Efficient Querying of Distributed Resources in Mediator Systems

Ioana Manolescu[1] Luc Bouganim[1, 2] Françoise Fabret[1] Eric Simon[1]

[1] INRIA Rocquencourt, France. <Firstname.Lastname>@inria.fr

[2] PRISM Laboratory, 78035 Versailles – France. <Firstname.Lastname>@prism.uvsq.fr

Abstract: This work investigates the integration of heterogeneous resources, such as data and programs, in a fully distributed peer-to-peer mediation architecture. The challenge in making such a system succeed at a large scale is twofold. First, we need a simple concept for modeling resources. Second, we need efficient operators for distributed query execution, capable of handling well costly computations and large data transfers. To model heterogeneous resources, we use the model of table with binding patterns. To exploit a resource with restricted binding patterns, we propose an efficient BindJoin operator, optimized for minimizing large data transfers and costly computations. Furthermore, the proposed BindJoin operator delivers most of its output in the early stages of the execution, which is an important asset in a system meant for human interaction. Our experimental evaluation validates the proposed BindJoin algorithm on queries involving expensive programs.

1 Introduction

There is a growing interest in the scientific community to allow disparate groups of users (a.k.a. virtual organizations) to share resources consisting of both data collections and programs. This vision is best reflected by recent initiatives such as the "Grid Computing" infrastructure, that aims at constructing a "meta computer": a large scale, distributed computing environment, providing transparent access to highly heterogeneous resources. A frequent domain of applications is that of international scientific cooperation, where remote laboratories share their data and programs.

LeSelect [18] is a mediator system developed at INRIA, which allows the users to publish their resources (data and functions – corresponding to programs) so they can be transparently accessed. In LeSelect, several distributed mediators (*a.k.a. servers*) cooperate in a peer-to-peer fashion to allow large-scale integration of data and functions. LeSelect is currently used in many earth-science cooperation projects like Thetis (coastal zone management) [7], Decair (air quality models) [7] and SIMBio (bio-corrosion monitoring) [8].

As an example of such projects, consider the following distributed scientific application. On site S_1, satellite images have been processed into a map of the ozone cover of the French territory. On site S_2, a survey of the traffic in the same area resulted in a set of records corresponding to the days when traffic was particularly

intense. On site S_3, a function *OzoneLevels→{level}* computes the set of distinct ozone density levels found in an ozone cover image. If a user on site S_4 wants to match heavy traffic data from S_2 with the days with low ozone levels in images found on S_1, the OzoneLevels function on S_3 needs to be invoked on images from S_1. In this example, answering the user query will *necessitate manipulation of large data* (e.g., images) and potentially *expensive function invocations*.
This work investigates data and program integration in a fully distributed peer-to-peer mediation architecture. The challenge in making such a system succeed at a large scale is twofold: First, a publication model needs to be chosen for representing the published data (e.g. satellite images and high traffic data), and functions (e.g. *OzoneLevels*). This model should be simple, since the publisher is not supposed to be a computer scientist. Second, we need efficient operators for distributed query execution, capable of handling costly computations and/or large data transfers. *Costly computations* arise with (i) expensive functions [3], [5], [14], [15] like *OzoneLevels*, reflecting a domain-specific knowledge, and performing complex mathematic computations, (ii) Web accesses [11], and (iii) correlated subqueries [14]. *Large data transfers* (e.g. satellite images) are necessary when functions only run on their native site, and cannot be shipped through the network. This is the case of the major part of scientific applications, in which the programs were written in isolation (without concern for their future integration in a larger setting) [3].

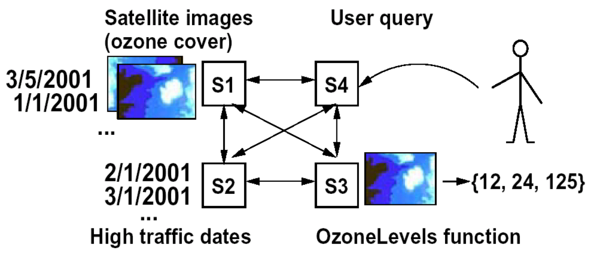

Fig. 1. Sample configuration and query on distributed data and functions.

Our general objective is to minimize the publisher task while providing the best performance for queries involving expensive functions or large data. Our approach can be summarized in three points:

1. We base the modeling of our resources on the concept of table with binding patterns (introduced in [21] for a different purpose), and the associated logical BindJoin operator. The logical *BindJoin* operator is a variant of the relational join operator to access tables with binding patterns. Binding patterns can naturally be used to model functions. We propose to use the same modelization for table with large binary objects (thereafter called *blobs*).
2. We analyze the impact of expensive functions and blobs on the design of the BindJoin operator and on its integration in the query execution plan (*QEP*). First, the *total work* (TW) and *response time* (RT) of queries including blob transfers and expensive function calls must be reduced, by employing caching and asynchronism techniques. Second, we show the importance of the *early tuple output rate* (ER), that is, we are interested by a QEP that returns as much results as possible *early on* during query execution. Indeed, in a data integration setting like the one above, queries are asked by human users that wait for the result in

front of their stations. Moreover, in many cases, several exploratory queries are asked before the user identifies the data segments he/she is really interested in. Therefore, users typically want to see at least part of the result as soon as possible; the same query pattern also appears in online decisional applications. *We therefore identify a good ER as an important performance requirement for the execution of distributed queries like the one in our example.*

3. We propose to include every optimization (caching, asynchronism and ER specific optimizations) in the BindJoin operator. While this approach complicates significantly the design and implementation of the BindJoin operator, it reduces to the minimum the publisher task. The publisher must only provide a call-based interface of the form *callFunction(arguments)* for functions and *readBlob(blobName)* for accessing blobs.

Our contribution, following our objectives of publisher task minimization and performance maximization, is twofold:

- First, we show how the model of tables with binding patterns can be used to uniformly model data sources including functions as well as blobs and explain why that modelization provides benefits similar to those of semi-joins, without their drawbacks.
- Second, we propose, implement and assess the performance of a highly efficient BindJoin operator, improving over the state-of-the-art algorithms by having much better ER properties.

This paper is organized as follows. In section 2, we show how the paradigm of tables with binding patterns can be used to describe data (including blobs) and functions and present the associated logical BindJoin operator. In section 3, we analyze the impact of expensive functions and large binary objects on the design of the BindJoin operator and on its integration in the QEP. We describe the optimizations that our BindJoin operator should include in order to provide a good ER behavior. Section 4 presents the associated physical operators and the algorithms used in case of limited memory. Section 5 demonstrates the good ER of our BindJoin operator through a series of experiments. Related work is presented in section 6. We conclude in section 7.

