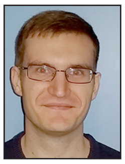
Don't Get Caught in the Cold, Warm Up Your JVM

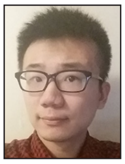
Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems

DAVID LION, ADRIAN CHIU, HAILONG SUN, XIN ZHUANG, NIKOLA GRCEVSKI, AND DING YUAN


David Lion is a graduate student in the Electrical and Computer Engineering Department of the University of Toronto. His research interest is in software systems and their performance. david.lion@mail.utoronto.ca

Adrian Chiu is an undergraduate student in Electrical Engineering at the University of Toronto. His research interests are in operating systems, distributed systems, and compilers. adrian.chiu@mail.utoronto.ca

Hailong Sun is an Associate Professor in the School of Computer Science and Engineering at Beihang University. His research interests include distributed systems, software engineering, and crowdsourcing. sunhl@ece.utoronto.ca


Xin Zhuang is a graduate student at the University of Toronto, studying computer engineering. His research interest is in software systems. xin.zhuang@mail.utoronto.ca

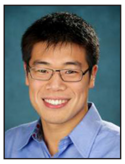
Many widely used, latency sensitive, data-parallel distributed systems, such as HDFS, Hive, and Spark choose to use the Java Virtual Machine (JVM) despite debate on the overhead of doing so. By thoroughly studying the JVM performance overhead in the above-mentioned systems, we found that the warm-up overhead, i.e., class loading and interpretation of bytecode, is frequently the bottleneck. For example, even an I/O intensive, 1 GB read on HDFS spends 33% of its execution time in JVM warm-up, and Spark queries spend an average of 21 seconds in warm-up. The findings on JVM warm-up overhead reveal a contradiction between the principle of parallelization, i.e., speeding up long-running jobs by parallelizing them into short tasks, and amortizing JVM warm-up overhead through long tasks. We therefore developed HotTub, a new JVM that reuses a pool of already warm JVMs across multiple applications. The speed-up is significant: for example, using HotTub results in up to 1.8x speed-ups for Spark queries, despite not adhering to the JVM specification in edge cases.

The performance of data-parallel distributed systems has been heavily studied in the past decade, and numerous improvements have been made to the performance of these systems. A recent trend is to further process latency sensitive, interactive queries with these systems. However, there is a lack of understanding of the JVM's performance implications in these workloads. Consequently, almost every discussion on the implications of the JVM's performance results in heated debate. For example, the developers of Hypertable, an in-memory key-value store, use C++ because they believe that the JVM is inherently slow. They also think that Java is acceptable for Hadoop because "the bulk of the work performed is I/O" [4]. In addition, many believe that as long as the system "scales," i.e., parallelizes long jobs into short ones, the overhead of the JVM is not concerning [7].

Our research asks a simple question: what is the performance overhead introduced by the JVM in latency sensitive data-parallel systems? We answer this by presenting a thorough analysis of the JVM's performance behavior when running systems including HDFS, Hive on Tez, and Spark. We had to carefully instrument the JVM and these applications to understand their performance.

Surprisingly, after multiple iterations of instrumentation, we found that JVM warm-up time, i.e., time spent in class loading and interpreting bytecode, is a recurring overhead. Specifically, we made the following three major findings. First, JVM warm-up overhead is significant even in I/O intensive workloads. For example, reading a 1 GB file on HDFS from a hard drive requires JVM to spend 33% of its time in warm-up. In addition, the warm-up time does not scale but, instead, remains nearly constant. For example, the warm-up time in Spark queries remains at 21 seconds regardless of the workload scale factor, thus affecting short-running jobs more. The broader implication is the following:

Nikola is the VP of Engineering at Vena Solutions Inc. Prior to this he worked at the IBM Compiler Group for 12 years, most notably as a Technical Lead for the x86 JIT Optimizer and Code Generator. He holds a master's degree in computer engineering from the University of St. Cyril and Methodius in Skopje, Macedonia. grcevski@gmail.com



Ding Yuan is an Assistant Professor in the Electrical and Computer Engineering Department of the University of Toronto. He works in computer systems, with a focus on their reliability and performance. yuan@ece.toronto.edu

There is a contradiction between the principle of parallelization, i.e., speeding up long-running jobs by parallelizing them into short tasks, and amortizing JVM warm-up overhead through long tasks.

