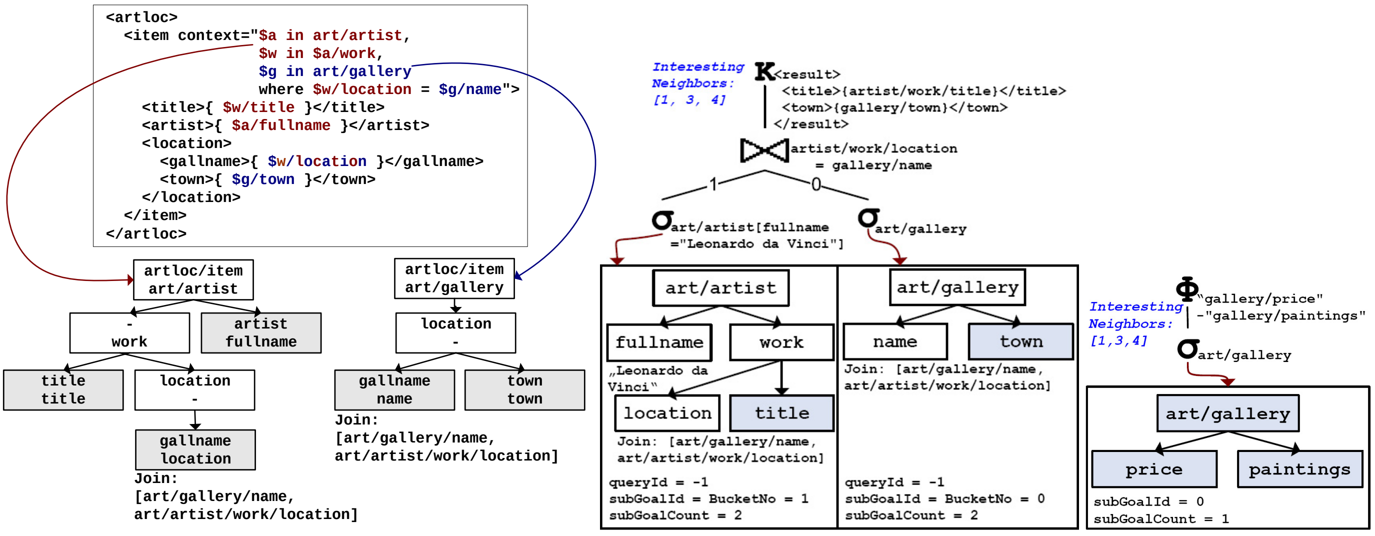
- for the query subgoal's context node path,
- for each element name or path that occurs in the query subgoal,
- especially for each exported symbol contained in the query subgoal, i.e., for each text node expected as result, and
- for each predicate of the query subgoal.

Checking all these conditions, we can decide whether a view subgoal fulfills a query subgoal and finally prune views and thus peers on schema-level. By combining view subgoals the query can be answered. In principle, the rewriting algorithm that we propose in this paper works in 9 steps:

1. receiving a query plan consisting of POPs
2. preprocessing
3. creating buckets
4. sorting view subgoals into buckets
5. creating combinations of buckets
6. creating query snippets
7. optimizing query snippets
8. optimizing subgoal combinations and creating remote queries
9. assembling the rewritten query plan

We explain these steps using two running examples: a simple select-project-join query and a query that contains a rank-aware query operator. The former consists of two *select* POPs at leaf level – both having the same *join* POP as parent. The root note (as parent of the *join* POP) is a *construct* POP. The query plan together with the corresponding subgoal trees is shown in Figure 7(b). It asks for works of Leonardo da Vinci and their location. The second query consists of only two levels. A *select* POP as leaf node and a *skyline* POP as root node. The POP tree as well as the corresponding subgoal tree are shown in Figure 7(c). This is the representation of the query that we introduced in Section 3.2 as an example for formulating queries in text format. It asks for a set of galleries that offer many but cheap paintings. Both queries are issued at P_0 (Figure 2) and hence are formulated in P_0's local schema.

