JavaScript for Extending Low-latency In-memory Key-value Stores

Tian Zhang Ryan Stutsman

University of Utah

Abstract

Large scale in-memory key-value stores like RAMCloud can perform millions of operations per second per server with a few microseconds of access latency. However, these systems often only provide simple feature sets, and the lack of extensibility is an obstacle for building higher-level services. We evaluate the possibility of using JavaScript for shipping computation to data and for extending database functionality by comparing against other possible approaches. Microbenchmarks are promising; the V8 JavaScript runtime provides near native performance with reduced isolation costs when compared with native code and hardware-based protections. We conclude with initial thoughts on how this technology can be deployed for fast procedures that operate on in-memory data, that maximize gains from JIT, and that exploit the kernel-bypass DMA capabilities of modern network cards.

1 Introduction

Research has led to in-memory key-value stores that scale to hundreds of machines that each perform millions of operations per second with sub-5 µs access times [9, 25]. These systems achieve high performance with a combination of DRAM, RDMA, and simple feature sets. These systems are evolving high-level features like recovery [10, 24], ordered indexes [15], and transactions [10, 17], but they still lack general mechanisms to support client-supplied logic.

Even with conventional key-value stores, the lack of extensibility has led to numerous ad-hoc and custom stores, each designed to support a specific application. For example, Facebook's TAO extends `memcached` to support its social graph data model [5]. Other systems incorporate elements of SQL [28], multi-attribute accesses support for Location Aware Services [23], application-specific actions [12], value-based access methods [27], and consistent multi-key access [7] on top of key-value stores. In industry, even the popular Redis [1] key-value store, internally supports several data structures and has added loadable module support.

As we sought to use RAMCloud as a platform for building other higher-level services, its lack of extensibility was a key limitation. Applications with inter-record dependencies were latency-bound. For example, when fetching one record and using the returned value to fetch another, even RAMCloud's 5 µs round trip time was too high. Clients suffer waiting for responses from the storage server. This is exacerbated by the fact that 5 µs is hard for the client to hide with conventional multithreading; it is long enough that spinning to wait for a response dramatically limits throughput, but it is short enough that context switching to a another thread and back wastes most of the gains [2]. Throughput-bound applications suffered as well; RAMCloud's simplistic (get/put) operations prevented pushing down operations like projection, selection, and aggregation, which forced massive overheads to transmit large results to apply client-side filtering.

There are many well-known models for shipping computation to servers but unique properties of how RAMCloud uses hardware and the types of applications it is likely to host were a mismatch with existing approaches. Looking at existing models for loading user code into RAMCloud servers, we considered five key criteria.

(Near) Native Performance. RAMCloud is fast, since all data is in DRAM; it uses kernel bypass networking, and it continuously polls the network card (NIC). Servers can dispatch an operation in 1.9 µs [25]. *Any* slowdown would be immediately apparent to applications and would cut into RAMCloud's primary value. Roughly, two categories of operations capture our performance concerns. The first are compute-bound operations, and the second are memory-bound operations that traverse data structures or that filter or project records.

Low Invocation Overhead. In aggregate, a single RAMCloud server might invoke millions of stored procedures per second. If invoking a procedure added a few cache misses (about 100 ns each) to the 1.9 µs dispatch cost, it could reduce server throughput by 10% or more.

Runtime Reconfigurable. Low-latency DRAM-based storage is expensive. Many applications and users are likely to share a single, large multi-tenant deployment. Consequently, it is impossible to take the system offline to install new user-provided procedures as some distributed in-memory stores require [1, 9].

Inexpensive Isolation. A single cluster may host thousands of tenants' procedures. Not only must procedure invocation be inexpensive, it must also allow rapid and efficient switching between protection domains. For example, conventional OS and hardware based isolation

Model	Fast Compile/Install	Fast Runtime Entry/Exit	Isolation	Pointers/ Data-Dependencies	Data-intensive Functions	Compute-bound Functions
SQL	✓	✓	✓	With ⋈	✓	DB Supplied
Native/C++	✗	✗	Hardware	✗	Difficult	✓
JavaScript	✓	✓	✓	✓	✓	✓

Table 1: Strengths and weaknesses of different approaches for hosting user-supplied logic within an in-memory database.

mechanisms are too expensive. On our hardware, a minimal context switch to another process and back takes 2.2 µs, which would double the time it takes for a server to process a basic RAMCloud operation.

