Resource Control for Database Extensions

Grzegorz Czajkowski, Tobias Mayr,
Praveen Seshadri, Thorsten von Eicken

Cornell University

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

Abstract

While object-relational database servers can be extended with user-defined functions (UDFs), the security of the server may be compromised by these extensions. The use of Java to implement the UDFs is promising because it addresses some security concerns. However, it still permits interference between different users through the uncontrolled consumption of resources. In this paper, we explore the use of a Java resource management mechanism (JRes) to monitor resource consumption and enforce usage constraints. JRes enhances the security of the server in the presence of extensions allowing for (i) detection and neutralization of denial-of-service attacks aimed at resource monopolization, (ii) monitoring resource consumption which enables precise billing of users relying on UDFs, and (iii) obtaining feedback that can be used for adaptive query optimization.

The feedback can be utilized either by the UDFs themselves or by the database system to dynamically modify the query execution plan. Both models have been prototyped in the Cornell Predator database system. We describe the implementation techniques, and present experiments that demonstrate the effects of the adaptive behavior facilitated by JRes. We conclude that, minimally, a database system supporting extensions should have a built-in resource monitoring and controlling mechanism. Moreover, in order to fully exploit information provided by the resource control mechanisms, both the query optimizer and the UDFs themselves should have access to this information.

1 Introduction

There has been much recent interest in using Java to implement database extensions. The SQL-J proposal [SQLJ] describes efforts by database vendors to support user-defined functions (UDFs) written in Java. Java UDFs are considered relevant in environments like internets and intranets, where large numbers of users extend a database server backend. In earlier work [GMS+98], we explored some of the security, portability, and efficiency issues that arise with Java UDFs. The main observation was that although Java UDFs are efficient, they do not solve all the security problems that arise when a server accepts untrusted extensions. Specifically, short of creating a process per UDF, there is no suitable mechanism to prevent one UDF from allocating large amounts of memory or using a large portion of the CPU time. This allows a malicious or buggy UDF to effectively deny service to all the other users of the database system. Another problem directly and negatively affecting deployment of Java-UDF-enabled database systems is the lack of an infrastructure for monitoring resource consumption and billing users for resources consumed by their UDFs.

In this paper, we describe the application of a Java resource accounting interface, JRes [CvE98], to address this issue. JRes has been incorporated into the Cornell Predator database system [Sesh98a] as part of the Jaguar project, and we base our observations on the resulting prototype. To the best of our knowledge the resulting system is the first database where extensibility based on a safe language is augmented with an ability to monitor usage of computational resources (we note that similar concurrent efforts are being made by vendors of several relational systems). In particular, our work further limits the amount of trust that the database server must have with respect to the behavior of extensions. Due to basing extensibility mechanisms on a safe language, our previous work ensured that the server is protected from extensions and the extensions are protected from one another, while still enjoying the performance benefits of executing all participating entities in a single address space. The current research demonstrates how a class of UDFs that may execute in a database server without affecting the execution of the server or other extensions can be enlarged to contain UDFs with unknown and potentially malicious or unbalanced resource requirements.

We question two implicit assumptions underlying previous work on optimizing queries with user defined functions: (i) that the costs of invoking a UDF will stay the same over the execution of the entire query, and (ii) that it is possible to provide realistic estimates on the costs of UDFs. A query executing on large tables and using costly UDFs will execute long enough that considerable fluctuations in resource availability will be observed while the query is running. As a consequence, the relative weights associated with different types of resources will change. Expensive UDFs also often execute complex code, making it difficult to accurately predict their cost. Finally, database cost estimates are typically not absolute; rather they simply need to be accurate *relative* to each other on some cost scale used by the database system developers (and usually not quantified in terms of real time). The user defining a new UDF has no way to position it on this internal cost scale.

Our work addresses some of these concerns. JRes provides feedback for adaptive query optimization by monitoring the use of resources by each UDF. Depending on the adopted system design, either each UDF requests information about resource consumption and adapts its runtime behavior accordingly, or the database server uses the feedback from the resource monitor to adapt the query's execution. Each model is desirable in certain situations, leading to the conclusion that a database system needs to support both models of resource control feedback.

The rest of the paper is structured as follows. An example-based motivation of our work is contained in the next section. This is followed by a description of selected details on Jaguar and JRes - systems used for experimentation in this study. Section 4 outlines a design space of applicability of dynamic resource controlling mechanisms for user defined functions. Section 5 shows how resource-limiting policies can be defined for Java UDFs. Taking advantage of resource availability feedback is discussed in Section 6 and is followed by a section quantifying performance gains obtained due to using the feedback information.

2 Motivation

In order to justify the need for management of computational resources in extensible database servers let us consider the following example. An amateur investor is planning future stock acquisitions and has purchased access to a database server that can be extended with used defined functions coded in Java. Among other data, users of the server can access the table `Companies`, which lists firms whose stock is currently sold and bought on the New York Stock Exchange. The table has two columns of interest for the investor: `Name` (the name of a company) and `ClosingPrices`, which is an array of floating point numbers corresponding to company's share prices. The array contains an entry for every day since the company entered the stock market.

The investor wants to find companies that meet all the following requirements: (i) the company is on the market for at least forty days, (ii) the price of a share forty days ago is smaller than the price today, and (iii) on any given day during the last thirty nine days the price has not changed by more than 2% from the previous day. This can be expressed as the following SQL query:

```
SELECT C.Name
FROM Companies C
WHERE LooksPromising(C.ClosingPrices)
```

where `LooksPromising` is a method of an investor-supplied Java class `StockAnalysis`. Such a class can be written by the investor, generated by a tool, or purchased from a software development house. An implementation is shown below:

```
public class StockAnalysis {
  private static final int NUMBER_OF_DAYS = 40;
  private static final int VAR = 0.02;

  public static boolean LooksPromising(double[] ts) {
    int size = ts.length;
if (size < NUMBER_OF_DAYS)
      return false;
if (ts[size - NUMBER_OF_DAYS] >= ts[size - 1])
      return false;
    for (int i = 1; i < NUMBER_OF_DAYS; i++) {
      double price = ts[size - i + 1];
      double prevPrice = ts[size - i];
      double v = (price - prevPrice)/prevPrice;
      if (Math.abs(v) >= VAR)
        return false;
    }
    return true;
  }
}
```

This kind of database extensibility has many benefits. Many complex filters can be coded much easier and more efficiently when using a programming language instead of SQL. UDFs can be used to integrate user-specific algorithms and external data sources. By limiting the extensions use of the network and the file system, and through the use of Java protection mechanisms, the server can ensure that its data is not corrupted or compromised. Cryptography-based protocols like Secure Socket Layer [SSL97] can be used to guarantee secure uploading of UDFs to the server. This means that if investors trust the server they can be assured that nobody else will see the code of their UDFs, which can be a concern when substantial effort was expended towards creating them.

