Client-Site Query Extensions

Tobias Mayr and Praveen Seshadri

Cornell University

mayr,praveen@cs.cornell.edu

Contact Author:

Praveen Seshadri
e-mail: praveen@cs.cornell.edu
Office Phone: (607)255-1045
Office Fax : (607)255-4428

Abstract

We explore the execution of queries with client-site user-defined functions (UDFs). Many UDFs can only be executed at the client site, for reasons of scalability, security, confidentiality, or availability of resources. How should a query with client-site UDFs be executed? We demonstrate that the standard execution technique for server-site UDFs performs poorly. Instead, we adapt well-known distributed database algorithms and apply them to client-site UDFs. The resulting query execution techniques are implemented in the Cornell Predator database system, and we present performance results to demonstrate their effectiveness.

We also reconsider the question of query optimization in the context of client-site UDFs. The known techniques for expensive UDFs are inadequate because they do not take the location of the UDF into account. We present an extension of traditional 'System-R' optimizers that suitably optimize queries with client-site operations.

1 Introduction

Optimization techniques have been studied thoroughly for object-relational SQL queries with expensive user-defined functions (UDFs). The assumptions made in these studies are that (a) the cost of each UDF invocation is known *a priori,* and invariant, (b) the UDF itself is a blackbox characterized by a single cost value (which may be broken into CPU and I/O costs). In some systems, the cost may be specified as a function of the sizes of the function arguments. These assumptions implicitly expect that the user is extending the *server* with a new function. However, experience with object-relational databases shows that extending the database server is difficult even for experienced programmers, and impossible for large numbers of non-expert users. In large-scale environments like the WWW, users need to incorporate *client-site UDFs* into SQL queries run at a server. Consider the following motivating example:

A DBMS offers stock market data to its clients over the WWW. The users connect to the database to analyze the performance of companies and to extract the necessary information about prospective candidates for their investments. Sophisticated investors will have their own local collections of data and analysis algorithms that must be integrated into the process of choosing and retrieving the desired information.

Take the following example query:

```
SELECT S.Name, S.Report
FROM   StockQuotes S
WHERE  S.Change / S.Close > 0.2 AND ClientAnalysis(S.Quotes) > 500
```

Figure 1: Use of a Client-Site UDF

The investor requests names and financial reports of companies that accord to his criteria. The first predicate, filtering companies on a 20%+ uptick, can be expressed with simple SQL predicates and will be executed on the server. However, the second predicate involves a UDF that has to be executed on the client site for a variety of reasons.

In this and many other examples, it becomes clear why client-site UDFs need to be supported:

a) The investor's analysis UDFs are a valued asset that is ideally not revealed.

b) The UDFs may use data that resides exclusively on the client. These data might only be available in a *client-specific* representation, or it might represent confidential information.

c) The UDFs may not be trusted by the server. In earlier work [GMHE98], we showed that the server can trust UDFs written in Java to a certain extent, and we are developing further security mechanisms [CSM98]. However, the security demands of the server constrain the UDFs. Further, many UDFs are not written in Java, and if these are allowed to run at the server, they could compromise its *security*.

d) The UDFs may be resource intensive and it may be inappropriate to burden the server with their execution.

e) In the context of such expensive operations, there is a serious scalability concern, since resource intensive UDFs of a multitude of users would together degrade the server performance.

In our research, the UDFs and their client-site execution environment were implemented in Java. However, there are many other architectural frameworks and distributed implementation models, like CORBA, DCOM, or JavaBeans, which we could have chosen instead, and to which the research results apply. For the rest of this paper, we will assume that the network connecting the clients with the server forms the bottleneck of client-site UDF execution. This applies for example to clients connected over the Internet, or over an asymmetric connection, where only the downlink has high bandwidth while the uplink will form the bottleneck.

1.1 Summary of Contributions

We believe that client-site UDFs are central to scalable object-relational applications. Existing query processing techniques for expensive UDFs are not appropriate for client-site UDFs. Indeed, the use of traditional approaches leads to slow and inefficient execution. This can be explained by three key observations:

a) Client-site UDF execution time can involve *network latency*. , the latency needs to be hidden through the appropriate use of concurrency.

b) Client-site UDF performance can depend on the optimized usage of *network bandwidth.* Specifically, the asymmetry between client uplink and downlinks needs to factor into query evaluation decisions. It may be possible to trade off bandwidth on the uplink for bandwidth on the downlink.

c) The *optimal placement* of client-site UDF operators in the query plan is different from the placement of expensive server-site UDFs.

The primary contribution of the paper is the development of techniques to process and optimize queries with client-site UDFs. These techniques blend object-relational query processing with the distributed database algorithms. Specifically, our research makes the following contributions:

1. We develop efficient execution algorithms for client-site UDFs, and describe their implementation.
2. We explore the tradeoffs between algorithms due to asymmetric network connections, and propose options that save bandwidth on the client's uplink at the cost of increased traffic on the downlink.
3. We present performance results of the prototype implementation in the Cornell Predator database system.
4. We present a simple cost model that allows us to determine the optimal choice of the execution algorithms and their parameters
5. We develop query optimization techniques for complex queries with client-site UDFs. The techniques are extensions of a traditional System-R style optimizer.

Our conclusion is that a database system needs to recognize the special characteristics of client-site UDFs and apply appropriate query evaluation and optimization strategies to such queries.

1.2 Related Work

To summarize, our work on queries with client-site UDFs builds on existing work on expensive UDF execution and distributed query processing. The main issues are: (a) how should the UDFs be executed, (b) how should query plans be optimized?

Client-site UDFs are expensive; they cannot simply be treated like built-in, cheap predicates. The existing research on the optimization of queries with expensive server-site functions is closely related. The execution of UDFs is considered straightforward; they are executed one at a time, with caching used to eliminate duplicate invocations. The process of efficient duplicate elimination by caching has been examined in [HN97]. Predicate Migration[HS93,Hel95] determines the optimal interleaving of join operators and expensive predicates on a linear join tree by using the concept of a rank-order on the expensive predicates. Its per-tuple cost and selectivity determine the rank of any operation. The concept was originally developed in the context of join order optimization [IK84, KBZ86, SI92]. The Optimization Algorithm with Rank Ordering [CS97] uses rank order to efficiently integrate predicate placement into a System-R style optimization algorithm. UDF optimization based on rank ordering assumes that the cost of UDF operators is only determined by the selectivity of the preceding operators. We show in Section 5 that rank order does not apply well to client-site operations. Our optimization algorithm does not rely on it. Another approach models UDF application as a relational join [CGK89, CS93] and uses join optimization techniques. Our approach to optimization takes this route.

There is a wealth of research on distributed join processing algorithms[SA80,S+79,ML86] that our work draws upon. The distribution of query processing between client and server has also been proposed independently of client-site UDFs in [FJK96], as a hybrid between data and query shipping. Joins with external data sources, specifically text sources, have been studied in [CDY95]. To avoid the per-tuple invocation overhead of accessing the text source, a *semi-join* strategy is proposed: Multiple requests are batched in a single conjunctive query and the set of results is joined internally. Earlier work on integration of foreign functions [CS93] proposes the use of semantic information by the optimizer. Our work is complementary in that semantic information can be used in PREDATOR to transform UDF expressions[S98]. We consider the execution of queries after such transformations have been applied.

To summarize, our work is incremental in that it builds upon existing work in this area. However, the novel aspects of the work are

(a) we identify client-side UDFs as an important problem and adapt existing approaches to fit the new problem domain,

(b) while earlier work modeled UDFs as joins for the purpose of optimization, we go further by using join algorithms for the purposes of execution too,

(c) we identify and exploit important tradeoffs related to network asymmetry that lead to interesting optimization choices.

2 Client-Site UDF Execution

In this section we explore different execution techniques for a single client-site UDF applied to all the tuples of a relation. For now, we ignore the issue of query optimization and operator placement. In the first subsection, we expose the poor performance of a naive approach that treats client-site UDFs like expensive sever-site UDFs. The next subsection models UDFs as joins, leading to the development of evaluation algorithms based on distributed joins. We use the example query in Example 1.

In our terminology, the input relation consists of the columns that are arguments to the UDF -- the argument columns (`Quote`) -- and the non-argument columns (`Report,Name`). The input relation has two different kinds of duplicates: those which are identical in all columns, called *tuple duplicates*, and those only identical in the argument columns, called *argument duplicates*. Simple predicates that rely on the values in the result columns, but can be executed on the client, for example `ClientAnalysis(S.Quotes)>500`, are called pushable predicates. Similarly, projections that can be applied immediately after the UDF are called pushable projections, as in our example the projection on Report and `Name`.