1. Receiving a Query Plan. The input query plan – corresponding to the original query – has already been processed locally and may contain intermediate local results (Section 4.1). The root node of the POP tree is assigned a set of neighbors that are to be considered for rewriting. This set is likely to contain a lot fewer neighbors than the peer is connected to because of data-level pruning. As rewriting costs depend on the number of used views, this information significantly improves performance. In our example, P_0 receives



```
<artloc>
  <item context="$a in art/artist,
                 $w in $a/work,
                 $g in art/gallery
                 where $w/location = $g/name">
    <title>{ $w/title }</title>
    <artist>{ $a/fullname }</artist>
    <location>
      <gallname>{ $w/location }</gallname>
      <town>{ $g/town }</town>
    </location>
  </item>
</artloc>
```

artloc/item art/artist
- work
artist fullname
title title
location -
gallname location
Join: [art/gallery/name, art/artist/work/location]
artloc/item art/gallery
location -
gallname name
town town
Join: [art/gallery/name, art/artist/work/location]
Interesting Neighbors: [1, 3, 4]
K <result>
<title>{artist/work/title}</title>
<town>{gallery/town}</town>
</result>
artist/work/location = gallery/name
1
0
σ art/artist[fullname ="Leonardo da Vinci"]
σ art/gallery
art/artist
fullname
work
„Leonardo da Vinci"
location
title
Join: [art/gallery/name, art/artist/work/location]
queryId = -1
subGoalId = BucketNo = 1
subGoalCount = 2
art/gallery
name
town
Join: [art/gallery/name, art/artist/work/location]
queryId = -1
subGoalId = BucketNo = 0
subGoalCount = 2
Interesting Neighbors: [1,3,4]
Φ "gallery/price" -"gallery/paintings"
σ art/gallery
art/gallery
price
paintings
subGoalId = 0
subGoalCount = 1

(a) View from P_0 to P_4 (b) Join Query (c) Skyline Query

Figure 7: Building Subgoal Trees for Views and Queries

the two example queries and identifies neighbors P_1, P_3, and P_4 to be relevant to both queries.

2. Preprocessing. Since *union* POPs are not part of conjunctive queries we need a preprocessing step if such operators are contained in a query. Given a query with a *union* POP, the two subqueries underneath are given identifiers. They are rewritten and processed independently from each other. After the peer has received the results of both subqueries, the result of the input query is the union of the results of the two subqueries (effectuated by a *union* POP as parent of the rewritten subqueries).

Rank-aware operators are not part of conjunctive queries either. To consider such operators for rewriting we exploit their additivity: $\phi(D_1, \ldots, D_n) = \phi(\phi(D_1), \ldots, \phi(D_n))$, where ϕ denotes a rank-aware operator and $D_1, \ldots, D_n$ the data sets of peers $P_1, \ldots, P_n$. Thus, to reduce network load and computational load at the initiator, neighbor peers receive a rewritten query that includes ϕ. When the initiator has received the results from its neighbors, it once more evaluates ϕ over the union of its local result and the results received from its neighbors. The preprocessing step takes care that ϕ is evaluated over that union.

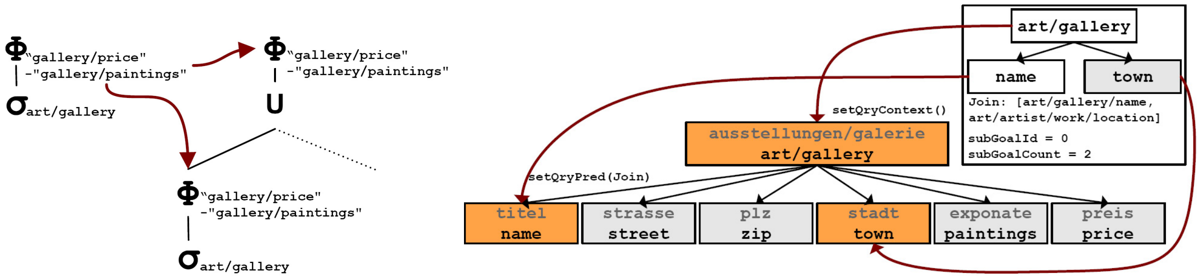
The corresponding query POP tree for our second example query is shown in Figure 8(a) where the original skyline POP of the query is cloned and used as root node of the rewritten query. The *union* POP underneath describes that the result of the local query (left child), i.e., the original input query, is merged with the results of the neighbors (right child, dotted line). As this is only the preprocessing step the right child of the *union* POP is not yet known but computed in the following steps of this algorithm.

3. Creating Buckets. At this point we need the subgoal representation of queries. For each *select* POP at leaf level one subgoal and a corresponding bucket is created (Figures 7(b) and 7(c)). Each bucket is assigned a subgoal ID (`subGoalId`). Furthermore, it knows about the total number of subgoals that have been created for the query (`subGoalCount`). This additional information allows for comparing queries by means of their identification instead of comparing their operator trees.

