Benchmarking and Performance Analysis for Distributed Cache Systems: A Comparative Case Study

Chapter · January 2018

DOI: 10.1007/978-3-319-72401-0_11

Haytham Salhi
Birzeit University
3 PUBLICATIONS **9** CITATIONS
SEE PROFILE

Adel Taweel
Birzeit University
134 PUBLICATIONS **1,254** CITATIONS
SEE PROFILE

Benchmarking and Performance Analysis for Distributed Cache Systems: A Comparative Case Study

Haytham Salhi[1](✉), Feras Odeh[1](✉), Rabee Nasser[1](✉), and Adel Taweel[1,2](✉)

[1] Birzeit University, Birzeit, Palestine
hsalhi89@gmail.com, ferasodh@gmail.com, rabinasser@gmail.com, ataweel@birzeit.edu

[2] King's College, London, UK

Abstract. Caching critical pieces of information in memory or local hard drive is important for applications' performance. Critical pieces of information could include, for example, information returned from I/O-intensive queries or computationally-intensive calculations. Apart from such, storing large amounts of data in a single memory is expensive and sometimes infeasible. Distributed cache systems come to offer faster access by exploiting the memory of more than one machine but they appear as one logical large cache. Therefore, analyzing and benchmarking these systems are necessary to study what and how factors, such as number of clients and data sizes, affect the performance. The majority of current benchmarks deal with the number of clients as *"multiple-threads but all over one client connection"*; this does not reflect the real scenarios where each thread has its own connection. This paper considered several benchmarking mechanisms and selected one for performance analysis. It also studied the performance of two popular open source distributed cache systems (Hazelcast and Infinispan). Using the selected benchmarking mechanism, results show that the performance of distributed cache systems is significantly affected by the number of concurrent clients accessing the distributed cache as well as by the size of the data managed by the cache. Furthermore, the conducted performance analysis shows that Infinispan outperforms Hazelcast in the simple data retrieval scenarios as well as most SQL-like queries scenarios, whereas Hazelcast outperforms Infinispan in SQL-like queries for small data sizes.

Keywords: Benchmarking · Performance analysis
Distributed cache systems · Hazelcast · Infinispan
Retrieval operations

1 Introduction

Studying the performance of distributed cache systems has received much attention in recent years due to their wide usage in improving latency and throughput significantly for various applications. In computing, cache is a software component that stores portions of datasets which would otherwise either take a long time to calculate, process, or originate from an underlying back-end system [16,21]. Caching is used mainly to reduce additional request round trips and sometimes to reduce database querying time for frequently used data [16,21].

A typical methodology for analyzing the performance of distributed cache systems is often done by performing a controlled- and an unbiased-study to investigate these systems' factors of influence. There have been extensive empirical studies conducted by researchers [13,24] and industry [2,7,8] that attempt to look into the performance of these systems. For example, Zhang et al. [24] analyzed the performance of three systems: Memcached, Redis, and the Resilient Distributed Datasets (RDD). More recently, Das et al. [13] studied the performance for Hazelcast only. Nevertheless, although some of these studies have studied more than one factor (such as number of client threads) and targeted many types of cache operations, they studied *multiple threads over one client connection*, and little attention has been paid to study *number of client connections*, where each client opens its own connection to distributed cache server, as well as the behavior when varying the data sizes.

Thus, this paper presents the performance analysis of retrieval operations (including *get* and *SQL-like* queries) of two popular open source distributed cache systems, namely Hazelcast (version 3.6.1) and Infinispan (version 8.1.2.Final), with a focus on two factors: *different number of concurrent clients* for *different sizes of data* managed by the cache. In other words, to be able to understand some of the intrinsic properties of distributed cache, the paper studies the performance behavior of the two systems by varying the number of concurrent client connections for different data sizes, through a controlled study, which is the main objective of this paper. In addition, the paper considers several potential benchmarking frameworks and identifies a suitable one, namely Yardstick. Yardstick is a benchmarking tool usually intended for general distributed systems and captures the behavior of performance as a function of time (i.e., the duration of benchmark) only, which is not sufficient to understand the exact performance behavior of distributed cache systems for our factors of interest. To overcome, an additional mechanism has been developed and integrated into Yardstick to benchmark distributed caches to capture the varying number of clients and data sizes to ensure proper synchronization of run-times.

The evaluation results show that there is a clear relationship between the performance of data retrieval operations and number of concurrent clients as well as data sizes. In addition, the performance behavior of the two selected systems can significantly be influenced by other implementation factors such as *data serialization*, *object formats*, and *indexing*. The rest of this paper is structured as follows: Sect. 2 presents related work. Section 3 describes the used study design and setup. Section 4 shows the conducted study results. Section 5 discusses the results and the drawn interpretations. Finally, Sect. 6 draws the conclusion and future work.