Low Installation Overhead. Ideally, applications would be able to install and remove procedures with very little overhead. Expensive procedure compilation times limit the ability of applications to install and run "one-shot" procedures.

We considered several approaches to hosting user-supplied code in low-latency in-memory key-value stores, and we characterize the tradeoffs made by each. Since the high performance of DRAM exposes any overheads in query execution, we initially expected that native code execution with lightweight hardware protections would be essential to the design. However, as we went along we kept coming back to the question: why not JavaScript?

Just-in-time (JIT) compilation means JavaScript has the potential to deliver near native performance. Isolation is a concern in browsers as well, so runtimes also expose protection domains and allow the host process to switch between them without special instructions or expensive traps. And, JIT makes JavaScript functions easy to install at runtime. An added bonus stems from the fact that JavaScript is the lingua franca of web development; pushing compute to storage via JavaScript is popular in many low performance stores, since it unifies the frontend, business logic, and storage programming languages [21].

JIT also provides an alternative way of running procedures written in C++ by compiling them to asm.js, a subset of JavaScript that maximizes the efficiency of the underlying JIT output [6]. Asm.js code loads fast, runs fast, and it retains the isolation benefits of the JavaScript runtime. Those who care most about performance can implement procedures in C++ to achieve better performance than vanilla JavaScript.

Can JIT bring these benefits to RAMCloud? The key questions are whether a) JavaScript can provide near-native speed for data-intensive loads, b) transitions between database logic and user logic are fast, and c) "context switches" between isolated runtimes are fast.

In this work, we take initial steps to answer these questions. First, we describe different approaches that could be used to push client logic into in-memory key-value stores, and we discuss their tradeoffs. Second, we perform several microbenchmarks to gage the suitability of the V8 JavaScript runtime [13] as a stored procedure runtime for RAMCloud. We find several simple procedures we try incur about a 2-10% slowdown versus native C++ code but that the cost of entering/exiting the runtime is 11.4-72× faster for JavaScript; compared to the cost of invoking native code isolated at the hardware or process level, JavaScript is a strong fit for many types of applications.

2 The Candidates

Shipping computation to data is a long-studied area, but low-latency stores work at odds with existing approaches. A key aspect of scale-out analytics frameworks like MapReduce [8] and Spark [30] is packaging and shipping code to bulk data to minimize communication costs. In these systems queries are massive, so latency is not a consideration. For example, these frameworks can take minutes before any actual data processing begins [8].

Table 1 summarizes the tradeoffs of different approaches for stored procedures; more details are given about each approach below. The left columns characterize the cost of installing new procedures, entering/exiting procedures, and context switch between procedures in different protection domains. The right columns characterize the fit of each approach for different types of procedures. The first type are procedures that "chase" data dependencies. For example, user profiles in a *users* table might be indexed by *user-id* and each profile might contain the user-ids of friends. Given a user-id, a procedure might return all of the profiles of the friends of that user with a single request, avoiding extra round trips to the key-value store. Such procedures are short and require fast runtime entry/exit to be efficient. Other data-intensive procedures may process many values, and compute-bound procedures may perform expensive functions on the data before returning results. These types of procedures are sensitive to runtime overhead and compiler optimizations.

SQL. SQL is a declarative query language used with both analytics (OLAP) and transaction processing (OLTP) workloads that also supports use as a stored procedure language. SQL may be the single most widely used approach to ship computation to data storage. In-memory

databases focused on combined analytic and transactional workloads have placed tremendous pressure on high performance SQL, resulting in approaches that infuse JIT and compiler technology into conventional SQL query processing [11, 22]. JIT blurs the line between the database and user code; queries run fast, and calls back-and-forth between the database and user logic are inexpensive. SQL is type safe, so it also facilitates lightweight isolation. Overall, the SQL's main drawback is that it is declarative. For most workloads, this is a benefit, since the database can use runtime information for query optimization; however, this also limits its generality. For example, implementing new database functionality, new operators, or complex algorithms in SQL is difficult and inefficient.

