Processing Queries with Expensive Functions and Large Objects in Distributed Mediator Systems

Luc Bouganim* ** Françoise Fabret* 
* INRIA Rocquencourt 78153 Le Chesnay France Firstname. LastName@inria.fr 
** PRiSM Laboratory 78035 Versailles France bouga@prism.uvsq.fr 
Fabio Porto* *** Patrick Valduriez**** 
*** PUC-Rio Departamento de Informática Rio de Janeiro, Brazil porto@inf.puc-rio.br 
**** University Paris 6 75005 Paris France Patrick.Valduriez@lip6.fr

Abstract

LeSelect is a mediator system which allows scientists to publish their resources (data and programs) so they can be transparently accessed. The scientists can typically issue queries which access distributed published data and involve the execution of expensive functions (corresponding to programs). Furthermore, the queries can involve large objects such as images (e.g., archived meteorological satellite data). In this context, the costs of transmitting large objects and invoking expensive functions are the dominant factors of execution time. In this paper, we first propose three query execution techniques which minimize these costs by taking full advantage of the distributed architecture of mediator systems like LeSelect. Then, we devise parallel processing strategies for queries including expensive functions. Based on experimentation, we show that it is hard to predict the optimal execution order when dealing with several functions. We propose a new hybrid parallel technique to solve this problem and give some experimental results.

1 Introduction

Internet as a global network makes it possible for scientists to share experimental data sources as well as processing programs. In the earth-science community, for instance, satellite images, pollution indexes, temperature tables, as captured by different instruments and satellites can now be shared between worldwide scientific groups. Sharing of processing programs is also very important. These programs, typically written in a 3GL language, implement image manipulation algorithms, weather index analyses and many other useful functions. However, sharing data sources and programs across many users through the Web may be very difficult because of the high cost of locating, extracting and using the relevant resources.

The solution we envision is to allow scientists to publish their resources (data and programs) to the community through a common mediator system [15]. Publication of a resource can be seen as its registration within the mediator system that will ease access to it. Thus, users of published resources are capable of combining local and remote resources transparently.

LeSelect [12] is a mediator system developed at INRIA, France, which supports the aforementioned capabilities. Client typically connects to a LeSelect server which can transparently access other servers. Resource publishing in LeSelect requires the construction of wrappers. Data wrappers map the structure and content of data sources to the LeSelect common data model, i.e., the relational model. Function wrappers interface a published program to LeSelect. The function wrapper converts incoming relational data into a format suitable for the program, invokes the program and converts the output data for further processing. Unlike previous mediators systems, LeSelect has a fully distributed architecture. Each participating site has a complete LeSelect server capable of publishing and accessing resources as well as processing SQL-like queries.

LeSelect was primarily motivated by two earth-science applications dealing with coastal zone management [14] and air quality models [7]. In one of the scenarios, scientists want to analyze pollution episodes that have already occurred in order to compare several air quality models. They typically issue queries which access distributed published data and involve the execution of complex programs. Furthermore, the queries frequently involve binary large objects (Blob’s) such as images. In this scientific cooperation context, the costs of transmitting Blob’s and invoking programs are the dominant factors of execution time. In this paper, our goal is to devise efficient query processing strategies which minimize these costs by taking full advantage of the distributed architecture of mediator systems like LeSelect.

The work done in the more general context of expensive functions is obviously related to ours. [8] describes three caching strategies which avoid redundant computation on duplicates. In [4, 5], expensive functions are modeled as virtual relations thus allowing to use traditional System R query optimization techniques. [9] propose a polynomial time algorithm to place expensive predicates (i.e., expensive functions with a predicate on the result of the function) in a query plan. [6] extend dynamic programming by analyzing different sequences of expensive predicates for the same set of joins. [13] focuses on executing expensive functions in a client/server architecture and proposes techniques to reduce
communication costs due to the low bandwidth between the client and the server. [12] proposes MOCHA, a middleware system that uses code shipping to pushdown functions to the data source's site in order to minimize network transmission.