2 Related Work

Das et al. [13] studied the performance degradation of Hazelcast and suggested to spawn fewer number of threads that process number of client requests in order to improve the performance. While this study aimed to perform a controlled study and add a capability to Yardstick for conducting a performance comparison in the context of multi-client connections, some studies added utilities to ease the process of performance analysis, such as an emulator e.g., *InterSense*, to aid the performance analysis of distributed big-data applications and to facilitate the sensitivity analysis of complex distributed applications [22], and other studies have performed empirical performance analysis on distributed systems like SQL engines [23]. Several papers proposed different methods for conducting performance analysis for general distributed systems [11,14,19,20]. However, our approach focuses on distributed cache systems that use both virtual and physical configurations, which introduce additional factors that need to be taken into consideration when conducting performance analysis.

Zhang et al. [24] analyzed the performance for in-memory data management of three systems: Memcached, Redis, and Resilient Distributed Datasets (RDD) implemented by Spark. The authors performed a thorough performance analysis of object operations such as *set* and *get*. The results show that none of the systems handles efficiently both types of workloads. The CPU and I/O performance of the TCP stack were the bottlenecks for Memcached and Redis. On the other hand, due to a large startup cost of the *get* job, RDD does not support efficient *get* operation for random objects.

Industry, on the other hand, especially the companies that offer distributed cache systems, whether open-source or commercial, usually build benchmarks to show the performance of their system or to compare it with another and publish their results as white papers or on their web sites, which may carry some bias. Hazelcast company [7], for example, built benchmark for *get* and *put* operations only, to compare their distributed cache (3.6-SNAPSHOT) with Red Hat Infinispan 7.2 (a version supported by Red Hat), using Radar-Gun benchmarking framework[1]. Based on their results, they claim that they are up to 70% faster than Infinispan [7]. However, this comparative study performed did not take into account the number of concurrent clients (where each client opens its own connection to the cluster) nor did consider other retrieval operations, such as SQL-like queries.

The Hazelcast company built other benchmarks comparing their distributed cache system to other systems, such as Redis [8], using Radar-Gun framework. In this study, the Hazelcast company investigated the effect of very small number of clients (1 and 4) with different number of threads and showed Hazelcast outperform Redis for the *get* operation [8]. In another comparison between Grid Gain and Apache Ignite [2] performed by Hazelcast, they studied the performance of several operations including *put/get* and *SQL-like* queries, using Yardstick framework. Others like Grid Gain company [1] built benchmarks, comparing

[1] https://github.com/radargun/radargun/wiki.

between their system and Hazelcast for several operation types such as *get*, *put*, *SQL-like* as well as transactional operations. In addition, they did the same for Apache Ignite and Hazelcast [4].

All the above mentioned studies, except the one done by Zhang et al. [24], show the performance behavior of cache operations as a function of time with a fixed number of clients, a fixed number of threads per client, and a fixed data size. These may be sufficient in some cases, but do not, however, reflect real-life performance behavior, where number of concurrent clients dynamically varies over the period of system run-time. To address this issue, the authors developed a mechanism to maintain varying number of concurrent clients and data sizes, along with the function of time, maintained by Yarkstick.

While other efforts have developed the "Yahoo! Cloud Serving Benchmark" (YCSB)[2] [12] into other extensions like YCSB+T [15] in order to produce metrics for database operations within transactions and detect anomalies from any workload, Yardstick[3] was chosen. Yardstick is a powerful framework, well-documented and intended for benchmarking distributed operations. Its benchmarks can be developed faster than other frameworks, such as Java Microbenchmarking Harness (JMH)[4], Radar Gun, and YCSB [12]. Moreover, Yardstick is open source, written in Java and allows contributions to enhance and enrich its framework.

3 Study Design and Setup

This section discusses the main aspects of the study design on which the setup relies. First, it presents the investigated key factors of interest that may have greater effect on the performance of retrieval operations for distributed cache systems. Second, it lists the queries that were used, with their specifications and complexities.

Finally, it describes the used topology of machines, their setup, and the mechanism of benchmarking.

3.1 Factors of Interest

The performance of a distributed cache system depends on several different factors including number of concurrent clients [10,18], data sizes [9], type of operations, complexity of queries, number of distributed caches, and so forth. Since the dependent variable of interest is *the performance of data retrieval operations*, this study is particularly concerned with the effect of two key factors as follows:

- **Number of concurrent clients:** The more the number of concurrent clients a system can handle efficiently, the more efficient the application is [10,18]. To achieve reasonable performance system outlook, the study is run against eight

[2] https://github.com/brianfrankcooper/YCSB/wiki.
[3] https://github.com/yardstick-benchmarks/yardstick.
[4] http://tutorials.jenkov.com/java-performance/jmh.html.

variations of concurrent client numbers which includes: **1, 2, 4, 8, 16, 32, 64, and 128** clients to log the performance change as the number of clients grows.

– **Data size:** As the data size increases in the cache, a distributed cache system needs to maintain the maximum number of entries it can store [9]. To achieve, this study examines five variations of data sizes as shown in Table 1. One million data size (or records) was tested in all cases, except in benchmarking SQL-like queries due to the huge number of records returned to the clients which caused memory heap exceptions in some cases.

