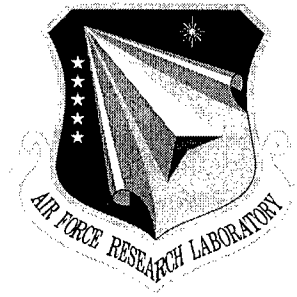


AFRL-IF-RS-TR-2001-198
Final Technical Report
October 2001



JAGUAR: EXTENDING THE PREDATOR DATABASE SYSTEM WITH JAVA

Cornell University

Philippe Bonnet and Johannes Gehrke

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

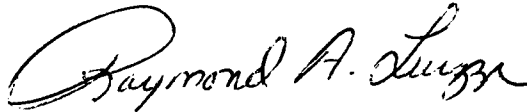
20020308 046

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

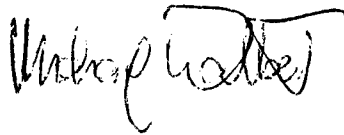
AFRL-IF-RS-TR-2001-198 has been reviewed and is approved for publication.

APPROVED:



RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR:



MICHAEL L. TALBERT, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE OCTOBER 2001		3. REPORT TYPE AND DATES COVERED Final Sep 98 - Aug 00
4. TITLE AND SUBTITLE JAGUAR: EXTENDING THE PREDATOR DATABASE SYSTEM WITH JAVA			5. FUNDING NUMBERS C - F30602-98-C-0266 PE - 62702F PR - 4600 TA - II WU - D1	
6. AUTHOR(S) Philippe Bonnet and Johannes Gehrke				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Cornell University 4130 Upson Hall Ithaca New York 14853			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTD 525 Brooks Road Rome New York 13441-4505			10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2001-198	
11. SUPPLEMENTARY NOTES Air Force Research Laboratory Project Engineer: Raymond A. Liuzzi/IFTD/(315) 330-3577				
12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The Jaguar project is aimed at breaking down the traditional barriers that require SQL query processing to reside on the database server. Indeed, database applications will soon be accessed by large number of clients ranging from Web applications to small-scale personal devices and they will in turn access large collections of data sources ranging from Web servers to mobile sensor devices. In such applications, a large amount of computing resources lie outside the database server: they should be utilized for performance and security reasons. The objective of the Jaguar project was to define portable query execution plans that could be executed either on the server, or on a client or on a remote data source (a web site, an active disk or a sensor device). Java was chosen as a platform for the execution of these portable execution plans. New techniques supporting the execution of portable query plans on the client-site or on the server-site are the major contributions of the Jaguar project. They have been implemented as extensions to the Cornell Predator object-relational system.				
14. SUBJECT TERMS Computers, Database Software, Computer Networks, Architectures			15. NUMBER OF PAGES 44	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1.	Summary	1
2.	The Jaguar System	2
2.1	Architecture	2
2.2	Server-Site and Client-Site Database Extensions	3
2.3	Database Extensions in Java	3
2.4	A Compression Framework for Query Results	3
2.5	Parallel Query Processing in Heterogeneous Environments	3
3.	Publications	4
	Appendix 1. Secure and Portable Database Extensibility	5
	Appendix 2. Client-Site Query Extensions	17

List of Figures

Figure 2:	The Jaguar architecture for ubiquitous query processing	2
-----------	---	---

1. Summary

The Jaguar project aimed at breaking down the traditional barriers that require SQL query processing to reside on the database server. Database applications will soon be accessed by a large number of clients ranging from Web applications to small-scale personal devices and they will in turn access large collections of data sources ranging from Web servers to mobile sensor devices. In such applications, a large amount of computing resources lie outside the database server: they should be utilized for performance and security reasons.

The objective of the Jaguar project was to define portable query execution plans that could be executed either on the server, or on a client or on a remote data source (a web site, an active disk or a sensor device). Java was chosen as a platform for the execution of these portable execution plans.

As a first step, we extended the Cornell Predator object-relational database system so that user defined functions (UDFs) could be defined in Java and executed with any query on the server-site. In this first step, we studied the feasibility of our approach and we explored security and performance issues. In a second step, we developed new techniques for the execution of portable Java UDFs on the client-site. We showed that it is inefficient to use a remote procedure call mechanism to invoke remote UDFs; instead we modeled remote UDFs as relations (we take advantage of the tabular representation of functions) and we reused distributed join techniques to incorporate them efficiently into a portable query execution plan. We successfully applied this new technique to access resources that should remain local to the client-site and also to access data produced by sensor devices. New techniques supporting the execution of portable query plans on the client-site or on the server-site are the major contributions of the Jaguar project. They have been implemented as extensions to the Cornell Predator object-relational system.

2. The Jaguar System

2.1. Architecture

Jaguar extends the Cornell Predator object-relational database engine. The architecture of the Jaguar system is shown in Figure 2. We are transforming the traditional client-server-storage database architecture to a ubiquitous query processing architecture where queries (or query fragments) can run at clients, servers, or storage (shown in Figure 2).

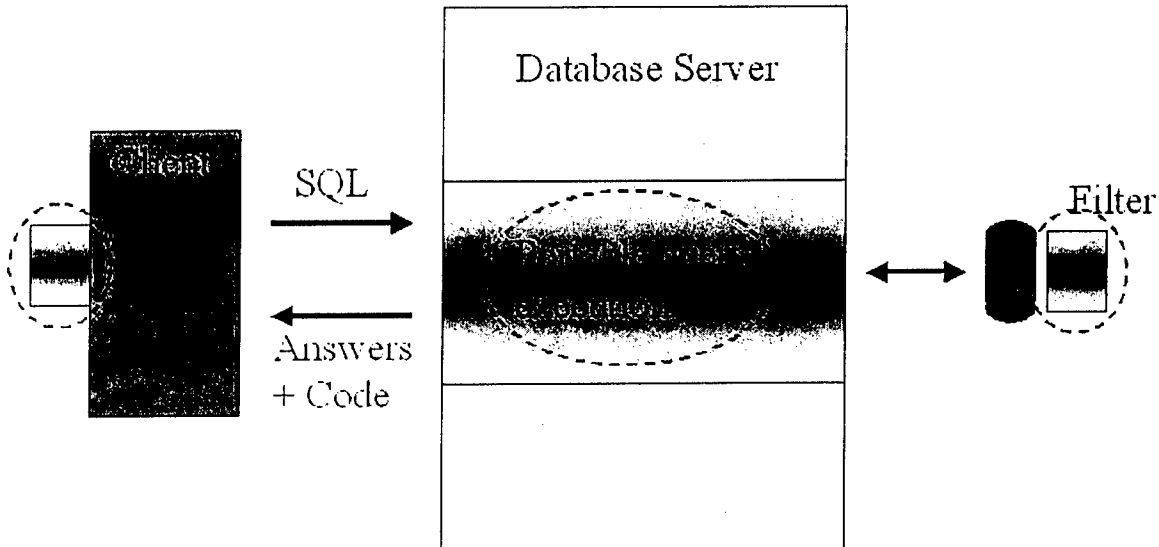


Figure 2: The Jaguar architecture for ubiquitous query processing.

The Cornell Predator System is an extensible object-relational system. The Jaguar project adds extensibility through Java user-defined functions (UDFs) that are executable on the server, or on the client. We have transferred parts of the native Predator query execution engine to Java to allow the migration of subplans to the client and to active storage. Predator, as a system implemented in C++, allows us to explore the integration of portable execution environments into a large, native code system.

The Java programming language is used as one possible underlying platform for portable execution. One motivation is portability: Java Virtual Machines exist on many different platforms, especially as integrated part of most web browsers. Another motivation lies in Java's security features: Untrusted client code can be executed safely on the server, while server code can be downloaded and executed on clients as an applet. Java is widely available and generally considered 'easy to use', which allows extensions by naive users.

The next sections give an overview of the contributions we have made in the Jaguar project in the areas of server-site and client-site server extension, resource control for Java server-site extensions, the usage of compression for moving query results across the

network, and parallel query processing in heterogeneous environments including clusters of servers connected to active storage components.

2.2. Server-Site and Client-Site Database Extensions

How can client functionality be added to a database server? This is what object-relational databases are supposed to accomplish. Our goal is to allow extensions to the database that are written, tested and debugged at the client, and transparently moved to the server. Further, the security and integrity of the server should not be compromised.

Can extensions that have to be executed on the client site be integrated efficiently? Motivations for such client-site extensions are the confidentiality of the client's data and algorithms, client-site specific resources, scalability, and the security of the server. We examined in how far known techniques for expensive server-site UDFs and techniques from distributed query processing apply. Our main observation was that function application should be viewed as a distributed join with a virtual table. Based on this, we developed efficient execution techniques for client-site UDFs and optimization algorithms for queries with such client-site extensions.

The server-site and client-site extensions supported by the Jaguar system were demonstrated at SIGMOD'99.

2.3. Database Extensions in Java

How can the resource consumption of a Java UDF be controlled? This is necessary to avoid Quality of Service attacks and allows to charge clients according to their actual usage. Beyond security, we explored how resource consumption feedback can be used to dynamically optimize Java UDFs. We explored some possible optimizations and examined the achievable performance improvements. Our implementation was based on the JRes Java Resource control. (The JRes project was part of the SLK project at Cornell aiming at providing operating systems infrastructure for extensible servers based on safe-language technology - see <http://www.cs.cornell.edu/slk/> for details.)

2.4. A Compression Framework for Query Results

Portable query processing requires that query computation be movable to different platforms. What is equally important is that the results of queries (which may be very large) also be efficiently moved across networks or stored efficiently. We have developed a comprehensive framework for compressing the results of database queries. It is possible to achieve significantly better compression ratios than a default tool like WinZip would. This is because we exploit the semantics of the query that created the result, and the structured nature of the data.

2.5 Parallel Query Processing in heterogeneous environments

The vision underlying the Jaguar project involves active storage and network components, clients and external sites contributing their data, functionality and processing power to make data processing more scalable, flexible and powerful. These

highly heterogeneous processing environments must be integrated to form reliable and scalable data processing systems.

Performance asymmetries in parallel systems are a significant problem for the classical data-flow paradigm. Past research has focused on the distribution of data. However, this approach is relatively coarse-grained and, while it alleviates sites that form bottlenecks, it does little for bottlenecks of a specific resource within a site. We proposed an extension of the data-flow paradigm that adds flexibility in the positioning of some of the operators and thus adapts the usage of specific resources across the sites of the system. We parallelized the PREDATOR system to study the performances of these techniques. The underlying idea is that parallel systems must be aware of the heterogeneity of their components to process data efficiently and to fully integrate all components.