In our context, the solutions proposed in [12, 13] are no longer applicable because they either require code shipping (which does not work for Fortran programs for instance) or are limited to a single centralized mediator, without considering the advantages of fully distributed query processing. The caching technique of [8] can be adapted to take into account large object transmission. The remaining techniques strive to produce the "best" query execution plan for queries involving expensive functions. They cover a large spectrum from simple functions, with a per-tuple execution time comparable to that of relational operators, to extremely expensive functions such as mathematical models. In the scientific cooperation context, considering the whole spectrum, in particular simpler functions, leads to unnecessary complexity. In this context it is reasonable to consider that the time to execute relational operators, including joins, and the time to transfer relational data are negligible compared to the time to process scientific programs and transport large objects. This assumption is quite realistic in the applications we consider and allows us to make the following contributions:

First, we propose three query execution techniques, namely DelayedTransfer, Caching and BackgroundTransfer, which minimize the costs of Blob transmission and expensive function invocation.

Second, we propose to fully exploit the inherent parallelism of the distributed architecture. We propose parallel processing strategies for queries including expensive functions. Based on experimentation, we show that it is hard to predict the optimal execution order when dealing with several functions. We propose a new hybrid parallel technique to solve this problem and give some experimental results.

The remainder of this paper is organized as follows. In Section 2, we present in more details the underlying architecture of LeSelect and describe more precisely the context. An illustrating example helps defining the problem. Section 3 describes the query execution techniques. Section 4 gives the query optimization strategy, focusing on the new hybrid parallel technique. Section 5 concludes.

2 Problem Formulation

In this section, we describe the context associated with scientific program publishing in LeSelect. Then we present a complete example which helps defining the problem.

2.1 Context

One of the key ideas of LeSelect is to allow publishing programs which have not been specifically designed to be published. The objective is, for instance, to allow scientists to publish existing complex programs (e.g., environmental models) without any rewriting. The programs that can be published are arbitrary programs which take input data and produce output data. Publishing a program requires the construction of a function wrapper that implements the logic for executing the program. In the following, the term program refers to a published program while the term function refers to its function wrapper.

In this paper, we focus on functions that take one tuple as input and produce one tuple as output. These functions iterate on the tuples of a given relation R and produce another relation R'. The schema of R' is the schema of R augmented with the output attributes of the function wrapper.

We only consider functions which must be executed in their publication site. This situation occurs in several cases [13]: (i) the program is written in a non-portable language, (ii) the program requires a special execution environment or internal data, (iii) the owner does not want to publish the program code or charges for the program usage. Thus, unlike code shipping strategies such as [12], we exclude the possibility of moving the program code to the site of its inputs.

The techniques proposed below apply to expensive functions. A function is said to be expensive as soon as its per-tuple execution time is several orders of magnitude higher than the per-tuple execution time of relational operators. As said in Section 1, we assume that the transmission cost of basic type data and the processing cost of relational operators can be neglected with respect to the costs of transmitting large objects and processing these expensive functions.

Finally, we assume that the different LeSelect servers are connected by links of roughly equivalent bandwidth and thus consider that the network per-byte transfer cost is the same between any two server sites.

2.2 Illustrating Example

Let us consider an earth-science application with cooperating scientists in Brazil and France. The query tries to correlate water pollution indexes with the vegetal cover percentage on data located in Rio de Janeiro. The data is published using two data wrappers which present it in the form of two relations P and V. P contains pollution measurements while each tuple of V consists of identifiers along with a reference to a raster image, stored as a binary file. The CV program published in Paris is written in Fortran and uses some image processing libraries in order to compute the vegetal cover percentage based on a satellite image. The CP program published in Sao Paulo computes a pollution index based on some pollution measurements. CP uses a mathematical model.

Figure 1 describes this example.

![Figure 1: An illustrating example.](image-url)
The scientists in Brasilia can send the following SQL-like query to the mediator system:

**Select** P.regId, CP(P.value), CV(V.image) **From** P, V
**Where** P.regId = V.regId and CP(P.value) > 1.5 and CV(V.image) < 0.3;

One possible query execution plan (QEP) for this query is to join relations P and V at the Rio site, apply CP at the Sao Paulo site, apply CV on tuples which satisfy the predicate CP(P.value) > 1.5 (denoted \( p_1 \)) at the Paris site and finally transmit tuples which satisfy the predicate CV(V.image) < 0.3 (denoted \( p_2 \)) to the original site in Brasilia. This plan is referred as QEP1 in the following and shown on Figure 2.

### 2.3 Problem Definition

We can now define the problem as follows. Given a distributed mediator architecture like that of LeSelect, the problem is to come up with query processing techniques to minimize the response times for queries involving expensive functions and large objects.

