Efficient Data and Program Integration Using Binding Patterns

Ioana Manolescu, Luc Bouganim, Françoise Fabret, Eric Simon

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient Data and Program Integration Using Binding Patterns

Ioana Manolescu — Luc Bouganim — Françoise Fabret — Eric Simon

N° 4239

August 2001

THÈME 3

ISSN 0249-6399 ISRN INRIA/RR--4239--FR+ENG

INRIA
ROCQUENCOURT

Efficient Data and Program Integration Using Binding Patterns

Ioana Manolescu [*] , Luc Bouganim [†] , Françoise Fabret [‡] , Eric Simon [§]

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Caravel

Rapport de recherche n° 4239 — August 2001 — 27 pages

Abstract: In this work, we investigate 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, sharing a resource should be easy; therefore, we need a simple concept for modeling resources. Second, we need an efficient architecture for distributed query execution, capable of handling well costly computations and large data transfers. To model heterogeneous resources, we propose using the unified abstraction of table with binding patterns, simple yet powerful enough to capture data and programs. To exploit a resource with restricted binding patterns, we propose an efficient BindJoin operator, following the classical iterator model, in which we build optimization techniques for minimizing large data transfers and costly computations, and maximizing parallelism. Furthermore, our BindJoin operator can be tuned to deliver most of its output in the early stages of the execution, which is an important asset in a system meant for human interaction. Our preliminary experimental evaluation validates the proposed BindJoin algorithms, and shows they can provide good performance in queries involving distributed data and expensive programs.

Key-words: data and program integration, distributed query processing, binding patterns

[*] INRIA, Caravel project. Contact: Ioana.Manolescu@inria.fr
[†] PRISM laboratory, University of Versailles and INRIA, Caravel project. Contact: Luc.Bouganim@prism.uvsq.fr
[‡] INRIA, Caravel project. Contact: Francoise.Fabret@inria.fr
[§] INRIA, Caravel project. Contact: Eric.Simon@inria.fr

Intégration efficace de données et de programmes utilisants des patterns d'accès

Résumé : Dans ce rapport, nous étudions l'intégration de données et de programmes dans une architecture symétrique, complètement distribuée. Pour qu'un tel système soit efficace à grande échelle, deux problèmes doivent être résolus. Premièrement, le partage d'une ressource doit être facile, d'où le besoin d'un mécanisme simple pour décrire ces ressources. Ensuite, nous avons besoin d'une architecture efficace pour l'exécution distribuée des requêtes, capable d'exécuter des calculs coûteux et des transferts de données volumineuses. Pour modéliser des ressources hétérogènes, nous proposons d'utiliser le concept unifié de table avec des patterns d'accès, simple mais assez puissant pour décrire des données et des programmes. Pour exploiter une ressource avec des patterns d'accès, nous proposons un algorithme efficace pour l'opérateur BindJoin, suivant le modèle itérateur, dans lequel nous mettons en oeuvre des techniques d'optimisation pour réduire les transferts de données volumineuses, réduire les calculs coûteux, et maximiser le parallélisme. De plus, notre opérateur de BindJoin peut être modifié pour produire la plupart de ses résultats au début de l'exécution, une qualité importante dans un système ayant des utilisateurs humains. Notre évaluation experimentale préliminaire valide les algorithmes de BindJoin proposés, et montre qu'ils peuvent atteindre des bonnes performances dans des requêtes sur des données distribuées et programmes coûteux.

Mots-clés : intégration de données et de programmes, traitement de requêtes distribuées, patterns d'accès

1 Introduction

There is a growing interest in the scientific community to allow communities of users (a.k.a. virtual organizations) to share resources consisting of both data collections and application programs. This vision is best reflected by recent initiatives such as the "Grid Computing" project [7], that aims at constructing a "meta computer": a large scale, distributed computing environment, that provides transparent access to highly heterogeneous resources.

To illustrate, consider a scientific application that involves two image databases[1]. On site S_1, there is a collection of infrared satellite images of the Earth, taken every three hours, stored in the GIF format. On site S_2, data from a different satellite survey has been processed into a map of the ozone cover of the Earth, taken every week; this data is available in HDF (*H*ierarchical *D*ata *F*ormat). On site S_3, an application LowOzone:HDF$\rightarrow$ boolean. Now, suppose that a user wants "to retrieve on site S_4 pairs of images taken from the two sources for those days when there were noticeable holes in the ozone cover". Such a query may require the following actions: (1) extract images from S_2 and transfer them to site S_3, (2) select those images that present holes in the ozone cover using the LowOzone program on site S_3 (we assume that programs can only execute on their native site and cannot be shipped through the network), and (3) obtain the dates associated with the resulting images, extract images for these dates from S_1 and transfer them to S_4.

Even if all these resources are interconnected via a middleware system, developing an application program that implements the query requiring the above actions could be tedious. Existing mediator or federated database systems, such as IBM's DataJoiner, Garlic [12], Tsimmis [8], etc., would help for the sharing of data, but they currently do not support the sharing of distributed, heterogeneous, user-defined functions.

LeSelect is a fully distributed mediator system that has been specifically designed to facilitate the publication of resources within a virtual community, and to enable users to formulate high-level queries over published resources [15]. To publish resources needed for our example problem, we install a LeSelect server on each site, except S_4. Each resource is published via a wrapper, provided by the publisher. Data collections and programs are published as relational tables through these wrappers yielding, for instance, the tables whose schemas are displayed in Figure 1. Given these tables, the SQL query shown in Figure 1 could be issued at site S_4 through a LeSelect client interface.

The contribution of this paper is to provide a query processing architecture and new algorithms to deal with the efficient processing of the kinds of queries with LeSelect could likely deal. We use the example to explain our technical contributions.

First, suppose that the measurements of S_2 cover a period of time four times longer than those of S_1. Thus, only 1/4 of the tuples of S_2 will join (on the date attribute) with tuples of S_1, and each tuple of S_2 will generate 8 tuples in the result of the join (S_1 has 8 images per day). Hence, the join between IRSat and OzoneSat is not selective, and most existing optimization techniques like [4, 13] will generate a QEP in which the predicate

[1] The image collections are very closely inspired from real-life data sources, available at `http://www.ssec.wisc.edu/data/comp/ir/` and `ftp:daac.gsfc.nasa.gov`.

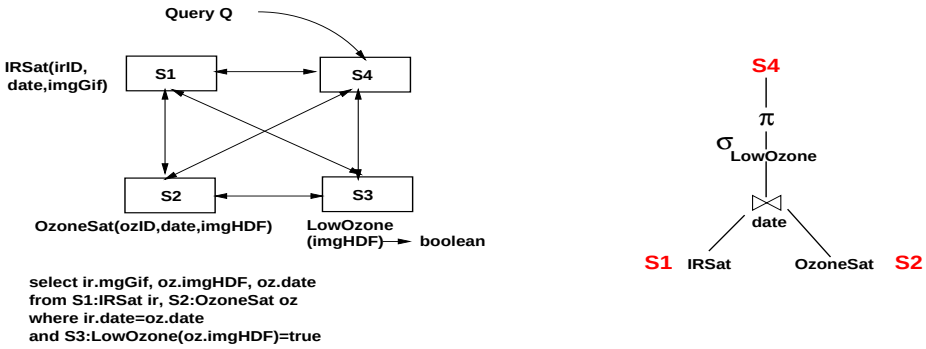

Figure 1: Sample query on distributed data and programs (left) and one possible execution plan (right).

"LowOzone=true" is performed before the join on date (as we did in actions 1 and 2 above). However, as noted in [2, 14], if a cache is used for LowOzone at S_3, it would be better to compute the join first, since LowOzone will be applied on 4 times fewer values of the image than it would if it were executed first. To cope with this observation, we present new cache-based algorithms to perform expensive selections that invoke published programs; these algorithms enable the generation of efficient QEPs, as described in [2].

Now, suppose that the join is executed first, as shown in figure 1; it is either performed on S_1 or on S_2. In both cases, this entails the transfer of a large amount of data between these two sites; similarly, the result of the join should be transmitted to S_3. However, by noting that the OzoneSat images are only needed on S_3 and S_4, while the IRSat images are needed directly on S_4, we could avoid many unnecessary image transfers. Building on these points, we present an architecture and algorithms that enable an operator of a QEP that needs to process a large data structure (in our case, LowOzone which is executed on site S_3) to directly fetch it from the site where it is published, which in our case is S_2.

Finally, our query example provides many opportunities for parallelism. Suppose that several computers are available on S_3 to execute the program LowOzone. In this case, the operator in the QEP responsible for the invocation of LowOzone should gracefully scale up to monitor multiple parallel executions of LowOzone (this amounts to intra-operator parallelism). LowOzone should be able to process OzoneSat images while new images are flowing through the network to site S_3. In this paper, we propose architectural components and parallel implementation of an algorithm that supports both intra- and inter-operator parallelism. Doing this, we generalize the asynchronous iteration technique presented in [9] to parallelize program executions, and transfers of program arguments.