2 Modeling and Querying Resources

In this section, we first describe the concept of table with binding patterns and the associated BindJoin and BindAccess operators. Then we explain how we use these concepts to model several classes of resources. Finally, we compare our approach with the classical semi-join technique.

2.1 The Concept of Table with Binding Patterns

Binding patterns [21] can be attached to a relational table to describe the restrictions that we encounter in accessing it. These restrictions may stem from confidentiality or performance reasons, or simply reflect the restricted nature of the resource: for example, web sources can be represented as virtual tables with a binding pattern where input parameters have to be provided in order to obtain the results.

A binding pattern *bp* for a table $R(X_1, X_2 \ldots X_n)$, is a partial mapping from $\{X_1, X_2 \ldots X_n\}$ to the alphabet $\{b,f\}$. The meaning of a binding pattern is the following: those X_i mapped to *b* are *bound*, i.e., their values must be supplied in order to obtain information from *R*, while values of attributes mapped to *f* are *free* and can be obtained from the data source, as soon as values for all *b* attributes are supplied. If a binding pattern maps all attributes of *R* to *f*, then tuples of *R* can be obtained without any restriction (just like a usual Scan). For example, if it was possible to obtain the full data contained in a web Yellow Pages source of the form *YP(name, address, phoneNo)*, this would be indicated by a $YP(name^f\ address^f\ phoneNo^f)$ binding pattern. On the contrary, $YP(name^b\ address^f\ phoneNo^f)$ specifies that the values of the name attribute have to be supplied in order to obtain addresses and phone numbers from *YP*.

2.2 The BindJoin and BindAccess operators

The presence of access restrictions, formalized using binding patterns, makes the regular set of relational operators insufficient in order to answer queries.
The BindJoin operator: The standard relational join operator does not capture well the semantics of combining two tables, if at least one has a restricted access pattern. To illustrate, consider a QEP fragment that joins the $YP(name^b\ address^f\ phoneNo^f)$ with an $Employee(name^f\ salary^f)$ table on their *name* field. Due to the commutativity of the standard join operator, we might try to write this fragment as $Employee \bowtie_{name} YP$, or as $YP \bowtie_{name} Employee$.

However, the first variant is valid, while the second one is not: as described by the binding pattern of *YP*, we cannot start by accessing it, before supplying some bindings for its *name* field. Thus, we adopt a variant of the relational join operator, the BindJoin logical operator (denoted $\overrightarrow{\bowtie}$ and used, e.g., in [9], [11], also known as dependent join, theta semi-join, functional join etc.) to capture this asymmetric behavior: the right-hand child of a BindJoin operator cannot be executed on its own, since it depends on the join values passed across the BindJoin operator.
The BindAccess operator: Due to the semantics of the binding pattern $YP(name^b\ phoneNo^f)$, we cannot perform a scan on the *YP* table. Instead, we have to supply some values for the *name* attribute in order to get *YP* tuples. Furthermore, the set of tuples that we can extract from *YP*, following this binding pattern, depends on the values that we supply for *name* (in contrast, the result of a Scan is always the same). To capture the special semantics of a restricted access, we use a special *BindAccess* operator. As an intuition, think of the BindAccess as being a "parameterized Scan", where the parameters are the values supplied for the bound attributes.

The formal semantics of BindAccess and BindJoin can be specified as follows. Consider two tables *R(X,Y)* and *S(U,V)*, where *X*, *Y*, *U* and *V* are pairwise disjoint sets of variables. Let $R(X^bY^f)$ and $S(U^bV^f)$ be binding patterns of *R* and *S*, and c be a set of values for *X*. Then, denoting the BindAccess by *BA*, we have: $BA(R(X^b\ Y^f), c) = s_{X î c}\ R(X, Y, Z)$.

Furthermore, the BindJoin of Scan($S(U^f\ V^f)$) and BA($R(X^b\ Y^f)$) has the following semantics: $\mathrm{Scan}(S(U^f\ V^f))\ \overrightarrow{\bowtie}_{U=X}\ \mathrm{BA}(R(X^bY^f)) = \{(u,v,x,y) \mid (u,v) \in S \wedge (x,y) \in R \wedge u=x\}$

While this formula shows that the set of tuples returned by a BindJoin is similar to the one returned by a regular join, keep in mind that the similarity stops here. Indeed, neither the optimization techniques for join queries nor the join operator algorithms can be directly reused for two distinct reasons: First, the asymmetric character of the BindJoin greatly impacts the optimizer's search space [9]. Second, BindJoins often involve costly computations, which have deep implications on the operator algorithm.

2.3 Modeling Resources Using Tables with Binding Patterns

A function is naturally represented as a table, whose binding pattern distinguishes the attributes that correspond to function arguments (which need to be supplied in the query) from the function results. For example, the binding pattern of the *OzoneLevels* function is *OzoneLevels(*img^b $level^f$*)*.

We propose a specific modeling for data resources involving blobs in order to optimize their transfer through the network. This modeling imposes some requirements on the publication of a table with blobs: for every blob attribute *B* of table *R*, *R* must also contain a small-sized attribute *Bid* that determines the value of *B*, i.e., such that the functional dependency *Bid* ®*B* holds. Furthermore, among the binding patterns of *R*, we require the presence of *R(*Bid^b B^f*)*. As an example, assume that the *id* field in the *SatImg* table determines the *img* field. Then, the binding pattern set for *SatImg* must at least contain *SatImg(*id^b img^f*)*. BlobIDs are system-generated in the case of published data residing in a DBMS. In a simpler setting, a blob is usually stored in a separate file, whose complete name (i.e. host/path) is used as a blobID.

The purpose of requiring a blob identifier is to reduce the blob transfers to a minimum. First, we transfer identifiers instead of blobs, whenever the blobs themselves are not needed; blob identifiers enable us to transport *only once only the necessary* blobs, as follows. First, when a set of blob transfers are necessary, by comparing the identifiers of two blobs, we can decide whether or not they are the same, and if yes, make the transfer only once. Second, if in a given QEP several selections and/or joins apply on the tuples containing blob identifiers, we avoid transferring blobs (for some further processing) once we know that some of them were not necessary (eliminated by joins or selections).

At this point, the similarity between calling a function and getting a blob becomes evident. Both are modeled by an access to a resource following a restricted binding pattern. Both are expensive operations, suggesting the usage of a cache for function results and blobs. Thus, in the following, we will no longer make the distinction between the two: they are accessed using the same operators (section 3), and the same techniques apply.

