Chaining, Referral, Subscription, Leasing: New Mechanisms in Distributed Query Optimization

Arnaud Sahuguet Benjamin C. Pierce Val Tannen

University of Pennsylvania

Abstract

This work outlines a flexible framework for optimizing and deploying distributed queries in wide area networks. The database field has developed very powerful techniques for finding efficient execution plans for declaratively specified queries. However, applying these optimization techniques in the setting of distributed information management requires centralized knowledge of the entire network and assumes passive behavior from the data sources. The reality of the Web is different. Future distributed query optimizers must handle (in fact, exploit!) a rich variety of information flow mechanisms like chaining, referral, proxying, brokering, publish-subscribe, leasing, etc. We look to mobile agent technologies for the combination of flexibility and precision needed for handling these mechanisms. Our language-based approach uses a mobile process calculus based on the pi-calculus in combination with a powerful query-plan language. The salient characteristic of the language is that messaging, migration, and database operations all live in the same semantic space and interact, creating new opportunities for optimization.

1 Introduction

Starting with the classic dynamic programming technique for join ordering [50], the database field has developed powerful techniques for finding efficient execution plans for declaratively specified queries [30, 21, 36]. However, applying these optimization techniques in the setting of distributed information management [45] requires a good deal of centralized knowledge in "master" sites and assumes stringent limits on the dynamic behavior of the other, "apprentice", sites.

The reality of the Web is different. Distributed query plans must cope with significant degrees of independence in the behavior of the data sources. A typical example, which we call *referral* (following [33]), is when a site A ships a query to a site B and may get back, instead of the answer to the query (as data), another query that will produce this answer, if executed by A. Executing this other query may cause A to ship a query to site C, and so on. Moreover, site B may be independent enough that B's choice between a straight data answer and a referral query cannot be determined by a query plan produced at site A. Related strategies include *chaining*, where a site acts as a proxy and forwards queries to the actual source, *subscription*, where a site receives periodically refreshed information corresponding to a subscription query installed elsewhere, and *leasing*, where resources are reserved in advance to guarantee fast distributed processing for specified amounts of time or until certain events occur.

Such mechanisms have not been considered in traditional distributed query processing. But in fact, extending the flexibility of distributed query plans should be seen as an opportunity rather than an

obstacle to efficiency. Sites almost always process sequences of queries. Local caching of partial results is often used for amortized improvements of such sequences of queries. We consider in this paper a more flexible mechanism, *remote caching,* that requests another site to compute and cache the results of a query, to be used later. Or, for volatile data, a remote site may accept to run repeatedly a subscription or *continuous query* [22] whose results are needed elsewhere. Also, while getting back a referral instead of an answer slows down site A, this may be a good decision for an optimizer at B that has to consider the total load there.

In this vision paper we explore how such mechanisms can be exploited in building complex distributed information management systems that integrate independent sites in a volatile environment like the Web. The difficult questions are obvious: (1) can this be done by deploying a relatively small generic infrastructure in each node, and (2) is it worthwhile? We believe that by borrowing from mobile agent technologies we can answer (1) positively, and that in turn this will enable us in the future to build prototypes that can settle (2).

We look to mobile agent technologies for the combination of flexibility and precision needed for expressing, optimizing, and deploying queries using mechanisms such as referral, subscription, etc. Mobile agent systems have been developed in a number of domains including e-commerce [59, 60], user-interfaces [12], knowledge management [38, 8, 14], active networks [56], and general distributed programming [15, 25]. Though details vary, these systems offer similar core functionality, including primitives for transparent migration, location naming, and inter-agent communication.

Our starting point is a pi-calculus [44] enriched with primitives for process migration and remote communication. To this we add the primitives of the query plan language used in [24, 47] to obtain a small and semantically clean language for expressing and evaluating distributed queries. The basic computational flavor of this language (process migration, channel-based message-passing, etc.) comes from the pi-calculus primitives; its basic datatypes and the operations on them come from databases.

The salient characteristic of the language that we construct by merging the pi-calculus with database primitives is that messaging, migration, and database operations all live in the same semantic space and interact, creating new opportunities for optimization. This contrasts with previous systems [8, 15] in which a messaging layer (e.g., KQML) and a query language (e.g., KIF, LOOM, SQL) are embedded in a general-purpose host language; in these systems the distributed logic of query plans is hard-coded using the host language and is thus not accessible to the optimizer. Using a single language allows the optimizer to systematically explore alternatives that exploit mechanisms like referral, subscription, etc.

The rest of the paper is organized as follows. In Section 2 we present some motivating examples and describe our query language. The architecture we envision for this system is outlined in Section 3. We present some more complex mechanisms (subscription and remote caching) in Section 4. In Sections 5 and 6 we present related work and conclude with remarks and open questions.

2 Motivating Examples

2.1 Chaining and referral

These mechanisms did not arise in traditional distributed databases, yet they should be familiar to our reader: referral (redirection) is part of HTTP [10]—the protocol underlying the Web—and both referral and chaining are part of LDAP [32, 33, 34] (Lightweight Directory Access Protocol), a protocol used, for example, by domain name servers and email clients. Although in HTTP and LDAP these

mechanisms are used for very special data and distribution models, we shall see that our language can express them in full generality, for any kind of distributed queries. Later in the paper we will discuss subscription, leasing, and related mechanisms.

The HTTP referral mechanism is often used when a resource available of a given server and identified by a URL has been moved. The server which cannot serve the resource anymore will send back to the client the new URL where the resource has been moved. The client will then have to follow this *redirection*.

