Distributed Query Optimization: Can Mobile Agents Help?

(extended abstract)*

Arnaud Sahuguet Benjamin C. Pierce Val Tannen

University of Pennsylvania

Abstract

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. The reality of the Web is different. Future distributed query optimizers will have to exploit a rich variety of information flow mechanisms (chaining, referral, proxying, etc.) and must regard the Web as both a repository of information and a computing model.

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 in combination with a powerful query-plan language, so that messaging, migration, and database operations all live in the same semantic space and interact, thus creating new opportunities for optimization.

1 Introduction

Starting with the classic dynamic programming technique for join ordering [20], the database field has developed powerful techniques for finding efficient execution plans for declaratively specified queries [6, 12]. However, applying these optimization techniques in the setting of distributed information management [17] requires a good deal of centralized knowledge.

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 [11]), 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. Another strategy is *chaining*, where a site acts as a proxy and forwards queries to the actual source.

Such mechanisms have not been considered in traditional distributed query processing because they do not seem to fit in the traditional framework. But in fact, extending the flexibility of distributed query plans should be seen as an opportunity rather than an obstacle to efficiency.

In this 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: (a) can this be done by deploying a relatively small generic infrastructure in each node, and (b) is it worthwhile? We believe that by borrowing from mobile agent technologies we can answer (a) positively, and that in turn this will enable us to build prototypes that can settle (b).

We look to mobile agent technologies for the combination of flexibility and precision needed for expressing, optimizing, and deploying queries using these new mechanisms. Mobile agent systems have been developed quite successfully in a number of domains including e-commerce, user-interfaces, knowledge management (see [3] for a review), active networks [24], and general distributed programming. 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 [16] enriched with primitives for process migration and remote communication. To this we add the primitives of the query plan language used in [9, 18] to obtain a small and semantically clean language for expressing and eval-

*A detailed description of these ideas has been submitted to the VLDB'2000 conference.

uating distributed queries.

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. By using a single language we allow the optimizer to systematically explore alternatives that exploit these mechanisms.

The rest of the paper is organized as follows. We first introduce some motivating examples of these new mechanisms before we describe an architecture to support them. We then present some related work before we offer some future directions for research.

2 New mechanisms

We now describe some new mechanisms that represent the building bricks of distributed strategies for query evaluation.

The various strategies we expose are illustrated in Figure 1.

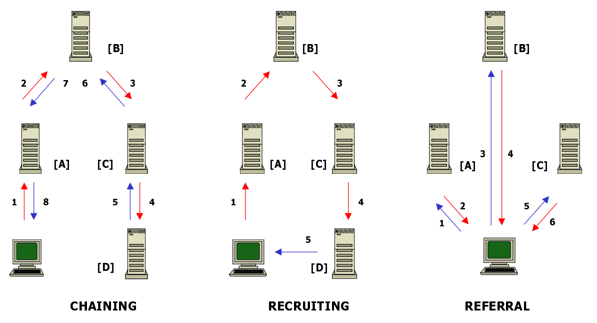

Figure 1: Information flow mechanisms.

2.1 Chaining, referral, recruiting

These mechanisms did not arise in traditional distributed databases, yet they should be familiar to our reader: referral (redirection) is part of HTTP —the protocol underlying the Web—and both referral and chaining are part of LDAP [11] (Lightweight Directory Access Protocol). Although in HTTP and LDAP these mechanisms are used for very special data and distribution models, our language can express them in full generality, for any kind of distributed queries.

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 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.

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

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 implementation is described in the tables below, where each column represents the running query processes at a given site and each row a step of process evaluation (PE), local query evaluation (LQ), or optimization (OPT). 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 meaning of the principal constructs of the language are presented in section 5.

K	A	B	step
`q@A`			Q
`new c in` `<go A do c@K!q> || ?c`			PE
`?c`	`c@K!q`		PE
`?c`	`c@K!(q'@B)`		OPT

Figure 2: Steps common to all mechanisms

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 2.

From this point on, the strategies differ. Referral relies on an optimization that migrates back to K the referral for evaluating q' at B (figure 3). For chaining (not detailed here), q' is evaluated at B, the result sent back to B then to K.

K	A	B	step
starting from the last step of Figure 2			
?c	<go K do c!(q'@B)>		OPT
?c || c!(q'@B)			PE
?c || c!(new c' in <go B do c'@K!q'> || ?c')			PE
?c || <go B do c@K!q'>			OPT
?c		c@K!q'	PE
?c		c@K!v	LQ
v			PE

Figure 3: Referral

Upon examination of these two mechanisms, a third alternative, **recruiting**, (not offered by LDAP, but available in KQML [13]) suggests itself. 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 4).