This paper is structured as follows. Section 2 explains the basic principles for publishing data and programs in LeSelect. In Section 3, we describe the query processing infrastructure of LeSelect, with a focus on its execution model and the efficient management of large data transfers; we then introduce the BindJoin and BindAccess logical operators, used for

program invocation and large data transfer. In Section 4, we describe efficient algorithms for the BindJoin and BindAccess operators, encapsulating cache and parallelism; also, we show how the presence of duplicates can be exploited to improve the early tuple output rate of our BindJoin algorithm. Section 5 presents experimental results evaluating the proposed BindJoin algorithm. We discuss related work and conclude in Section 7.

2 Publishing data and programs in LeSelect

The following steps are required in order to publish a *resource* (data collection or program) in LeSelect: (1) describe it as one or more relational tables, (2) create a wrapper for it and (3) register the wrapper with a LeSelect server. Thus, on any publishing site, a LeSelect server has to be running. In this section, we present the generic model of tables with binding patterns, and we describe the interface between wrappers and LeSelect servers.

2.1 Using binding patterns for resource description

In LeSelect, all published resources are modeled as tables, and we describe the set of parameterized select-project queries that can be processed over each individual table using the concept of binding pattern introduced in [17]. A binding pattern bp for a table $T(a_1, a_2, \ldots, a_n)$, is a mapping from $\{a_1, a_2, \ldots, a_n\}$ to the alphabet $\{b, f\}$. The meaning of the binding pattern is the following: those a_i that bp maps to b are *bound*, i.e., their values must be supplied in order to obtain information from T, while values of attributes mapped to f are free, i.e. they can be obtained from the data source as soon as values for all b attributes are supplied.

In the case of a data resource modeled as a table T, a binding pattern that maps all attributes of T to f indicates that the tuples of T can be directly obtained via a Scan operation. For instance, the binding pattern OzoneSat(ozID^{f}date^{f}imgHDF^f) indicates that a full scan of this table is allowed (upper scripts denote the mapping for each attribute). A restricted binding pattern (i.e., in which some attributes map to b) may reflect an optimized access method for those attributes, or an access restriction imposed by the resource manager. Thus, OzoneSat(ozID^{f}date^b imgHDF^f) specifies that the value of the date attribute has to be supplied in order to obtain tuples from OzoneSat.

Requirements on blob publication As we explained in Section 1, special optimization techniques can be used for queries involving blobs. In order to apply these techniques, we pose some requirements on the modeling of a table with blobs.

Definition (Blob) An attribute A of table R is a blob iff:

- A is declared to be of the LeSelect data type binary;
- if A is mapped to f by some binding pattern of R, then R contains a small-sized attribute $blobID$ which identifies A (i.e., such that the functional dependency $blobID \rightarrow A$ holds);

- R has at least the binding pattern $R(blobID^b A^f)$.

The choice of declaring an attribute as a blob is left to the publisher; as in commercial DBMSs, attributes with a size greater than a given threshold (e.g., a few Kb) should be declared as blobs. For any blob that is published, or created by a program, if the blob may be obtained in a query (*i.e.*, is mapped to f in at least one binding pattern), we require a blobID attribute for caching purposes and for optimizing blob transfer. BlobIDs are system-generated in the case of published data residing in an DBMS. In a simpler setting, a large data object is usually stored in a separate file, whose complete name (i.e. host/filename) can be used as a blobID. For example, suppose that an ImgConvert program is applied on the OzoneSat images in the sample query in Figure 1, to convert it into GIF format. Such a program takes as input a blob File_1, the format codes of the input and output file, and produces a File_2 blob result. ImgConvert must have among its binding patterns ($\text{File}_2\text{ID}^b\text{File}_2^f$): the File_2ID attribute is required since the program exports a binding pattern where File_2 is free. Such an attribute is not necessary for File_1 since there is no real data to be extracted - on the contrary, only File_1 has to be supplied. The wrapper of a resource that creates blobs must assign to each produced blob a system-wide unique identifier, e.g., by appending the host name and a local reference. Finally, the restricted access pattern to blobs is used by the optimizer to construct QEP handling blobs efficiently, as we show in the next section.

In LeSelect, a program[2] can be assimilated to a virtual table (a.k.a table function) as in [3, 5]. We use binding patterns to distinguish the attributes that correspond to program arguments (which need to be supplied in the query) from those that correspond to program results. For example, the set of binding patterns of the ImgConvert program described above includes (Format_1^b Format_2^b File_1^b File_2ID^f). Accessing ImgConvert following this pattern only returns the ID of the output image; the (File_2ID^b File_2^f) binding pattern presented above provides for proper blob handling. In LeSelect, any published resource is thus modeled as a table with binding patterns.

2.2 Wrappers in LeSelect

In a data integration system, a wrapper has two important functions. First, it holds metadata concerning the resources that it exports, like table and attribute names, attribute types, etc. Such information is used, for instance, by the query optimizer, or by human users browsing the data catalog. Second, wrappers may provide query processing capabilities. We aim at keeping the effort required for publishing a resource as low as possible. In this section we describe the simple metadata and minimal query processing capabilities required from a wrapper to enable the efficient execution of queries over tables with binding patterns.

[2] Throughout this paper, we restrict ourselves to programs that take a tuple of values as input and return one or several tuples of values as output. Programs taking as input one or several tables and returning tables in the output are modeled differently in LeSelect [15].

Wrapper metadata A wrapper W manages a set of tables $\{T_1, T_2, \ldots, T_n\}$, where each T_i corresponds to a resource. For every T_i, W exports two kinds of metadata. The first one concerns the definition of the table: columns number, names and types, as well as a set of binding patterns over T_i's attributes $\{BP_i^1, \ldots, BP_i^k\}$.

The second kind of metadata consists of statistics on the data published via T_i, which are intensively exploited by the query optimizer. The wrapper exports a distinct set of statistics for each binding pattern *BP;* these statistics depend on the form of *BP*. If *BP* is of the form $ff \ldots f$, the wrapper declares the number of tuples in T_i, and for each attribute a, the number of distinct values n of the attribute. This number is important in deciding whether or not to use cache. If the pattern *BP* is restricted, the wrapper exports two cost parameters: the average per-call cost, c_{BP}, and the average number of returned tuples per tuple of arguments, s_{BP}. We consider c_{BP} to be the time elapsed between the moment when the restricted resource is accessed, and the moment when all the returned results are available. The s_{BP} value also reflects the selectivity of the access following *BP*, since there may be no answer for a given argument set. The publisher of a restricted resource may also specify the optimal number of parallel accesses to the resource ($opt_{\|}$). In some cases, this number may be statically determined (e.g., a program running in batch mode on five computers), but in others it is very difficult to predict since it may depend on parameters like machine load, network speed, etc. To deal with such situations, we present in Section 4.2 an algorithm that adaptively determines the optimum degree of parallelism for exploiting a resource.

Wrapper capabilities The query capabilities of a wrapper are provided as a set of boolean methods like *canJoin*(), *canProject*(), etc. A wrapper can only apply an operator to resources that it publishes. At query execution time, a wrapper agrees to execute a sub-plan *sQEP* of a global query plan iff (a) it publishes all the resources in the *sQEP*'s leaf nodes and (b) the wrapper has declared itself as being capable of executing all operators found in the internal *sQEP*'s nodes. The wrapper corresponding to a full-fledged DBMS may declare itself as capable of executing complex subqueries, since it delegates them to the underlying DBMS.

To keep the publication process simple, we require only minimal query processing capabilities from a wrapper. First, it has to execute "leaf" operators providing access to its resources. For tables with a $ff \ldots f$ binding pattern, this amounts to accept Scan subqueries. In the case of a restricted access resource, we require that the publisher provides a call-based interface, of the form *tuple*[] *callResource*(*tuple arguments*); on top of this interface, we provide a simple configurable wrapper that uses this interface to access the resource. Second, for any table containing a blob, the wrapper must provide a method $readBlob(blobID, startPos, endPos, buffer)$, to ensure that the blob is physically accessible.

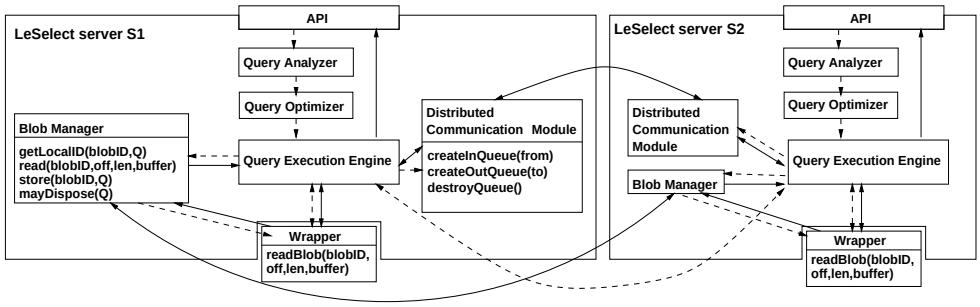

Figure 2: General outline of a LeSelect server.

3 LeSelect query processing infrastructure

In this section, we describe the query processing framework of LeSelect. We present the general architecture of a server, and outline the processing of a distributed user query. We then recall the general iterator model followed by all operators in LeSelect, and explain the usage of the Exchange operator for inter-site transfers of small data items. Next, we introduce the logical BindJoin and BindAccess operators, used to access resources with restricted binding patterns. Finally, we present the design of a Blob Manager that we use to efficiently handle blob transfers between sites.

