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 runtime 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 level[1]. 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 minimum[2].

[1] In 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]).

[2] The 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 generalizable 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 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 expensive[5]. This is usually reflected in the efficiency of the system. Note

[3] A 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.

[4] Most 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].

[5] In 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 than 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 $\langle i, j \rangle$ 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 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.

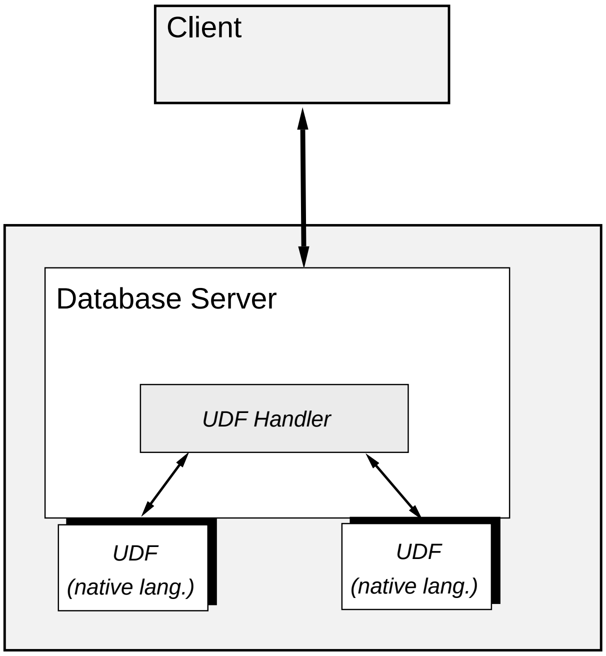

Figure 1: Design 1: Integrated Native UDFs

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.