3. Publications

- M.Godfrey, T.Mayr, P.Seshadri, and T. von Eicken. Secure and Portable Database Extensibility. In Proceedings of the 1998 ACM-SIGMOD Conference on the Management of Data, Seattle, WA, June 1998.
- G.Czajkowski, T.Mayr, P.Seshadri, and T.von Eicken. Resource Control for Database Extensions. COOTS'99.
- Tobias Mayr and Praveen Seshadri. Client-Site Query Extensions. In Proceedings of the 1999 ACM-SIGMOD Conference 1999, Philadelphia, PA, May 1999.
- Philippe Bonnet, Kyle Buza, Zhiyuan Chen, Victor Cheng, Randolph Chung, Takako M. Hickey, Ryan Kennedy, Daniel Mahashin, Tobias Mayr, Ivan Oprenca, Praveen Seshadri, Hubert Siu: The Cornell Jaguar System: Adding Mobility to PREDATOR. In Proceedings of the 1999 ACM-SIGMOD Conference 1999, Philadelphia, PA, May 1999.
- Tobias Mayr, Philippe Bonnet, Johannes Gehrke, Praveen Seshadri. Query Processing with Heterogeneous Resources. Technical Report TR00-1790, Cornell University, Computer Science Department, Ithaca, NY, March 2000.
- Z.Chen, P.Seshadri. An Algebraic Compression Framework for Query Results. In Proceedings of the International Conference on Data Engineering ICDE'00, San Diego, CA, March, 1999

Two publications are attached. They describe the heart of the Jaguar system: server-site and client-site database extensions in Java.

Secure and Portable Database Extensibility

Michael Godfrey

Tobias Mayr

Praveen Seshadri

Thorsten von Eicken

Computer Science Department

Cornell University, Ithaca, NY 14853

{migod,mayr,praveen,tve}@cs.cornell.edu

Abstract

The functionality of extensible database servers can be augmented by user-defined functions (UDFs). However, the server's security and stability are concerns whenever new code is incorporated. Recently, there has been interest in the use of Java for database extensibility. This raises several questions: Does Java solve the security problems? How does it affect efficiency?

We explore the tradeoffs involved in extending the PREDATOR object-relational database server using Java. We also describe some interesting details of our implementation. The issues examined in our study are security, efficiency, and portability. Our performance experiments compare Java-based extensibility with traditional alternatives in the native language of the server. We explore a variety of UDFs that differ in the amount of computation involved and in the quantity of data accessed. We also qualitatively compare the security and portability of the different alternatives. Our conclusion is that Java-based UDFs are a viable approach in terms of performance. However, there may be challenging design issues in integrating Java UDFs with existing database systems.

1 Introduction

In an extensible DBMS, the database server can be extended dynamically with new functionality. An important class of such systems are "universal" database servers (*e.g.*, Informix, DB2, Oracle 8) which support user-defined functions (UDFs). While extensibility increases the functionality and flexibility of such a system, there are also serious concerns with respect to security. The focus of this paper is on the deployment of extensible client-server database technology in a user environment such as the World Wide Web (WWW). For example, consider a database of stock market data that is accessible through a web site. A valid user is any amateur

investor with a web browser, a credit card, and an investment formula *InvestVal*. The following query would then find technology stocks of interest to the user:

```
SELECT *
FROM   Stocks S
WHERE  S.type = 'tech' and
       InvestVal(S.history) > 5;
```

Here, *InvestVal* is a user-defined function. Ideally, it should be possible (and relatively straightforward) for a large number of such users in a web environment to create their own UDFs and use them within SQL queries. If there are many users, each desiring to extend the system without special knowledge about its architecture, several issues arise:

- **Security:** Since the UDFs are supplied by unknown or untrusted clients, the DBMS must be wary of UDFs that might crash the database system, that modify its files or memory directly, circumventing the authorization mechanisms, or that monopolize CPU, memory or disk resources leading to a reduction in DBMS performance (*i.e.*, denial of service). Even if the developer of a UDF is not malicious, the new code might inadvertently cause some of these problems. Clearly, some security mechanism is needed.
- **Portability:** How portable are the UDFs and how easy are they to develop? Users need to be able to develop, test and debug their UDFs on their local machines. It should then be possible to register the UDFs with the server. Do the security mechanisms adversely affect the portability and ease of extensibility by users?
- **Efficiency:** How does the security mechanism affect the performance of queries? Does the portability of UDFs affect their efficient execution?

Until recently, the UDF extensibility mechanisms used in database systems have been unsatisfactory with respect to security and portability. However, with the growing acceptance of Java as a relatively secure and portable programming language, the question arises: can the use of Java aid database extensibility? We are exploring this question through implementation and performance measurement in the PREDATOR OR-DBMS[SLR97].

Specifically, this work is performed in the context of the *Jaguar* project which explores various benefits of incorporating Java into PREDATOR. The motivation of the project is the next-generation of database applications that will be deployed over the web. In such applications, a large number of physically distributed end-users working on diverse

platforms interact with the database server through their web browsers. Because of the large user community with diverse needs, the utility of UDFs increases, along with concerns for the security of the system. In this environment, Java seems a good choice as a language for UDFs, because Java byte code can be run with security restrictions within Java Virtual Machines (JVMs) supported by web browsers on diverse platforms. The full scope of the project envisions UDFs which must be run exclusively at the client, or at the server, or at either site. This paper represents our initial work on this subject, and is limited to studying the execution of UDFs at the database server.

Many vendors of universal database servers are in the process of adding Java-based extensibility [Nor97]. However, to the best of our knowledge, there has been no study of the design needed or of the tradeoffs underlying various design decisions. This paper presents such a qualitative study, and a quantitative comparison of Java-based UDFs with other UDF technologies. The experimental conclusions are consistent with results from the Java benchmarking community [NCW98].

- Java UDFs suffer marginally in performance compared to native UDFs when the functions are computationally intensive. Given current trends in JIT compiler technology, we expect the difference in computation time to become insignificant.
- For functions with significant data accesses, Java exhibits relatively poor performance because of run-time checks. However, this is a reasonable price to pay for security. Our experiments also indicate that when analogous run-time checks are added to native code UDFs that run outside of the server, performance is comparable to (but still somewhat better than) that of Java UDFs.

The paper also discusses specific issues that arise when integrating Java into a typical database server. Although the Java language has security features, current Java environments lack resource control mechanisms needed to fully insulate the server from malicious or buggy UDFs. Consequently, some traditional security mechanisms are still needed to protect the resources of the server. Further, many database servers use proprietary implementations of operating system features like threads. The server-side support for Java UDFs can be non-trivial, since the Java virtual machine can interact undesirably with the database operating system. Consequently, it may be undesirable to embed an off-the-shelf Java Virtual Machine within the database server. Finally, we present the implementation details in PREDATOR that allow Java UDFs to be developed in a portable fashion, so that they can be used at either client or server.

2 Related Technologies

In this section, we outline research and technology relevant to this paper. We divide the work into four categories: (a) web-based database deployment (b) work on database extensibility, (c) work on secure kernel extensions in operating systems, and (d) work on safe programming languages such as Java.

2.1 Web-Based Database Deployment

The architectures of web-based database applications fall into two broad categories: Two-Tier and Three-Tier architectures. In both categories, a database server runs on a

machine accessible via the Internet, and user interact with web browsers on their local machines.

In a Two-Tier architecture, a Java applet running within the web browser also acts as the database client, meaning that it directly connects to the database server, sends requests to the server and displays the results to the user. This resembles the familiar "query-shipping" architecture of client-server database systems [FJK96]. The Java applets that act as client programs are downloaded from a web server (i.e., HTTP server) running on the same machine as the database server. In a Three-Tier architecture, the work of the client program is divided into two components: presentation and program logic. The program logic is abstracted into a separate tier of software which usually runs on the same machine as the web server (and is sometimes implemented as an extension of the web server). This "middleware" tier is responsible for connecting to the database server, issuing queries and receiving replies. The presentation tier runs within the user's browser and handles the graphical input and output functionality. In such an environment, the application developers who build the middleware are typically the "users" who would create UDFs. Our work applies to applications developed using either architecture; however, for the rest of the paper, we will assume the simpler Two-Tier architecture.

2.2 Database Extensibility

Since the early 1980s, database servers have been built to be *extensible*; that is, to allow new application-specific functionality to be incorporated. While extensibility mechanisms were developed in both object-relational (OR) and object-oriented (OO) databases, similar issues apply in both categories of systems. In this paper, we focus on OR-DBMS systems, because they are the dominant commercial database systems, and because PREDATOR falls into this category. However, our results apply equally to OO-DBMSs as well.

While some research has addressed the ability to add new data types [Sto86, SRG83] and new access methods [SRH90, HCL⁺90], most extensible commercial DBMSs and large research prototypes have been built to support user-defined functions (UDFs) that can be added to the server and accessed within SQL queries. The motivation for server-side extensibility (rather than implementing the same functionality purely at the database client) is efficiency; a user-defined predicate could greatly reduce query execution time if applied at the early stages of a query evaluation plan at the server. Further, this may lead to a smaller data transfer to the client over the network.

Given the focus on efficiency, most research on UDFs has investigated the interaction between database query optimization and UDFs. Specifically, cost-based query optimization algorithms have been developed to "place" UDFs within query plans [Hel95, Jhi88]. Some recent research has explored the possibility of evaluating queries partially at the server and partially at the client (this has been called "hybrid-shipping") [FJK96]. However, this work has not been applied to extensible systems. Portability and ease of extensibility have largely been neglected by current OR-DBMS technology.

Traditionally, it has been assumed that most database extensions would be written by authorized and experienced "DB Developers", and not by naive users. This assumption was reasonable because extending a database server required non-trivial technical knowledge, and because few automatic mechanisms were available to verify the safety of

untrusted code. Consequently, a large "third-party vendor" industry has evolved around the relational database industry, developing and selling database extensions (e.g., Virage, Verity). Commercial extensible database systems usually provide three options to those customers who prefer to write UDFs themselves: (a) incorporating UDFs directly into the server (and thereby incurring the substantial risks that this approach entails), (b) running UDFs in a separate process at the server, providing some simple operating system security guarantees, or (c) running UDFs on the client-side in a client environment that mimics the server environment. We describe these options in detail in Section 3.

2.3 Secure Kernel Extensions

The operating systems community has explored the issue of security and performance in the context of kernel extensions. The main sources of security violations considered are illegal memory accesses and the unauthorized invocation of procedures. One proposed technique is to use safe languages to write the extensions, and to ensure at compile and link time that the extensions are safe. The Spin project [Ber95], for example, uses a variant of Modula-3 and a sophisticated linker to provide the desired protection. Another proposed mechanism, called Software Fault Isolation (SFI) [WLAG93], instruments the extension code with run-time checks to ensure that all memory access are valid (usually by checking the higher order bits of each address to ensure that it lies within a specific range). This work on kernel extension has recently seen renewed interest with particular emphasis on extending applications using similar techniques. Extensible web servers are a prime example, since issues such as portability and ease of use are especially important. When extending a server process, another option is to run the extension code in a separate process and use a combination of hardware and operating system protection mechanisms to "sandbox" the code; the virtual memory hardware prevents unauthorized memory accesses, and system call interception examines the legality of any interaction between the extension code and the environment.

One of the shortcomings of all the work on extensions we are aware of is that only the safety of memory accesses and control transfers is taken into account. In particular, the memory, CPU, and I/O resource usage of individual extensions are not monitored or policed, and this makes simple denial-of-service attacks (or simple resource over-consumption) possible.