3.1 General architecture of LeSelect

Figure 2 presents the outline of a LeSelect server and traces the possible interactions between two servers during query processing. In this figure, solid lines represent data flow, while dashed lines trace command or statistic information flow.

Users can pose queries on any LeSelect server. Each table T involved in the query has a universal name, constructed from the name of the wrapper W that publishes it and the address of the LeSelect server S to which W is attached. Queries are restricted to select-project-join operations. After parsing, the query is handed to the optimizer, which uses the metadata published by the wrappers of all the tables identified in the from clause to construct a distributed QEP; the optimizer applies a dynamic programming algorithm, enhanced with binding patterns [6]. In this plan, some operations may be delegated to resource wrappers, while others will be performed by LeSelect servers that publish the resources, or to the server where the query is posed. The query execution engine on the initial query site coordinates the execution of the plan constructed by the optimizer. During execution, the query execution engine of the server S_1 may request that the engine of a server S_2 executes a sub-query (dashed arrow among the two execution engine in Figure 2). All transfers of blob objects between two servers that the execution might entail are performed by those server's Blob Managers; all other types of data are transferred via the two Distributed Communication Modules.

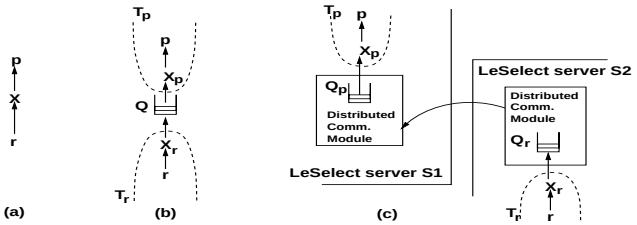

Figure 3: QEP using Exchange(a) and its implementation within one site (b) and on several sites (c).

In the next subsections, we focus on LeSelect components that include innovative features: the query execution engine and the Blob Manager.

3.2 Iterators and the Exchange operator

In LeSelect, each operator of a QEP is implemented as an iterator, following [11]. Iterators are self-scheduling data processing units; their API consists of an initialization *open*() call, a *next*() method producing one tuple at a time, and a *close*() method to release resources and terminate. To answer a *next*() call, the top-level operator in a QEP may call the *next*() methods of one or more of its inputs, depending on its implementation. The basic iterator model is synchronous; if an operator p is the parent of an operator r, when p issues a *next*() call to r, p's execution blocks until r returns a tuple.

The Exchange iterator was introduced to decouple the execution of several operators, allowing them to work in parallel [10]. Figure 3 shows an Exchange operator X inserted between an operator p and its child r. On $X.open()$, the communication queue Q is created, and X is split into two independent processes. T_p runs operator p and the X_p consumer part of the Exchange, while T_r executes X_r, the producer part of the Exchange. T_r is no longer driven by the upper part of the plan; it runs independently and iteratively issues *next*() calls to r until an *eof* is reached. An Exchange operator inserted between two operators r and p running on the same site acts as a synchronization buffer, absorbing bursty output from r and providing it to p at p's required pace.

If operators p and r run on remote sites, Q is split into two queues Q_p and Q_r, each on the site of one operator, with Q_r's contents being transferred into Q_p by some data communication mechanism; T_r and T_p run in parallel. In this case, the size of Q allows us to control the de-synchronization of T_r and T_p. The synchronous iterator model corresponds to a queue of size 0, where a tuple is processed by p immediately after being processed by r. By increasing Q's size, we allow a slack among the tuples processed by r and p. The size of Q is a parameter of the Exchange operator, set at creation time by the optimizer.

In LeSelect, the implementation of a distributed Exchange operator relies on a *distributed communication module* (DCM) that creates the communication queues Q_p and Q_r, as in Figure 3(c). Within each DCM, communication daemons are in charge of transferring tuples

among the queues Q_r and Q_p. Note that all transfers of small-size data between two LeSelect servers are done via the distributed Exchange operators implemented by the DCM.

3.3 The BindJoin and BindAccess operators for exploiting restricted resources

The access to a resource according to a restricted binding pattern is performed via a logical BindJoin operator, also called dependent join [6] or θ semi-join. Using our notations, we explain the semantics of a BindJoin operator, as it was initially introduced in a centralized database context.

Definition (BindJoin) The BindJoin of two arbitrary relations R_1 and R_2 is defined by the following formula, where $\overrightarrow{\bowtie}$ denotes the BindJoin, a vector $\overrightarrow{X}$ denotes a vector of attributes, and $\overrightarrow{x}$ denotes a vector of values for the attributes in $\overrightarrow{X}$:

$$R_1(\overrightarrow{X}^f\overrightarrow{Z}^f)\overrightarrow{\bowtie}_{\overrightarrow{X}}R_2(\overrightarrow{X}^b\overrightarrow{Y}^f) = \{(\overrightarrow{x},\overrightarrow{y},\overrightarrow{z})|(\overrightarrow{x},\overrightarrow{z}) \in R_1 \wedge \overrightarrow{y} \in R_2.callResource(\overrightarrow{x})\}$$

However, in a distributed context, modeling this operation as a single operator raises a performance issue. Suppose that R_1 is on site S_1 and R_2 is on site S_2. Since $R_2.callResource()$ has to be executed on site S_2, the BindJoin operator must also execute on site S_2, and tuples of R_1 must flow from S_1 to S_2. Now, if the query result must be returned on S_1, there is a performance penalty, which consists of the useless transfer of the $\overrightarrow{z}$ part of the $(\overrightarrow{x},\overrightarrow{z})$ tuples of R_1: $\overrightarrow{z}$ is unnecessary for the computation of the $\overrightarrow{y}$ result. Instead, a better design would be to transfer only $(\overrightarrow{x})$ values to S_2, compute the corresponding $\overrightarrow{y}$ values on S_2, return $(\overrightarrow{x},\overrightarrow{y})$ tuples to S_1, and finally output the resulting $(\overrightarrow{x},\overrightarrow{y},\overrightarrow{z})$ tuples at S_1. However, as explained in the previous subsection, we retain the general principle that any data flow in a QEP between operators running on two different sites be modeled by a distributed Exchange operator between the two operators. This led us to introduce a new operator, called BindAccess (BA, for short), that is always the "inner" child of a BindJoin operator, as follows:

Definition (BindJoin and BindAccess) Let R_1 and R_2 be two relations, and $BJ = R_1(\overrightarrow{X}^f\overrightarrow{Z}^f)\overrightarrow{\bowtie}_{\overrightarrow{X}}R_2(\overrightarrow{X}^b\overrightarrow{Y}^f)$ denote the BindJoin of these relations. Then the BindAccess associated with BJ is the operator noted BA, defined by the following formula:

$$R_1(\overrightarrow{X}^f\overrightarrow{Z}^f)\overrightarrow{\bowtie}_{\overrightarrow{X}}R_2(\overrightarrow{X}^b\overrightarrow{Y}^f) = \{(\overrightarrow{x},\overrightarrow{y},\overrightarrow{z})|(\overrightarrow{x},\overrightarrow{z}) \in R_1 \wedge (\overrightarrow{x},\overrightarrow{y}) \in BA[\{\overrightarrow{x}\}](R_2)\}$$

$$\text{where } BA[\{\overrightarrow{X}\}](R_2) = \{(\overrightarrow{x},\overrightarrow{y})|\overrightarrow{x} \in \{\overrightarrow{X}\} \wedge \overrightarrow{y} \in R_2.callResource(\overrightarrow{x})\}$$

This notation emphasizes that the BindAccess is always parameterized by a set of $\overrightarrow{x}$ values, henceforth called the *binding arguments*. The BindAccess and the BindJoin are implemented following the iterator model, like other LeSelect operators. A BindJoin can have an arbitrary operator tree as a left-hand child, but only BindAccess operators are allowed as the right-hand child. As shown in [6], this choice does not restrict the set of queries that we can execute. Thus, the BindAccess provides an iterator envelope around a

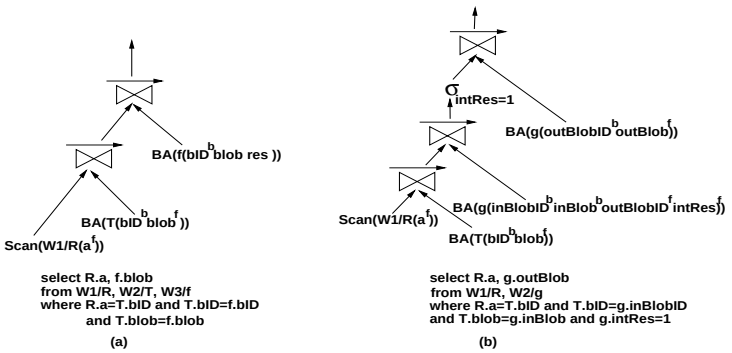

Figure 4: Sample QEPs using BindAccess and BindJoin.

restricted resource access, while the BindJoin provides binding arguments and collects the access results.

Figure 4 shows two sample queries involving data and programs, and possible corresponding QEPs. The query in Figure 4(a) chains two BindJoin operators, the first one fetching a blob, and the second one calling function f on this blob, where f returns a small-size result. Function g in Figure 4(b) takes as input a small attribute and returns a blob and an integer value $intRes$; its binding pattern set is $\{in^b outBlobID^f intRes^f, outBlobID^b outBlob^f\}$. The selection on $intRes$ may eliminate some of the blobs produced; the remaining ones are needed in the output. The blob is fetched directly where it is needed, via the last BindAccess-BindJoin pair.

