Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System

Mehul A. Shah
mashah@cs.berkeley.edu

Samuel Madden
madden@cs.berkeley.edu

Michael J. Franklin
franklin@cs.berkeley.edu

Joseph M. Hellerstein
jmh@cs.berkeley.edu

ABSTRACT

Database system designers have traditionally had trouble with the default services and interfaces provided by operating systems. In recent years, developers and enthusiasts have increasingly promoted Java as a serious platform for building data-intensive servers. Java provides a number of very helpful language features, as well as a full run-time environment reminiscent of a traditional operating system. This combination of features and community support raises the question of whether Java is better or worse at supporting data-intensive server software than a traditional operating system coupled with a weakly-typed language such as C or C++.

In this paper, we summarize and discuss our experience building the Telegraph dataflow system in Java. We highlight some of the pleasures of coding with Java, and some of the pains of coding *around* Java in order to obtain good performance in a data-intensive server. For those issues that were painful, we present concrete suggestions for evolving Java's interfaces to better suit serious software systems development. We believe these experiences can provide insight for other designers to avoid pitfalls we encountered and to decide if Java is a suitable platform for their system.

1. INTRODUCTION

From the original relational prototypes and onwards, database system designers have complained about the interfaces and features provided in popular operating systems [23]. Over the course of many years, a number of these interfaces have been changed by the OS vendors – or subverted by the DBMS vendors – in order to achieve correct behavior and good performance.

In recent years, developers and enthusiasts have increasingly promoted Java as a serious platform for building data-intensive systems. Among Java's main attractions are its programming language features that speed development and prevent subtle bugs – these include strict type checking, array bounds checking, and mandatory exception handling. Also attractive are Java's lower-level features including built-in memory management, convenient I/O and threading libraries, dynamic linking, built-in synchronization mechanisms, and support for secure execution of foreign code. In recent years, Java compilers have developed to the point where they do almost as well as traditional C compilers in generating efficient code on microbenchmarks [10]. But in addition to traditional user-level code specification, a Java platform includes a *virtual machine*, which (as is clear from the description above) provides many of the abstractions traditionally associated with an operating system. This combination of features and community support raises the question of whether the Java language and platform is better at supporting database system functionality than a traditional operating system coupled with a weakly-typed language like C or C++.

We have developed the first version of the Telegraph adaptive dataflow system, which is intended to form a basis for data-intensive applications over volatile environments including sensor networks and the Internet. Along with our first implementation of the basic Telegraph system, we also built our first prototype application, which supports OLAP-style queries over "deep web" data sources [7] via a web screen-scraper, relational algebra operators, and online aggregation interfaces. We launched out prototype as a public web service one month before the 2000 Presidential election. The system joined Presidential campaign finance websites with other "deep web" information including home prices, celebrity lists, neighborhood crime information, and a map server. The prototype attracted local media attention and was used by thousands of clients on the Internet.

In choosing a development platform for Telegraph, we were attracted by Java's ease of use. We were also curious to see if it could provide good performance for a data-intensive system. Our Telegraph prototype is a departure from a standard DBMS in three major respects: it is based on eddies [6] rather than on a static query optimizer, it is currently read-only with no transactional support, and it uses pipelining dataflow operators to support online aggregation. Despite these distinctions, it does stress the underlying machine services in a manner analogous to traditional database systems. It must handle multiple concurrent users, it requires careful memory management for efficiency, and it taxes both the network and disk I/O subsystems. In addition, the prototype was designed as an extensible framework in which users can register their own functions and web wrappers.

In this paper we summarize and discuss our experiences with Java's interfaces and supporting computing environment: core libraries, Java virtual machine (JVM), and tools. Like earlier critics of operating system support, we find Java's support lacking in a number of respects. Some of these limitations stem from the familiar problem that narrow, "conve-

nient" interfaces suited to simple programs do not provide the degree of control needed to make complex, I/O intensive software perform well. In other cases, we found that some of Java's mechanisms had high overhead implementations, forcing us to code *around* these mechanisms in order to obtain acceptable performance. Some of these latter problems were artifacts of the particular JVM and operating system we used, others are inherent in Java's interfaces. Some of these observations have been noted in other contexts [14], [11], [25], [24]; however, we are unaware of any work that collectively presents these ideas in the database systems context, which has its own needs and peculiarities. In addition to identifying these problems, we highlight some of the pleasures of coding with Java. We present our results not as an indictment of the Java platform, but as suggestions for evolving that design to better suit I/O intensive software systems development. Also, we hope these results provide insight for other developers to avoid pitfalls we encountered and to determine if Java is the right platform for their system.

1.1 Structure of the Paper

In Section 2, we describe the architecture of our initial Telegraph prototype. In Section 3, we describe the advantages and drawbacks of the Java memory management model. Then, in Section 4, we discuss multi-programming issues in Java and how they impacted the Telegraph design. In Section 5, we describe how Telegraph leverages Java's introspection mechanism for UDFs and catalog evolution. Section 6 contains our experiences with the Java core libraries and development tools. Section 7 highlights pieces in traditional database systems that might stress Java's interfaces but were not implemented in the Telegraph prototype. Section 8 surveys the related work, and Section 9 concludes.

2. ARCHITECTURE

In this section, we provide a summary of the initial Telegraph architecture. As mentioned previously, the first application of Telegraph was to perform interactive, read-only queries over web-based sources. To run these queries efficiently in such volatile environments, we abandoned the traditional approach of using a cost-based optimizer and static query processing engine. Instead, we chose to build the Telegraph prototype using the eddy [6] framework for adaptive query processing. The Telegraph prototype runs on a single computer, though shared-nothing parallelism in Telegraph is a subject of ongoing research in our group. Below, we describe the components of the Telegraph query processor and how they interact.

The principal components of the Telegraph prototype query processor are the eddy, dataflow modules like relational operators, and the queues through which they communicate. Figure 1 shows an example of a Telegraph query on two sources R and S. The query includes an equality join on the b fields of the sources and selection modules on $R.a$ and $S.c$.