Finally, the use of complex software stacks aggravates warm-up overhead. A Spark client loads 19,066 classes executing a query, which is three times more than Hive despite Spark's overall latency being shorter. These classes come from a variety of software components needed by Spark. In practice, applications using more classes also use more unique methods, which are initially interpreted. This results in increased interpretation time.

To solve the problem, our key observation is that the homogeneity of parallel data-processing jobs enables a significant reuse rate of warm data, i.e., loaded classes and compiled code, when shared across different jobs. Accordingly, we designed HotTub, a new drop-in replacement JVM that transparently eliminates warm-up overhead by reusing JVMs from prior runs. The source code of HotTub and our JVM instrumentations are available at https://github.com/dsrg-uoft/hottub.

Analysis of JVM Warm-up Overhead

What follows is an in-depth analysis of the JVM warm-up overhead in three data-parallel systems, namely HDFS, Hive running on Tez and YARN, and Spark SQL running with Spark. We will show that on each system the JVM warm-up time stays relatively constant. The HDFS experiment further shows how warm-up can dwarf I/O, while the Spark and Hive experiments explain the implications of warm-up overhead for parallel computing. All experiments are performed on an in-house cluster with 10 servers connected via 10 Gbps interconnect. Each of them has at least 128 GB DDR4 RAM and two 7,200 RPM hard drives. The server components are long running and fully warmed-up for weeks and have serviced thousands of trial runs before measurement runs. Details on our study methodology and the JVM instrumentation can be found in our OSDI paper [5].

HDFS

We implement three different HDFS clients: sequential read; parallel read, with 16 threads, that runs on a server with 16 cores; and sequential write. We flush the OS buffer cache on all nodes before each measurement to ensure the workload is I/O bound. Note that interpreter time does not include I/O time, because I/O is always performed by native libraries.

Figure 1 shows the class loading and interpreter time under different workloads. The average class loading times are 1.05, 1.55, and 2.21 seconds for sequential read, parallel read, and sequential write, respectively, while their average interpreter times are 0.74, 0.71, and 0.92 seconds. The warm-up time does not change significantly with different data sizes. The reason that HDFS write takes the JVM longer to warm up is that it exercises a more complicated control path and requires more classes. Parallel read spends less time in the interpreter than sequential read because its parallelism allows the JVM to identify the "hot spot" faster.

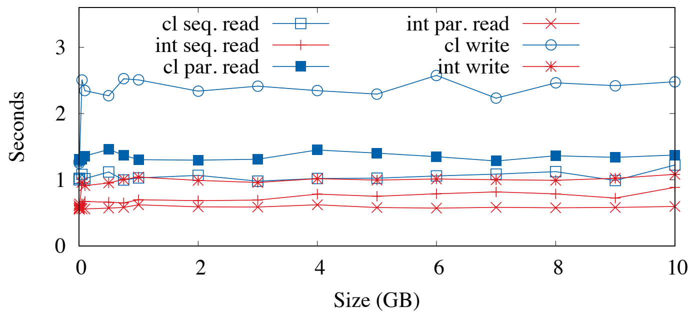


Figure 1: JVM warm-up time in various HDFS workloads. "cl" and "int" represent class loading and interpretation time, respectively. The x-axis shows the input file size.

Figure 2 further shows the significance of warm-up overhead within the entire job. Short-running jobs are affected the most. When the data size is under 1 GB, warm-up overhead accounts for more than 33%, 48%, and 30%, respectively, of the client's total execution time in sequential read, parallel read, and sequential write. According to a study [8] published by Cloudera, a vast majority of the real-world Hadoop workloads read and write less than 1 GB per-job as they parallelize a big job into

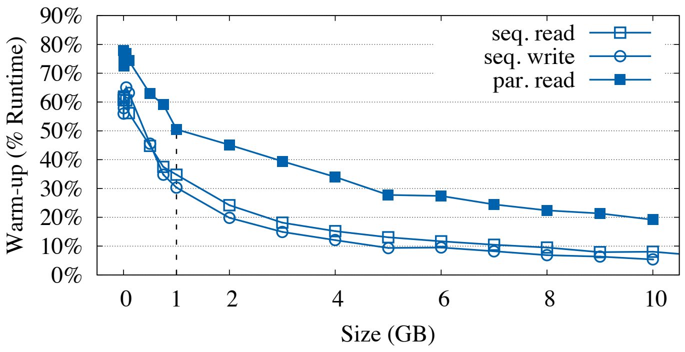


Figure 2: The JVM warm-up overhead in HDFS workloads measured as the percentage of overall job completion time

smaller ones. The study further shows that for some customers, over 60% of their jobs read less than 1 MB from HDFS, whereas a 1 MB HDFS sequential read spends over 60% of its time in warm-up.