However, at the current state of the art of extensible database technologies [GMS+98] several important issues are still not addressed. These problems are discussed in the subsections below. They include dealing with denial-of-service attacks, accounting for resources consumed by a user's particular UDFs, and supporting system scalability. For extensible databases where the UDFs are executed in the controlled environment of a safe language , these problems, to a large extent, boil down to the ability to monitor computational resources such as main memory, CPU usage, and network resources.

2.1 Denial-of-Service Attacks

The code of `LooksPromising` is not necessarily well behaved: People make mistakes - for instance, a programmer could forget to increment `i` in the `for` loop which can lead to a non-terminating execution of `LooksPromising` for some inputs. In addition to making mistakes, some code is developed with malicious purposes in mind. One could omit incrementing the loop counter on purpose, or, for instance, insert into `LooksPromising` code to allocate an infinite list so that all available main memory is monopolized by a single instance of the UDF. Regardless of whether such programs are created on purpose or unintentionally, they are equally dangerous in that they can monopolize important computational resources. Except for a few trivial cases, it is virtually impossible to decide by means of static code analysis if a Java UDF will use more resources than a particular limit. Dynamic mechanisms that constrain resource usage are needed to prevent denial-of-service attacks. Traditional operating systems use hardware protection and coarse-grained process structure to enforce resource limits. Extensible object-relational database environments, in many ways subsuming the role of an operating system, need to provide the same functionality.

2.2 Accounting for Consumed Resources

Many database servers use accounting mechanisms to charge customers for service. The same will likely happen to extensible database servers based on Java. An immediate problem is that no mechanisms exist that enable accounting for resources consumed by Java UDFs. For instance, CPU time and heap memory used by an invocation of `LooksPromising` are unknown, since Java provides no support for gauging their usage.

Ideally, one should be able to run a UDF and obtain a list of all the resources consumed by it. For instance, in the case of `LooksPromising`, the maximum amount of memory and CPU time used during the invocation and should be available. These can be used for profiling the code and for charging investors for resources consumed during the execution of their queries. Thus, obtaining resource consumption traces from a running UDF is valuable for query optimizers.

2.3 Scheduling and Scalability

Another problem with deploying extensible database servers based on safe languages such as Java is the difficulty of managing large numbers of extensions. Since virtually no information about resource consumption can be obtained, the system does not know what UDFs are particularly resource-hungry and which resources will be stressed when a large number of copies of a particular UDFs are executing simultaneously. This potentially leads to unbalanced resource consumption patterns. For instance, let us imagine several thousand copies of `LooksPromising` running at the same time. If the UDFs do not adapt their behavior, they face the prospect of slow execution, of deadlock, of being stopped temporarily, or even of being killed by the system, depending on the local policy. This is likely to result in wasted resources since queries and/or UDFs will be aborted halfway through. Providing dynamic information about resources available to UDFs allows database systems to implement admission control policies that minimize the number of aborted UDFs. The UDFs themselves may be coded in a smart way to adapt to changing resource demand and supply. However, in order to be able to perform such coding, a interface is necessary that allows the UDFs to learn about the loads during their execution.

2.4 An Approach to Manage Resources in Extensible Database Servers

The objective of this work is to provide mechanisms for selected components of resource management in an extensible database where UDFs are executed in a single running copy of the Java Virtual Machine . This includes (i) accounting for resource (CPU time, heap memory, network) usage on a per-UDF basis, (ii) setting limits on resources available to particular UDFs, and (iii) providing the ability to define a specific action to be taken when a resource limit is exceeded. To this end we have extended Java and consequently the JVM serving as an extensibility mechanism with a resource accounting interface, called JRes. The extension does not require any changes to the underlying JVM and relies on dynamic bytecode rewriting and a small native component, coded in the C language. As will be demonstrated later in the paper, most of the problems discussed in this section are addressed in our prototype.

3 Selected Details on Jaguar and JRes Environments

This section contains a brief description of features of Jaguar and JRes relevant for the work described in this paper. Both systems have been described in detail elsewhere [GMS+98, CvE98].

3.1 Jaguar

The Jaguar project extends the Cornell Predator object-relational database system [Sesh98a] with portable query execution. The goals of the project are two-fold: (a) to migrate client-side query processing into the database server for reasons of efficiency, (b) to migrate server-side query processing to any component of the channel between the server and the ultimate end-user. In short, the project aims to eliminate the artificial server-client boundaries with respect to query execution. 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 and mutually independent applications interact with the database server. In this context, portable query execution can translate into greater options for efficient evaluation and consequently reduced user response times.

The Predator database server is written in C++, and permits new extensions (new data types and UDFs, also written in C++). To explore goal (a) of the Jaguar project, the database server has been enhanced with the ability to define UDFs with Java. This provides clients with a portable mechanism with which to specify client-side operations and migrate them to the server. Java seems to be a good choice as a portable language for UDFs, because Java byte code can be run with security restrictions within the Java Virtual Machine.

In the current implementation, Java functions are invoked from within the server using either Sun's Java Native Interface [JNI] or Microsoft's Raw Native Interface [RNI]. The first step is to initialize the Java Virtual Machine (JVM) as a C++ object. Any classes that need to be used are loaded into the JVM using a custom interface. When methods of the classes need to be executed, they are invoked through JNI or RNI, depending which vendor's JVM is currently used. Parameters that need to be passed to Java UDFs must be first mapped to Java objects.

The creation of the 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.

3.2 JRes

Through JRes, the trusted core of Java-based extensible databases can (i) be informed of all new thread creations, (ii) state an upper limit on memory used by all live objects allocated by a particular thread or thread group, (iii) limit how many bytes of data a thread can send and receive, (iv) limit how much CPU time a thread can consume, and (v) register *overuse callbacks*, that is, actions to be executed whenever any of the limits is exceeded. Both trusted core and untrusted extensions can learn about resource limits and resource usage.