An LDAP-based network directory can be viewed as a highly distributed database, in which the directory entries are organized into a hierarchical tree-like namespace and can be accessed using database-style search functions. An LDAP server, when asked for a lookup, can face the following situations: (1) the resource points to its own namespace or (2) the resource points to outside of its namespace. For (1), the server will simply return the resource if it exists. For (2), the server will try to resolve the naming context by walking up or down the LDAP tree.

For a given incoming query from the client, this tree traversal can be performed in two ways. With **chaining** (Figure 1) the server being asked for a resource will cooperate with other LDAP servers to get the result back to the client. The query resolution is completely invisible to the client. With **referral** (Figure 1) the server will simply tell the client which server to contact to get the corresponding information. This is similar to the HTTP redirection and there might be multiple rounds of referrals. Both strategies have their pros and cons, depending on the costs of the different connections.

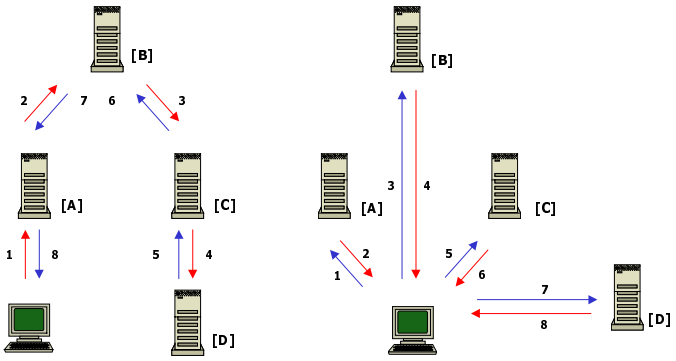

Figure 1: Information flow mechanisms: chaining and referral

In our language we will use *query process migration* and *channel-based communication* between query processes in order to capture these mechanisms.

2.2 Our Query Process Language

We now discuss in more detail the language in which our distributed query processes are expressed. This language can be described as a combination of primitives from the *nomadic pi-calculus* [52] with the primitives of a language for expressing database query plans[24]. We focus here on the aspects of the language dealing with distributed query plans; the database primitives are discussed in section 2.4 below.

The *basic values* of the calculus are the entities that can be the final results of evaluating an expression or sent from one process to another along a communication channel. The basic values are: pure data values, written v (including large values such as relations); communication channels, written c; and site

names, written s. In addition, we assume a set of abstract resource names, written R, which generalize entry point names in schemas, such as relation names, class extent names or document names, are also basic values. We assume that the set of abstract resource names is partitioned among the sites, so that each name is associated with just one site.

The syntax of our calculus is divided into two parts (cf. Figure 2): processes (P) and expressions (e). Processes simply compute until they are finished and then terminate. Expressions, on the other hand, are expected to return a value that will be used by some enclosing computation.

Processes			Expressions		
P ::=	$\mathbf{0}$	inert process	e ::=	s	site name
	$P \parallel P'$	parallel composition		$\mathcal{R}$	abstract resource name
	$*P$	replicated process		v	basic value
	new c in P	channel creation		x	variable
	c@s!e	output		$P \parallel e$	process spawning
	c?x. P	input		new c in e	channel creation
	<go s do P>	migration		e@s	remote evaluation
				?c	channel
				(omitted)	database primitives

Figure 2: Our query process language

Most of the process constructors are familiar from the pi-calculus: the inert process $\mathbf{0}$ has no behavior; the parallel composition $P \parallel P'$ runs P and P' as separate lightweight threads; the channel creation expression new c in P ensures that c is a fresh name, different from any other name used anywhere else in the system, and then behaves like P; the input process $c?x.\ P$ reads a value from the channel c, binds it to the variable x, and executes P; the replicated process $*\ P$ behaves like an infinite number of copies of P running in parallel. Two more novel constructs (both taken from nomadic pi-calculus) are the located output primitive c@s!e, which evaluates the expression e and sends the result on the channel c to any receiver at the site s (if there is no receiver on c currently running at s, the message is held at s until there is one), and the migration <go s do P>, which starts the process P running at the site s.

The spawning expression $P \parallel e$ runs a process P in parallel with the evaluation of the expression e—it is the analog in the "expression world" of parallel composition of processes. Similarly, channel creation new c in e is analogous to channel creation in processes. Remote evaluation e@s is the remote execution of e at site s; the result is sent back and it becomes the value of $e@s$. The channel input expression $?c$ waits for a communication on c and yields the value read as its result. Other syntactic forms used in the examples (e.g., sequential expressions $e; e'$) can be derived from these basic forms as syntactic sugar.

The most interesting construct here is the remote evaluation of expressions. This is implemented by rewriting it in terms of migration and channel communication. The expression expr@site calls for the evaluation of expr at site. When we come to evaluating such an expression on site A, we replace it by new c in <go B do c@A!e> $\parallel$?c. That is, we create a new private channel c, spawn a process that migrates

to site B and evaluates e, and listen on c for the result sent by this process.

2.3 Implementing chaining, referral, and more

We show here how to express in our query process language chaining and referral. We describe these mechanisms for *uni-target* queries, i.e., queries that need data residing on just one server. The only interesting part of answering such queries is locating this server and getting the information back to the site where it is needed. In the process, we describe a third mechanism, recruiting that arises naturally as a likely improvement to both chaining and referral. We chose its name in reference to a KQML [38] performative with related meaning.