C++/Native Code. Using native code in low-latency in-memory stores is attractive. It works well for compute-bound tasks, and, seemingly, it should work well for data-intensive tasks too. The challenge with native code is preventing bugs from crashing the database and protecting against malicious procedures. We considered several approaches including running procedures in separate processes, software fault isolation, and techniques that abuse hardware virtualization features (§3.2). These techniques show little slow down while running code, but they greatly increase the cost of control transfer between user-supplied code and database code. For example, process-based isolation means procedure invocation requires an OS context switch (thousands of nanoseconds) both on entry and exit. In RAMCloud, many operations take less than 2 µs, so even just invocation costs dramatically impact performance. Worse, a single procedure call may access millions or billions of records (for example, selections, projections, or aggregations); if a procedure called into the database for each record, it would be prohibitive.

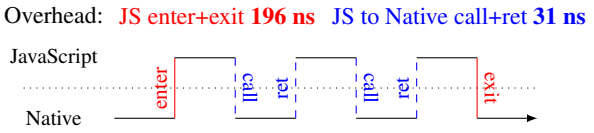
JavaScript. JavaScript has the potential to overcome these limitations. It is safe and sandboxed, it doesn't require hardware protections that impede domain switches, and JIT can make compute-bound tasks fast. To see if JavaScript will work well, we explore its overheads through a series of microbenchmarks.

3 Microbenchmarks

Experiments are run on an Intel Xeon E5-2630 v3 (Haswell) at 2.40 GHz with 64 GB of DDR4 running at 2133 MHz (an Emulab [26, 29] Dell D430) with Ubuntu Linux.

3.1 Installation, Invocation & Entry/Exit Costs

Each procedure mechanism incurs three forms of invocation overhead: the cost to compile/install a procedure, the cost to invoke the procedure, and the cost for the procedure to invoke database functionality. We do not consider garbage collection costs, since short-lived invocations will not be interrupted, which can support an inexpensive, stack-like allocation strategy. To better understand these costs, we first test a no-op JavaScript procedure (`function () {}`) invoked from a C++ host processs.




```
var chase = function (key) { return get(get(key)); }
```

Figure 1: The `chase` procedure requires six runtime crossings.

Mechanism	Context Creation	Code Compilation	Context Switch
v8::Context/JS	889 µs	427 µs	98 ns
Processes/C++	763 µs	58,000 µs	1,121 ns
`VMFUNC`[1]/C++	-	58,000 µs	138 ns
sthreads[2]/C++	2 µs	58,000 µs	150 ns

Table 2: Installation and context switch cost; **1.** `VMFUNC` only includes the cost of the instruction, not the full context switch; **2.** sthread times are from [3].

Creating a new context in the V8 runtime takes 889 µs, which takes 17% longer than forking a separate process for isolation. This is a one time cost that has little impact as long as tenants can reuse contexts from call to call. Conversely, invoking the JavaScript procedure from C++ takes 196 ns; whereas, invoking a procedure in another hardware-protected process takes 2×1,121 ns (§3.2). So, JavaScript is 11.4× faster on this key metric.

More importantly, procedures must be able to invoke database routines to access data; a no-op C++ function can be called from JavaScript in just 31 ns. This is critical for functions that touch millions or billions of records scattered across gigabytes of RAM. Hardware protection would be 72× slower than using JavaScript, since it requires thousands of cycles per record access.

For example, Figure 1 shows a simple procedure that fetches one value based on the contents of another. In addition to the cost of procedure invocation, this procedure must call into the database and back twice. Using processes for isolation requires six OS context switches (entry and exit for the procedure and two gets) at 1,121 ns for a total of 6.7 µs. Boundary crossing overheads alone increase the time it would take to process such a request by 26× compared to using JavaScript, which only requires 258 ns (196 ns for invocation and 31 ns for each database routine).