In the table below, at left, we show the tables with binding patterns and the user query corresponding to the scientific data integration scenario that we presented:

S_1:	SatImg(id, date, img);	Select	i.img, i.date, h.date, ol(i.img)
	SatImg(id^f $date^f$), SatImg(id^b img^f)	From	S_1:SatImg i, S_2:HighTraffic h,
S_2:	HighTraffic(date); HighTraffic($date^f$)		S_3:OzoneLevels ol
S_3:	OzoneLevels(img, level);	Where	(i.date>=h.date) and (i.date<
	OzoneLevels($img^b$$level^f$)		h.date+3) and (ol(i.img) < 45)

Figure 2 shows one possible QEP for evaluating this query; we circled together operators successively executed on the same site, while blob transfer edges are shown in thick lines. The bottom join operator correlates the dates from the *SatImg* and *HighTraffic* sources. The join may pair some tuples from *SatImg* with several tuples from *HighTraffic*; other *SatImg* tuples are eliminated by the join.

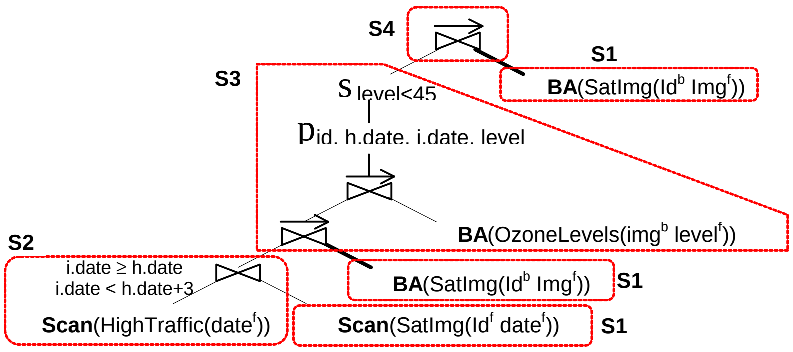
S4
S1
BA(SatImg(Id^b Img^f))
S3
S level<45
Pid, h.date, i.date, level
BA(OzoneLevels(img^b level^f))
S2
i.date ≥ h.date
i.date < h.date+3
BA(SatImg(Id^b Img^f))
S1
Scan(HighTraffic(date^f))
Scan(SatImg(Id^f date^f))
S1

Fig. 2. QEP for our sample query, using BindJoins.

Then, images that survived the join predicate are fetched (and cached locally) on site S_3 by a BindJoin, and a second BindJoin invokes *OzoneLevels* on them. We then project out the images, and apply the selection on the result of the *OzoneLevels* function; this selection eliminates part of the tuples, and thus some image identifiers. The last BindJoin retrieves on site S_4 only the images corresponding to identifiers that survived both the join and selection predicate. Note that we perform two BindJoins with the table *SatImg(id*^{b}*blob*^{f}*)*, corresponding to the two unavoidable transfers: from S_1 to S_3, and from S_1 to S_4. In both cases, we only transfer the images that are actually needed on the destination sites.

A well-known method for achieving the performance gains of the QEP in figure 2 consists of optimizing with semi-joins, introduced in [2]. A QEP using semi-joins equivalent, in terms of blob transfers and function calls, to the one in figure 2, is presented in [19]. Such a QEP, and in general, a semi-join based solution, may incur a high processing overhead, and hinder pipelined execution (see section 3). Finally, optimization with semi-joins is quite complex. Instead, we adopted a lighter approach: assuming only that our BindJoin operator uses a cache, our modeling provides the advantages of semi-joins, without their drawbacks, for the specific redundant operations that we consider.

3 Designing an Efficient BindJoin Operator

In this section, we describe a physical BindJoin providing for efficient execution of distributed queries involving blobs and/or expensive functions.

Traditionally, query processing performance is assessed using three measures [17]: *total work* (TW), including all processing and data transfer costs, *response time* (RT), measuring the time elapsed until the query result has been completely received on the query site, and *time to the first tuple* (FT), accounting only for the time elapsed until the result starts arriving. Note that in fact, the FT metric typically accounts for the early tuple output rate (also called "low latency" in [16]): the property of a QEP to produce as much results as possible early on during query execution. Rather than FT, we use the more expressive "*early output rate*" (ER) term to designate this property. As mentioned in section 1, ER is an important performance goal in the context that we consider and is thus more detailed in the next sections.

We propose a new physical BindJoin that helps reduce the TW and RT of distributed queries with blobs and expensive functions; the main innovation of this operator is its good ER, significantly improved over the state-of-the-art algorithms (as we show in section 5). However, an efficient operator must integrate well with, and take advantage of, standard query execution and optimization techniques.

This section demonstrates both our specific contribution in the design of the operator and its good integration with the existing techniques. We first introduce some simple notations to support our exposition. We then discuss both *local optimizations* (i.e., applying at the operator level) and *global optimizations* (i.e., that must be decided at the QEP level). Sections 3.1 and 3.2 show which of the existing optimizations for reducing TW respectively RT can be combined with our BindJoin. Section 3.3 is specific to our contribution, as it presents the special techniques we employ to provide our BindJoin operator a good ER behavior (local optimization).

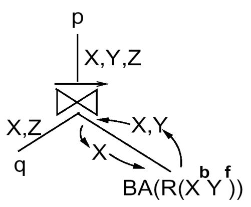

Fig. 3. The BindJoin and BindAccess operators.

We use capital letters, e.g., *X, Y,…* , for attribute names, and corresponding lower case letters, like *x, y,…* for attribute values. We consider a BindJoin operator which receives from its left-hand child operator, denoted *q* in figure 3, tuples of the form *(X, Z)*, and uses the *X* arguments to access the resource *R*, following its binding pattern $R(X^bY^f)$. The BindAccess operator returns *(x, y)* tuples for each *x*, and the BindJoin concatenates these tuples with the *z* attribute, not needed to access R (figure 3).

3.1 Reducing Total Work (TW)

Local optimization: Reducing total work at the level of the BindJoin operator can be done by using *caching techniques* as suggested in [14]. Caching is profitable for function evaluation and blob transfers as soon as (i) retrieving a function result, respectively blob from the cache is less expensive that computing the results, respectively transferring the blob and (ii) there are duplicate values in the input table. Consider a tuple *(x,z)* coming from *q*. If the y values associated with *x* are stored in a cache, the tuple enriched with *y* can be output directly, short-circuiting the BindAccess operator. Otherwise, a (potentially expensive) access to the restricted resource *R* is made with *x* as an argument. *We decide therefore to include a cache in our BindJoin operator.* Several caching techniques have been proposed in, e.g., [14]

Global optimization: Global optimization of queries with expensive functions has been extensively studied [6, 7, 17, 22, 26]. Due to our modeling based on binding patterns, these results apply for function calls as well as for blob transfers. Query optimization in the context of tables with binding patterns is addressed in [9].