4. Sorting View Subgoals into Buckets. For each peer contained in the list of interesting neighbors the corresponding view subgoals are compared to the query subgoals of the buckets. If a view subgoal is able to answer a query subgoal, i.e., it meets the mapping conditions stated above, it is sorted into the corresponding bucket. Starting at the context nodes the source names and the predicates are compared. A query predicate that cannot be fulfilled in the view subgoal would yield an empty answer for the rewritten query. If source paths or names are different or missing, no mapping can be found. Thus, such view subgoals can be dropped. Especially, join predicates of a query have to be checked against the view subgoals. Since for rewriting the input query the view subgoals have to be combined, the view has to contain either the same join predicate or an exported symbol, i.e., a corresponding text node, so that the join may be computed with another view. As in MiniCon [21] we look at the predicates at this early stage of the rewriting process to avoid a large-scale containment test for rewritings of views that obviously do not yield any results.

In a subgoal tree that is inserted into a bucket all its matching nodes are tagged. Figure 8(b) shows the result for the query subgoal of bucket 0 (join query) and a view subgoal of P_3. Additionally, each view is assigned a list of bucket IDs that its subgoals have been sorted into.

5. Creating Combinations of Buckets. The buckets and the view subgoal trees sorted into them only represent parts of the query. The view subgoals within the buckets have to be combined to answer the whole query. This is done first by building the Cartesian product between the buckets' view subgoal trees. This Cartesian product may be rather large and may as well contain a lot of redundant combinations. To remove them, the bucket IDs that have been assigned to the views are consulted. All combinations that do not fulfill the following requirement are pruned: $\forall x \neq y \wedge x \, is \, combined \, with \, y : Colors(x) \cap Colors(y) = \emptyset$, where $Colors(x)$ denotes the set of query subgoals that view x fulfills. In case a view fulfills multiple query subgoals, it has been given multiple bucket IDs. By checking the requirement above, the minimal set of additional views is found for answering the whole query. All other combinations can be pruned. This step is similarly done in the MiniCon algorithm [21]. For the remaining combinations the algorithm checks whether the predicates of combined subgoals originating from different views are contradictory. Since they would yield empty result sets such combinations are pruned, too.



(a) (b)

Figure 8: (a) Preprocessing of a Skyline Query, (b) Sorting a View Subgoal into a Bucket and Tagging Matching Elements

For our join query two buckets have been created. View subgoals of P_3 and P_4 have been sorted into bucket 0, view subgoals of P_1 and P_4 into bucket 1. The Cartesian product results in four combinations, but as the view of P_4 is able to answer the whole query, all combinations of P_4's view and other views can be dropped. At the end only the combinations (3,1) and (4,4) remain. Note that the buckets contain view subgoals not views. Thus, the combination (4,4) says that two subgoals that are both supported by P_4 need to be combined.

6. Creating Query Snippets. By now, it is checked whether a view subgoal is useful for rewriting the query (step 4) and whether a combination would possibly yield new and non-empty result sets (step 5). In this step, the tagged view subgoal tree nodes are rewritten to answer the corresponding query subgoal, i.e., a mapping of the query symbols has to be found. Thus, for each view subgoal tree in a combination we create a *query snippet*. It is a linear POP tree and has the following basic structure: a *remote query* POP as root, a *construct* POP as child of the *remote query* POP, and a *select* POP as child of the *construct* POP.

The snippet POP tree is created in a bottom-up fashion. Thus, the creation starts with the *select* POP. Starting at the root node of the view subgoal tree we traverse the tree downwards as long as the nodes have only one tagged child node. We stop at that node that is the first one to have more than just one tagged child node. The path of the node we stopped at determines the expression of the *select* POP, which may be extended by additional query predicates.

Based on that node in the view subgoal tree that corresponds to the query context node the expression for the *construct* POP is built. As the view subgoal tree contains the schema elements of both source and target, the construct expression is built using the source XML structure as basis and the insertion of target element paths. Finally, the *remote query* POP is created providing information about the receiver of the query, the subquery and subgoal ID, and the number of subgoals.

Based on the tagged view subgoal of bucket 0 (join query) in Figure 8(b), Figure 9 shows the corresponding query snippet.