The implementation is described in the table below, where each column represents the running query processes at a given site and each row a step of process evaluation, local query evaluation, or optimization. The client K requests the result of query q from server A. Server A knows that the desired answer can be obtained by running query q' at server B.

The rewriting of q to q'@B is an optimization step. We regard it as such because in general q and q' may not retrieve information in the same way. When q is simply an abstract resource name (such as a relation name), q'@B is more a "definition" than an optimization, but for simplicity we shall let the optimizer take care of these cases too.

All this corresponds to the four common steps in Figure 3. From this point on, the strategies differ. Chaining continues with what is actually the standard process evaluation in our language. q' eventually evaluates at B to a value v that is passed back on two channels to the client (Figure 4). Referral relies on an optimization that migrates back to K the referral for evaluating q' at B (figure 5).

client K	server A	server B	step type
q@A			orig query
new c in <go A do c@K!q> || ?c			process eval
?c	c@K!q		process eval
?c	c@K!(q'@B)		optimization

Figure 3: Steps common to all mechanisms

client K	server A	server B	step type
starting from the last step of Figure 3			
?c	c@K!(new c' in <go B do c'@A!q'> || ?c')		process eval
?c	c@K!(?c')	c'@A!q'	process eval
?c	c@K!(?c')	c'@A!v	local query
?c	c@K!v		process eval
v			process eval

Figure 4: Chaining

client K	server A	server B	step type
starting from the last step of Figure 3			
?c	<go K do c!(q'@B)>		optimization
?c ‖ c!(q'@B)			process eval
?c ‖ c!(new c' in <go B do c'@K!q'> ‖ ?c')			process eval
?c ‖ <go B do c@K!q'>			optimization
?c		c@K!q'	process eval
?c		c@K!v	local query
v			process eval

Figure 5: Referral

Upon examination of these two mechanisms, a third alternative, **recruiting**, (not offered by LDAP) suggests itself (see Figure 6). The strategy is the following: the server A already has the name of the channel on which the client K expects the answer. It then simply asks B to evaluate q' and send the answer on that channel (see Figure 7).

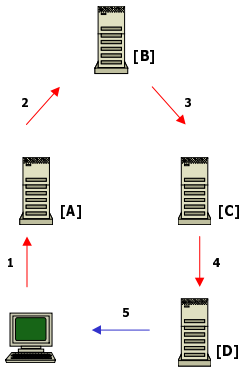

Figure 6: Recruiting

client K	server A	server B	step type
starting from the last step of Figure 3			
?c	<go B do c@K!q'>		optimization
?c		c@K!q'	process eval
?c		c@K!v	local query
v			process eval

Figure 7: Recruiting

For referral, we can also capture the case where server A sends back to client K the answer as a partial

result *value* combined with a query that needs to be evaluated by the client on server B. Going back to Figure 5, we could write the first step for server A as: <go K do *value* U c!(q'@B)>. This strategy is very common for directory services where information is partitioned. A similar approach has been presented in a different context to handle unavailable data sources using *parachute queries* [13].

An important remark is that the proper handling of the mechanisms described requires close dynamic cooperation between process evaluation, expression evaluation, and query optimization (as we shall see in section 3).

The three mechanisms have their pros and cons [1]. Chaining puts more overhead on the server which needs to keep connections open while waiting for answers to come back. On the other hand, chaining is completely transparent for the client. Referral puts more overhead on the client; for the server, connections are closed as soon as the referral has been sent to the client. In terms of total communication costs, referral is in many cases better than chaining. However in the case where connections from client to servers are expensive (e.g., client and servers live on different sides of a firewall), chaining could be better than referral.

If we look only at communication steps (see Figure 1), recruiting is better than both referral and chaining. However, the total cost picture can be more complicated. Here are two examples in which we have relaxed the uni-target assumption. In a case when the client's query (shipped to server A) is a join between a big relation located on A and a small relation located on B, recruiting may result in the big relation being sent from A to B while chaining could be resolved just by sending the small relation from B to A. In a case when the client has additional information (in a cache for example) that can help with evaluating queries on B, a referral to B sent back from A could be re-optimized into a better plan. This opportunity is short-circuited by the recruiting mechanism.

The bottom line is that all these mechanisms, and more, should be made available to optimizers,

2.4 Multi-target queries

Query migration can be used also to capture various optimization techniques for *multi-target*, i.e. queries that need data residing on several servers. Semijoin programs [11] are an example of such a technique. Although its utility was deemed questionable for standard distributed RDBMS [43, 42], this method could be very useful in our context. It could for example be used to minimize the unnecessary shipping of bulky multimedia information.

But more interestingly, the core idea of semijoin programs (ship only necessary data) can be generalized to make use of physical access information such as join indexes, access support relations, gmaps, etc., cached in convenient locations. To do this successfully, our optimizers need be able to *"rewrite queries using views"* [39] where "views" is interpreted broadly to also mean cached queries and cached physical access data. Our recent work [47] offers a technique for doing this. The optimizers envisioned here incorporate this method for query rewriting.

This is why we have chosen the language of [24] for the database fragment of our query process language. The examples in this vision paper will not need too many details of the query language. Also, for simplicity, we will use here select-from-where or relational algebra notations, which are expressible (as syntactic sugar) in our language.