K	A	B	step
starting from the last step of Figure 2			
?c	<go B do c@K!q'>		OPT
?c		c@K!q'	PE
?c		c@K!v	LQ
v			PE

Figure 4: 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 3, we could write the first step for server A as: <go K do *value* U c!(q'@B)>. This strategy is frequent for directory services (information is partitioned).

2.2 Reducing data shipping

Query migration can also be used to capture various optimization techniques for *multi-target*, i.e. queries that need data residing on several servers. Semijoin programs [2] are an example of such a technique, where the *idea* is to ship around only necessary data.

Even more interestingly, the core idea of semijoin programs can be generalized to make use of physical access information such as join indexes, gmaps, etc., cached in convenient locations. To do so, our optimizers need to *"rewrite queries using views"* [14] where "views" is interpreted broadly to also mean cached queries and cached physical access data.

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 relations R and S and that M is a "mediator" site (see Figure 5) that needs the result of the following "generalized" join: $R \bowtie S$.

We could use a semijoin-like technique to actually compute the join. There exist several plans to evaluate it and we list three main ones, and give for each its corresponding query plan process for an agent at site M:

Plan 1: M sends $\Pi_1(JI_{RS})$ to A, 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 to 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 5).

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.

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

2.3 Other important mechanisms

There are two other mechanisms (orthogonal to the ones considered previously)that we briefly mention

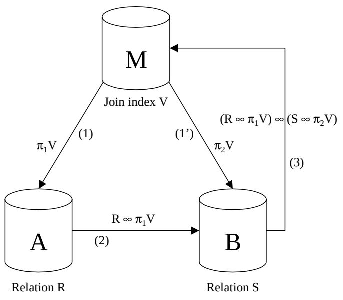

Figure 5: Plan 2, where $V = JI_{RS}$

(they are described thoroughly in the full version of the paper): subscription and leasing.

The concept of subscription has been around for a while: just look at newspapers. When a query q is recurring, we may decide to install a subscription at site Publisher for q.

Subscriptions can also be used to store remotely (*outsource*) the result of local computations. The remote site could send the data back to a housekeeping process that will cache it locally. Servers can also establish *mirrors* to bring information closer to clients.

In order to manage such mechanisms – that might be greedy in terms of resources –, *leasing* is a good candidate. A lease is a contract that grants use of access to a resource. The interesting aspects of leasing are that: (1) it is an intuitive concept; (2) it fits both client and server concerns; and (3) it is being already used for distributed architectures[1].

3 Architecture

The infrastructure available at each node is presented in Figure 6. At the top level, each node's process evaluator runs a collection of query processes in parallel, which may grow as incoming query processes migrate in or shrink as query processes terminate or migrate elsewhere. The process evaluator interacts with the **migration manager** who handles both incoming and outgoing process migrations 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**.

The infrastructure also contains a **continuous optimizer** [7] that runs as a separate thread. It is responsible for identifying some interesting local *patterns*

[1]It is already heavily used by protocols like DHCP or more recently JINI.

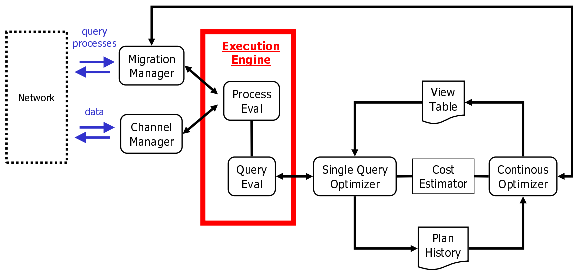

Figure 6: The architecture at each node

and for installing local/remote caches and subscriptions useful in future optimizations.

The continuous optimizer periodically examines a **history** of plans chosen by the single query optimizer. By keeping track of historical and structural information (similar to [21, 1, 7]), using techniques such as common subexpression identification, and using various statistics in order to make decisions about cost and amortized cost, the continuous optimizer can decide when a certain expression is worthy of attention.

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

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 its 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[2].

4 Related work

Distributed Query Processing: The state of the art in distributed query processing is nicely presented in the recent survey [12] by Kossmann while the classic work in distributed databases is covered in [17]. The traditional optimization algorithm used to generate the best query plan is an extension of the dynamic

[2]Note that, as usual, several rewritings may be possible and the optimizer will choose between plans based on cost.

programming algorithm (in System R* [20]) and its *textbook* generalization is described in [12].

In contrast with the relatively centralized approach of System R*, Mariposa [23] offers a decentralized solution where every node is governed by economic motivations. Optimization decisions are based on bids and offers with negotiated prices and given budgets. Although decentralized, the approach of Mariposa and ours are complementary. Moreover it does not focus on information flow mechanisms, that is of interest to us.