Variables that may affect the performance of data retrieval operations in a distributed cache systems, such as those related to the environment (like CPU, RAM, etc.) or related to the cache itself, were controlled and made similar as much as possible for both systems. Systems' internal configurations were kept on default configurations. Furthermore, indexing on both systems was enabled.

Table 1. Data size variations used in initializing the distributed cache.

Number of entries	Size (B)
100	9 KB
1000	89.8 KB
10,000	898 KB
100,000	8.8 MB
1,000,000	87.7 MB

3.2 Queries Specifications

In this study, the *map* data structure was used as data representation in both systems. Two main types of queries on this data structure were investigated. The first, is a basic query type, the *get* operation, a popular operation on map. The second, is an SQL-like type, which can be used to retrieve a collection of objects. In addition, we formulated four SQL-like queries, each with a different complexity level. The reason behind choosing SQL-like queries is that they are very useful for retrieving complex data from caches. Table 3 summarizes the queries with their complexities. The complexity is generally defined as follows: *"The greater the query, in terms of SQL operations (i.e., the number of logical and comparison operations), the higher the complexity value".* The metric for calculating the value of a query complexity is described below.

The complexity is computed based on the number of logical operations (e.g., AND, OR, LIKE, etc.) and comparison operations ($=, >, <$, etc.). The defined metrics assign a complexity value to each query based on the type and the number of its operations. The complexity value for each query in Table 3 below is computed by aggregating the complexity values for each operation appearing in the query. Table 2 shows the complexity value for each operation. The higher the complexity value, the greater its computation needs in terms of CPU and memory.

Table 2. Complexity definitions for each SQL operation.

SQL operation	Complexity value
$>, <, =$	1
AND	1
OR, LIKE	2

In order to be close to a real world scenario, two entities (Employee and Organization) were used as objects to hold data. *Employee* entity has four attributes: *id*, *name* (indexed), *age* (indexed), *password*, and *organization*, whereas *Organization* has five attributes: *id*, *name* (indexed), *acronym*, and *numberOfEmployees*. The relationship between the two entities is a one-to-many association. The single-attribute index was used for both systems. Moreover, in Hazelcast, non-ordered index was used in order to make the index settings as much neutralized as possible to match Infinispan's index settings.

Table 3. Retrieval queries used in benchmarking with their complexities.

Label	Query	Complexity
get	get(i) where i is a random number	N/A
SQL-like0	SELECT employee FROM Employee WHERE age >50	1
SQL-like1	SELECT employee FROM Employee WHERE age >25 AND age <75	3
SQL-like2	SELECT employee FROM Employee WHERE age <25 OR age >75	4
SQL-like3	SELECT employee FROM Employee WHERE age >50 AND name like 'A%' AND organization.name LIKE '%tum%'	7

3.3 Topology and Mechanism

As shown in Fig. 1, the study setup included five machines and one switch to conduct experiments within a local isolated network, to eliminate external networking issues. Out of the five, four machines, named host 1, 2, 3, and 4 were used to run the client benchmarks. The nodes of cache cluster were set up on the fifth machine, named workstation. The specifications for the machines are detailed below.

In this study, HP Z230 Tower Workstation was used as the server machine. Table 4 lists the specification of the server machine.

For clients, four machines were used, three of them with equal specifications. The fourth machine has a different CPU (Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz) tough. Table 5 lists the specifications of the client machines. One host (host 1) was used to manage the running of benchmark clients, and the usage of the other hosts is described in the algorithms below.

Table 6 lists the network switch specifications used during the study to enable the networking between the four hosts and the workstation.

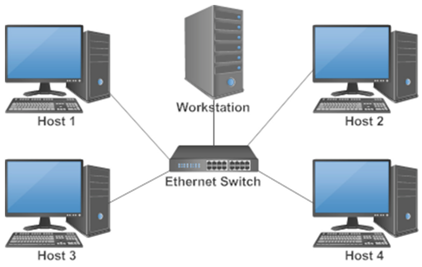

Fig. 1. Topology of experimental setup.

Table 4. Server machine specifications

Environment variable	Specs
Operating system	Ubuntu 14.04 (64 bit)
Platform	Java 1.8 (64 bit)
CPU	Intel(R) Xeon(R) CPU E3-1241 v3 @ 3.50 GHz
RAM	16 GB

Table 5. Client machines specifications

Environment variable	Specs
Operating system	Ubuntu 14.04 (64 bit)
Platform	Java 1.8 (64 bit)
CPU	Intel(R) Core(TM) i7-4770 CPU @ 3.40 GHz
	Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz
RAM	8 GB

Table 6. Network specifications

Environment variable	Specs
Switch	Cisco Catalyst 3560-X Series WS-C3560X-24P-S 24 PoE+ 715W
Switching Fabric	160 Gbps
DRAM	256 MB (51 2 MB for 3750X-12S and 3750X-24S)
Flash	64 MB (128 MB for 3750X-12S and 3750X-24S)
Total VLANs	1005
VLAN IDs	4 K

In order to set up and run the experiments, a cluster of distributed cache nodes were set up on the workstation, and a script that generates clients for requesting cache data on the clients machines (namely host 1, 2, 3, and 4). The purpose of these four hosts is to run the client drivers. The steps to run the cache cluster on the workstation are described in Algorithm 1.