2.4 Safe Languages

Strongly typed languages such as Java, Modula-3, and ML enforce safety of memory accesses at the object level¹. This finer granularity makes it possible to share data structures between the system core and the extensions. Access to shared data structures is confined to well-defined methods that cannot cause system exceptions. Additional mechanisms allow the system designer to limit the extension's access rights to the necessary minimum².

¹In a *strongly typed* language each identifier has a type that can be determined at compile time. Any access using such an identifier has to accord to the rules of that type. The necessary information that cannot be determined statically, like array bounds and dynamic casts, is checked at runtime (for a survey of type systems, see [Car97]).

²The security community calls this the 'least privilege' principle [SS75]. Every user is granted the least set of privileges necessary.

Safe languages depend on the trustworthiness of their compilers: the compiled code is guaranteed to have no invalid memory accesses and perform no invalid jumps. Unfortunately, these properties cannot, in general, be verified on resulting compiled code because the type information of the source program is stripped off during compilation. Possible solutions to this problem are the addition of a verifiable certificate to the compiled code either in the form of proof carrying code [Nec97] or as typed assembly language [MWCG98].

Another approach is the use of typed intermediate code as the target language for compilation. This code can be verified and executed by platform-specific interpreters while the code itself remains platform independent. The safety of strongly-typed languages is preserved without the need for a trusted compiler. The negatives of this approach include the need for and overhead of an interpreter on each platform, and the overhead of verifying the type-safety of the code. Java uses exactly this design: source programs are compiled into Java bytecode that is verified by the Java virtual machine (JVM) when loaded. Typically, the JVM also compiles parts of the byte codes to machine code before execution.

Since the JVM is a controlled execution environment, it can apply further constraints to the executed programs, including absolute bounds on the memory usage (for example, the JVM in the Netscape 4.0 browser uses a limit of 4MB for the memory usage of Java applets). However, the current JVMs do not provide any form of generic resource management.

2.5 Contrast with Databases

Database systems provide an attractive application environment for user extensions, and therefore some of the work from other areas mentioned in this section is applicable to DBMS UDFs as well. However, there are some subtle differences in perspective:

- In the case of database systems, the portability of the UDFs is an important consideration. The users who are developing UDFs may have different hardware/OS platforms.
- The portability of the entire DBMS server is also a concern; it is undesirable to tie the UDF mechanism to a specific hardware/OS platform.
- In OS research, there is usually some concern at the initial overhead associated with running new code (e.g., time to start a new process). This may not be a concern in a database system, since the cost can be amortized over several invocations of the UDF on an entire relation of tuples. Similarly, the overhead associated with compilation of new code is often not a concern, since it can be performed offline.
- In OS research, there is usually concern over the per-invocation overhead for new code (e.g., message passing overhead). Since there are several invocations of the UDF in a database environment, it may be possible to reduce the overhead through batching.

3 UDF Design Alternatives

We now examine the various design alternatives for adding UDFs to a DBMS. Specifically, we examine two broad issues: *Location* (i.e., where the UDF runs), and *Language* (i.e.,

how the UDF is specified). For each design alternative, we are interested in its effect on efficiency, security, and ease of use. We assume that the database server is written in a language (like C or C++) that is compiled and optimized to platform-dependent machine code. We call this language “native” in contrast to languages with platform-independent portable code, like Java. The clients are not necessarily implemented in the native language and may run on diverse platforms.

Location: There are three alternatives.

- The UDF runs at the server site, within the server process.
- The UDF runs at the server site, in a process isolated from the server.
- The UDF runs at the client site.³

Language: The UDF could be written in the native language of the DBMS or in a different language. If the UDF is run at the client, the availability of language tools (compilers, interpreters, etc.) at the client is an important consideration. Languages that are supported on a wide range of clients are obviously preferable. If the UDF is run at the server site within the server process, there must be some interface mechanism from the native language to the UDF language.

To make the discussion concrete, we will assume in this paper that the native language of the DBMS is C++,⁴ and we will consider C++ and Java as representative UDF languages. These assumptions also correspond to our implementation. Our results with respect to C++ should generalize to any native language that is compiled into platform-dependent machine code without strong security features like type and array bounds checking.

3.1 Client-Side UDF Execution

The client-side execution of a UDF is obviously secure for the server; however it can lead to unacceptably poor performance. For example, consider a function REDNESS(I) that computes the percentage of red pixels in image I. The following query finds images of bright sunsets from upstate New York:

```
SELECT *
FROM Sunsets S
WHERE REDNESS(S.picture) > 0.7 and
      S.location = 'fingerlakes'
```

If the UDF were not available at the server, all the images would need to be shipped to the client where their “redness” would be checked as a post-processing filter. This would correspond to the “data-shipping” approach used by object-oriented databases [Fra96] which is known to be a poor choice for certain queries, as both the server and the network perform significant unnecessary work. An alternative strategy is for the server to contact the client for each UDF execution. This too has obvious drawbacks in the latency of many such calls (UDFs are often applied to each tuple of a relation) and the cost of shipping the function

³A fourth alternative is for the UDF to run at some intermediate site. However, we consider this equivalent to running it at the client site, since the advantages of server-side execution as well as the connected security problems are not present.

⁴Most database servers including PREDATOR are written in C or C++, making this a reasonable assumption. In an interesting development, a few research projects and small companies are building database systems totally in Java [Cim97].

arguments to the client. A further problem which is often overlooked is that UDFs may require access to other functions and facilities in the database server (for example, to store intermediate results). Consequently, we will focus on server-side UDFs in this paper. In future work, we intend to explore client-side UDFs and find query optimization techniques to choose between server-side and client-side execution.

3.2 Server-Side UDF Execution

Table 1 shows the design space for server-side UDFs. There are four possible designs: the language of the UDF can be the native server language or a non-native language, and the UDF can be integrated within the same process or in an isolated process.

Language	Same Process	Different Process
Native (C++)	Design 1 (C++ Integrated)	Design 2 (C++ Isolated)
Non-Native (Java)	Design 3 (Java Integrated)	Design 4 (Java Isolated)

Table 1: Design Space for Server-Side UDFs

Clearly, *Design 1* will have the best performance of all the options since it essentially corresponds to hard-coding the UDF into the server. However, the obvious concern is that system security might be compromised. Buggy UDF code could cause the server to crash, or otherwise result in denial-of-service to other clients of the DBMS. Malicious code could modify the server’s memory data structures or even the database contents on the local disks. Low-level OS techniques such as software fault isolation (see Section 2.3) can address only some of these concerns. Additionally, it may be difficult for a client to develop a UDF in the server’s native language without access to the server’s compilers and its environment.

Using *Design 2*, one could prevent the UDF from directly crashing the server process. However, the UDF could still compromise security by modifying files or killing the server. While *Design 2* is less efficient than *Design 1*, the concerns about ease of use (or lack thereof) are similar. One of the attractions of *Design 2* is that since the UDF computation occurs in a separate process, system call interception techniques can be used to control its behavior (see Section 2.3).

This paper explores the possibilities of *Design 3*, comparing it to the other alternatives. A Java UDF has some very desirable properties: it is portable and supported on most platforms. With an adequate environment on the client and the server side, the UDF can be developed and tested at the client and then migrated to the server. In Section 6, we describe such an environment built in PREDATOR. Because Java was designed with the intent to allow secure and dynamic extensibility in a network environment, the addition of an UDF and its migration between client and server is well supported by the language features (see Section 6). However, there are some possible drawbacks with Java UDFs. Java code may run more slowly than corresponding native code. Further, whenever the language boundary is crossed, there is an “impedance mismatch” that may be expensive⁵. This is usually reflected in the efficiency of the system. Note

⁵In our case, the impedance mismatch is incurred by using the Java native interfacing mechanism (e.g., JNI). There are different implementations available from Sun [JNI] and Microsoft [RNI].

that the language boundary needs to be crossed for each UDF invocation, and there may be several such invocations.

In this paper, we quantify the efficiency tradeoffs between the design alternatives, so that database developers and UDF builders may balance them against the qualitative advantages in the areas of security and portability. We do not consider Design 4 explicitly — we assume that its behavior can be extrapolated as a combination of Design 2 and Design 3.

4 Implementation in PREDATOR

PREDATOR is an object-relational database system developed at Cornell [SLR97]. It provides a query processing engine on top of the Shore storage manager [CDF⁺94]. The server is a single multi-threaded process, with at least one thread per connected client. While the server is written in C++, clients can be written in several languages, including C++ and Java. Specifically, considerable effort has been invested in building Java applet clients that can run within web browsers and connect directly with the database server [PS97].

The feature of PREDATOR most relevant to this paper is the ability to specify and integrate UDFs. The original implementation supports only Design 1 (i.e., UDFs implemented in C++ and integrated into the server process). No protection mechanism (like software fault isolation) was used to ensure that the UDF is well-behaved. From published research on the subject [WLAG93], we expect such a mechanism to add an overhead of approximately 25%. For the purposes of this study, we have also implemented Design 2 (C++ UDFs run in a separate process) and Design 3 (Java UDFs run within the server process). We now discuss these implementations. The main details of interest are the mechanisms used to pass data/parameters to and results from the UDF. Further, some UDFs may require additional communication with the database server. For example, a UDF that extracts pixel (i, j) of an image may be given a handle to the image, rather than the entire image. The UDF will then need to ask the server for the appropriate data, based on the parameters i and j . We call such requests “callbacks”.

The actual mechanism used to load UDFs is not relevant to this paper; either recompilation or dynamic loading can be used. We assume that UDFs are free of side-effects; without this assumption, it is difficult to describe the semantics of an SQL query that uses a UDF. Since PREDATOR is not a parallel OR-DBMS, all expressions (including UDFs) are evaluated in a serial manner.

4.1 Isolated Execution of Native UDFs

We added the ability to execute C++ UDFs in a separate process from the server. When a query is optimized, one remote executor process is assigned to each UDF in the query. These executors could be assigned from a pre-allocated pool, although in our implementation, they are created once per query (not once per function invocation). The task of a remote executor is simple: it receives a request from the server to evaluate the UDF, performs the evaluation, and then returns the evaluated result to the server. Communication between the server and the remote executors happens through shared memory. The server copies the function arguments into shared memory, and “sends” a request by releasing a semaphore. The remote executor, which was blocked trying to acquire the semaphore, now executes the function and

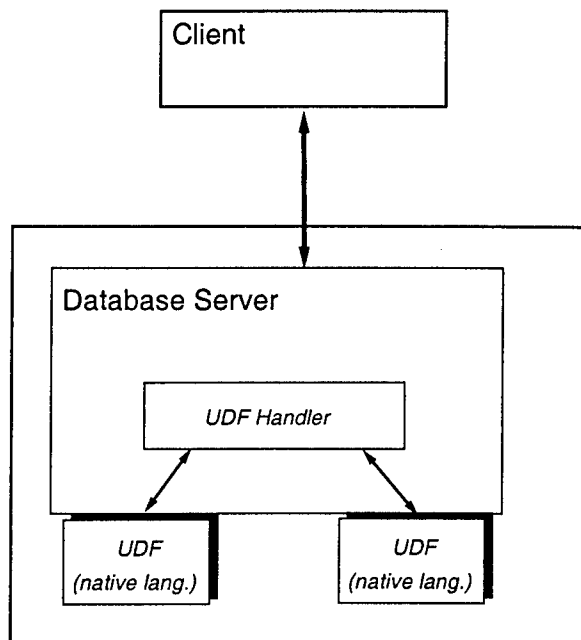


Figure 1: Design 1: Integrated Native UDFs

places the results back into shared memory. The hand-off for callback requests and for the final answer return also occur through a semaphore in shared memory.