The following example illustrates the implementation in our language of the idea of query rewriting and decomposition for minimizing communication costs. Suppose that servers A and B host respectively

[1] A more detailed comparison between chaining and referral can be found in [33].

relations R and S and that M is a "mediator" site (see Figure 8) that needs the result of the following "generalized" join:

Join: $J(R,S) \stackrel{def}{=}$ select $E(r,s)$ from $R\ r,\ S\ s$ where $B(r,s)$

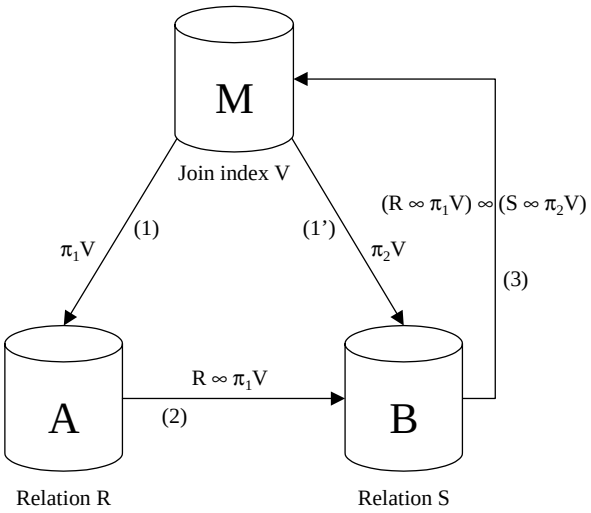

Figure 8: Plan 2, where $V = JI_{RS}$

We could use a semijoin technique with:

Semijoin: $SJ(R,S) \stackrel{def}{=}$ select r from $R\ r,\ S\ s$ where $B(r,s)$

A more interesting assumption is that M caches the following:

Join index: $JI_{RS} \stackrel{def}{=}$ select $r.tid, s.tid$ from $R\ r,\ S\ s$ where $B(r,s)$

(Here *tid* stands for "tuple-id" aka *surrogates*) Based on the following equivalences (where $\ltimes$ denotes the natural semijoin):

$$\begin{aligned} J(R,S) &= J(SJ(R,S),S) \\ J(R,S) &= J(SJ(R,S),SJ(S,R)) \\ SJ(R,S) &= R \ltimes \Pi_1(JI_{RS}) \end{aligned}$$

there exist several plans for evaluating $J(R,S)$. We list three main ones, and for each we give the corresponding query plan process generated at site M:

Plan 1 M sends $\Pi_1(JI_{RS})$ to A and gets back $R \bowtie \Pi_1(JI_{RS})$, and in parallel sends $\Pi_2(JI_{RS})$ to B and gets back $S \bowtie \Pi_2(JI_{RS})$; finally M computes the result locally.

Plan 2 M sends $\Pi_1(JI_{RS})$ to A, asking A to compute $R \bowtie \Pi_1(JI_{RS})$ and send it to B; at the same time, M sends $\Pi_2(JI_{RS})$ to B and asks it to compute the result (using what comes from A) before sending it back to M (see Figure 8).

Plan 3 is the same scenario, except that M does not send the data along with the query but asks the remote nodes to fetch it (by migrating to M a process that sends the value on a channel, and listening locally on this specific channel).

Plan 1	new c_a, c_b in <go A do c_a@M!(R $\bowtie \Pi_1(JI_{RS})$)> ‖ <go B do c_b@M!(S $\bowtie \Pi_2(JI_{RS})$)> ‖ ?c_a $\bowtie$?c_b
Plan 2	new c,c' in <go A do c'@B!(R $\bowtie \Pi_1(JI_{RS})$)> ‖ <go B do c@M!(?c' $\bowtie$ (S $\bowtie \Pi_2(JI_{RS})$))> ‖ ?c
Plan 3	new c,c' in <go A do c'@B!(R $\bowtie$ (new c_a in <go M do c_a@A!($\Pi_1(JI_{RS})$)> ‖ ?c_a))> ‖ <go B do c@M!(?c' $\bowtie$ (S $\bowtie$ (new c_b in <go M do c_b@B!($\Pi_2(JI_{RS})$)> ‖ ?c_b)))> ‖ ?c

The full measure of the flexibility of our approach can be seen if we add the following complication: suppose that A does not store R but R is an abstract resource name for which A has a definition (i.e., R is an unmaterialized view) that may involve data stored (or, in turn, views defined (!)) at other sites. This view definition is not available to the client K.

Then, the optimizer at K comes up with a plan as seen above, where a subordinate plan referring to R migrates to A. But here, this subordinate plan gets re-optimized, beginning with the "concretization" of the definition of R. Multi-step optimization has been considered extensively (eg., [41, 19, 53, 31]) but previous approaches assume a degree of centralization.

3 Architecture

We will assume that the infrastructure in each node (each site) contains an **execution engine** for query processes. The tasks of this execution engine can be grouped into **process evaluation** and **query evaluation**. Without necessarily committing to implementing these as separate components, we shall talk about a process "evaluator" and a query "evaluator" who keep calling each other. The overview of the architecture is presented in Figure 9.

At the top level each node's process evaluator runs a collection of query processes in parallel (as lightweight threads). This collection may grow as incoming query processes migrate to our node or shrink as query processes finish their execution or migrate elsewhere. Some of the incoming query processes contain client queries; others contain subordinate query plans from some plan execution in another node; yet others contain subscriptions or remote caches that another node wants to install in this one. The process evaluator interacts with the **migration manager** who handles both incoming and outgoing process migration and with the **channel manager** who handles communication.