Algorithm 1. RunningDistrbutedCachesOnServer

```
1: Let $cacheSystems$ =[Hazelcast, Infinispan]
2: Let $dataSizes$ =[100, 1000, 10000, 100000, 1000000]
3: for each $cacheSystem \in cacheSystems$ do
4:     for each $dataSize \in dataSizes$ do
5:         Run four nodes
6:         Create a distributed map
7:         Initiate the map with the data
8:         Monitor the CPU/memory usage
9:         Invoke Algorithm 2                    ▷ client benchmarks run here
10:        Wait until client benchmarks finish
11:    end for
12: end for
```

After the cache cluster was started and run in a stable mode, the four hosts ran the client benchmarks. Each benchmark started generating requests for a period of 180 s (the first 30 s are for warm-up). The benchmark recorded the throughput (operations/sec) for the 150 s period. Each benchmark resulted in a CSV file containing the throughput over 150 s for a specific number of concurrent clients and a specific data size.

The generated results were then taken and further analyzed to produce an overall throughput for each number of concurrent clients. To record throughput with a varying number of clients and data sizes, managing the runs of benchmarks is achieved through Algorithm 2, as shown below.

Algorithm 2. RunningBenchmarksOnClients

```
1: Let $clientsNumbers$ =[1, 2, 4, 8, 16, 32, 64, 128]
2: Let $queries$ =[get, SQL-like0, SQL-like1, SQL-like2, SQL-like3]
3: for each $clientsNumber \in clientsNumbers$ do
4:     for each $query \in queries$ do
5:         if $clientsNumber = 1 or 2$ then
6:             Monitor the CPU/memory usage on host 1
7:             Run benchmarks of $clientsNumber$ concurrently on host 1
8:         else
9:             Let $n = clientsNumber/4$ ▷ Number of client benchmarks on each host
10:            Monitor the CPU/memory usage on host 1
11:            Run benchmarks of $n$ concurrently on host 1 asynchronously
12:            Monitor the CPU/memory usage on host 2
13:            Run benchmarks of $n$ concurrently on host 2 asynchronously
14:            Monitor the CPU/memory usage on host 3
15:            Run benchmarks of $n$ concurrently on host 3 asynchronously
16:            Monitor the CPU/memory usage on host 4
17:            Run benchmarks of $n$ concurrently on host 4 asynchronously
18:        end if
19:        Wait until all benchmarks finish
20:    end for
21:    Aggregate data and covert results into throughput per number of clients
22: end for
```

All required benchmarks were implemented in Java using Yardstick framework. Algorithms 1 and 2, described above, were implemented using both Java and Shell programming languages. The project including the benchmarks as well as shell scripts used in this study can be found on a public Github repository. Here is the link[5].

This design was developed and enhanced over many iterations of dry-runs and trials. During the wet-run of the study, CPU/memory usage was also monitored to ensure high fidelity of the study and make sure that systems' performance is not affected by machine limitations.

4 Study Results

The obtained study results are shown formatted below so that the relevant performance results of both systems are brought together to compare between the two systems. Each chart below contains more than one curve, each representing the behavior of performance for a specific system and a specific data size, where the Y-axis is the throughput (ops/sec), and the X-axis represents the number of concurrent clients. Since readings were taken for only a subset of concurrent clients (i.e., 1, 2, 4, 8, 16, 32, 64, 128), a linear approximation between points was used.

4.1 Performance of *get* Query

As shown in Fig. 2, the throughput increases for both systems starting with 1 client increasing to 64 concurrent clients, for which Infinispan obviously does better in this range. However, the throughput of Hazelcast drops down when moving from 64 client to 128 clients, while Infinispan throughput keeps increasing. It is worth noting that Infinispan did not reach a maximum throughput in this case.

When increasing the data size from 100 to 1000000, the behaviour remains the same over all concurrent client variations. Throughput, on the other hand, drops down by around 19% on average for Hazelcast, whereas Infinispan drops down by around 0.2%, as shown in Fig. 2. The average, minimum, and maximum throughput for each data size for Hazelcast and Infinispan are shown in Tables 7 and 8, respectively. Each color in the leftmost column indicates a curve in Fig. 2.

4.2 Performance of *SQL-like* Queries

For *SQL-like* queries, which are more complex than the primitive *get* query, the throughput of both systems is significantly small compared to what the case is in the *get* query, as shown next. Moreover, it is clear that there is a significant drop in throughput for both systems for a shift from 100 to 100000 data size. The results also show that the effect of the number of concurrent clients becomes less significant on larger data sizes.

[5] https://github.com/ferasodh/Distributed-Caches-Benchmarking-Experiment.