We expect that there will be some overhead associated with the synchronization and process switching. This overhead will be independent of the computational complexity of the UDF, but possibly affected by the size of the data (arguments and results) that has to be passed through shared memory.

4.2 Integrated Execution of Java UDFs

In our implementation, Java functions are invoked from within the server using the Java Native Interface (JNI) provided as part of Sun's Java Development Kit (JDK) 1.1 [JNI]. The first step is to instantiate a Java Virtual Machine (JVM) as a C++ object. Any classes that need to be used should have been compiled from Java source (.java files) to Java bytecodes (.class files). The classes are loaded into the JVM using a specified interface. When methods of the classes need to be executed, they are invoked through the JNI interface. Parameters that need to be passed must first be mapped to Java objects.

The creation of a JVM is a heavyweight operation. Consequently, a single JVM is created when the database server starts up, and is used until shutdown. Each Java UDF is packaged as a method within its own class. If a query involves a Java UDF, the corresponding class is loaded once for the whole query execution.

The translation of data (arguments and results) requires the use of further interfaces of the JVM. Callbacks from the Java UDF to the server occur through the “native method” feature of Java. There are a number of details associated with the implementation of support for Java UDFs. Importantly, security mechanisms can prevent UDFs from performing unauthorized functions. We describe these details in Section 6.

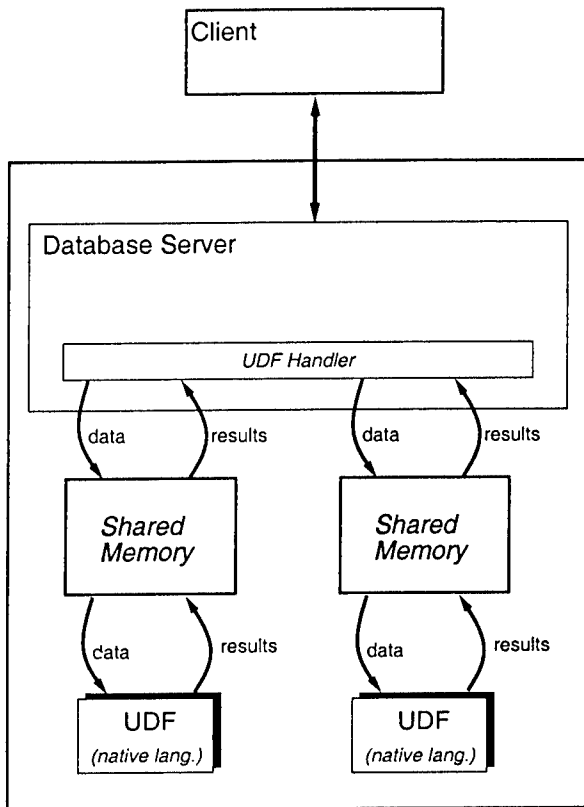


Figure 2: Design 2: Isolated Native UDFs

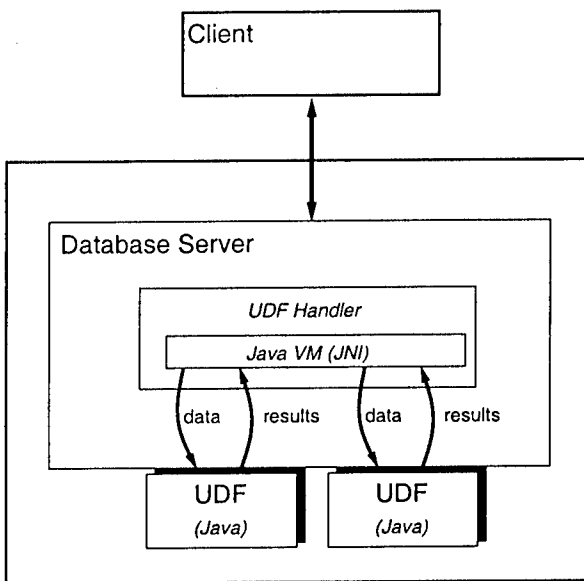


Figure 3: Design 3: Java UDFs

5 A Performance Study

We now present a performance comparison of three implementations of UDF support:

1. Design 1: C++ within the server process [Marked "C++" in the graphs]
2. Design 2: C++ in a separate (isolated) process [Marked "IC++"]
3. Design 3: Java within the server process using the JNI from Sun's JDK 1.1.4 [Marked "JNI"]

The purpose of the experiments was to explore the relative performance of the different UDF designs while varying three broad parameters:

- *Amount of Computation*: How does the computational complexity of the UDF affect the relative performance?
- *Amount of Data*: How does the total amount of data manipulated by the UDF (as parameters, callbacks, and result) affect the relative performance?
- *Number of Callbacks*: How does the number of callbacks from the UDF to the database server affect the relative performance?

The three UDF designs were implemented in PREDATOR, and experiments were run on a Sparc20 with 64MB of memory running Solaris 2.6. In all cases, the JVM included a JIT compiler.

5.1 Experimental Design

Since UDFs can vary widely, the first decision to be made is: how does one choose representatives of real UDFs? Real UDFs may vary from something as simple as an arithmetic operation on integer arguments, to something as complex as an image transformation. We used a "generic" UDF that takes four parameters (ByteArray, NumDataIndepComps, NumDataDepComps, NumCallbacks) and returns an integer.

- The first argument (ByteArray) is an array of bytes of variable size. This models all the data passed as parameters to the UDF and during callback requests. By varying the size of the bytearray, we explore the effect of variable data access.
- The second argument (NumDataIndepComps) is an integer that controls the amount of "data independent" computation in the UDF. The computation within the UDF performs a simple integer addition operation several times within a loop — the number of iterations is specified by NumDataIndepComps.
- There is also a separate loop in which the entire bytearray is repeatedly iterated over, as many times as specified by NumDataDepComps, the third parameter. This is meant to model many real UDFs (such as image transformations) in which the amount of computation depends on the size of the parameters.
- The fourth parameter (NumCallbacks) specifies the number of callback requests that the UDF makes to the database server during its execution. No data is actually transferred during the callback; instead, all data transfers are modeled in the first parameter (ByteArray). While this is slightly inaccurate (real callbacks involve the transfer of data), we chose this model for its simplicity.

The simplest UDF has values of 0 for its second, third and fourth parameters. In all our experiments, parameter values are 0 unless otherwise specified.

In all our experiments, we used three relations of cardinality 10,000. Each relation has an attribute of type ByteArray and all the bytearrays in tuples of the same relation are of the same size. Relations Rel1, Rel100, and Rel10000 have byte arrays of size 1, 100, 10000 bytes respectively in each tuple. The basic query run for each experiment is:

```
SELECT UDF(R.ByteArray, NumDataIndepComps,
          NumDataDepComps, NumCallbacks)
FROM Rel* R
WHERE <condition>
```

We vary the number of UDFs applied by specifying restrictive (and inexpensive) predicates in the WHERE clause. In all experiments, our goal is to isolate the cost of applying the UDFs and ignore the basic cost of scanning the relations. All the graphs measure response time along the Y-axis, while a single parameter is varied along the X-axis.

5.2 Calibration

The first two experiments act as calibration for the remaining measurements. We first measure the basic cost of executing the query in Figure 5.1 with a trivial integrated C++ function that does no work. In Figure 4, the number of UDF invocations is varied along the X-axis. The different lines correspond to different sizes of bytearrays in the relations (the larger bytearrays being more expensive to access). These numbers represent the basic system costs that we subtract from the later measured timings to isolate the effects of UDFs. In most experiments, we will use 10,000 UDF invocations — the last point on the X-axis.

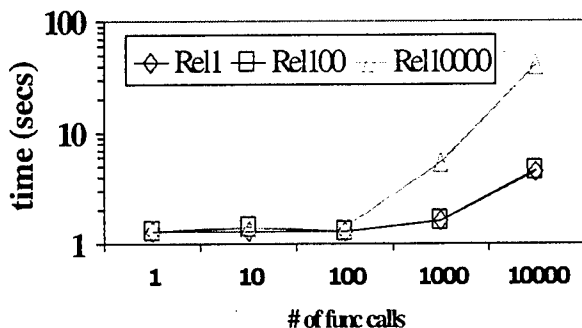


Figure 4: Calibration: Table Access Costs

In Figure 5, the number of UDF invocations is fixed at 10,000. The three UDF designs (C++, IC++ and JNI) are compared as the bytearray size is varied along the X-axis. The UDFs themselves perform no work. Note that 10,000 invocations of a Java UDF incurs only a marginal cost. In fact, for the smaller bytearray sizes, the invocation cost of IC++ is higher than for JNI. This indicates that the cost of using the various JNI interfaces is lower than the context switch cost involved in IC++. For the highest bytearray size, JNI performs marginally worse than IC++, probably because of the effect of mapping large bytearrays to Java.

However, for both JNI and IC++, the extra overhead is insignificant compared to the overall cost of the queries.

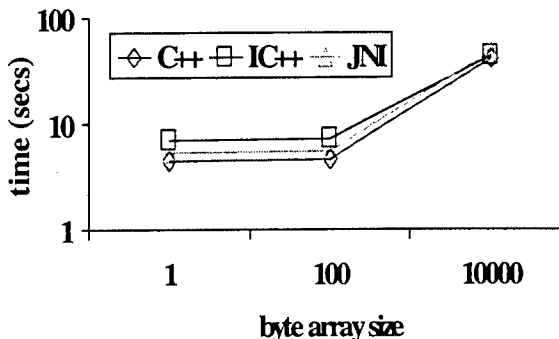


Figure 5: Calibration: Function Invocation Costs

5.3 Effect of Computation

In this set of experiments, our goal is to measure the effect of computationally intensive UDFs. The number of UDF invocations is set at 10,000 and the bytearray size is set at 10,000 bytes. Along the X-axis is the UDF parameter NumDataIndepComps that controls the amount of computation. We expected Java UDFs to perform worse than compiled C++. The results in Figure 6 indicate that JNI performs worse than both C++ options. However, the difference is a constant small invocation cost difference that does not change as the amount of computation changes. This indicates that the Java UDF is run as efficiently as the C++ code (essentially, the result of a good JIT compiler).

The lower graph shows the performance of IC++ and JNI relative to the best possible performance (C++). Even when the number of computations is very high, there is no extra price paid by JNI. In the UDFs tested, the primary computation was integer addition. While other operations may produce slightly different results, the results here lead us to the conclusion that it is perfectly reasonable to expect good performance from computationally intensive UDFs written in Java.