The JRes interface is presented in Figure 1. It consists of two Java interfaces, one exception, and the class `ResourceManager.` Except for `initialize()`, all presented methods of `ResourceManager` deal with threads and each of them has a counterpart method (not shown) managing resources on a per thread group basis. The `ResourceManager` class defines constants identifying resources (not shown) and exports several methods. The first one, `initialize(cookie)`, handles an authenticating object (a *cookie*; it can be any object) to the resource accounting subsystem and initializes it. The purpose of the cookie is to ensure that only the database server itself has privileged access to the resource management subsystem. This prevents UDFs from interfering with the resource management policies of a given system.

The method `setThreadRegistrationCallback(cookie, tCallback)`, hands an object implementing the `ThreadRegistrationCallback` interface to the resource management subsystem. Consequently, whenever a new Java thread `t` is created, `tCallback.threadRegistrationNotification(t)` will be invoked. The callback is meant to work in conjunction with the method `setLimit(cookie, resType, t, limit, period, oCallback`), which can be invoked each time a new thread creation is detected and has the effect of setting a limit of `limit` units of a resource `resType` for a thread `t`. The `period` argument is relevant for setting limits on CPU time and network resources and determines in conjunction with `limit` the "bandwidth" of a resource available to the UDF. If `period` is set to `0`, the limit is absolute, not periodic. Network and memory usage and limits are expressed in bytes, while CPU time is registered in milliseconds. Resource limits are cleared by invoking `clearLimit(cookie, resType, t).`

Whenever a limit is exceeded by a thread `t`, the method `resourceUseExceeded(resType, t, resValue)` will be invoked on the registered object `oCallback`. The parameter `resValue` passed to the callback provides information about current resource consumption. A particular resource management policy may choose to throw the exception `ResourceOveruseException` as an action taken against UDFs using too many resources. Finally, the method `getResourceUsage(resType, t)` queries the resource management

```
public interface ThreadRegistrationCallback {
  public void threadRegistrationNotification(Thread t);
}

public interface OveruseCallback {
  public void resourceUseExceeded(int resType, Thread t, int resValue);
}

public class ResourceOverusedException extends RuntimeException {
  ResourceOverusedException(int resType, long limitingValue, long usedValue);
}

public final class ResourceManager {
  public static boolean initialize(Object cookie);

  public static boolean setThreadRegistrationCallback(Object cookie,
                                   ThreadRegistationCallback tCallback);

  public static boolean setLimit(Object cookie, int resType,
         Thread t, long limit, long period, OveruseCallback oCallback);
  public static boolean clearLimit(Object cookie, int resType, Thread t);

  public static long getResourceUsage(int resType, Thread t);
  public static long getResourceLimit(int resType, Thread t);
}
```

Figure 1. The JRes interface.

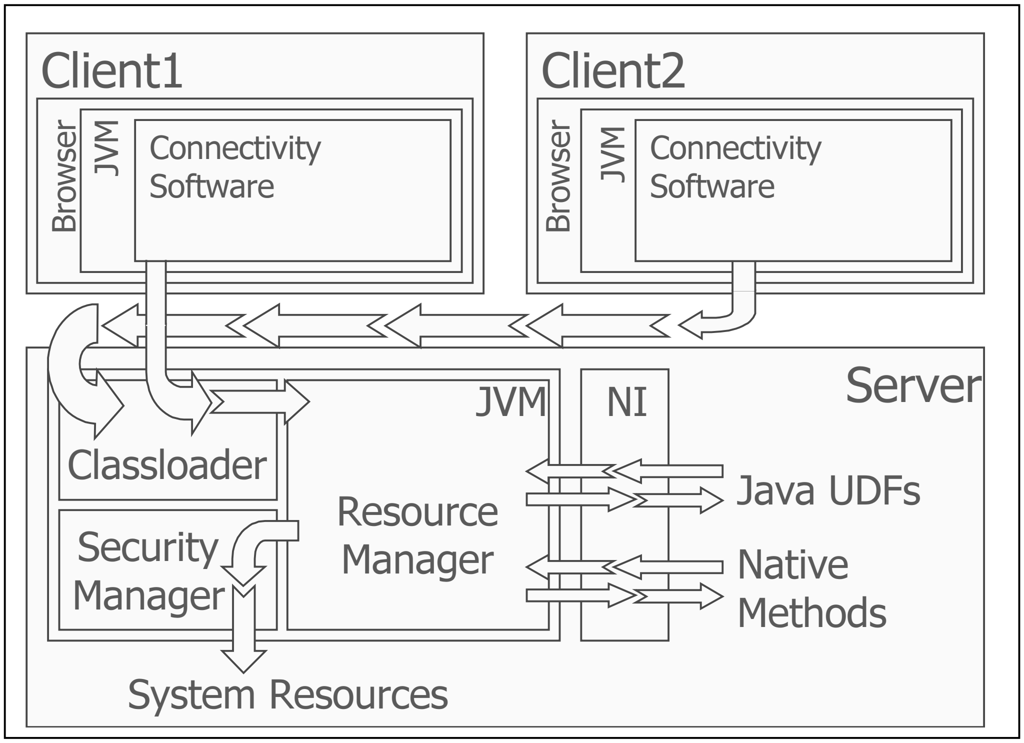

Figure 2. The design and operation of Jaguar extended with Resource Manager.

subsystem about resource usage of a particular thread `t`; `getResourceLimit(resType, t)` can be used to query resource limits.

The actions that are available when a resource bound is exceeded are limited by the control mechanisms that are part of the JVM. For instance, it is possible to lower a thread's priority but it is impossible to change the thread scheduling algorithm. Another limitation is the inability to track memory allocated in the native code. This is due to the fact that most of JRes is implemented through bytecode rewriting [CvE98].

The design and operation of our current prototype is shown in Figure 2. In this example setup, two remote clients submit their queries through a Web interface. The UDF code (i.e. Java classes) is loaded by the Jaguar class loader. The subsequent execution is controlled by what the standard Java Security Manager and the JRes Resource Manager allow.

Functionality	UDFs' involvement	Server's involvement
Optimization	UDFs that query the database server environment in order to adjust their execution and improve performance.	Database servers that optimize query execution through utilizing resource consumption information.
Security	UDFs that query the database server environment in order to avoid using more resources than allowed.	Database servers that monitor resource consumption of UDFs to detect malicious behavior.