2.1 Traditional UDF Execution

Current object-relational databases support server-site UDFs. It is tempting to treat a client-site UDF as a server-site UDF that happens to make an expensive remote function call to the client. If `ClientAnalysis` were a server-site UDF, the established approach is to treat it as a black-box extension. The evaluation pseudo-code for the classical 'iterator-model' query processor is shown below.

```
While (Input.available())
  Record := Input.getRecord()
  Result := UDF( getArguments( Record ) )
  output.putRecord( addColumn( Record, Result )
```

The encapsulation of the client communication within a black-box UDF makes some optimizations impossible. On each call to `ClientAnalysis`, the full latency of network communication with the client is incurred. This is because most iterator-model execution engines do not apply one operator of the query plan pipeline to multiple tuples concurrently. (We show the timeline of execution in Figure 2a).

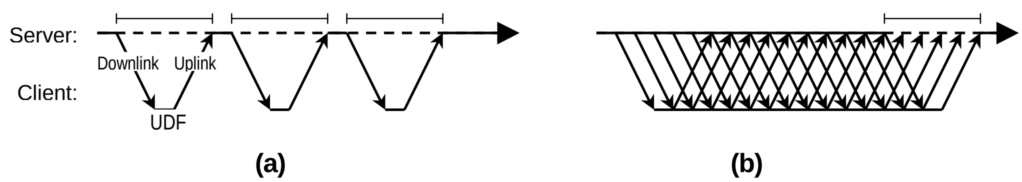

Figure 2: Timeline of Nonconcurrent and Concurrent Execution

The key observation here is, that even if the client might not process multiple tuples concurrently, the network is capable of accepting further messages while others are already being transferred. This means that we can keep a number of messages concurrently in the pipeline formed by downlink, client UDF-processing, and uplink. We refer to this number as the *pipeline concurrency factor*. Figure 2(b) shows the timeline for a concurrency factor of 5.

Another problem of the traditional approach is the ignorance of *network bandwidth*. But it is possible to vary the bandwidth usage using different execution techniques. Consider the UDF in Figure 1: It seems straightforward to simply send the quotes and wait for the results. Then the selection that depends on the results can be applied on the server site. Depending on the networking environment the performance might be far from optimal. For example, assume that the client's uplink turns out to be the bottleneck, as is the case with modern communication channels like ADSL, cable modems, and wireless networks. We might accept additional traffic on the downlink if we could in exchange reduce the demand on the uplink. We will explore different execution strategies that allow these kinds of tradeoffs.

2.2 UDF Execution as a Join

It is possible to model UDF application on a table as a *join operation*: The user defined function in Figure 1 can be seen as a virtual table with the following schema:

`ClientAnalysis(< PriceQuoteArgument :: TimeSeries , Rating :: Integer >)`

The **`PriceQuoteArgument`** column forms a key, and the only access path is an "indexed" access on the key value. Indexed access in this manner incurs costs independent of the size of the table. UDF execution as a join with such a UDF table would work analogously to an equi-join with a relation indexed on the join columns. The pseudo code for the join of a relation with the UDF is shown below:

```
for each tuple t1 from outer relation
  retrieve tuple t2 with matching argument columns from virtual UDF table
  join t1 with t2 on argument columns
  output result
```

Since UDF application is modeled as a join, client-site UDF application is modeled as a multi-site join. We now examine distributed join algorithms as they apply to this context.

2.3 Distributed Join Processing

There are three standard distributed algorithms[SA80,ML86] to join the outer relation *R* and the inner *F,* residing on sites *S(erver)* and *C(lient)*:

- Join at *S* : Send *F* to *S* and join it there with *R*. (Not feasible for UDFs since there is no file-scan access to *F)*
- Join at *C* : Send *R* to *C* and join it there with *F*.
- Semi-Join : Send a projection on the join columns of *R* to *C*, which returns all matching tuples of *F* to *S*, where they are joined with *R*.

Identifying *S* with the server and *C* with the client, we get two variants for client-site UDF application from the last two options. We will briefly introduce each one now, and go into more detail in the later part of this section.

2.3.1 Semi-Join

Semi-joins are a natural 'set-oriented' extension of the traditional 'tuple-at-a-time' UDF execution strategy. Consider the pseudo code below:

```
For each batch of tuples in R:
  Step 0: eliminate duplicates                           (server)
  Step 1: send a set of unique S.x values to the client    (downlink)
  Step 2: evaluate UDF(S.x) on all S.x values            (client)
  Step 3: send results back to the server                (uplink)
  Step 4: 'join' each result with the corresponding tuples (server)
```

Note that steps 0 through 4 may be executed concurrently because they use different resources (except 0 and 4). If the set sent in step 1 consists of only one argument tuple, then this is the 'tuple-at-a-time' approach described in the previous section. If the entire relation R is treated as the 'batch', we have a classical semi-join. The details of the different steps vary depending on the execution strategy. It is convenient to model this conceptually by Figure 3 below, where the different steps are identified as components of a pipeline, with the potential for pipeline concurrency.

For server-site UDFs it is considered acceptable if the execution mechanism blocks for each UDF call until the UDF returns the result. However, for client-site UDFs a large part of the over-all execution time for one tuple consists of network latencies -- steps 1 and 3 above. Instead, we can ship several tuples on the downlink at the same time, while another tuple is processed by the UDF, and other results are being sent back over the uplink. Concurrency between the server, the client, and the network can hide the latencies. To obtain this goal we will architecturally separate the sender of the UDF's arguments from the receiver of its results, and have them and the client work concurrently. These components form a pipeline, whose architecture is shown in Figure 3.

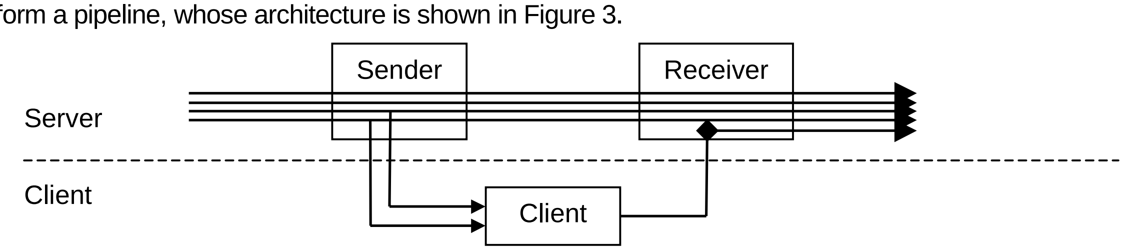

Figure 3: Semi-Join Architecture

The joining of the UDF results with the processed relation depends in its complexity on the correspondence between the tuple streams coming to the receiver from the client and from the sender. Since the sender eliminates duplicates, the receiver has to do an actual join between the two streams. Any join technique (for example, hash-join) is applicable at the receiver. If the sender sorts and groups its input on the argument column before sending it to the client, then the receiver has to perform a merge-join. We will assume this in the rest of the paper.

2.3.2 Join at the Client

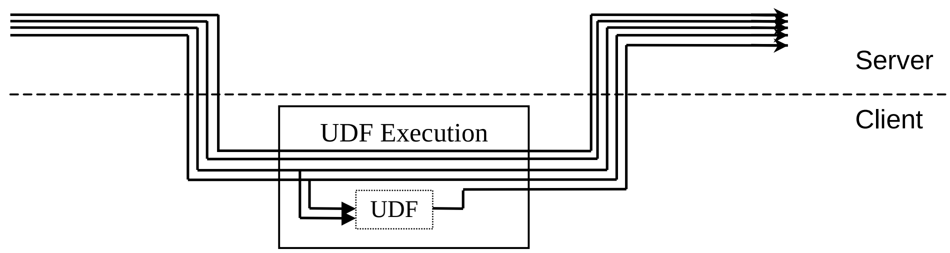

Figure 4: Client-Site Join Architecture

Join at the client-site is possible by sending the entire stream of tuples from the outer relation to the client site. The UDF is applied to the arguments from each tuple, and the UDF result is added to the tuple and shipped back to the receiver. The sender and receiver of the tuple streams on the server do not need to coordinate, since the entire tuples (with duplicates) flow through the client. (as shown in Figure 4.). Note that this does not mean that the client makes duplicate UDF invocations, since the server may sort the stream of tuples on the argument attributes.