In LeSelect, a restricted resource is *always* exploited by a BindJoin-BindAccess pair; two important particular cases are program invocation, and access to a blob by its blobID. In this last case, since the BindAccess uses the special binding pattern for blob access $blobID^b blob^f$, such a BindJoin-BindAccess pair always retrieves a single blob attribute. Thus, the most important components of a query's processing cost, namely program invocation and blob transfer, are performed using the BindJoin and BindAccess operators. Therefore, these operators are a good target for incorporating the optimizations described in Section 1, without requiring any change to the remaining operators.

3.4 BlobManager for storing and transferring blobs

We now describe the BlobManager (BM) module. The main role of the BM of any site S is to provide an uniform interface to any blob for all operators running on S, whether the blob has been published or produced by a program, on site S or elsewhere. To achieve this, the BM is in charge of storing and managing the blobs produced by a program running on S, or transferred from other sites to S in order to be consumed by programs running on S. Additionally, the BM of a site optimizes the transfer of blobs, by choosing where to transfer a blob from, if several copies the blob exist.

When a BindJoin operator running on S_1 indirectly invokes a program that produces a new blob, the BindJoin issues a *store* call to the local BM to require the storage of this blob. The parameters of the *store* call are the *blobID* of the blob and the *id* of the query within which the BindJoin is executed. The BM allocates a local blobID to any blob it stores which uniquely identifies a blob in BM's site.

To access a blob, an operator running within a query Q at site S, proceeds in two steps. First, the operator issues a *getLocalID(blobID,Q)* call to the BM of S, where *blobID* is the system-wide identifier of the blob. In a first step, the BM obtains a copy of the required blob (if it was not already available on S), and returns a *local blobID* to the operator requiring the blob. In a second step, the operator uses the local blobID to read the blob. This way, the BM acts as an intermediary, hiding the original storage of a blob to the operator that requires it, as well as the possible communications involved in retrieving it.

As an optimization, we might allow copied or produced blobs to persist in the BM of S not only for the needs of the BindJoin which has issued the corresponding *getLocalID* or *store* call, but also for other BindJoins, either part of Q's QEP, or executed in a different query. Suppose that a blob has been copied from S_1 to S, and is needed later on S_2. When a BindJoin on S_2 issues a getLocalID call to the BM on S_2, this BM has the choice of transferring the blob to S_2 either from S or S_1. This choice may be interesting for several reasons: if getting a blob from its original repository costs money, and if site S allows us to exploit its copy for free, the benefit is obvious; also, at runtime, network bandwidth estimates may suggest that $S \rightarrow S_2$ transfers are faster than $S_1 \rightarrow S_2$, or that the transfers should be split in two, half from the original source and half from the temporary copy of the blob. To do this, we parameterize a BindJoin requiring a blob with the list of BMs in which the blob has been stored for the purposes of the current query. The BindJoin provides this list to its local BM, which may operate the choice.

Lifespan of blobs in the Blob Manager In general, the choice of whether and how long will the BM of S store a blob b it has acquired from another site for the needs of the query Q is determined statically by the query optimizer. This decision is guided by metadata published by the site S, specifying its storage policy. Several such policies may be envisioned, ranging from fully egotistic (S keeps a blob only as long as it uses the blob), to fully cooperative in single-query mode (all sites involved in a query Q keep any blob they copy until the end of Q) to fully cooperative in multi-query mode (blobs transferred for the needs of a query are kept to profit to other queries, too). The choice of a particular policy for a site S is mainly determined by the available space at S, but may also depend on other considerations, e.g., service or data subscriptions among users on different sites, query priorities etc.

In this paper, we consider that all sites store all blobs they use to answer Q until the execution of Q ends. Note that all blob manipulations are done by BindJoin operators; following this line, the *close* method of a BindJoin executed within the QEP of Q sends a $mayDispose(Q)$ message to the local BM, informing it that the blobs stored for the usage of Q are no longer needed. Since the *close* call is propagated from the top of the QEP, we

know that all operators above the BindJoin have finished processing tuples when a BindJoin receives the *close*.

We make here the following remarks. First, keeping b until the end of Q's execution is in some cases more than what is needed. For example, if a blob is produced by a BindJoin on a site S and is never used by another operator up in the query plan, the blob does not need to be stored on S after the BindJoin has produced it. For simplicity, we consider that blobs are kept until the end of Q's execution, and make the assumption that enough storage space is available to this purpose. Second, while sharing transferred blobs among several queries is interesting and feasible, a complete study of query processing in this framework is out of the scope of this work.

4 Algorithms for Bind Join and Bind Access

We now define algorithms for BindJoin and BindAccess that can take advantage of caching, independent parallelism, and adaptive intra-operator parallelism to improve the query response time.

4.1 Cache-based optimization

The main justification for caching is to avoid redundant access to a restricted resource, when the left input of a BindJoin operator provides duplicate arguments. This can significantly improve query response time in the presence of duplicates, as long as the tuple input rate in the BindJoin operator is faster than the processing speed of the BindAccess. Also, under the same condition, the presence of duplicates allow us to significantly improve the BindJoin's tuple output rate *early* in the execution.

The presence of a cache requires specific design decisions to suit our distributed context. The first decision concerns the cache localization: to reduce data transfer, it has to reside in the same site as the BindJoin. Indeed, suppose that the BindJoin and its left child operator run on a site S_1, while the BindAccess runs on another site S_2. Obviously, by caching results on S_1 we avoid sending to S_2 arguments for which the result was already computed. This decision impacts on the physical architecture of the BindAccess operator. Since the BindJoin controls the cache, it is the BindJoin that extracts tuples from its left child operator, while the BindAccess has to obtain binding arguments from these tuples, through the BindJoin. As a result, we have to provide the BindAccess with the capability of extracting argument data from a (possibly remote) data structure filled by the BindJoin. Therefore, we decompose the BindAccess into two physical operators named *ComputeResult* and *GetBinding*; ComputeResult will run on S_2, GetBinding on S_1. Following the general principles of the distributed iterator model, the arguments are transferred between these operators through a distributed Exchange.

To explain the functioning of the physical operators for BindAccess and BindJoin, we use the following notation. The left child of the BindJoin is denoted as r, and its parent operator as p. A tuple coming from r is of the form (x_i, z_i); for the purpose of the explanation, let

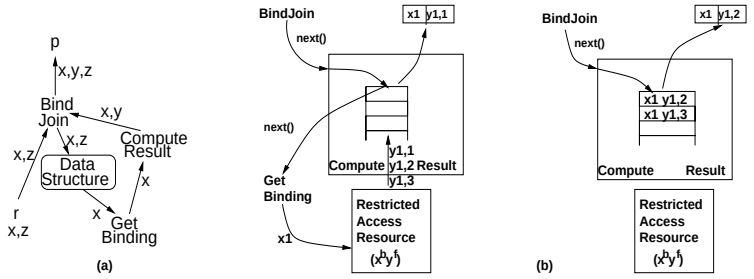

Figure 5: Physical operators for BindJoin and BindAccess (a); the ComputeResult operator (b).

us denote by x_i a tuple of values for all the binding arguments, and by z_i all the remaining attributes. The results of a resource access with the binding argument x_i is a set of $(x_i, y_{i,j})$ tuples, where each $y_{i,j}$ represents one result tuple returned for the arguments value tuple x_i. The outline of a physical QEP for a BindJoin-BindAccess pair is given in Figure 5(a).

The role of ComputeResult is to encapsulate the access to the restricted resource, as illustrated in Figure 5(b). On a *next*() call issued by the BindJoin, ComputeResult has to return a result tuple. To achieve that, it needs to (1) obtain a binding argument and (2) invoke the *callResource* method provided by the restricted resource publisher, as explained in Section 3.3. To obtain the argument, ComputeResult issues a *next*() command to the GetBinding operator. This operator, in turn, extracts the argument attributes from one of the tuples accumulated in the data structure by the BindJoin. Note that in the case where the call to the restricted resource returns several tuples, the ComputeResult is in charge of managing those tuples, in order to return them one at a time, transparently to the BindJoin. In Figure 5(b), ComputeResult simply returns a tuple resulted from a previous computation to answer the second *next*() call. Furthermore, for a given argument x_i, ComputeResult will return first all resulting tuples, and then a special "end-of-call" tuple, informing the BindJoin that there are no more results to be obtained for the x_i argument value.

We interject at this point the following remarks. First, ComputeResult is the only *physical* operator that a wrapper must implement; GetBindings and BindJoin run within the LeSelect server. Second, a ComputeResult iterator is quite generic, and a pre-defined wrapper implementing it can easily be devised using the *callResource* interface provided by the publisher.

We now discuss the design of algorithms for the physical BindJoin operator using a cache. For any execution time, such a BindJoin operator contains: a cache of argument-result pairs $(x_i, y_{i,j})$ for all tuples from r that have already been processed by ComputeResult, and a cache containing tuples received from r but whose processing is not finished yet. Also, when processing an (x_i, z_i) tuple for which several $(x_i, y_{i,j})$ are in the cache, the BindJoin has several $(x_i, y_{i,j}, z_i)$ tuples to output. To answer *next*() calls, the BindJoin maintains an internal result queue Q_1 from which it transparently returns these tuples one at a time.