The query evaluator is called on expressions with database primitives and abstract resource names. Each time the query evaluator is called, it begins by invoking the **single query optimizer**. Such dynamic optimization is necessary in order to support mechanisms such as referral and in order to take advantage of, e.g., subscriptions or caches that can be made known to the optimizer continuously.

Indeed, in addition to the single query optimizer, the infrastructure also contains a **continuous optimizer** [22] that runs as a separate thread and is responsible for identifying some interesting *patterns* – at the level of processes running locally – and for installing local/remote caches and subscriptions that are likely to be useful in future optimizations.

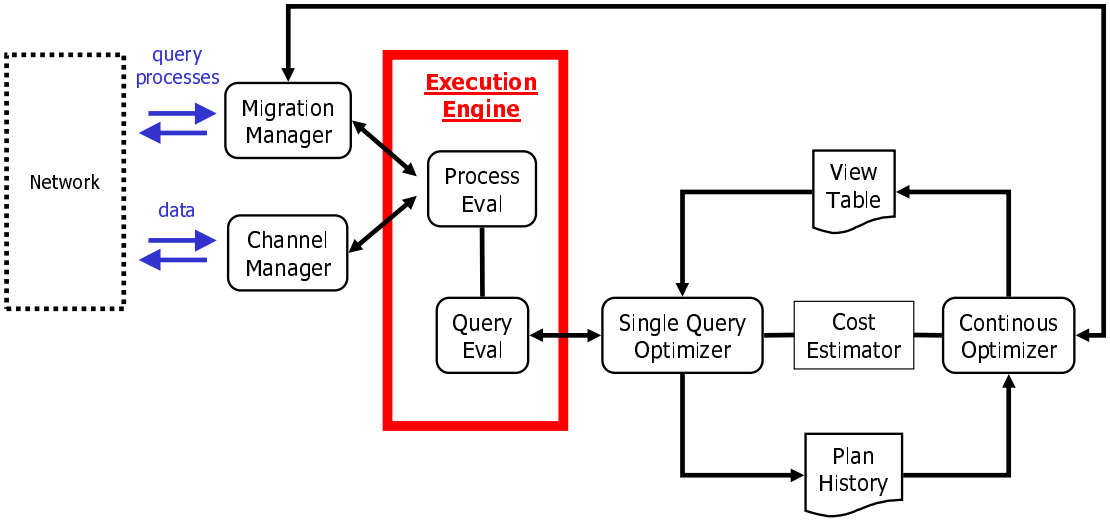

Figure 9: The architecture at each node

Concretely, the continuous optimizer periodically examines a **history** of plans chosen by the single query optimizer for the various queries it has processed. Keeping track of historical and structural information similar to [51, 6, 22], using techniques such as common subexpression identification, and using various statistics in order to make decisions about cost and amortized cost, the continuous optimizer decides when the data captured by a certain expression is worthy of attention.

Example of continuous optimizations include: frequently used or constructed relations could be cached; recurring (sub)queries could be transformed into subscription-based queries; frequently usable physical access structures such as join indexes could be materialized; etc. For simplicity, we shall call them all "views".

Depending on location distribution, costs, and estimates of data refreshing rates, the continuous optimizer installs the identified views as either local caches, remote caches, or subscriptions. Like the process evaluator, the continuous optimizer interacts with the migration manager to ship to the right node the subscription and remote cache processes it generates.

The information about available views is stored in a **view table** updated by the continuous optimizer and used by the single query optimizer, eg.:

ViewExpression	ViewImplementation
JI_{RS}	scan(localCache)
R@Publisher	?chanSubscr

Through the rewriting-using-views procedure, one or more occurrences of the ViewExpression is replaced in the query by the corresponding ViewImplementation. The latter could be as simple as a scan of a local cache, or listening on a channel for subscription data, as illustrated by examples above. Note that, as usual, several rewritings may be possible and the optimizer will choose between plans based on cost.

Retrieving the content of a remote cache is explained in section 4.2. Even more complicated combinations are easily expressible: having a local cache for a remote subscription subscriptions that use other subscriptions (subscription chaining), and various forms of replication.

4 More Complex Mechanisms

4.1 Subscriptions

The concept of subscription has been around for a while: just look at newspapers. The key idea behind it is that subscribers will express interest to publishers for some item, and that publishers will ship the item to subscribers, according to an agreed upon policy defined by the subscription itself. For Web information management, the same idea has been *resurrected* as the "push" in "push/pull". Normal retrieval on-demand is a pull, and a push is when data is sent to users without them having to request it every time. The advantages of push are: no need to know where the data is actually located (server can move), no need to perform frequent polling to detect new information. In the framework we envision, subscriptions are a key element because they establish a long-lived relationship between a client and a server.

When the continuous optimizer (see Section 3) at site A detects a recurring subquery of the form q@Publisher, it may decide to install a subscription at site Publisher for q. It does this by creating a new channel chanSubscr and by spawning a query process that will migrate to Publisher. As we saw in section 3 the view implementation for the view expression q@B is simply ?chanSubscr. The process spawned has the form:
<go Publisher do ∗ event?x. chanSubscr@A!q>.

This uses process replication and leads to the repeated execution of event?x. chanSubscr@A!q. This process will block until it receives input on channel event. In this manner we can express waiting for a time pulse (eg., every n minutes a "clock" process at Publisher sends something on event or waiting for events other events generated elsewhere, such as "on change" that could be generated by a DBMS upon executing an update.