Unlike a static query plan, the modules in an eddy

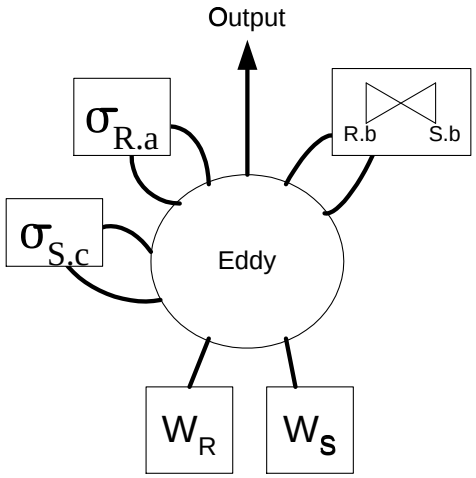

Figure 1: A query on two sources R and S in Telegraph. Modules are shown in boxes, and edges represent queues. The eddy routes data through the modules for computing the query.

are decoupled and work asynchronously. The responsibility of the eddy is to route data from the source wrappers, W_R and W_S, through the appropriate modules for computing the query, and then send the results to the output. The eddy makes routing decisions at runtime on a per-tuple basis based on observed throughput of the modules. A more detailed discussion of routing policies for eddies can be found in [6].

The dataflow modules consume data provided by the eddy on their input queues and return the results of their computation back to the eddy on their output queues. These modules can be relational operators like selection, aggregation, and join, or any arbitrary user-defined function. In Section 4, we discuss in depth how modules are scheduled.

Telegraph's memory organization is also different from that of a traditional database system. The memory in Telegraph is divided between the modules' private state and a tuple pool. Because the Telegraph prototype is targeted at network data streams, it has no need for a paginated, disk-based buffer pool. Instead Telegraph has a single tuple pool for all in-flight tuples, i.e. the tuples being passed among the queues and in the eddy. Since variable length tuples are removed and inserted into the tuple pool in an unpredictable order, the tuple pool can become fragmented. Thus, there is a need for memory compaction. Furthermore, to avoid copies, tuples are passed by reference on the queues to the modules. Hence, data sharing in the tuple pool can occur when a tuple is broadcast to multiple modules. Compaction and data sharing have implications for memory management and synchronization mechanisms discussed in the next sections.

We store the data for each tuple in a Java bytearray, and marshal the tuple's fields into its native Java type on every field access. This design decision is a result of Telegraph's need for tighter control over

memory management. More detail on this topic is also presented in the next section.

To conclude, we illustrate the life of a tuple as it is created from the wrappers and its travels through the modules. This process is similar to the way a traditional query processor flows data through a query plan. First, a wrapper must convert raw text data from a web-source into a formatted byte-array in the eddy tuple pool, and place a handle to the byte-array on its output queue. The eddy receives a handle to a tuple from a wrapper, say W_R, through the wrapper's output queue, and can pass the handle to the tuple to any one of the modules for R in the query. The module has several choices: it can deallocate the tuple, it can make a copy for later use, or it can pass the handle back to the eddy. The selection module, for example, may deallocate the tuple if it does not satisfy the predicate, or pass the handle back if it does. The hash join, on the other hand, would make a private copy of the tuple and deallocate it from the tuple pool. Once a tuple has passed through all the necessary modules, it remains in the tuple pool until copied from the output to a client.

Our initial Telegraph application was deployed in September, 2000 at the site http://fff.cs.berkeley.edu/. The experiments in this paper were performed on a single site system running on a dual-processor 667 MHz Pentium III with Linux 2.2.16. We used the SUN JDK 1.3 for development, and the Java HotSpot Server JVM build 1.3.0 as the execution environment.

3. MEMORY MANAGEMENT

3.1 Allocation and Deallocation

The task of a query processing engine is to efficiently manage the flow of data through modules by using the available physical memory and disk resources intelligently. Thus, careful memory management is essential to the performance of a query processor. Over-utilization of memory can lead to unpredictable and poor performance from the underlying virtual memory system, and under-utilization can result in excessive I/O costs. Full utilization of physical memory can be achieved through tight control over memory operations.

Most commercial database systems include a special set of memory management services [13], [15] to facilitate this control, while providing some conveniences as well. A standard technique is to wrap OS memory allocation routines within specialized server routines, which allow the caller to "tag" (or "color") the memory they allocate, either explicitly or via some "current-tag" state. This interface makes it possible to deallocate all memory regions that have a particular tag[1]. Support for bulk deallocation on a per-tag granularity allows for quick, "sloppy" development of those portions of a system that are not memory-sensitive. For example, a query parser can be written without worrying about deallocating any memory; at the end of the parsing phase, the final parse-tree can be copied to memory allocated with a new tag, and memory from the old parser tag can be deallocated *en masse*. In addition to the convenience of bulk deallocation, commercial DBMS memory management services often exercise performance-oriented control over how memory is used, centralizing resource management logic and hence minimizing bugs and inefficiencies. For example, memory allocation per tag can be limited in application-specific ways – e.g., a particular query or module might be limited (statically or dynamically) in the amount of additional memory it can allocate.

In Java, memory management is handled by the JVM and the garbage collector. Dynamic allocation is done on the Java heap, and the garbage collector periodically searches for and deallocates unreferenced objects. Allocation is performed using the `new` construct and deallocation is performed implicitly by dropping all references to an object. This memory management interface is excellent for many situations. In fact, it helped us conveniently manage short-lived local data structures used by the catalog, queues, wrappers, modules, eddy, and the query parsing and translation engine. We cannot stress how useful garbage collection is for avoiding subtle memory leaks. However, this model simply lacks the control needed to do much of the work done by a high-throughput query processor. In particular, it prevents temporal and spatial control over memory management operations.

Temporal control over memory allocation provides the ability to invoke deallocation operations immediately. For example, when a hash join completes one partition during its probe phase, the memory it consumed can be immediately reused for the next partition. If the garbage collector postpones collection, the join will not be able to reuse the memory. Furthermore, it is possible that the next garbage collection phase may occur during a period of high CPU utilization. The collection process would slow down the ongoing computation, and it could further hamper it by destroying cache locality. In short, a garbage collector can reduce throughput if invoked at the wrong time. Java provides the `System.gc()` call for invoking the garbage collector; however, this call only provides a "best effort" guarantee, and may not affect the region we are interested in deallocating.

Spatial control over memory allocation provides the ability to control the amount of physical memory allocated, to control how it is partitioned, and to control memory operations on the individual partitions. A query processor needs to adjust the amount of memory available to each query and to each query processing module based on observed data and execution characteristics. The Java API does not allow clients to explicitly partition memory among modules, nor does it offer an analog to C's `sizeof` interface to determine how much physical memory is being utilized. Thus,

[1]The interested reader is referred to POSTGRES [4] an open-source example of a DBMS with this facility. Postgres' memory manager was used much more aggressively when the code was commercialized by Illustra Information Technologies.