Query processing is classically done in two steps. The query optimizer first generates an “optimal” QEP for a query using a query optimization technique. The QEP is then executed within the distributed architecture using appropriate query execution techniques. Obviously, the query optimization technique and the query execution techniques are highly correlated since the former assumes the existence of the latter.

Our goal is therefore to devise query execution techniques which minimize response time by reducing data transmission, expensive function calls, and exploiting parallelism. Based on that, we must define a suitable query optimization technique which fully exploit the parallelism of the architecture.

### 3 Query Execution Techniques

In this section, we first present three basic query execution techniques to handle expensive functions, namely DelayedTransfer, Caching, and BackgroundTransfer. We then discuss the interdependencies between these techniques and their effect on the optimization of queries with expensive functions and Blob’s.

### 3.1 Avoiding Useless Blob Transmission

In the naive execution of QEP, described above, the images of V are transferred first to Sao Paulo and then to Paris. Furthermore, the images associated with the tuples that do not satisfy predicate \( p_2 \) are transmitted to Sao Paulo.

Since we consider equivalent network links between all LeSelect servers, there is no interest in transmitting a large object to a site where it is not used. Therefore, the transmission of a given Blob must be delayed until this Blob is effectively used by a function or for delivering the final result.

To achieve delayed Blob transmission, the idea consists in using Blob identifiers. During query processing, Blob’s are substituted by Blob Unique Identifiers (BlobId’s), composed of the origin site URL concatenated with the Blob local reference. Using BlobId’s instead of Blob’s allows transmitting only “small” attributes, i.e., basic data types or BlobId’s. When the Blob becomes necessary, the Blob’s site is contacted and the Blob is fetched. This operation, called GetBlob, can be achieved either by the function wrapper of the function which requires the Blob or by the LeSelect server itself, to compose the final result. The GetBlob operation is similar to a file transfer using the FTP protocol. LeSelect must therefore be enhanced by adding the counterpart of the GetBlob (similar to an FTP daemon) running on each LeSelect server.

Using the DelayedTransfer technique with QEP1, the execution time of the naive execution becomes 36000 s. Note that Blob’s associated with tuples that do not satisfy predicate \( p_2 \) are not transmitted since the Blob’s are actually transmitted when the function CV is applied.

The performance gains that can be obtained using DelayedTransfer mainly depends on (i) the size of the transferred Blob’s; (ii) the site distance, which represents the number of sites between the site where the Blob’s reside and the site where the Blob’s are used for the given QEP; and (iii) the selectivity of the predicates which minimize the number of transmitted Blob’s. Using DelayedTransfer incurs very little overhead for the transmission of BlobId’s.

### 3.2 Avoiding Redundant Function Calls and Blob Transfer

Data redundancy can severely slow down query execution since it may incur multiple transmissions of the same Blob and/or redundant computation of expensive functions on the same values. Data redundancy may happen during query execution because (i) the data can be itself redundant, and (ii) join operations can increase the number of tuples, thus creating redundancy.

Three caching techniques have been proposed to deal with redundant function calls. Sorting requires sorting the input relation on the input attributes in order to maintain a cache for only one input/output pair. Conversely, using memoization we need to maintain input/output pairs for every tuple in a main-memory hash table. The idea of hybrid caching [8] is to apply the hybrid hashing strategy for the caching of input/output pairs. Without considering parallelism, any of the three caching techniques can be used (see Section 4.1).

When considering programs which process Blob’s, two situations should be distinguished: (i) the program has only one input parameter of type Blob and (ii) the program has several
input parameters of which one or several are of type Blob. In
the first case, avoiding redundant function calls and redundant
Blob transmission requires only to cache (BlobId, output) pairs.
The retrieved Blob can be discarded as soon as the function has
processed it since the function cache will be used for further
tuples with the same BlobId. In the second case, two distinct
caches must be used: one for the function and one for the
associated GetBlob(s). The function cache contains the
input/output pairs with BlobId's in the place of Blob's. The
GetBlob(s) cache(s) contains the (BlobId, Blob) pairs. In that
case, the retrieved Blob's must be kept in the cache until the last
tuple has been processed. Note that, for a given QEP, the
DelayedTransfer technique along with the Caching technique
allows to reduce Blob transmissions to its lower bound. With
QEPi, both techniques lead to a response time of 9000s.