Notice that the subscription installed by A only sends the data to A. It is possible that several sites will subscribe to the same query q. This would have to be detected and exploited by the continuous optimizer at Publisher who may set up a sharing mechanism.

4.2 Remote caching

Subscriptions are used when we need remote data that changes often. But sometimes we would like to evaluate a query and store its result remotely even if the data is stable. Evaluating a query remotely has obvious potential benefits. Assuming we do not wish to re-compute the query, where would we cache the result? The remote site could send the data back to a housekeeping process that will cache it locally. But sometimes caching the result remotely is a better idea. For instance, the remote node may have more storage resources. More subtly, the result may be quite big and we may, in the future, avoid moving it all by using semijoin techniques. Servers can also be interested in establishing remote caches (*mirrors*) to bring information closer to the clients[2].

To implement remote caching by A of q!B, the continuous optimizer at A spawns the following process:
new rcChan in <go B do storeLocal(q,Cache); *(rcChan?x. x@A!scan(Cache))>.

Meanwhile, the view table at A creates the following entry:

[2]Such services – along with some smart routing and load-balancing algorithms – are offered for Web contents by companies like [1, 3, 2]

ViewExpression	ViewImplementation
q@B	new c in rcChan@B!c || ?c

What happens here is that for every use of the remote cache in some plan at A, a channel name c is created and sent through the originally established channel rcChan to a process at B. That process receives the channel name c and uses it to send the content of the remote cache back to A.

4.3 Leasing

So far we have assumed that all sites are happily co-operating by welcoming any incoming migrating query. This is not a realistic assumption. Making all these mechanisms work in the context of independent data sources will require some control mechanisms. Leasing is such a mechanism. Expensive services like remote caching or real-time subscription should be welcomed subject to approval and for a limited amount of time.

We can imagine that the continuous optimizers of different sites engage in a negotiation protocol to define the terms of a *lease* both in terms of duration and content. Once leases enter the picture, they will play an important role in optimization.

Knowing that a leasing contract holds, a site can (and sometimes must) use optimizations that rely on the contract itself. For example, the view implementations in the view table may have a limited lifetime. This may encourage alternative implementations for the same view. In a different context, we can imagine that a client will share a contract with the server that guarantees that the relation it exports is always sorted or duplicate free which enables some optimizations.

How do we implement leasing in our query process language framework? One aspect is the negotiation protocol. Here we could be guided by related work in distributed agents[15]. Once a lease is agreed upon, it takes the form of a *time-bomb* that is incorporated in the subscription or remote cache processes. Implementing time-bombs in process calculi best uses a *failures* mechanism as in [25]. We leave the details to future work.

5 Related work

Distributed Query Processing: The state of the art in distributed query processing is nicely presented in the recent survey [36] by Kossmann while the classic work in distributed databases is covered in [20, 45].

The standard approach remains the one of System R* [41] with master and apprentice sites: the master site is in charge of developing the global plan and make the inter-site decisions; apprentice sites take the portion of the global plan that are relevant for them and develop a local plan for themselves, within the constraints of the global plan. In [61], the same system can also capture environments with local and remote objects.

The optimization algorithm used to generate the best plan is an extension of the dynamic programming algorithm [50]. The *textbook* generalization is described in [36]. A recent improvement called *iterative dynamic programming* can be found in [37].

In contrast with the relatively centralized approach of System R*, Mariposa [53, 54] offers a decentralized solution where every node is governed by economic motivations. Optimization decisions are based

on bids and offers with negotiated prices, for a given budget. The use of economic models permits to define in a decentralized way the global behavior of the system. Although decentralized, the approach of Mariposa and ours are complementary. Maripose does not focus on the information flo mechanism that is of interest to us.

Other systems that perform distributed query optimization include Garlic [31], DISCO [57], DIMSUM [28] and ObjectGlobe [16]. Distributed query optimization has also been considered for some more specific application like directory services [7] and even semi-structured data [55, 4]. Previous work on parallel database and query grouping [51] is also relevant to our work. More recently the use of continuous queries [40, 22] has required the use new optimization techniques to handle subscriptions.

Query rewritings using views and caches: Many of the optimizations shown in our paper use query rewritings with materialized views and caches. Previous work on using caches include [6, 5]. There has been much work on rewriting with views, most recently [24, 47, 48], see the nice survey [39] by Levy.

Cost models: Relevant work on cost models includes total-cost estimates [43] and response-time estimates [29]. The importance of cost models has been re-emphasized for mediator-based architectures in [49].

Data warehousing: An obvious solution to the information integration problem is to simply build a big warehouse – with a unique big schema – and to store all the data available (multiple references). This option is commonly used by big organizations that need to integrate corporate data (after an acquisition, for instance), but it does not scale for systems that integrate many volatile Web-like sources subject to variability of schema and data. It should be kept in mind that warehousing in not a one-time process but a continuous one, with significant costs devoted to maintenance.

Software Agents: A survey of recent developments in agent-based technologies appears in [15]. SIMS [35] for instance is a an agent-based system for gathering information in which the agents are mediators. Its communication layer is KQML [38] and its query language is LOOM, an object-oriented schema logic. SIMS performs some semantic optimizations by rewriting LOOM expressions, but any use of alternative information flow mechanisms would have to be hard-coded in the agent programs.