More recently the use of continuous queries [15, 7] has pointed out the need for the need for continuous optimization. The importance of cost models has been re-emphasized for mediator-based architectures in [19]. The system that is closest to what we try to capture is ObjectGlobe [10].

Query rewritings using views and caches: Many of the optimizations shown in our this use query rewritings with materialized views and caches. Previous work on using caches include [1]. There has been much work on rewriting with views, most recently [9, 18], see the nice survey [14] by Levy.

Software Agents: A survey of recent developments in agent-based technologies appears in [3]; [8] discusses more specifically the benefits of mobile agents.

Mobile Processes: Previous work on process calculi has mainly focus on communications and exchange of small messages. Cardelli's recent work on service combinators [5] and mobile ambients [4] considers more general computations that address database related concerns.

5 Future Directions

In this extended abstract, we have proposed a flexible framework for representing, optimizing, and evaluating distributed queries, combining the strengths of distributed database and mobile agent technologies. We are starting to build a prototype to test these ideas. An interesting trade-off here is whether the start with a distributed query system and add process behavior or with a mobile agent system and build a query layer on top. Still, many interesting questions need to be answered before we can assess whether this approach is feasible and worthwhile.

Do the process primitives discussed here offer enough expressiveness? Or perhaps they are too powerful and do not admit sufficiently efficient implementations? If so, do we need to restrict them via a type system?

The most important questions pertain to adapting traditional distributed database cost models to our richer framework. Where do we cost referral/recruiting queries? How do we organize sharing/updating of cost information between nodes? Can the same (extended) cost model be used by both the *Single* and the *Continuous* query optimizer?

Finally, we are hoping to find additional interactions between process and database primitives that can be exploited in optimization. An important issue here is how to define the correctness of optimizations when caches and subscriptions can supply fresh or stale information.

References

[1] S. Adali et al. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD*, pages 137 – 148, 1996.

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

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

[4] Luca Cardelli. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, chapter Abstractions for Mobile Computation. Springer, 1999.

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

[6] Surajit Chaudhuri. An overview of query optimization in relational systems. In *SIGMOD*. ACM Press, 1998.

[7] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*. ACM Press, 2000.

[8] 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.

[9] Alin Deutsch, Lucian Popa, and Val Tannen. Physical Data Independence, Constraints and Optimization with Universal Plans. In *VLDB*, September 1999.

[10] Reinhard Braumandl et al. ObjectGlobe: Ubiquitous Query Processing on the Internet. Technical report, Universität Passau, 1999.

[11] Tim Howes et al. *Understanding and Deploying LDAP Directory Services*. MacMillan, Jan 1999.

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

[13] Yannis Labrou and Tim Finin. A Proposal for a new KQML Specification. Technical Report TR CS-97-03, University of Maryland, February 1997.

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

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

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

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

[18] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A Chase Too Far? In *SIGMOD*, May 2000.

[19] 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 *VLDB*, September 1999.

[20] P. G. Selinger et al. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[21] Timos K. Sellis. Global query optimization. In Carlo Zaniolo, editor, *SIGMOD*. ACM Press, 1986.

[22] Peter Sewell, Pawel T. Wojciechowski, and Benjamin C. Pierce. Location independence for mobile agents. 1999. To appear.

[23] M. Stonebraker and al. Mariposa: A New Architecture for Distributed Data. In Ahmed K. Elmagarmid and Erich Neuhold, editors, *ICDE*, Houston, TX, February 1994. IEEE Computer Society Press.

[24] D. L. Tennenhouse et al. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1), Jan. 1997.

Our query process language

Our language can be described as a combination of primitives from the *nomadic pi-calculus* [22] with the primitives of a language for expressing database query plans[9]. We focus here on the aspects of the language dealing with distributed query plans, omitting on purpose the database primitives.

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 syntax of our calculus is divided into two parts: 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.

Most of the process constructors are familiar from the pi-calculus: the inert process **0** has no behavior; the parallel composition $P \,||\, P'$ 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 $P \,||\, e$ 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> || ?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.

Processes	
0	inert process
P || P'	parallel composition
$*$ P	replicated process
new c in P	channel creation
c!e	local output (send e on channel c)
c@s!e	output (send e on channel c at site s)
c?$x.P$	input (receive x on channel c)
<go s do P>	migration
Expressions	
s	site name
R	abstract resource name
v	basic value
x	variable
P || e	process spawning
new c in e	channel creation
e@s	remote evaluation of e at site s
?c	channel
(omitted)	database primitives

Figure 7: Our language