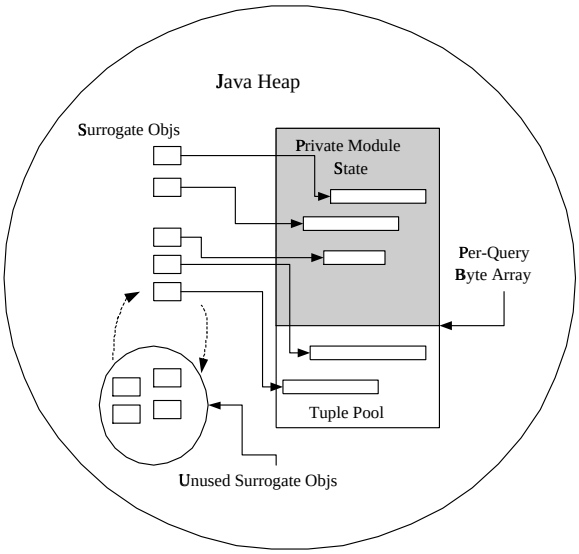

Figure 2: Overview of Telegraph memory implementation

a module or query cannot know how much memory it is using, and hence it can over-allocate memory, taking away resources from more deserving modules or queries. Deallocation of specific regions at various granularities is also useful for speeding up memory operations. Again, consider an out-of-core hash join, which allocates and deallocates main-memory hashtables the size of disk partitions. The only memory to be considered upon deallocation should be the hashtable partitions. A general purpose garbage collector would do much more work in searching and identifying unreferenced regions, thereby reducing throughput. The lack of control over memory management forced us to duplicate memory management functionality at user-level, as we describe next.

3.2 Telegraph Memory Management

To work around the lack of control offered by Java memory interfaces, we implemented our own management scheme. In this section, we outline our scheme and compare its performance against the Java garbage collector.

For each query, at startup, we allocate a large Java byte-array which is then divided among the eddy and the modules. A portion of this space, called the tuple pool, is allocated for in-flight tuples, and modules are allocated regions for their private state. For example, a hash join would receive a region for its partitions. The memory allocation assignments are statically specified[2] and parameterized by global variables much as in other DBMSes[16]. The memory regions can be subdivided into smaller regions, and deallocation operations can be performed on individual subregions, or on a region as a whole. The organization of memory regions for private module state is similar to the tuple pool, so we describe the tuple pool in greater detail.

The tuple pool is a subregion of a query's byte-array that holds the data for each tuple (see Figure 2). Each tuple contains, in addition to its data, meta-data for locating fields, and a pin count. The pin count is used for reference counting. When the pin count reaches zero, the tuple is "deallocated", i.e. its region is marked free for later reuse. Upon deallocation, a small fraction of the memory region is compacted to reduce fragmentation. A disadvantage of having to managing our own pin counts is that we are more prone to memory errors and memory leaks if we forget to pin and unpin.

Each tuple has associated with it a surrogate object. A surrogate object is a Java object allocated outside of the tuple pool that holds an offset to the tuple's location in the tuple pool. The surrogate object has several purposes. Its main purpose is to serve as the monitor object for latching the tuple when modifying its pin count[3] and reading its data. The latch is necessary because a tuple may be shared among modules in different threads. It is these surrogate objects that are passed to the modules in an eddy. Access to tuple data can only be accomplished through method calls on the surrogate object preventing accidental or intentional corruption to other memory regions. To prevent corruption, accessor methods perform explicit array bounds checks which are more expensive than the implicit bounds checks done by the JVM for Java arrays. We disable these explicit checks for trusted modules.

If Telegraph left the management of surrogate objects to Java, then Telegraph would still be susceptible to the vulnerabilities arising from the lack of temporal control. To alleviate this problem, we preallocate surrogate objects, place them in a "free pool", a collection of objects that can be reused, and recycle them as necessary.

To gauge the effectiveness of the Telegraph memory manager, we compared its performance against the Java garbage collector in a series of experiments. We executed the probe-phase of a Grace hash join in these experiments. In the TeleMM experiment, the Telegraph memory manager managed all the surrogate objects, the tuple pool, and the private state of modules. In the JavaGC experiment, the surrogate objects and Java byte-arrays for each tuple were allocated on the Java heap, leaving memory management entirely to the Java garbage collector. We used the default generational garbage collector provided with the Sun Java HotSpot Server JVM 1.3.0.

In these experiments, each partition of the building relation contained 100,000 tuples. There were a total of 50 partitions, and each tuple was 100 bytes. Thus, in the TeleMM experiment, processing each partition only required a memory region of 10MB for the tuples. Including the other miscellaneous memory space that Telegraph and the JVM need, the minimum Java heap

[2]Adaptive resource allocation in Telegraph is the subject of ongoing work in our group.

[3]The Java byte-code specification does not guarantee that its increment and decrement instructions are atomic.

size that allowed us to run our experiments was 45MB, a fixed overhead of about 35MB. Also, in these experiments, after processing each partition, the hashtable was explicitly deallocated, which entailed setting all entries to `null` in the JavaGC experiments and calling the deallocation methods in the TeleMM experiment. The Grace hash join was running in a single thread, and our machine has two CPUs, so the Java garbage collector thread ran in parallel with the join. In the TeleMM experiment, the memory management routines did their work in the same thread as the hash join, and the Java garbage collector sat idle on the second CPU.

Figure 3 compares the completion time for executing the entire probe phase in both of these scenarios. The top curves shows the completion times of the JavaGC experiments run with various Java heap sizes. The lower dashed line indicates the completion time of the TeleMM experiment with a Java heap size of 45 MB. The TeleMM experiment ran the same at larger heap sizes because the hash join did not take advantage of the additional space. At a heap size of 45MB, the JavaGC experiment took over three times longer to complete. At around 55MB, where the Java garbage collector had twice as much space to manage each partition, the JavaGC experiment took about twice as long as the TeleMM experiment. At large heap sizes, the performance of the JavaGC experiment approaches the performance of the TeleMM experiment.

We examined these experiments in further detail to understand when the Java garbage collector was doing its work. Figure 4 compares the performance of the probe phase for each partition of the join. Here, the Java heap size was set to 80MB in the JavaGC experiment, and 45MB in the TeleMM experiment. We see the TeleMM experiment performance was consistent since the join could control the memory operations. The Java generational garbage collector seems to have done its work at unpredictable moments. We reran the JavaGC experiment using the incremental garbage collector provided with the Sun JDK 1.3 release hoping it would display worse, but more consistent performance. However, this was not the case.