Mobile Process Calculi: The past few years have seen the development of a number of compact and elegant formalisms capturing various aspects of mobile agent computation. The join-calculus [26, 27] forms the core of a general-purpose distributed programming language. The emphasis is on powerful primitives for grouping agents into nested "locations" that fail atomically and permanently, and whose failures can be observed from other locations. The Ambient Calculus of Cardelli and Gordon [17] and Vitek and Castagna's SEAL calculus [58] focus on administrative domains and the capabilities needed to enter and leave them. Advantages of mobile software agents are discussed in [23].

6 Future Directions

We have proposed a flexible framework for representing, optimizing, and evaluating distributed queries, combining the strengths of distributed database and mobile agent technologies. Of course, as befits a vision paper, most of the interesting questions still lie ahead. We sketch here some of the most important avenues for further investigation.

Rewriting Rules

By combining a process calculus with the database primitives of [47, 24], we hope to take advantage of the powerful and practical rewriting techniques presented in [47]. However, this is not clear how to tune the interaction of the *Single Query Optimizer* with the *Continuous Optimizer* in order to get good performance. In our framework, the cost model will play a central role and needs to capture information distributed over nodes, amortized cost for subscription and caching and should be self-reconfiguring in order to cope the variability of the environment.

Implementation

The next major step is implementing the proposed infrastructure for migration, communication, process management, and optimization. Low-level database operations can be farmed out to existing servers. It is not clear either if the implementation of such a system requires a implementation from scratch, an enriching of a database framework with new delivery mechanisms, or an enriching of a messaging service like [9] with some database capabilities.

There are also issues of high-level language design to be addressed. What we have described here is a "pure" mobile pi-calculus as it would be seen by the run-time system and the optimizer. For the convenience of programmer writing queries in this calculus, we would want to provide a variety of high-level syntactic forms (e.g. the ability to define and use functions, common data structures, etc.) either as libraries or as syntactic sugar. Our experience with the Pict language design [46] should come in handy here.

Tuning the Framework

Finally, the simple calculus we have described here leaves several points to be addressed.

Location-aware vs. location-independent communication. The nomadic pi-calculus provides both point-to-point communication and "location independent" communication where the sender need not know the site where the receiver is currently located (the run-time system has the job of locating the receiver and delivering the message). We have not considered the latter form of communication since it is much more expensive to implement, but for some examples it may be very useful.

Process-passing. For some examples, it appears that it may be useful to be able to pass not only values but *processes* along communication channels. "Higher-order" process calculi that allow this sort of thing have been studied in the pi-calculus literature.

Other mobility primitives. The nomadic pi-calculus includes some other primitives that, for simplicity, we have not considered here. These include the ability to group processes into *agents*, and to test for the presence of a given agent at the local site. It may be useful to include some of these (or their many variants). Other primitives such as the service combinators presented in [18] are useful to describe timeouts and competition among processes.

Some more philosophical issues also remain to be understood. In particular, the *correctness* of optimizations is a subtle matter. In conventional process calculi, the usual notion of correctness is contextual equivalence: an optimized version of a process is equivalent to the original if there is no observing context that can tell the difference between them. In the present setting, this demand seems much too strong: when we used cached intermediate results in answering a query, for example, we do not expect to obtain precisely the same results, since information in the master copy of the database may have

changed. For some applications even slightly stale results will be unacceptable; for others, timeliness will be much more important than freshness. Reasoning rigorously about such "correct enough" results poses a significant challenge.

Acknowledgements. Alin Deutsch and Lucian Popa collaborated with us in the initial phase of this work. We thank them and Wang Chiew Tan for useful comments on the draft of this paper.

References

[1] Akamai Inc. `http://www.akamai.com`.

[2] Inktomi Inc. `http://www.inktomi.com`.

[3] SandPiper Inc., now Digital Island Inc. `http://www.digisle.net`.

[4] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 122–133. ACM Press, 1997.

[5] Sibel Adah, Kasim Selcuk Candan, Yannis Papkonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2), 1996.

[6] S. Adali, K. Selcuk Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD*, pages 137 – 148, 1996.

[7] Sihem Amer-Yahia, Divesh Srivastava, and Dan Suciu. Distributed Evaluation if Directory Queries. Submitted for publication, Feb 2000.

[8] Y. Arens and C. Knoblock. SIMS: retrieving and integrating information from multiple sources. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):562–563, June 1993.

[9] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *Middleware Workshop at the International at the Conference on Distributed Computing Systems 1999*, 1999.

[10] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0, May 1996. Status: INFORMATIONAL.

[11] P. Bernstein and D. Chiu. Using semijoins to solve relational queries. *Journal of ACM*, 28(1), 1981.

[12] Krishna Bharat and Luca Cardelli. Distributed applications in a multimedia setting. In *Proceedings of the First International Workshop on Hypermedia Design*, 1995.

[13] Philippe Bonnet and Anthony Tomasic. Partial Answers for Unavailable Data Sources. Technical Report 3127, INRIA, March 1997. Available at http://www-rodin.inria.fr/publications.

[14] Jeffrey M. Bradshaw. KAoS: Toward an Industrial-Strength Open Agent Architecture. In Jeffrey M. Bradshaw, editor, *Software Agents*, chapter 17. MIT Press, 1997.

[15] Jeffrey M. Bradshaw, editor. *Software Agents*. MIT Press, April 1997.

[16] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Alexander Kreutz, Stefan Prls, Stefan Seltzsam, and Konrad Stocker. ObjectGlobe: Ubiquitous Query Processing on the Internet. Technical report, Universität Passau, 1999.

[17] L. Cardelli and A. Gordon. Mobile ambients, 1998.