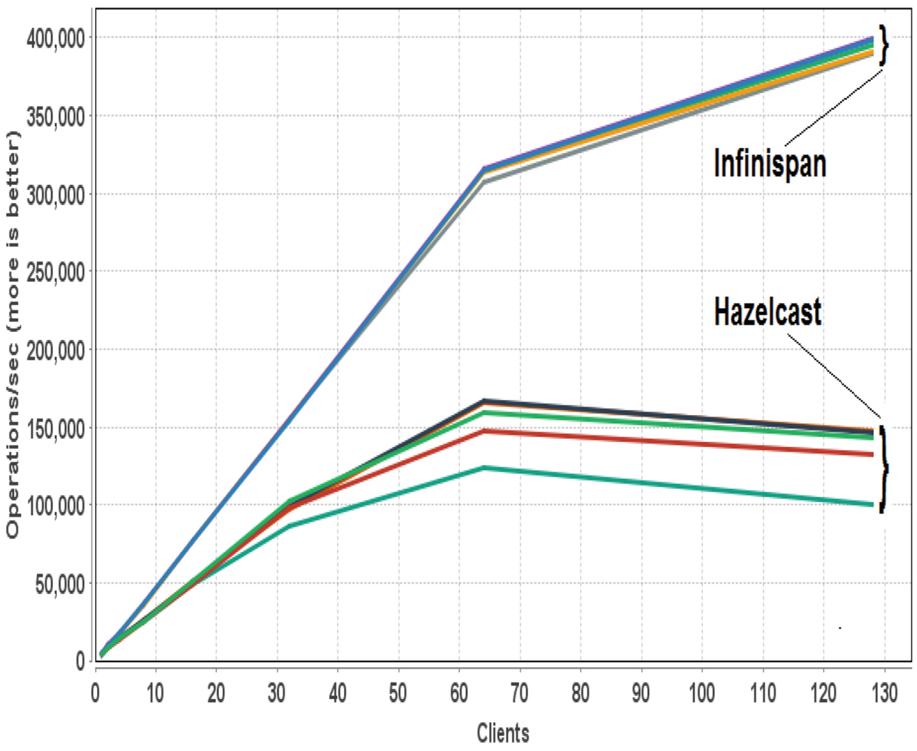

Fig. 2. Behavior of *get* performance in terms of throughput (ops/sec) as a function of number of clients. Each color in the leftmost column of Tables 7 and 8 indicates a curve in this figure. (Color figure online)

Table 7. Per data size average, minimum, and maximum throughput (in thousands ops/sec) for *Hazelcast*.

-	Data Size	Avg	Min	Max
🟩	100	63.33	3.33	159.76
🟫	1000	59.48	3.37	147.36
⬛	10000	64.21	3.33	166.52
🟧	100,000	64.13	3.37	165.68
🟦	1,000,000	51.33	3.39	124.33

Performance of *SQL-like0* Query: For *SQL-like0* query, the number of rows returned is directly proportional to data size. Hazelcast outperforms Infinispan for 100 data size by 39.8%, as shown in Fig. 3. The throughput increases for both systems for the range 1 to 64 clients. However, Infinispan outperforms Hazelcast for all bigger data sizes. Moreover, it clearly shows Infinispan has more average

Table 8. Per data size average, minimum, and maximum throughput (in thousands ops/sec) for *Infinispan.*

-	Data Size	Avg	Min	Max
🟦	100	126.51	4.80	397.97
🟪	1000	127.04	4.82	399.33
🟨	10000	125.39	4.79	390.36
▨	100,000	124.62	4.74	389.51
🟩	1,000,000	126.25	4.73	395.70

throughput than Hazelcast by 34.6%, 64.7%, and 43% in 1000, 10000, and 100000 data sizes, respectively.

Performance of *SQL-like1* Query: For *SQL-like1* query, the throughput increases for both systems as number of clients increases for 100 data size, while for bigger data sizes the maximum throughput is reached at 16 clients. For this, Hazelcast outperforms Infinispan for 100 data size by 57.9%, as shown in Fig. 4. However, Infinispan outperforms Hazelcast for all bigger data sizes. Moreover, it shows that Infinispan has a better average throughput than Hazelcast by 51%, 66.3%, and 66.9% at 1000, 10000, and 100000 data sizes, respectively.

However, with large data size at 10000, there is a drop in Infinispan performance for the range of 16 to 64 clients, while Hazelcast achieves minimum throughput at 100000 data size and 128 clients.

Performance of *SQL-like2* Query: None of the systems reach the maximum throughput in *SQL-like2* query for 100 data size, while for bigger data sizes the maximum throughput is reached with 32, 16, and 8 clients. In this case, Hazelcast outperforms Infinispan for 100 data size by 55.9%, as shown in Fig. 5. However, Infinispan outperforms Hazelcast for all bigger data sizes.

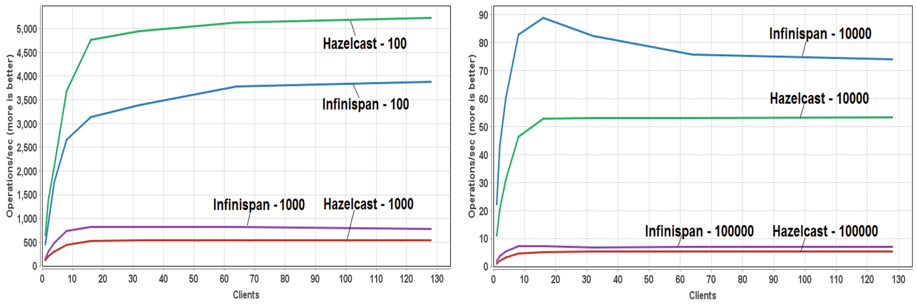

Fig. 3. Behavior of *SQL-like0* performance in term of throughput (ops/sec) as a function of number of clients (1, 2, 4, 8, 32, 64, 128) for all data sizes.