3.2 Isolation

In V8, many applications can safely share a single runtime instance; to do this, each application allocates its own *context*, which is passed to the runtime on invocation [14]. Hosts of V8 can multiplex applications by switching between contexts, just as conventional protection is implemented in the OS as process context switch. We compared

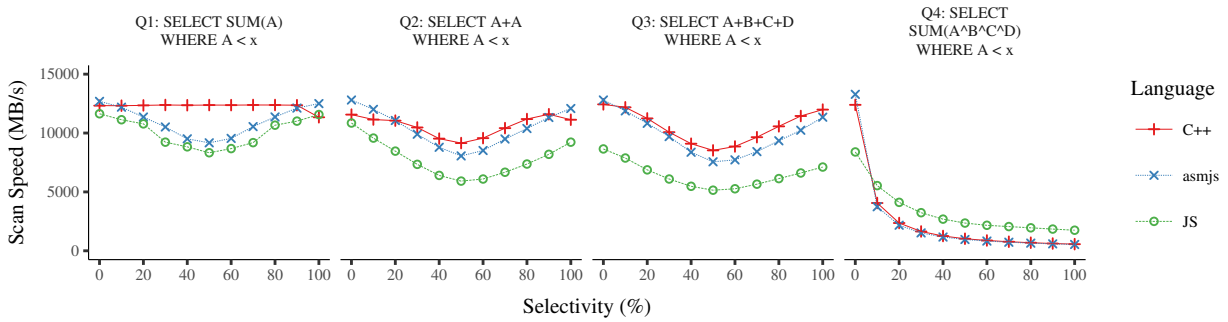


Figure 2: C++ and JavaScript procedures running simple queries over 1 GB of 64 B records.

the cost of a V8 context switch to other approaches; the results are shown in Table 2. The time shown is the time taken to cross a protection domain boundary to enter a (no-op) tenant procedure starting from the C++ host process. A V8 context switch is just 8.7% of the cost of conventional a process context switch.

We also explored exotic approaches to hardware protection to lower costs. One approach is sthreads [4], which provide fork-like isolation with thread-like context switch costs. Its costs were carefully minimized by using Dune [3] to give the sandbox direct control over kernel state via hardware virtualization support. Even with these aggressive optimizations, V8 context switch time is comparable to sthread context switch time. We also tried `VMFUNC`, an Intel virtualization instruction that lets VMs swap their underlying extended page tables (physical-to-machine page mappings) without kernel or hyper calls. Alone, it is insufficient for sandboxing, but others have used it to isolate untrusted code [18]. On our hardware, `VMFUNC` alone takes 138 ns even without the needed functionality for isolation and correct execution, already making it nearly as costly as sthreads.

The low context switch cost of V8 is attractive for our target environment where we expect large numbers of tenants to share the database system. V8 will allow more tenants, and it will allow more of them to be active at a time at a lower cost.

3.3 Memory and Compute-bound Procedures

Procedures suffer overheads at runtime entry/exit, but with JavaScript they also suffer overhead as they run. For example, the JIT compiler doesn't optimize as aggressively as a conventional C++ compiler; it includes overheads from garbage collection; and the lack of strong typing increases the number of branches needed, since all member accesses must be prepared to deal with objects of differing type. A full analysis of all of the types of procedures RAMCloud should support is challenging; here, we illustrate with some simple examples.

Figure 2 shows the results. Each plot shows the performance of a procedure written in both C++ and JavaScript that operates over a relation and performs logic equivalent to a small SQL query. Each query processes 1 GB of records that each consist of 16 32-bit integers consecutive in memory. The selectivity (x-axis) of each query is varied by changing the number of values in the relation that match the predicate `A < x`. Q1 is limited by scan speed; it performs little compute and outputs one value. Q2 performs little compute, but it outputs a percentage of the values given on the x-axis. Q3 is similar to Q2 but reads more fields from each record. Q4 performs three floating point `pow` operations and is compute bound.

In Q1, Q2, and Q3 suffer most when selectivity is 50%. This is a well-known phenomenon for selection scans [31]; at 50% branch prediction breaks down and thwarts CPU speculation. It is reassuring that JavaScript implementations exhibit this behavior, since it shows they are efficient enough to exhibit performance that is attributable low-level architectural issues. On average JavaScript is 18%, 27%, and 39% slower for Q1, Q2, and Q3, respectively. These functions are mostly memory-bound with Q2 and Q3 being more so. Q4 is compute bound and is 2.2× faster in JavaScript than the native version which relies on glibc's `pow` implementation.

Figure 2 contains a third line that represents the performance of each of the queries when they are written in C++, compiled to asm.js, and run inside V8. On average asm.js is just 10%, 2%, 7%, and 5% slower for each query respectively. Notice, Q4's performance degrades to match the original C++ performance; this is because the slower `pow` from glibc is compiled in. These results suggest that asm.js may be sufficient for applications that need the highest performance. More exploration is needed to determine how fragile these results are, especially for more complex operations.