An advantage of this strategy is that pushable selections and projections can be moved to the client site. This reduces the bandwidth used on the client-server uplink. On the other hand we have to send back the full records minus applicable projections, and not just results, as for the semi-join. Compared to the semi-join, more data is also sent on the downlink. Further, on both downlink and uplink, the semijoin method eliminates argument duplicates, whereas the client-site join performs no duplicate elimination. The difference between semi-join and client-site join is visualized in Figure 5. The left side shows what is being sent by each join method, the right side shows what is being returned. The horizontals correspond to the transferred columns while the verticals correspond to rows. We will quantify and experimentally evaluate these tradeoffs in the next section.

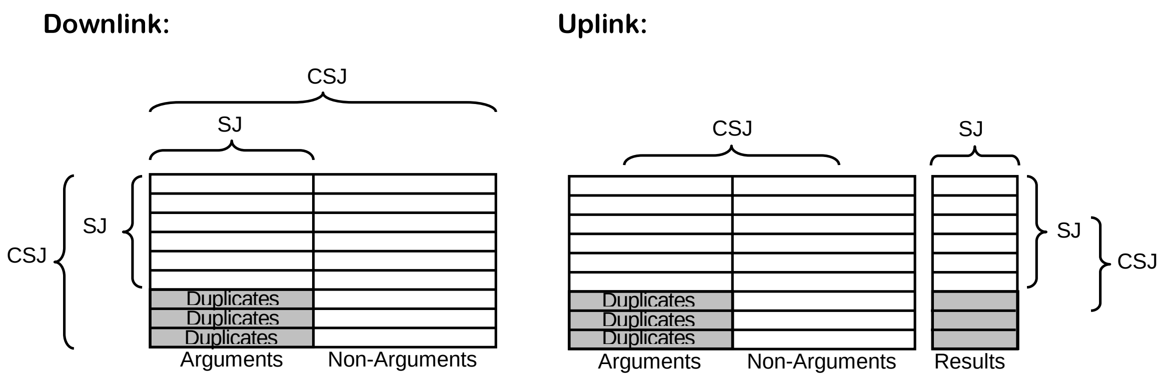

Figure 5: Tradeoffs between Client-Site and Semi-Join

3 Implementation

We have implemented relational operators that execute client-site UDFs in the Cornell PREDATOR ORDBMS. All server components were implemented in C++ and all client-site components are written in Java. Three different execution strategies were implemented:

a) Naive tuple –at-a-time execution

b) Semi-join

c) Client-site join

We first describe the implementation of the algorithms, and then compare their performance. Our goals for the performance evaluation are:

- Demonstrate the problems of the naive evaluation strategy.
- Show the tradeoffs between semi-join and client-site join evaluation of the UDF.

3.1 Join Implementation

3.1.1 Semi-Join

This relational operator implements the semi-join of a server-site table with the non-materialized UDF table on the client site. In our architecture (see Figure 3), the server side consists of three components: the sender, the receiver, and the buffer with which both communicate records. The sender gets the input records from the child operators and, after sending off the argument columns, enqueues them on the buffer. The receiver dequeues the records from the buffer and then attempts to receive the corresponding results from the client. Sender and receiver are implemented as threads, running concurrently. The buffer as a shared data structure is needed to keep the full records, while only the arguments are sent to the client. Also, records whose argument columns form duplicates of earlier records have to be joined with cached results at the receiver.

3.1.2 Concurrency

The size of the buffer that holds records that are 'between' sender and receiver, corresponds to the *pipeline concurrency factor:* The number of tuples that are transferred and processed on the client concurrently. A concurrency factor of 1 corresponds to tuple-at-a -time evaluation.

How large should the concurrency factor be? Analytically, we would expect that the number of records between sender and receiver should equal the number of records that can be processed by the pipeline sender - client - receiver in the time that it takes for one tuple to pass this pipeline. Let B be the minimum of the bandwidths of the downlink, the client UDF processor, and the uplink. One of these forms a bottleneck of the pipeline and thus limits the overall bandwidth. Let T be the time that it takes for one argument to travel to the client, for the result to be computed, and to be returned to the server. This is the time for which a record stays in the buffer, after its argument columns have been sent off until its result is received. The number of records that can be processed in this time is simply B * T, which is the necessary size for the buffer.

3.1.3 Client-Site Join

The client-site join uses a variation of this architecture: The sender dispatches the whole records to the client, which sends back the records with the additional argument column. We have the same components as above, but without the buffer between sender and receiver. The client-site join does not require any synchronization between both components, in contrast to the semi-join, where the buffer is used to synchronize sender and receiver.

3.2 Cost Model

We show in the performance evaluation section that the network latency problems of tuple-at-a-time UDF execution can be solved through concurrency (either semi-join or client-site join). Consequently, we focus in our cost-model on these two smarter algorithms. Both algorithms incur nearly identical costs at the client and on the server. We assume that neither client nor server is the pipeline bottleneck, and propose a simple cost model based on network bandwidth. We do recognize that this is a simplification and that a mixture of server, client and network costs may be more appropriate in certain environments (as was shown for distributed databases[ML86]). We also ignore the possibly significant cost of server-site duplicate elimination because the issues are well understood [HN97] and not necessarily central in the Web/Internet large-scale environment that we address.

3.2.1 Cost Model for Semi-Join and Client-Site Join

We now analyze and empirically evaluate the involved tradeoffs with respect to the factors that were visualized in Figure 5. To quantify the amount of data sent across the network, we define the following parameters:

- *A* : Size of the argument columns / total size of the input records
- *D* : Number of different argument tuples / cardinality of the input relation
- *S* : Selectivity of the pushable predicates
- *P* : Size of projected output record / size of output record before pushable projections are applied (i.e. the column selectivity of the projections)
- *I* : Size of one input record
- *R* : Size of one UDF result
- *N* : Asymmetricity of the network: (bandwidth of the downlink / bandwidth of the uplink.)

On a per- tuple basis, a semi-join will send the (duplicate free) argument columns:

$D * (A * I)$ *(semi-join, bytes transferred on downlink, per tuple average)*

The client will return the results without applying any selections or projections:

$N * D * R$ *(semi-join, bytes transferred on uplink, per tuple average)*

The client-site join will send the full input records, without eliminating duplicates:

I *(client-site join, bytes transferred on downlink, per tuple average)*

The client will return the received records, together with the UDF results, after applying pushable projections and selections:

$N * (I + R) * P * S$ *(client-site join, bytes transferred on uplink, per tuple average)*

The bandwidth cost incurred at the bottleneck link is the maximum of the costs incurred at each link. *N,* the network asymmetricity weighs these costs in the direct comparison. The link with maximum cost will be the link whose used bandwidth is closer to its capacity and who will thus determine the turnaround for the join execution.

3.2.2 Duplicate elimination

The proportion of duplicates present in the input relation influences down- and uplink cost identically. For the semi-join, it reduces the necessary bandwidth because duplicate arguments and the corresponding duplicate results are never transferred. The client-site join cannot exploit the presence of argument duplicates because it transfers the whole record, including the columns on which such duplicates might differ.

Duplicate elimination on the client site could be used with both join methods to reduce the processing time on the client. If sorting is used, the duplicate elimination could be prepared on the server site, but again, without affecting the necessary network bandwidth.

4 Performance Measurements

We present the results of four experiments: We demonstrate the problems of the naive approach by measuring the influence of the pipeline concurrency factor. The next two experiments show the tradeoffs between semi-join and client-site join on a symmetric and an asymmetric network. Finally we show these tradeoffs in their dependence on the size of the returned results for different selectivities.

Our results show that client-site joins are superior to semi-joins for a significant part of the space of UDF applications. Exploiting the tradeoffs between both join methods, especially in the context of asymmetric networks, allows essential performance improvements.

All of our experiments were executed with the server running on a 300Mhz Pentium PC with 130 Mbytes of memory. The client ran as a Java program on a 150Mhz Pentium with 80 Mbytes of memory, connected over a 28.8KBit phone connection. The asymmetric network was modeled on a 10Mbit Ethernet connection by returning *N* times as many bytes as actually stated.

4.1 Concurrency

We evaluated the effect of the concurrency factor on performance for the following simple query:

```
SELECT UDF(R.DataObject) FROM Relation R
```

`Relation` is a table of 100 `DataObjects`, each of the same size. `UDF` is a simple function that returned another object of the same size. Figure 6 gives the overall execution time of the query in seconds, plotted against the concurrency factor (size of the buffer) on the x-axis, for object sizes 100, 500, and 1000 bytes.