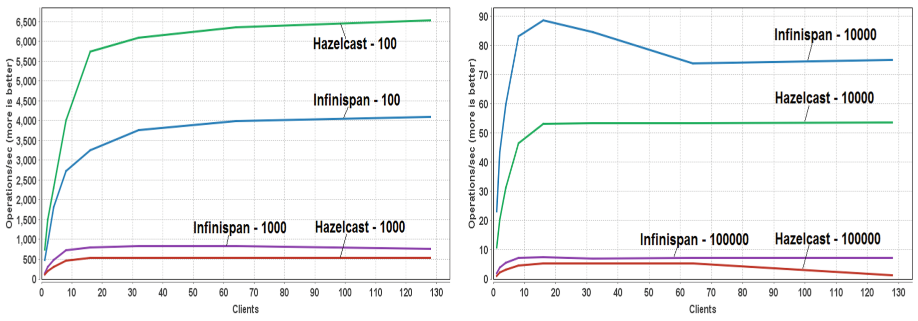

Fig. 4. Behavior of *SQL-like1* performance in term of throughput (ops/sec) as a function of number of clients (1, 2, 4, 8, 32, 64, 128) for all data sizes.

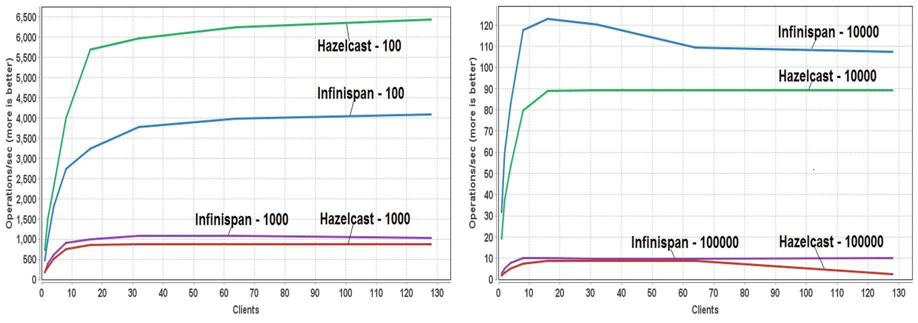

Fig. 5. Behavior of *SQL-like2* performance in term of throughput (ops/sec) as a function of number of clients (1, 2, 4, 8, 32, 64, 128) for all data sizes.

Moreover, the results also show Infinispan has a better average throughput than Hazelcast by 20.5%, 37.6%, and 40.9% at 1000, 10000, and 100000 data sizes, respectively. For 10000 data size, there is a drop in Infinispan performance for 16 to 64 clients, and Hazelcast achieves minimum throughput at 100000 data size and 128 clients.

Performance of SQL-like3 Query: For *SQL-like3* query, none of the systems reaches the maximum throughput for 100 data size. For 1000 data size Hazelcast does not reach a maximum throughput while Infinispan reaches a maximum throughput at 32 clients. For bigger data sizes, the maximum throughput is reached with 16 and 8 clients. Hazelcast outperforms Infinispan for 100 and 1000 data size by 261.65% and 285.7%, respectively, as shown in Fig. 6. However, Infinispan outperforms Hazelcast for 10000 and 100000 data sizes.

It also shows that Infinispan has a better average throughput than Hazelcast by 37.62% and 40.86% in 10000 and 100000 data sizes, respectively. However for 10000 data size, there is a drop in Infinispan performance for 16 to 64 clients. Hazelcast achieves minimum throughput on 100000 data size and 128 clients.

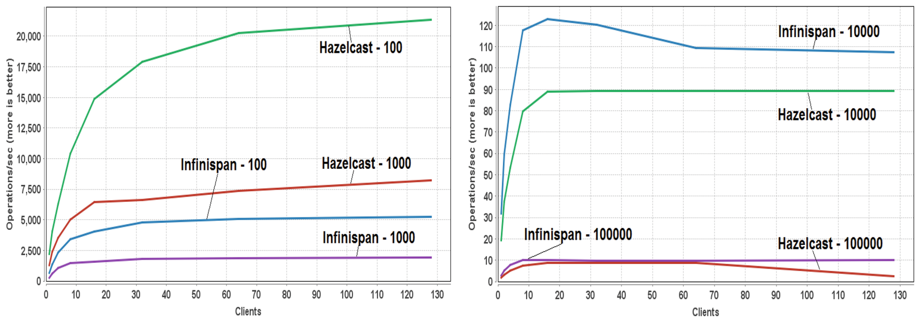

Fig. 6. Behavior of *SQL-like3* performance in term of throughput (ops/sec) as a function of number of clients (1, 2, 4, 8, 32, 64, 128) for all data sizes.