It is important to note that there are cases when the Java garbage collector outperforms the Telegraph memory manager. For example, when we ran a simple scan and filter query, using only the Java garbage collector for memory management, Telegraph processed on average 1×10^6 100-byte tuples per second. Using the Telegraph memory manager, Telegraph only processed on average 5×10^5 tuples per second. We speculate the difference is because all our memory management routines suffer the overhead of Java type and array-bounds checking, and our compaction algorithm is not as finely tuned as the garbage collector. It would be interesting if Java allowed us to deactivate these default security measures in the server for trusted, well-tested code. This would help us isolate the cause of the overhead, and help tune our memory management routines further.

3.3 Memory Management: Conclusions

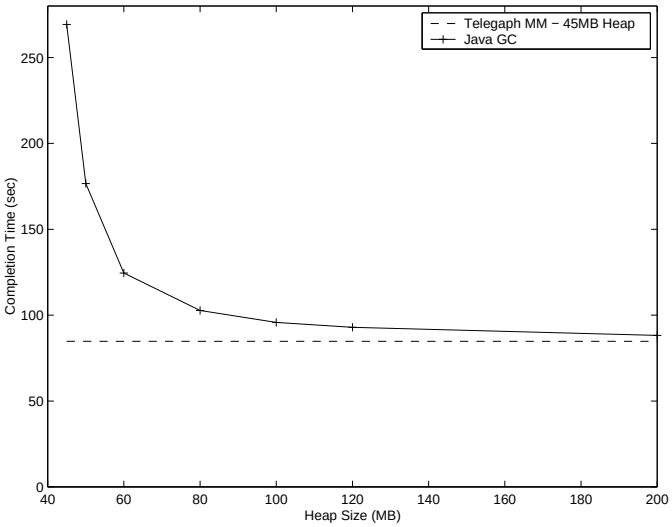

Figure 3: A comparison of the performance of the Java garbage collector versus the Telegraph memory manager. We ran the entire probe phase of a Grace hash join. The Java garbage collector achieves the performance of the Telegraph memory manager only at very large heap sizes.

Database query processing algorithms are designed around the careful and explicit management of memory resources. The performance of these algorithms can suffer enormously if they are unable to manage memory accurately. Although our experiments only represent a small sample of garbage collection algorithms for our workload, we believe that the lack of temporal and spatial control over memory management is a fundamental weakness of Java's transparent memory allocation model. We are skeptical of any particular JVM's implementation of garbage collection in its ability to accurately predict memory allocation and reuse patterns for a data-intensive query engine workload. However, we do appreciate the convenience of implicit memory management for a good deal of the "support code" in a query processor.

Hence it seems worthwhile for Java to provide garbage collection, enhanced with control interfaces similar to those found in traditional DBMS memory managers, including the timing and scope of garbage collection, and the exposure of accurate metrics of memory utilization. These features would allow sophisticated programmers to control the behavior of garbage collection, without sacrificing the other benefits of Java's memory management, such as protection and automatic reference counting.

4. MULTIPROGRAMMING

In this section, we discuss the multiprogramming issues in Java as they relate to the Telegraph architecture. In our opinion, threading and synchronization mechanisms are elegantly integrated into the Java programming language via its built-in threads, and its

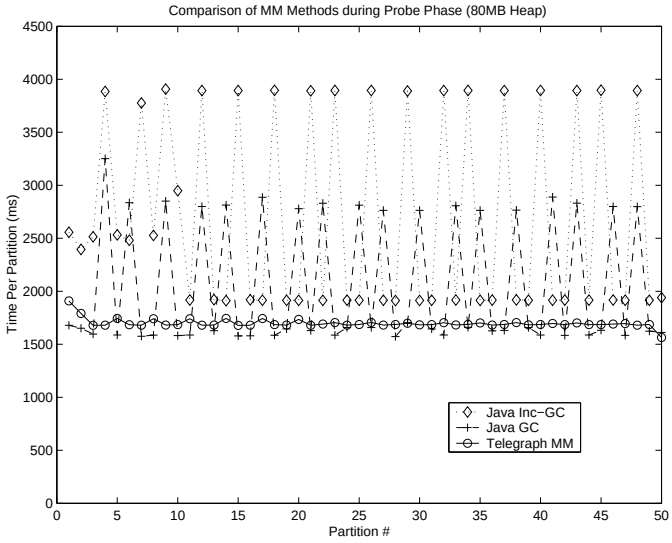

Figure 4: A comparison of several memory management alternatives: the Java incremental garbage collector, the default Java garbage collector, and the Telegraph memory manager. During the probe phase of a Grace hash join, the Telegraph memory manager provides consistent response times for each partition because we can control when and on which region it performs work.

monitor-style synchronization. While this does not necessarily make it easier to debug multi-threaded programs, it does make it simpler to write them. Furthermore, a multi-threaded program written in Java is portable to a variety of platforms. Given these positive aspects, we discuss the drawbacks of Java's synchronization mechanism, and the scalability limitations of the threads on our platform. These deficiencies forced us to make specific decisions for the Telegraph architecture which we discuss below.

4.1 Synchronization

Monitors[19] are the only synchronization primitive provided by Java. A monitor can be viewed as a semaphore and queue associated with an object or a class. A Java thread will try to acquire a monitor if it jumps to a `synchronized` method in the associated object or reaches a critical section enclosed in an `synchronized` block. If the thread fails to acquire the monitor, it is placed at the end of the queue and blocks until the monitor is available. Java provides method calls `notify()` and `notifyAll()` to explicitly wake up threads waiting for the monitor. As with the garbage collector, we found that while monitors are sufficient for most simple programs, they lack functionality, and are too heavy-weight for Telegraph.

There are three main disadvantages to monitors. First, there is no option for acquiring Java monitors *conditionally*, i.e. acquiring a monitor without blocking if it is already held. Second, monitors impose undue memory overheads. Third, uncontended acquires and releases can be expensive.

Table 1: Synchronization Overheads. ***Time to invoke synchronized and unsynchronized null-method call and array copy method call, averaged over 10000 calls.***

Method	Avg. Time Per Call (ms)
Unsynch. Null Function Call	2.2×10^{-5}
Synch. Null Function Call	1.4×10^{-4}
Unsynch. 10,000 El. Array Copy	.14
Synch. 10,000 El. Array Copy	.16

The Telegraph memory manager relies heavily on synchronization operations and would benefit from *conditional* semaphores. During compaction, it repeatedly acquires and releases monitors on tuples that it needs to copy to a new location. In this scenario, these monitors are used as short-duration semaphores associated with physical resources, thus serving the same function as *latches* [20], [21] in traditional database systems. If the compaction routine encounters a tuple that is already latched, then the compaction routine blocks. To avoid blocking compaction, we are careful to write modules which latch tuples for a short duration or copy the tuple to private space. Unfortunately, all modules with a handle to an object can acquire its associated monitor. So, untrusted modules can latch tuples and block compaction indefinitely as a form of denial-of-service attack (discussed further in Section 5). The ability to acquire a monitor conditionally would allow the compaction routine to skip the latched tuple and proceed. The new JDK 1.4 provides the `Thread.hasLock()` method to test if a particular thread already has acquired a monitor. However, using this for simulating conditional semaphores does not scale well, because we must call this method for each potentially conflicting thread. It is possible to build conditional latches on top of monitors (e.g. by using a shared boolean variable) but this requires all accesses to follow a new, agreed upon protocol.