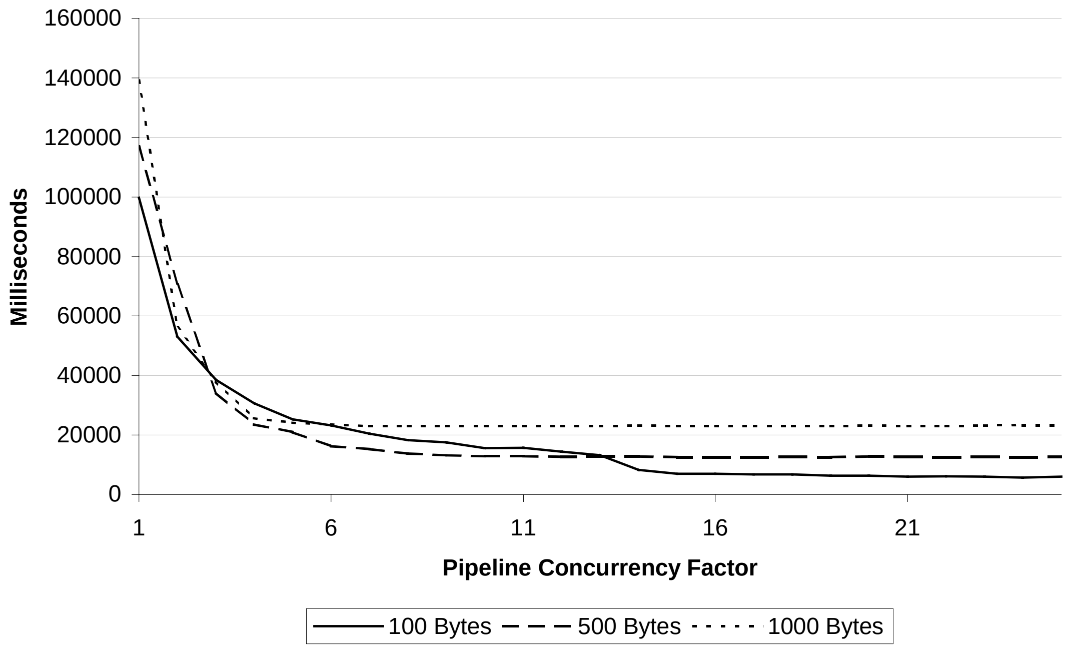

Figure 6: Effect of Concurrency

Our analysis suggested that the optimal concurrency factor is bandwidth times latency: The number of tuples that can be processed concurrently, while one tuple travels through the whole pipeline. Following our assumption, the network is the bottleneck and its bandwidth limits the overall throughput. In this graph, we can observe that the optimal level for 1000 bytes is reached at 5 and for 500 bytes at 10: This would correspond to 5000 bytes as the product of bandwidth and latency. Presumably, for 100 byte object, the optimal concurrency level would be 50.

The presented data were determined with a nonthreaded implementation of the presented architecture: This facilitates the simple manipulation of the concurrency factor. All further experiments ran on an implementation that simply uses different threads for sender and receiver.

4.2 Client-Site Join and Semi-Join on a Symmetric Network

Our analysis suggests that the uplink bandwidth required by the client-site join is linear in the selectivity while the downlink bandwidth is independent of the selectivity. For the total execution time, this means that as long as the downlink is the bottleneck, selectivity will have no effect, but when the uplink becomes the bottleneck, the execution time will increase linearly with selectivity. The semi-join is not affected by a change in selectivity.

We measured the overall execution time for the query in Figure 7. `Relation` has 100 rows, each consisting of two data objects, together of size 1000 bytes. *A* was fixed at 50%: The `Argument` and the NonArgument object were each 500 bytes. *P*, the projection factor is adjusted to the result size, such that: $P*(I+R) = I*(1-A)+R$, meaning that no arguments have to be returned by the client-site join, only the non-argument columns and the results. `UDF1` takes an object from the `Argument` column and returns true or false, while `UDF2` takes the same object and returns a result of known size.

```
SELECT R.Argument, R.NonArgument, UDF2(R.Argument)
FROM   Relation R
WHERE  UDF1(R.Argument)
```

Figure 7: Measured Query

In Figure 8, we plot the overall execution time of the client-site join relative to that of the semi-join against the selectivity of `UDF1` on the x-axis. Thus, the line at *y = 1.0* represents the execution time of the semi-join. We varied the selectivity from 0 to 1.0 and plot curves for result sizes 100, 1000, 2000, and 5000 bytes. The execution time of a semi-join is independent of the selectivity because semi-joins do not apply predicates early on the client. Thus all client-site join execution time values of one curve are given relative to the same constant. In this, as in all other experiments, we set *D=1.*

We will first discuss the *shape* of each curve, meaning the slope of the different linear parts, and then its *height.* It can be observed that for each result size the curve runs flat up to a certain point and from then on rises linearly. For the flat part of the curve the downlink is the bottleneck of the client-site join's execution. Only from a certain selectivity on will its uplink form the bottleneck and thus determine the shape of the curve. For result size 1000 bytes, this point is at selectivity 0.6, when the returned data volume (S * (P*(I+R)) = 0.6 * 1500) approaches the received data volume (I = 1000). The larger the result size, the earlier this point will be reached because the ratio of received to returned data changes in favor of the latter. The received data are independent of the selectivities: As long as the the downlink dominates, the curve is constant. The increasing, right part of the curves is part of a linear function going through the origin of the graphs: At zero selectivity the uplink would incur no cost. Its cost is linear in the amount of data sent on it, which is linear in the selectivity of the predicate.

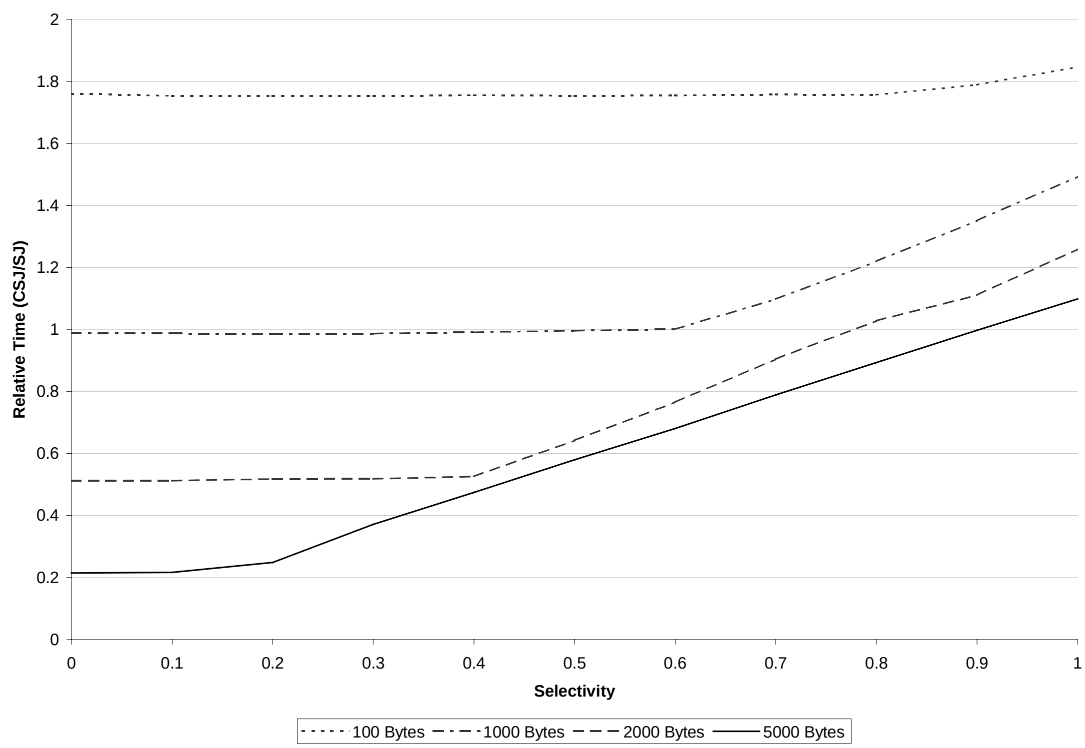

Figure 8: Client-Site Join versus Semi-Join on a Symmetric Network

The height of the curve is influenced by the relative execution time of the semi-join. With larger result sizes the flat part of the curve on the left side of the graph will run deeper, because of the relatively higher costs of the up-link dominated semi-join, compared to the downlink dominated client-site join.