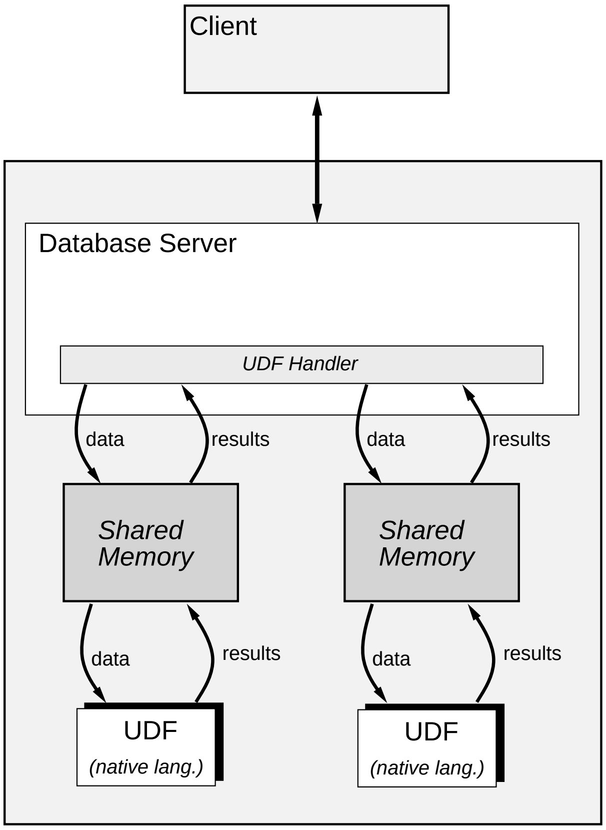

Figure 2: Design 2: Isolated Native UDFs

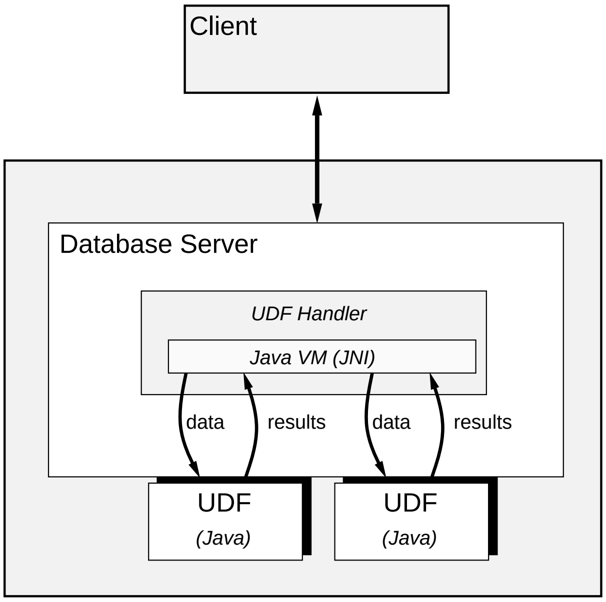

Figure 3: Design 3: Java UDFs

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 byte array 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.

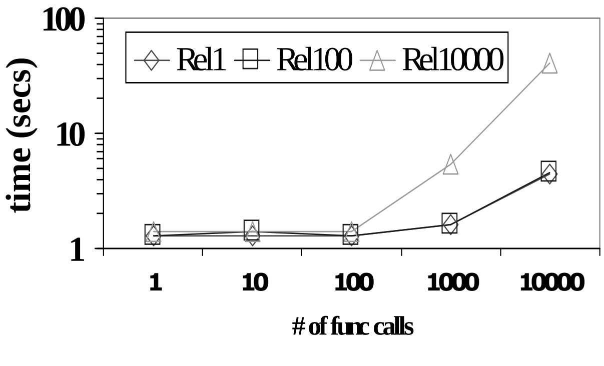

Figure 4: Calibration: Table Access Costs

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.