In our context, the performance improvement which we can
expect using Caching mainly depends on (i) the number of
duplicate values; (ii) the function cost or, for Blob caching, the
Blob's transmission cost and (iii) the input/output pair size.

3.3 Parallelizing Blob Transfer and Function
   Execution

When considering functions which process Blob's data, the
elapsed time of the function execution can be reduced by
parallelizing the Blob's transmission and the function execution
because these two operations use distinct resources [10].
Moreover, in the presence of several Blob attributes as function
input, it is beneficial to parallelize the transmission of each
Blob attribute as soon as (i) the Blob's are transmitted from
distinct sites and (ii) the network is not overloaded. We refer to
this technique as BackgroundTransfer.

Using BackgroundTransfer reduces the total execution time
of QEPi to 7000s. The performance improvement of BackgroundTransfer depends on the relative costs of
the function evaluation and the Blob transmissions. In the most
common case of functions which take one Blob as input,
BackgroundTransfer can divide the elapsed time by a factor 2
(when both transmission and evaluation costs are equal).

3.4 Discussion

Let us consider the different QEP's which can be examined
by a query optimizer for the query example. Discarding the
QEP's with Cartesian products, the 5 remaining QEP's are
shown on Figure 2. For each query execution plan, QEPo, we
have computed the response time of an execution which uses
one or more of the above execution techniques, namely
DelayedTransfer (DTi), Caching (Ci) and BackgroundTransfer
(BT). ALL, is indicated when all three techniques are used,
while Ni corresponds to the naive execution. We should note
that the three techniques are always beneficial, regardless of
the QEP. This is not surprising given the potential performance
 gains that they can yield compared with the extremely small
induced overheads.

In the following, we discuss the interdependencies between
these techniques and their effect on the optimization of queries
with expensive functions and Blob's:

- **Caching impacts Optimization**: As pointed out in [8, 9],
  without caching, evaluating expensive functions after a join that
  produces duplicate values may give disastrous results while it
  can be beneficial with caching. With Caching, C1 is better than
  C3 and C4 while without Caching, N1 is worse than N3 and N4
  because joining V and P creates duplicates.

- **The DelayedTransfer technique impacts Optimization**: Without the DelayedTransfer technique, query optimizers
  would certainly discard potentially optimal QEP's as soon as
  they incur useless Blob transmission given the high cost of
  these transmissions. For instance, in Figure 3, DT1 is better than
  DT5, while N1 is worse than N5. Another alternative is to use the
  semi-join technique [2] in order to avoid such transmission,
  thereby complicating the optimization process.

- **Caching impacts the DelayedTransfer technique**: Caching
  requires BlobId's, as defined in Section 3.1. Indeed, using just a
  local reference is not sufficient since a query may well perform
  the union of two relations containing distinct Blob's with the
  same local reference. Thus, as soon as Caching is used, the
  DelayedTransfer technique can be used for free since the
  transmission of BlobId's is necessary for Caching.

The DelayedTransfer technique impacts BackgroundTransfer: Using the specific GetBlob operation for Blob
transmission allows to easily parallelize it with the function
execution. Again, using the semi-join technique, would lead to
a complex problem of parallel optimization.
4 Parallel Query Optimization Technique

Our query execution techniques allow relaxing the constraints of function placement in the QEP thus simplifying much the query optimization process. Because of Caching for instance, we may envision to execute a function after a join, even though it produces duplicate tuples, in order to take advantage of join's reducing power on distinct values. Similarly, DelayedTransfer allows us to ignore Blob's location during QEP generation and thus to evaluate expensive functions when it is most beneficial. Given these techniques and the cost of function execution, it seems obvious that functions should be evaluated as late as possible, i.e., after all select and join operators which do not use function results. This corresponds to the PullUp heuristic proposed in [9] which makes the case for caching. However, this work was done for a centralized context and thus does not mention the need for DelayedTransfer.

Using our query execution techniques with the PullUp heuristic allows us to generate QEP's very simply for queries with a single function. We can optimize the query ignoring the function and then place the function as late as possible in the QEP. But with more than one function, there are two other optimization opportunities. First, parallelism should be exploited when the functions are located on distinct sites or consume distinct resources on the same site (e.g., one is I/O bound and the other CPU bound [10]). Second, if predicates are defined on function results, functions should be ordered so as to minimize response time.