For example, the curve for 2000 goes flat at 0.5 (1000 bytes on s.j.downlink / 2000 bytes on c.s.j.uplink).

4.3 Client-Site Join and Semi-Join on an Asymmetric Network

In this experiment, we explored the same tradeoffs as above in a changed setting: The network is asymmetric with the downlink bandwidth being hundred times as much as that of the uplink (*N=100*). This choice was motivated by assuming a 10Mbit cable connection as a downlink that is multiplexed among a group of cable customers. With a 28.8Kbit uplink this would result in *N = 350* for exclusive cable access and, as a rough estimate, *N = 100* after multiplexing the 10Mbit cable.

The same query as above is executed (Figure 7). The argument columns consist of 4000 bytes and the non-argument columns of 1000 (*A=80%*), and again, only the non-argument columns and the results are returned after the pushable projections (*P*(I+R) = I*(1-A)+R*). The selectivity is varied along the x-axis from 0 to 1 and we give curves for result sizes 500, 1000, and 5000 bytes. The relative execution time of the client-site join with respect to the semi-join is given in Figure 9.

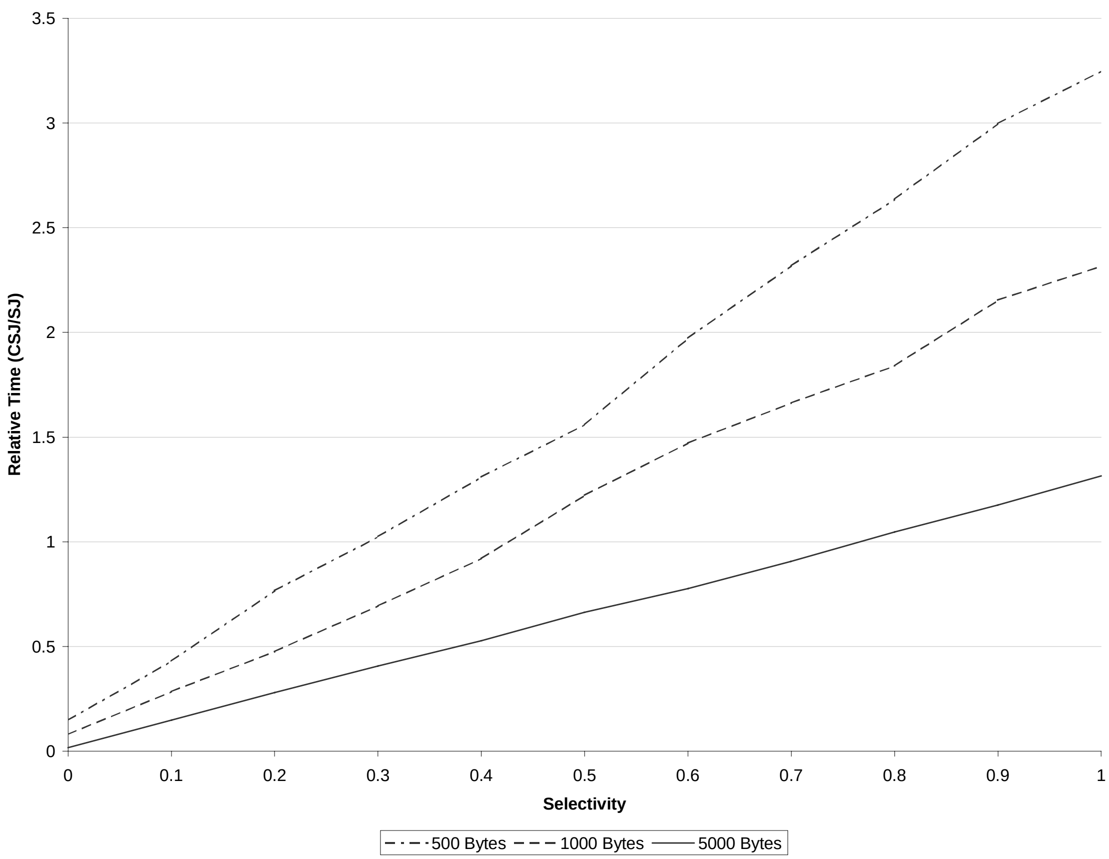

Figure 9: Client-Site Join versus Semi-Join on Asymmetric Network

As our cost model predicts, the bandwidth of the uplink depends linearly on the selectivity. The flat part of the curves in the last graph is absent because the downlink never forms a bottleneck. Our model predicts a selectivity of less than: *I/(N*P*(R+I)) = 0.0083* to make the downlink the bottleneck of the lowest curve (result size 5000 bytes).

4.4 Influence of the Result Size

Finally, we fixed the selectivity *S* and varied the result size *R* along the x-axis from 0 to 2000 bytes. Four different curves are shown, for selectivities 25%, 50%, 75%, and 100%. The argument size was 100 bytes, the overall input size 500 bytes. Again, only non-arguments and results are returned and, as in the second experiment, the network is symmetric. The resulting execution times of the client-site join relative to those of the semi-join are given in Figure 10.

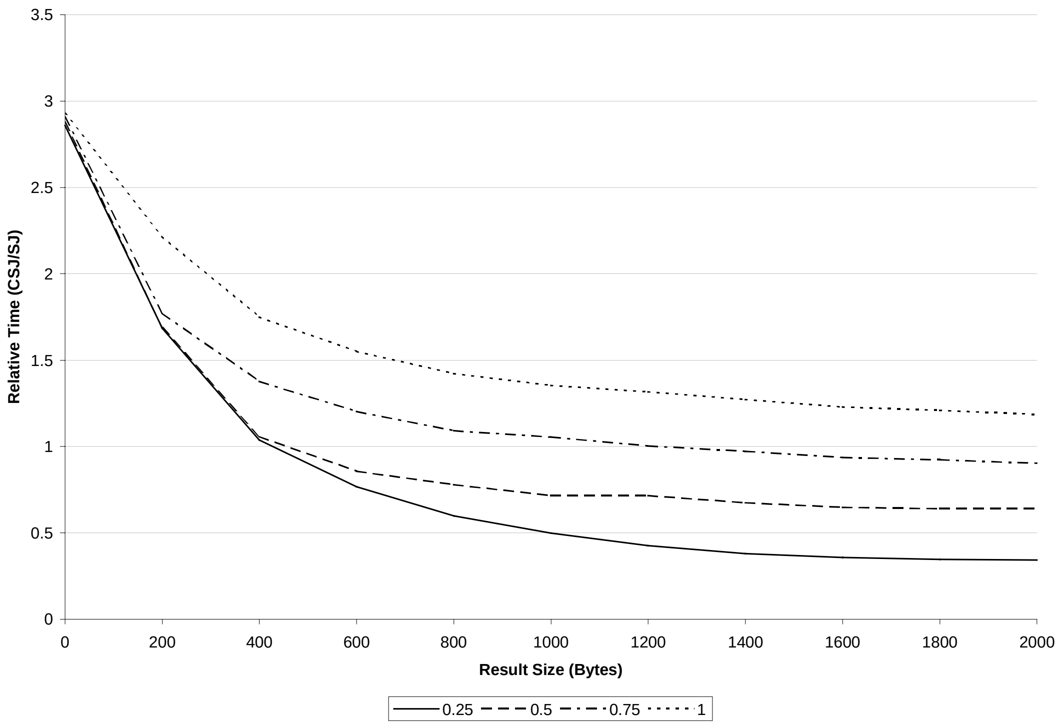

Figure 10 : Influence of the Result Size

It can be seen that the client-site join will only be cheaper if the pushable predicates are selective enough to reduce the uplink stream sufficiently and if the results are large enough to realize the gain in comparison to the records that have to be shipped on the downlink. The steep initial decline of the curve represents the change from a downlink bottleneck to an uplink bottleneck. While the former is disadvantageous for the client-site join, the latter emphasizes the role of pushed down predicates and projections. The crossing points of the curves with the 1.0 line satisfies, as expected, that the client-site join's returned data times the selectivity are equal to the semi-join's returned data. The curve for selectivity one will never cross that line. The curves' slope decreases because the size difference between the client-site joins and the semi-joins returns becomes less significant as results are getting larger. The curves assymptotically approach the horizontal lines that correspond to their selectivity.