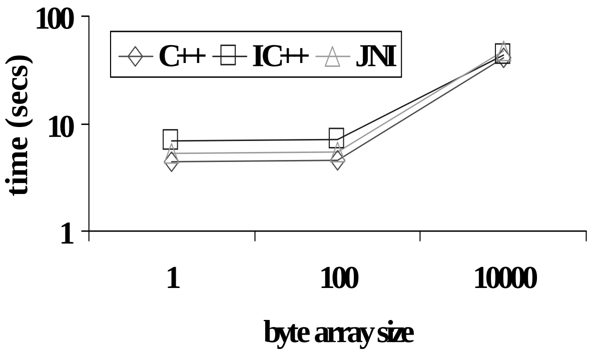

Figure 5: Calibration: Function Invocation Costs

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

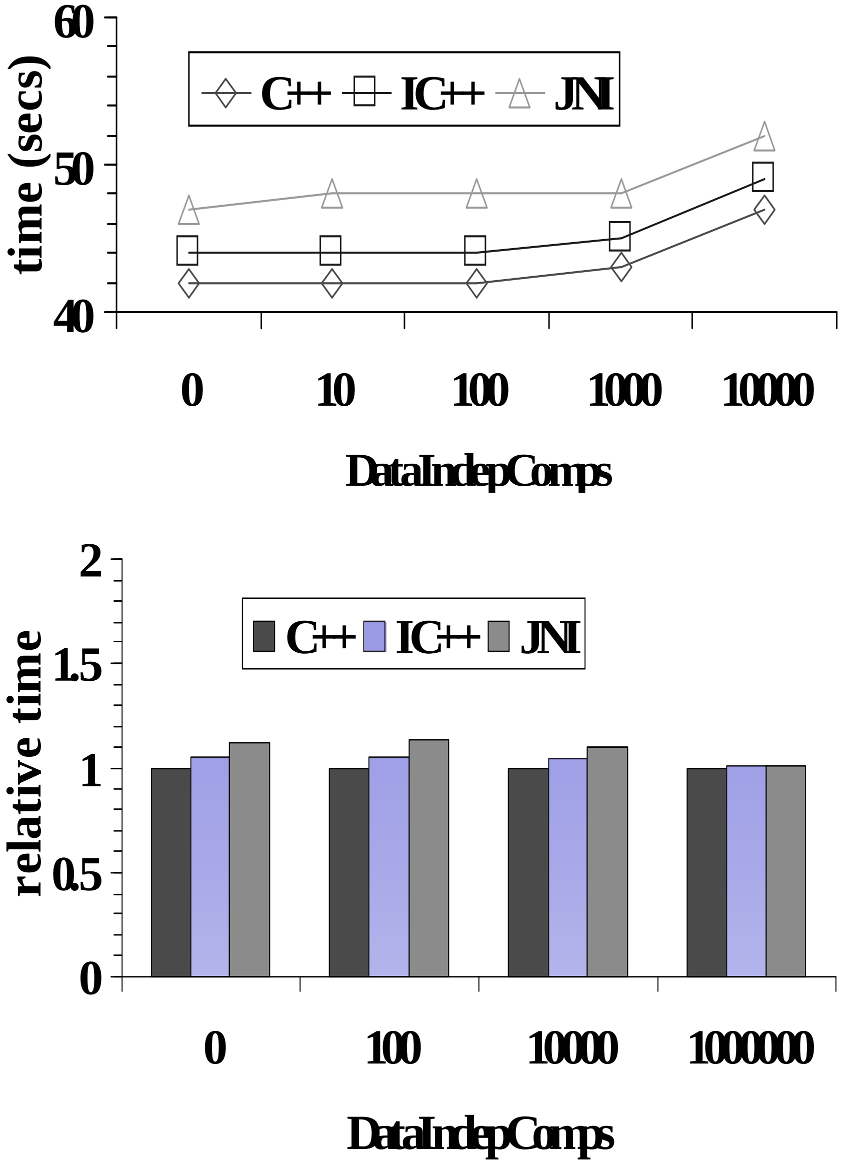

Figure 6: Pure Computation

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.

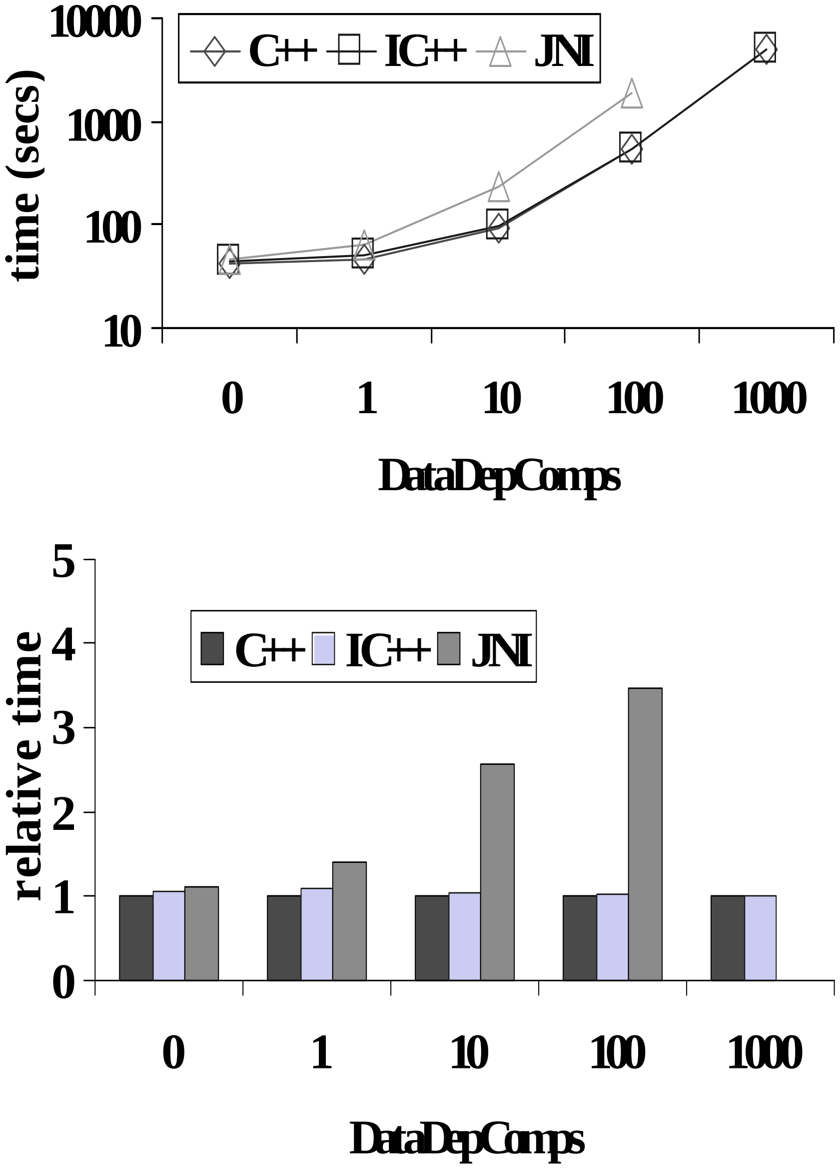

Figure 7: Data Access

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

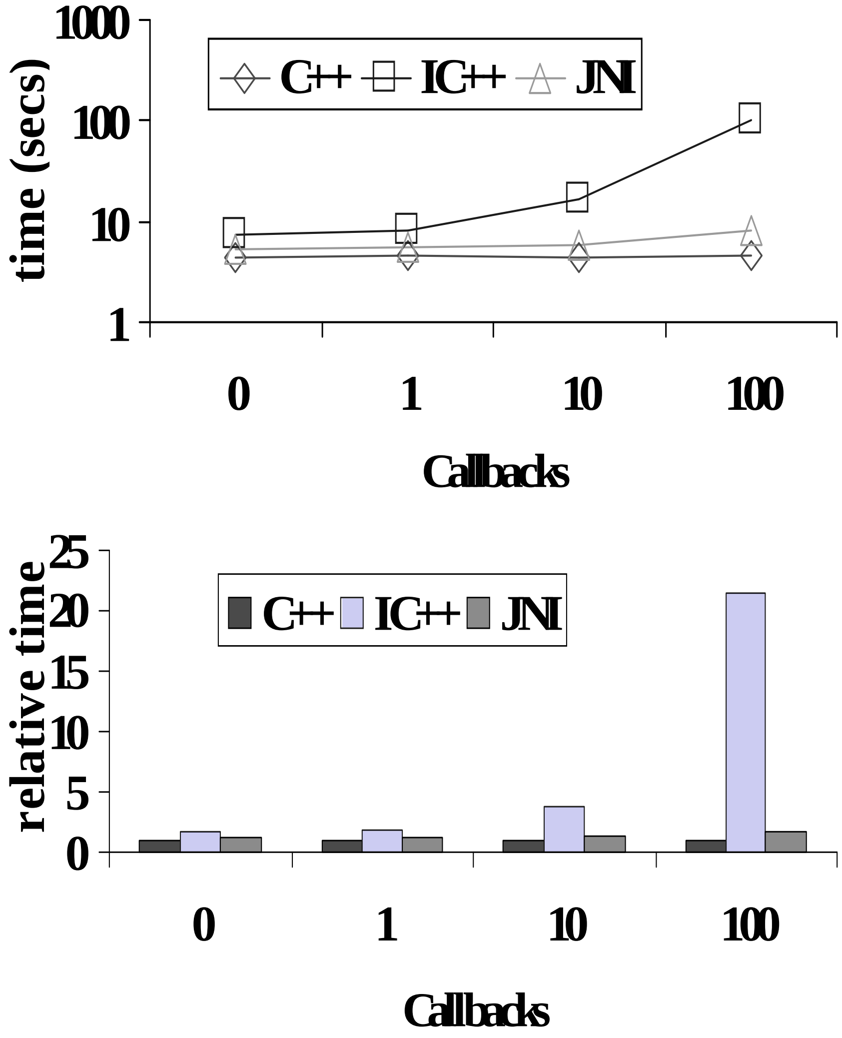

Figure 8: Callbacks

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 correctness 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 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 straight-forward 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 [vEHCCH98] 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 lead 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 server[6]. 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

[6] The 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 midflight: 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 Crary, 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 Lnaguages (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.

[vEHCCH98] 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