5.4 Effect of Data Access

The next step is to measure performance when there is significant data access involved. Once again, we fix the number of UDF invocations at 10,000 and the bytearray size at 10,000. The data dependent computation, NumDataDepComps, varies along the X axis. The other UDF parameters, NumDataIndepComps and NumCallbacks, are set to 0 to isolate the effect of data access.

Java performs run-time array bounds checking which we expect will slow down the Java UDFs. The results in Figure 7 reveal that this assumption is indeed valid, and there is a significant penalty paid. We did not run JNI with 1000 NumDataDepComps because of the large time involved. The lower graph shows the relative performance of the different UDF designs.

In a sense, this is an unfair comparison, because the Java UDFs are really doing more work by checking array bounds. To establish the cost of doing this extra work, we tested a second version of the C++ UDF that explicitly checks

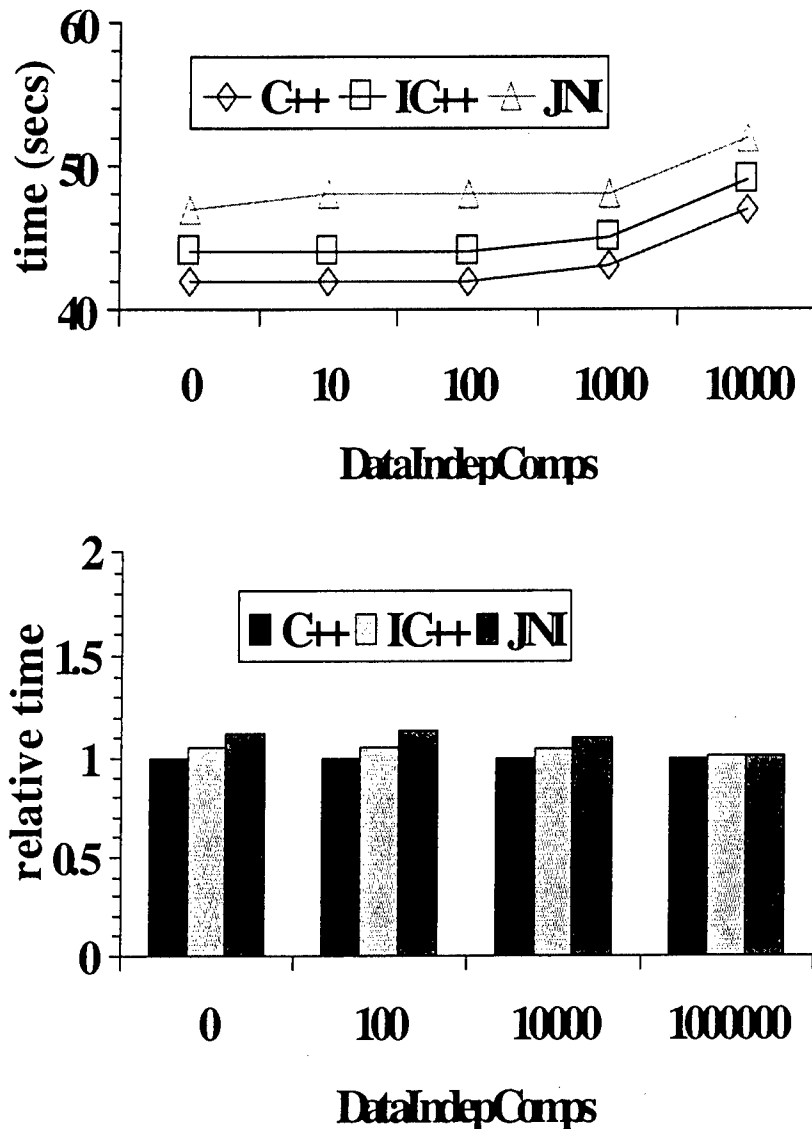


Figure 6: Pure Computation

the bounds of every array access. When compared to this version of a C++ UDF, JNI performs only 20% worse even with large values of NumDataDepComps. It is evident that the extra array bounds check affects C++ in just the same way as Java.

Most UDFs are likely to make no more than a small number of passes over the data accessed. For example, an image compression algorithm might make one pass over the entire image. For a small number of passes over the data, the overall performance of Java UDFs is not very much worse than C++.

5.5 Effect of Callbacks

In our final set of experiments, we examine the effects of callbacks from UDFs to the database server. It is our experience that many non-trivial methods and functions require some database interaction. This is especially likely for functions

that operate on large objects such as images or time-series, but require only small portions of the whole object (a variety of Clip() and Lookup() functions fall in this category). For each callback, the boundary between server and UDF must be crossed.

In Figure 8, the number of callbacks varies along the X-axis, while the functions themselves perform no computation (data dependent or independent). The isolated C++ design performs poorly because it faces the most expensive boundary to cross. For Java UDFs, the overhead imposed by the Java native interface is not as significant. The higher values of NumCallbacks occur rarely; one might imagine a UDF that is passed two large sets as parameters, and computes the "join" of the two using a nested loops strategy. Even for the common case where there are a few callbacks, IC++ is significantly slower than JNI.

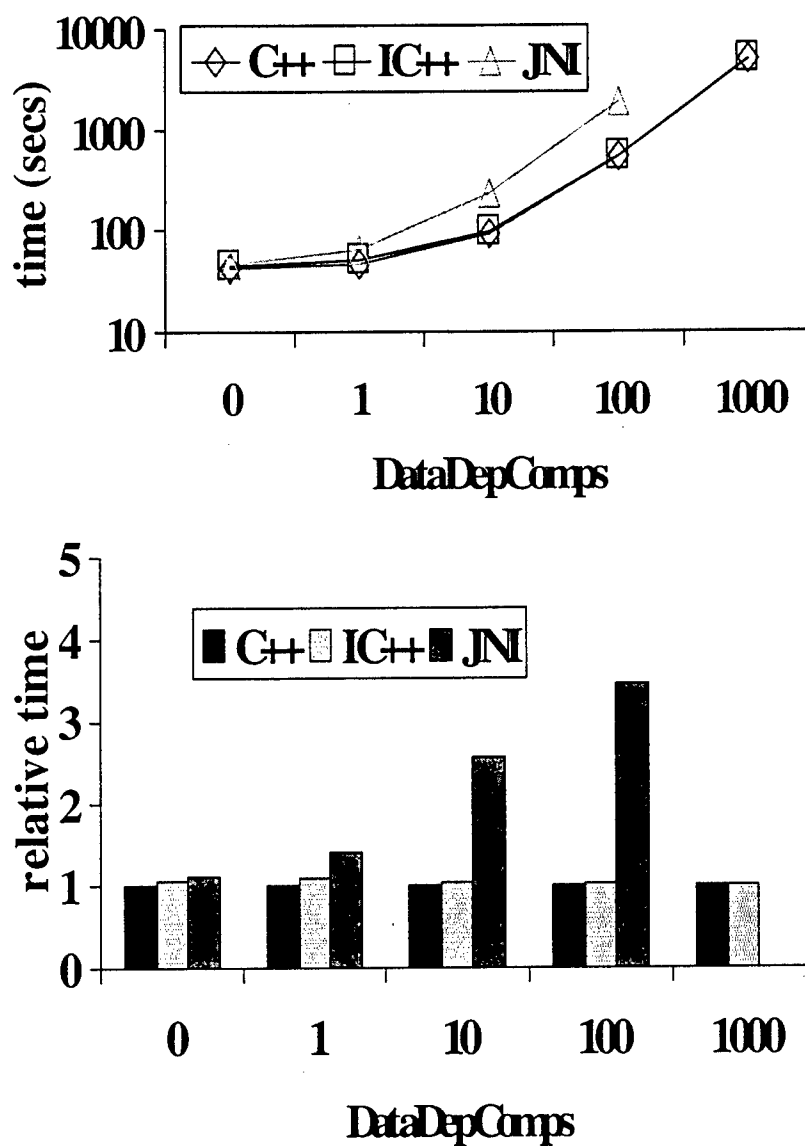


Figure 7: Data Access

5.6 Conclusion from Study

To summarize the conclusions of our performance study:

- Java seems to be an acceptable choice to build UDFs. Its performs poorly relative to C++ only when there is a significant data-dependent computation involved. This is the price paid for the extra work done in guaranteeing memory accesses (array bounds checking).
- Remote execution of C++ functions incurs small overheads due to the cost of crossing process boundaries. While this overhead is minimal if incurred only once per UDF invocation, it may be more significant when incurred multiply due to UDF callbacks.
- There is a tradeoff in the design of a UDF that accesses a large object. Should the UDF ask for the entire object (which is expensive), or should it ask for a handle to the object and then perform callbacks? Our experiments indicate the inherent costs in each approach. In fact, our

experiments can help model the behavior of any UDF by splitting the work of the UDF into different components.

6 Java-based UDF Implementation

Based on our experience with the implementation of Java-based UDFs, we now focus on the following issues generally relevant to the design of Java UDFs:

- **Security and UDF isolation:** Our goal was to extend the database server without allowing buggy or malicious UDFs to crash the server. On the other hand, limited interaction of the UDFs and the server environment is desirable.
- **Resource management:** Even when a restrictive security policy is applied, we face the problem of denial-of-service attacks. The UDF could consume excessive amounts of CPU time, memory or disk space.
- **Integration of a JVM into a database server:** The execution environment of the UDF is not necessarily compatible

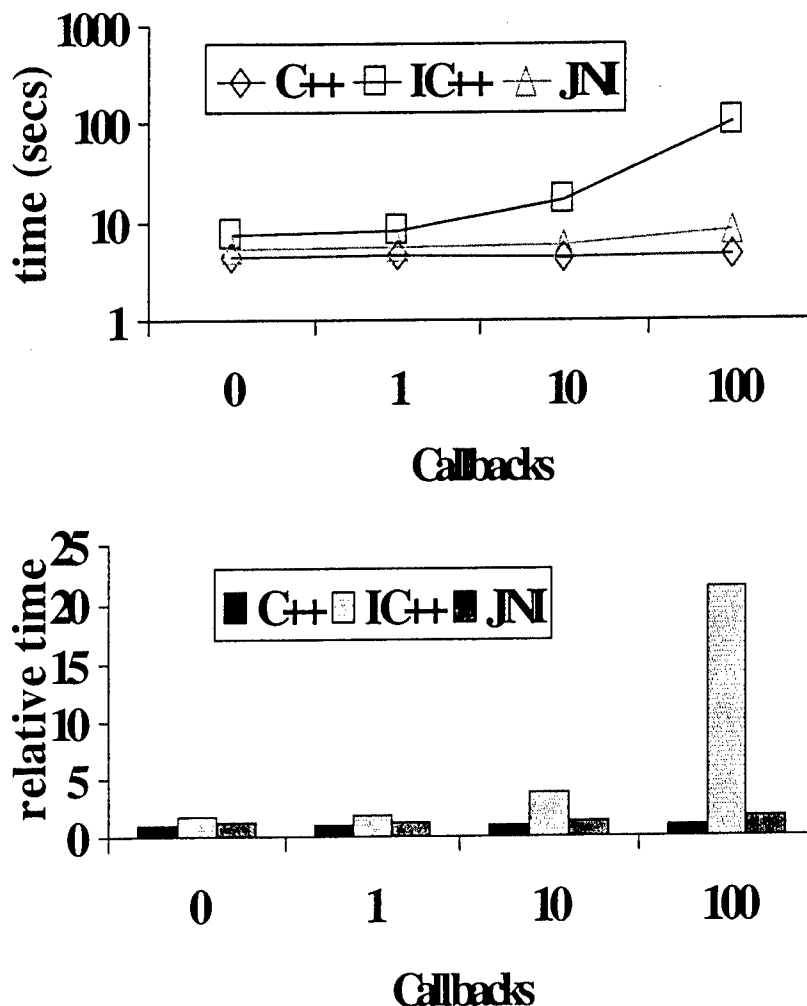


Figure 8: Callbacks

with the operating environment of the database system.