There are two key takeaways from our analysis:

1. Procedure invocation and interactions between JavaScript and the host database process are 11.4× and 72× faster than using native code and hardware-based protections. Short and data-intensive procedures will benefit from JavaScript.

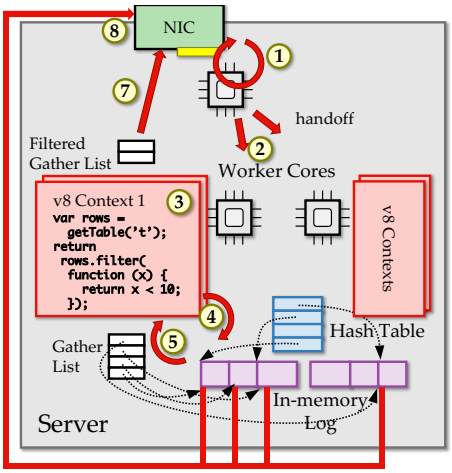


Figure 3: A core dispatches incoming requests to a worker core that runs the requested procedure with V8. The procedure fetches and filters a *gather list* of records that it passes to the NIC, which issues zero-copy DMA requests for the records.

2. V8 with asm.js is only 2-10% slower than native code, so compute-bound workloads are okay thanks to aggressive compiler optimization. V8 is harder on CPU branch prediction than native code.

4 Design for Efficiency

Our initial experiments are encouraging; careful embedding of JavaScript into RAMCloud promises to accelerate many of the applications we would like to build on top of it. They have also clarified that how we embed the runtime will impact its performance. Figure 3 gives an overview of how we envision V8 can be embedded into a low-latency request processing flow while minimizing data copying, maximizing JIT optimization opportunities, and using modern kernel-bypass networking. Five key features drive its design.

Leverage JavaScript Types and JIT. Database state must be protected. With native code it would be impossible to provide fine-grained direct access to database records unless tenant state were strictly partitioned among memory pages. With JavaScript, the database can safely return pointers to records and rely on the runtime to enforce access boundaries. As a result, JavaScript stored procedures can avoid extra `memcpy` overhead and can work with records directly where they live. To do this, RAMCloud will export new scatter-gather list abstractions to JavaScript similar to `mbufs` [20]. A major advantage of this approach is that the JavaScript runtime is aware of references, so the JIT can optimize around the underlying types of the records and fields being exposed to it. For example, the JIT will be able to fully exploit instruction-level parallelism and low-level hardware features like SIMD instructions when processing database records as a result.

Minimize Data Movement. This also leads to a second powerful optimization: since many procedures will manifest as a set of transformations on these scatter-gather lists, JavaScript procedures will be able to pass them to the database which can forward them directly to the network card for zero-copy DMA [16]. That is, JavaScript routines will be able to transmit results to clients that the CPU has never touched. For example, this can result in significant savings for operations that select wide records using a small number of subfields.

Exploit Semantics for Garbage Collection. In a low-latency store, procedures are invoked as part of a request-response cycle and most complete quickly. This can be exploited to eliminate garbage collection except in the cases where a procedure runs for long periods. Eliminating garbage collection improves performance and significantly improves jitter [19].

Expose Database Abstractions. JavaScript procedures will have access to the full capabilities of the RAMCloud process in which they are embedded. Using RAMCloud's recovery logging facilities, it should be possible to implement interesting functionalities; for example, JavaScript procedures may be able to implement transactions, indexes, triggers, and pub/sub callbacks.

Fast Protection Domain Switch. Finally, tenant runtime state can be kept lightweight, so each server should be able to efficiently manage tens of thousands of tenants. We are also considering allowing tenant JavaScript runtime state to be made persistent using RAMCloud's fast remote replication protocol.

5 Conclusions and Looking Forward

Developing new systems and applications on RAMCloud, we have repeatedly run into the need to push computation into storage servers. The low latency and low overhead of in-memory storage and fast networking led us to think native code was the right approach to extending the storage nodes. However, we kept coming back to JavaScript. After an initial analysis of its costs and overheads, it still seems promising. Its low invocation and isolation costs combined with safe, direct client access to fine-grained database state make it appealing.