The CacheFIFO BindJoin algorithm We start by presenting a simple synchronous BindJoin algorithm using cache. The executions of BindJoin, ComputeResult and GetBinding are serialized: each *next*() call is blocking for the caller. In CacheFIFO, at any moment, the data structure contains at most one (x_i, z_i) tuple, which has been received from r and whose processing is not yet finished ; we call this tuple the *current* tuple. When execution begins, the data structure is empty.

In response to a *next*() call, the BindJoin returns a result tuple from Q_1. If Q_1 is empty, the algorithm iteratively runs one or another of the following s_1 and s_2 steps, until some result tuples are obtained.

The choice of whether to execute a step s_1 or a step s_2 depends on whether a current tuple is present in the data structure or not. Step s_1 tries to obtain result tuples corresponding to the current tuple (x_i, z_i) by calling the *next*() method of ComputeResult. If ComputeResult returns a tuple of the form $(x_i, y_{i,j})$, the algorithm inserts it into the cache, and returns the tuple $(x_i, y_{i,j}, z_i)$. If ComputeResult returns an "end-of-call" mark, the current tuple is removed from the structure, and step s_1 finishes without having produced a result. Step s_2 attempts to obtain results using a new tuple from r. First, the algorithm extracts the tuple $t = (x_i, z_i)$ via an $r.next()$ call. Two cases may arise, depending on whether the x_i value is in the cache or not. If x_i is in the cache, the algorithm constructs and inserts into Q_1 one $(x_i, y_{i,j}, z_j)$ tuple for each $(x_i, y_{i,j})$ tuple found in the cache, and outputs the first such $(x_i, y_{i,j}, z_i)$ tuple in response to *next*(). Otherwise, if x_i is not in cache, t is inserted in the data structure, and becomes the current tuple; step s_2 has not produced any result.

Output rate of the CacheFIFO algorithm When the execution of this simple BindJoin algorithm starts, the cache is empty. The processing of most of the tuples received from r will entail a call to ComputeResult; thus, the tuple output rate is close to the processing rate of this operator. As execution continues, the cache is progressively filled, and tuples are output directly from the cache. Since we assume that cache lookup time is negligible compared to ComputeResult's processing time, the output rate towards the end of the execution is significantly higher than at the beginning. Such an uneven output rate is unavoidable in the presence of cache, but tuple bursts *at the beginning* of the execution are much better than a sudden burst *at the end.* (If the BindJoin is the top QEP operator, tuples are returned to the user faster; if its parent operator receives tuples fast, accelerating the early output of the BindJoin could propagate early tuples towards the output.)Because of this fact, we present a second algorithm that aims at a fast output rate in the early stages of the execution.

The CacheParallel algorithm If tuples can be obtained from r faster than the processing of ComputeResult, cache and duplicates provide three opportunities to improve the early output rate of the BindJoin,

1. First, for each $t = (x_i, z_i)$ incoming tuple such that an $(x_i, y_{i,j})$ is already in the cache, $(x_i, y_{i,j}, z_i)$ may be output directly by the BindJoin, and in parallel with the on-going processing in ComputeResult. This improvement comes at the price of losing input

order. Also, it requires the ability to temporarily store tuples within the BindJoin, since we cannot "directly" obtain from r's output tuples for which the results are already in the cache.

2. Second, by allowing tuples coming from r, for which the result was not already in the cache, to accumulate within the BindJoin's data structure, we increase the chances that the BindJoin outputs several tuples together. Whenever the BindJoin receives an $(x_i, y_{i,j})$ from ComputeResult, it identifies *all* (x_i, z_k) tuples in the data structure, and outputs all the $(x_i, y_{i,j}, z_k)$ at once.

3. Finally, GetBinding may *choose* which arguments to provide to ComputeResult in a way that maximizes the output tuple burst. To this purpose, GetBinding picks the most popular x_i value in the data structure, i.e., the one corresponding to the biggest tuple group in the data structure.

The cornerstone of our algorithm that implements these three requirements is a *cache buffer* storing all the information needed at any given moment of the execution (the cache and the tuples received from r and not yet fully processed). The trick is to allow different entities to run in parallel synchronizing their actions through the cache buffer. More formally, we specify a cache buffer by four access functions $(Xset, state, Y, Z)$, and two management rules r_1 and r_2. We first present the access functions.

- At any moment, $Xset$ denotes the set of x *values* that are currently present in the cache buffer.
- For any value x among those returned by $Xset$, the $state(x)$ function returns the state of this value, which can be either *done*, *running* or *waiting*. An x value is *done* if it has been fully processed, i.e., ComputeResult has returned to the BindJoin all the results for this value. The value is *running* if it has been chosen by GetBindings to be processed, but it is not yet done. Finally, a value of $Xset$ that is neither done nor running is *waiting*.
- GetBinding can only choose waiting values.
- For any x_i element of $Xset$ such that $state(x_i)$ is done, the function $Y(x_i)$ returns the set of $(x_i, y_{i,j})$ result *tuples* obtained by accessing the restricted resource with the argument x_i.
- For any x_i element of $Xset$, the function $Z(x_i)$ denotes the set of *tuples* of the form (x_i, z_i) present in the cache buffer.

Rule r_1 governs tuple insertion. An (x_i, z_i) tuple obtained from r can be inserted in the cache buffer only in one of the two following situations. If $x_i \notin Xset$, then this value has never been seen before in the BindJoin; the tuple is inserted, and state(x_i) on the insertion of this tuple becomes waiting. Otherwise, if $x_i \in Xset$ but state(x_i)$\neq$ done, the tuple is also inserted in the cache buffer; this insertion does not influence the value of state(x_i).

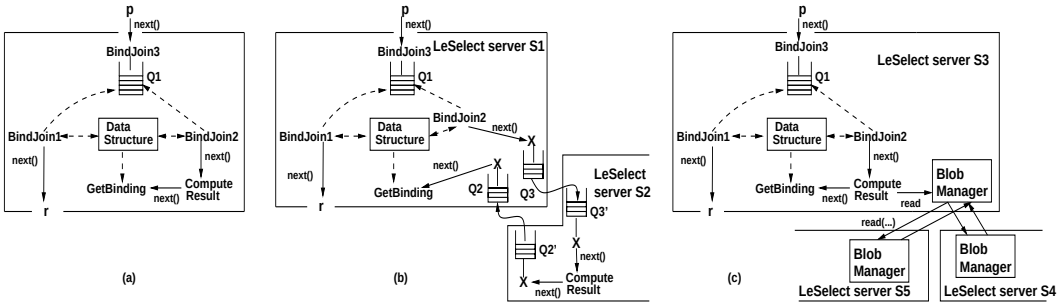

Figure 6: Architecture of the CacheParallel variant of BindJoin: local ComputeResult (a), remote ComputeResult (b), blob fetching (c).

Rule r_2 concerns tuple removal. A tuple $t = (x_i, z_i)$ is removed from the cache buffer if the state of x_i changes from running to done. This state change entails the following actions. First, for each $(x_i, z_i) \in Z(x_i)$ and for each $(x_i, y_{i,j}) \in Y(x_i)$, an $(x_i, y_{i,j}, z_i)$ tuple is sent to the output; second, all $(x_i, z_i) \in Z(x_i)$ are eliminated from the cache buffer.

To achieve a maximum of parallelism, we share the cache buffer management between three independent entities, running in parallel. The first one, noted BindJoin_1, performs two actions: first, it extracts r tuples, then for each incoming tuple $t = (x_i, z_i)$ BindJoin_1 tries to insert t into the cache buffer following rule r_1. If the insertion is not possible, meaning that x_i is in $Xset$, and its state is done, then for each element $y_{i,j} \in Y(x_i)$, it outputs a result tuple $(x_i, y_{i,j}, z_i)$. The second entity, noted BindJoin_2, is in charge of extracting and handling the results produced by ComputeResult. The results are grouped by their x value. For a given x_i value, BindJoin_2 enters the corresponding $y_{i,j}$ results one by one into the cache buffer. When ComputeResult finally returns an "end-of-call", BindJoin_2 changes the state of x_i from running to done and applies rule r_2. Finally, a third entity noted BindJoin_3 desynchronizes the BindJoin from its parent operator p.

Besides BindJoin_1 and BindJoin_2, the cache buffer is also accessed and updated by GetBinding. When GetBinding chooses an x argument, it changes its state from waiting to running. As shown in Figure 6(a), when the BindJoin and ComputeResult run on the same site, GetBinding, BindJoin_2, and ComputeResult can run synchronously without degrading performances. When BindJoin and ComputeResult run on two different sites, GetBinding must be run in parallel with BindJoin_1 and BindJoin_2 (see Figure 6(b)). Nevertheless, it is convenient to limit drastically the capacity of exchange queues between GetBinding and ComputeResult. Indeed, allowing too many elements in the queues leads to choosing x values a long time before they are effectively processed, making the choice non optimal.