- **Portability and Usability:** The Java UDF design should establish mechanisms to easily prototype and debug UDFs on the client-side and to migrate them transparently between client and server.

6.1 Security and UDF Isolation

Isolating a Java UDF in the database is similar to isolating an applet within a web browser. The four main mechanisms offered by the JVM are:

- **Bytecode Verification:** The JVM uses the bytecode verifier to examine untrusted bytecodes ensuring the proper format of loaded class files and the well typedness of their code.
- **Class Loader:** A class loader is a module of the JVM managing the dynamic loading of class files. New restricted class loaders can be instantiated to control the behavior of all classes that it loads from either a local repository or from the network. A UDF can be loaded with a special class loader that isolates the UDF's namespace from that of other UDFs and prevents interactions between them.
- **Security Manager:** The security manager is invoked by the Java run-time libraries each time an action affecting the execution environment (such as I/O) is attempted. For UDFs, the security manager can be set up to prevent many potentially harmful operations.
- **Thread Groups:** Each UDF is executed within its own thread group, preventing it from affecting the threads executing other UDFs.

Under the assumption that we trust the correctness of the JVM implementation, these mechanisms guarantee that only safe code is loaded from classes that the UDF is allowed to use[Yell96]. These can include other UDF classes, but, for example, not the classes in control of the system resources. The security manager allows access restriction with a finer granularity: a UDF might be allowed by its class loader to load the 'File' class, but only with certain path arguments, as determined by the security manager. The use of thread groups limits the interactions between the threads of different UDFs.

We note that while these mechanisms do provide an increased level of security, they are not foolproof; indeed, there is much ongoing research into further enhancements to Java security. The security mechanisms used in Java are complex and lack formal specification [DFW96]. Their correct-

ness cannot be formally verified without such a specification, and further, their implementations are complex and have been known to exhibit vulnerabilities. Additionally, the three main components: verifier, class loader, and security manager are strongly inter-dependent. If one of them fails, all security restrictions can be circumvented. Another problem of the Java security system is the lack of auditing capabilities. If the security restrictions are violated, there is no mechanism to trace the responsible UDF classes. Although we are aware of these various problems, we believe that the solutions being developed by the large community of Java security researchers will also be applicable in the database context.

6.2 Resource Management

One major issue we have not addressed is resource management. UDFs can currently consume as much CPU time and memory as they desire. Limiting the CPU time would be relatively straightforward for the JVM because each Java thread runs within its own system thread and thus operating system accounting could be used to limit the CPU time allocated to a UDF or the thread priority of a UDF. Memory usage, however, cannot currently be monitored: the JVM does not maintain any information on the memory usage of individual UDFs. The J-Kernel project at Cornell [vEHCC98] is exploring resource management mechanisms in secure language mechanisms, like JVMs. Specifically, the project is developing mechanisms that will instrument Java byte-codes so that the use of resources can be monitored and policed. Such mechanisms will be essential in database systems.

6.3 Threads, Memory, and Integration

It may be non-trivial to integrate a JVM into a database server. In fact, some large commercial database vendors have attempted to use an off-the-shelf JVM, and have encountered difficulties that have led them to roll-their-own JVMs [Nor97]. The primary problem is that database servers tend to build proprietary OS-level mechanisms. For instance, many database servers use their own threads package and memory management mechanisms. Part of the reason for this is historical — given a wide variance in architectures and operating systems on which to deploy their systems, database vendors typically chose to build upon a “virtual operating system” that can be ported to multiple platforms. For example, PREDATOR is built on the SHORE storage manager which uses its own non-preemptive threads package. Systems like Microsoft’s SQLServer which run on limited platforms may not exhibit these problems because they can use platform-specific facilities.

- *Threads and UDFs:* The JVM uses its own threads package, which is often the native threads mechanism of the operating system. The presence of two threads packages within the same program can lead to unexpected and undesirable behavior. The thread priority mechanisms of the database server may not be able to control the threads created by the JVM. If the database server uses non-preemptive threads, there may be no database thread switches while one thread is executing a UDF (this is currently the case in PREDATOR). Further, with more than one threads package manipulating the stack, serious errors could result.
- *Memory Management:* Many commercial database servers implement proprietary memory managers. For example, a common technique is to allocate a pool of memory for a

query, perform all allocations in that pool, and then reclaim the entire pool at the end of the query (effectively performing a coarsely-grained garbage collection). On the other hand, the JVM manages its own memory, performing garbage collection of Java objects. The presence of two garbage collectors running at the same time presents further integration problems. We do not experience this problem in PREDATOR, because there is no special memory management technique used in our implementation of the database server.

6.4 Portability and Usability

We have developed a library of Java classes that helps developers build Java applets that can act as database clients. The details of this library are presented in [PS97]. It is roughly analogous to a JDBC driver (in fact, we have built a JDBC driver on top of it) with extensions for handling complex data types. The user sits at a client machine and accesses the PREDATOR database server through a standard web browser. The browser downloads the client applet from a web server, and the applet opens a connection to the database server.

Our goal is to be able to allow users to easily define new Java UDFs, test them at the client, and migrate them to the server. This mechanism is currently being implemented. The basic requirement is that there should be similar interfaces at the client and at the server for UDF development and use. Every data type used by the database server is mirrored by a corresponding ADT class implemented in Java. These ADT classes are available both to the client and the server⁶. Each ADT class can read an attribute value of its type from an input stream and construct a Java object representing it. Likewise, the ADT class can write an object back to an output stream. Thus the arguments of an UDF can be constructed from a stream of parameter values, and the result can be written to an output stream. At both client and server, Java UDFs are invoked using the identical protocol; input parameters are presented as streams, and the output parameter is expected as a stream. This allows UDF code to be run without change at either site.

6.5 Experience

We have described a relatively well-understood usage of the Java security mechanisms that is essentially identical to running multiple applets within a web browser. Our implementation has developed a common internal interface that can be supported at both client and server for the development of portable Java UDFs.

There are interesting design issues in integrating a JVM into the database server, especially in dealing with threads and memory allocation. Based on our experiments, we observe that the cost of isolated-process UDFs is reasonable unless there are a large number of callbacks. Consequently, it may be practical to consider running the JVM in a separate process from the database server. The attraction of this solution lies in its simplicity and the ability to use off-the-shelf JVMs.

7 Conclusion

This paper presented an initial study of the issues involved in extending database systems using Java. The conclusion is

⁶The client can download Java classes from the server-site.

that an extensible database system can support secure and portable extensibility using Java, without unduly sacrificing performance. We are currently developing the infrastructure to move Java UDFs between clients to servers, and optimization mechanisms to choose between the various execution options. We also intend to build applications that will test this infrastructure in the real world.

References

- [Ber95] Brian Bershad. Extensibility, safety and performance in the spin operating system. In *Fifteenth Symposium on Operating Systems Principle*, 1995.
- [Car97] Luca Cardelli. Type Systems The Computer Science and Engineering Handbook 1997: 2208-2236
- [CDF⁺94] M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O. Tsatalos, S. White, and M.J. Zwilling. Shoring up persistent objects. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data*, Minneapolis, MN, pages 526-541, 1994.
- [Cim97] Cimarron Taylor. Java-Relational Database Management Systems. <http://www.jbdev.com/>, 1997.
- [DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond 1996 IEEE Symposium on Security and Privacy, Oakland, CA
- [Fra96] M.J. Franklin. Client Data Caching. Kluwer Academic Press, Boston, 1996.
- [FJK96] M.J. Franklin, B.T. Jonsson and D. Kossman. Performance Tradeoffs for Client-Server Query Processing. In *Proceedings of ACM SIGMOD '96 International Conference on Management of Data* 1996.
- [HCL⁺90] L. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.
- [Hel95] Joseph M. Hellerstein. *Optimization and Execution Techniques for Queries With Expensive Methods*. PhD thesis, University of Wisconsin, August 1995.
- [Jhi88] Anant Jhingran. A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures. In *Proceedings of the Fourteenth International Conference on Very Large Databases*, pages 88-99, 1988.
- [JNI] JNI - Java Native Interface <http://www.javasoft.com/products/jdk/1.1/docs/guide/jni/index.html>
- [MWCG98] Greg Morrisett, David Walker, Karl Cray, and Neal Glew. From System F to Typed Assembly Language To appear in the 1998 Symposium on Principles of Programming Languages
- [NCW98] Just In Time for Java vs. C++ <http://www.ncworldmag.com/ncworld/ncw-01-1998/ncw-01-rmi.html>
- [Nec97] George C. Necula. Proof-Carrying Code Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France, 1997.
- [Nor97] Anil Nori. Personal Communication, 1997.
- [RNI] Microsoft Raw Native Interface <http://premium.microsoft.com/msdn/library/sdkdoc/java/htm/rni.introduction.htm>
- [PS97] Mark Paskin and Praveen Seshadri. Building an OR-DBMS over the WWW: Design and Implementation Issues. Submitted to SIGMOD 98, 1997.
- [SLR97] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Case for Enhanced Abstract Data Types. In *Proceedings of the Twenty Third International Conference on Very Large Databases (VLDB)*, Athens, Greece, August 1997.
- [SRG83] M. Stonebraker, B. Rubenstein, and A. Guttman. Application of Abstract Data Types and Abstract Indices to CAD Data Bases. In *Proceedings of the Engineering Applications Stream of Database Week*, San Jose, CA, May 1983.
- [SRH90] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125-142, March 1990.
- [SS75] Jerome H. Saltzer, Michael D. Schroeder. The Protection of Information in Computer Systems <http://web.mit.edu/Saltzer/www/publications/protection>
- [Sto86] Michael Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *Proceedings of the Second IEEE Conference on Data Engineering*, pages 262-269, 1986.
- [vEHCCCH98] Thorsten von Eicken, Chris Hawblitzel, Chi-Chao Chang, Gzegorz Czajkowski, and Deyu Hu. Implementing Multiple Protection Domains in Java to appear, Usenix 1998 Annual Technical Conference, June 15-19, New Orleans, Louisiana.
- [WLAG93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Fourteenth Symposium on Operating Systems Principle*, 1993.
- [Yell96] Frank Yellin. Low Level Security in Java <http://www.javasoft.com:81/sfaq/verifier.html>

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 server-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).