Furthermore, in situations like above, a simple synchronization primitive like compare-and-swap on a single byte would suffice. Yet, Java forces the programmer to dynamically allocate an object on the heap because monitors are intimately tied with objects. On our Java platform, the overhead for an object is 12 bytes, thus each surrogate object, including its 4-byte offset variable, is 16 bytes, which is paid on a per-tuple basis. Such an overhead can place undue memory pressures if a large number of semaphores are required.

Finally, the synchronization operations in Java are relatively expensive. We ran two micro-benchmarks to test the overhead of `synchronized` methods. In Table 1, we see that for a null function call, the synchronization overhead is a factor of 5, and for a function call doing a 10,000 element array copy, a substantially more time-consuming operation, the synchronization overhead is 14 percent. To understand the impact of synchronization on our memory management rou-

tines, we ran a query with a simple scan operator which scanned data from a 10MB file. The scan produced tuples into the tuple pool, and an output operator deallocated tuples from the tuple pool. We ran the query using both synchronized and unsynchronized versions of our memory management routines. The operators were running in single thread, scheduled by the eddy, and there was no contention for the tuples. In this experiment, just the overhead of `synchronized` method calls in the Telegraph memory management routines doubled the execution time of scanning a 10MB file.

The memory management routines require only a simple synchronization primitive such as a test-and-set or compare-and-swap. Instead, they are forced to use monitors. Currently, to avoid these overheads, we are careful to write operators that can be run in the same thread, or we stage data to modules in different threads in properly synchronized memory regions. We are also investigating "lock-free" optimistic algorithms for memory management.

4.2 Threads

We have noticed that threads on the Linux platform do not scale well, and others have confirmed this observation [26], [18]. For example, an experiment reported in [26] shows that for a workload involving an 8KB read of a cached-file, our platform can only support 8 concurrent threads at maximum throughput, and at 64 concurrent threads the throughput falls precipitously. Thus, as were processes in Unix 20 years ago [23], threads are a precious resource on our platform. Since Java native threads are essentially Linux threads, the Telegraph implementation is similarly constrained. As a result we must be careful about how we write modules to work with an eddy.

In an eddy, to hide the latency of long-running operations like fetches from web-based sources, we require that modules work asynchronously from one another. Thus, modules written in the traditional iterator interface style must be placed in separate threads to prevent them from blocking the eddy and other modules. Since threads are a precious resource, and synchronization overheads are high in Java, this approach can seriously limit the number of concurrent queries Telegraph can run on a single machine.

A common solution to this limitation is to multiplex lightweight user-level threads on top of heavy-weight threads. However, since Java provides no mechanisms for implementing a context-switch, we were unable to build our own user-level thread library. Hence, we simulated user-level threads by writing our engine in an event-driven programming style [26], [22].

In this methodology, multiple threads of control are divided into actions which are triggered by events and multiplexed within a single thread. In our case, the eddy and modules share a thread, and the eddy explicitly schedules modules by calling their `process()` routines (i.e. actions) when they have some data (i.e. events) to process. Because logical threads of control are decomposed and intermixed in this programming style, we found both implementation and debugging to be more complicated than in traditional iterator-style programming. In the `process()` routines, we needed to carefully save the state of the computation by hand and restore it on subsequent calls. Moreover, we had to be careful about doing only a bounded amount of work for each call. In particular, these routines were not allowed to block on potentially long-running system calls, so we dispatched these calls to an external thread pool.

The need for non-blocking operations raises another important issue in Java concerning high-latency I/O requests. Java JDK 1.3 only provides blocking network I/O library calls. Although, we dispatch these requests to a pool of worker threads, achieving peak throughput from many web-based sources requires numerous simultaneous outstanding requests. For example, from Berkeley, we can achieve a peak throughput of 0.5 MB/s from the Federal Election Commission website with 30 simultaneous connections using custom non-blocking I/O libraries in Java [24]. We achieved a peak throughput of 5 MB/s from a popular search-engine using 200 simultaneous connections before they started dropping our requests. Devoting that many Java threads for non-blocking I/O is unreasonable considering the scalability limitations of threads on our platform. Others [24] also have observed the limitations of Java threads for implementing non-blocking I/O, and they provide custom implementations of low-level, non-blocking (NBIO) networking libraries in Java for Linux. Moreover, the recently released JDK 1.4 [2] provides low-level APIs for performing non-blocking I/O from the network and disk. However, the higher-level HTTP protocol routines that the Java core libraries provide remain blocking in JDK 1.4. The NBIO routines are also incompatible with these high-level routines. Reimplementing all the high-level networking library support in Java is a daunting task.

In fairness, other projects building data-intensive systems have not had such negative experiences with Java threads. The Mercator [14] project built a custom crawler in Java for data mining on the web. Using a custom JVM and proprietary operating system that can scale to 500 threads per machine, they were able to run their crawler full-tilt, saturating their incoming bandwidth with five machines.

Finally, since external user-defined operators cannot be trusted, Telegraph is forced to devote a thread to each. The Java threads API no longer supports the `stop()`, `suspend()`, `resume()` methods, and the `destroy()` method is currently unimplemented. This makes it difficult to kill runaway UDFs. In general, as noted in the JRes [9] project, Java provides weak CPU resource control and accounting mechanisms which permits denial-of-service by untrusted code.

4.3 Multiprogramming: Conclusions

The two multiprogramming issues we discussed in this section were the drawbacks of synchronization primitives in Java and the techniques we used to circumvent the scalability limitations of threads on Linux.

Synchronization primitives are essential for many tasks in database systems and for memory management in Telegraph. The correctness and performance

of these algorithms rely heavily on the primitives provided by the underlying platform. Java provides monitors which are convenient and easy to use, but offer limited functionality: they do not allow a conditional acquire. Furthermore, they have unreasonable memory management overheads. One suggestion would be for the Java language to incorporate an atomic operation like "test-and-set" from which database developers can develop more sophisticated synchronization primitives to suit their needs. An alternate middleground would be to provide a richer set of inexpensive acquisition modes for monitors.