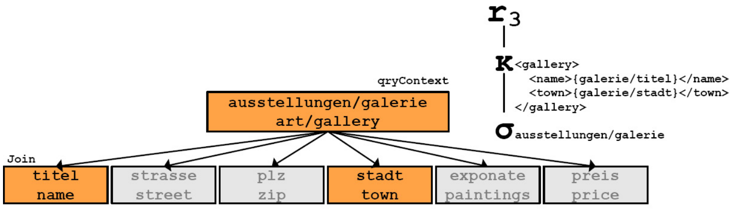


Figure 9: Creating Query Snippets from Tagged Subgoal Trees

7. Optimizing Query Snippets. Query snippets up to now are remote queries that contain selections, projections, nesting, unnesting, and renaming of elements. In principle, it is already possible to use them to assemble the final query plan as a query snippet represents a *select* POP on leaf-level of the original query. However, such a strategy would result in a transformation of all data sets provided by the target peer. This data would be transformed and sent through the network. Although the result would be correct, performance regarding computational load at the initiator would decrease and network traffic would increase using this strategy. Thus, in this step query snippets are optimized by considering additional POPs of the original query. The expression of the *construct* POP provides all information for rewriting additional POPs and for pushing them down underneath the *construct* POP. Thus, computational load is shared among peers and smaller result sets reduce network load.

This is especially useful when a query contains *skyline* or *topn* POPs. In our example skyline query the *skyline* POP can be pushed down under the remote query POP such that the receiving peer can already apply the skyline operator to its local data und thus reduce the size of the result set. The result is shown in Figure 10 illustrating how the *construct* POP's expression is used to rewrite the ranking functions of the *skyline* POP.

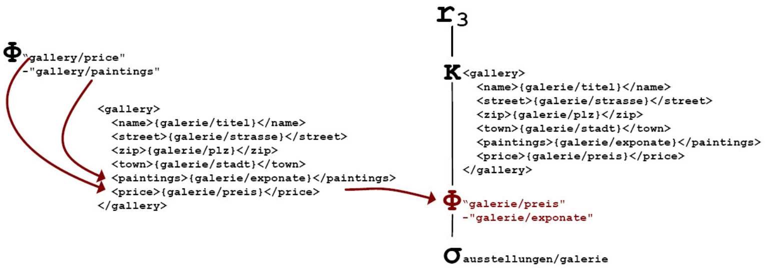

Φ "gallery/price" -"gallery/paintings"
<gallery>
<name>{galerie/titel}</name>
<street>{galerie/strasse}</street>
<zip>{galerie/plz}</zip>
<town>{galerie/stadt}</town>
<paintings>{galerie/exponate}</paintings>
<price>{galerie/preis}</price>
</gallery>
r3
K <gallery>
<name>{galerie/titel}</name>
<street>{galerie/strasse}</street>
<zip>{galerie/plz}</zip>
<town>{galerie/stadt}</town>
<paintings>{galerie/exponate}</paintings>
<price>{galerie/preis}</price>
</gallery>
Φ "galerie/preis" -"galerie/exponate"
σ ausstellungen/galerie

Figure 10: Optimizing Query Snippets

8. Optimizing Subgoal Combinations and Creating Remote Queries. According to the combinations that we received as result from step 5 query snippets are combined to subqueries using the original query as template. Take the POP tree corresponding to our join query as example (Figure 7(b)). In step 5 we have identified two view subgoal combinations (3,1) and (4,4) that need to be considered. Thus, a subquery is built by replacing the *select* POPs at leaf level with the corresponding query snippets. Thus, we retrieve two subqueries. The one for the combination (4,4) is shown in the upper part of Figure 11. Each of these two subqueries contains two remote queries.

It is possible under certain circumstances to combine those remote queries (that would otherwise result in two independent queries) into just one remote query. Whenever two query snippets of the same neighbor are "connected" via joins in a subquery (the connection might involve multiple joins), we reorganize the tree such that the *remote query* POP can be pushed up over the *join* POP. If the view definition additionally contains the same join as the query, then the data is already stored in the joined format at the neighbor. Thus, we can remove the join and merge the snippets into

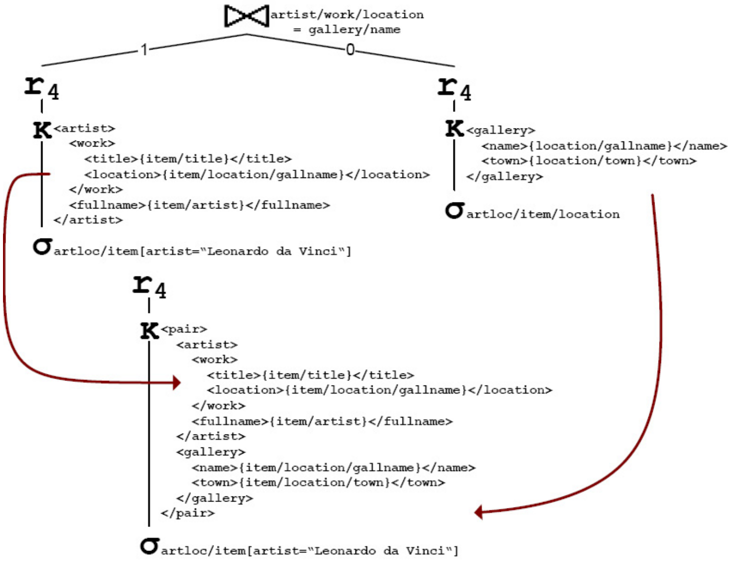

⋈ artist/work/location = gallery/name
1
0
r4
K <artist>
<work>
<title>{item/title}</title>
<location>{item/location/gallname}</location>
</work>
<fullname>{item/artist}</fullname>
</artist>
σ artloc/item[artist="Leonardo da Vinci"]
r4
K <gallery>
<name>{location/gallname}</name>
<town>{location/town}</town>
</gallery>
σ artloc/item/location
r4
K <pair>
<artist>
<work>
<title>{item/title}</title>
<location>{item/location/gallname}</location>
</work>
<fullname>{item/artist}</fullname>
</artist>
<gallery>
<name>{item/location/gallname}</name>
<town>{item/location/town}</town>
</gallery>
</pair>
σ artloc/item[artist="Leonardo da Vinci"]

Figure 11: Optimizing Combinations of Subgoals and Query Snippets

just one. This is illustrated in the lower part of Figure 11 for our join query example and the combination (4,4).

9. Assembling the Rewritten Query Plan. In the last three steps remote queries have been created and optimized. In this step the query plan as the result of the rewriting process is built by combining the original query and all subqueries by means of *union* POPs. Figures 12(a) and 12(b) show the resulting final query plans for our two example queries.

6. EVALUATION

This section discusses our evaluation results that we obtained by integrating the presented techniques into SmurfPDMS. As the application of routing indexes is optional, we did not apply data-level pruning in our tests and focused on schema-level pruning.

6.1 Rewriting vs. Schema Indexing

Our first test results illustrate the benefit of our techniques in comparison to a network that does not use query rewriting. For this purpose, we used the network of 7 Peers shown in Figure 2 – providing 42 data records. The existence of a directed mapping between any two peers is illustrated with an arrow. If there is an arrow from P_a to P_b, then P_a holds a mapping to P_b.

It is obvious that in such a scenario an approach that does not consider any rewriting at all would never be able to find all result records. The reasons are heterogeneity and the impossibility to rewrite queries. Thus, the query would have to be forwarded in its orignal form referring to the initiator's schema. This would only work in a scenario where the peers' local data originates from a vertical and horizontal partitioning of an original data set. A second problem with such a strategy is that it would have to flood the network since no routing information is available.

In comparison to our techniques such a strategy would of course be inferior. Thus, we chose to make the comparison more fair. First, we adapted the network of Figure 2 such that XML nodes representing the same data are named the same in all the peers' local schemas. Second, we decided to give the strategy a little help on doing some schema-level pruning: each peer stores a kind of routing index that indexes schema elements. This means that a peer has information about the existence of schema elements that can be accessed by forwarding a query to a specific neighbor.

Thus, we have two scenarios: (1) the techniques presented in this paper on the network of Figure 2 with mappings and (2) the non-rewriting approach on the altered network of Figure 2 with schema indexes. In total, we issued the 6 queries listed in Table 2 with peer P_0 as initiator. To give both test scenarios the same chances the expressions contained in the operators only refer to XML nodes and paths that are common for peers in both scenarios.

QueryID	Query Type	#Levels in POP tree
0	Projection	1
1	Projection and Transformation	2
2	Projection and Selection	2
3	Join with Transformation	3
4	Top-N with Transformation	3
5	Skyline with Transformation	3

Table 2: Query Mix

The most important cost factor in distributed environments are network load (data volume) and the number of messages that are necessary to answer a query. Thus, we discuss our evaluation results using these two measures and neglect local execution costs.

Let us first consider the non-incremental variant (QS) of our distributed query processing strategy described in Section 4. With respect to network load in terms of transferred data volume the rewriting strategy should produce less load since in contrast to the non-rewriting strategy only remote queries are sent to neighboring peers instead of the whole query. With respect to the number of messages we expect more or less the same result since both strategies use additional information about the schemas of neighboring peers.

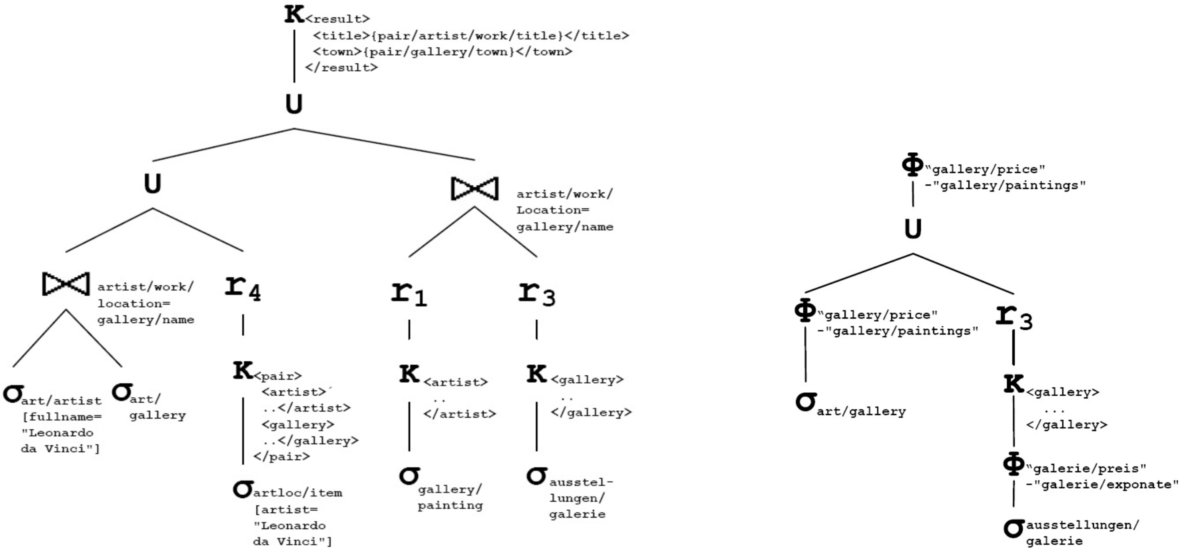
The results of our tests are listed in Table 3 – summarizing data volume and the number of messages for the whole query load. As we have expected data volume is reduced using our rewriting techniques. Furthermore, also the number of messages is reduced. This is due to the fact that the rewriting strategy is able to prune more neighbors since only those neighbors are queried that provide all elements named within the query.

	Rewriting	No Rewriting	Difference
Total Number of Messages	50	68	$-26,5\%$
Data Volume in kByte	133.053	241.568	$-44,9\%$

Table 3: Results for QS

We did the same tests using the incremental variant of our query processing strategy (IMS, Section 4). Using IMS peers do not have an obligation to answer a received query – using QS peers always send answers to signal the sender that the query has been processed and that the sender can stop waiting for answers. Thus, the difference between using rewriting or not should be less obvious than in the tests for QS. Still, in the non-rewriting strategy more queries should be forwarded to neighboring peers due to the reasons stated above. But without the obligation to answer each received query the impact of asking some additional (non-necessary) peers should be smaller than in the QS tests. For the data volume we expect the same tendencies as for QS.

The test results shown in Table 4 support our anticipation. The difference between the rewriting and the non-rewriting strategy is smaller than in Table 3. Furthermore, in comparison to QS, IMS needs fewer messages to answer the query since there is no obligation to answer queries. In addition to that, IMS delivers first


K<result> <title>{pair/artist/work/title}</title> <town>{pair/gallery/town}</town> </result>
U
U
artist/work/ Location= gallery/name
artist/work/ location= gallery/name
r4
r1
r3
σart/artist [fullname= "Leonardo da Vinci"]
σart/ gallery
K<pair> <artist> ..</artist> <gallery> ..</gallery> </pair>
σartloc/item [artist= "Leonardo da Vinci"]
K<artist> .. </artist>
σgallery/ painting
K<gallery> .. </gallery>
σausstel- lungen/ galerie
Φ"gallery/price" -"gallery/paintings"
U
Φ"gallery/price" -"gallery/paintings"
r3
σart/gallery
K<gallery> ... </gallery>
Φ"galerie/preis" -"galerie/exponate"
σausstellungen/ galerie

(a) Join Query

(b) Skyline Query

Figure 12: Rewritten Queries

results almost instantaneously to the user and incrementally adds more records to the result.

	Rewriting	No Rewriting	Difference
Total Number of Messages	49	54	$-9,2\%$
Data Volume in kByte	135.581	205.407	$-34,0\%$

Table 4: Results for IMS

Our first results show that the rewriting method is preferable over a non-rewriting strategy since the knowledge contained in the mappings allows for a more effective schema-level pruning. Furthermore, the additional load that a peer has to deal with when rewriting a query is negligible in comparison to the benefits. Another positive effect of using mappings is that a peer is able to query schemas different from its own one. In comparison to that the schema index approach can only query data originating from a partitioning of a global database.

6.2 The Influence of the Choice of Neighbors

Query Processing in PDMS depends on the choice of neighbors, i.e., it depends on what neighbors a peer has established mappings to. In order to show that our techniques still work in a scenario where a query has to be rewritten more often than just once, we created two networks each consisting of 20 peers. The data of these peers can be divided into four topics: galleries, artists, art objects (paintings or sculptures) and styles.

In scenario 1 the network structure corresponds to a star with P_0 having connections to all other peers. Thus, scenario 1 represents a standard data integration system where P_0 serves as mediator and provides a common schema using LAV mappings. In scenario 2 we arranged the peers in clusters determined by a similarity measure. As measure for schema similarity we used the number of possible schema element correspondences between the schemas of any two peers. For each link we defined mappings in both directions, i.e., if P_0 has a mapping to P_1 then P_1 also has a mapping to P_0. While this increases reachability of data it also increases the number of cycles that have to be dealt with. Peers with similar data (data of the same topic) are clustered in our network, i.e., there are many links between such peers. Since some of the peers store data of more than just one topic, clusters overlap. In order to answer queries correctly that perform joins between the topics we established additional links that interconnect clusters.

In both scenarios we issued 13 queries with at most 4 subgoals at P_0 applying QS. Two of the queries included top-N or skyline operators. Due to the favorable choice of neighbors, results in both scenarios should be the same although queries need to be rewritten more than just once in scenario 2. In addition to this we expect duplicates in the result set of scenario 2 since cycles are only detected if the same rewritten query has been received at the same peer in the same way twice. If for example a peer receives remote queries from different peers the peer does not always realize that both query the same data for the same original query.

Scenario	Result Size	Result Size after Removing Duplicates
(1) Data Integration	304	294
(2) PDMS	353	294

Table 5: Comparison of a Data Integration System (1) and Distributed Query Processing in PDMS (2)

Table 5 shows our results with respect to the two scenarios summarizing the results for all test runs and queries. In both scenarios all result tuples have been retrieved. Since some data records are stored at multiple peers both scenarios retrieve a small number of duplicates. As anticipated the second scenario retrieves more duplicates due to the reasons indicated above.

As our tests have shown when optimal neighbors are chosen with respect to (i) the similarity of schemas and (ii) the combinability of neighbor data for join operations, all the data in the system can be accessed. Performance can still be improved by elaborating cycle detection and influencing the network topology.

6.3 Benefits of Considering Rank-Aware Operators for Rewriting

In our final tests we wanted to examine the benefits of considering high-level operators for rewriting. Since we have designed our algorithms for the purpose of reducing costs by considering such operators we expect the costs to be lower in comparison to a rewriting strategy that does not consider such options. For comparison we used the implementation of our rewriting strategy in SmurfPDMS and bypassed all optimizations that consider rank-aware operators.

For our tests we used the PDMS network of 20 peers that we have used in scenario 2 in Section 6.2. We chose a 3-level top-N query (*topn* POP, *construct* POP and *select* POP) with $N = 5$, issued it twice at the same peer – once using QS and once using IMS. In our tests we also varied the number of records whose structure fits the query: 20 and 300 data records distributed among all peers. The main benefit of considering rank-aware operators is to reduce local computation load and network traffic. Thus, the data volume should be reduced in comparison to the strategy that does not consider rank-aware operators. This tendency should be more obvious when there is a higher number of relevant records to the query.

The results of our tests are shown in Figure 13. As we have expected, in general message volume is reduced when considering rank-aware operators for rewriting. These results also show that for low numbers of relevant data the savings are smaller than for high numbers.

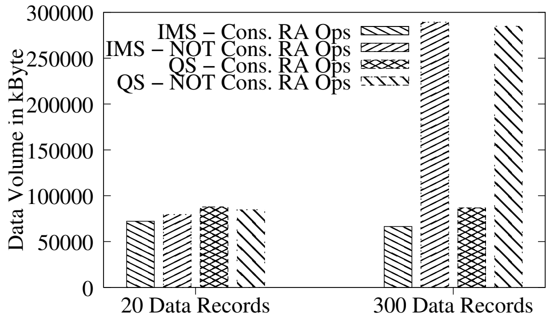


Figure 13: Benefits of Considering Rank-Aware Operators for Rewriting, Data Volume

7. CONCLUSION

Although query rewriting is an important component of PDMS, interoperability is more than just rewriting queries. For efficiency and practicability reasons such systems require distributed query processing strategies that try to minimize execution costs. Part and parcel of such strategies is pruning neighbors from consideration using information on both schema-level (mappings) and data-level (distributed data summaries respectively routing indexes). Further reduction of costs can be achieved by considering rank-aware query operators and relaxation.

In this paper, we have elaborated on semantic interoperability in distributed environments such as PDMS. We provided a holistic view on all aspects that are important for query processing in such systems and emphasized their interrelationships and dependencies. For this purpose, we used SmurfPDMS as an example and discussed techniques that we developed and conclusions that we came to working on that project. However, as we have remarked throughout the paper, there are still many open issues that remain to be solved and/or improved in future work.

8. REFERENCES

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: building histograms without looking at data. *SIGMOD Rec.*, 28(2):181–192, 1999.

[2] A. Bonifati, E. Chang, A. Lakshmanan, T. Ho, and R. Pottinger. HePToX: Marrying XML and heterogeneity in your P2P databases. In *VLDB '05*, pages 1267–1270, 2005.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE'01*, pages 421–432, 2001.

[4] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS '02*, pages 23–32, July 2002.

[5] O. Duschka and M. Genesereth. Query planning in infomaster. In *SAC '97*, pages 109–111, 1997.

[6] A. Halevy. Answering Queries using Views: A Survey. *The VLDB Journal*, 10(4):270–294, 2001.

[7] A. Halevy, Z. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza Peer Data Management System. *TKDE*, 16(7):787–798, 2004.

[8] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov. Schema Mediation in Peer Data Management Systems. In *ICDE 2003*, pages 505–, 2003.

[9] K. Hose, A. Job, M. Karnstedt, and K. Sattler. An Extensible, Distributed Simulation Environment for Peer Data Management Systems. In *EDBT'06*, pages 1198–1202, 2006.

[10] K. Hose, M. Karnstedt, A. Koch, K. Sattler, and D. Zinn. Processing Rank-Aware Queries in P2P Systems. In *DBISP2P 2005*, pages 238–249, 2005.

[11] K. Hose, M. Karnstedt, K. Sattler, and D. Zinn. Processing Top-N Queries in P2P-based Web Integration Systems with Probabilistic Guarantees. In *WebDB '05*, pages 109–114, 2005.

[12] K. Hose, D. Klan, and K. Sattler. Distributed Data Summaries for Approximate Query Processing in PDMS. In *IDEAS '06*, pages 37–44, 2006.

[13] K. Hose, C. Lemke, J. Quasebarth, and K. Sattler. SmurfPDMS: A Platform for Query Processing in Large-Scale PDMS. In *BTW '07*, pages 621–624, 2007.

[14] K. Hose, C. Lemke, and K. Sattler. Processing Relaxed Skylines in PDMS Using Distributed Data Summaries. In *CIKM '06*, pages 425–434, 2006.

[15] M. Karnstedt, K. Hose, E. Stehr, and K. Sattler. Adaptive Routing Filters for Robust Query Processing in Schema-Based P2P Systems. In *IDEAS '05*, pages 223–228, 2005.

[16] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping data in peer-to-peer systems: semantics and algorithmic issues. In *SIGMOD '03*, pages 325–336, 2003.

[17] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[18] M. Lenzerini. Data integration: a theoretical perspective. In *PODS '02*, pages 233–246, 2002.

[19] A. Y. Levy. Obtaining Complete Answers from Incomplete Databases. In *The VLDB Journal*, pages 402–412, 1996.

[20] Y. Petrakis, G. Koloniari, and E. Pitoura. On Using Histograms as Routing Indexes in Peer-to-Peer Systems. In *DBISP2P*, pages 16–30, 2004.

[21] R. Pottinger and A. Halevy. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2-3):182–198, 2001.

[22] J. Quasebarth. Distributed Query Rewriting in PDMS (in German). Master's thesis, TU Ilmenau, 2007.

[23] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[24] P. Rodríguez-Gianolli, A. Kementsietsidis, M. Garzetti, I. Kiringa, L. Jiang, M. Masud, R. J. Miller, and J. Mylopoulos. Data sharing in the Hyperion peer database system. In *VLDB '05*, pages 1291–1294, 2005.

[25] A. Roth and F. Naumann. System P: Query Answering in PDMS under Limited Resources. In *IIWeb*, 2006.

[26] J. D. Ullman. Information integration using logical views. In *ICDT '97*, pages 19–40, 1997.