Figure 6(c) illustrates the usage of the algorithm for fetching blobs from remote sites. Remember that our goals were first never to transfer the same blob on the same path twice, and second to enable the retrieval of a blob from one or several sites where a copy exists. The first goal is trivially met by the cache-aware BindJoin: the x arguments in this case

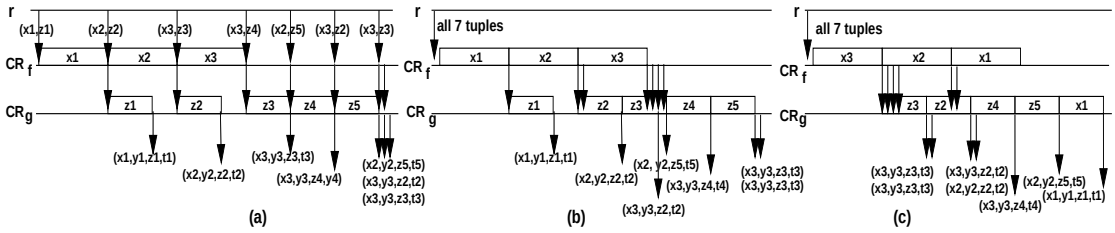

Figure 7: CacheFIFO (a) and CacheParallel: choose oldest value (b), choose most popular value (c).

are global blobIDs, and the BindJoin sends only distinct values to ComputeResult. All operators run on the BindJoin site; the physical localization of the source blob is considered not at the operator level, but in the physical blob transfer layer as follows. ComputeResult issues *getLocalID* commands to the BM on site S_3. As explained in Section 3.4, this BM was informed, on BindJoin.*open*() that a blob published on S_4 has already been copied, within the current query, on S_5. Thus, it can decide, on a per-blob basis, using money and time cost considerations, where to fetch the blob from.

Output rate of the CacheParallel algorithm The algorithm improves the output rate early in the execution. Figure 7 illustrates the behavior of a QEP fragment consisting of two BindJoins: $(r \overrightarrow{\bowtie} BA(f(x^b y^f)) \overrightarrow{\bowtie} BA(g(z^b t^f))$. In this example, the per tuple costs of accessing restricted resources f and g are set to $c_f = 5$ and $c_g = 3$ respectively, and both resources return exactly one tuple per argument; different indices for x, y, z, t indicate different values. Arrows follow tuple flow among r, CR_f (the ComputeResult operator accessing f) and CR_g; we assume all seven tuples can be extracted from r as fast as desired.

When the BindJoins are implemented by the CacheFIFO algorithm (see Figure 7(a)), tuples from r are extracted only at the pace of the bottleneck restricted access (f, then g). Figures 7(b) and (c) show two CacheParallel variants. In this case, we consider that the extraction of tuples are from r is very fast, so that all seven tuples are accumulated within the first BindJoin before the execution starts. It can be seen that CacheParallel helps solve some synchronization problems (tends to eliminate g's idle time in the first phase of the execution), and that unlike CacheFIFO, it produces tuple bursts early in execution. As Figure 7 shows, choosing the most popular value provides a bigger early output rate than choosing the oldest. But the biggest advantage of this choice policy is that it is a *stable* technique, capable to cope with various input orders, even if the duplicates come last, as in figure 7. Also, since in a chain of BindJoins we allow each to choose its own processing order, the decisions of the first BindJoin do not imply a bad ordering for those that follow.

4.2 Introducing adaptive intra-operator parallelism

In this section, we show how to adapt the operator architecture for BindJoin and BindAccess to allow several parallel accesses to a restricted resource. We then provide a simple algorithm for determining the degree of intra-operator parallelism that should be used within a BindJoin-BindAccess pair. Our algorithm is designed to exploit the full capacity of the resource, while sharing it equally with all other queries using the same resource at the same time; also, it dynamically adapts to changes in the per-call execution time, which may be due to unpredictable parameters such as the machine load on the site executing a program, the network congestion, etc.

To allow N parallel accesses to a resource, we simultaneously run N instances of GetBinding, ComputeResult, and BindJoin_2. All GetBinding and BindJoin operators consult and update the cache buffer; all BindJoin instances produce in Q_1, multiplying the chances of large output packets. In the case of a remote ComputeResult, all instances consume their input from Q'_3 and produce in Q'_2. When running several instances of GetBinding, we use locking on the data structure's entries to avoid race conditions, whether ComputeResult is local or remote. From an implementation point of view, the *open*() call of BindJoin creates one instance of BindJoin_1, which supervises the execution and decides on the appropriate number of BindJoin_2 instances to create. In turn, each created instance of BindJoin_2 *open*s its ComputeResult and GetBinding operators.

Determining the right degree of parallelism Consider the case when a single BindJoin operator uses the restricted resource: its goal is to determine the maximum degree of parallelism possible. We propose an algorithm which greedily runs at any moment as many instance as possible. The execution proceeds in stages; at the end of each stage, BindJoin_1 decides on the number of instances to run in the next stage.

In the first stage, one instance is run. In the second stage, the BindJoin tries to execute two instances in parallel, and measures their average per-tuple execution time avg. To decide whether to increase, decrease, or keep the same number of instances, the BindJoin compares avg with $incTreshold{*}t_{min}$ and $decTreshold{*}t_{min}$. t_{min} is the minimum execution time, per instance and per tuple, of a call to the restricted resource: it can be declared by the publisher, or may be recorded over previous executions. The parameters $incTreshold$ and $decTreshold$ must be greater than 1.0, and satisfy $decTreshold > incTreshold$. If $avg < incTreshold * t_{min}$, then the last recorded average time is quite close to the best measured one, and BindJoin_1 decides to add an extra instance in the next stage. If $avg > decTreshold * avg_{min}$, the last average time is too far from the optimum, and BindJoin_1 eliminates an instance in the next stage, trying to attain again better average running times. If avg falls within the window $[avg_{min} * incTreshold, avg_{min} * decTreshold]$, we keep the same number of instances, since neither increasing nor decreasing it seem interesting choices. If this window is too narrow, the number of instances oscillates between some value n and $n + 1$; if the window is too wide, the algorithm reacts too slowly to correct a bad decision. This algorithm greedily runs as many instances as possible.

Fair sharing the access to a restricted resource We now address the problem of fairness in sharing a parallel resource among several BindJoins. We can do this simply by modifying the algorithm above as follows. For every restricted access published on S_1, the execution engine stores the number of parallel instances running for each BindJoin at a given moment. When a BindJoin decides, following the window scheme presented above, that parallelism is profitable and that it needs to add an extra instance, it may do so only if its current number of instances is the lowest among all the BindJoins concurrently exploiting the same resource. Similarly, whenever a BindJoin decides to eliminate an instance, it has to verify, by inspecting everybody's counters of running instances, that its number of instance is the highest. This simple "politeness" mechanism ensures both maximal exploitation of a restricted resource and equitable repartition of instances among all BindJoins running in parallel, as we show in the next section.

5 Experimental assessment of the CacheParallel algorithm

In this section we evaluate the behavior of the CacheParallel BindJoin algorithm. We are mainly concerned with two issues: first, the tuple output rate in the early stages of the execution, and second, its capacity to adapt its degree of parallelism to changes in the processing cost of a tuple.

Improving early tuple output rate Figure 8 presents the tuple output rate from QEPs corresponding to a query of the form "select $f(r.x)$, $g(r.z)$ from r". We take $c_f = 10$ time units and $c_g = 100$, and assume for simplicity that f and g always return a single tuple. The cardinality of $r(x^f z^f)$ is $10,000$, tuples arrive in no particular order and accumulate in the first BindJoin before the beginning of the execution. We neglect cache lookup and tuple transfer time, since they simply add up to execution times for all algorithms. We consider two QEP's for the query: a QEP where f precedes g and another where g precedes f. We suppose that the two $\overrightarrow{\bowtie}$s are implemented using the same physical algorithm in a given QEP. We consider three implementations of the $\overrightarrow{\bowtie}$s: CacheFIFO, CacheParallel with GetBinding choosing the oldest value, and CacheParallel with GetBinding choosing the most popular. We vary the data distributions of x and y, and the order of the two BindJoin operators.

The top left graph corresponds to the QEP $r \overrightarrow{\bowtie} BA(f(x^b y^f)) \overrightarrow{\bowtie} BA(g(z^b t^f))$; x and y follow independent uniform distributions. The CacheFIFO output rate is dictated by g, until t $\simeq$ 100,000, when g has finished evaluating all distinct z values; from this point on, tuples are output from the cache. In contrast, the CacheParallel variants achieve a much better output rate. Since f runs ten times faster than g, it immediately outputs (x, y, z) if x is in the cache, and outputs tuple packets whenever possible. Thus, g's data structure is filled, and it may choose the most frequent z value for processing.