3.2 Reducing Response Time (RT)

Running several tasks in parallel may reduce response time as soon as that tasks consume distinct resources. This simple principle applies at the local level, i.e. intra-operator parallelism, and at the global level, i.e. inter-operator parallelism.
Local optimization: Our interest in *intra-operator parallelism* is restricted to the costlier operations, namely function invocation and blob transfer. Performing in parallel several function calls or blob transfers allows to fully exploit the query processing - respectively the network transfer – capacity [11]. This could be useful when several processors are available in order to compute a function, or when the same blobs exist in several source sites. Our physical BindJoin operator is designed to include such intra-operator parallelism; due to space limitations, this aspect is relegated to the extended version of this article [19].
Global optimization: *Inter-operator parallelism* is interesting in a distributed QEP where BindJoins run on distinct sites, but also within a single site if the BindJoins consume different resources. Depending on whether a producer-consumer dependency exists between two BindJoins, they run in pipeline, respectively, independent parallelism. *De-synchronizing the BindJoin from its neighbor operators* (*p* and *q* in figure 3) allows such inter-operator parallelism. As a consequence, during the execution, a BindJoin may accumulate tuples waiting to be processed, if *q* outputs tuples faster than the BindJoin can consume them. Conversely, it may also accumulate result tuples, input for a slower parent operator.

3.3 Improving ER Behavior

The BindJoin informally described so far is pipelined, and treats tuples from *q* in the order of their arrival. At the beginning of the execution of a query, the cache is empty, and most tuples take a long time to get through the BindJoin, corresponding to an access to the restricted resource. During the execution, the cache is progressively filled, and probably tuples are output at a faster rate towards the end. The early tuple output rate is likely to be small, and most tuples are output in the last stages of the execution.
Local optimization: If the operator denoted *q* in figure 3 has a good ER behavior, several tuples output by *q* may accumulate in the BindJoin, waiting to be processed. These waiting tuples can be split in two categories: those for which the result has already been computed and is in the cache; and those for which we need to access the restricted resource in order to get the result. If waiting tuples are processed in the arrival order, a tuple *t* from the first category can only be output after processing all tuples *r* from the second category, such that *r* arrived before *t*. However, tuples like *t* and *r* could very well be processed in parallel, since they have distinct requirements: to output *t* it is enough to access the cache, while for *r* we need to access the restricted resource.

Finally, when selecting the next *x* value to be processed, we may choose the most advantageous *x* value with respect to the ER behavior: the *x* value corresponding to the currently largest number of *(x, z)* tuples waiting to be processed (we term this value the *most frequent*). The advantage of choosing the most frequent *x* value is obvious: when the restricted access to *R* using *x* is finished, a large number of *(x, y, z)* tuples corresponding to *x* can be sent simultaneously to the output, improving even more the ER of the BindJoin.

Global optimization: With respect to the global ER of a QEP, we make the following remarks. (i) The delay incurred by a single blocking operator in a QEP is a direct loss for the QEP's ER; therefore, when ER is a concern, pipelined operators are required. (ii) The ER of an operator is a measure relative to the rate of its input operators: of course, if the child of operator *op* is blocking and does not output any tuple in the first minute of query execution, there is little that *op* can do to improve the query's ER. As a consequence, *op* has good ER behavior if its ER is *as good as it can be* with respect to the ER of its input operators. (iii) The good ER behavior of several operators in a QEP re-enforce each other: if a leaf operator (e.g. Scan) has an important ER, and if its parent is able to exploit it, then ER of the QEP rooted at the parent is important, too. Thus, *an important early tuple rate propagates upward in the QEP*, up to the topmost operator, whose ER directly benefits the user. In this paper we propose a BindJoin operator optimized for ER; relational operators for reordering [22], join [13], [16], [26] or online aggregation [13] with good ER behavior have already been described, and should be used together with our BindJoin.

While a global optimization strategy [6], [7], [9], [15], [20], [24] will order BindJoins after selections (to reduce the number of tuples on which an expensive operation is performed), it may place BindJoins *after join operators*, as soon as a *cache* is used in the BindJoin. Indeed, joins can produce many tuples but *will never produce new values*, and in particular, values for the BindJoin’s inputs. This remark increases the interest of the ER optimizations we propose, since duplicates are more likely to appear in the input of a BindJoin as a consequence of a join operator.

Finally, one should note that ER optimization may also reduce response time by absorbing synchronization problems between several BindJoins (see section 5.3).

4 Operator Implementation

This section describes the physical operators that implement the BindJoin and BindAccess operators. The internal data structure used for these optimizations is depicted in section 4.2. Section 4.3 explains how our BindJoin algorithm deals with limited memory execution environments.

4.1 Physical Operators for BindJoin and BindAccess

We now describe the physical operators for the BindJoin and BindAccess, implemented as iterators, following [12]. The API of an iterator consists of an initialization *open()* method, a *next()* method producing one tuple at a time, and a *close()* method to release resources and terminate.

Figure 4 depicts our proposed decomposition of the logical BindJoin and BindAccess operators into physical operators, implementing the techniques described in the previous section. A single data structure, belonging to the BindJoin, is used to hold (a) tuples waiting to be processed, (b) a result cache and (c) processed tuples waiting to be output.

Physical operators for the BindJoin: We decompose the BindJoin in four physical operators, termed *BJStore*, *BJCompute*, *BJGetBindings* and *BJGet* in figure 4.

1. BJStore retrieves *(x, z)* tuples from *q* and checks *x* against the cache. If *x* is present in the cache, then BJStore inserts the tuple in the set of processed tuples; otherwise, in the set of waiting tuples.
2. On a *next* call from the BindAccess, BJGetBindings chooses the next *x* value to be processed and returns it.
3. BJCompute retrieves *(x, y)* tuples from the BindAccess, and updates the cache and the set of processed tuples accordingly.
4. Finally, BJGet answers *next()* calls from *p*, the operator above the BindJoin in the QEP, returning an *(x,y,z)* tuple to *p*. This tuple is erased from the set of tuples waiting to be output.