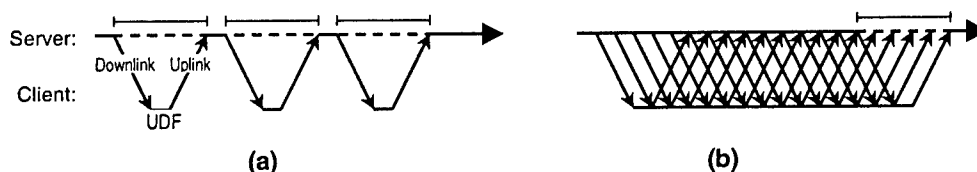


Figure 2: Timeline of Nonconcurrent and Concurrent Execution

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.

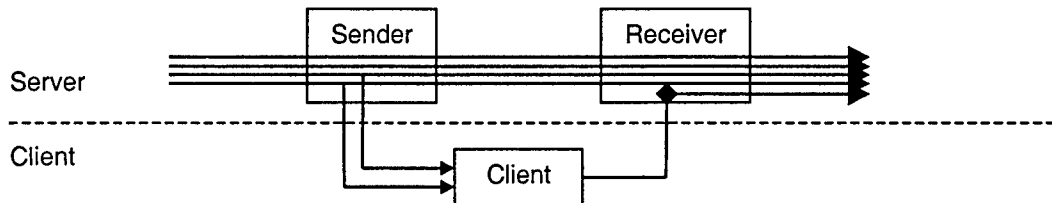


Figure 3: Semi-Join Architecture

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

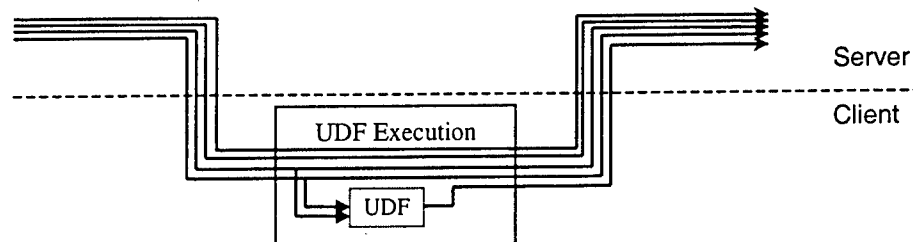


Figure 4: Client-Site Join Architecture

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.

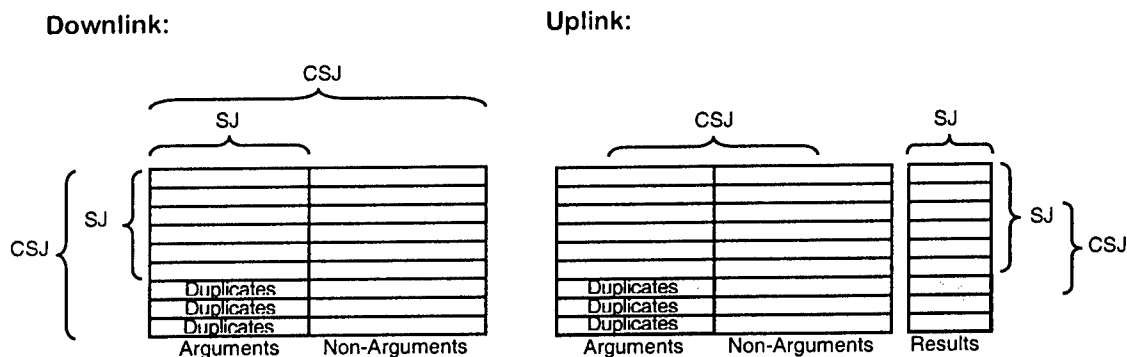


Figure 5: Tradeoffs between Client-Site and Semi-Join

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 : Asymmetry 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:

$$D * (A * I) \quad (\text{semi-join, bytes transferred on downlink, per tuple average})$$

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

$$N * D * R \quad (\text{semi-join, bytes transferred on uplink, per tuple average})$$

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

$$I \quad (\text{client-site join, bytes transferred on downlink, per tuple average})$$

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

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

The bandwidth cost incurred at the bottleneck link is the maximum of the costs incurred at each link. N , the network asymmetry 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.

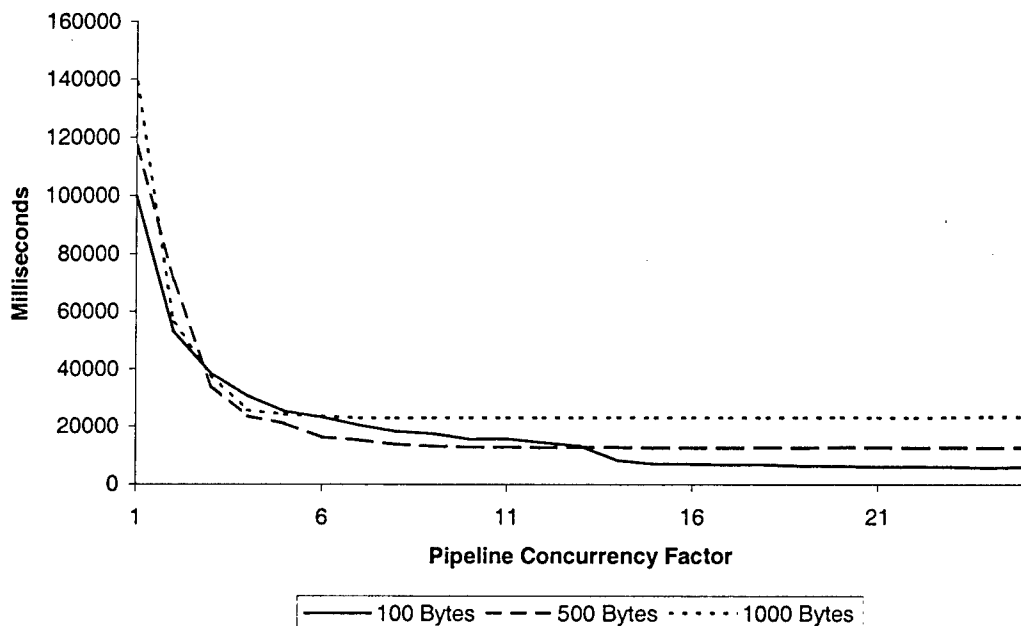


Figure 6: Effect of Concurrency

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: $P \cdot (I + R) = I \cdot (1 - A) + R$, 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.

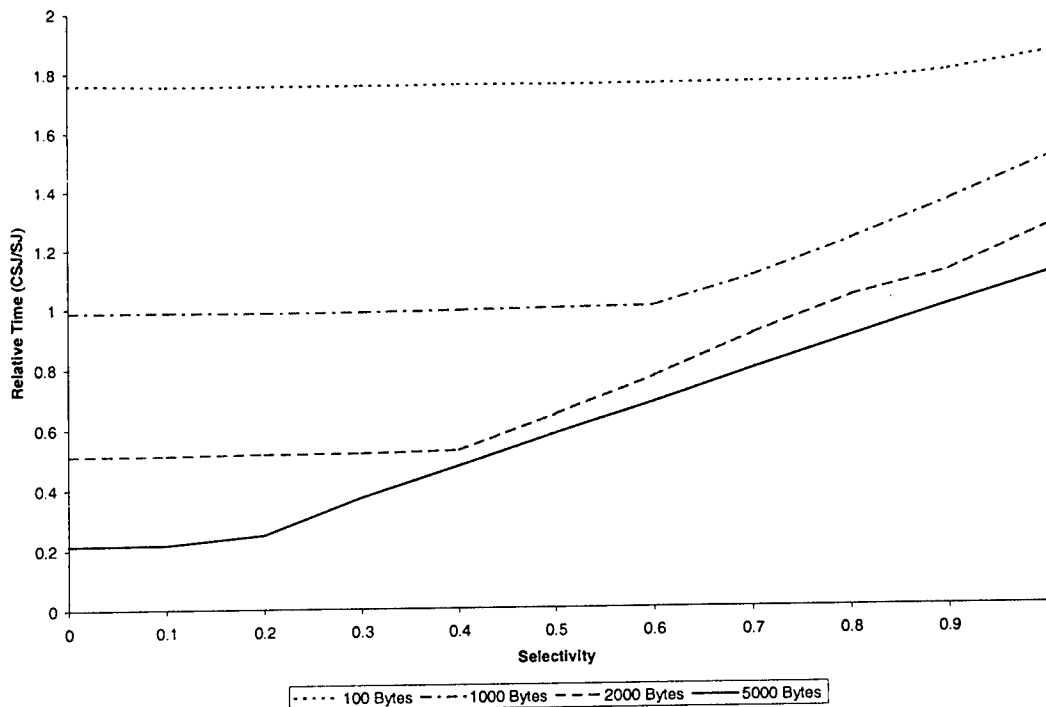


Figure 8: Client-Site Join versus Semi-Join on a Symmetric Network

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.

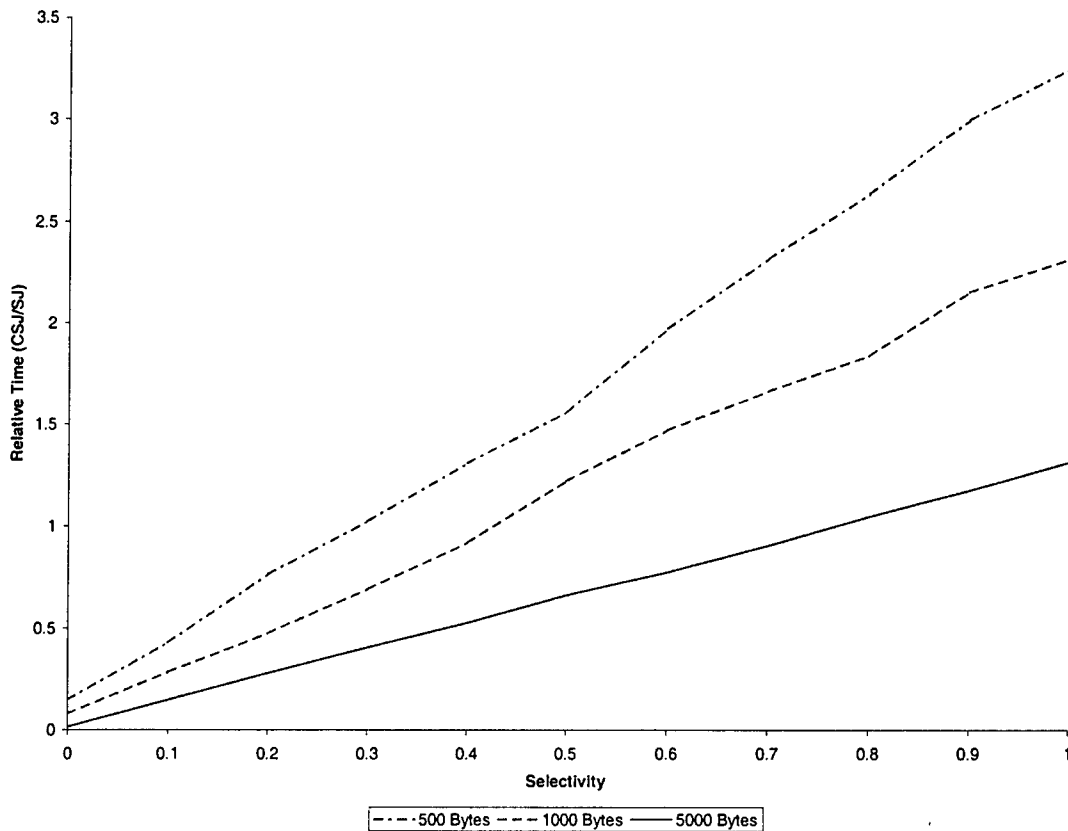


Figure 9: Client-Site Join versus Semi-Join on Asymmetric Network

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: $1/(N*P*(R+I)) = 0.0083$ 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.