In the rest of this section, we first show the benefits of pipeline and independent parallelism for queries with several functions and propose a parallel QEP generation strategy which combines both parallelisms. Then we present practical considerations which may invalidate (i) the execution ordering of expensive functions with predicates on their results, called expensive predicates for short; and (ii) the pipeline execution. We propose a new hybrid parallel technique to solve these problems and give experimental results.

4.1 Parallelism between Functions: Pipeline versus Independent Parallelism

As for traditional query parallelization [Gra93], pipeline parallelism between several functions can always be introduced but induces necessary synchronizations. On the other hand, independent parallelism does not require synchronization but is not always possible. Let us illustrate the advantages and disadvantages of pipeline and independent parallelism using our query example. Since functions CV and CP are independent, we can consider pipeline or independent parallelism.

QEP’s 1 and 2 (Figure 2) exhibit pipeline parallelism: In QEP1, the output of function CP is pipelined, after reduction by predicate pCV to CV. This allows using predicate pCV to minimize the number of CV invocation. Conversely, in QEP2, CV is executed first, and CP is used to minimize the number of CP calls. The two execution plans, obviously, do not give the same performance and one should decide on the best one. Moreover, the synchronization between the functions may well incur idle times, i.e., one function has no more tuples to process and waits for the other.

QEP, exhibits independent parallelism: the functions CV and CP directly consume their inputs from the original relations. Note that the join operation is done after the function execution, which is contradictory to the PullUp heuristic. Even if this can be solved easily using for instance semi-joins, the plans with independent parallelism may be worse than pipelined QEP’s since they do not benefit from the selections on the function results. Note however that using pipeline parallelism between several functions invalidates the sorting technique for function input caching since it breaks the pipeline.

Let us now define an ideal parallel QEP as the parallel QEP which maximizes resource utilization and minimizes total work. Thus, independent parallelism is not ideal since it may not minimize total work and pipeline parallelism is not ideal since it may incur resource idleness. Nevertheless, one strategy to generate the parallel QEP could be:

1. optimize the plan (without the expensive functions) using traditional algorithms;
2. place the expensive functions as late as possible in the QEP;
3. apply independent parallelism between expensive functions with no predicates;
4. apply pipeline parallelism for expensive functions which are dependent;
5. for predicates over independent functions, choose an execution order and apply pipeline parallelism.

In the following, we focus on step 5 since the other ones are either well known or rather straightforward.

4.2 Practical Considerations: Expensive Predicate Order

Choosing an execution order for expensive predicates, as in step 5, can be achieved by adapting to the distributed context the techniques devised in [9, 6] or by using exhaustive search. Exhaustive search consists in generating all possible predicate orders and choosing the "best" one by applying a parallel cost model. Exhaustive search is viable as long as the number of expensive predicates in a query is rather small, which we feel to be quite realistic. Anyway finding the "best" order with any query optimization strategy strongly relies on parameters that may be very difficult to assess. While this is a well-known problem in query optimization, it is exacerbated in our context. We present below some of these parameters and discuss the problems of predicting their values.

- **Function cost**: The per-tuple function cost should be provided by the function publisher or computed by the function wrapper itself based on historical logs. The actual per-tuple function cost, however, depends on the machine load. Moreover, predicate ordering techniques assume constant function costs which is not the case for functions whose execution time depend on their input (e.g., image processing).

- **Predicate Selectivity**: Predicate selectivity is usually computed based on some detailed statistics (e.g., histograms) on the selected data. While this approach is effective with base relations, it is inadequate for predicates over function results. Indeed, one cannot store detailed statistics on nonexistent data (since they are not yet computed). In the best case, vague estimations can be based on the function output domain. Consider for instance the predicate AveragedColor(image) = "Red". If the function

\[ \text{AveragedColor(image)} = \text{"Red"}. \]
AveragedColor produces only 10 color values, we can roughly approximate the predicate selectivity to 10%. Even if these approximations may be refined using, for instance, output value frequencies, they are doomed to inaccuracy.

- **Number of distinct values:** Since we use Caching, the overall function cost depends on the number of distinct input values. This parameter may be difficult to assess since duplicate values can be generated (conversely distinct values eliminated) by previous joins. Note that the number of distinct values also affects the predicate selectivity parameter.