5 Query Optimization

We showed that existing UDF execution algorithms are inadequate for client-site UDF queries. Now we show that existing query optimization techniques are also inadequate. There are two reasons for this:

(a) Multiple client-site operations can exhibit interactions that affect their cost. Even for plans with a *single* client-site UDF these interactions are relevant, because the result operator of every plan, which ships the results to the client, can be modeled like a client-site "printing" UDF.

(b) The cost of the client-site join is sensitive to the number of duplicates in its input stream.

The existing approaches rely on the concept of a rank order: Every operation has a rank, defined as its cost per tuple divided over one minus its selectivity. Unless otherwise constrained, expensive operations appear in the plan ordered by ascending rank. The validity of a rank-order optimization algorithms is based on two assumptions that are violated by client-site UDFs:

a) The per-tuple execution cost of an operation is known a priori, *independent of its position* in the query plan.
b) The total execution cost of an operation is its per-tuple cost times the size of the input *after duplicate removal.* UDFs can be pulled up over a join, without suffering additional invocations on duplicates in the argument columns.

Neither assumption is valid for network- intensive client-site UDFs. The cost of a client-site operation is strongly dependent on its location next to other such operations with which it can be combined. And client-site joins as well as combinations of semi-joins are dependent on the number of duplicates.

We propose an extension of the standard System-R optimization algorithm for such queries. As a running example, we will use the query in Figure 11. A client tries to find cases in which his analysis results in the same rating than that of a broker. `Ratings` contains the ratings of many companies' stocks by several brokers.

```
SELECT S.Name, E.BrokerName
FROM   StockQuotes S, Estimations E
WHERE  S.Name = E.CompanyName AND
       ClientAnalysis(S.Quotes) = E.Rating
```

Figure 11: Example Query : Placement of Client-Site UDF `ClientAnalysis`

5.1 UDF Interactions

It is important to observe that the execution costs of a client-site UDF depend on the operations executed before and after it. If a client-site operation's input is produced by another client-site operation, the intermediate result does not have to be shipped back to the server. If such operations share arguments, they can be executed on the client as a group and the arguments are shipped only once. For example, a client-site UDF that is executed immediately before the result operator can be executed together with it, without ever shipping back its results. We will first discuss the case of client-site joins, then that of semi-joins.

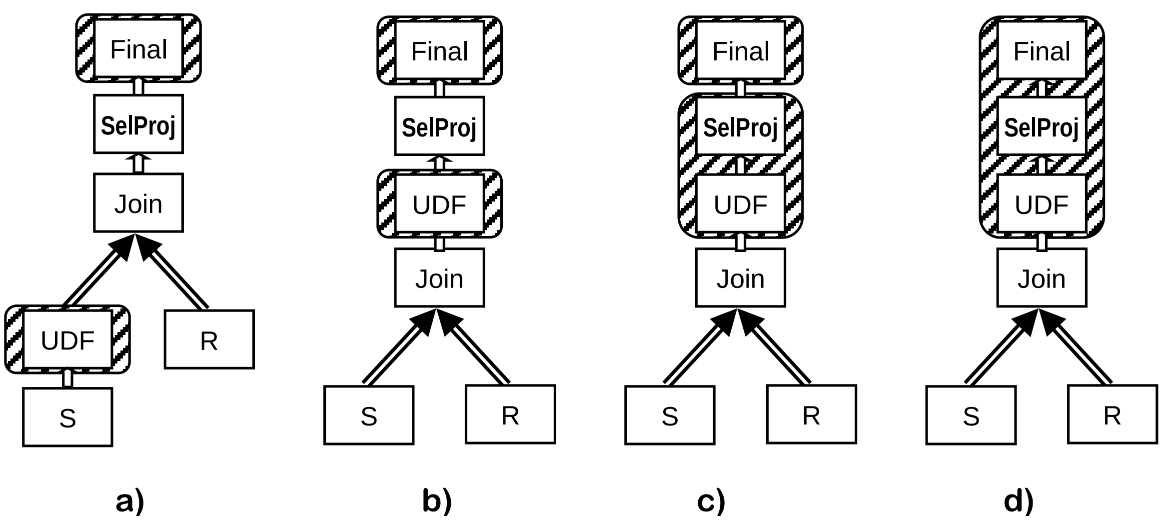

Figure 12 : Possible Plans for the Query in Figure 11

5.1.1 Client-Site Join Interactions

Consider our example from Figure 11 with the possible query plans shown in **Figure 12**. There are only two possible orderings of the operators, one executing the client-site function before the join, one after. In the latter case there are three different options. We describe all four options in more detail and give possible motivations:

a) UDF before the join: This avoids duplicates that the join might generate. The result of the UDF can also be used during the join, for example, to use an index on `Rating`.

b) UDF after the join: The number of tuples and/or the number of distinct argument tuples in the relation might be reduced by the join.

c) UDF and pushable operations after join: If the UDF uses the client-site-join algorithm, the selection can be pushed down to the client site, reducing the size of the result stream. Further, projections may also be pushed to the client. In this example, only `Name` and `BrokerName` of the selected records are returned to the server.

d) UDF combined with result delivery: For many queries, the results need to be delivered to the client (this is not true for INSERT INTO queries). Since there is no other server-site operation between the UDF and the final result operator, the UDF with the pushable operations can be executed in combination with the final operator. This avoids the costs of returning any results from the client and also of shipping the final results.

It can be seen that the locations of UDFs in the query plan (a vs. b) determines the available options for communication cost optimizations: The cost of a UDF application is dependent on the operators before and after it! These locations and the locations of pushable predicates need special consideration during plan optimization. Similar observations can be made about semi-joins, which we consider in the following section.

5.1.2 Semi-Join Interactions

Semi-joins differ from client-site joins in their interactions: Neither the final result operator, nor pushable selections or projections are relevant for grouping. There are three motivations for grouping semi-joins:

- The result of one client-site UDF is input to another. This avoids sending the results back on the uplink and transferring them, with the other arguments of the second UDF, on the downlink. The superset of the arguments is sent to the first and only duplicates on this superset are eliminated.
- The arguments of one function are a subset of the arguments of another. This saves the costs of sending the subset twice, but implies transferring all duplicates that are not duplicates in all of the superset's columns.
- The argument sets of two functions intersect. In this case it is not generally true that we save communication costs when sending the superset instead of the two subsets. Especially, when considering the duplicates sent on each subset because they are not duplicates on the whole superset.

As an example consider the query in Figure 11 with an additional expression in the select clause: `Volatility(S.Quotes, S.FuturePrices)`. The client requests an estimation of the price volatility for the company stocks selected in the query, as computed by the client-site UDF. Some query plans of interest are shown in Figure 13.

The first two options are extensions of Figure 12(a), while the last two are extensions of Figure 12 b) and (c):

a) `Volatility` is pushed down to the location of `ClientAnalysis`, so that both can be executed together: The columns `Quotes` and `Futures` are shipped once for both UDFs. This saves shipping `Quotes` twice, but it does not allow the elimination of all duplicates in this column. Identical quotes that are paired with different `Futures` objects, have to be shipped several times. In this plan, `ClientAnalysis` does not benefit from the join's selectivity, `Volatility` waives both the join's and the selection's selectivities.

b) `ClientAnalysis` is executed before the join, for example, because its result is used for index access to `Estimates`. `Volatility` is executed after the last selection, to benefit from combined selectivity. It is not joined with the result operator as a client-site join because then its arguments would have to be sent with duplicates.

c) If `ClientAnalysis` is moved after the join, it can be executed together with `Volatility`. Both benefit from the join's selectivity, while the duplicates generated by the join in both needed input columns can be eliminated. Again, the input of `ClientAnalysis` input might involve some duplicates.

d) To avoid all duplicates on `Quotes`, `ClientAnalysis` is executed separately, with the selection pushed down. `Volatility` is also not merged with the result operator, to avoid duplicates in its input columns.