[18] Luca Cardelli and Rowan Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, May/June 1999.

[19] Michael J. Carey and Hongjun Lu. Load balancing in a locally distributed database system. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, pages 108–119. ACM Press, 1986.

[20] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, 1984.

[21] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 34–43. ACM Press, 1998.

[22] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas,TX, 2000.* ACM Press, 2000.

[23] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Technical Report RC 19887 (88465), IBM, T.J. Watson Research Center, 1995.

[24] Alin Deutsch, Lucian Popa, and Val Tannen. Physical Data Independence, Constraints and Optimization with Universal Plans. In *International Conference on Very Large Databases (VLDB)*, September 1999.

[25] Cédric Fournet. *The Join-Calculus: A Calculus for Distributed Mobile Programming.* PhD thesis, École Polytechnique, Paris, France, 1998.

[26] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Principles of Programming Languages*, January 1996.

[27] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag. LNCS 1119.

[28] Michael J. Franklin and Stanley B. Zdonik. "Data In Your Face": Push Technology in Perspective. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 516–519. ACM Press, 1998.

[29] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 9–18. ACM Press, 1992.

[30] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[31] Laura M. Haas, Renée J. Miller, B. Niswonger, Peter M. Schwarz Mary Tork Roth, and Edward L. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

[32] T. Howes and M. Smith. RFC 1823: The LDAP application program interface, August 1995. Status: INFORMATIONAL.

[33] Tim Howes, Mark C. Smith, Gordon S. Good, and Timothy A. Howes. *Understanding and Deploying LDAP Directory Services*. Number 1578700701 in Network Architecture and Development Series. MacMillan, Jan 1999.

[34] H. V. Jagadish, Laks V. S. Lakshmanan, Tova Milo, Divesh Srivastava, and Dimitra Vista. Querying Network Directories. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1999.

[35] Craig A. Knooblock and Jose Luis Ambite. Agents for information gathering. In Jeffrey M. Bradshaw, editor, *Software Agents*, chapter 16. MIT Press, 1997.

[36] Donald Kossmann. The State of the Art in Distributed Query Processing . Submitted to ACM Computing Surveys.

[37] Donald Kossmann and Konrad Stocker. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. Submitted to ACM TODS, 1999.

[38] Yannis Labrou and Tim Finin. A Proposal for a new KQML Specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250, February 1997.

[39] Alon Levy. Answering queries using views: a survey. Submitted for publication, 1999.

[40] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[41] Guy M. Lohman, Dean Daniels, Laura M. Haas, Ruth Kistler, and Patricia G. Selinger. Optimization of nested queries in a distributed relational database. In Umeshwar Dayal, Gunter Schlageter, and Lim Huat Seng, editors, *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings*, pages 403–415. Morgan Kaufmann, 1984.

[42] Hongjun Lu and Michael J. Carey. Some experimental results on distributed join algorithms in a local network. In Alain Pirotte and Yannis Vassiliou, editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, pages 292–304. Morgan Kaufmann, 1985.

[43] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 149–159. Morgan Kaufmann, 1986.

[44] Robin Milner. A complete axiomisation for observational congruence of finite state behaviours. *Information and Computation*, 81:227–247, 1989.

[45] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2 edition, 1999.

[46] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1999.

[47] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A Chase Too Far? In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 2000.

[48] Rachel Pottinger and Alon Levy. A Scalable Algorithm for Answering Queries Using Views. submitted for publication.

[49] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas. Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In *International Conference on Very Large Databases (VLDB)*, pages 599–610, September 1999.

[50] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979. Reprinted in *Readings in Database Systems*, Morgan-Kaufmann, 1988.

[51] Timos K. Sellis. Global query optimization. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, pages 191–205. ACM Press, 1986.

[52] Peter Sewell, Pawel T. Wojciechowski, and Benjamin C. Pierce. Location independence for mobile agents. 1999. To appear in an edited collection of papers (in Springer LNCS) from the *Workshop on Internet Programming Languages*, June 1998, Loyola University.

[53] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A New Architecture for Distributed Data. In Ahmed K. Elmagarmid and Erich Neuhold, editors, *Proceedings of the 10th International Conference on Data Engineering*, pages 54–67, Houston, TX, February 1994. IEEE Computer Society Press.

[54] Michael Stonebraker, Robert Devine, Marcel Kornacker, Witold Litwin, Avi Pfeffer, Adam Sah, and Carl Staelin. An economic paradigm for query processing and data migration in mariposa. Technical Report S2K-94-49, University of California, Berkeley.

[55] Dan Suciu. Distributed Query Evaluation on Semistructured Data. 1997.

[56] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[57] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Heterogeneous Databases and the Design of Disco. In *ICDCS*, 1996.

[58] J. Vitek and G. Castagna. A calculus of secure mobile computations.

[59] Jan Vitek and Giuseppe Castagna. Towards a Calculus of Secure Mobile Computations. Electronic commerce objects, Centre Universitaire d'Informatique, University of Geneva, July 1998.

[60] Jim White. Mobile agents white paper, 1996.

[61] Paul F. Wilms, Bruce G. Lindsay, and Patricia G. Selinger. I wish I were over there": Distributed Execution Protocols for Data Definition in R*. In David J. DeWitt and Georges Gardarin, editors, *SIGMOD'83, Proceedings of Annual Meeting, San Jose, California, May 23-26, 1983*, pages 238–242. ACM Press, 1983.