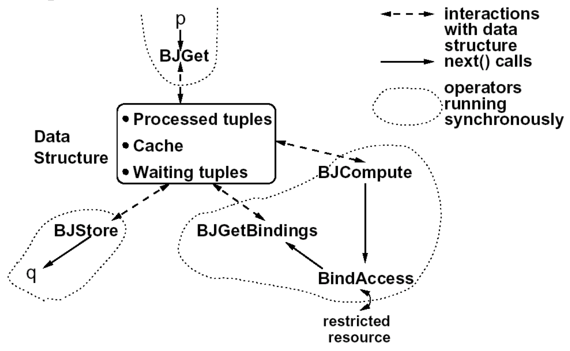

Fig. 4. Physical operators for a BindJoin and a BindAccess on the same site

Physical BindAccess operator: We use a single physical BindAccess operator, as shown in figure 4, to implement the logical BindAccess. This operator obtains *x* values from BJGetBindings, and performs a call to the restricted resource providing *x* as argument. On a *next()* call from BJCompute, BindAccess returns a *(x,y)* tuple. Thus, the access to the resource is encapsulated within BindAccess; this operator is generic and can be provided by the integration system, making the publication process easy for the resource owner. The only thing required to "plug" a BindAccess on a given restricted resource is a call interface to that resource (e.g. *callFunction(*arg_1*,*arg_2*,…,*arg_k*)* for functions and *readBlob(blobID, startOffset, endOffset, memBuffer)* for accessing blobs).

Operator synchronization: In figure 4, we represented the case when the BindJoin and the BindAccess operators run on the same site. In this case, BJCompute, BindAccess and BJGetBindings run synchronously, since there is no gain in parallelizing these operators within a single site. However, note that in parallel with these three operators run, on one hand, *p* and BJGet, and on the other hand, *q* and BJStore. Decoupling in such a way the execution of the BindJoin-BindAccess pair from the rest of the QEP allows for inter-operator parallelism.

4.2 Organization of the BindJoin's internal data structure

The data structure holding the data internal to the BindJoin is basically organized as a hash table. Every hash bucket contains a set of cells; one cell corresponds to a given value for *x*, the argument value for accessing the restricted resource. Within one bucket, cells are organized in a linked list. Two extra data structures are maintained among the cells in the hash table. First, a doubly-linked frequency list connects cells corresponding to *x* values not yet in the cache, in the order of their frequency. Second, we also keep a processed tuple set, containing the cells for which *x* is already in the cache, and besides, there currently are some *(x,y,z)* tuples produced and not output yet. More details on the data structure and its API are provided in [19].

4.3 BindJoin behavior in the presence of limited memory

The techniques presented so far assume that the data structure holds in memory, which may not be the case. Note that if *X*, *Y* or *Z* are blobs, their storage is delegated to a special BlobManager component [19], which flushes them to disk; in this case, the data structure will contain the blobID, not the blob. Since blob transfers are achieved by BindJoins (with cache), *a blob is never transmitted twice by the same operator*, and thus will not be repeatedly loaded in memory and flushed to disk.

There is, however, a risk of overflow even if X, Y and X are not blob attributes. In this case, our goal is to conserve as much as possible its good ER properties. To that purpose, we attempt to keep in memory all *(X,Y)* pairs, so that we can process in parallel tuples with new values and those for which the results are in the cache.

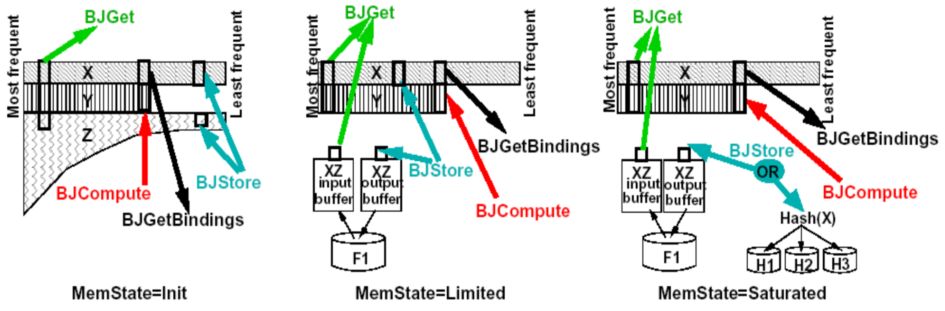

Fig. 5. The BindJoin behavior in the presence of memory limitations.

The BindJoin's memory consumption due to the internal data structure has a continuously increasing component, the *(X,Y)* cache, and a variable component, due to *(X,Z)* tuples waiting to be processed, or *(X,Y,Z)* tuples waiting to be output. We distinguish four execution phases, corresponding to four memory states. (i) In the Init phase, there is enough memory for all the data structure. (ii) In the Limited phase, all the *(X,Y)* pairs produced so far still hold in memory, but there is no place to store the *Z* values. (iii) In the Saturated phase, there is no place left to produce new *(X,Y)* pairs, so some data has to be temporarily stored to disk. (iv) Finally, during the Cleanup phase, data previously flushed to disk is processed.

The data structure and the behavior of the BindJoin's physical operators are depicted in figure 5. We represent the three types of information stored in the data structure (*X*, *Y*, and *Z* values) in separate areas. Graphically, the *Y* and *Z* values appearing “under” a given *X* value are associated to it, in the sense that *(X,Z)* tuples corresponding to these values were received from *q* and *Y* is the value obtained by accessing the resource with the argument *X*. From left to right in the data structure, there are less and less *Z* values for a given *X* value. The thick arrows designate the tuples inserted/extracted by the physical operators from the data structure.

The Init phase: At the beginning of the execution, BJStore inserts *(X,Z)* tuples, BJGetBindings picks the most frequent *X* not yet processed. BJCompute inserts in the data structure the *Y* results (when they are available) for this *X*, while BJGet extracts *(X,Y,Z)* tuples. When memory runs out, we enter the Limited phase.

The Limited phase: In this phase, we first flush to disk all *(X,Z)* pairs obtained so far (whether *Y* has been computed or not), in the decreasing order of *X* frequency. Whenever it obtains a new (*X,Z*) tuple, BJStore inserts *X* in the data structure, and sends *(X,Z)* to the "*XZ* output buffer", to be written to the temporary FIFO file F_1. Note that we need to store *(X,Z)* pairs, not just *Z* values, in order to be able to re-compute the tuples. BJGet outputs tuples from a buffer of data sequentially read from F_1; the first such buffer to be brought in memory corresponds to the most frequent *X* value, for which the result has probably already been computed. For each *(X,Z)* tuple read, if the result is already in the memory cache, BJGet outputs the *(X,Y,Z)* tuple; otherwise, its waits for the result to become available. When the available memory is insufficient for storing the *(X,Y)* pairs, we enter the Saturated phase.

The Saturated phase: In this phase, whenever BJStore obtains a new *(X,Z)* tuple, if *X* is already in memory, its corresponding frequency counter is updated, and *(X,Z)* is sent to F_1. Otherwise, we apply the hashing function *H(X)* to distribute the *(X,Z)* tuple in the disk buckets noted H_1, H_2 and H_3 in figure 5 at right. This phase ends when BJStore encounters *EOF*.