Our next goal is to embed V8 into the RAMCloud server; to develop a smart API for procedures that exposes rich, low-level database functionality; and to begin experimenting with realistic and large-scale applications.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1566175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Discussion Topics

Setting out to build a stored procedure system for RAMCloud, we initially expected native code and novel hardware isolation mechanisms might be needed for high performance. After our initial assessment, we are no longer convinced that is the best route. JavaScript/V8 now seems to be a strong isolation mechanism that is ubiquitous, well-optimized, and can still even support fast languages like C/C++ (through asm.js).

The first key point of discussion this idea raises is whether JavaScript is a good model for pushing computation to low-latency in-memory storage. More generally, is an extensible low-latency in-memory key-value store a useful building block? If such system were available, what interesting applications could be built with it and benefit from it? We have some specific applications in mind. For example, distributed concurrency control operations, relational algebra operators, materialized view maintenance, partitioned bulk data processing as in MapReduce, and custom data models like Facebook's TAO can all be implemented as extensions.

Secondly, for now we are optimistic about V8, but JavaScript/V8 performance is likely to be more fragile than approaches that rely on hardware protection. We saw this ourselves in our microbenchmarks. One key question that must be assessed for JavaScript to be successful in a low-latency multi-tenant store is what types of procedures are likely to perform poorly in V8? Are those cases prohibitive?

Thirdly, a point of discussion this idea is likely to surface is how much fast kernel-bypass networking and in-memory data change the picture. JavaScript for databases/stored procedures is not new, but the constants have all changed. For example, we believe this puts pressure on the boundaries between the host environment and the JavaScript runtime that are less pronounced in other systems. We are eager to understand where this creates interesting challenges and where this combination simply results in straightforward engineering issues.

Finally, we are still seeking a better understanding of other isolation approaches, particularly for latency sensitive and short operations, to see if we have missed obvious, promising comparisons. For example, other software fault isolation techniques like Native Client could also make sense, but so far V8 has been promising enough that we have focused our exploration there.

References

[1] Redis. `http://redis.io/`. 7/24/2015.

[2] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, Mar. 2017.

[3] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, 2012. USENIX.

[4] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[5] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.

[6] D. Herman, L. Wagner, and A. Zakai. asm.js. `http://asmjs.org/spec/latest/`, 2017.

[7] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[9] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, Apr. 2014. USENIX Association.

[10] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance . In *SOSP*, pages 85–100, 2015.

[11] C. Freedman, E. Ismert, and P. Larson. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Engineering Bulletin*, 37(1):22–30, 2014.

[12] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key-value store. In *OSDI*, pages 323–336, 2010.

[13] Google Inc. Chrome V8. `http://developers.google.com/v8/`, 2017.

[14] Google Inc. Embedder's Guide. `http://github.com/v8/v8/wiki/Embedder%27s-Guide`, 2017.

[15] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. SLIK: Scalable Low-Latency Indexes for a Key-Value Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 57–70, Denver, CO, June 2016. USENIX Association.

[16] A. Kesavan, R. Ricci, and R. Stutsman. To Copy or Not to Copy: Making In-Memory Databases Fast on Modern NICs. In *Proceedings of the 4th VLDB Workshop on In-Memory Data Management and Analytics*, IMDM '16, 2016.

[17] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 71–86, New York, NY, USA, 2015. ACM.

[18] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619. ACM, 2015.

[19] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[20] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2014.

[21] MongoDB, Inc. MongoDB for GIANT Ideas | MongoDB. `http://www.mongodb.com/`, 2017.

[22] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[23] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 7–16. IEEE, 2011.

[24] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[25] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 33(3):7:1–7:55, Aug. 2015.

[26] R. Ricci and E. Eide. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architecturesand Applications. *; login:*, 39(6):36–38, 2014.

[27] Y. Tang, A. Iyengar, W. Tan, L. Fong, and L. Liu. Write-optimized indexing for log-structured key-value stores. Technical report, Georgia Institute of Technology, 2014.

[28] The Apache Software Foundation. Apache Cassandra. `http://cassandra.apache.org/`.

[29] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *In Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[31] S. Zeuch and J.-C. Freytag. Selection on Modern CPUs. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Management and Analytics*, IMDM '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM.