OptimAX: optimizing distributed continuous queries

Serge Abiteboul, Ioana Manolescu, Spyros Zoupanos

OptimAX: optimizing distributed continuous queries

S. Abiteboul I. Manolescu
S. Zoupanos*
INRIA Futurs, Gemo group & LRI, Université Paris Sud, 4, rue J. Monod, 91893 Orsay Cedex
firstname.lastname@inria.fr

1 Setting

Fulfilling the vision of a decentralized Web of peers requires efficient mechanisms for decentralized dissemination of information. RSS feeds are part of this vision: incremental updates to XML documents are pushed from a given producer to a set of subscribers along known paths.

In this work, we envision processing *continuous XML queries*. Such queries are expressed in some XML query language on *XML data streams* (as opposed to XML queries on static XML data sources). Continuous queries are evaluated incrementally as soon as there are new data items in any of their input streams. For instance, the query "get all the cinemas and starting hour for the movie *Shinobi* in *Paris*", evaluated as a continuous query on a stream of cinema information, should return some results as soon as the movie becomes available in some Paris cinema.

Our work is inherently placed in a distributed setting. First, the data sources are often remote from the users. Second, a single user may be interested in distinct remote data sources. Finally, we consider that remote sites may help processing queries asked by a user, even if those sites are not data providers, and even if the user ignores their presence. Our interest is on leveraging distributed query processing abilities in order to obtain better performance.

The system we propose to present, OptiMAX, applies the principles of distributed query optimization [7] to the problem of distributed evaluation of continuous XML queries. OptiMAX is an *optim*izer for *A*ctive *X*ML documents (AXML in short). It is implemented as a module which can be used next to an AXML peer, and it may be invoked whenever users ask queries on their AXML documents. The optimizer draws up an initial query plan, and then attempts to rewrtite it using a combination of heuristics and cost information in order to improve the plan's performance estimates. An interesting feature is that all plans are AXML documents themselves. When the optimizer has retained a plan, it hands it to the AXML peer, which evaluates it directly following the decisions taken by the optimizer.

The rest of this document is organized as follows. Section 2 briefly presents the (A)XML data model and the main elements of our plan algebra. Section 3 describes the demonstration scenario. We briefly relate OptiMAX to similar projects in Section 4 and we conclude.

2 AXML and its algebra

In this section, we review the basic concepts framing our OptimAX work: ActiveXML doc-

*This work is part of the WebContent RNTL project.

uments and their associated evaluation algebra.

2.1 ActiveXML documents and continuous queries

ActiveXML documents are XML documents including some special elements labeled sc (for *s*ervice *c*all). We now describe the basic features of the language as they have been laid out in [6]. Elements labeled sc describe a given Web service that should be called, and may include XML parameters of the call. When the call is *activated*, a request message (including the parameters) is sent to the Web service, and when the results are received, they are inserted in the AXML document as siblings of the sc node.

While the basic mode of interaction with a Web service consists of a single request and a single response message, as part of an ongoing thesis in the Gemo group, *stream services* are being developed. Once a call to a stream service has been activated, a stream of XML answers will be returned asynchronously, and they are all inserted in the caller AXML document as siblings of the sc node. After all answers have been returned, a special token "end-of-stream" (or *eos*) is returned. Observe that a "regular" service (returning just 1 answer) is a special case of continuous service returning a short stream. Thus, from now on, without loss of generality, we will refer to continuous services only.

We are mainly concerned with *declarative services*, defined by means of XQuery [8] queries. A few optimization techniques are also possible for non-declarative services, however, these are more limited. We consider a distributed setting, where different *peers* provide different services and/or host AXML documents. We consider a set of distinct *peer identifiers* of the form $p_1, p_2, \ldots$, and a set of distinct *document identifiers* inside each peer, of the form $d_1, d_2, \ldots$. Inside a given document, all nodes are uniquely identifiable by their *node IDs* of the form $n_1, n_2, \ldots$ etc.

2.2 Extensions for optimization

The basic AXML language has been complemented with a few *special Web services* in a recent work [2]. We outline them below, with a few simplifications.

The *newNode(tree, address)* service provided by peer p copies the XML *tree* as a child of the node whose address is given as a second parameter. If *address* is null, the tree is installed as a new standalone document hosted by p. Once activated, this service returns a single response message with the ID of the tree installed at p.

The *send(data, address)* service provided by peer p continually sends *data* as children of the receive service call node identified by *address*. Different from *newNode*, *send* can transfer a whole stream (if *data* is a stream) as children of the destination nodes, as they arrive. Observe that *send* can transfer all the results obtained from the call to a continuous service, but we restrict it so that it does not transfer activated service calls which have not received *eos* yet. Intuitively, we do not want to "migrate executing calls"; instead, we are allowed to move calls which have not yet been activated, and calls which have completed execution.

The *receive(data, address)* service provided by peer p is used as a counterpart to the *send*. The *receive* is a place marker indicating where data from a *send* should be inserted. Thus, the calls to *receive* are created as part of the *send*'s execution, and they are activated immediately afterwards. The successful transfer of a stream of results takes place between two activated calls, one to *send*, one to *receive*. This is in the spirit of communication channels in pi-calculus [4]. The *data* argument of receive is a description of the data that are going to be received and the *address* is the identifier of the corresponding *send* that transmits data.

Moreover, the following extension has been introduced in [2]:

***Generic* services** are Web services identified by their name and WSDL signature, but whose provider peer is unknown. We designate such services by *s@any*, where *any* stands for any peer. Similarly, *generic documents* of the form *d@any* designate any replica of a given document. Observe that a generic resource (document or service) needs to be resolved into a particular concrete resource prior to being used.

2.3 AXML optimization

Given a document d on peer p and a query q, we are concerned with the efficient evaluation of $q(d)$. We need to compute the answers to q as if it was evaluated on the fully materialized document d (that is, as if all service calls in d were activated and fully evaluated prior to the evaluation of query q). Observe that this process can be quite complex, in particular because a service call result may include other service calls. Furthermore, given that the services called inside d may be provided by remote peers, the evaluation of q has an intrinsic distributed flavor.

The *default evaluation strategy* for the above problem is the following. Peer p activates all service calls in d, ensuring that in cases when a service call sc_1 is a parameter of sc_2, sc_1 is activated before sc_2. All intermediary service results transit through p. When the evaluation is finished, p evaluates q on the resulting document.

The *default evaluation strategy* is valid for not*special* service calls. Whenever a *special* call is encountered, it is activated directly without

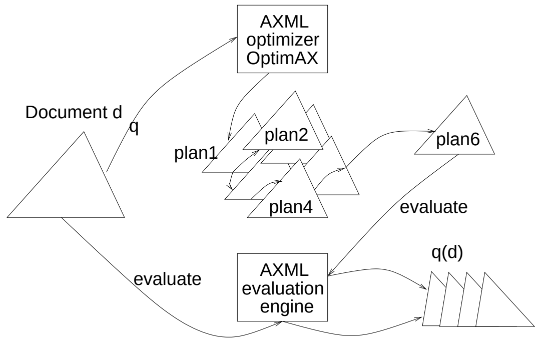

Figure 1: Outline of AXML optimization.

activating its descendant service calls. This order can be overriden by using an explicit *after-Activated* service call attribute to encode order dependencies between calls activations.

The process of *optimizing an AXML computation* consists of enumerating a set of *equivalent strategies* and of choosing one assumed to have lower computation costs. Here, *equivalent* means that the same query result is returned to the user at p, however, different peers, document and services may have been used during the computation. As for costs, we are first interested in reducing the *response time*, and second, reducing the *total work.* The global optimization and evaluation process is outlined in Figure 1.

Cost models for OptimAX For the moment, we consider cost models focused on communications (messages). Our simplest model $\mathcal{M}_1$ assigns a cost of 1 to each message crossing peer boundaries (i.e. whenever peer p activates a call to $s@p'$). All other computations, in particular local queries, are considered to have a cost of zero. This simple model reflects our experience [1, 5] that communications are by far more expensive than intra-peer computations. The second model we consider, $\mathcal{M}_2$, assigns a cost of bw_{p_1,p_2} to all messages exchanged between peers p_1 and p_2, where bw_{p_1,p_2} is a constant reflect-

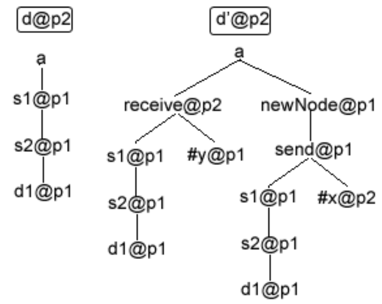

Figure 2: Delegation example.

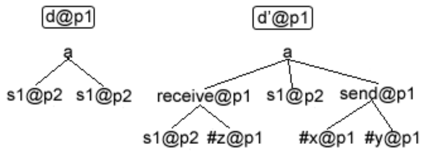

Figure 3: Factorization example.

ing the transfer latency between these two peers. The model $\mathcal{M}_2$ takes into account the difference between fast transfers in an intranet, and more lengthy ones, e.g. between a peer in France and one in China.

OptimAX optimizes AXML computations by applying *successive rewriting steps* on its AXML plans. The initial plan is a document containing one or more calls to ad-hoc services s_q. A s_q service is defined by the query q, and its parameters[1]. Subsequently, the optimizer rewrites this document by applying a set of rules. The following simple examples illustrate this process.

Instantiation Consider that a peer p has a plan $a(f@any)$ where a is a simple node and f a generic service. Before evaluating this document any has to be changed to p.

Delegation Consider the services $s_1@p_1$ and $s_2@p_1$ and let the plan $d@p_2$ be: $a(s_1@p_1(s_2@p_1(d_1@p_1)))$. Here, the (result of the) call to $s_2@p_1$ is an argument of the call to $s_1@p_1$. The default evaluation strategy would transfer $d_1@p1$ to p_2, then ship it back to p_1 in order to evaluate the call to s_2, receive these results at p_2, then ship them back to p_1 in order to evaluate the call to s_1, finally ship the results to p_2. Instead, the optimizer may rewrite $d@p_2$ into $d'@p_2$, which is being depicted at Figure 2. This plan represents the delegation of the whole computation to p_1. p_2 would only be responsible to receive the final result from p_1. More specifically the evaluation of d' at p_2 would result to calls to $recieve@p_2$ and $newNode@p_1$. The evaluation of the latter would establish the $send@p_1(s_1@p_1(s_2@p_1(d_1@p_1)), \sharp x@p_2)$ at p_1's repository as a new tree[2]. As soon as this tree is installed, its evaluation starts. Before $send@p1$ starts its execution, the subtree $s_1@p_1(s_2@p_1(d_1@p_1))$ has to be materialized (*afterActivated* is omitted in Figure 2 for readability). Note that *send* can start its execution as soon as some results of $s_1@p_1$ appear. Destination address of $send@p1$ is the address of $receive@p_2$ and has the form $peerId$. $docId$. $nodeId$ (for simplicity in the figure is depicted as $\sharp x@p_2$). The respective applies for $\sharp y@p_1$, which is the address of the send. Note that the execution of $s_1@p_1(s_2@p_1(d_1@p_1))$ at p_1 does not generate any communication cost, because document and services are hosted by the same peer.

Factorization Assume a document d at peer $p1$ that has two equivalent calls to the same service e.g. $s_1@p_2$ (see Figure 3). Calling more than once the same service leads to unnecessary communication cost that can be easily avoided. The solution is being depicted at the rewriting of d,

[1] For simplicity reasons $s_q@p(param_1, ..., param_n)$ can be written also as $q@p(param_1, ..., param_n)$.

[2] The second argument of new node is *null* and that's why it is not included in Figure 2.

d' (Figure 3). According to d' only one call is made to $s_1@p_2$. As soon as the results arrive at p_1, *send* and *receive* services are used to transfer them to the necessary positions. It must be noted that in the specific example, send, instead of having a service call as its *data* parameter it has an identifier ($\sharp x@p_1$). The latter identifies the service call that will produce the results ($s_1@p_2$ in our case).

Query pushing Consider a document of the form $f@p_1(g@p_2(d@p_3))$ where f is a selection service (filter), g any kind of service and d a simple document. Because of its nature, f can be "pushed" closer to $d@p_3$ without affecting the final result[3]. So the rewritten plan would be $g@p_2(f@p_1(d@p_3))$.

A legitimate question is how each peer learns about available peers, documents and services, as well as cost information. OptimAX is put to work in two settings: first, in a "network of friends" where peer p learns about other peer's existence gradually, as p calls services provided by these peers; second, in a DHT-based network, where a global distributed index is available to all peers. In this second setting, peers also insert in the index cost (bandwidth) information which all other peers can use.

3 Demo scenario and highlights

Our demo scenario concerns the collaborative development and distribution of software packages[4]. Several developers (marked as D_i in Figure 4), distributed all over the world, write updates for a set of software packages. Each developer works at his own location (D_i), and pushes

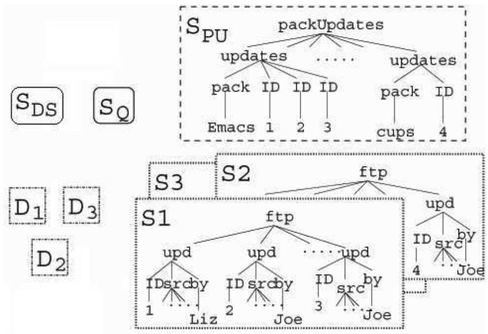

Figure 4: Distributed software management scenario.

his updates to one ore more servers geographically close to him (S_i). Each server hosts some, but not all, of the distribution's packages. In this context a typical user query is: *Whenever there is a new update on the Emacs package, I want to receive the update and the name of its developer.* From now on we will refer to this query as q.

The servers (and documents) that are going to concern us are the following:

- The document *packUpdates.xml* at site S_{PU} records associations between developers and the update(s) that they produced for each package.
- $S_1, S_2, ..., S_k$ are servers hosting updates. On each of them, the list of updates is in a file *ftp.xml.*
- S_{DS} contains the list of the S_i servers.
- S_Q is the peer where a query q is asked.

Our scenario assumes available the following services:

- `getUpdates@`S_i returns the updates from server S_i;
- `getServers@`S_{DS} returns the available S_i servers;
- `getUpdateInfo@`S_{PU} returns for each package the update ids that are related with that package;

[3]Peers p_1, p_2, p_3 may or may not coincide.

[4]We encountered this application in the *EDOS* [3] EU project, concerning the automatic management of the Mandriva(formerly known as Mandrake) Linux distribution.

To answer query q, we must make a call to the continuous services: `getUpdateInfo@`S_{PU} and `getUpdates@`S_i. In order to call the latter service, we must call `getServers@`S_{DS} to determine the S_i servers that are online. After retrieving at peer S_Q updates and update information, we must perform a *selection* on the update information based on the package name (*Emacs*) and then *join* the remaining ids with the ids of the updates.

The scenario described above is clearly not the best execution strategy. All the new updates (even the ones that are not related with *Emacs*) are received at S_Q. This results in an unnecessary network traffic that can be easily avoided. There are two obvious optimization steps that can improve the performance of q:

1. restrict the `getUpdateInfo` results only on the *Emacs* results. This can be done by pushing the *selection* to peer S_{PU}.

2. make the *join* of ids at S_i in order to avoid transferring unnecessary updates.

Thus, an efficient evaluation strategies is the following: For each available S_i, delegate to it the q. Moreover ask S_i to delegate the subquery that performs the *Emacs* selection over `getUpdateInfo` results to S_{PU}. This strategy can be represented by a plan with two nested delegations for each S_i; details are omitted here due to space limitations.

The demo will follow the rewriting process, showing at every step the current AXML plan via a GUI. We will highlight the interactions among peers, as well as the progressive gathering of information concerning the peers in the network and the costs of communications among them. We will demonstrate OptimAX both in a structured and unstructured network context.

The technologies that will be used for the demo are Java, AXML, Axis 2, XQuery and Web Services.

4 Related works and conclusion

The work is in the lines of distributed query optimization [7] put in a new light by the use of XML and Web services and contributing to the Web 2.0 vision of complex automated distributed data management. A characteristic of our work is the seamless transition between data, queries and query plans (which all are AXML). This is due to the inherent dual character of AXML documents, mixing intentional and extensional data. This common format simplifies things conceptually, but raises particular challenges at the level of handling plans during optimization, which we are currently working on in OptimAX. The ideas behind OptimAX have been presented in [2]; we ellaborate it and propose an actual system.

References

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *SIGMOD Conference*, 2004.

[2] S. Abiteboul, I. Manolescu, and E. Taropa. A framework for distributed XML data management. In *EDBT*, 2006.

[3] EDOS. Environment for the development and Distribution of Open Source software.

[4] R. Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[5] N. Ruberg, G. Ruberg, and I. Manolescu. Towards cost-based optimization for data-intensive web service computations. In *SBBD*, 2004.

[6] The Active XML Team. Active XML Primer . Technical report, Gemo, 2003.

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

[8] W3C. XQuery: An XML Query Language 1.0.