The Cleanup phase: At this point, BJGetBindings reads F_1 page by page, the corresponding *Y* results are taken from the cache or computed, and the *(X,Y,Z)* tuples output. When F_1 is finished, the current *(X,Y)* cache can be completely discarded, since no tuple on disk has an *X* value among those in the cache. Then, the partitions made by the function *H* are loaded in memory one by one, and their tuples are processed as in the Init phase. To that purpose, the function *H* is chosen so that each partition fits in memory, by a technique similar to the one proposed in [14].

The BindJoin’s behavior in the case of limited memory incurs a minimal overhead; indeed, no tuple is written to disk more than once during the processing.

5 Experimental evaluation of the BindJoin ER

This section compares the ER of our BindJoin operator with that provided by the state-of-the-art algorithms for handling expensive functions.

5.1 Experimental platform

We experimented with several algorithms, QEPs, data distribution parameters, data delivery rate and tuple orderings. For space reasons, we only include here the most significant ones, and comment on some others. More results are described in [19].

BindJoin algorithms: We compare our BindJoin physical operator (denoted ERBJ) with two algorithms previously proposed for handling expensive functions [5], [14]. The simplest one uses a *hash-based (memoization) cache* and will be denoted HBBJ. The second one is *sort-based* (SBBJ): before accessing an expensive resource, the arguments are materialized and sorted, and thus a cache of just one value is sufficient. To ensure a fair comparison, HBBJ is de-synchronized from its parent and child operators through the standard Exchange operator [12].

QEPs tested: We study the tuple output rate from the following three QEPs:

- QEP1: $q(X, Z) \overset{\rightarrow}{\underset{X}{\bowtie}} BA(f(X^b Y^f))$
- QEP2: $q(X, Z) \overset{\rightarrow}{\underset{X}{\bowtie}} BA(f(X^b Y^f)) \overset{\rightarrow}{\underset{Z}{\bowtie}} BA(g(Z^b T^f))$
- QEP3: $\sigma_s (q(X, Z) \overset{\rightarrow}{\underset{X}{\bowtie}} BA(f(X^b Y^f))) \overset{\rightarrow}{\underset{Z}{\bowtie}} BA(g(Z^b T^f))$

X, *Y*, *Z* and *T* are integer attributes, *q* is a given QEP producing tuples of the form *(X,Z)*, and *s* is a selection condition on the result of function *f*. While very simple, such QEPs are very general, as the subplan *q* may be arbitrarily complex. Indeed, as mentioned in section 3.3, global optimization techniques [6], [15], [24] would order relational operator *before* expensive BindJoins. QEP3 is represen[tative of plans with restrictions on the function results.

Characteristics of the tuples produced by q: We use a data generator which constructs the set of *(X,Z)* tuples output by *q* according to a set of parameters: (i) the number of tuples; (ii) the number of distinct values for each attribute; (iii) the distribution law, assuming that the distributions of the attributes are mutually independent; and (iv) the rate at which tuples can be obtained from *q*. This rate is important, since an ERBJ can only accumulate tuples if the tuple input rate is larger than its processing rate. Finally, we are able to deliver the tuples in specific orders. When generating *X* and *Z* according to a uniform data distribution, we did not enforce perfect uniformity; rather, we used a uniformly distributed random variable to draw 10,000 values out of 2,500 possible, yielding 2450 values. For Zipfian distributions, we used a low zipf factor ($\alpha = 0.2$) representative of real-life databases distributions delivering a total of 1,450 distinct values in 10,000 tuples.

Graphs: Each graph presented in the sequel shows the number of result tuples as a function of the running time. Thus, the response time is indicated by the width of the curve while the good ER behavior is shown by the convexity of the curve.

5.2 ER behavior of the BindJoin on simple query plans

Our first four experiments study the ER behavior of QEP1 and QEP2, in which all BindJoins are implemented using as HBBJ, SBBJ, and ERBJ, when *q*'s output follows a uniform, respectively Zipfian input data distribution (see figure 6-10). In this section, we assume that all tuples output by *q* have been transmitted towards its parent operator before the BindJoins start to run. This assumption will be lifted in the next sections; however, it is quite realistic if *q* consists only of regular relational operations, while the accesses to *f* and *g* are much more expensive (this configuration is also considered in [3], [6], [11]).

In experiments 1-4, HBBJ delivers few tuples at the beginning, since most *X* values that *f* (or *g*) processes are new; *f* (or *g*) must be computed. As the cache gets filled, toward the end of the execution, the output rate increases. With a Zipfian distribution, the ER behavior is slightly improved since some values are very frequent.

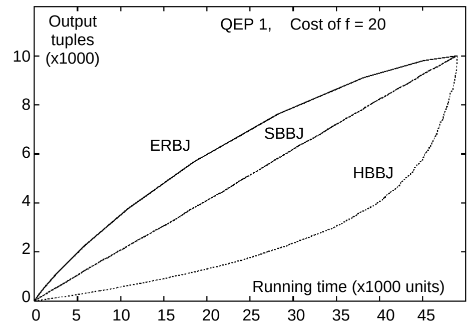

Fig. 6. Exp.1: one BindJoin, uniform dist.

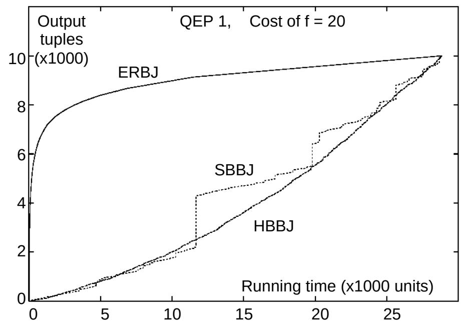

Fig. 7. Exp. 2: one BindJoin, Zipfian dist.

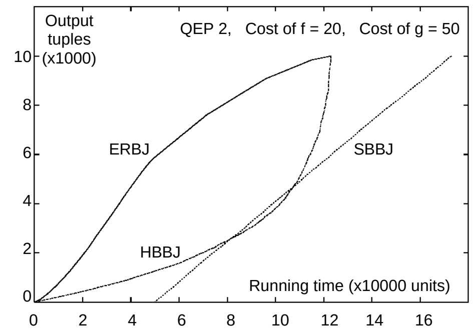

Fig. 8. Exp.3: two BindJoins, uniform dist.