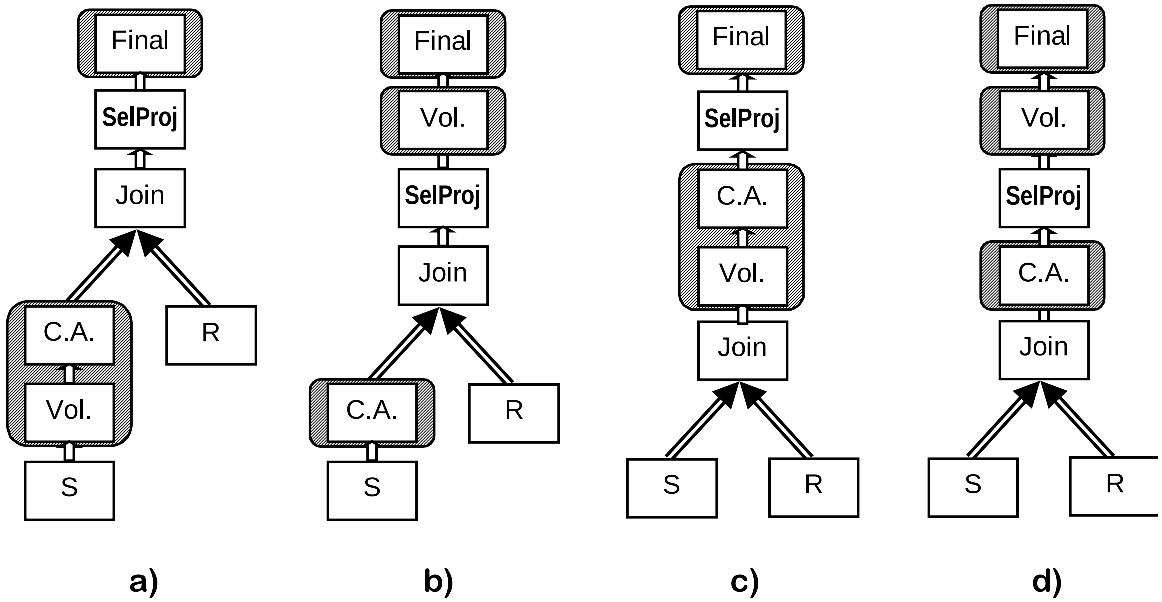

Figure 13 : Possible Plans for the Query in Figure 11 with Additional UDF

5.2 Optimization Algorithm

We will start by presenting the basics of System-R style optimization, then we discuss the standard extensions for expensive server-site UDFs, before we finally present our algorithm.

5.2.1 System-R Optimizer

System R[SAC+79] uses a bottom-up strategy to optimize a query involving the join of N relations[1]. Assume that there are join predicates between every pair of relations (this is not very realistic but one can always assume the existence of a trivially true predicate). Three basic observations influence the algorithm:

- Joins are commutative
- Joins are associative
- The result of a join does not depend on the algorithm used to compute it. Consequently, dynamic programming techniques may be applied.

Initially, the algorithm determines the cheapest plans that access each of the individual relations. In the next step, the algorithm examines all possible joins of two relations and finds the cheapest evaluation plan for each pair. In the next step, it finds the cheapest evaluation plans for each three-relation join. With each step, the sizes of the constructed plans grow until finally, we have the cheapest plan for a join of N relations. At each step, the results from the previous steps are utilized.

This last principle is not totally justified, because the *physical properties* of the result of a join can affect the cost of some subsequent joins (thereby violating the dynamic programming assumptions that allow expensive plans to be pruned). The System R optimizer deals with this by maintaining the cheapest plan for every possibly useful interesting property, thereby growing the search space. These properties were called “interesting orders”, since at the time, sort ordering was the primary property of interest.

The System-R optimizer also applies some heuristics that further limit the plans considered:

[1] A description of the algorithm, relevant to expensive UDF placement, can be found in [CS97].

- Only binary join algorithms are considered. Consequently, a three-relation join evaluation plan involves the combination (i.e. join) of a two-relation join result and a stored relation.
- In order to find the best plans for K-relation join, the only combinations examined use (K-1)-relation joins and stored relations. Other possible combinations (e.g. K-2 and 2) are not considered. The resulting query plans that look like ''left-deep" trees.
- While the intermediate results of a join can act as inputs for another join, they cannot appear as the inner relation of a nested-loops join algorithm.
- Selections and projections are always applied as early as possible, assuming that such operations are cheap.

The optimization algorithm with rank ordering, proposed in [CH97] uses the concept of physical properties to integrate rank-ordered application of expensive operations into this optimization algorithm. The idea is to tag each plan with the set of operations that are not yet applied in the plan. A plan that already applied an expensive UDF should not be pruned because of another, cheaper plan that yet has to apply it. The former can turn out to be optimal because of the early application of the operation, or the latter may be optimal because of the late application. The optimizer cannot decide this and keeps both plans. When there are many expensive UDFs in the query, ranks are used to reduce the number of possibly optimal interesting properties and thus the complexity of the algorithm.

5.2.2 Client-Site Join Optimization

We will first explain our proposed algorithm in terms of client-site joins and introduce analogous techniques for semi-joins later. In this discussion we will only talk about client-site operations, joins, pushable predicates and projections. Our strategy is to treat client-site UDFs in the same way as join operators. This approach has been followed before [LDL] in the case of expensive UDFs, but for client-site operations we also have to consider physical location of the operation (like [FJK96][SA80]).

Our running example will be the construction of the optimal plan for the query in Figure 11, as executed by our optimization algorithm (shown in Figure 15). The steps of the algorithm, iterations of the outermost loop, are shown as horizontal layers in Figure 14.

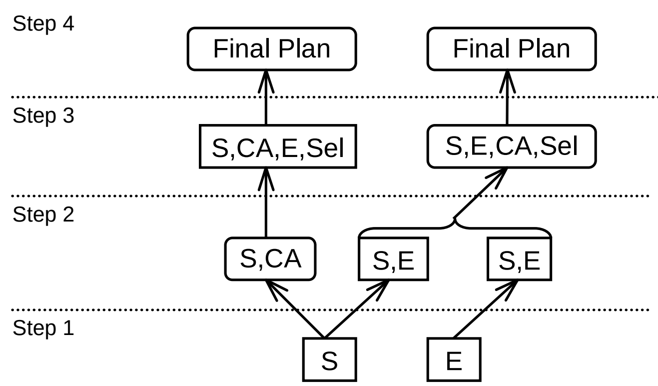

Figure 14: Client-Site Join Optimization of the Query in Figure 11

We introduce a new bi-valued physical property, a plan's *site*, indicating the location of its results: In a server-site plan (cornered boxes), the last applied operation is executed on the server. In a client-site plan (round boxes), the last applied operation is a client-site UDF. As an example for a client-site plan, take the plan that applies `ClientAnalysis` on relation `S`, resulting in a relation residing on the client. Joining `S` with `E` forms a server-site plan because the result of the join resides on the server.

When applying the next operation to a plan, we have to determine the communication costs with respect to the plan's site. A real join applied on a client-site plan requires that the records are shipped from the client to the server, while a client-site function applied on a server-site plan requires the opposite. Take the application of the final result operator to the right plan in step 3: It will not incur any additional communication costs because the relation already resides on the client. Operations have to move the records to the site where they are needed and leave their results on the site of their execution.

To take the final site of a subplan as a *physical property* implies that only subplans that end on the same site will be compared and pruned if suboptimal. To be more precise, only subplans that joined the same set of relations, that applied the same set of client-site operations, and that end up with their result on the same site will be compared and pruned. In Figure 14, pruning happens after steps 2 and 4: in the latter case all plans have the same physical properties after the final operator moved their results to the client.

A client-site UDF is executed by a join with a given inner table -- the virtual UDF table. To unify our handling of virtual and real joins we will see joins as operations with a given inner table. Every relation in the query introduces such a join operator: In our example we have to consider three operations: The join with `S`, the join with `E`, and the client-site join with `ClientAnalysis`. Thus real joins are applied in the same way as UDF joins. The application of a join to a yet empty plan simply results in the base relation of the join. The algorithm for the set of real and virtual joins J_1 to J_m is given in Figure 15.

```
FOR i:=1 TO m DO
{ FOR ALL J ⊆ {J1,...,Jm} s.t. |J|=i DO
  { BestPlan := dummy plan of infinite cost
    FOR ALL j, J’ s.t.: |J’| = i AND
                        {j} ∪ J’ =  J DO
    { P := BestApplication(OptPlan[J’], j)
      IF cost(P) < cost(BestPlan)
      THEN BestPlan := P
    }
  OptPlan[S] := BestPlan
  }
}
RETURN( OptPlan[{O1,...,Om}] )
```

Figure 15 : Client-Site UDF Optimization Algorithm

5.2.3 Semi-Join Optimization

For the semi-join UDF algorithm, a small modification is necessary. We need to capture the fact that the results of plans after a semi-join are *distributed between client and server.* To do so, we introduce locations for each column of the intermediate results as physical properties. As an example consider again the plans for the query of Figure 11, extended with `Volatility(S.Quotes, S.FuturePrices)` in the select clause. We show part of the optimization process in Figure 16, omitting all plans that do not start with the join of `S` and `E`.