Responsibility for using resource information.

Table 1. Dimensions of applicability of UDFs.

4 Design Space

Before describing specific solutions and approaches let us take a look at possible dimensions along which a resource monitoring facility can be taken advantage of in an extensible object-relational database system. The first dimension roughly quantifies the UDF programmer's involvement in monitoring the resources. One end of the spectrum is populated by UDFs that monitor their own resource consumption and the resource limits to adjust their execution patterns with respect to changing resource availability. A UDF that dynamically adapts the accuracy of the produced results to the availability of resources forms an example. The other end of the spectrum consists of systems that monitor the resources available to extensions and apply this information to change execution of queries containing UDFs. A database server dynamically reordering conjunctive predicates depending on their resource usage would be placed here.

The other, orthogonal dimension is the domain of application of knowledge about both system-wide and per-UDF resource consumption. One such domain is security - detection of malicious UDFs and preventing denial of service attacks. Another domain is optimization, where combining knowledge of resource demands and their availability may lead to improved execution times of UDFs.

It is important to stress that the same system may occupy more than one quadrant in the outlined space. Using information concerning resource utilization and availability for optimization does not preclude its usage for enhancing system security. Similarly, both the UDFs and an object-relational database itself can independently take advantage of JRes feedback at the same time. Table 1 summarizes the classification introduced above and gives examples belonging to each of the groups.

5 Enhanced Database Security using JRes

As stated earlier, protection provided by a safe language is only one component of the necessary security infrastructure provided by extensible environments. Another vital part, neglected so far in available designs, is the ability to control resources available to extensions and the subsequent ability to detect and neutralize malicious or otherwise *resource-unstable*UDFs. Since the class of database servers discussed in this paper falls into the 'extensible environments' category, it is crucial for an unimpeded development and deployment of this new data access technology to pay attention to resource monitoring issues.

Figure 3 shows one possible policy that limits each UDF to one thread only. Moreover, such a thread is limited to no more than 50kB of memory and less than 10 milliseconds of CPU time out of every 100 milliseconds. Whenever the memory limit is exceeded, an appropriate exception is thrown. In addition to signalling a problem, this effectively prevents the operation of object creation from completion. Exceeding the time limit results in lowering the offending thread's priority; if the priority cannot be lowered any more, the thread is stopped using a method `stopThread()` (not shown in Figure 3). It must be pointed out that stopping threads should be dealt with carefully, since threads may own state or other resources, like open files, which may need to be saved or cleaned up appropriately before killing the thread.

6 Design of Resource Control Feedback for Java UDFs

The JRes interface allows for retrieving information about current system-wide resource availability and per-UDF consumption. This information can be used in several ways to improve either overall system performance or the performance of "smart" UDFs. In this section we describe several scenarios that show usage and applicability of the Jaguar/JRes resource monitoring. In the next section, we demonstrate the performance impact when Jres is used in this fashion.

6.1 Obtaining UDF Costs as a Function of Input Arguments

[Sesh98b] explores optimizations on the boundary between relational query execution and the execution of UDFs and method extensions. The paper identifies four categories of optimization opportunities and studies techniques applicable to each of the categories. An important category requires knowledge of the resource consumption of the UDFs. Our work provides a practical framework in which resource utilization information can be obtained and used for improving query plans. For instance, let us consider the following query

```
SELECT C.Name
```

```
class ExtensibleDBServerRMP
  implements ThreadRegistrationCallback, OveruseCallback {

  private Object cookie;

  private ExtensibleDBServerRMP(Object cookie) { this.cookie = cookie; }

  public static synchronized void initialize() {
    Object cookie = new Object();
    ResourceManager.initialize(cookie);
    ExtensibleDBServerRMP rmp = new ExtensibleDBServerRMP (cookie);
    ResourceManager.setThreadRegistrationCallback(cookie, rmp);
  }

  public void threadRegistrationNotification(Thread t) {
    if (t.getThreadGroup().getName().equals("system")) { return; }
    if (udfHasThreadsAlready(t)) {
      stopThread(t);
    }
    ResourceManager.setLimits(cookie, RESOURCE_CPU, t, 10, 100, this);
    ResourceManager.setLimits(cookie, RESOURCE_MEM, t, 50, 0, this);
  }

  public void resourceUseExceeded(int resType, Thread t, long value) {
    if (resType == RESOURCE_CPU) {
      int priority = t.getPriority();
      if (priority == Thread.MIN_PRIORITY) {
        stopThread(t);
      } else {
        t.setPriority(priority - 1);
      }
    } else if (resType == RESOURCE_MEM) {
      throw new JResResourceExceededException(“memory”);
    }
  }
}
```

Figure 3. An example resource controlling policy for user defined functions.

```
FROM Companies C
WHERE LooksPromising(C.ClosingPrices) = true
  AND ExternalRating(C.Name) > 0.9
  AND Profitability(C.Name) = “Outstanding”
```

The three UDFs-predicates are “black boxes” from the viewpoint of both the underlying database and the module managing the extensibility. In order to generate the optimal plan, the query optimizer must know the selectivity and cost of each predicate involved. Thus, an off-line or on-line gathering of performance and selectivity data is necessary in order to provide the query optimizer with the required information. In the example above, some predicates may access the network (for instance, `ExternalRating` may have to communicate with other databases), some may be very CPU-intensive, and others may use large quantities of memory.

Applying JRes to off-line generate a table associating input sizes with execution time, bytes sent and received, and the maximum amount of memory is simple. However, such a table makes sense only if the input size determines the resource consumption. The process of generating such tables may sometimes uncover that there is simply no correlation between the argument size and the resources consumed by the UDF.

6.2 Dynamic Predicate Reordering Based on Resource Consumption

It is often not possible to execute a query off-line - for instance when it has been submitted by a user during an interactive session with a database server. In such settings, Jaguar augmented with JRes is used to gather dynamic resource profiles. The information can then be fed dynamically to the execution engine, which may change the order of predicate execution based on similar criteria as in the static case.

Dynamic resource monitoring has one advantage over static monitoring: relative values of resources are known, so better localized adjustments can be performed. Let us assume that in the example query from the previous subsection `Profitability` (very CPU-intensive) is applied after the equally selective `ExternalRating` (which consumes large quantities of network bandwidth). The order of predicates will change during the same query execution whenever the system detects that due to the presence of other queries and UDFs in the system there is currently contention for the network while a relatively large amount of CPU time is available. The predicates with high costs, in terms of the currently sparse resources, are executed later, thus benefitting from the selectivity of earlier predicates.