The advantages of CacheParallel are smaller in the top graph at right; since there are many z values, the bottleneck in query execution is always g. However, the CacheParallel

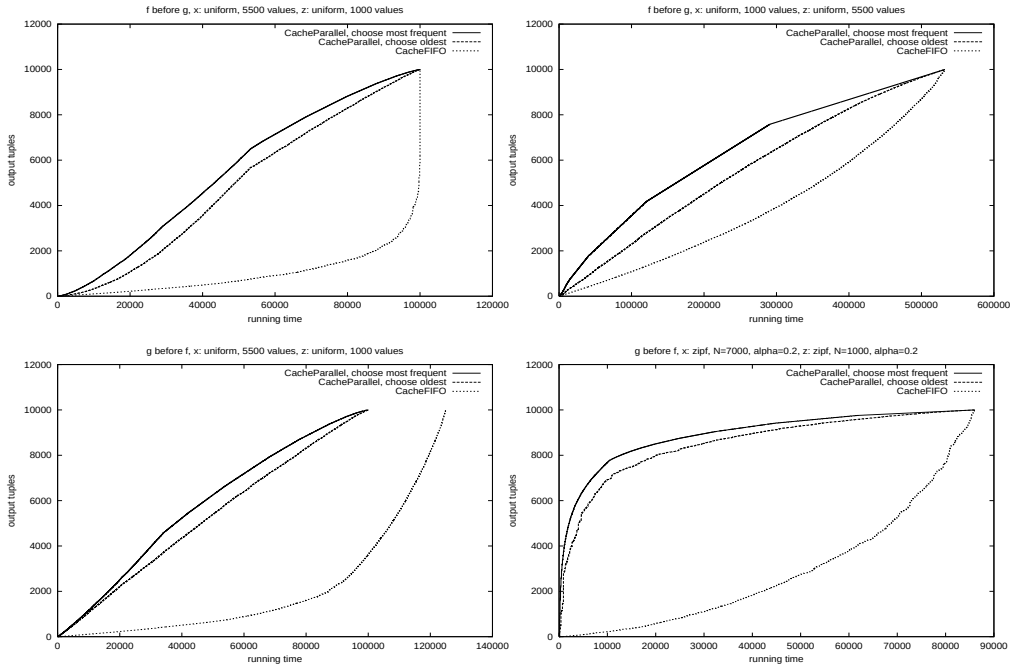

Figure 8: Sample tuple output rates.

variants achieve some improvements by allowing parallel passage of duplicates and packets of tuples sent to the output. The slope of the CacheParallel "choose most popular" variant reflects the size of the tuple packets that g is able to output (4, then 3, 2, 1); since f is ten times faster, g has at any point at least 10 tuples to choose from, but there are few very frequent values in z.

In the bottom left graph, the CacheFIFO BindJoin chain performs badly since g is the bottleneck for a long time, keeping f mostly idle; only when g has processed all unique values of z does the output rate grow. In contrast, CacheParallel maintains a steady rate during the entire execution, and reduces the running time by 20% since f is kept busy by the packets output from g. In the previous two graphs, f and g were well synchronized, and the total running time was the maximum of the running times of the two BindJoins considered in isolation; in this third graph, synchronization problems are eliminated and pipeline parallelism achieved by the CacheParallel's optimizations.

The graph at bottom right in Figure 8 shows very important gains of CacheParallel with respect to CacheParallel, since x and z now follow *zipf* distributions, with $\alpha = 0.2$. By picking the most popular values early, large tuple packets are sent very fast to the output by both BindJoins.

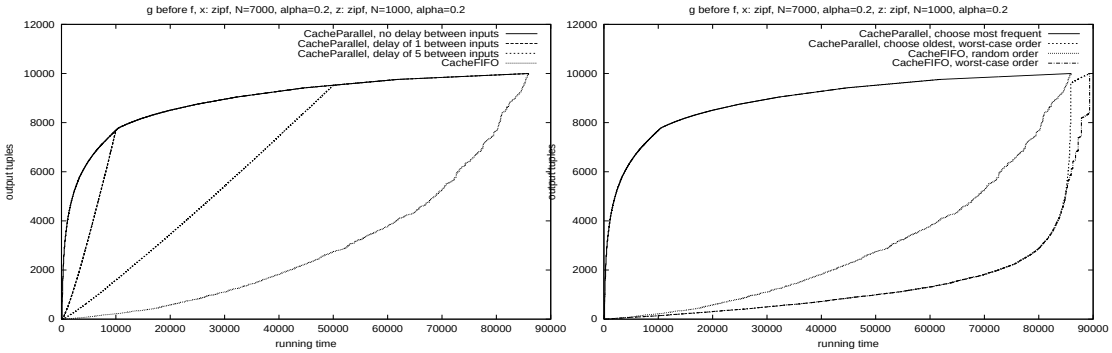

Figure 9: CacheParallel behavior in the presence of limited input rate (left) and bad data order (right).

We explain the small difference between the two CacheParallel variants in Figure 8. In the case of uniform distributions, in the absence of very popular values, picking the most popular value cannot improve a lot over picking the oldest one, since there are no big output bursts anyway. In the case of zipf distributions, since the very popular values are scattered over the 10,000 tuples, the BindJoins are sure to process those frequent values quite early, even by picking the oldest value; afterwards, subsequent copies of popular values are output directly from the cache by both CacheParallel variants.

In Figure 9 at left, we evaluate the impact of a limited input rate on the CacheParallel output rate (the "choose most frequent") variant. The data and function parameters are unchanged from the last graph in Figure 8. We show two extra curves, corresponding to two input rates. One curve corresponds to an input rate of one tuple per time unit, and the other to an input rate of one tuple per five time units. When the curves follow a linear slope, all tuples have not been received yet, and the input rate is the only limitation of the output rate. Note that the incoming tuple rate (e.g., 1/5) is in theory better than both function rates (1/10, and 1/100 respectively); however, CacheParallel is able to exploit the duplicates to produce tuples much faster than the "nominal" rate of the two BindJoins. When all tuples have been received (at t = 10,000 or t = 50,000), the curves join the one without input rate limitations. We note that even if only two tuples can be received during one execution of f, the CacheParallel output rate is still much better than CacheFIFO.

The graph at right in Figure 9 compares the effect of the incoming data order on several BindJoin algorithms. Data and function parameters are the same as above, except for the curves labeled "worst-case": they correspond to the same tuple set sorted in the increasing number of value frequencies, that is, the first tuples contain mostly singletons, and the most frequent values come last. We show a single curve for CacheParallel choosing the most frequent value, since, not surprisingly, by choosing among the whole accumulated tuple set, it is not affected by data order. CacheFIFO in the worst-case order behaves worse than in the case of a random order. The most interesting observation in this worst-case measure is that the CacheParallel variant choosing the oldest value is as bad as CacheFIFO most of

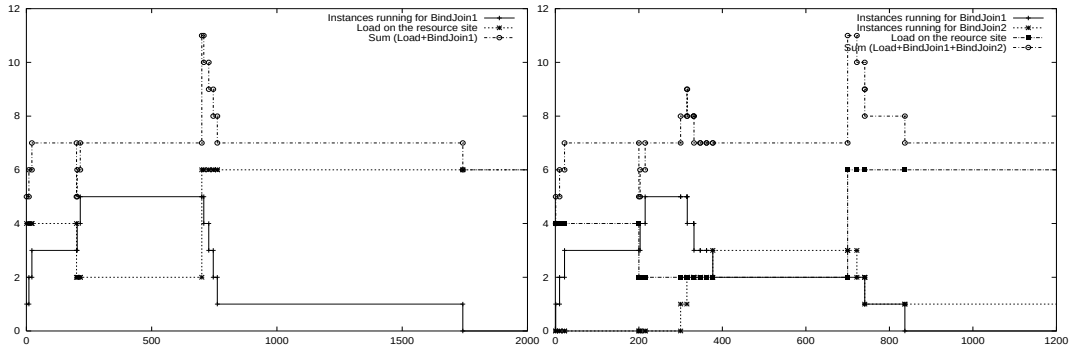

Figure 10: Sample behavior of the adaptive algorithm for determining the optimal parallelism degree.

the time, since old values are very infrequent. We conclude that CacheParallel, even with limited tuple input rates and disadvantageous incoming data order, provides much better output rates than CacheFIFO.

Adaptive tuning of the degree of parallelism We now describe experiments that validate the adaptive algorithm for choosing the optimal degree of parallelism in exploiting a restricted resource. We consider that before execution we know (from having measured previous executions) that $t_{min} = 10$. We assume the following behavior from the restricted resource: up to the optimal (unknown) degree of parallelism $opt_{\|}$, any instance processes one tuple in 10 time units. If n, the number of instances run, is greater than $opt_{\|}$, the processing time of a tuple per instance becomes $t_{min} * opt_{\|}/n$. For $t_{min} = 10$, we obtained good results with $incTreshold = 1.2$ and $decTreshold = 1.5$. We assume that if the machine where the resource is located were idle, we could run 7 parallel instances with good performance; if the machine is loaded, this number diminishes.

The curve at left in Figure 10 shows how a single BindJoin adapts its number of running instances to the machine load. At the beginning, the load on the machine is equivalent to 4 running instances; the BindJoin starts one, then two, then three instances, using the resource to a maximum. When the load decreases (at $t \simeq 200$), the BindJoin measures decreased avg times, and adds two extra instances. When the machine load increases to 6, the BindJoin adapts and restricts itself to only one running instance.

At right in Figure 10, we demonstrate how two BindJoins adapt to the machine load and to the presence of each other. The load starts at 4, then varies successively to 2, 3 and 7. The first BindJoin starts at $t = 0$ and runs three, then five, parallel instances, until the second one starts at $t = 300$. When the BindJoins share the resource, they share it evenly; the first BindJoin finishes execution at $t \simeq 1600$, and from that moment the remaining one exploits the resource on its own, since it knows it no longer has a concurrent.