Next we break down class loading and interpreter time using the 1 GB sequential read as an example. Figure 3 shows the warm-up time in the entire client read. A majority of the class loading and interpreter execution occurs before a client contacts a datanode to start reading.

Further drilling down, Figure 4 shows how warm-up time dwarfs the datanode's file I/O time. When the datanode first receives the read request, it sends a 13-byte ACK to the client, and immediately proceeds to send data packets of 64 KB using the `sendfile` system call. The first `sendfile` takes noticeably longer than subsequent ones since the data is read from the hard drive. However, the client takes even longer (15 ms) to process the ACK because it is bottlenecked by warm-up time. By the time the client finishes parsing the ACK, the datanode has already sent 11 data packets, and thus the I/O time is not even on the critical path. The client takes another 26 ms to read the first packet, where it again spends a majority of the time loading classes and interpreting the computation of the CRC checksum. By the time the client finishes processing the first three packets, the datanode has already sent 109 packets. In fact, the datanode is so fast that the Linux kernel buffer becomes full after the 38th packet and has to block for 14 ms so that the kernel can adaptively increase its buffer size. The client, on the other hand, is trying to catch up the entire time.

Figure 4 also shows the performance discrepancy between interpreter and compiled code. Interpreter takes 15 ms to compute the CRC checksum of the first packet, whereas compiled code only takes 65 μs per-packet.

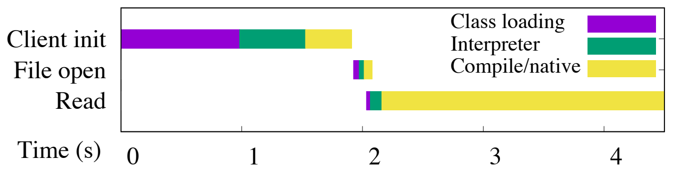


Figure 3: Breakdown of sequential HDFS read of 1 GB file

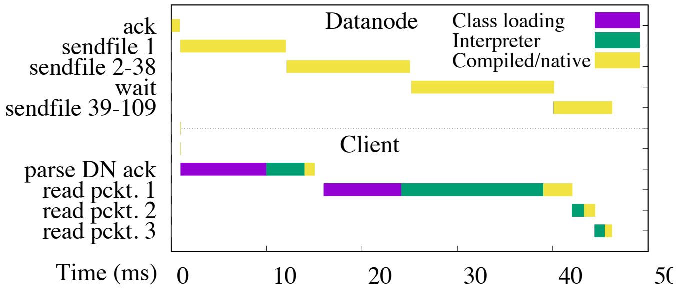


Figure 4: Breakdown of the processing of data packets by client and datanode

Break Down Class Loading

The HDFS sequential read takes a total of 1,028 ms to load 2,001 classes. Table 1 shows the breakdown of class loading time. Reading the class files from the hard drive only takes 170 ms. Because Java loads classes on demand, loading 2,001 classes is broken into many small reads: e.g., 276 ms are spent searching for classes on the classpath, which is a list of file-system locations. The JVM specification requires the JVM to load the first class that appears in the classpath in the case of multiple classes with identical names. Therefore it has to search the classpath linearly when loading a class. Another 411 ms are spent in define class, where the JVM parses a class from file into an in-memory data structure.

	Read	Search	Define	Other	Total
Time (ms)	170	276	411	171	1,028

Table 1: Breakdown of class loading time

Spark versus Hive

Figure 5 shows the JVM overhead on Spark and Hive. Surprisingly, *each query spends an average of 21.0 and 12.6 seconds in warm-up time on Spark and Hive, respectively.* Similar to HDFS, the warm-up time in both systems does not vary significantly when data size changes, indicating that its overhead becomes more significant in well parallelized short-running jobs. For example, 32% of the Spark query time on 100 GB data size is on warm-up. In practice, many analytics workloads are short running. For example, 90% of Facebook's analytics jobs have under 100 GB input size [1, 2], and a majority of the real-world Hadoop workloads read and write less than 1 GB per-task [8].