Table 9. Per data size average, max, and min throughput (ops/sec) for each query for *Hazelcast*.

Data size	Avg	Min	Max	Avg	Min	Max
	SQL-like0			SQL-like1		
100	3,485.25	650.43	5,224.75	4,157.87	741.37	6,531.27
1,000	401.24	114.05	538.07	406.09	112.15	541.37
10,000	40.18	11.08	53.30	40.36	10.89	53.77
100,000	4.06	1.12	5.47	3.60	1.13	5.46
	SQL-like2			SQL-like3		
100	4,101.74	731.26	6,428.29	12,154.58	2,177.01	2,177.01
1,000	651.77	175.98	874.01	5,099.81	1,260.55	8,203.85
10,000	68.24	19.40	89.20	68.24	19.40	89.20
100,000	5.80	1.81	8.81	5.80	1.81	8.81

Tables 9 and 10 summarize results for the SQL-like queries showing the average, minimum, and maximum throughput for each query and for each data size for Hazelcast and Infinispan, respectively.

5 Discussion

As shown above, Infinispan outperforms Hazelcast in all benchmarks except in SQL-like queries at small data sizes. There are several factors that affect the performance of Infinispan and Hazelcast and could be reasons for performance bottlenecks; some of these factors are discussed below:

- **Data serialization:** In order to transfer cache objects across a network between clients and a cache cluster or between cache cluster peers, objects need to be serialized into bytes. When read by the application, those bytes

Table 10. Per data size average, max, and min throughput(ops/sec) for each query for *Infinispan.*

Data size	Avg	Min	Max	Avg	Min	Max
	SQL-like0			SQL-like1		
100	2,493.22	449.31	3,875.48	2,633.41	480.84	4,103.19
1,000	612.90	140.89	827.48	613.40	149.24	829.77
10,000	66.20	22.35	88.85	66.44	23.09	88.61
100,000	5.81	2.07	7.27	6.01	2.11	7.59
	SQL-like2			SQL-like3		
100	2,631.53	479.09	4,077.35	3,360.91	667.01	5,242.86
1,000	785.25	182.37	1,085.06	1,322.45	282.71	1,899.73
10,000	93.91	31.86	122.82	93.91	31.86	122.82
100,000	8.17	2.92	10.10	8.17	2.92	10.10

need to be converted back to objects or deserialized. Whenever a request comes to a cache system, about 20% of the processing time is spent in serialization and deserialization in most configurations [5]. Obviously, data serialization is one of the key factors that affects cache performance. However, the default Java implementation, which is the used implementation for Hazelcast, is slow in terms of CPU cycles and produces unnecessarily large bytes [3,5]. On the other hand, Infinispan uses Jboss marshalling framework[6] as its default serialization scheme.

Jboss marshalling framework do not write full class definitions to the stream, instead each known type is represented by a single byte by using magic numbers [5]. Moreover, Infinispan forces developers of applications to register an "externalizer" for their application types to make use of Jboss marshalling [5,6]. Based on this, serialization has a significant impact on both systems' throughput and explains why Infinispan has a better performance in most cases.

– **In-memory objects format:** When objects are stored in Hazelcast or Infinispan they are serialized to byte arrays and deserialized when they are read. In Hazelcast, the default format is the binary format. However, this format is not efficient if the application is doing a lot of SQL-like queries where serialization/deserialization happens on the server side. Moreover, Hazelcast provides other formats like object and native format. One drawback of Object format is that it adds an extra serialization/deserialization step for *get* and *put* operations, while native format is only available for Hazelcast Enterprise HD version [17]. This study used the default Hazelcast binary format which explains the low performance of Hazelcast on SQL-like queries, especially with large data sizes.

Even though the cost of serialization/deserialization maybe small for smaller

[6] http://jbossmarshalling.jboss.org/.

data sizes, it will become large for larger data sizes especially in the SQL-like queries where the returned result set is large.
- **Indexing:** One of the most significant factors in query performance is indexing. Although indexes were added to both systems, Hazelcast and Infinispan, it may be possible that the engine for Infinispan, which is based on hibernate search and Apache Lucene, is more optimized than Hazelcast default indexing mechanism.

6 Conclusion and Future Work

Changing the number of concurrent clients and varying processed data sizes affect the performance of data retrieval operations of distributed cache systems. In this study, all known variables that can affect the performance of the two systems except the factors of interest (i.e., number of concurrent clients and data size) were as much as possible considered and neutralized.

Results show that studying performance analysis of systems with dynamically varying number of concurrent clients and data sizes is critical in determining a more accurate performance readings. Measuring performance with static independent variable or factors may provide misleading results, particularly in systems where cache is a critical part of a system function or design. These require building benchmarking tools that consider such dynamically changing variables to reflect and replicate real-life usage of systems. The other significant factors that may well affect cache systems' performance are data serialization, in-memory object formats and indexing, which their exact chosen implementation may improve a system's performance over another. However, these need further study and investigation in how best to address the implication of these varying variables or factors.

In addition, results show that Infinispan (version 8.1.2.Final) outperforms Hazelcast (version 3.6.1) in all tested cases except in SQL-like queries with small data sizes. Moreover, the study shows that the concurrent clients, where each client opens its own connection, has a considerable impact on the performance of *get* and *SQL-like* queries. The data size, on the other hand, has very small impact on the performance of *get* query but large impact on the performance of *SQL-like* queries.

Further, based on the mechanism followed in this study, a more integrated benchmarking framework, as proposed above, need to be developed that takes into account the varying number of concurrent clients and data sizes for distributed caches. There are several other interesting issues to consider, first, it would be interesting to study the effect of different data storage formats such as compressed and uncompressed. Second, understanding the effect of larger data sizes and cache access patterns would shed light on performance variations. Future work may also include developing new techniques that improve the performance with respect to data representations and communication protocols.

References

1. Gridgain vs. hazelcast benchmarks. http://go.gridgain.com/Benchmark_GridGain_vs_Hazelcast.html. Accessed 28 May 2016
2. Gridgain/apache ignite vs hazelcast benchmark. https://hazelcast.com/resources/benchmark-gridgain/. Accessed 28 May 2016
3. Hazelcast documentation. http://docs.hazelcast.org/docs/3.6/manual/html-single/index.html#distributed-query. Accessed 28 May 2016
4. Ignite vs. hazelcast benchmarks. http://www.gridgain.com/resources/benchmarks/ignite-vs-hazelcast-benchmarks/. Accessed 28 May 2016
5. Infinispan. http://www.aosabook.org/en/posa/infinispan.html#fn10. Accessed 25 June 2017
6. Infinispan documentation. http://infinispan.org/docs/8.2.x/index.html. Accessed 01 May 2016
7. Red hat infinispan vs hazelcast benchmark. https://hazelcast.com/resources/benchmark-infinispan/. Accessed 28 May 2016
8. Redis 3.0.7 vs hazelcast 3.6 benchmark. https://hazelcast.com/resources/benchmark-redis-vs-hazelcast/. Accessed 28 May 2016
9. Agrawal, S., Chaudhuri, S., Das, G.: Dbxplorer: a system for keyword-based search over relational databases. In: Proceedings of 18th International Conference on Data Engineering, 2002, pp. 5–16. IEEE (2002)
10. Chen, S., Liu, Y., Gorton, I., Liu, A.: Performance prediction of component-based applications. J. Syst. Softw. **74**(1), 35–43 (2005)
11. Chen, X., Ho, C.P., Osman, R., Harrison, P.G., Knottenbelt, W.J.: Understanding, modelling, and improving the performance of web applications in multicore virtualised environments. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, pp. 197–207. ACM (2014)
12. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)
13. Das, A., Mueller, F., Gu, X., Iyengar, A.: Performance analysis of a multi-tenant in-memory data grid. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pp. 956–959. IEEE (2016)
14. Denaro, G., Polini, A., Emmerich, W.: Early performance testing of distributed software applications. In: Proceedings of ACM SIGSOFT Software Engineering Notes, vol. 29, pp. 94–103. ACM (2004)
15. Dey, A., Fekete, A., Nambiar, R., Röhm, U.: YCSB+T: benchmarking web-scale transactional databases. In: Proceedings of 2014 IEEE 30th International Conference on Data Engineering Workshops (ICDEW), pp. 223–230. IEEE (2014)
16. Engelbert, C.: White paper: caching strategies. Technical rep., Hazelcast Company. https://hazelcast.com/resources/caching-strategies
17. Evans, B.: White paper: an architect's view of hazelcast. Technical rep., Hazelcast Company. https://hazelcast.com/resources/architects-view-hazelcast/
18. Fedorowicz, J.: Database performance evaluation in an indexed file environment. ACM Trans. Database Syst. (TODS) **12**(1), 85–110 (1987)
19. Khazaei, H., Misic, J., Misic, V.B.: Performance analysis of cloud computing centers using m/g/m/m+r queuing systems. IEEE Trans. Parallel Distrib. Syst. **23**(5), 936–943 (2012)
20. Klems, M., Anh Lê, H.: Position paper: cloud system deployment and performance evaluation tools for distributed databases. In: Proceedings of the 2013 International Workshop on Hot Topics in Cloud Services, pp. 63–70. ACM (2013)

21. Paul, S., Fei, Z.: Distributed caching with centralized control. Comput. Commun. **24**(2), 256–268 (2001)
22. Wang, Q., Cherkasova, L., Li, J., Volos, H.: Interconnect emulator for aiding performance analysis of distributed memory applications. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, pp. 75–83. ACM (2016)
23. Wouw, S.V., Viña, J., Iosup, A., Epema, D.: An empirical performance evaluation of distributed SQL query engines. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, pp. 123–131. ACM (2015)
24. Zhang, H., Tudor, B.M., Chen, G., Ooi, B.C.: Efficient in-memory data management: an analysis. Proc. VLDB Endowment **7**(10), 833–836 (2014)