All these considerations play against a statically predicted order of expensive predicate evaluation.

### 4.3 Practical Considerations: Pipeline Execution

Even with a good predicate order, synchronizations during pipeline execution may well increase response time. These problems occur when a function has no more tuples to process and waits for the other. Consider for instance, two expensive predicates $P$ and $Q$ with respective costs of 50 and 10 seconds per tuple. Since $P$ is more expensive than $Q$, executing the pipeline $P \cdot Q$ a-priori leads to $Q$’s idleness. Hopefully, duplicate inputs for $P$ may reduce $Q$’s idleness, since, for these duplicates, $P$’s evaluation cost is negligible (i.e., cache lookup). Using buffers between $P$ and $Q$ may absorb the synchronization problems. In fact, the resource utilization for the pipeline $P \cdot Q$ depends on the distribution of distinct values in $P$ and $Q$ inputs and on the selectivity of $P$.

These synchronization problems are extremely hard to predict. To assess them, we rely on experimentation based on a data set generator and a distributed pipeline execution system which we implemented. In the following, we describe the experimentation platform used also for subsequent experiments, and present the results.

We want to measure the execution time of a pipeline chain with several predicates applied on a single relation. The execution time of a pipeline of expensive predicates obviously depends on the costs and selectivities of expensive predicates but also on the input data, i.e., the number and the distribution of distinct input values for the predicates. Thus, we need to introduce our data generator and the way we simulate execution. Our data generator takes as input the relation size and, for each attribute, the number of distinct values. It randomly generates a relation with the following properties: (i) attributes are independent (no functional dependencies); (ii) the number of duplicates per distinct value is random; and (iii) the data distribution is random.

To simplify the assessment of a pipeline execution with $n$ predicates, we generate relations containing exactly $n+1$ attributes: a key, and for each predicate, an input attribute. Thus, for the pipeline of two predicates $P$ and $Q$, we generate a relation $R$ with triples $(K, A_P, A_Q)$ where $A_P$ is input of $P$ and $A_Q$ input of $Q$. To simulate the pipeline execution of $P$ and $Q$, we only need to implement two processes running on two distinct machines. The first process takes $R$ tuples as input, simulates the tuple processing using caching and sends the selected tuples to the second process.

Let us now try to quantify the synchronization problems. In this experiment, we used a relation with 200 tuples. Attribute $A_P$ has 40 distinct values and $A_Q$ has 160 distinct values. The selectivities of $P$ and $Q$ are 40% and 60%. We ran the pipeline $P \cdot Q$ on several relations generated with the same parameters. Since $P$ and $Q$ have costs of 50 and 10 seconds per tuple (respectively), an estimation for the execution time, assuming no synchronization problems, will give 1000s, corresponding to applying $P$ on 40 distinct values, and applying $Q$ for the last tuple (since all others $Q$ evaluations are overlapped with $P$ evaluation).

![Figure 4: Trace of the pipeline execution P-Q](image)

The execution time measured, 2520s, is much higher. Figure 4 shows for each processed tuple the time when $P$ has been evaluated ($t_P$) and the same for $Q$ ($t_Q$) where $i$ is the tuple number (in X axis). The explanation is quite simple. When execution begins, $P$ has 100% chance to process a distinct value (the cache is empty!). As long as the execution proceeds, the probability to process a distinct value decreases. In the graph, we can see that after processing 85 tuples, 95% of distinct values for $P$ have been processed. Thus, in a first part of this execution, $Q$ is always waiting for $P$ ($t_Q = t_P + 10$ for $i < 85$).

When the probability becomes very low, $Q$ is overloaded by tuples processed very quickly by $P$ since it corresponds to duplicate values. This scenario, that we observe on several input data sets, illustrate the synchronization problem which may occur while executing in pipeline. In the worst case, pipeline execution can yield response time close to that of sequential execution. Because of duplicates and selectivities, it can happen that few executions are overlapped, e.g., the attribute is distinct for the first predicate and duplicated for the second in a first part, and conversely in a second part.