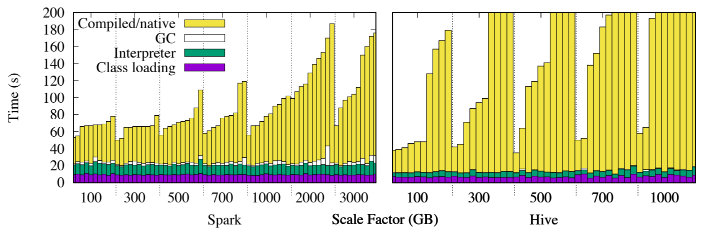


Figure 5: JVM overhead on BigBench. Overhead breakdown of queries from BigBench [3] across different scale factors. Only the 10 shortest queries from BigBench are analyzed because of our focus on latency-sensitive queries. The scale factor corresponds to the size of input data in GB. The queries are first grouped by scale factor and then ordered by runtime. Note that Hive has a larger query time compared to Spark.

Software Layers Aggravate Warm-up Overhead

The difference in the warm-up times between Spark and Hive is explained by the difference in number of loaded classes. The Spark client loads an average of 19,066 classes, compared with Hive client's 5,855. Consequently, the Spark client takes 6.3 seconds in class loading whereas the Hive client spends 3.7 seconds. A majority of the classes loaded by Spark client come from 10 third-party libraries, including Hadoop (3,088 classes), Scala (2,328 classes), and Derby (1,110 classes). Only 3,329 of the loaded classes are from Spark packaged classes.

A large number of loaded classes also results in a large interpreter time. The more classes being loaded, the greater the number of different methods that are invoked, where each method has to be interpreted at the beginning. On average, a Spark client invokes 242,291 unique methods, where 91% of them were never compiled by JIT-compiler. In comparison, a Hive client only invokes 113,944 unique methods, while 96% of them were never JIT-compiled.

Breaking Down Spark's Warm-up Time

We further drill down into one query (query 13 of BigBench with scale factor 100) to understand the long warm-up time of Spark. While different queries exhibit different overall behaviors and different runtimes, the pattern of JVM warm-up overhead is similar, as evidenced by the stable warm-up time. Figure 6 shows the breakdown of this query. The query completion time is 68 seconds: 24.6 seconds are spent on warm-up overhead of which 12.4 seconds are spent on the client while the other 12.2 seconds come from the executors. Note that a majority of executors' class-loading time is not on the critical path: executors are started immediately after the query is submitted, which allows executors' class loading time to be overlapped with the client's warm-up time. However, at the beginning of each stage the executor still suffers from significant warm-up overhead that comes primarily from interpreter time.

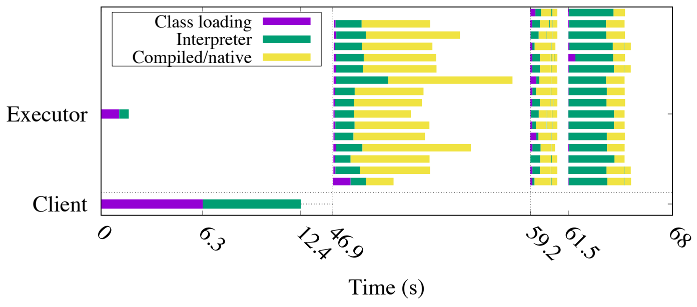


Figure 6: Breakdown of Spark's execution of query 13. It only shows one executor (there are a total of 10 executors, one per host). Each horizontal row represents a thread. The executor uses multiple threads to process this query. Each thread is used to process three tasks from three different stages.

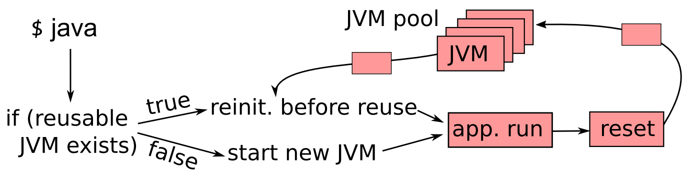


Figure 7: Architecture of HotTub

Hive

Hive parallelizes a query using different JVM processes, known as containers, whereas each container uses only one computation thread. Therefore within each container the warm-up overhead has a similar pattern to the HDFS client shown earlier. Hive and Tez also reuse containers to process tasks of the same query, and therefore the JVM warm-up overhead can be amortized across the lifetime of a query.