6.3 Dealing with Resource Shortages without Reduced Quality

As described in detail in [Pang94], queries executing in a priority scheduling environment face the prospect of continually having resources taken away and then given back during their lifetime. The same statement is true for UDFs as well, especially for those in queries with long lifetimes; typically this category would include UDFs operating on large data inputs. Let us take a look at UDFs for which the quality of a result may not suffer but the completion time may worsen. For instance, let us consider a query that invokes a UDF in order to determine whether one image contains another:

```
SELECT P.name
FROM Paintings P, Cats C
WHERE Contains(P.image, C.image) = true
```

The images are stored in a compressed format and `Contains()` has to decompress them in order to run a pattern-matching algorithm. If memory is scarce, only parts of images may be decompressed. This will make the pattern matching process more time intensive while the results will be the same, and, more importantly, invocations of `Contains` will not be prematurely aborted because of lack of memory.

6.4 Adjusting Quality of UDF Results when Necessary Resources are Scarce

In some scenarios, adapting to resource scarcity may be accomplished by degrading the quality of output. Examples include faster image operations resulting in worse quality of results that are nevertheless useful for the end user. Another such example can be seen through the eyes of a user of a financial database. Her UDFs return approximations of the standard deviation of an input time series. The CPU time available to any UDF invocation can be limited system-wide in order to make quick response times more likely for a large population of users. In this setting, the UDF must complete without using more resources as given - otherwise, it will be terminated and no result will be produced. Thus, while there is no bound on the length of the time series, the time available to the UDF is bounded. The UDF can query JRes for the CPU time available to itself. This, in turn, can be used to compute the number of entries of the input series that can be processed before using up the quota. If less then the whole series can be processed, it is up to the UDF to decide which ones; the most plausible choices include sampling with a fixed step size or using the most recent section of the time series. The return value may be less precise than whatever could be computed with unlimited resources, but is still a much better alternative than getting nothing back because the UDF's execution has been aborted.

6.5 Exploiting Resource Tradeoffs

In some scenarios, one resource can be traded off for another in order to mask temporary or recurring fluctuation in resource availability. One example has been presented in Section 6.3. Another one is, for instance, a UDF that sends data back directly to the client via a network connection may choose to send compressed results or to send the data "as is". In the first case, more CPU time but less bandwidth is needed; the reverse holds in the second scenario. The most common form of trading resources off for one another is caching, where memory (main or disk) is traded off for whatever resources were consumed to generate cached data. Let us take a look at the following join, where the UDF `Similar` detects a similarity between two time series, retrieved from some other table or from a file system:

```
SELECT D1.name, D2.name
FROM Data D1, Data D2
WHERE Similar(D1.name, D2.name) > 0.75
```

A naïve way of coding `Similar` is to retrieve time series based on the names of arguments, compare inputs, and return the value describing the similarity. However, since the UDF is invoked repeatedly in this query, simple optimizations are possible. If the query is executed using a nested-loop join algorithm (scanning D1, and for each tuple, finding a "matching" tuple in D2), the UDF will be invoked several times with the same first argument. The UDF code could choose to cache the first argument, thereby using memory to reduce CPU and I/O time.

7 Performance Study of Run-Time Adaptation

While previous parts of this paper discussed possible uses of resource monitoring mechanisms, this section focuses on quantifying the impact of using JRes in Jaguar. The experimental results presented below were obtained on a Pentium II 300 MHz computer with 128 MB of RAM, running Windows NT Workstation 4.0. The Java Virtual Machine used by Jaguar was Microsoft Visual J++, v. 1.1. Each of the experiments uses a table T with 1000 distinct tuples, each holding two integers, one of which is an identifier for a time series.

To set the stage for our experiments, let us consider three UDFs: UDF-1, UDF-2 and UDF-3. Each of them takes as an argument an integer identifying a certain time series and returns a boolean value. The costs of these UDFs are considerably larger than the costs of simple predicates (e.g. integer comparisons). The first two UDFs use caching to internally store results of their computations - sorting and computing various statistical moments of time series. UDF-3 does not cache its results and thus its execution time does not depend on the amount of memory available to it. Figure 4 shows the average execution time of each of the UDFs as a function of the amount of memory available per UDF. There were a hundred distinct time series involved; all of them fit into a 800kB cache. Two points are worth stressing here. First, the results of Figure 4 can be easily obtained and can then be used by a static query optimizer. Second, UDF-1 and UDF-2 are examples of user defined functions that utilize a possible resource tradeoff. In this particular case, increased consumption of main memory (caching) versus reduced need for CPU time. The two experiments shown in the following subsections indicate possible performance gains due to employing Java resource consumption monitoring.

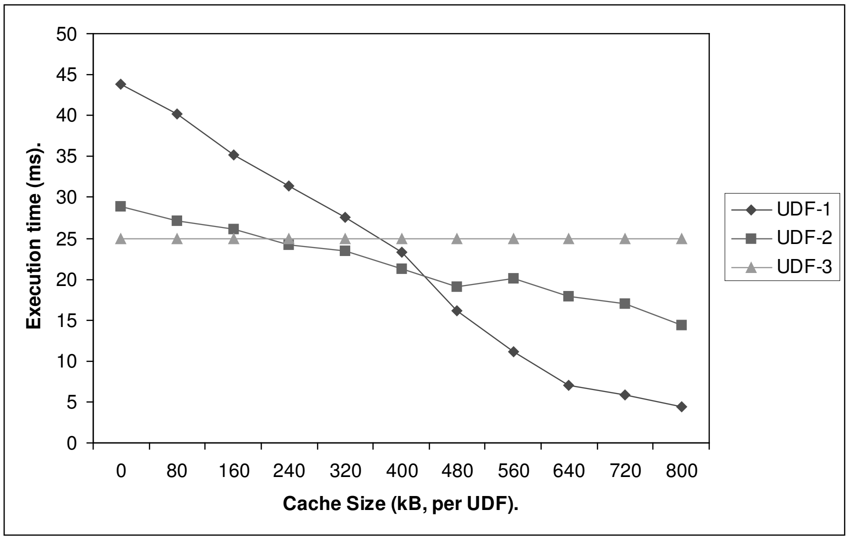

Figure 4. Execution time of three UDFs as a function of available memory.