### 4.4 Parallelism between Functions: an Hybrid Approach

From the previous sections, one may wonder whether pipeline execution is appropriate for expensive predicates. The execution order is hard to predict and synchronization problems hurt response time. Using independent parallelism could maximize resource usage but does not benefit from the selections on the function results (which reduce execution time, even if it is hard to predict). Thus, we propose a hybrid approach where we execute using independent parallelism different pipeline chains corresponding to different execution orders. The pipeline chain chosen by the ordering algorithm (exhaustive search or based on [9, 6] has priority over the other concurrent pipeline chains.)
Let us illustrate with the execution of predicates \( P \) and \( Q \) on relation \( R \) with the predicate order \( P \cdot Q \). This plan is easily done with 2 instances of \( P \), \( P_1 \) and \( P_2 \), on \( P \)’s site, and \( Q_1 \) and \( Q_2 \), on \( Q \)’s site (see Figure 5). \( R \) tuples are consumed by either \( P_1 \) or \( P_2 \). The tuples selected by \( P_1 \) are sent to \( Q_1 \), while those selected by \( Q_2 \) are sent to \( P_2 \). A union is necessary on the result of each pipeline chain. Let us assume that the ordering \( P \cdot Q \) is chosen by the ordering algorithm. It is enough that synchronization problems that occur. Since \( P \cdot Q \) pipeline chains is done dynamically based on the tuples. Thus, the distribution of \( R \) tuples between the two \( P \) has \( \alpha \) priority over \( P \cdot Q \) site for \( P \). \( P \cdot Q \) has priority over \( Q \) at \( Q \)’s site for \( P \cdot Q \) to have priority over \( Q_2 \) to \( P_2 \). The advantage is obvious. When there are synchronization problems in \( P \cdot Q \), \( Q_2 \) is waiting for tuples sent by \( P_1 \). Then, by the simple rule of thread priorities, \( Q_2 \) takes over from \( Q_1 \) and starts consuming \( R \) tuples. Thus, the distribution of \( R \) tuples between the two pipeline chains is done dynamically based on the synchronization problems that occur. Since \( P_1 \) and \( P_2 \) (resp. \( Q_1 \) and \( Q_2 \)) share the input/output cache, having two instances of \( P \) (resp. \( Q \)) does not incur significant extra cost (function re-execution) and avoids idleness in the system. Resource usage is maximized and predicate selectivity is profitable. Figure 5 shows the hybrid query execution plan for this example.

This approach resembles the competition model [1] which executes several QEP’s of the same query concurrently and observes until a certain time when all plans but the best one are aborted. In our case, we do not abort any pipeline chain since they all do useful work.

![Figure 5: Hybrid Parallelism](image)

To generalize this approach to the execution of a pipeline with \( n \) predicates, the hybrid plan has \( n \) concurrent branches which is sufficient to guarantee optimal resource usage and one branch has priority over the other. The general algorithm works as follows:

1. Using exhaustive search or based on \([9, 6]\), choose the best predicate ordering;
2. Permute the order to obtain concurrent branches, each starting with a different predicate.

### 4.5 Performance evaluation

In order to assess the performance of the hybrid strategy, we have extended our experimental platform, using different threads with priorities on the same machine. Then, for 2 predicates \( P \) and \( Q \), we varied all the parameters and compare the resulting execution times. The problem with such experimentation is that measurements take a long time (several minutes per measurement), thus making hard the analysis according to several parameters (predicate selectivity and cost, number of distinct values per attribute). Therefore, we decided to simulate pipeline and hybrid executions to obtain results more rapidly (each execution takes less than 0.1s). In our simulation, data are generated as before. However, the execution for 2 predicates is simply simulated with 4 time counters for \( P_1 \), \( P_2 \), \( Q_1 \), and \( Q_2 \). We also used the experimental platform to calibrate our simulator. Except for the overhead of thread scheduling (less than 0.1% of execution time), the results were identical.

We now present performance comparisons between the pipeline and hybrid strategies based on our simulator. We used a relation of 200 tuples \((K, A_P, A_Q)\) where \( A_P \) is input of \( P \) and \( A_Q \) input of \( Q \). We varied 5 parameters: number of distinct values for \( A_P \) and \( A_Q \), selectivity of \( P \) and \( Q \), relative cost of \( P \) and \( Q \). For each parameter, we compute a random value between a minimum and a maximum value (see Table 2). A combination of 5 random values makes a scenario. We generate 10000 scenarios in order to obtain steady averaged results. Moreover, for each scenario, we generated 20 data sets in order to study the random effect of data generation. For each of the 200 000 generated data set, we simulate the execution of the four strategies (pipeline and hybrid with the two orderings). All the results and data sets were stored in a DBMS.