HotTub

The design goal for HotTub is to allow applications to share the "warm" data, i.e., loaded classes and compiled code, thus eliminating the warm-up overhead from their executions. HotTub is implemented by modifying OpenJDK's HotSpot JVM and is made to be a drop-in replacement. Users simply replace `java` with HotTub and run their Java application with normal commands.

Figure 7 shows the architecture of HotTub. When `java` is first called there are no existing JVMs to reuse, so a new JVM must be created for the application to run on as it normally would. Once the application finishes, the JVM must first be reset before it can be added to a pool of JVMs for later reuse. When there are JVMs in the pool, a call to `java` will attempt to find a valid JVM for reuse. If a JVM is found it will be reinitialized, and then the application will run on the already warm JVM with nearly zero warm-up overhead.

The main challenge of this process is to ensure that the application's execution on HotTub is *consistent* with the execution on an unmodified JVM. Next we discuss some techniques HotTub uses to ensure consistency. More detailed discussions can be found in our OSDI paper [5].

Class Consistency

When choosing a JVM to reuse we must make sure any class that will be reused is the same as the class that would have been dynamically loaded in a normal execution. To do this HotTub ensures for a JVM the classpath and classes on the classpath are the same for both the new application and the previously run applications. This also ensures there is a large amount of potential overlap between the new application and the already loaded classes and compiled code. It is possible to be more strict and only reuse a JVM if the application is more similar to what was previously run, but being less strict would not be able to guarantee consistency.

Data Consistency

At the reset phase all stale data is cleaned up off of the critical path before the JVM is put back in the pool. All application threads are cleaned up, so there are no more stacks left, and all file descriptors opened by the application are closed. HotTub also zeroes out all static data from classes. HotTub now runs garbage collection to remove all the stale data, but since there are no root references from the stack at this point, and roots from static data are all zero, practically all heap data is dead and collected quickly.

Once a JVM has been chosen to be reused it will perform the reinitialization phase, which sets the new file descriptors and runs the class initialization code of all loaded classes to correctly initialize the static data since it had been previously set to zero. The order this is done in is important because dependencies between classes can exist. HotTub maintains the correct order by recording the order of class initializations when they are first initialized and replaying the initializations in the same order before each reuse. There are some limitations to reinitializing static data, since known bad practices such as class dependence cycles and real time static initialization dependencies will cause HotTub to be inconsistent. However, these cases are extremely uncommon in practice.

Handling Signals and Explicit Exit

HotTub has to handle signals such as SIGTERM and SIGINT and explicit exit by the application, otherwise it will lose the target server process from our pool. If the application registers its own signal handler, HotTub forwards the signal. If SIGKILL is used or the application exists through a native library, the JVM will die and cannot be reused.

Privacy Limitation

The use of HotTub raises privacy concerns. HotTub limits reuse to the same Linux user, as cross-user reuse allows a different user to execute code with the privileges of the first user. However, our design still violates the principle "base the protection mechanisms on permission rather than exclusion" [6]. Although we carefully clear and reset data from the prior run, an attacker could still reconstruct the partial execution path of the prior run via timing channel since previously loaded classes and JIT-compiled methods can be seen.

Workload	Completion Time (s) Unmod.	Completion Time (s) HotTub	Speed-up
HDFS read 1 MB	2.29	0.08	30.08x
HDFS read 10 MB	2.65	0.14	18.04x
HDFS read 100 MB	2.33	0.41	5.71x
HDFS read 1 GB	7.08	4.26	1.66x
Spark 100 GB best	65.2	36.2	1.80x
Spark 100 GB median	57.8	35.2	1.64x
Spark 100 GB worst	74.8	54.4	1.36x
Spark 3 TB best	66.4	41.4	1.60x
Spark 3 TB median	98.4	73.6	1.34x
Spark 3 TB worst	381.2	330.0	1.16x
Hive 100 GB best	29.0	16.2	1.79x
Hive 100 GB median	38.4	25.0	1.54x
Hive 100 GB worst	206.6	188.4	1.10x

Table 2: Performance improvements by comparing the job completion time of an unmodified JVM and HotTub. For Spark and Hive we report the average times of the queries with the, best, median, and worst speed-up for each data size. Speed-up values were calculated using full-precision values, not the rounded values shown as completion times in this table.