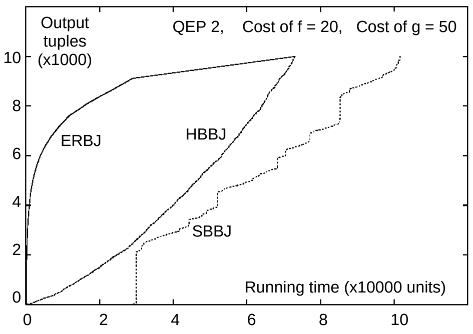

Fig. 9. Exp.4: two BindJoins, Zipfian dist.

In experiment 1 (figure 6), the curve for SBBJ is almost a straight line: after each processed value, SBBJ outputs in average 4 tuples (2.450 distinct values in 10.000 tuples). With the Zipfian distribution, in experiment 2 (figure 7), SBBJ outputs bursts of tuples corresponding to the groups of tuples sharing the same *X* value. Unfortunately, SBBJ is not able to exploit the most frequent values, since it is encountered somewhere in the middle of the value range. In the case of QEP2 (figures 8 and 9), the SBBJ output is delayed until the first BindJoin has finished, since the tuples have to be re-sorted on *Z* before the second BindJoin can start. The overall ER of the plan using SBBJ is poor.

In figure 6, ERBJ does slightly better than SBBJ, since it chooses the most frequent values first, even if there is only a very small frequency variation in uniform distributions (in our randomly generated distribution the frequency varies from 3 to 6 with an average of 4). With Zipfian distributions (figures 7 and 9), ERBJ exploits the presence of the few very popular values (characteristic to Zipf) by a very large pack of tuples output right at the beginning of the execution.

Conclusions: HBBJ has a bad ER behavior since tuples for which the result was already in the cache have to wait for their turn before being output. However, HBBJ is non-blocking and works in pipeline. SBBJ does not take advantage of the most frequent values, since it sorts in a *value based* data order in which most frequent values do not necessarily come first. But the main disadvantage of SBBJ is its blocking aspect, which leads to very poor ER and increased response time. Our proposed *ERBJ* consistently outperforms the others. Like HBBJ, it is non-blocking; like SBBJ, it outputs simultaneously several tuples sharing an argument value, when the result for this value is available. All three algorithms use a cache, and therefore are useful if there are duplicates in their input. However, the advantage of the ERBJ over the two others is increased by the presence of skewed distributions, since it processes first the more popular values to improve its output rate.

5.3 ER behavior and response time with more complex plans

In experiment 5 (figure 10), we study the output rate of *QEP_1* when *q* only provides one tuple every time unit (since, e.g., it retrieves input data from a remote site or performs a complex subplan). For comparison, we also plotted the curve for ERBJ when there is no input limitation (the same as in figure 7). The curve corresponding to ERBJ with delay between inputs has an almost linear aspect for the first *10,000* running time units; then, it joins the curve of ERBJ without delays. The join occurs after 10,000 time units; thus, even with tuple buffer limited by the slow input, the ERBJ has been able to choose frequent values to process (otherwise, it would have met the non-restricted curve even later). The ERBJ curve is quite close to the absolute optimum in the presence of restrictions, which would be to output 1 tuple every time unit (as soon as it arrives). Note, however, this optimum is not achievable, since new values require processing. The ERBJ is so good because the most frequent values are scattered uniformly over a data set following a Zipf distribution. It would not be the case with larger delays and a very bad data order (e.g. the most frequent values last), since the ERBJ could not see (and choose) frequent values early. HBBJ is not affected by the delay between inputs since it is de-synchronized from *q*. Obviously, with a larger delay, idle time may occur. Finally, as expected, SBBJ has to wait 10,000 time units before it can sort them and proceed.

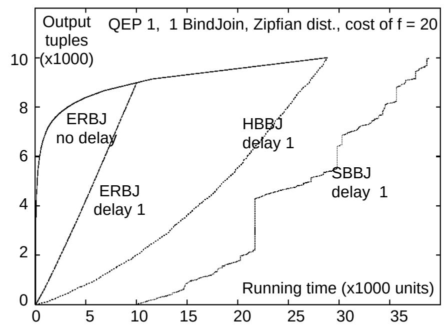

Fig. 10. Exp.5: effect of delays

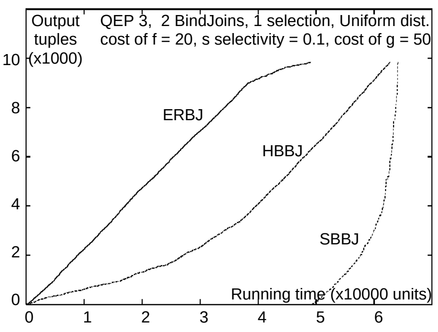

Fig. 11. Exp.6: selection with 2 BindJoins

Experiment 6 (figure 11) studies the output of QEP_3, when the selection σ_s eliminates nine *(X,Y,Z)* tuples out of ten. HBBJ not only has a worse ER than the ERBJ, but also a longer response time. At the beginning of the execution, the first HBBJ outputs few tuples, of which very few survive the selection. The second HBBJ is therefore often idle. Towards the end of the execution, as the cache of the first BindJoin fills, it generates tuples at a faster rate, and the second BindJoin, even after the selection, becomes overloaded with new values. This behavior (first idle, then overloaded) translates into the increased running time. The same synchronization problem can arise in a variety of settings – for example, if we replace σ_s by a regular join, a mixture of joins and selections etc. The same holds for QEP_2, if g is significantly more expensive than f [19]. By contrast, when using ERBJs, the large early output rate of the first one, even if trimmed by the selection, means that the second one is always busy. Therefore, the pipeline between the two is perfect (in figure 11, the total running time is that of the first BindJoin). The curve corresponding to sort-based BindJoins is delayed by the running time of the first BindJoin.

5.4 Other Experiments and Conclusion

We summarize here the results of some other measures that we performed but do not present here (see the extended version of this paper [19]).

A BindJoin's input data can come in various orders, resulting from the processing of the subplan *q*. The ERBJ and the SBBJ are not sensitive to such orders, since they perform their own re-ordering (unless delays proscribe it). In contrast, the HBBJ is very sensitive to the order.

We have performed experiments on QEPs with up to four BindJoins. We have noticed that as the number of BindJoin increases, if they have different costs, produce several output tuples per input tuple, or if there are interspersed selections, the probability of synchronization problems for hash-based BindJoins (as in figure 10) increases. The explanation is that some BindJoins are successively idle then overloaded. Therefore, the advantage of ERBJ over hash-based BindJoins is more important.

In conclusion, ERBJ provides always a significantly better ER than the state-of-the-art algorithms for accessing restricted resources, This advantage depends on the presence of duplicates in the input, and increases with the non-uniform distribution of input values. It is remarkably stable with variations in the input order, delays between two successive input tuples, and synchronization problems in the QEP. Furthermore, its excellent ER properties may improve the overall RT of complex queries.