Threads are a limited resource on the Linux platform, and the scalability of Java programs is constrained by the underlying operating system threads package. Thus, to conserve threads we had to write our engine in event-driven programming style [26], [22]. We had to work around Java's blocking network I/O interfaces by dispatching these requests to a pool of worker threads. While this approach was sufficient for the Telegraph prototype, we learned that achieving peak throughput from high-latency sources requires numerous outstanding connections. To manage these connections in a scalable way, we suggest that Java developers include non-blocking versions of all networking library calls, especially the high-level ones.

5. INTROSPECTION

Java provides an elegant type introspection mechanism (alternately called *reflection* in the Java nomenclature) with dynamic class loading, which makes installing new user-defined data sources, wrappers, and data types very easy.

Introspection combined with dynamic class loading is both powerful and dangerous. Programs can load class files, examine their methods, and create instances of them for use. This has allowed us to easily load and execute user code for wrappers, and it has allowed us to simplify catalog evolution. However, it also poses significant security risks in a distributed environment where users are allowed to add data types and wrapped sources, since it is difficult to guarantee that dynamically loaded code is not malicious or buggy.

5.1 User Defined Functions

Java introspection is an excellent general technique for dynamically loading and installing user defined functions. In the Telegraph project, we are concerned with allowing users to access a variety of non-traditional database sources such as web sites, sensors, and functions (UDFs) as though they are standard tables. We do this by implementing a number of different wrappers. The wrapper for a particular source is stored in the `wrapperid` field in the catalog. Wrappers are themselves stored in the catalog table `WRAPPERS`, which, for each wrapper, names the class file which implements the wrapper as well as some of its basic properties. We use the Java class loader to dynamically load the wrapper for each source, which can contain arbitrary Java code for parsing the raw data.

This flexibility is very nice, but it does introduce some significant security concerns, particularly if it is possible for users outside our database to add wrappers to the catalog. This is a realistic if one imagines external web-site developers providing wrappers for their sites to our deep-web query engine. The problem is that such wrappers can act maliciously against Telegraph. Unlike in a standard DBMS in which UDFs run in an entirely separate process, Java can dynamically load and efficiently *sandbox* UDFs within the same process to prevent them from executing potentially damaging functions or reading or writing local files [11]. However, it does not prevent UDFs from consuming huge amounts of memory or monopolizing the CPU. As discussed in [9], the proper solution to this problem is to build resource tracking and enforcement mechanisms into the JVM. Until such mechanisms are widely available, claims that Java's sandboxing facility provides adequate security for server-applications cannot be trusted. Resource tracking mechanisms are also important to the Eddy for determining how to dynamically modify the query execution plan.

5.2 Catalog Management

Like most database systems, Telegraph stores the meta-information describing wrapped sources and their properties in the catalog. To simplify catalog management, we chose to store the Telegraph catalog in XML format which is parsed according to a description file. This file, similar to a DTD, describes the schema of the catalog and includes the column types for catalog tables. At startup Telegraph uses the Java introspection facilities to dynamically create the in-memory catalog data structures and accessors based on the description file. It then fills in the catalog data from the XML catalog file. This technique allows Telegraph to startup from any version of the catalog without recompilation. Hence, catalog evolution is simplified because it is separated from the software base.

We illustrate our approach with an example. Consider adding a logo field which contains an image to associate with each source in the catalog. In C or C++, one would have to either modify the ".h" files containing the structure corresponding to the TABLES table, and add a pointer to an image structure. Then a recompilation would be necessary to allow the server to load image files. With our approach, we simply modify the description file to indicate that the TABLES table now contains a new field which stores the image datatype. In Java, with introspection, we can dynamically load the image class specified by the description file and instantiate an image object with the data stored in the catalog.

5.3 Introspection: Conclusions

The introspection and dynamic class loading facilities in Java have simplified two typically cumbersome tasks in traditional database systems: loading and executing UDFs, and catalog evolution. Resource consumption in Java, however, goes unchecked allowing external code to monopolize resources such as CPU or memory. To avoid such denial-of-service attacks, we concur with [9] that Java should provide resource tracking and enforcement mechanisms.

6. LIBRARIES AND TOOLS

6.1 Core Libraries

Java provides an extensive set of standard libraries that make development of substantial applications very easy. These include networking support, including high level protocols like HTTP, the Swing GUI development environment, and many standard data structures like hash tables and balanced trees. These libraries, combined with ease of memory management via the garbage collector, made it possible for us to build and deploy a functional web-service in about four months (twelve man-months) that could handle several thousand complex queries a day without serious problems. This development effort included both a client-side applet UI and the query engine itself. Although we were very satisfied with these APIs, there are a few areas where the standard libraries need to mature before Java is truly suitable for developing complex server and database systems like Telegraph.

6.1.1 Over-Synchronization

Most of the Java standard libraries are thread-safe, because the language designers wanted their data structures to be usable in multi-threaded programs. Because synchronization has a large overhead, newer versions of the Java libraries have introduced copycat versions of earlier data structures which are not thread-safe. For example, in the `java.util` package, the `HashMap` and `ArrayList` data structures are the non-thread-safe versions of the `Hashtable` and `Vector` classes. Unfortunately, it is not possible to tell from a class name whether it is thread-safe or not. Thus, initially, we used many thread-safe versions of library classes which impaired the performance of uncontended sections of Telegraph. We do not object to the inclusion of thread-safe libraries; however, we believe that it should be possible to easily identify a thread-safe or non-thread-safe version of any class rather than having to dig through the Java documentation to find it.

Researchers at Compaq have noted other instances of abusive over-synchronization [14]. For instance, the class `java.util.StringBuffer`, which is used for string concatenation, uses synchronization, even when strings are being allocated statically. This can have severe performance impacts on programs. After we first built Telegraph, profiling revealed that string allocation was the single most expensive operation we regularly performed. These were mainly due to debugging statements that implicitly created strings. As a workaround, we introduced a preprocessor in our build scripts that could remove debugging statements.

6.1.2 Abusive Memory Allocation

One problem with Java lies in its tendency to encourage object allocation. This is dangerous because memory allocation takes time and wastes heap space, and stresses the abilities of the garbage collector. As noted earlier, each Java object has 12 bytes of storage overhead on our platform. Many of the Java libraries are built in such a way as to encourage these overheads. For instance, the `Vector` and `Hashtable` classes, which one often wants to use to store primitive values like integers, can only store values which inherit from `Object`. This implies that whenever an object is inserted into one of these data structures, the programmer is required to allocate an additional 12 bytes of storage.