7.1 Dynamic Predicate Reordering

The three UDFs were coded so that they have the same selectivity (on the average, each of them returns `true` for 30% of its inputs) and in fact always return the same answer if given the same input argument (i.e. whenever UDF1 is true, so are UDF2 and UDF3, and vice versa). Consider the following query:

```
SELECT T.timeseries
```

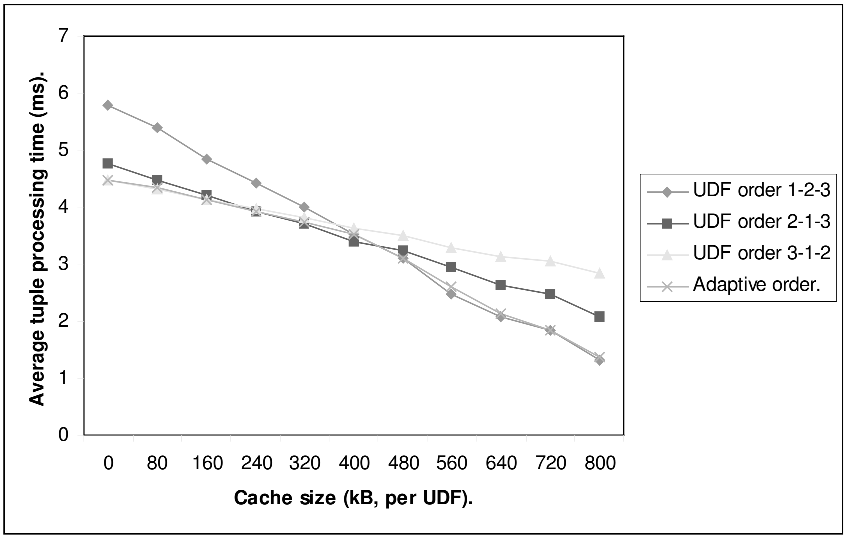

Figure 5. Execution time of different predicate ordering strategies.

```
FROM T
WHERE UDF1(T.timeseries) AND UDF2(T.timeseries) AND UDF3(T.timeseries)
```

The execution time depends on the order in which the predicates are applied and on the amount of memory available to the UDFs . Every nontrivial predicate is associated with a certain cost and a certain selectivity. The latter determines the average ratio of tuples on which the predicate results in true. Selective and cheap predicates should be applied before less selective and more expensive predicates to reduce the overall execution cost. We picked three different evaluation orders: 1-2-3, 2-1-3, and 3-1-2[1], and compared their costs with the cost of an dynamically

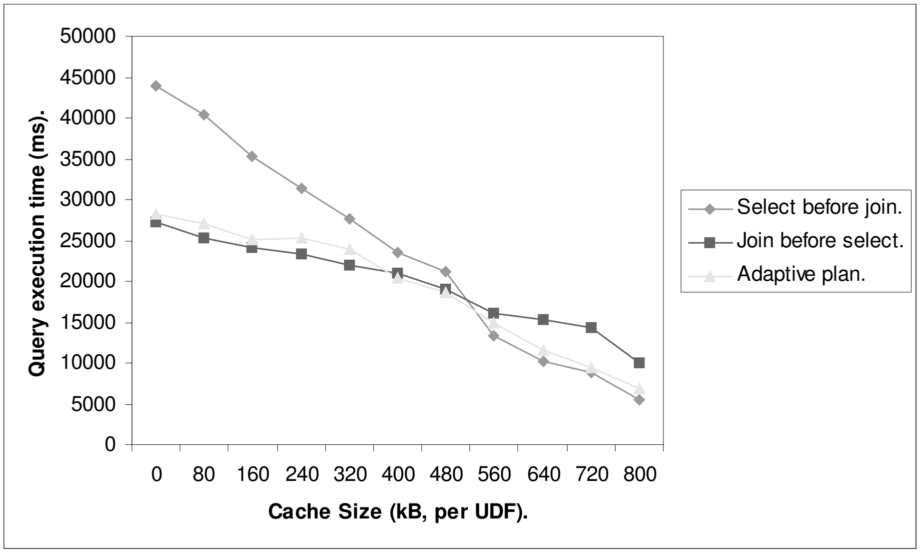

Figure 6. Execution time of different plans.

[1] Because all three UDFs have the same results on identical arguments, it only matters which predicate is evaluated first: If it

adapted order. We varied the available cache size, changing the relative costs of the predicates and thus their optimal order. Figure 5 shows the average per-tuple processing time for each of the three given evaluation orders and for an adaptive strategy. The latter monitored available memory and applied this information to dynamically optimize the evaluation order. Incurring a small overhead for the dynamic plan modification, the adaptive strategy always chooses the best order for the predicates.

If the three UDFs were coded as one large UDF invoking the three tests by itself, the reordering could be done inside the UDF as well. Thus, in the case of predicates applied to the same input, it is possible (with a bit of additional work) to re-code them as a single predicate. The resulting UDF can manage the order of its sub-predicates by utilizing the resource consumption and availability feedback coming from JRes.

7.2 Reordering Join and Selection Operations

Let us now consider the following query, operating on a table T (with 1000 tuples, each of them consisting of two integers; the first one serves as a reference to a stored time series) and a table S (containing 10000 tuples, each of them also consisting of two integers):

```
SELECT *
FROM T, S
WHERE T.a = S.a and UDF1(T.a)
```

Due to the equality predicate that is used in the join between T and S, the join has a certain selectivity with respect to the table T. The application of UDF1 can take place either before or after the join, changing the cost of the overall query execution. Applying UDF1 before the join results in an invocation of UDF1 on each tuple of T, but reduces the number of tuples of T that have to be joined. On the other hand, applying the join first requires less invocations of UDF1. The total cost of the query is different in both cases. Our prototype can change the plan dynamically (e.g. during query execution). Figure 6 shows how the two static strategies perform under changing memory availability and contrasts it with the performance of the dynamically adapted plan. The adaptation - i.e. applying selection before or after the join - is done similarly to the previous experiment: the resource monitoring information is used by Jaguar to change the plan while it is executing. As Figure 6 demonstrates, the performance gains can be quite substantial when memory availability changes frequently. As in the previous experiment, with a small overhead the adaptive strategy follows the best, hybrid plan. Let us note that in this particular experiment, unlike in the previous one, the query plan reordering can only be the responsibility of the database query execution module -- it cannot be taken over by an adaptive UDF.

