Efficient In-memory Data Management: An Analysis

Hao Zhang†, Bogdan Marius Tudor†, Gang Chen#, Beng Chin Ooi†
†National University of Singapore, #Zhejiang University
†{zhangh,bogdan,ooibc}@comp.nus.edu.sg, #cg@cs.zju.edu.cn

ABSTRACT

This paper analyzes the performance of three systems for in-memory data management: *Memcached*, *Redis* and the *Resilient Distributed Datasets (RDD)* implemented by Spark. By performing a thorough performance analysis of both analytics operations and fine-grained object operations such as *set*/*get*, we show that neither system handles efficiently both types of workloads. For Memcached and Redis the CPU and I/O performance of the TCP stack are the bottlenecks – even when serving in-memory objects within a single server node. RDD does not support efficient *get operation* for random objects, due to a large startup cost of the *get* job. Our analysis reveals a set of features that a system must support in order to achieve efficient in-memory data management.

1. OBJECTIVE AND EXPERIMENTAL METHODOLOGY

Objective. Given the explosion of Big Data analytics, it is important to understand the performance costs and limitations of existing approaches for in-memory data management. Broadly, in-memory data management covers two main types of roles: (i) supporting analytics operations and (ii) supporting storage and retrieval operations on arbitrary objects. This paper proposes a performance study of both analytics and key-value object operations on three popular systems: Memcached [2], Redis [3] and Spark's RDD [7].

Workloads setup. To test the analytics performance, we use the PageRank algorithm implemented in a Map/Reduce style. In the Map phase, we compute the contributed rank for the neighbors of every web page, and distribute this information to other nodes. In the Reduce phase, each node computes the new ranks for the local web pages based on the contributed ranks.

Spark naturally supports Map/Reduce computations, and we use the default PageRank implementation shipped as part of Spark 0.8.0 examples. RDDs are persisted into memory before we use it. We use Spark 0.8.0/Scala 2.9.3 with Java 1.7.0.

Memcached is a key-value store that only supports operations such as *set*/*get*. To implement PageRank algorithm on top of Memcached, we implement a driver program to do the computations. The driver uses Sypmemcached Client 2.10.3 [4] to connect to the Memcached servers. Specifically, a driver program is hosted inside each Memcached server node to manage its local server and communicate with remote servers. We coordinate all driver programs with a master program that instructs the drivers into map and reduce steps. The TCP protocol is used for the communication between all Memcached servers and the drivers. We use Memcached 1.4.15 compiled using gcc 4.6.3 with the default settings. Figure 1 shows the architecture of the analytics operations on top of Memcached.

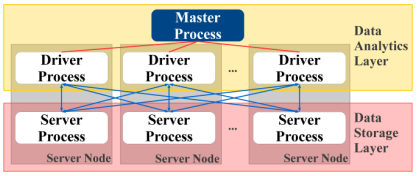

Figure 1: Analytics Architecture over Memcached and Redis

Like Memcached, Redis is a key-value store with basic *set*/*get* operations and a set of advanced functions such as pipelined operations, server-side scripting, and transactions. A similar setup as Memcached is used for Redis, as described in Figure 1, termed *Redis client-side*. The driver uses Aredis Client 1.0 [1] to connect to the servers. Unlike Memcached, Redis supports server-side scripting. Thus, the PageRank processing can be done directly by the Redis servers via Lua scripts, without relying on a driver program. We refer to this manner as *Redis server-side* data analytics. We use Redis 2.6.16 compiled using gcc 4.6.3 with the default settings.

The implementation of PageRank in both Memcached and Redis requires one key-value object to hold the neighborhood information of each node in the graph, and one for the PageRank information of each node. Spark's PageRank implementation uses two RDDs. The first RDD stores each graph edge as a key-value object, and the second stores the computed PageRanks.

Memcached and Redis servers, and RDD worker are configured with cache size of 5 GB; Redis persistence is disabled during the experiments. We use the default number of threads for all the server systems (e.g. 4 threads for Memcached). To stress-test the performance of Memcached/Redis servers, we use drivers that support multi-threaded asynchronous connections to the servers. Based on a tuning experiment, we select the thread configurations that achieve the best performance: five threads for Memcached driver, and six threads for the Redis driver.

Datasets. We run PageRank for 10 iterations using two datasets. The first dataset is Google Web Graph Dataset [5] of 875,713 nodes and 5,105,039 edges. The size of this dataset on disk is 72 MB. When loaded into a single node's memory, it takes 85 MB in Redis, 135 MB in Memcached and 3.9 GB in RDD. The second dataset is Pokec Social Network Dataset [5] consisting of 1,632,803 nodes and 30,622,564 edges. On disk this dataset takes 405 MB, and

Proceedings of the VLDB Endowment, Vol. 7, No. 10

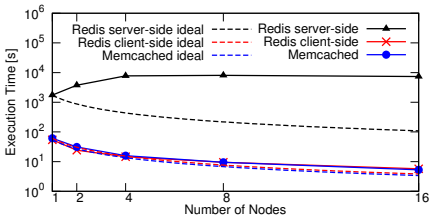

Figure 2: Execution time of PageRank - Memcached/Redis

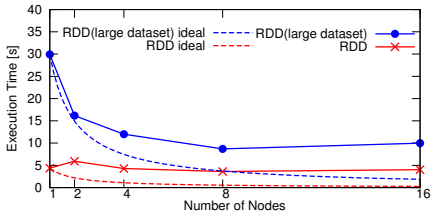

Figure 3: Execution time of PageRank - RDD

when loaded in RDD, it exceeds the maximum cache size of 5 GB of a single node. The second dataset is only used to show the scalability of RDD, since the first one is too small for RDD.

In Memcached/Redis, the data distribution among cluster nodes is based on the hash of the graph node id. Due to the hashing mechanism, Memcached/Redis cluster nodes are storing datasets of equal sizes. RDD uses Spark's default partition mechanism.

Systems setup. We use two types of systems for performance measurements:

1. *Cluster setup*. We perform scalability analysis on a cluster with 16 Intel Xeon X3430 nodes, each with 4 cores/4 hardware threads, 8 GB DDR3 RAM, 256 KB L1 Cache, 1 MB L2 Cache, 8 MB L3 Cache, inter-connected using 1Gbps Ethernet, and running 64-bit Linux kernel 2.6.18.

2. *Single-node setup*. For efficiency analysis, we need to remove the network I/O bottleneck, and perform the experiments inside a single server node. The node has two Intel Xeon X5650 processors, each with 6 cores/12 hardware threads, and 24 GB of DDR3 RAM. The memory architecture is *Non Uniform Memory Access*, where each processor directly connects to 12 GB of RAM, and the two processors are interconnected by a fast *Intel QuickPath* bus that can support transfers at 25.6 GB/s. We control the CPU affinity such that we use one processor to host the server and the other processor to host the client/driver. This ensures that the two programs do not interfere with each other, thus approximating the execution over a very fast lossless network. This node is running 64-bit Linux kernel 3.2.0.

Due to the differences in the types of nodes, we never compare directly the performance of these types of server nodes. Instead, the setups are used for different types of performance analysis.

We use the *perf* tool to access the hardware event counters required for the architectural-level analysis, the *time* command to measure user and kernel time, and the *strace* tool to track the system calls. All jobs are running alone in the server nodes, and we stop unnecessary background programs. To minimize performance variability, all the runs are repeated three times and the average among these runs is reported. However, none of the experiments exhibit significant differences among these three runs.

2. PERFORMANCE OF IN-MEMORY ANALYTICS

2.1 Cluster Performance

Figures 2 and 3 show the execution time for one iteration of all four versions of PageRank implementations: Memcached, Redis client-side, Redis server-side and RDD when using up to 16 nodes. Both figures plot the execution time as a function of the number of nodes, and the ideal scalability lines.

The most surprising result is the gap in the performance between RDD and Memcached/Redis. RDD finishes one iteration 1.3-14× faster than Memcached, 1.4-12× faster than Redis client-side, and over 400× faster than Redis server-side.

Redis server-side has poor performance because the Redis server operates in a single-threaded manner. Within a long-running scripting job, the server cannot service requests from other servers. Thus, increasing the number of servers worsens the performance, due to the increasing waiting time among servers.

Memcached, Redis client-side, and RDD exhibit different degrees of scalability. Memcached and Redis client-side generally scale well, and their execution time is close to the ideal scalability line. RDD, although finishing much faster, scales worse. We notice that RDD exhibits different ratios of computation time to communication time as the number of nodes increases. On small number of nodes, the graph size exceeds the maximum cache size. Thus, during the computation of PageRank, RDD must cycle between loading from disk and performing computations on the in-memory data. The communication time is typically small, because it involves a small number of nodes. Thus, on small number of nodes, the computation time of PageRank dominates the execution time. In contrast, on large number of nodes, the dataset can fit completely in the memory, and the computation requires much less disk activity. Thus, when using large number of nodes, the communication time becomes the dominant part. Because the communication cost depends linearly on the number of nodes, the PageRank execution time does not decrease anymore when adding more nodes. This behavior ameliorates as the dataset size increases, because larger datasets shift the bottleneck to the computation phase.

In the next subsection we investigate the reasons for the performance gap between RDD and Memcached/Redis.

2.2 Single-Node Analysis

We investigate the reasons for the large performance difference between RDD and Memcached/Redis. We run all four programs on the single-node setup and collect important OS-level performance metrics, summarized in Table 1. We measure the cumulative CPU utilization (average number of cores used by the programs; maximum allowed is 12), the percentage of the CPU time spent in the kernel and the number of system calls performed per second.

Metric	RDD		Memcached		Redis client-side		Redis
	Worker	Driver	Server	Driver	Server	Driver	Server-side
CPU utilization	3.79	0.22	0.326	2.166	0.354	4.008	0.71
System time [%]	<0.01	<0.01	61	23	31	9	68
Syscalls/s	–	–	63093	665891	33567	115309	32700

Table 1: OS-level performance of PageRank

RDD, Memcached and Redis client-side exhibit good CPU utilization, and benefit from the multi-core architecture. However, RDD spends very little time in the kernel, while both the drivers and the servers of Memcached/Redis spend 9% to 61% of the CPU time inside the kernel, which is further confirmed by the large number of system calls performed by these programs.

Using the *strace* system call profiler, we see that the Memcached and Redis drivers perform a large number of system calls per sec-

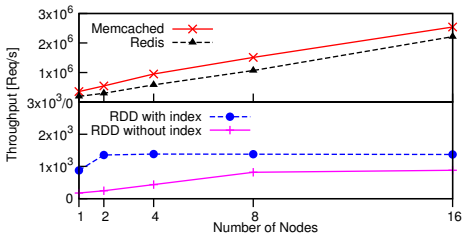

Figure 4: Multi-node *get* throughput

ond. On closer investigation we discover that 90% and 99% of these calls are *futex* system calls, caused by the synchronization among the drivers' threads. However, these calls solve very fast, and do not explain the long time spent in kernel mode. The most time-consuming system calls of the drivers are related to reading from and writing to the network. Similarly, on the server side, more than 98% of the system-level CPU time is spent in a sequence of system calls related to reading from and writing to the network.

For each data object request, the driver performs two system calls, while the Memcached server performs three system calls, and the Redis server performs six system calls. As the main bottleneck is linked to the performance of the object operations via TCP, it motivates us to perform a deep analysis of the suitability of TCP for transporting key-value objects, which is shown in Section §3.

The main reason for the good performance of RDD is that the analytics operations are performed inside the same process as the data. In contrast to Memcached and Redis, RDD does not perform any system calls to access the data, relying on direct access to the process heap. Good parallelization by using Java threads further increases the efficiency of analytics operations over RDD.

3. PERFORMANCE OF OBJECT OPERATIONS

We perform an analysis of the throughput of *set*/*get* operations on arbitrary objects. However, RDD does not support *set* operations. Thus, for RDD we profile only the *get* operation, by writing a program that performs *get* operations using the *look-up* API. Spark does not provide an index for the RDD objects. To support efficient *get*, we implement an indexing mechanism in Spark by hashing all the keys to the offset inside the RDD partitions. The experiments are conducted in both multi- and single-node configurations. The performance of *set* and *get* operations are very similar, and thus we only discuss one of them in different scenarios.

3.1 Concurrency Tuning

To make sure that the performance of the server is maximized, we perform an experiment during which we vary the number of clients, the number of network connections between clients and servers, and the number of threads of the server for Memcached (Redis is single-threaded). We then select the configuration that maximizes the throughput. All subsequent experiments described in this section are performed with these configurations as follows.

In the multi-node experiments, we set (i) four threads for Memcached servers, three client nodes per server node, each client node holding 10 clients, each client using a single TCP connection per server; (ii) same settings for Redis expect for the thread configuration; (iii) 64 threads for RDD driver and default settings for Spark.

In the single node experiment we use (i) one Memcached server using 12 threads, four clients, each using 50 concurrent connections; (ii) one Redis server, three clients each with 50 concurrent connections; (iii) same settings for RDD as in the multi-node.

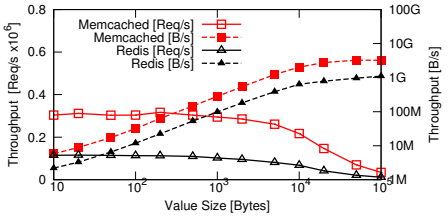

Figure 5: Single-node *set* throughput

3.2 Multi-node Throughput

We run the three systems on up to 16 nodes. After data populating phase (10^7 records with random keys), the clients/drivers continuously perform *get* operations. We sample the throughput in units of two seconds, for 10 times, and report the maximum observed throughput. Figure 4 shows the throughput of a *get* operation for small objects (10B for both key and value).

We see that the throughput of Memcached/Redis scales almost linearly with the number of nodes, and is in the range of millions of requests per second. In contrast, the *get* throughput of RDD is three orders of magnitude lower and does not scale beyond 8 nodes. Surprisingly, even a hash-based index for RDD only improves the throughput by 1.6-5× compared to RDD without index.

The large disparity in the performance is caused by two factors. First, because of the job isolation mechanism of Spark, there is only one driver that can serve requests to all its RDDs. This architecture puts a large performance burden on the driver, which can quickly become the bottleneck when serving requests to multiple nodes. In contrast, Memcached and Redis allow an indefinite number of clients to query the same data. In our case, even with a multithreaded driver for RDD, the performance plateaus after two nodes for RDD with index and eight nodes for RDD without index. Second, each *get* operation of RDD is associated with a job startup overhead of about 0.8 milliseconds, for both versions of RDD. This long response time amplifies the first performance problem.

Memcached and Redis scale well with the number of nodes, but the throughput per-node is still substantially below the peak network throughput: 6-17 MB/s out of 125 MB/s.

3.3 Single-node Throughput

3.3.1 Memcached and Redis

We use the single-node setup to analyze the throughput of both *set* and *get* operations as a factor of key and value size for Memcached and Redis. As the key size and value size have similar effects on the throughput, we only show the results for the value size. The results, shown in Figure 5 reveal an unexpected behavior. The requests-per-second throughput is almost constant when the value size changes between 10 bytes and 1 kB. From 1 kB to 10 kB there is a slight dip. It implies that it is almost as expensive to transfer a 10 bytes object as a 10 kB object. In contrast, from 10 kB onwards the requests-per-second throughput decreases significantly, which shows better utilization of the available network bandwidth.

To understand the reason for the poor performance of transferring small objects, we perform an analysis of the CPU activities of the Memcached and Redis servers during the transfers of objects of different sizes. We discover serious inefficiencies in the way the Memcached and Redis servers make use of the CPU architecture.

Table 2 lists some important CPU performance metrics for values of different sizes (10 bytes key), which show two consistent trends for both Memcached and Redis, in the *set*/*get* operations: (i) Instruction cache miss rate is high for small objects and decreases

Metric	10 bytes		100 bytes		10 kB	
	Memcached	Redis	Memcached	Redis	Memcached	Redis
CPU utilization	4.87	0.99	4.95	0.99	5.36	0.99
System time [%]	72.43	57.20	71.40	58.63	72.63	54.24
L1D miss rate [%]	9.82	7.49	10.19	7.75	15.56	15.59
L1I miss rate [%]	19.78	14.80	19.45	14.56	16.56	13.91
LLC miss rate [%]	9.40	10.61	9.54	10.82	5.70	10.50

Table 2: CPU performance metrics for different value sizes

with an increase in the value size; (ii) Data cache miss rate has an opposing trend.

The instruction cache miss rate of 20% for Memcached and 15% for Redis when transferring small key-value objects are unexpected, since their sources do not exceed 20,000 lines of C code. Furthermore, Memcached/Redis spends all the service time inside a loop where it checks the sockets using the `epoll_wait` system calls, and subsequently `read`/`write` them. For comparison, a highly memory-bounded program such as *SP* from NASA Parallel Benchmark that is performing analytics on a penta-diagonal matrix has 0.01-0.05% instruction cache miss rate on the same hardware.

The high instruction cache miss rate has deep implications on the efficiency of using the CPU. Modern CPU cores operate using an *out-of-order execution*, which allows multiple data cache misses to be fetched from L2/L3 caches or the main memory, but only one instruction cache miss. Thus, the latency of the memory is hidden under many concurrent data requests, but for instruction cache misses, the full latency penalty must be incurred. Due to this, CPU appears to be active during Memcached/Redis server execution, but instead is spending more than 70% of the CPU time waiting for instructions to be fetched from memory without doing useful work.

We use *perf record* to gather traces of the kernel function calls. During Memcached/Redis execution, the kernel spends most of the time in the TCP code and the epoll mechanism. But most of the L1 instruction cache misses come from the TCP code: by visual inspection, we see that kernel functions prefixed with *tcp* and *__inet* trigger more than 30% of L1 instruction cache misses for both Memcached and Redis when using objects of less than 100 bytes.

We conclude that TCP is inefficient at handling small objects. But analytics computations such as PageRank mostly encapsulate numbers inside key-value objects, thus generating many small objects. To efficiently transfer key-value objects, all objects should ideally be larger than 100 kB, as seen in Figure 5. The implications of this conclusion are further discussed in Section 4.

A recent paper which analyzed the performance of Memcached in web cache deployments also noticed unexpected low performance for object sizes between 1 kB and 4 kB [6]. They also observed that network stack is the main culprit, but their analysis does not link the architectural-level characterization to the OS-level analysis. And in web deployments, the key-value object requests cannot be coalesced into larger meta-objects. The solution proposed by them is a custom hardware that handles network connections without interventions from CPU. In contrast, most key-value objects in data analytics are less than 20 bytes, which can be coalesced into meta-objects of up to hundreds of kilobytes. Thus, a software solution relying on coalescing small objects may achieve good performance.

3.3.2 RDD

The single-node performance of RDD on *get* operation is shown in Figure 6, for varying number of records stored in the RDD.

In the original RDD design, for small number of records, the startup cost of the *get* job is the bottleneck. Past 10^6 records, due to the sequential scan used by RDD, the memory access becomes the bottleneck. This can be seen by the strong correlation between throughput loss and last level cache (L3) miss rate.

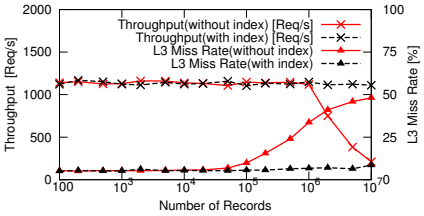

Figure 6: Single-node *get* throughput for RDD

The throughput of RDD with index is similar to the original design, for number of records less than 10^6. This confirms that the startup cost is the bottleneck. Even past 10^6 records indexing does not lead to significant advantages, because the startup cost compromises the achievable throughput. We conclude that indexing does not bring significant advantages to the throughput of *get* operations because the job startup time dominates the response time.

4. CONCLUSIONS AND SYSTEM DESIGN IMPLICATIONS

This paper provided a detailed analysis on the performance of Memcached, Redis, and RDD for both analytics operations and fine-grained key-value object operations. We showed that neither system handles both tasks efficiently. The architecture of Memcached and Redis forces the computation component of the application to use TCP to access the in-memory data, and is shown to be detrimental to the performance. RDD does not support efficient object *get* operations due to a large job startup cost.

Our analysis provides a set of insights on designing a system that efficiently handles both analytics and fine-grained object operations. First, a system must support fast inter-process communication (IPC) within the same node, and efficient transfer of small key-value objects. Analytics applications such as PageRank generate many such objects because they need to encapsulate numbers. Second, IPC across nodes must still use TCP, as UDP is unreliable and may lead to errors if requests are dropped, especially for *set* requests. However, it must avoid as much as possible to transfer small key-value objects, potentially by coalescing several key-value objects into a larger meta-object. Third, an index is required for fast random access to an object. Fourth, a lightweight architecture for accepting *set*/*get* requests is required; otherwise the startup cost of an object operation will compromise the achievable throughput.

5. ACKNOWLEDGMENTS

This work was supported by A*STAR project 1321202073.

6. REFERENCES

[1] Aredis java redis client. http://aredis.sourceforge.net/.
[2] Memcached. http://memcached.org.
[3] Redis. http://redis.io.
[4] Spymemcached memcached client. https://code.google.com/p/spymemcached/.
[5] Stanford large network dataset collection. https://snap.stanford.edu/data/.
[6] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *ISCA*, 2013.
[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, 2012.