We have already mentioned the string concatenation operator, which creates a `StringBuffer` implicitly when it is used. This case is surprisingly dangerous because concatenation has the illusion of being a lightweight operator. We wrote a simple `for` loop, with a body $x =$ ``foo'' + i, where x is a string, and i is the loop index variable (an integer), and + implies string concatenation. This was an experiment to test the cost of string concatenation. Java engages in a remarkable amount of work to execute each pass through this loop. It allocates five separate objects, makes seven method calls, three of which are synchronized, and allocates more than 100 bytes of memory.

All of these objects must eventually be garbage collected. Its tempting to believe that the garbage collector is an all-powerful entity that makes memory allocation and deallocation free. However, as was shown in our previous discussion of memory management, garbage collection can take significant amounts of time, and object allocation consumes both time and valuable memory storage.

6.2 Tools Support

When we first started building Telegraph, there were very few good public domain development tools on the Linux platform for debugging, profiling, and optimizing Java byte-code. The default Java debugger, `jdb`, was itself buggy, and the profiling feature in the HotSpot JVM 1.3.0 was broken. We relied heavily on Java features like mandatory exception handling and human-readable stack traces to perform a good deal of our debugging. We used the standard `System.out` and `System.err` for the rest of our debugging. We also achieved some performance optimizations by profiling Telegraph using previous versions of the JVM.

One reason for the lack of sophisticated tools was that the Sun JDK 1.3 was just recently released and there was not much support for it. Over time, we began to learn of commercial products like VTune [5], OptimizeIT [3], and JInsight [1] for debugging the performance of Java programs. We also learned of tools like JAX [1] that optimize the byte-code of Java programs for both size and efficiency. As Java matures and becomes an integral part of the existing software code-base in industry, we expect that the development support for the language will be more widespread and robust.

6.3 Libraries and Tools: Conclusions

Java library support is extensive and the core libraries were instrumental in helping us quickly build the Telegraph prototype. However, there are a number of dangers associated with these libraries. If used indiscriminately, it is easy to write code with overheads of excessive synchronization and object allocation. We suggest that the Java language designers

make an effort to build APIs that clearly reflect the synchronization overhead, and provide libraries that avoid excessive object allocation. Moreover, the support for development tools in Java was limited when we began building our prototype; thus, we relied on Java features like exception handling and stack traces for debugging.

7. OTHER LIMITATIONS FOR A DBMS

Our initial Telegraph prototype is akin to the query processing engine in a traditional database system. Our experiences building the Telegraph prototype have brought to light many deficiencies of Java in its support for database development. However, there are a number of pieces of traditional database systems that we did not implement in Telegraph which could possibly stress the Java interfaces further. In this section, we speculate on the components of a traditional database system in which the Java interfaces could be a limiting factor and briefly describe why.

The Java interfaces provide no method for accounting for consumed memory, and provide no spatial and temporal control. Databases typically have large allocated regions like the buffer pool and sort buffers for which they need tight control over the amount of memory consumed. Many of the problems that arise with limited spatial and temporal are commonly due to inefficient reuse of memory. The buffer pool, log tail, sort buffers, and latch request blocks are common data structures in which memory is immediately reused. Thus, the Java garbage collector would perform poorly for managing these structures.

Most database systems need to acquire cheap, short-duration semaphores, *latches*, associated with a physical resource. For example, latches associated with buffer pool pages are typically used in a variety of places: index concurrency control algorithms, the background process which writes dirty pages to disk, etc. Latches gain most of their performance benefits because they are statically allocated [20]. However, in Java all synchronization operations are associated with Java objects, which must be dynamically allocated.

Threads are a limited resource on out platform, and the Java threads API provides no control over killing threads. The `Thread.destroy()` routine is not implemented, and `Thread.stop()` has been deprecated in the Java API[2], leaving the responsibility of writing well-behaved threads to the programmer. Database servers implemented in a thread-per-transaction model need the ability to kill threads for aborting transactions, particularly during deadlock which often cannot be anticipated or avoided. Moreover, database systems rely on a number of background processes such as the log flusher, deadlock detector, etc., which all consume threads, and thus limit the number of available threads for supporting multiple users.

Finally, until the recently released JDK 1.4, the Java core libraries provided no interface for forcing individual pages of a file to disk. Database write-ahead logging recovery algorithms [20] rely heavily on this functionality. The *newio* libraries in JDK 1.4 provide an API for mapping files to memory regions, flushing those regions back to disk, and issuing asynchronous requests. Still, there is no unmap facility, so a mapped region cannot be immediately reused, which could lead to inefficiencies like those exhibited in our join experiment.

8. RELATED WORK

There is a plethora of work in the operating systems on building better interfaces to suit database development, and much work in the database community in identifying the underlying services and mechanisms required for building database systems. This literature is vast, and we refer the interested reader to [23], [12], [17], [8] as a starting point. Instead we focus on recent work on the convergence of databases and Java, and recent work on building scalable, server-side, I/O intensive software in Java. We survey the work that is most relevant to our discussion.

Cloudscape [27] is a light-weight database built in Java. They claim one significant advantage of their implementation is that external Java code can run more efficiently within their database kernel than with other database systems.

The Predator [11] project evaluated alternatives for integrating Java UDFs in traditional database systems. Their conclusion was that Java UDFs are viable in terms of performance; however, there are other issues like threading and memory management that make integration difficult.

JRes is a resource management mechanism for tracking resource consumption and enforcing usage constraints in Java. In [9], they used JRes to obtain feedback on UDFs for dynamic optimization.

Mercator is a project that aims to build infrastructure for doing "web archaeology". Their parallel crawler is written entirely in Java. In [14] they report on some similar problems to the ones we experienced with Java's core libraries, including excessive synchronization and heap allocation.

The Berkeley Jaguar [25] system is a JVM enhancement that provides infrastructure for fast communication of Java objects in a cluster of workstations. This scheme avoids the overhead of marshaling Java objects to and from C code through the JNI interface.

The Ninja project is investigating how to build scalable internet services. Their code base is entirely in Java. In SEDA, [26], a component of Ninja, they are investigating structured event-driven programming methodologies, analogous to the scheme mentioned here, for circumventing the scalability problems of threads.

9. CONCLUSION

We have completed an initial prototype of the Telegraph adaptive dataflow system on the Java platform. As a guide for the reader, in Table 2 we highlight the main benefits and drawbacks of Java as they were presented in each section of the paper.