7.3 Overheads Introduced by JRes

The benefits of on-line resource monitoring come at a price of runtime overheads. For the UDFs used in our experiments, the overheads are within 3-6% of execution time. The overheads are directly proportional to the number of objects allocated by UDFs and in some cases can be substantial [CvE98]. The overheads may be reduced if JRes is implemented integrated in a JVM. Still, increased system security and the ability to adapt both execution plans and UDF execution has to be weighed against the slightly increased execution time.

8 Related Work

Past work related to our research falls into three broad categories: (i) predicting and controlling resource consumption in existing database systems, (ii) resource accounting and enforcing resource limits in traditional and extensible operating systems, and (iii) using safe language technologies as a mechanism for providing extensibility. In this section we summarize the most important work from these areas influencing our research.

8.1 Database Systems Approaches

Several database systems and standards allow the implementation of functions in C, C++ or Java, either as predicates or as general functions. The examples include POSTGRES [SR86], Starburst [HCL+90], Iris [WLH90], and several commercially available systems - for instance Informix, DB2, Oracle 8. The issue of expensive predicate optimization was first raised in the context of POSTGRES [Sto91] and a practically applicable theory addressing the issue was developed in [HS93]. The goal of recent work by Hellerstein and Naughton [HN97] is to optimize the

returns false, the later two are not evaluated; if it returns true, both others are evaluated. Thus the three picked permutations are equivalent in their complexity to 1-3-2, 2-3-1, and 3-2-1, respectively.

execution of queries with expensive predicates by caching their arguments and results. The resulting technique, Hybrid Caching, is promising in the presence of repeated invocations of a predicate on the same arguments.

Obtaining realistic estimates of the costs of user defined methods is difficult and quite often imprecise [Hel95]. Typically, it is assumed that, along with estimating selectivity, the creator or user of a UDF will provide a cost estimate as well. Assuming that cost estimates are correct and remain constant throughout the entire execution of the query, it is possible to efficiently generate an optimal plan over the desired execution space [CS96].

Another line of research refines query optimization by focusing on join reordering where an important working assumption is that predicates are zero-cost [IK84, KBZ86, SI92]. A general formulation of query optimization for various buffer sizes can be found in [INS+92]. This runtime parameter is typically unknown before the actual query execution. By constructing various plans in advance, the most appropriate one can be chosen at run-time just before the query is executed, when the available buffer size is known. Another technique helping with estimation of the query size is adaptive sampling [LNS90], where statistical methods are used to predict the result size based on selective runs of the estimated query. Completing joins and sorts under fluctuating availability of main memory has been the subject of recent research by [Pang94].

Dynamic query optimization was incorporated into a commercially available Rdb/VMS system [Ant93]. The research suggests that it is cost-effective to run several local plans simultaneously with proportional speed for a short time, and then select the “best” plan to be run for a long time. An optimization model that assigns the bulk of the optimization effort to compile-time and delays carefully selected optimization decisions until runtime is described in [CG94]. Dynamic plans are constructed at compile-time and the best one is selected at runtime, when cost calculations and comparisons can be performed. The approach guarantees plan optimality. However, none of these approaches deals with unknown and changing costs of user defined functions.

Our work differs from the research mentioned above in our focus on UDFs and on monitoring the environment in which UDFs execute. In addition to providing the ability to run queries off-line to get estimates of their cost, our system constantly monitors resource utilization. This information is available directly both to the UDFs themselves and the query execution module. Both the database system and UDFs can utilize this knowledge directly and dynamically.

8.2 Operating Systems Approaches

Enforcing resource limits has long been a responsibility of operating systems. For instance, many UNIX shells export the `limit` command, which sets resource limitations for the current shell and its child processes. Among others, available CPU time and maximum sizes of data segment, stack segment, and virtual memory can be set. Enforcing resource limits in traditional operating systems is coarse-grained in that the unit of control is an entire process. The enforcement relies on kernel-controlled process scheduling and hardware support for detecting memory overuse.

The architecture of the SPIN extensible operating system allows applications to safely change the operating system’s interface and implementation [BSP+95]. SPIN and its extensions are written in Modula-3 and rely on a certifying compiler to guarantee the safety of extensions. The CPU consumption of untrusted extensions can be limited by introducing a time-out. Another example of an extensible operating system concerned with constraining resources consumed by extensions is the VINO kernel [SES+96]. VINO uses software fault isolation as its safety mechanism and a lightweight transaction system to cope with resource hoarding. Timeouts are associated with time-constrained resources. If an extension holds such a resource for too long, it is terminated. The transactional support is used to restore the system to a consistent state after aborting an extension.

The main objective of extensible operating systems is to allow new services to be added to the kernel and for core services to be modified. Their built-in and “hard-coded” support for resource management is adequate for an operating system. In contrast, the main motivation behind JRes is building extensible, safe and efficient Internet environments implemented entirely in a safe language, such as Java. An extension may be an entire application and various billing, accounting, and enforcing policies may have to be effective at the same time.

8.3 Programming Languages Approaches

Except for the ability to manipulate thread priorities and invoke garbage collection, Java programmers are not given any interface to control resource usage of programs. Several extensions to Java attempt to alleviate this problem, but none of them share the goals of JRes. For instance, the Java Web Server [JWS97] provides an administrator interface that displays resource usage in a coarse-grained manner, e.g. the number of running threads; however, the information about memory or CPU used by each individual thread is not accessible.. PERC (a real-time implementation of Java) [Nils96] provides an API for obtaining guaranteed execution time and assuring resource availability. While the goal of real-time systems is to ensure that applications obtain *at least* as many resources as necessary, the goal of JRes is to ensure that programs do not exceed their resource limits.

A very recent specialized programming language PLAN [HKM+98] aims at providing an infrastructure for programming Active Networks. PLAN is a strictly functional language based on a dialect of ML. The programs replace packet headers (which are viewed as 'dumb' programs) and are executed on Internet hosts and routers. In order to protect network availability, PLAN programs must use a bounded amount of space and time on active routers and bandwidth in the network. This is enforced by the language runtime system. JRes is similar to PLAN in that it limits resources consumed by programs. The main difference is that PLAN pre-computes resources available to programs based on the length of the program. The claim is that resources for an Active Networks program associated with a packet should be bounded by a linear function of the size of the packet's header. JRes allows the implementation of flexible policies that regulate the resource availability for UDFs dynamically in dependence of the current workload.