6 Related Optimization and Execution Techniques

Significant work has been done on online and adaptive query processing; see, e.g., [1], [13], [16], [22], [25]. These works do not address the specific BindJoin operator but are however complementary. Indeed, as mentioned in section 3.3, the ER behavior of a QEP is the result of the good ER behavior of all its query operators.

The ObjectGlobe [4] project uses Java user-defined operators, loadable from external code repositories. Mocha [23] addresses query optimization for user-defined functions that may be shipped across the network. When possible, this technique is very profitable, since it avoids data transfer, however, its application is limited, as restricted resources depend on a particular environment, or cannot be copied.

In [5], the sort-based BindJoin algorithm, with which we compared in section 5, is presented. In [14], the authors propose a *hybrid cache* algorithm that degrades gracefully if the cache outgrows the available memory. Compared with [5], [14], our BindJoin exploits duplicates and parallelism to improve its ER; also, we use it for avoiding duplicate blob transfers. We have shown in section 4.3 how our ERBJ deals with memory limitations: thus, it has the good properties of the hybrid cache algorithm, plus an improved ER behavior.

[20] studies query execution in a client-server context, with expensive UDFs. They recognized that UDF can be executed as joins, and that existing work on distributed join processing, and semi-joins, could be reused. Our techniques have a broader scope, since they apply for *any* restricted resource access, and improve ER.

Two approaches for modeling expensive functions exist, and therefore two classes of optimization algorithms. LDL [7] models a function as a table. While this requires little modification to a regular optimizer, the number of functions is reflected exponentially in the size of the search space, just like the number of regular tables. This drawback is avoided by the second approach, in which expensive functions are assimilated with selections. Optimization methods based on *predicate ranking* have been proposed in, e.g., [15], [6]; in a distributed setting, they are no longer optimal, due to data transfer costs [20]. In [24], efficient optimization algorithms improve over predicate ranking by considering interesting data orders, and bushy QEPs.

Our modeling is closer to LDL, for the following reason. In general, a restricted, expensive resource is *not* a function or a predicate. It may be a full-parameterized sub-query, optimized as a complex operator tree QEP (caching for sub-query results has been suggested in [14]). Ignoring such parameterized sub-plans may lead to loss of optimality [9]. To combine a sub-plan, providing bindings, with another parameterized sub-plan requiring them, we need a *binary* BindJoin operator, not a selection one. Query optimization algorithms for tables with binding patterns, using joins and BindJoins, are provided in [9], which shows that in practical cases, the presence of access restrictions drastically limits the size of the search space.

7 Conclusion

In this paper, we investigate the publication model and algorithms for resource sharing in a fully distributed peer-to-peer mediation architecture. We showed how binding patterns can be used to uniformly model data sources including functions as well as blobs, providing an attractive alternative to semi-joins. We analyzed the impact of expensive functions and blobs on the design of the BindJoin operator and on its integration in the query execution plan. We considered three performance goals: (i) total work; (ii) response time; and (iii), the more specific early tuple output rate. The main specificity of our BindJoin is that it exploits the presence of duplicates in its input to provide an important early tuple output rate, so that the user obtains most of the query results fast. Since our BindJoin operator includes all the optimizations, the publisher task is significantly reduced, while providing good query performance.

References

[1] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. *IN PROC. OF ACM SIGMOD CONF.,* 2000.

[2] P. Bernstein and D W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM,* 1981.

[3] L. Bouganim, F. Fabret, F. Porto, and P. Valduriez. Processing queries with expensive functions and large objects in distributed mediator systems. *ICDE* 2001.

[4] R. Braumandl, M. Keidl, A. Kemper, and D. Kossmann et al. ObjectGlobe: Ubiquitous query processing on the internet. In *Workshop on Technologies for E-Services,* 2000.

[5] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. of the VLDB Conf.,* 1993.

[6] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transaction on database system (TODS),* 2(24), 1999.

[7] The Decair and Thetis projects. Available at http://www-caravel.inria.fr/Econtrats.html.

[8] The Ecobase Team. The Ecobase project: Database and web technologies for environmental information systems. *SIGMOD Record,* 30(3), 2001.

[9] D. Florescu, A. Levy, I. Manolescu, and D. Suciu Query optimization in the presence of limited access patterns. *In Proc. of ACM SIGMOD Conf.,* 1999.

[10] S. Ganguly, W. Hassan, and R. Krishnamurthy. Query optimization for parallel execution. *In Proc. of ACM SIGMOD Conf.,* 1992.

[11] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. *In Proc. of ACM SIGMOD Conf.,* 2000.

[12] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys,* 25(2), June 1993.

[13] P. Haas and J. Hellerstein. Ripple joins for online aggregation. *SIGMOD Conf.,* 1999.

[14] J. Hellerstein and J. Naughton. Query execution techniques for caching expensive methods. *In Proc. of ACM SIGMOD Conf.,* 1996.

[15] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *In Proc. of ACM SIGMOD Conf.,* 1993.

[16] Z. Ives, D. Florescu, M. Friedman, D. Weld, and A. Levy. An adaptive query execution system for data integration. *In Proc. of ACM SIGMOD Conf.,* 1999.

[17] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys,* 2000.

[18] The LeSelect Project. Available at http://www-caravel.inria.fr/LeSelect.

[19] I. Manolescu, L. Bouganim, F. Fabret, and E. Simon. Efficient data and program integration using binding patterns. Tech. Report no. 4239, INRIA. Extended version available at: www-rocq.inria.fr/~manolesc/BJ-extended.ps

[20] T. Mayr and P. Seshadri. Client-site query extensions. *In Proc. of ACM SIGMOD.,* 1999.

[21] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Proc. of the ACM PODS,* San Jose, CA, 1995.

[22] A. Raman, B. Raman, and J. Hellerstein. Online dynamic reordering for interactive data processing. In *Proc. of the VLDB Conf.,* 1999.

[23] M. Rodriguez-Martinez and N. Roussopoulos. MOCHA: A self-extensible database middleware system for distributed data sources. In *Proc. of ACM SIGMOD Conf,* 2000.

[24] W. Scheufele and G. Moerkotte. Efficient dynamic programming algorithms for ordering expensive joins and selections. In *Proc. of the EDBT Conf.,* 1998.

[25] T. Urhan and M. Franklin. XJoin: a reactively scheduled pipelined join operator. In *IEEE Data Engineering Bulletin,* 2000.

[26] A.N. Wilschut and P.M.G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the PDIS Conf.,* 1991.