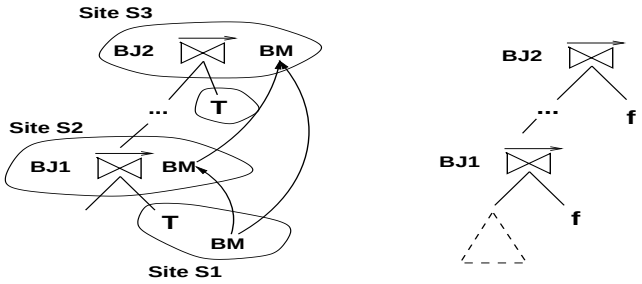

Figure 11: QEPs involving blob transfer (left) and blob production (right).

6 Query execution in the presence of space limitations

In our discussion, we have not considered space limitations on any LeSelect site. In particular, we assumed that the results of any program execution, as well as the argument-result cache pairs, are kept on the program's site at least until the end of the query that produced them. Also, we assumed that whenever a BM is asked to retrieve a blob from a different site, the necessary storage space is available.

At any moment, a given site has a certain amount of space available for a given query, which varies following the needs of the operators running on that site. We consider this space to consist of memory and disk taken together, and we do not investigate when is memory used rather than disk, or vice versa. We assume that when memory runs out on a given site, operations are continued using disk space; in this section, we investigate the consequences of disk space limitations on query execution.

Space limitations may affect query execution in several ways. First, if the space available for the data structure in a CacheParallel BindJoin is limited, the BindJoin will stop fetching new arguments from its left child until some space is freed; thus, no special measure is necessary. Second, in the case of a BindJoin (either CacheFIFO or CacheParallel) producing small results, when the argument-result cache is full, some cache entries are dropped, e.g., according to an LRU policy. In this case, the same result may be computed many times during the execution of the query. Special care needs to be taken to ensure queries involving blobs are correctly executed in the presence of space limitations, as we explain next.

Blobs and space limitations To understand the issues involved, consider the QEPs shown in figure 11.

In the QEP at left, BJ_1 commands the BM of S_2 to retrieve blobs from S_1. The second BindJoin BJ_2, executed on S_3, needs the same blobs; as described in section 3.4, we had allowed the BM of S_3 to choose where to retrieve blobs from. In our case, the choice is between the BMs of S_1, respectively S_2. However, if on S_2 the disk space is limited, the BM on S_2 may decide to discard some transferred blobs. Thus, the BM on S_3 may no longer be sure that all blobs can be found in both locations. Therefore, in the presence of space

limitations for blob storage, this blob transfer optimization might be restricted, in the sense that some temporary blob sources are ignored, and blob retrievals are more likely to be made from the original location.

The QEP at right in figure 11 exhibits another potentially dangerous case. BJ_1 applies the function f to some arguments; for each tuple of argument values, f produces a blob and an integer. The blobID and the integer are sent from BJ_1 upwards in the QEP, and upper in the query plan BJ_2 tries to fetch the produced blobs, using the blobIDs. If there are several BindJoins consuming the blob produced by BJ_1, consider BJ_2 to be the highest one in the QEP. There are two potential pitfalls in this case.

First, if on the site of BJ_1 there is limited space, the BM may have discarded a produced blob before BJ_2 has the time to ask for it; this would entail a run-time error. Second, some of the blobs produced by BJ_1 may be useless, i.e., they are never retrieved by BJ_2, since all tuples containing their blobIDs are eliminated by some intermediary operators between BJ_1 and BJ_2. However, the BM storing produced blobs does not know which blobs are useless, and may pin useless blobs in its cache, while discarding useful blobs. Eventually, BJ_1 may be filled with useless blobs, and its execution stops for lack of space for new results, thus leading to a deadlock.

To avoid these pitfalls, we enforce some synchronization among the operators in the QEP, from BJ_1 to BJ_2. Obviously, some of the blobs must be discarded before the execution of BJ_1 resumes. A basic solution is the following. When the BM of BJ_1 has no more place for the query in which BJ_1 runs, we stop BJ_1's execution until all blobs it has produced may be discarded from the BM. To do this, we force the evacuation of the tuple processing chain between BJ_1 and BJ_2 as follows. The first *next()* call received by BJ_1 (where the associated BM is full) is answered with a special synchronization tuple. The following *next()* received by BJ_1 blocks until its BM signals that the space has been freed. Upon receiving a synchronization tuple, all operators in the chain from BJ_1's parent to BJ_2 must first finish processing their current tuples, then forward the synchronization tuple to the above operator. When BJ_2 receives this tuple, it must retrieve the blobs corresponding to all the blobIDs it has received before the synchronization tuple. Thus, the BM of BJ_2 sends to the BM of BJ_1 the necessary *read()* calls, followed by a special *read(null)*, interpreted by the BM of BJ_1 as "all blobs produced by BJ_1 may be discarded". At this point, BJ_1 receives the notification that space is now available, and may resume the execution. Since BJ_2 is the last operator in the QEP to need the blobs produced by BJ_1, upper operators do not need to receive the synchronization tuple; therefore, BJ_2 does not propagate it further. To this purpose, the synchronization tuple must be marked as addressed to BJ_2; this information is available at query compile time.

7 Discussion of related work and conclusion

Our work compares with the Mocha project [19], integrating data and programs; however, they assume that a program can be shipped to the site where its arguments are, while we consider programs that cannot be moved, thus addressing the extra difficulty of efficiently

parallelizing data transfer and program execution. Hash- and sort-based caching algorithms for expensive functions have been studied in [14]; sort-based algorithms do not apply in our context since it would be a performance loss to stop the pipelined query execution for a sorting step. They propose a hybrid hash scheme to ensure the cache fits in memory. We envision using cache as soon as the cost of a program invocation is more expensive than a cache lookup; also, the proposed hybrid hash can be adapted to our algorithm to efficiently manage the cache buffer. In [4, 13], query optimization in the presence of costly predicates is addressed, but unfortunately, the results developed there do not apply to our context. The reason is that "predicate ranking" [13] and the methods described in [4] are based on the function costs being *constant per tuple*, which no longer holds when using caching as we do.

Two recent papers use asynchronism in executing specific kinds of expensive functions: in [16], the network asymmetry in a client-server context is compensated for by issuing a batch of function calls, without waiting for the first one to complete. However, they do not consider multiple mediators and arbitrary placement of operators; also, caching and parallelism are not used. WSQ/DSQ [9] proposes a single mediator processes queries that can involve accessing Web sources. The authors note that logically independent HTTP calls can be issued in parallel, but no guidelines are given as to the optimal degree of parallelism. A single control operator, placed on the mediator, is responsible for matching the call arguments with their results; such an operator is infeasible in a fully distributed context. In our work, these tasks are split among the several actors running within a BindJoin-BindAccess pair. "Eddy" operators, introduced in [1] route tuples among join operators, giving more tuples to the faster running joins; thus, Eddie adapts the data flow to the processing speed. Unfortunately, Eddies cannot be incorporated into our framework, because it would be unpractical to have *all* tuples involved in a query transit, potentially many times, through a single LeSelect server. Our work is also related to existing results on improving the output rate for *part* of the query result, as described, e.g., in [18, 20]. However, our work is different in that we used parallelism, asynchronism and duplicates to improve the output rate of the distributed BindJoin operator. As directions of future work, we plan to extend our framework to sharing blob and function cache among several queries, and to processing of expensive programs taking as input whole tables.

References

[1] Ron Avnur and Joseph M. Hellerstein. Continuously adaptive query processing. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 261–272, 2000.

[2] L. Bouganim, F. Fabret, F. Porto, and P. Valduriez. Processing queries with expensive functions and large objects in distributed mediator systems. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 91–98, 2001.

[3] Surajit Chaudhuri and Kyuoseok Shim. Query optimization in the presence of foreign functions. In *Proc. of the VLDB Conf.*, 1993.

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

[5] D. Chimenti, R. Gamboa, and R.Khrishnamurty. Towards an open architecture for LDL. In *Proc. of the VLDB Conf.*, Amsterdam, 1989.

[6] Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 311–322, 1999.

[7] Ian Foster, Carl Kessekman, and Steven Tuecke. The anatomy of the Grid : Enabling scalable virtual organizations. *The International Journal of Supercomputer Applications*, 2001.

[8] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[9] Roy Goldman and Jennifer Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 2000.

[10] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 102–111, 1990.

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

[12] Laura Haas, Donald Kossmann, Edward Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proc. of the VLDB Conf.*, Athens, Greece, 1997.

[13] Joseph M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems (TODS)*, 23(2):113–157, 1998.

[14] Joseph M. Hellerstein and Jeffrey F. Naughton. Query execution techniques for caching expensive methods. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 423–434, 1996.

[15] http://www-caravel.inria.fr/LeSelect.

[16] Tobias Mayr and Praveen Seshadri. Client-site query extensions. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 1999.

[17] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proc. of the Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.

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

[19] Manuel Rodriguez-Martinez and Nick Roussopoulos. MOCHA: A self-extensible database middleware system for distributed data sources. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 213–224, 2000.

[20] T. Urhan and M. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proc. of the VLDB Conf.*, 2001.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399