9 Conclusions

The security and functionality of an extensible database server can be enhanced by providing resource controlling mechanisms to the language used for creating user defined functions. Because of the combination of portability, security and object-orientation Java emerges as a premier language for creating extensible environments. Our work evaluation, in the context of extensible database servers, of a resource controlling interface we have developed for general purpose Java programs. To the best of our knowledge, no database system supporting UDFs (or, for that matter, no other extensible server system that does not rely on hardware protection) currently provides the functionality we have added to Jaguar. The presented description of the system design, the evaluation of resource monitoring, and the provision of mechanisms for adaptive behavior are important steps towards practical extensible servers.

In particular, our work further limits the amount of trust that the database server must have with respect to the behavior of extensions. The standard JVM controls access of UDFs to security-sensitive resources such as files and network. This paper demonstrates that a class of UDFs that may execute in a database server without affecting the execution of the server or other extensions has been enlarged to contain UDFs with unknown and potentially malicious or unbalanced resource requirements. Moreover, the paper shows that the execution cost of a UDF may depend on the dynamic supply of computational resources. Thus, changing query plan dynamically, during the query execution, is necessary to achieve optimal performance. Jaguar extended with JRes provides appropriate mechanisms for achieving this goal.

Even though this work is carried out in the context of an extensible object-relational database and Java extensions, the conclusions generalize to any system where Java code dynamically extends an execution environment, like a Web browser or an extensible Web server. The security can be enhanced and performance concerns can be addressed in such environments in a similar way to our prototype implementation analyzed in the paper.

10 References

[Ant93] G. Antoshenkov. *Dynamic Query Optimization in Rdb/VMS.* International Conference on Data Engineering 1993: pages 538-547.

[BSP+95] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System.* 15th ACM Symposium on Operating Systems Principles, p.267-284, Copper Mountain, CO, December 1995.

[CG94] R. Cole and G. Graefe. *Optimization of Dynamic Query Evaluation Plans.* ACM SIGMOD'94.

[CS96] S. Chauduri, K. Shim. *Optimization of Queries with User-defined Predicates.* Proc. 23rd International Conference on Very Large Database Systems, 1996.

[CvE98] G. Czajkowski, and T. von Eicken. *JRes: A Resource Accounting Interface for Java.* ACM OOPSLA'98, Vancouver, BC, October 1998.

[GMS+98] M. Godfrey, T. Mayr, P. Seshadri, T. von Eicken. *Secure and Portable Database Extensibility.* ACM SIGMOD'98.

[HCL+90] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey and E. Shekita. *Starburst Mid-Flight: As the Dust Clears*. IEEE Trans. on Knowledge and Data Engineering, March 1993.

[Hel95] J. Hellerstein. *Optimization and Execution Techniques for Queries with Expensive Methods*. Ph.D. Thesis, University of Wisconsin-Madison, May 1995.

[HKM+98] Hicks, M, Kakkar, P, Moore, J, Gunter, C, and Nettles, S. *PLAN: A Programming Language for Active Networks.* Submitted to ACM SIGPLAN Conference on Programming Language Design and Implementation, 1998.

[HS93] J. Hellerstein, M. Stonebraker. *Predicate Migration: Optimizing Queries with Expensive Predicates.* ACM SIGMOD'93.

[HN97] J. Hellerstein, J. Naughton. *Query Execution Techniques for Caching Expensive Methods*. ACM SIGMOD'97.

[HKM+98] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. *PLAN: A Packet Language for Active Networks.* To appear in the International Conference on Functional Programming (ICFP) '98.

[IK84] T. Ibaraki, T. Kameda. *Optimal Nesting for Computing N-Relational Joins.* ACM Transactions on Database Systems, October 1984.

[INS+92] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis. *Parametric Query Optimization*. Proc. 18th International Conference on Very Large Database Systems, 1992.

[Java] JavaSoft. *JavaServer Documentation*. http://java.sun.com/products/java-server.

[JNI] Java Native Interface. *http://www.javasoft.com/products/jdk/1.1/docs/guide/jni/index.html.*

[JWS97] Java Web Server. http://jserv.javasoft.com/products/webserver.

[KBZ86] R. Krishnamurthy, H. Boral and C. Zaniolo. *Optimization of Nonrecursive Queries.* Proc. 12th International Conference on Very Large Database Systems, 1986.

[LNS90] R. Lipton, J. Naughton, D. Schneider. *Practical Selectivity Estimation through Adaptive Sampling.* ACM SIGMOD'90.

[Nils96] Nilsen, K. Issues in the Design and Implementation of Real-Time Java. Java Developer's Journal, 1996.

[Pang94] Hwee Hwa Pang. *Query Processing in Firm Real-Time DatabaseSystems.* Ph.D. Thesis, University of Wisconsin Madison, March 1994.

[RNI] Raw Native Interface. *http://www.microsoft.com/java/sdk.*

[SES+96] M. Seltzer, Y. Endo, C. Small, and K. Smith. *Dealing with Disaster: Surviving Misbehaved Kernel Extensions*. 2nd Symposium on Operating Systems Design and Implementation (OSDI), p. 213-227, Seattle, WA, October, 1996.

[Sesh98a] P. Seshadri. *Enhanced Data Types in PREDATOR*. VLDB Journal 1998.

[Sesh98b] P. Seshadri. *Relational Query Optimization with Enhanced ADTs.* Technical Report TR98-1693, Cornell University, Computer Science Department, Ithaca, NY, 1998.

[SI92] A. Swami and B. Iyer. *A Polynomial Time Algorithm for Optimizing Join Queries.* Research Report RJ8812, IBM Almaden Research Center, June 1992.

[SSL97] Secure Socket Layer. *http://developer.netscape.com/docs.*

[SQLJ] SQL: Embedded SQL for Java - Tutorial. *http://www.oracle.com/st/products/jdbc/sqlj/sql_specs.html*. June 1997.

[Sto91] M. Stonebraker. *Managing Persistent Objects in a Multi-Level Store.* ACM SIGMOD'91.

[SR86] M. Stonebraker and L. Rowe. *The Design of POSTGRES.* ACM SIGMOD'86.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. *The Iris Architecture and Implementation*. IEEE Transactions on Knowledge and Data Engineering, March 1990.