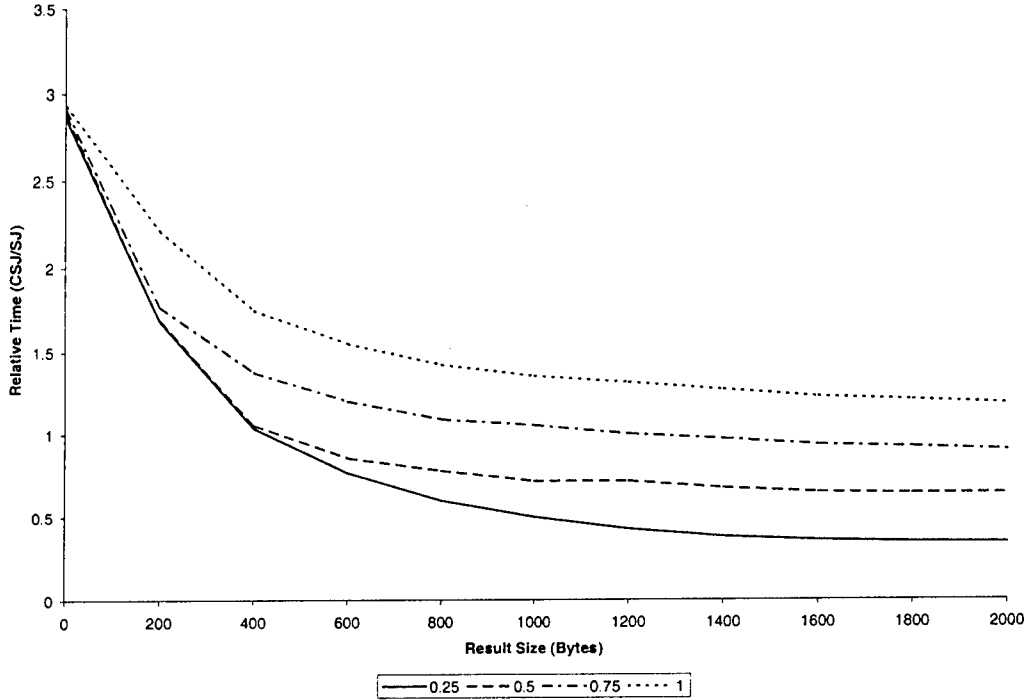


Figure 10 : Influence of the Result Size

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 asymptotically 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:

- 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.
- 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:

- The per-tuple execution cost of an operation is known a priori, *independent of its position* in the query plan.
- 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.

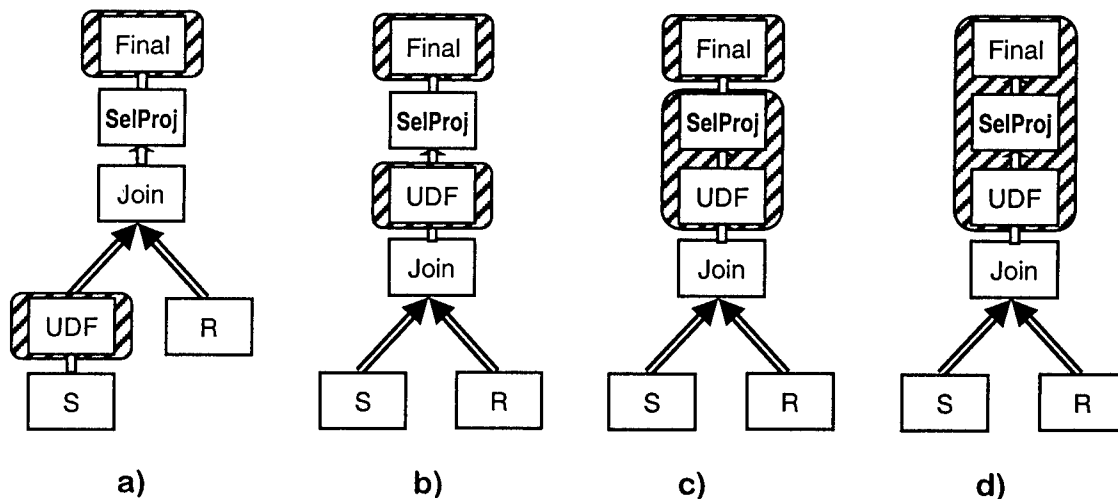


Figure 12 : Possible Plans for the Query in Figure 11

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.

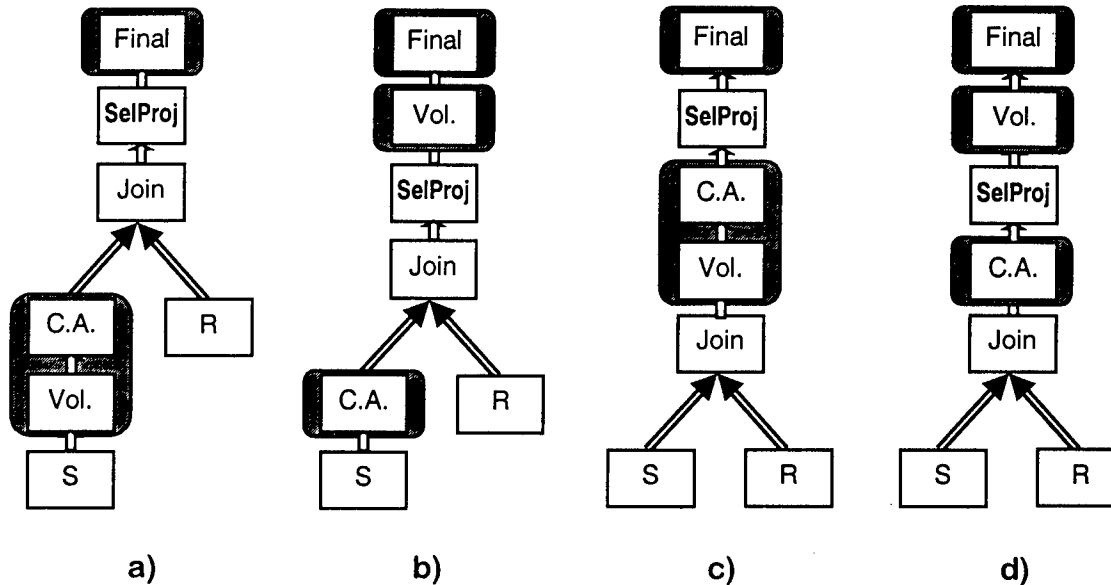


Figure 13 : Possible Plans for the Query in Figure 11 with Additional UDF

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 relations¹. 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.

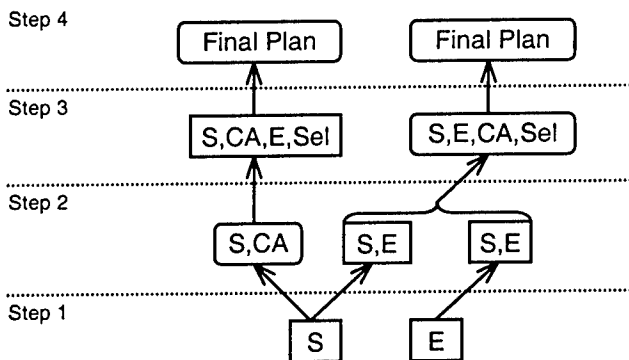


Figure 14: Client-Site Join Optimization of the Query in Figure 11

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  $J \subseteq \{J_1, \dots, J_m\}$  s.t.  $|J|=i$  DO
  { BestPlan := dummy plan of infinite cost
    FOR ALL  $j, J'$  s.t.:  $|J'| = i$  AND
                       $\{j\} \cup J' = J$  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, $S \otimes E$, 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.

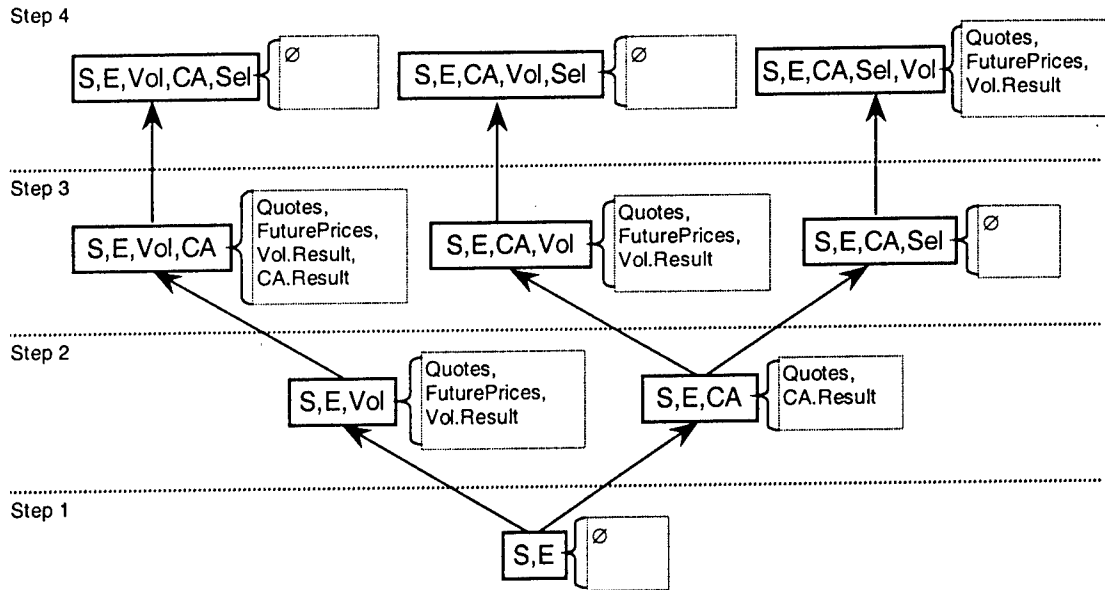


Figure 16 : Semi-Join Optimization for the Query in Figure 11

5.2.4 Features of the Optimization 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 and 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.Czajkowski, 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.Zaniolo. 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.

***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

*The advancement and application of Information Systems Science
and Technology to meet Air Force unique requirements for
Information Dominance and its transition to aerospace systems to
meet Air Force needs.*