Java has significant strengths in ease of programming, and in some of its convenient system services. We particularly benefited from its language features

for exception handling and implicit memory deallocation – these are invaluable for reducing the kinds of subtle bugs and memory leaks common in traditional systems software development. Java's introspection facility allowed us to elegantly load user-code and support catalog evolution. As expected, Java's security features were helpful in running user code fairly safely.

In many cases, however, Java provides a less-than-satisfactory environment for writing high-performance, I/O-intensive code. Similar to the problems cited in early operating systems [23], Java hides too many resource management decisions behind its narrow interfaces. In our case, this meant that either our code could not control resources in the way we wanted it to, or we had to awkwardly program around Java's interfaces to get control. Java's memory management interfaces do not provide the spatial or temporal control essential for efficient query processing, nor do they provide the ability to easily measure and control memory utilization. Java's sole support for synchronization is monitors, which are both overly heavyweight and insufficiently function-rich to support our latching needs. Java's I/O libraries lack the non-blocking APIs necessary to conveniently hide network latencies. In all of these cases, we had to replicate functionality at the user level or code around these deficiencies to build a high-throughput concurrent system.

Some of our difficulties were less fundamental, and can be attributed to immature implementations which we expect to be fixed over time. On our (very standard) Sun JVM and Linux OS, multi-threading is not very scalable, which forced us into some unusual coding styles. Debugging and profiling support is not robust, which often resulted in painful development. A hodgepodge of tools for code optimization are available, and different development environments provide very different tools and features. Java's built-in libraries do a poor job of separating expensive thread-safe features from cheap but unsafe features.

Many of our more critical problems with Java are likely to recur in any large systems project, unless changes are made to the language. A number of resource management issues arose from poor interfaces, and hence will recur regardless of how good new Java implementations are – i.e. they will arise independent of the JVM and OS. Moreover, most of the basic systems issues we raised are quite universal (memory management, threading, synchronization), and Java's problems will adversely affect any I/O-intensive server.

In sum, our experience building the Telegraph prototype in Java was mixed. Java presents some problems for database system developers that are reminiscent of those from early operating systems. We proposed a number of minor but important changes to Java – particularly with regard to memory management and synchronization – which would alleviate the worst of these problems. Despite our criticisms, we were able to code around most of the problems we found, with significant effort but tolerable losses in performance. We believe that Java is on track to becoming a viable server development and deployment environment, with the potential to provide system developers with significant benefits over the current state of the art. We encourage further communication among system builders and Java designers to speed progress along this track.

10. ACKNOWLEDGEMENTS

We would like to thank all the people that helped build and deploy the initial Telegraph prototype: Vijayshankar Raman, Nick Lanham, Sirish Chandrasekaran, Amol Deshpande, Mohana Lakhamraju. We thank Fred Reiss for stimulating discussions and comments on this paper. We also want to thank Matt Welsh and Marcel Kornacker.

11. REFERENCES

[1] IBM alphaWorks Home Page. *http://www.alphaworks.ibm.com/*, 2001.

[2] Java 2 Platform API Specification, JDK 1.4. *http://java.sun.com/j2se/1.4/docs*, 2001.

[3] OptimizeIt Home Page. *http://www.optimizeit.com/*, 2001.

[4] PostgreSQL. *http://www.postgresql.com/*, 2001.

[5] VTune Tool Home Page. *http://developer.intel.com/vtune/analyzer/*, 2001.

[6] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

[7] M. K. Bergman. The Deep Web: Surfacing Hidden Value, White Paper. *http://www.brightplanet.com/deepcontent/index.asp/*, September 2001.

[8] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. *SOSP*, 1995.

[9] G. Czajkowski, T. Mayr, P. Seshadri, and T. von Eicken. Resource Control for Java Database Extensions. *5th USENIX Conference on Object-Oriented Technologies and Systems*, May 1999.

[10] O. Doederlein. The Java Performance Report - Part III. *http://www.javalobby.org/fr/-html/frm/javalobby/features/-jpr/part3.html*, September 2000.

[11] M. Godfrey, T. Mayr, P. Seshadri, and T. von Eicken. Secure and Portable Database Extensibility. *SIGMOD 1998*, pages 390–401, 1998.

[12] J. Gray and A. Reuter. *Transaction Processing – Concepts and Techniques*. Kaufmann, 1993.

[13] Hamilton, James. *Personal Communication*. Feb 2001.

[14] A. Heydon and M. Najork. Performance Limitations of the Java Core Libraries. *ACM Java Grande*, June 1999.

[15] Hong, Wei. *Personal Communication*. Feb 2001.

[16] IBM DB2. IBM DB2 Reference Manual. Version 6.1.

Table 2: Summary of experiences with Java

Benefits	Drawbacks	Proposals
Garbage collection useful for memory-insensitive code.	Garbage collection offers poor spatial and temporal control over memory allocation for memory-*intensive* code.	Provide "current-tag" memory setting, and `sizeof` features. Support modes where the timing and tag-scope of garbage collection are controllable.
Threading and synchronization integrated into the programming language.	Synchronization is expensive and monitors provide limited functionality. Threads are not scalable. No high-level, non-blocking networking API.	Provide an inexpensive low-level mutex primitive. Provide more scalable threads. Provide high-level, non-blocking networking API.
Extensive support in core libraries for GUI, networking, disk, data structures, etc. Mandatory exception handling eases debugging.	Core libraries promote abusive memory allocation and over synchronization.	Provide two well-named versions of each core library, one of which is thread-safe, one of which is not.
Introspection, dynamic class loading, and security manager simplify loading and execution of UDFs.	No resource consumption tracking and enforcement.	Useful proposals given in [9].

[17] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. *SOSP*, 1997.

[18] D. Kegel. The C10K Problem. *http://www.kegel.com/c10k.html*, 2001.

[19] B. Lampson and D. Redell. Experiences with Processes and Monitors in Mesa. *CACM*, 1980.

[20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *TODS*, pages 94–162, 1992.

[21] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. *SIGMOD*, pages 371–380, 1992.

[22] D. C. Schmidt. Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events. *Pattern Languages of Programs Conference*, August 1994.

[23] M. Stonebraker. Operating System Support for Database Management. *CACM*, 24(7):412–418, 1981.

[24] M. Welsh. Non-blocking I/O for Java. *http://www.cs.berkeley.edu/ mdw/proj/java-nbio*, September 2001.

[25] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency Practice and Experience*, Dec 1999.

[26] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SOSP-18*, October 2001.

[27] N. Wyatt and A. Carlson. Cloudscape 3.6, A Technical Overview. *A Cloudscape White paper*, March 2001. see http://www.cloudscape.com/products/whitepapers.jsp.