The initial plan, S⊗E, can be extended by applying either `ClientAnalysis` or `Volatility`. Each client-site UDF can deliver its result column and its argument columns on the client site, available for any further operation. If `Volatility` is applied first, `ClientAnalysis` can follow without shipping its arguments because its arguments are already on the client.

The application of `Volatility` after `ClientAnalysis`, on the left side of the tree, cannot use the `Quotes` column on the client: Duplicates were eliminated on it that were originally paired with different `FuturePrices` values. Everything has to be shipped back to the server before the adequate columns can be transferred. Similarly, server-site operations, like the selection, always ship everything back to the server before their execution.

The described plan generation happens with the algorithm given in the previous section. All described modifications are an extension of the set of relevant physical properties and new variations for the described execution operators: Any client-site UDF can be applied as a semi-join that is executed duplicate-free, as a semi-join that accepts duplicates to avoid shipping, and as a client-site join. The latter has to return client-site results of semi-joins to the server before it can ship the full records to the client. This is also true for the final result operator.

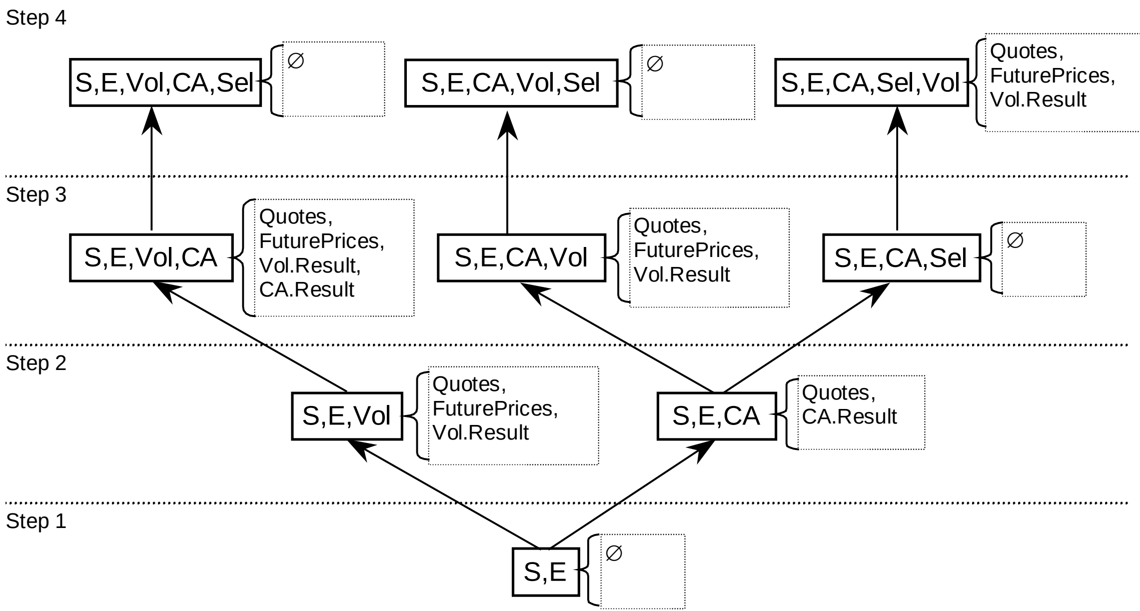

Figure 16 : Semi-Join Optimization for the Query in Figure 11

5.2.4 Features of the Optimizatio n Algorithm

The key characteristics of this algorithm are:

- The number of joins in the plan is $2^{(\#joins+\#c.s.udfs)}$, that is, the algorithm is exponential in the number of real joins plus the number of client site UDFs.
- Simple, pushable selections and projections are not modeled as operations, although they are, where possible, pushed to the client.
- For query nodes that apply client-site UDFs, an additional physical property is introduced: The distributed location of the optimized subplan's result relation: The subset of its columns that resides on the client. If none, server-site operations incur no communication cost -- if all, client-site joins don't have to transfer data. For a certain set of columns that is a superset of an UDF's arguments, there is a choice of using the columns on the client, including possible duplicates, or of returning them and shipping only the arguments, duplicate-free.
- Grouping of client-site operations, motivated by shared arguments or by result dependencies, is integrated in a uniform way, using the location property.

6 Conclusions

Client-site query extensions (UDFs) will play an increasingly important role in extensible database systems due to scalability, confidentiality, and security issues. We demonstrate that existing UDF evaluation and optimization algorithms are inappropriate for client-side UDFs. We present more efficient evaluation algorithms, and we study their performance tradeoffs through implementation in the Cornell PREDATOR database system. We also present a query optimization algorithm that handles the client-site UDFs appropriately and finds an efficient query plan.

Acknowledgements

This work on the Cornell Jaguar project was funded in part through an IBM Faculty Development award and a Microsoft research grant to Praveen Seshadri, through a contract with Rome Air Force Labs (F30602-98-C-0266) and through a grant from the National Science Foundation (IIS-9812020).

Bibliography

[CDY95] S.Chaudhuri, U.Dayal, T.Yan. Join queries with external text sources: Execution an optimization techniques. In Proceedings of the 1995 ACM-SIGMOD Conference on the Management of Data. pages 410-422. San Jose, CA.

[CGK89] D.Chimenti, R.Gamboa, and R.Krishnamurthy. Towards an open architecture for LDL. In Proceedings of the 15th International VLDB Conference, pages 195-203, Amsterdam, August 1989.

[CS93] S.Chaudhuri and K.Shim. Query optimization in the presence of foreign functions. In Proceedings of the 19th International VLDB Conference, Dublin, Ireland, August 1993.

[CS97] S.Chaudhuri and K.Shim. Optimization of queries with user-defined predicates. Technical Report MSR-TR-97-03, Microsoft Research, 1997.

[CSM98] G.Czajikowski, P.Seshadri, and T.Mayr. Resource Control for Database Extensions. Submitted for Publication. 1998.

[FJK96] Michael J. Franklin, Björn Þór Jónsson, Donald Kossmann: Performance Tradeoffs for Client-Server Query Processing. In Proceedings of the 1996 ACM-SIGMOD Conference on the Management of Data, pages 149-160.

[GMHE98] M.Godfrey, T.Mayr, P.Seshadri, and T. von Eicken. Secure and portable database extensibility. In Proceedings of the 1997 ACM-SIGMOD Conference on the Management of Data, pages 390-401, Seattle, WA, June 1998.

[Hel94] J.M.Hellerstein. Practical predicate placement. In Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data, pages 325-335, Minneapolis, MN, May 1994.

[Hel95] J.M.Hellerstein. Optimization and Execution Techniques for Queries with Expensive Methods. PhD thesis, University of Wisconsin, Madison, May 1995.

[HN97] J.M.Hellerstein and J.F.Naughton. Query execution techniques for caching expensive methods. In Proceedings of the 1997 ACM-SIGMOD Conference on the Management of Data, pages 423-434, Tucson, AZ, May 1997.

[HS93] J.M.Hellerstein and M.Stonebraker. Predicate Migration: Optimizing queries with expensive predicates. In Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data, pages 267-276, Washington, D.C., May 1993.

[IK84] Y.E.Ioannidis, R.Ng, K.Shim, and T.Sellis. Parametric query optimization. In Proceedings of the 1990 ACM-SIGMOD Conference on the Management of Data, pages 312-321, Atlantic City, NJ, May 1990.

[KBZ86] R.Krishnamurti, H.Boral, and C.Zanialo. Optimization of nonrecursive queries. In Proceedings of the International VLDB Conference, pages 128-137, Kyoto, Japan, August 1986.

[ML86] L.F.Mackert, G.M.Lohman R* Optimizer Validation and Performance Evaluation for Distributed Queries. VLDB 1986: 149-159.

[S98] P.Seshadri: Enhanced Abstract Data Types in Object-Relational Databases. VLDB Journal 7(3): 130-140 (1998).

[S+79] Patricia G. Selinger, Michel E. Adiba: Access Path Selection in Distributed Database Management Systems. ICOD 1980: 204-215.

[SA80] P.G.Selinger, M.Adiba. Access path selection in distributed database management systems. IBM Research Report RJ2883 (36439). August 1980.

[SI92] A.Swami and B.R.Iyer. A polynomial time algorithm for optimizing join queries. Research Report RJ 8812, IBM Almaden Research Center, June 1992.