Performance of HotTub

We conduct a variety of experiments on HotTub in the same manner as our JVM warm-up performance analysis to evaluate its performance. Table 2 shows HotTub's speed-up compared with an unmodified HotSpot JVM. We ran the same workload five times on an unmodified JVM and six times on HotTub. We compared the average runtime of the five unmodified runs with the average runtime of the five reuse HotTub runs, excluding the initial warm-up run. For Spark and Hive, we ran the same 10 queries that we used in our study.

The results show that HotTub significantly speeds up the total execution time. For example, HotTub reduces the average job completion time of the Spark query with the highest speed-up by 29 seconds on 100 GB data, and can speed up HDFS 1 MB read by a factor of 30.08. Among nearly 200 pairs of trials, a job running in a reused HotTub JVM always completed faster than an unmodified JVM. Enabling our performance counters, we observe that indeed HotTub eliminates the warm-up overhead. In all the experiments, the server JVM spends less than 1% of the execution time in class loading and interpreter.

In addition to evaluating the speed-up of HotTub in our paper, we evaluated many other aspects. We also found that the majority of speed-up comes in the first reuse run. When inspecting hardware performance counters we saw a large reduction in memory accesses due to avoidance of class loading and interpretation. We found that when reusing JVMs that were warmed up with a different query than the one being run, HotTub still achieved similar speed-ups since different jobs still tend to use similar framework code in these systems. Also, the management overhead of HotTub turned out to be low, only adding a few hundred milliseconds to the critical path.

Conclusion

We started this project curious to understand the JVM's overhead on data-parallel systems, driven by the observation that systems software is increasingly built on top of it. Enabled by non-trivial JVM instrumentations, we observed the warm-up overhead and were surprised by the extent of the problem. We then pivoted our focus on to the warm-up overhead by first presenting an in-depth analysis on three real-world systems. Our results show the warm-up overhead is significant, bottlenecks even I/O intensive jobs, increases as jobs become more parallelized and short running, and is aggravated by multi-layered systems. We further designed HotTub, a drop-in replacement of the JVM that can eliminate warm-up overhead by amortizing it over the lifetime of a host. Evaluation shows it can speed up systems like HDFS, Hive, and Spark, with a best case speed-up of 30.08x.

Acknowledgments

We greatly appreciate the insightful feedback from our anonymous OSDI reviewers, our OSDI shepherd Andrea Arpaci-Dusseau, and Rik Farrow. We thank Yu Luo, Serhei Makarov, Michael Stumm, Jenny Ren, Kirk Rodrigues, Guangji Xu, Yongle Zhang, and Xu Zhao for the useful and thought-stimulating discussions. We thank Yu Luo for setting up and maintaining the server cluster environment used in our experiments. His help was invaluable. This research is supported by NSERC Discovery grant, NetApp Faculty Fellowship, and an NSERC USRA award. Hailong Sun is supported by National Key Research and Development Program of China (2016YFB1000804) and National Natural Science Foundation of China (61370057).

References

[1] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*, 2012: https://www.usenix.org/system/files/conference/nsdi12/pacman.pdf.

[2] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs. Scale-out for Hadoop: Time to Rethink?" in *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*, 2013.

[3] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, "Bigbench: Towards an Industry Standard Benchmark for Big Data Analytics," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, 2013.

[4] Hypertable: "Why We Chose CPP over Java": https://code.google.com/p/hypertable/wiki/WhyWeChoseCppOverJava.

[5] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan, "Don't Get Caught in the Cold, Warm Up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, 2016: https://www.usenix.org/system/files/conference/osdi16/osdi16-lion.pdf.

[6] J. H. Saltzer, "Protection and the Control of Information Sharing in Multics," *Communications of the ACM*, vol. 17, no. 7 (1974), pp. 388–402.

[7] "StackOverflow: Is Java Really Slow?": http://stackoverflow.com/questions/2163411/is-java-really-slow.

[8] Yanpei Chen, Cloudera, "What Do Real-Life Apache Hadoop Workloads Look Like?": http://blog.cloudera.com/blog/2012/09/what-do-real-life-hadoop-workloads-look-like/.