#### Table 2: Input parameters for scenarios generation.

<table>
<thead>
<tr>
<th>Description</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of distinct values for ( A_P ), input of predicate ( P )</td>
<td>20</td>
<td>200</td>
</tr>
<tr>
<td>Number of distinct values for ( A_Q ), input of predicate ( Q )</td>
<td>20</td>
<td>200</td>
</tr>
<tr>
<td>Selectivity (over distinct values) of predicate ( P )</td>
<td>10%</td>
<td>90%</td>
</tr>
<tr>
<td>Selectivity (over distinct values) of predicate ( Q )</td>
<td>10%</td>
<td>90%</td>
</tr>
<tr>
<td>Relative cost of predicates (( \text{Cost}(P)/\text{Cost}(Q) ))</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Based on the results, we can make the following observations. First, even though we have the actual parameter values for choosing a predicate order, our choice may not be a good one. In fact, on about 1/3 scenarios, we noticed that the good predicate order for pipeline execution varies from one data set to the other. This is due to the assumption of uniform distribution (of duplicates for instance) and synchronization problems.

Second, for a given predicate order, the hybrid strategy yields response times which are inferior to those of pipeline execution in 89% of the cases and equal in 11%. The average loss of pipeline versus hybrid is 53% using the wrong predicate order for pipeline and hybrid execution, and only 5% using the good one.

In the worst cases for pipeline execution, hybrid strategy divides the response time by a factor between 2 and 13. These cases correspond to 14% of the scenarios. In these scenarios, either selectivities are high (e.g., between 10% and 20%) or there are many duplicates. Since in the good predicate order, the first predicate discards almost all tuples, changing the ordering of predicate execution has a strong impact on response time.

Finally, the strongest aspect of the hybrid strategy is that execution time does not change much according to the predicate order statically chosen. In average, hybrid execution time with a bad predicate order is close to the one obtained with the good order (12%). In the pipeline execution, choosing the wrong order increases, in average, the execution time by 67%.
The extreme points are even more significant: an error in the choice of pipeline execution ordering can yield a QEP with an execution time 14 times worse, whereas the degradation with hybrid for the same QEP is only 30%.

Figure 6 summarizes the measurements, showing the distribution of the ratios between pipeline and hybrid execution times (i.e., performance loss of pipeline versus hybrid) for the good predicate order and the bad one. In summary, the hybrid technique is robust wrt. errors in predicting the good ordering of predicates. Furthermore, it allows absorbing synchronization problems and thus to improve response time by maximizing resource utilization.

![Figure 6: Performance Ratio (performance loss of pipeline versus hybrid) for both predicate orders](image)

5 Conclusion

In this paper, we addressed the problem of efficient query processing for queries involving expensive functions and large objects in distributed mediator systems. This work is done in the context of LeSelect, a distributed mediator system developed at Inria to allow scientists to publish their data and programs on the Web. It was primarily motivated by performance problems arising in two earth-science applications (coastal zone management and air quality modeling) for which LeSelect is used. Typical queries may involve distributed data, complex programs (expensive functions) and Blob’s and can take a very long time to process.

Based on the assumption (realistic in our scientific cooperation context) that the time to process relational operators is negligible compared to that of processing expensive functions and transport large objects, we made two major contributions.

First, we proposed three query execution techniques, namely DelayedTransfer, Caching and BackgroundTransfer, which minimize the costs of Blob transmission and expensive function invocation. We showed that they are always beneficial and must be used together. They also simplify the query optimization process.

Second, we proposed to fully exploit the inherent parallelism of the distributed architecture. We showed the benefits of pipeline and independent parallelism for queries with several functions and proposed a parallel QEP generation strategy which combines both parallelisms. Based on experimentation, we show that it is hard to predict the optimal execution order when dealing with several functions. We proposed a new hybrid parallel technique to solve this problem and showed that the hybrid technique is robust wrt. errors in predicting the good ordering of predicates. Furthermore, it allows absorbing synchronization problems and thus to improve response time by maximizing resource utilization.

We are convinced that the capability of program publishing, coupled with distributed data management, is of high interest for many emerging applications on the Web, in particular scientific applications.

6 Bibliography